Pushdown Automata and Inclusion Problems

by

NGUYEN VAN TANG

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Professor MIZUHITO OGAWA

School of Information Science
Japan Advanced Institute of Science and Technology

March, 2009

Abstract

Pushdown automata naturally model the control flow of sequential computation in typical programming languages with nested, and potentially recursive, invocations of program modules such as procedures and method calls. Consequently, a variety of program analysis, compiler optimization, and model checking questions can be formulated as decision problems for pushdown automata.

The automata-theoretic approach to verification, advocated by Vardi and Wolper in 1986, connects model checking to the inclusion problem of automata. That is, a system model A satisfies a specification B if and only if $L(A) \subseteq L(B)$, where A and B are automata corresponding to the model and specification respectively, and $L(A)$ and $L(B)$ are the languages recognized by A and B, respectively.

- When A and B are finite automata, the standard methodology for the inclusion problem is to, compute the complement $\overline{L(B)}$, take the intersection between $L(A)$ and $\overline{L(B)}$, then check for its emptiness, i.e., $L(A) \subseteq L(B) \iff L(A) \cap \overline{L(B)} = \varnothing$.
- This also works when A is extended to a pushdown automaton, but fails when B is extended to a pushdown automaton.

Model checking pushdown models against regular properties (specified by finite automata or logical characterizations such as LTL) is decidable. This follows from classical results on pushdown automata: the product of a pushdown automaton and a finite-state automaton gives a pushdown automaton, and the emptiness of the language of a pushdown automaton can be checked in polynomial-time. However, unfortunately, the problem of checking context-free properties of pushdown automata is undecidable in general. This is because pushdown automata are not closed under intersection and complementation. To obtain decidability results for inclusion, subclasses of pushdown automata (PDA) such as superdeterministic pushdown automata and visibly pushdown automata have been proposed:

1. **Superdeterministic Pushdown Automata.** Superdeterministic pushdown automata (SPDA), proposed by Greibach and Friedman in 1980, are a subclass of deterministic pushdown automata. If the acceptance condition is by final location, the language inclusion $L(A) \subseteq L(B)$ is decidable for a PDA A and an SPDA B.
2. **Visibly Pushdown Automata.** Visibly pushdown automata (VPAs), proposed by Alur and Madhusudan in 2004, are special pushdown automata whose stack

operations are driven by the input. The class of VPAs recognizes some non-regular context-free languages while having the good closure properties and decidability results as those of finite automata. The inclusion problem for VPAs is EXPTIME-complete.

The aim of this thesis is to study the inclusion problems for subclasses of pushdown automata. We make the following main achievements and contributions towards this goal:

- We refine the alternate stacking technique used in Greibach-Friedman's proof of the language inclusion problem $L(A) \subseteq L(B)$, where A is a pushdown automaton and B is a superdeterministic pushdown automaton. In particular, we propose a product construction of a simulating pushdown automaton, whereas the one given by the original proof encoded everything as a stack symbol. This construction avoids the need for the "liveness" condition in the alternate stacking technique, and the correctness proof becomes simpler.

- Alur and Madhusudan proved that visibly pushdown automata are determinizable. We give detailed counter examples to show that natural extensions of visibly pushdown automata such as *ordered 2-visibly pushdown automata* (consequently, 2-visibly pushdown automata) and *visibly stack automata* are not determinizable.

- The antichain algorithm limits determinization steps as minimum as possible and gives a fast decision procedure for the universality problem of finite automata. We extend the method to visibly pushdown automata with combining $\mathcal{P}$-automata techniques. Preliminary experiments on randomly generated visibly pushdown automata show significant improvement compared to the standard automata-theoretic approach, especially when the universality / the inclusion do not hold.

- We introduce the class of *event-clock* visibly pushdown automata (ECVPAs) as an extension of event-clock automata. The class of ECVPAs is, on one hand, enough to model simple real-time pushdown systems and, on the other hand, determinizable and closed under boolean operations. We also show that for a *timed* visibly pushdown automaton (TVPA) A and an ECVPA B, the inclusion problem $L(A) \subseteq L(B)$ is decidable. This provides an algorithm for checking if a TVPA model satisfy a context-free real-time specification given as ECVPA.

Key Words: Formal Verification, Model Checking, Pushdown Automata, Visibly Pushdown Automata, Inclusion Problem, Antichains, Timed Automata, Event-Clock Automata, Verification of Real-time Systems.

Acknowledgements

This thesis would not have been possible without the guidance, generosity, and goodwill of many people. I feel grateful and indebted to have received all their help.

First of all, I am greatly grateful to Professor Mizuhito Ogawa, my supervisor, for his guidance, wisdom and support he has provided me throughout my doctoral education. He always helps me to clarify my research issues, inspires new ideas and enhances my thinking ability. Without him, this thesis would be impossible. The work presented in this thesis in effect should be regarded as a result of collaborations with him.

I would like to express my deep gratitude to thank to Professor Takuya Katayama for giving me a chance to have an interview for this Ph.D. position when I met him for the first time in Hanoi.

I also would like to sincerely thank the dissertation committee members, who gave me instructive suggestions and comments in the evaluation of the preliminary version of the thesis. They are Professor Hiroyuki Seki from NAIST, Professor Kunihiko Hiraishi, Associate Professor Toshiaki Aoki, and Associate Professor Kazuhiro Ogata from JAIST.

I wish to continue my sincere thanks to my sub-theme supervisor Professor Kunihiko Hiraishi and Dr. Koichi Kobayashi for conducting me to a very interesting field and widening my view. The topic of hybrid systems is very likely to be another target of my future research.

I am very indebted to Professor Dang Van Hung, my master thesis's supervisor, who has always been available for me at anytime since I met him for the first time in Hanoi. I have not met Professor Dang Van Hung without the recommendation of Professor Doan Van Ban, my supervisor at Hanoi Institute of Information Technology. My special thanks also go to Professor Ho Tu Bao for his valuable discussions and advices in the daily life. I would like to express my appreciations to all of them.

During the last three years, especially the last two years of my Ph.D., I had a lot of useful discussions with Dr. Nao Hirokawa. He also gave me many suggestions and comments to improve my work. Besides, he helped me a lot in computer's problems such as using pdflatex & TikZ/PGF, doing experiments, and my technical presentation. Nao, thank you very much for your help.

Let me say special thanks to the members of Software Verification Labs, Dr. Li Guoqiang, Dr. Li Xin, Mrs. Do Ngoc, Mr. Song Lin, and Mr. Dominik Klein for their valuable discussions and comments.

Last but definitely not least, during my study, I always get the support and encouragement from my family and friends. Thank you all.

Funding. This research is supported by the 21st Century COE "Verifiable and Evolvable e-Society" of Japan Advanced Institute of Science and Technology, funded by Japanese Ministry of Education, Culture, Sports, Science and Technology.

This thesis has been prepared according to the new standard version of the LaTeX with the supplemental package TikZ/PGF version 1.10, and compiled with LaTeX 2_ε. This software makes the typesetting for scientific report much easier. Our prototype tool has been implemented in Java 1.5.0/NetBeans IDE 6.0.1.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The class of context-free languages (CFL) plays an important role in several areas of computer science. Besides its definition using context-free grammars it has various other characterizations, the most prominent being the one via nondeterministic pushdown automata. Pushdown automata (PDAs) are finite automata augmented with a pushdown stack. A PDA 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, PDAs naturally model the control flow of sequential computation in typical programming languages with nested, and potentially recursive, invocations of program modules such as procedures and method calls. Consequently, a variety of program analysis, compiler optimization, and model checking questions can be formulated as decision problems for PDAs.

It is well known that the class of PDAs does not enjoy good closure properties, e.g., it is not closed under complement or intersection, and that several interesting problems are undecidable, e.g., checking whether a context free language is regular, or whether it contains all words (universality), or whether a language is contained in another one (language inclusion) [9, 72]. We must therefore contend ourselves with formalisms of lesser expressive power but having good decidability results. The aim of this thesis is to study the inclusion problems for subclasses of PDAs. We first briefly present motivations of our research from the model checking point of view.

1.1 Model Checking

It has long been known that computer software programs, computer hardware designs, and computer systems in general exhibit errors. Working programmers may devote more than half of their time on testing and debugging in order to increase reliability. A great deal of research effort has been and is devoted to developing improved testing methods.

Testing

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 scenarios is impractical. The designer of the tests, whether a person or a specially designed program, chooses a selection of important cases. 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. Testing, therefore, cannot guarantee that a program is correct.

The most promising approach depends on the fact that programs and more generally computer systems may be viewed as mathematical objects with the behaviours that is in principle well-determined. This makes it possible to specify using mathematical logic what constitutes the intended (correct) behaviours. Then one can try to give a formal proof or otherwise establish that the program satisfies its specification. This line of study has been attracted extensively attention over three past decades.

Model Checking

Model checking, proposed in 1980 by E.M. Clarke and E.A. Emerson [20] and J.P. Quielle and J. Sifakis [59] independently, is an automatic verification technique for finite-state concurrent systems. The *model checking problem* is easy to state:

> Given a program (system model) A and a correctness specification B, determine whether or not the behaviour of A satisfies the specification B?.

Model checking provides an automated technique for verifying concurrent finite state systems that uses an efficient and flexible graph search, to determine whether or not the ongoing behaviour described by a temporal property holds in the system's state graph. The method is algorithmic and efficient because the system is finite state, despite reasoning about infinite behaviour. If the answer is *yes* then the system satisfies its specification. If the answer is *no* then the system violates its specification; in practice, the model checker can usually produce a counterexample for debugging purposes (Figure 1.1).

Figure 1.1: Model Checker Structure

Model checking has a number of advantages compared to other verification techniques such as automated theorem proving or proof checking. The user of a Model Checker does

not need to construct a correctness proof. In principle, all that is necessary is for the user to enter a description of the circuit or program to be verified and the specification to be checked and press the "return" key. The checking process is automatic. In practice, model checking is fast compared to other rigorous methods such as the use of a proof checker, which may require months of the user's time working in interactive mode.

Models and specifications are two important components of model checking. Usually, system models $\mathcal{M}$ are described as Kripke structure (i.e, finite state-transition graph or finite automata) and specifications $\mathcal{S}$ are given in temporal logic [56] such as CTL (computation tree logic) and LTL (linear time logic) formulas. Model checking for finite state systems has been successfully implemented in automatic tools such as SMV/NuSMV [1] and SPIN [2] leading to efficient applications to hardware verification.

1.2 Inclusion Problems and Model Checking

The automata-theoretic approach to verification, advocated by Vardi and Wolper [75] in 1986, connects model checking to the inclusion problem of automata. That is, a system model A satisfies a specification B if and only if (iff) $L(A) \subseteq L(B)$ which is equivalent to $L(A) \cap \overline{L(B)} = \varnothing$, where A and B are automaton corresponding to the model and specification respectively, and $L(A)$ and $L(B)$ are the sets of behaviours recognized by A and B, respectively. Thus, the decidability of a model checking problem is ascribed to the inclusion problem of automata. The automata-theoretic approach also enables on-the-fly model checking [33]. Automata-based methods have been implemented in industrial automated verification tools such as SPIN [38]. Finite model checking is successful in hardware verification that has a finite state space in nature.

Pushdown Model Checking

However, finite state verification is not always adequate for analyzing software. Automatic software validation is not easy, because the state space of software is inherently infinite. The infinities come from infinite program structures (e.g., nested procedure calls, recursions), infinite data domains (e.g., integers), concurrency, etc. Automatic software validation demands efforts from program analysis and abstraction to model checking techniques on infinite state space. One of the simplest kinds of infinite state system that we can verify are *pushdown automata* [72, 9]. 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

[1] `http://www.cs.cmu.edu/modelchecker/code.html`
[2] `http://www.spinroot.com/`

procedure calls. Consequently, a variety of program analysis, compiler optimization, and model checking questions can be formulated as decision problems for pushdown automata.

Pushdown Model Checking for Regular Properties

The automata-theoretic approach works for pushdown model checking on *regular properties* (e.g., written in CTL, LTL), since the intersection of a context-free language and a regular language is closed, i.e. still context-free. The practical algorithms for pushdown model checking have been developed based on the key that regular pushdown stack valuations are closed under forward and backward reachability [30, 28, 29]. Recently, some practical model checking algorithms on the pushdown models, the finite-state system with an unbounded stack memory, have been developed [29, 58] and implemented as tools, such as Moped [3] and Slam [4].

Pushdown Model Checking for Context-Free Properties

Almost all existing work on model checking considers regular specification formalism that defines regular set of words. In practice, however, many desired behaviours are *non-regular* and cannot be specified by a finite automaton. For example, the properties that require inspection of the stack or matching of calls and returns are context-free. More examples of useful non-regular properties are given in [62], where the specification of unbounded message buffers is considered. Although the general problem of checking *context-free properties* of pushdown automata is undecidable, algorithmic solutions have been proposed for checking some kinds of non-regular properties. In particular, Alur et al. recently introduced the logic CaRet [6]. CaRet is a linear temporal logic that can specify some non-regular properties. Pushdown model checking for CaRet logic is decidable in exponential time [6]. We note that CaRet is less expressive than pushdown automata. The study of CaRet inspired Alur and Madhusudan [5] to introduce *visibly pushdown automata* (VPA). These are pushdown automata where the push or pop actions on the stack are determined externally be the input alphabet. Such a restriction on the use of the stack allows VPA to enjoy good closure properties, and the inclusion problem for VPA is decidable. Therefore, VPAs are relevant to several applications that use context-free languages such as the model-checking of software programs using their pushdown models [5, 6, 8]. Recent work has shown promising applications in other contexts: in modeling semantics of effects in processing XML streams [50, 46], in identifying larger classes of pushdown specifications that admit decidable problems for infinite games on pushdown graphs [52], and in VPA-based aspects [55].

[3] `http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/`

[4] `http://research.microsoft.com/slam/`

1.3 Contributions

Motivated by pushdown model checking for non-regular context-free properties, the aim of this thesis is to study the inclusion problems for subclasses of pushdown automata. We make the following contributions towards this overall goal:

We first present the basic notions and properties of pushdown automata, and $\mathcal{P}$-automata technique for checking reachability (equivalent to emptiness) of pushdown automata in Chapter 2.

Superdeterministic Pushdown Automata

Our first contribution, presented in Chapter 3, is an improvement on the alternate stacking technique used in Greibach-Friedman's proof of the language inclusion problem $L(A) \subseteq L(B)$, where A is a *pushdown automaton* (PDA) and B is a *superdeterministic pushdown automaton* (SPDA). An SPDA is a deterministic PDA (DPDA) satisfying:

1. *finite delay* (i.e., a bounded number of ϵ-transitions in a row can be applied to any configuration), and

2. for two configurations sharing the same control state, transitions with the same symbol lead to configurations sharing the same control state such that the length change of stacks is the same.

Greibach and Friedman [34] used the alternate stacking technique [74] to show that the inclusion problem $L(A) \subseteq L(B)$, where A is a PDA and B is an SPDA, is decidable. The key idea of their proof is to construct a simulating pushdown automaton M such that $L(A) \subseteq L(B)$ iff $L(M) = \emptyset$. However, the original construction encodes everything as stack symbols (in an intricate way), and thus control states and transition rules of M could not be given in details. Furthermore, to decide the emptiness of M, one has to use an auxiliary procedure to check whether a configuration of the PDA A is *live* (i.e., whether a configuration reaches an accepting configuration) or not. These properties of their simulating PDA M lead to a complicated proof of soundness and completeness for the decision procedure [34].

To deal with these drawbacks, we propose a product construction of a simulating PDA M, whereas the one given by the original proof encoded everything as a stack symbol. This construction avoids the need for the "liveness" condition in the alternate stacking technique, and the correctness proof becomes simpler.

Language Extensions of VPA Are Difficult

In Chapter 4, we present the notion and properties of visibly pushdown automata. We also give detailed counter-examples to show that natural extensions of visibly pushdown

automata such as *ordered 2-visibly pushdown automata* (consequently, 2-visibly pushdown automata) and *visibly stack automata* are not determinizable.

Visibly pushdown automata [5] are pushdown automata whose stack behaviour (i.e. whether to execute a push, a pop, or no stack operation) is completely determined by the input symbol according to a fixed partition of the input alphabet. As shown in [5, 7], this class of visibly pushdown automata enjoys many good properties similar to those of the class of regular languages, the main reason for this being that each nondeterministic VPA can be transformed into an equivalent deterministic one. As each nondeterministic VPA can be determinized, all problems that concern the accepted language such as universality and inclusion problems are decidable for VPAs. Visibly pushdown automata have turned out to be useful in various context, e.g., as specification formalism for verification for pushdown models [5], and as automaton model for processing XML streams [7, 50, 46].

Visibly pushdown automata with multiple stacks have been considered recently and independently by Carotenuto et al. [17] and Torre et al. [69]. The purpose of these papers is to exploit the notion of visibility further to obtain even richer classes of languages while preserving important closure properties and decidability of verification-related problems such as emptiness and inclusion. The emptiness problems for these extensions, however, are undecidable.

To retain the good closure properties, Torre et al. [69] consider a subclass, named *k-MVPAs*, of multiple-stack VPAs with restrictions that: *an input word can be divided into at most k phases such that, in every phase, pop actions can occur in at most one stack*. Then, the emptiness problem becomes decidable. Although k-MVPAs are not determinizable, they are closed under Boolean operations [69].

In [17], the approach to gain decidability and good closure properties is to exclude simultaneous pop operations by introducing an ordering constraint on stacks. This restricted subclass of multi-stack VPAs is called *ordered* n-VPAs (n-OVPAs). For instance, in 2-OVPAs, a pop action on the second stack occurs only after the first stack becomes empty. Then, the emptiness problem turns out to be decidable in polynomial time. They also claimed the determinizability of 2-OVPAs (2-VPAs).

In Chapter 4, we first give detailed counter examples to refute the claim of Carotenuto et al. about the determinizability of 2-OVPAs (consequently, 2-VPAs). Second, we introduce the class of *visibly stack automata* (VSAs) as an extension of VPAs by combining ideas of visibility and stack automata [35]. We also give a counter example to show that VSAs are not determinizable.

Checking Universality and Inclusion of Visibly Pushdown Automata

Our third contribution, presented in Chapter 5, consists of antichain-based algorithms for checking universality and inclusion of visibly pushdown automata.

One of the most important properties is that nondeterministic VPAs can be determinized, and determinization plays a key role in universality and inclusion checking [5]. However, the determinization is much harder to obtain than in the case of finite automata. In particular, for a nondeterministic VPA with n states, the determinization has a $O(2^{n^2})$ worst case complexity. To check universality for a nondeterministic VPA A over Σ (that is, to check if $L(A) = \Sigma^*$), the classical approach is first to make it complete, determinize it, and then checks for reachability of nonaccepting states of the determinized VPA. To check the inclusion problem $L(A) \subseteq L(B)$, the standard approach computes the complement of B, takes its intersection with A, and then check for emptiness. This is costly as computing the complement necessitates determinization. This explosion is in some sense unavoidable, as the universality and language inclusion problem for VPAs are known to be EXPTIME-complete [5].

During the recent years, a lot of research has been done to implement efficiently operations like complementation [48, 49] and universality or inclusion checking on nondeterministic word, Büchi, or tree automata [77, 24, 14]. The solutions in [77, 24, 14] is so-called *antichain technique*, an antichain is a finite set of incomparable elements. Its idea comes from an analysis of the complementation, which consists of two steps (1) the determinization and (2) the alternation of final states. In a determinization, the subset construction generates determinized states, each of which is the collection of destination states of transitions of a word. A (forward) antichain further reduces it to minimal determinized states only. The idea is that, in either the universality or the inclusion checking, the final step is the emptiness checking, i.e., whether there exists a word reachable to a rejecting determinized state, regardless of which word is an instance. Although the antichain algorithm does not improve the complexity in theory, it is significant in practice. For instance, there had been virtually no implementations of complementing a Buchi automaton (which is $O(2^{n \log n})$), but one was given by antichains [24] and implemented as ALASKA [78].

In this Chapter 5, we first apply the standard method to check universality and inclusion problems for nondeterministic VPA. The method includes two main steps: determinization and reachability checking for non-accepting configurations. For determinization, we use the Alur-Madhusudan's procedure. For reachability checking, we apply the symbolic technique $\mathcal{P}$-automata [29, 58] to compute the sets of all reachable configurations of a VPA. We implement this standard approach in a prototype tool written in Java 1.5.0/NetBeans 6.0. We test on randomly generated VPA. However, the performance of this method is very low. The program stuck with very mall size of input VPAs.

To improve the standard method, we propose and experimentally evaluate new efficient methods for checking universality and inclusion problems of VPAs: *on-the-fly* method and *antichain-based* method.

- **On-the-fly method.** To improve efficiency, we perform determinization and reachability checking on-the-fly manner. More precisely, we construct determinized VPA and $\mathcal{P}$-automaton simultaneously. For checking universality of nondeterministic VPA M, we first create the initial state of the determinized VPA M^d and a $\mathcal{P}$-automaton A to represent the initial configuration. Second, construct new transitions departing from the initial states, and update the $\mathcal{P}$-automaton A. Then, using new states and transitions of A (which correspond to pairs of the states and topmost stack symbols of M^d), update the determinized VPA M^d, and so on. When a nonaccepting state is added to A, we can stop and report that M is not universal.

- **Antichain-based method.** We extend the antichain-based algorithms [77] to visibly pushdown automata. In particular, as determinization is expensive, we first construct an algorithm for checking universality by keeping determinization implicitly. The main idea is to try to find at least one word not accepted by the VPA. For this sake, we follow the simultaneous technique as in the on-the-fly method. Besides, an ordering over transitions of determinized VPA is introduced to perform a kind of *minimal* symbolic simulation of the $\mathcal{P}$-automaton to cover all runs necessarily leading to non-accepting states. We also give a new algorithmic solution to inclusion checking for VPAs. Again, no explicit determinization is performed. To solve the language-inclusion problem for nondeterministic VPAs, $L(A) \subseteq L(B)$, the main idea is to find at leat one word w accepted by A but not accepted by B, i.e., $w \in L(A) \setminus L(B)$.

To evaluate the proposed algorithms, we have implemented them all in a prototype tool and tested them in a series of experiments. Although the standard approaches (as well as ours) have the same worst case complexity, our prototype implementation outperforms those approaches where determinization is explicit. Preliminary experiments on randomly generated visibly pushdown automata show a significant improvement of on-the-fly and antichain-based methods compared to the standard method, especially when the universality / the inclusion do not hold. For the cases of universal VPAs, our experimental results show that the antichain-based method is considerably faster than the standard method.

Timed Extensions of Visibly Pushdown Automata

Our fourth contribution, presented in Chapter 6, is the class of *event-clock* visibly pushdown automata (ECVPAs) as an extension of event-clock automata. The class of ECVPAs is, on one hand, enough to model simple real-time pushdown systems and, on the other hand, determinizable and closed under boolean operations. We also show that for a *timed* visibly pushdown automaton A and an ECVPA B, the inclusion problem $L(A) \subseteq L(B)$ is decidable.

Timed automata (TAs) were introduced by Alur and Dill in [2], and have become a standard modeling formalism for real-time systems. A timed automaton is a finite automaton augmented with a finite set of real-valued clocks, in which constraints on the clocks are used to restrict the behaviors of an automaton. The theory of timed automata allows the solution of certain verification problems for real-time systems [2, 37, 15], e.g., reachability and safety properties. These solutions have been implemented in automatic tools such as UPPAAL [5].

However, the general verification problems (i.e., language inclusion) for timed automata is undecidable. Therefore, for the verification purpose, one has to work either with deterministic specifications or with a restricted class of timed automata which has the required closure properties. One such restricted case is the class of *event-clock automata* (ECAs) [3, 25, 26]. The key feature of these automata is that they have a pair of implicit clocks associated with each input symbol. The event clocks record the time elapsed after the last occurrence of the associated symbol, as well as the time that will elapse before the next occurrence of the associated symbol. When an ECA reads a timed word, clock valuations depend only on the input word itself rather than on the choice of nondeterministic transitions. Hence, ECAs are determinizable and closed under Boolean operations.

During the last years, there has been much extensive research on the inclusion problem for timed automata [42, 27, 16]. In particular, it was shown that the inclusion problem $L(A) \subseteq L(B)$, for timed automata A and B, becomes *decidable* if B has at most *one clock* [42]. The key idea of the proof is to encode this inclusion problem as the reachability problem for well-structured transition systems. However, over infinite timed words, one clock is enough to make the inclusion problem undecidable [1].

A *timed pushdown automaton* (TPDA) [13] is a timed automaton augmented with a pushdown stack. Decision problems for TPDAs such as emptiness is decidable [13]. However, the inclusion problem for TPDAs is undecidable, since the corresponding problem is already undecidable for pushdown automata. One, therefore, has to deal with formalism of less expressive power. One such candidate is the class of *visibly* pushdown automata (VPAs, c.f. Chapter 4), in which the stack pushes and pops are determined explicitly by an input alphabet. VPAs are closed under all Boolean operations, and the inclusion problem for VPAs is decidable. Motivated by real-time software verification, Emmi and Majumdar [27] introduced *timed* visibly pushdown automata (TVPAs) as the timed extension of VPAs. However, for TVPAs A and B, the inclusion problem $L(A) \subseteq L(B)$ is *undecidable* even when B has exactly *one clock* [27].

In Chapter 6, inspired by the ideas of ECAs [3] and VPAs [5], we introduce the class of *event-clock visibly pushdown automata* (ECVPAs). The class of ECVPAs is expressive

[5] `http://www.uppaal.com/`

enough to specify common context-free real-time properties such as "if p holds when a procedure is invoked, then the procedure must return within d time units and q must hold at the return state". Besides, the class of ECVPAs is closed under all Boolean operations. Our results are summarized as follows:

1. We show the essence behind the notion of event clocks is that every ECVPA can be translated into an untimed VPA, which interprets timing constraints symbolically, and vice-versa. Therefore, the closure properties and the decidability results of ECVPAs can be reduced to those of VPAs.

2. We use the translation technique to prove that the inclusion problem $L(A) \subseteq L(B)$ for a TVPA A and an ECVPA B is decidable.

3. We show that class of duration automata (DAs) [65] is a special case of ECVPAs. Thus, the inclusion problem for DAs is decidable.

Summary

In summary, we have considered the inclusion problems for subclasses of pushdown automata. The results of this thesis were published in (or, were submitted to) several papers: superdeterministic pushdown automata [66], languages extensions of visibly pushdown automata [68], antichains for visibly pushdown automata [41], and timed extensions for visibly pushdown automata [65, 67].

Chapter 2

Pushdown Automata

In this chapter we present the basic notions and properties of pushdown automata as well as the $\mathcal{P}$-automata technique for solving reachability of pushdown automata.

2.1 Definition of Pushdown Automata

Let $\Sigma = \{a, b, c, ...\}$ be a finite alphabet. The set Σ^* denotes all finite words over Σ. The *empty word* is denoted by ε. A subset of Σ^* is called a *language*. Given a nonempty word $w \in \Sigma^*$ we write $w = a_1 a_2 \cdots a_n$ where $a_i \in \Sigma$ denotes the *i-th symbol* of w for all $1 \leq i \leq n$. Let *head(w)* denote the first letter of w, i.e., *head(w)* $= a_1$. The *length* $|w|$ of w is n and $|\varepsilon| = 0$. The notation $|\cdot|$ also denotes the *cardinality* of a set, the *absolute value* of an integer, and the *size* of a pushdown automaton.

Definition 2.1. *A* pushdown automaton *(PDA) A over an alphabet Σ is a tuple $A = (Q, \Sigma, \Gamma, Z_0, \Delta, q_0, F)$, where*

1. *$Q = \{p, q, r, ...\}$ is a finite set of control states,*
2. *$\Gamma = \{X, Y, Z, ...\}$ is a finite set of stack symbols such that $Q \cap \Gamma = \varnothing$,*
3. *$Z_0 \in \Gamma$ is the initial stack symbol (sometimes, we use $\perp$ for initial symbol instead of Z_0),*
4. *Δ is a finite set of transition rules of the form $(p, X) \xrightarrow{a} (q, \alpha)$ where $p, q \in Q$, $a \in \Sigma \cup \{\epsilon\}$, $X \in \Gamma$, and $\alpha \in \Gamma^*$, and $\epsilon \notin \Sigma$ (empty input word) is a special symbol,*
5. *q_0 is the initial control state, and*
6. *$F \subseteq Q$ is a set of final control states.*

For a rule $(p, X) \xrightarrow{a} (q, \alpha) \in \Delta$, we call (p, X) the *mode* of the rule with input a; if $a = \epsilon$, this is an *ϵ-rule*. We say that a rule $(p, X) \xrightarrow{a} (q, \alpha)$ is a *push*, *internal*, or *pop* rule if $|\alpha| = 2$,1, or 0, respectively. For a given mode (p, X), we define:

- if no rule is defined for (p, X) in $Q \times \Gamma$, (p, X) is a *blocking mode*.
- if no ϵ-*rule* is defined for mode (p, X) and (p, X) is not a blocking mode, we call it a *reading mode*.

Definition 2.2. *We consider two subcases of PDAs:*

- *A PDA is called* real-time *(*RPDA*) if* $(p, X) \xrightarrow{a} (q, \alpha) \in \Delta$ *implies* $a \neq \epsilon$.
- *A PDA is called* deterministic *(*DPDA*) if for every* $p \in Q$, $X \in \Gamma$ *and* $a \in \Sigma \cup \{\epsilon\}$ *we have: (1)* $|\{(q, \alpha) \mid (p, X) \xrightarrow{a} (q, \alpha)\}| \leq 1$, *and (2) if* $(p, X) \xrightarrow{\epsilon} (q, \alpha)$ *and* $(p, X) \xrightarrow{a} (q', \alpha')$ *then* $a = \epsilon$.

Definition 2.3. *Let us denote* $\mathrm{St} = \Gamma^*$. *The set* $Q \times St$ *is the set of* configurations *of a PDA. A pair* $(p, X\beta) \in Q \times St$ *is a configuration* with mode (p, X), *written* $mode((p, X\beta)) = (p, X)$ *(the rightmost symbol is the top-of-stack symbol). For a configuration* $c = (p, y)$, *the control state of* c *is* $state(c) = p$, *and the* stack height *of* c *is* $|c| = |y|$.

The transition relation between configurations is defined by:

1. $(p, X) \xrightarrow{a} (q, \alpha) \in \Delta$, then $(p, X\beta) \xrightarrow{a} (q, \alpha\beta)$ for any $\beta \in \Gamma^*$, and we call it *one-step computation*.
2. A transition $(p, X\beta) \xrightarrow{\epsilon} (q, \alpha\beta)$ is an ϵ-*transition*.
3. $c_1 \xrightarrow{u} c_2$ and $c_2 \xrightarrow{v} c_3$, we write $c_1 \xrightarrow{uv} c_3$ and call it a computation from c_1 to c_3 on the input uv.
4. For any configuration c, we write $c \xrightarrow{\tau} c$, and we call it a *zero-step computation*, where $\tau a = a\tau = a$ for all $a \in \Sigma$.
5. A sequence $c_1 \xrightarrow{a_1} c_2 \cdots \xrightarrow{a_n} c_{n+1}$ of one-step computations is an *n-step computation*.
6. If we have an n-step computation $c_1 \xrightarrow{a_1} c_2 \xrightarrow{a_2} c_3 \cdots \xrightarrow{a_n} c_{n+1}$ with $|c_1| \leq |c_i|$, $1 \leq i \leq n+1$, we write $c_1 \uparrow (a_1 \cdots a_n) c_{n+1}$. This is a *stacking computation*.

Definition 2.4. *The configuration* (q_0, Z_0) *is the* initial *configuration. For a configuration* c, c *is* accessible *if* $(q_0, Z_0) \xrightarrow{w} c$ *for some* $w \in \Sigma^*$. *The configuration* c *is live if* $c \xrightarrow{w} (q, \varepsilon)$ *for some* $q \in F$ *and some* $w \in \Sigma^*$.

Definition 2.5. *A PDA A is of* delay d *if, whenever there is a sequence of one-step computations: $c_1 \xrightarrow{\epsilon} c_2 \xrightarrow{\epsilon} c_3 \cdots \xrightarrow{\epsilon} c_n$, then $n-1 \leq d$ (i.e., at most d ϵ-rules in a row can be applied to any configuration). A PDA A is d* finite delay *if it is of delay d for some $d \geq 0$. It is easy to see that if a PDA is of delay 0, then it is* real-time.

Definition 2.6. *We consider PDAs accepting by a final state and an empty stack. A language accepted from a configuration c is $L(c) = \{w \in \Sigma^* \mid c \xrightarrow{w} (q, \varepsilon), q \in F\}$. The language accepted by a PDA A is $L(A) = L((q_0, Z_0))$.*

Definition 2.7. *The PDAs M_1 and M_2 are equivalent, denoted as $M_1 \equiv M_2$, if they accept the same language, i.e., $L(M_1) = L(M_2)$. Configurations c_1 in M_1 and c_2 in M_2 are equivalent, denoted as $c_1 \equiv c_2$, if $L(c_1) = L(c_2)$.*

Definition 2.8. *A finite automaton is a PDA that only has internal transitions, i.e., the stack can be ignored. In other words, a finite automaton can be treated as PDA without the stack.*

2.2 Properties of Pushdown Automata

Unlike finite automata, PDAs do not enjoy good closure properties and decidability results. This is because product construction fails for PDA, and further, the class of PDA cannot be determinized. The following theorems recall basic properties, which can be found at any textbook of automata theory (e.g., see [72]) of PDA.

Theorem 2.1 ([72])**.** *The emptiness problem for PDAs is decidable in polynomial time.*

Theorem 2.2 ([72])**.** *The class of PDA is closed under union, but not closed under intersection and complementation. Moreover, the inclusion problem for PDA is undecidable.*

However, when the specification are finite automata, the next theorem holds:

Theorem 2.3 ([72])**.** *The inclusion problem $L(A) \subseteq L(B)$, where A is a PDA and B is FA, is decidable.*

For the purpose of our work, it is convenient to use a normal form of pushdown automata.

Definition 2.9. *A pushdown automaton $A = (Q, \Sigma, \Gamma, Z_0, \Delta, q_0, F)$ is* normalized *if*

1. *for all $p \in Q$ and $X \in \Gamma$, (p,X) is not a blocking mode,*
2. *for all $p \in Q$, all rules in δ of the form $(p, X) \xrightarrow{a} (q, \alpha)$ either satisfy $a \in \Sigma$ or all of them satisfy $a = \epsilon$, but not both,*

3. *every rule in* δ *is of the form* $(p, X) \xrightarrow{a} (q, \varepsilon)$, $(p, X) \xrightarrow{a} (q, X)$, *or* $(p, X) \xrightarrow{a} (q, YX)$ *where* $a \in \Sigma \cup \{\epsilon\}$.

States which admit only ϵ-*rules* (see property (2)), are called ϵ-*states*. The next lemma enables us to convert an arbitrary PDA into an equivalent normalized PDA.

Lemma 2.4 ([45])**.** *For every PDA (DPDA) there is a normalized PDA (DPDA) that recognizes the same language.*

Proof. Given a PDA A, we construct PDAs A_1, A_2, A_3 recognizing the same language as A such that A_1 satisfies (3), A_2 satisfies (3) and (2), A_3 satisfies (3), (2), and (1), respectively. In what follows, the components of the automata A_1, A_2 and A_3 will be indexed accordingly, i.e., for example the set of control states of A_1 will be denoted by Q_1, the set of rules of A_2 will be denoted by δ_2 etc.

For A_1, we first use the standard construction that yields an automaton A' equivalent to A but in which every rule $(p, X) \xrightarrow{a} (q, \alpha)$ satisfies $|\alpha| \leq 2$. Then, we construct A_1 as described in Lemma 10.2 of [72]; A_1 has $Q_1 = Q' \times \Gamma$ as the set of states and is defined to satisfy the property that $(p, X\alpha) \xrightarrow{w} (q, Y\beta)$ holds iff $(p, X)\alpha \xrightarrow{w} (q, Y)\beta$ does. It also follows from the lemma that if A is deterministic, then so is A_1 (actually, the lemma is stated for DPDA, but the construction works in general).

The PDA A_2 is the final result of the following procedure: For every pair of rules of A_1 of the form $(p, X) \xrightarrow{a} (q, \alpha)$ and $(p, Y) \xrightarrow{\epsilon} (r, \beta)$ such that $a \in \Sigma$, add a state p_X not in Q_2 and new rules $(p, X) \xrightarrow{\epsilon} (p_X, X)$ and $(p_X, X) \xrightarrow{a} (q, \alpha)$ to A_1 and remove the rule $(p, X) \xrightarrow{a} (q, \alpha)$ from δ_1. Clearly, A_2 satisfies (2) and, since A_1 satisfies (3) so does A_2. Observe that if A_1 is deterministic then A_2 is also deterministic.

The PDA A_3 is obtained by adding to A_2 a dead state d, the rules $(d, X) \xrightarrow{a} (d, X)$ for every $X \in \Gamma$ and $a \in \Sigma$, a rule $(p, X) \xrightarrow{a} (d, X)$ for every non-ϵ-state $p \in Q_2$ and every $X \in \Gamma$ and $a \in \Sigma$ whenever $(p, X) \xrightarrow{a} (q, \alpha)$ is not already in δ_2 for some q and a, and, a rule $(p, X) \xrightarrow{\epsilon} (d, X)$ for every ϵ-state $p \in Q_2$. □

2.3 Deciding Emptiness of PDAs

The classical solution to the emptiness problem of PDAs is based on Pumping Lemma (e.g., see [72]). It however is difficult to implement this method in practice, and thus there are no implementation based on this approach. Recently, Finkel et al. [30] and Esparza et al. [28, 29] have introduced an efficient symbolic method to check the emptiness of PDAs. The key of their technique is to use a finite automaton so-called $\mathcal{P}$-*automaton* to encode infinite sets of configurations of a pushdown automaton. Thus, checking emptiness is reduced to the problem of computing forward (backward) reachability based on the PDA

and $\mathcal{P}$-automaton. This method has been implemented in the efficient automatic tool Moped. In the following, we briefly recall $\mathcal{P}$-automata technique.

Given a pushdown automaton $\mathcal{P} = (P, \Sigma, \Gamma, \perp, \Delta, s_0, \mathcal{F})$, a $\mathcal{P}$-automaton is used in order to represent sets of configurations C of $\mathcal{P}$. A $\mathcal{P}$-automaton uses Γ as the input alphabet, and P as set of initial states (we consider automata with possibly many initial states). Formally,

Definition 2.10 ([28, 58]). *A $\mathcal{P}$-automaton is a finite automaton $A = (Q, \Gamma, \delta, P, F)$ where Q is the finite set of states, $\delta \subseteq Q \times \Gamma \times Q$ is the set of transitions, P is the set of* initial states *and $F \subseteq Q$ is the set of* final states. *A $\mathcal{P}$-automaton* accepts *or* recognizes *a configuration (p, w) if $p \xrightarrow{w} q$, for some $p \in P$, $q \in F$. The set of configurations recognized by $\mathcal{P}$-automaton A is denoted by $Conf(A)$. A set of configurations of $\mathcal{P}$ is regular if it is recognized by some $\mathcal{P}$-automaton.*

For a PDA $\mathcal{P} = (Q, \Sigma, \Gamma, \perp, \Delta, s_0, \mathcal{F})$ and the set of configurations C, let A be a $\mathcal{P}$-automaton representing C. The $\mathcal{P}$-automaton A_{post^*} representing the set of configurations reachable from C ($Post^*(C)$) is constructed as follows: We compute $Post^*(C)$ as a language accepted by a $\mathcal{P}$-automaton A_{post^*} with ϵ-moves. We denote the relation $q(\xrightarrow{\epsilon})^* \cdot \xrightarrow{\gamma} \cdot(\xrightarrow{\epsilon})^* \cdot p$ by $\Longrightarrow^{\gamma}$. A_{post^*} is obtained from A in two stages:

- For each pair (q', γ') such that $\mathcal{P}$ contains at least one rule of the form $(q, \gamma) \rightarrow (q', \gamma'\gamma'')$, add a new state $p_{(q',\gamma')}$.
- Add new transitions to A according to the following saturation rules:

1. If $(q, \gamma) \xrightarrow{a} (q', \epsilon) \in \Delta$ and $q \Longrightarrow^{\gamma} p$ in the current automaton, add a transition (q', ϵ, p).
2. If $(q, \gamma) \xrightarrow{a} (q', \gamma') \in \Delta$ and $q \Longrightarrow^{\gamma} p$ in the current automaton, add a transition (q', γ', p).
3. If $(q, \gamma) \xrightarrow{a} (q', \gamma'\gamma'') \in \Delta$ and $q \Longrightarrow^{\gamma} p$ in the current automaton, first add $(q', \gamma', p_{(q',\gamma')})$, and then add $(p_{(q',\gamma')}, \gamma'', p)$.

Example 2.1. *Let us illustrate the definition of $\mathcal{P}$-automata by an example. Consider the PDA with control locations $\mathcal{P} = \{q_0, q_1, q_2\}$ and $\Delta = \{r_1, r_2, r_3, r_4\}$, where:*

- $r_1 = (q_0, \gamma_0) \xrightarrow{a} (q_1, \gamma_1\gamma_0)$, $r_2 = (q_1, \gamma_1) \xrightarrow{a} (q_2, \gamma_2\gamma_0)$
- $r_3 = (q_2, \gamma_2) \xrightarrow{b} (q_0, \gamma_1)$, $r_4 = (q_0, \gamma_1) \xrightarrow{b} (q_0, \epsilon)$.

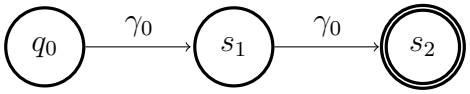

Figure 2.1: The finite automaton C

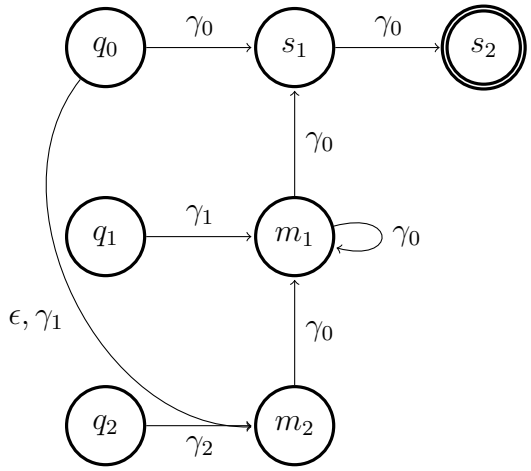

Figure 2.2: The $\mathcal{P}$-automaton $post^*(C)$

Let A be the automaton that accepts the set of configurations $C = \{(q_0, \gamma_0\gamma_0)\}$*, shown in Figure 2.1.*

Then, the automaton shown in Figure 2.2 is the result of the algorithm and accepts $post^*(\{(q_0, \gamma_0\gamma_0)\})$. The result is achieved by the following steps:

1. First, we add the new states: $r_1 = (q_0, \gamma_0) \rightarrow (q_1, \gamma_1\gamma_0)$ leads to $p_{(q_1,\gamma_1)}$ (denoted by m_1) and $r_2 = (q_1, \gamma_1) \rightarrow (q_2, \gamma_2\gamma_0)$ to $p_{(q_2,\gamma_2)}$ (denoted by m_2).

2. Since $q_0 \xrightarrow{\gamma_0} s_1$ matches the left-hand side of rule r_1, the first transitions we add are $(q_1, \gamma_1, p_{(q_2,\gamma_2)})$ and $(p_{(q_1,\gamma_1)}, \gamma_0, s_1)$.

3. The first of these new transitions together with rule r_2 gives rise to two more transitions: $(q_2, \gamma_2, p_{(q_2,\gamma_2)})$ and $(p_{(q_2,\gamma_2)}, \gamma_0, p_{(q_1,\gamma_1)})$.

4. Because of $(q_2, \gamma_2, p_{(q_2,\gamma_2)})$ and r_3 the next step is to add $(q_0, \gamma_1, p_{(q_2,\gamma_2)})$.

5. This in turn, together with r_4 leads to the ϵ-edge from q_0 to $p_{(q_2,\gamma_2)}$.

6. We now have $q_0 \xrightarrow{\gamma_0}{}^* p_{(q_1,\gamma_1)}$ and can apply rule r_1 once more. Because $(q_1, \gamma_1, p_{(q_1,\gamma_1)})$ has been added before, we just add $(p_{(q_1,\gamma_1)}, \gamma_0, p_{(q_1,\gamma_1)})$.

7. No unprocessed matches remain, so the procedure terminates.

Remark 2.1. It is easy to note that the emptiness problem of PDA P is a subcase of the problem computing configurations reachable from the initial configuration of P.

2.4 Applications to Pushdown Model Checking

Pushdown automata naturally model the control flow of sequential computation in typical programming languages with nested, and potentially recursive, invocations of program modules such as procedures and method calls. Consequently, a variety of program analysis, compiler optimization, and model checking questions can be formulated as decision problems for pushdown automata. For instance, in contemporary software model checking tools, to verify whether a program P (written in C, for instance) satisfies a regular correctness requirement φ (written in linear temporal logic, for instance), the verifier first abstracts the program into a pushdown model P^a with finite-state control, compiles the negation of the specification into a finite-state automaton $A_{\neg\varphi}$ that accepts all computations that violate φ and algorithmically checks that the intersection of the languages of P^a and $A_{\neg\varphi}$ is empty. Namely, the following equalities hold:

$$P \models \varphi \iff L(P^a) \subseteq L(A_\varphi) \iff L(P^a) \cap L(A_{\neg\varphi}) = \varnothing$$

The automata-theoretic approach works for pushdown model checking on regular properties, since the intersection of context-free languages and regular languages are closed, i.e. context-free. The pushdown model checking problem on CTL (computation tree logic) is known to be DEXPTIME-complete. Whereas pushdown model checking on LTL (linear time logic) is known to be polynomial. The problem of checking regular requirements of pushdown models has been extensively studied in recent years [31, 28, 29]. Tools implementing pushdown model-checking (e.g. Moped) are an essential back-end component of high-profile software model checkers such as Slam [1].

However, when the specification is extended to context-free properties, the model checking problem becomes undecidable. Alur et al. recently introduced the logic CaRet [6]. CaRet is a linear temporal logic that can specify non-regular properties. Pushdown model checking for CaRet logic is decidable in exponential time [6]. We note that CaRet is less expressive than pushdown automata. The study of CaRet inspired Alur and Madhusudan [5] to introduce *visibly pushdown automata* (VPA). In this thesis, we are interested in the inclusion problems for subclasses of pushdown automata such as visibly pushdown automata and superdeterministic pushdown automata, which we will present in details in the following chapters.

[1] `http://research.microsoft.com/slam/`

Chapter 3

Superdeterministic Pushdown Automata

Recent interest in model checking makes us recall inclusion problems. Typically, the automata theoretic explanation of model checking on finite transition systems is the decidability of the inclusion problem $L(A) \subseteq L(B)$ among finite automata, where A and B describe a model and a specification, respectively. The standard methodology for the inclusion problem is to, (1) take the complement $\overline{L(B)}$, (2) take the intersection between $L(A)$ and $\overline{L(B)}$, and (3) check its emptiness. This also works when A is extended to a *pushdown automaton* (PDA), but fails when B is extended to a pushdown automaton. To our knowledge, for decidable inclusion with a general pushdown automaton A, the largest class of B is the *superdeterministic pushdown automata* (SPDAs), proposed by Greibach and Friedman [34]. An SPDA is a DPDA satisfying:

1. *finite delay* (i.e., a bounded number of ϵ-transitions in a row can be applied to any configuration), and

2. for two configurations sharing the same control state, transitions with the same symbol lead to configurations sharing the same control state such that the length change of stacks is the same.

In [34], the authors used the alternate stacking technique [74] to show that the inclusion problem $L(A) \subseteq L(B)$, where A is a PDA and B is an SPDA, is decidable. The key idea of the original proof [34] is to construct a simulating pushdown automaton M such that $L(A) \subseteq L(B)$ iff $L(M) = \emptyset$. However, the original construction encodes everything as stack symbols (in an intricate way), and thus control states and transition rules of M could not be given in details. Furthermore, to decide the emptiness of M, one has to use an auxiliary procedure to check whether a configuration of the PDA A is *live* (i.e., whether a configuration reaches an accepting configuration) or not. These properties of their simulating PDA M lead to a complicated proof of soundness and completeness for the decision procedure [34].

In this chapter, we refine the alternate stacking technique [74] used in Greibach-Friedman's proof [34]. Basically, there are three main steps in the proof of the decidability of the inclusion problem $L(A) \subseteq L(B)$, where A is a PDA and B is an SPDA. First, establishing Key lemma (Lemma 3.3 [34]) to find a bounded number k that is used for alternate stacking. Second, constructing a simulating PDA M by using the alternate stacking technique (Section 3.2). Third, based on the construction of M in the second step, proving soundness and completeness of the construction $L(A) \subseteq L(B)$ iff $L(M) = \emptyset$ (Section 3.3). Our refinement contributes to the last two steps. In particular, we give a more direct product construction of the simulating PDA M, which is different from the one given by the original proof, where everything is encoded as a stack symbol. This construction avoids the need for the "liveness" condition, and the correctness proof becomes simpler.

3.1 Superdeterministic Pushdown Automata

Superdeterministic pushdown automata (*SPDAs*) were first introduced by Greibach and Friedman [34]. In this section, we briefly recall the standard notion and key properties of SPDAs. Readers are referred to the original paper [34] for more details.

Definition 3.1. *A PDA* $A = (Q, \Sigma, \Gamma, Z_0, \Delta, q, F)$ *is* superdeterministic *if it satisfies the following conditions.*

1. *A is deterministic and of finite delay,*
2. *for all accessible configurations in reading mode* c_1, c_2, c_1', c_2' *and* $w \in \Sigma^*$*, if both of the following are satisfied:*
   - $state(c_1) = state(c_2)$*,*
   - $c_1 \xrightarrow{w} c_1'$ *and* $c_2 \xrightarrow{w} c_2'$*,*

   then, $state(c_1') = state(c_2')$ *and* $|c_1| - |c_1'| = |c_2| - |c_2'|$*.*

Remark 3.1. *In [34], Greibach and Friedman considered the blocking condition on PDAs (middle, pp.677): "Unlike Valiant, we do not allow the pda to operate with empty stack (no rules* (q, ϵ, a, p, y)*). This avoids some complications in notation but does not affect the classes of languages involved because we allow endmarkers". In particular, the blocking condition is not an essential restriction if we use two special symbols # (start-maker) and $ (end-marker), where # pushes a special stack symbol, and $ pops it. This assumption was used to prove Key lemma (Lemma 2). More precisely, it was used to show the claim (middle, pp.684 [34]) that: "Hidden in many of our arguments is the following consequence*

of determinism and acceptance by empty store. Suppose $L(c_1) \subseteq L(\bar{c}_1)$ *with* $\bar{c}_1$ *(but not necessarily* c_1*) a configuration in a deterministic pda,* $c_1 \xrightarrow{w} c_2$*, and* $\bar{c}_1 \xrightarrow{w} \bar{c}_2$*. Then* $L(c_2) \subseteq L(\bar{c}_2)$*".*

Definition 3.2. *A language* L *is superdeterministic if there is an SPDA* M *such that either* $L = L(M)$ *or* $L\$ = L(M)$ *for an end-marker* \$*.*

Note that the language $\{a^n b^n \mid n \geq 0\}$ is superdeterministic. However, according to Condition 2 in Definition 3.1, the language $L = \{a^n b^m \mid m \geq n\}$ is not accepted by any SPDA (pp.678 [34]). Suppose on the contrary that there is an SPDA A accepting L. While reading a, A pushes a symbol, and while reading b, A pops the same symbol. Thus, for instance, after reading a^5 and a^{10}, A will be in two configurations, c_1 and c_2, such that $state(c_1) = state(c_2)$. Now concatenating b^{10}, A will lead to configurations c_1' (for $a^5 b^{10}$) and c_2' (for $a^{10} b^{10}$), respectively. However, $|c_1'| - |c_1| = 0 - 5 \neq |c_2'| - |c_2| = 0 - 10$. This violates the definition of SDPAs. Moreover, as shown in [34], the class of *superdeterministic languages* (languages accepted by SPDAs) contains the *generalized parenthesis languages*, which is a superclass of both *parenthesis languages* [43] and *Dyck sets*.

Remark 3.2. *It is undecidable whether a given context-free language is superdeterministic. However, it is decidable whether a given PDA* M *is an SPDA (pp.678 [34]): "It is decidable whether a dpda* M *is finite delay (using the decidability of emptiness and finiteness for* context-free grammars *and the standard construction of grammars from machines), and if* M *is of finite delay, an upper bound* d *on the delay can be computed from a description of* M*. Knowing that* M *is of delay* d*, it can be determined whether or not* M *is superdeterministic by examining only computations* $c \xrightarrow{a} c'$ *for a symbol* a *with* c *and* c' *in reading mode. Since it is decidable for* q *in* Q*,* y *in* Γ^* *whether there is a* u *in* Γ^* *with* (q, uy) *accessible, it is decidable whether a dpda is superdeterministic. It is not known if it is decidable whether a deterministic context-free language is superdeterministic, just as it is not known whether it is decidable whether a deterministic context-free language is* finite-turn *or* one-counter *[74]. Standard arguments show that it is undecidable whether an arbitrary context-free language is superdeterministic".*

A PDA is called *one-increasing* if the stack height increases by at most one per move. As is well known, each PDA can be transformed into an equivalent one-increasing PDA.

Lemma 3.1 (Key Lemma 3.3 [34])**.** *Let* A *be a normalized PDA, and* B *be a one-increasing SPDA of delay* d*. Let* c_1 *be a configuration in* A *and* c_1' *be an accessible configuration in* B *with* $L(c_1) \subseteq L(c_1')$*. Suppose we have in* A *a computation* $c_1 \uparrow (w)c_2$*, with* c_2 *live, and in* B *a computation* $c_1' \xrightarrow{w} c_2'$*. Then,*

1. $|c_1'| - |c_2'| \leq k$,

2. *and if* $|c_1| = |c_2|$ *then* $|c_2'| - |c_1'| \leq k$, *where*

   - $k = (d+1)(k_1+1)n(m+1)^{2k_2} + 2d$,
   - $k_1 = n+3$, $k_2 = 1 + 2n^2m^2(n^2+4)$,
   - $n = |Q_A| + |Q_B|$, $m = |\Gamma_A| + |\Gamma_B|$.

Based on this property, in the next section, we show that the inclusion problem $L(A) \subseteq L(B)$ is decidable for a PDA A and an SPDA B.

3.2 Alternate Stacking Technique

The *alternate stacking technique*, proposed by Valiant [74], involves a simulation of two PDAs A and B using a single stack machine M whose stack contents $u_1v_1 \cdots u_tv_t$ encode the stack $u_1 \cdots u_t$ of A and $v_1 \cdots v_t$ of B; the machine M uses u_i to simulate one step of A and v_r for one step of B. In the general case, the simulating machine M is not a PDA. Alternate stacking "succeeds" when the stacks can be interwoven in such a way that M can be implemented as a PDA. Valiant [74] showed that if A and B are *nonsingular DPDAs* [1] and $L(A) = L(B)$, then the interweaving can indeed be done so that a uniform bound can be placed on the length of segments u_i and v_i so long as the configurations of A and B are live. Then the PDA M can be built so that if the stack segments exceed the bound, M accepts, knowing that $L(A) \neq L(B)$. Hence $L(A) = L(B)$ iff $L(M) = \varnothing$.

3.2.1 Simulating Pushdown Automata

In this subsection, we construct a simulating PDA M such that M will search for possible members of $L(A) \setminus L(B)$. In principle, similar to [34], the key is to use the alternate stacking technique to construct M. In our approach, however, the control states, stack symbols, and transition rules of M are defined in the form of pairs of states, stack content, and transition rules of two PDAs, respectively.

We assume that $A = (Q_A, \Sigma, \Gamma_A, Z_A, \Delta_A, q_A^0, F_A)$ is a normalized PDA, and $B = (Q_B, \Sigma, \Gamma_B, Z_B, \Delta_B, q_B^0, F_B)$ is a normalized SPDA of delay d with an assumption that $0 \notin Q_B$.

1. Let $\$_1$ and $\$_2$ be fresh symbols to mark the bottom of the stack of A and B, respectively.

[1] A DPDA M is *nonsingular* if and only if there exists $m \geq 0$ such that for any two accessible configurations (q, ww') and (q', w') where $|w| > m$, if $L((q, ww')) = L((q', w'))$ then $L((q', w')) = \emptyset$.

2. Let $f : \Gamma_B^* \cup \Gamma_B^*\$_2^* \to \Gamma_B^*$ be a function such that $f(y) = f(y\$_2^*) = y$ for all $y \in \Gamma_B^*$.

3. Let $r > 0$ be an integer and let us take $2r$ as the segment bound for simulating the stack content of B.

4. Denote $\Gamma'_B = \{[y], [y\$_2] \mid y \in \Gamma_B^*, 0 \leq |y| \leq 2r\}$.

A simulating PDA $M = M(A, B, r)$ can be constructed for any choice of r, and the next theorem, Theorem 3.4, will show that if the bound r is appropriately selected ($r = k + 1$, where k was computed from A and B as in Theorem 3.1), then we can conclude that $L(A) \subseteq L(B)$ iff $L(M(A, B, k + 1)) = \varnothing$. Formally, the simulating PDA $M = M(A, B, r)$ is constructed as follows:

Definition 3.3. *A simulating PDA of two PDAs A and B is a tuple $M = M(A, B, r) = \langle Q_M, \Sigma, \Gamma_M, Z_M, \Delta_M, p_M^0, F_M \rangle$, where:*

- $Q_M = \{p_M^0\} \cup (Q_A \times Q_B) \cup (Q_A \times \{0\}) \cup (Q_A \times Q_B \times \Gamma'_B)$ *is the set of finite states,*
- p_M^0 *is the initial state,*
- $F_M = (F_A \times (Q_B \backslash F_B)) \cup (F_A \times \{0\})$,
- $\Gamma_M = (\Gamma_A \cup \{\$_1\}) \times \Gamma'_B$, $Z_M = (\$_1, [\$_2])$,
- *The transition relation $\Delta_M \subseteq Q_M \times \Gamma_M \times \Sigma \times (Q_M \times \Gamma_M^*)$ is defined as follows:*

Case I: *Simulating an internal-transition of A with a transition of B:*

1. $\langle (p_1, p_2), (X, [Zv]) \rangle$ [2] $\xrightarrow{a} \langle (p'_1, p'_2), (X, [yv]) \rangle$ *if:* $(p_1, X) \xrightarrow{a} (p'_1, X) \in \Delta_A$, $(p_2, Z) \xrightarrow{a} (p'_2, y) \in \Delta_B$, $yv \neq \epsilon$, *and* $|f(yv)| \leq 2r$.

2. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{a} \langle (p'_1, 0), (X, [yv]) \rangle$ *if:* $(p_1, X) \xrightarrow{a} (p'_1, X) \in \Delta_A$, $(p_2, Z) \xrightarrow{a} (p'_2, y) \in \Delta_B$, *and* $yv = \varepsilon$ *or* $|f(yv)| = 2r + 1$.

3. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{a} \langle (p'_1, 0), (X, [Zv]) \rangle$ *if:* $(p_1, X) \xrightarrow{a} (p'_1, X) \in \Delta_A$ *and* (p_2, Z) *has no rules with input a*

4. $\langle (p_1, 0), (X, [v]) \rangle \xrightarrow{a} \langle (p'_1, 0), (X, [v]) \rangle$ *for all* $[v] \in \Gamma'_B$ *if* $(p_1, X) \xrightarrow{a} (p'_1, X) \in \Delta_A$.

5. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{\epsilon} \langle (p'_1, p_2), (X, [Zv]) \rangle$ *if:* $(p_1, X) \xrightarrow{\epsilon} (p'_1, X) \in \Delta_A$, *and* (p_2, Z) *has no ϵ-rules.*

6. $\langle (p_1, p_2), (X, [\$_2]) \rangle \xrightarrow{a} \langle (p'_1, 0), (X, [\$_2]) \rangle$ *if* $(p_1, X) \xrightarrow{a} (p'_1, X) \in \Delta_A$

Case II: *Simulating a push-transition of A with a transition of B.*

[2] For readability, we use $\langle ., . \rangle$ to denote a configuration of the simulating PDA M.

1. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{a} \langle (p_1', p_2'), (X', [yv])(X, [\$_2]) \rangle$ *if:* $(p_1, X) \xrightarrow{a} (p_1', XX') \in \Delta_A$, $(p_2, Z) \xrightarrow{a} (p_2', y) \in \Delta_B$, *and* $head(yv) = \$_2, |f(yv)| \leq r$.

2. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{a} \langle (p_1', p_2'), (X, [\varepsilon])(X', [yv]) \rangle$ *if:* $(p_1, X) \xrightarrow{a} (p_1', X'X) \in \Delta_A$, $(p_2, Z) \xrightarrow{a} (p_2', y) \in \Delta_B$, *and* $head(yv) \neq \$_2, |f(yv)| \leq r$

3. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{a} \langle (p_1', p_2'), (X', [v'])(X, [v']) \rangle$ *if:* $(p_1, X) \xrightarrow{a} (p_1', X'X) \in \Delta_A$, $(p_2, Z) \xrightarrow{a} (p_2', y) \in \Delta_B$, $r < |f(yv)| \leq 2r$, $yv = v''v'$, *and* $|v''| = r$.

4. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{a} \langle (p_1', p_2'), (X', [v'])(X, [v']) \rangle$ *if:* $(p_1, X) \xrightarrow{a} (p_1', X'X) \in \Delta_A$, $(p_2, Z) \xrightarrow{a} (p_2', y) \in \Delta_B$, $|f(yv)| = 2r + 1$, $yv = v''v'$, *and* $|v''| = r + 1$.

5. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{a} \langle (p_1', 0), (X', [Zv])(X, [\varepsilon]) \rangle$ *if:* $(p_1, X) \xrightarrow{a} (p_1', X'X) \in \Delta_A$, *and* (p_2, Z) *has no rules with input* a.

6. $\langle (p_1, 0), (X, [v]) \rangle \xrightarrow{a} \langle (p_1', 0), (X', [v])(X, [v]) \rangle$ *for all* $[v] \in \Gamma_B'$ *if:* $(p_1, X) \xrightarrow{a} (p_1', X'X) \in \Delta_A$.

7. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{\epsilon} \langle (p_1', p_2), (X', [Zv])(X, [\$_2]) \rangle$ *if:* $(p_1, X) \xrightarrow{\epsilon} (p_1', X'X) \in \Delta_A$, $head(Zv) = \$_2, |f(Zv)| \leq r$, *and* (p_2, Z) *has no* ϵ*-rules.*

8. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{\epsilon} \langle (p_1', p_2), (X', [Zv])(X, [\varepsilon]) \rangle$ *if:* $(p_1, X) \xrightarrow{\epsilon} (p_1', X'X) \in \Delta_A$, $head(Zv) \neq \$_2, |f(Zv)| \leq r$, *and* (p_2, Z) *has no* ϵ*-rules.*

9. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{\epsilon} \langle (p_1', p_2), (X', [v"])(X, [v']) \rangle$ *if:* $(p_1, X) \xrightarrow{\epsilon} (p_1', X'X) \in \Delta_A$, $r < |f(Zv)| \leq 2r, Zv = v''v', |v''| = r$, *and* (p_2, Z) *has no* ϵ*-rules.*

10. $\langle (p_1, p_2), (X, [\$_2]) \rangle \xrightarrow{a} \langle (p_1', 0), (X, [\$_2])(X', [\$_2]) \rangle$ *if:* $(p_1, X) \xrightarrow{a} (p_1', X'X) \in \Delta_A$.

Case III: *Simulating a pop-transition of* A *with a transition of* B*:*

1. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{a} \langle (p_1', p_2', [yf(v)]), \varepsilon \rangle$, *and* $\langle (p_1', p_2', [f(yv)]), (X', [v']) \rangle \xrightarrow{\epsilon} \langle (p_1', p_2'), (X', [f(yv')v]) \rangle$ *if:* $(p_1, X) \xrightarrow{a} (p_1', \varepsilon) \in \Delta_A$, $(p_2, Z) \xrightarrow{a} (p_2', y) \in \Delta_B$, *and* $|f(f(yv')v)| \leq 2r$.

2. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{a} \langle (p_1', p_2', [f(yv)]), \varepsilon \rangle$, *and* $\langle (p_1', p_2', [f(yv)]), (X', [v']) \rangle \xrightarrow{\epsilon} \langle (p_1', 0), (X', [\varepsilon]) \rangle$ *if:* $(p_1, X) \xrightarrow{a} (p_1', \varepsilon) \in \Delta_A$, $(p_2, Z) \xrightarrow{a} (p_2', y) \in \Delta_B$, *and* $yv'v = \varepsilon$ *or* $|f(f(yv')v)| \geq 2r + 1$.

3. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{a} \langle (p_1', 0), \varepsilon \rangle$ *if:* $(p_1, X) \xrightarrow{a} (p_1', \varepsilon) \in \Delta_A$, *and* (p_2, Z) *has no rules with input* a.

4. $\langle (p_1, 0), (X, [v]) \rangle \xrightarrow{a} \langle (p_1', 0), \varepsilon \rangle$ *for all* $[v] \in \Gamma_B'$ *if* $(p_1, X) \xrightarrow{a} (p_1', \varepsilon) \in \Delta_A$

5. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{\epsilon} \langle (p'_1, p_2, [f(Zv)]), \varepsilon \rangle$, *and* $\langle (p'_1, p_2, [f(Zv)]), (X', [v']) \rangle \xrightarrow{\epsilon} \langle (p'_1, p_2), (X', [f(Zv)v']) \rangle$ *if:* $(p_1, X) \xrightarrow{\epsilon} (p'_1, \varepsilon) \in \Delta_A$, $Zv = \$_2$, *and* (p_2, Z) *has no* ϵ*-rules.*

6. $\langle (p_1, p_2), (X, [\$_2]) \rangle \xrightarrow{a} \langle (p'_1, 0), \varepsilon \rangle$ *if:* $(p_1, X) \xrightarrow{a} (p'_1, \varepsilon) \in \Delta_A$.

Case IV: *When stack of* A *is empty.*

1. $\langle (p_1, p_2), (\$_1, [Zv]) \rangle \xrightarrow{\epsilon} \langle (p_1, p'_2), (\$_1, [yv]) \rangle$ *if* $(p_2, Z) \xrightarrow{\epsilon} (p'_2, y) \in \Delta_B$.

2. $\langle (p_1, p_2), (\$_1, [Zv]) \rangle \xrightarrow{\epsilon} \langle (p_1, 0), \varepsilon \rangle$ *if:* $(p_2, Z) \xrightarrow{a} (p'_2, y) \in \Delta_B$ *with* $a \neq \epsilon$, *or* (p_2, Z) *is blocked.*

Case V: *When configurations of* A *are in the reading modes, while states of* B *have* ϵ*-transitions.*

1. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{\epsilon} \langle (p_1, p'_2), (X, [yv]) \rangle$ *if:* (p_1, X) *is in the reading mode,* $(p_2, Z) \xrightarrow{\epsilon} (p'_2, y) \in \Delta_B$, *and* $|f(yv)| \leq 2r$.

2. $\langle (p_1, p_2), (X, [Zv]) \rangle \xrightarrow{\epsilon} \langle (p_1, 0), (X, [\varepsilon]) \rangle$ *if:* (p_1, X) *is in the reading mode,* $(p_2, Z) \xrightarrow{\epsilon} (p'_2, y) \in \Delta_B$, *and* $|f(yv)| \geq 2r + 1$.

Case VI: *The starting transition:* $\langle p^0_M, (\$_1, [\$_2]) \rangle \xrightarrow{\epsilon} \langle (q^0_A, q^0_B), (Z_A, [Z_B \$_2])(\$_1, [\$_2]) \rangle$.

Before defining configurations of M, let us briefly explain the intuition behind its transition rules.

- Rules I(2), III(2), and V(2) are called *stacking-fail* transitions. Taking a stacking-fail transition, M changes its control to states in the set $Q_A \times \{0\}$. After entering this set $Q_A \times \{0\}$ of states, M continues simulating transitions of A only by using rules I(4), II(6), or III(4).

- Rules I(3), II(5), and III(3) are used when B is blocked where reading an input. In this cases, M changes its control state to the set $Q_A \times \{0\}$. After entering a state in $Q_A \times \{0\}$, M only simulates transitions of A by using I(4), II(6), or III(4).

- Rules II(1), II(2), II(3), and II(4) are used to simulate a push-transition of A with a transition of B, which has the same label.

- Rules III(1), III(2) are used to simulate a pop-transition of A.

- Rules IV(1) and IV(2) are used when the stack of A is empty; in this case, M simulates ϵ-transitions of B using a zero-step computation of A. Recall that B is finite delay of d, and thus rules IV(1) and IV(2) can be applied at most d times in a sequence.

- Rules I(5), II(7)(8)(9), and III(5) are used to simulate an ϵ-transition of A with a non-ϵ transition of B.
- Rules I(6), II(8), and III(6) are used to simulate a non-ϵ transition of A when B's stack is empty (i.e., when B is blocked).

Definition 3.4 (Configuration)**.** *For a simulating PDA M, we define:*

- *A configuration of M is of the form $c = \langle s, (X_t, [v_t]) \cdots (X_1, [v_1])(\$_1, [\$_2])\rangle$, where $s \in Q_M$ and $(X_i, [v_i]) \in \Gamma_M$ for $1 \leq i \leq t$.*
- *The configuration c is accepting if $c = \langle s, \varepsilon\rangle$, $s \in F_M$.*
- *For a given configuration $c = \langle s, (X_t, [v_t]) \cdots (\$_1, [\$_2])\rangle$, we say that c encodes $c_1 = (p_1, X_t...X_1)$ of A and $c_2 = (p_2, f(v_t)...f(v_1))$ of B, with t levels.*

Note that $f(v_t)...f(v_1) \in \Gamma_B^*$, and M can determine whether the stack of B is empty by examining if $v_t = \$_2$, i.e., $|c_2| = 0$ iff $v_t = \$_2$. This is because, based on the transition rules II(1) and II(7), if $v_t = \$_2$ then $v_i = \$_2$ for all $1 \leq i \leq t$ (for these rules, we need to check if $head(yv) = \$_2$).

Remark 3.3. *There are three main steps in the proof of the decidability of the inclusion problem $L(A) \subseteq L(B)$, where A is a PDA and B is an SPDA. First, establish the Key lemma to find a bounded number k that is used for alternate stacking. Second, construct a simulating PDA M by using the alternate stacking technique. Third, based on the construction of M in the second step, prove the soundness and completeness of the construction $L(A) \subseteq L(B)$ iff $L(M) = \emptyset$. Our refinement contributes to the last two steps. In particular, in the original proof [34], the liveness condition is stated in the construction case II (pp. 693 [34]) and it is used for searching words that are accepted by the PDA A, but rejected by the SPDA B. In our encoding, control states, stack symbols, and transition rules M are defined in the form of pairs of states, stack content, and transition rules of two PDAs A and B, respectively. Thus, we do not need to use the "liveness" condition, because such violation of the inclusion is represented by transition rules in our product construction of M. As we will see in Section 4, a proof of "liveness" is not needed and the whole correctness proof for the decision procedure becomes simpler.*

3.2.2 An Illustrating Example

This subsection provides an example to illustrate our construction of the simulating pushdown automata. In the following figures, for simplicity, we describe control states of each PDA as nodes of a graph. We adopt the following conventions to represent edges: for a transition rule $(p, X) \xrightarrow{a} (q, y)$, we label the edge from p to q as $a, X \rightarrow y$.

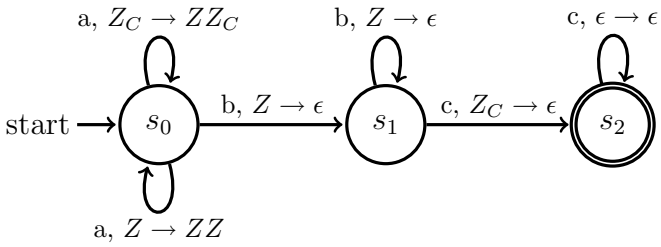

Figure 3.1: Pushdown automaton C

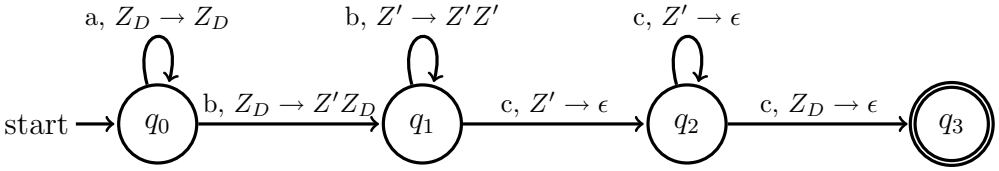

Figure 3.2: Superdeterministic pushdown automaton D

Example 3.1. *Consider two PDAs C and D over the input alphabet $\Sigma = \{a, b, c\}$, where D is an SPDA. The PDA $C = (\{s_0, s_1, s_2\}, \Sigma, \{Z, Z_C\}, Z_C, \Delta_C, \{s_0\}, \{s_2\})$, where Δ_C is defined as:*

- $(s_0, Z_C) \xrightarrow{a} (s_0, ZZ_C)$, $(s_0, Z) \xrightarrow{a} (s_0, ZZ)$, $(s_0, Z) \xrightarrow{b} (s_1, \varepsilon)$
- $(s_1, Z) \xrightarrow{b} (s_1, \varepsilon)$, $(s_1, Z_C) \xrightarrow{c} (s_1, Z_C)$, $(s_1, Z_C) \xrightarrow{c} (s_2, \varepsilon)$

The SPDA $D = (\{q_0, q_1, q_2, q_3\}, \Sigma, \{Z', Z_D\}, Z_D, \Delta_D, \{q_0\}, \{q_3\})$, where Δ_D is defined:

- $(q_0, Z_D) \xrightarrow{a} (q_0, Z_D)$, $(q_0, Z_D) \xrightarrow{b} (q_1, Z'Z_D)$, $(q_1, Z') \xrightarrow{b} (q_1, Z'Z')$
- $(q_1, Z') \xrightarrow{c} (q_2, \varepsilon)$, $(q_2, Z') \xrightarrow{c} (q_2, \varepsilon)$, $(q_2, Z_D) \xrightarrow{\epsilon} (q_3, \varepsilon)$

It is easy to see that the languages $L(C) = \{a^n b^n c^m \mid n \geq 1, m \geq 1\}$ and $L(D) = \{a^m b^n c^n \mid m \geq 0, n \geq 1\}$.

The simulating pushdown automaton $M = M(C, D, 1)$ is illustrated in Figure 3.3. In this case, $r = 1$, it is sufficient to consider stack symbols of the forms $(X, [v])$ with $|v| \leq 2$. For simplicity, in the figure, we abbreviate the states and labels of the transitions. In particular, the states are:

$$p_M^0, \quad p_M^1 = (s_0, q_0), \quad p_M^2 = (s_1, q_1, [Z'Z_D]), \quad p_M^3 = (s_1, 0), \quad p_M^4 = (s_2, 0).$$

The labels in edges of this PDA are given in detail as follows:

- $\alpha_1 \equiv \epsilon, (\$_1, [\$_2]) \to (Z_C, [Z_D\$_2])(\$_1, [\$_2])$

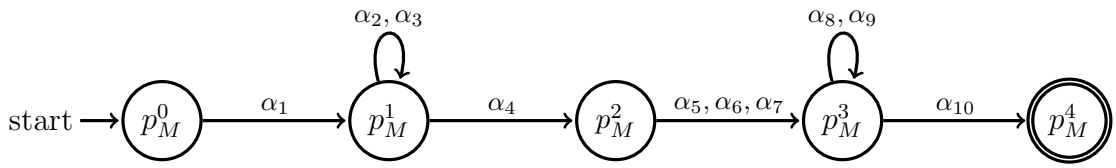

Figure 3.3: The simulating PDA $M(C, D, 1)$

- $\alpha_2 \equiv a, (Z_C, [Z_D\$_2]) \to (Z, [Z_D\$_2])(Z_C, [\$_2])$
- $\alpha_3 \equiv a, (Z, [Z_D\$_2]) \to (Z, [Z_D\$_2])(Z, [\$_2])$
- $\alpha_4 \equiv b, (Z, [Z_D\$_2]) \to \epsilon$
- $\alpha_5 \equiv \epsilon, (Z, [Z_D\$_2]) \to (Z, [\epsilon])$
- $\alpha_6 \equiv \epsilon, (Z, [\$_2]) \to (Z, [\epsilon])$
- $\alpha_7 \equiv \epsilon, (Z_C, [\$_2]) \to (Z_C, [\epsilon])$
- $\alpha_8 \equiv b, (Z, [v]) \to \epsilon$
- $\alpha_9 \equiv c, (Z_C, [v]) \to (Z_C, [v])$
- $\alpha_{10} \equiv c, (Z_C, [v]) \to \epsilon$.

The language of M is $L(M(C, D, 1)) = \{a^n b^n c^m \mid n \geq 1, m \geq 1\}$.

3.3 Soundness and Completeness

In this section, we show that the construction presented in the preceding section is sound and complete, i.e., $L(A) \subseteq L(B)$ if and only if $L(M(A, B, k+1)) = \varnothing$, where k was computed from A and B as in Lemma 3.1.

3.3.1 Soundness

Lemma 3.2. *$L(A) \nsubseteq L(B)$ implies $L(M(A, B, r)) \neq \varnothing$ for all $r \geq 1$.*

Proof. Let $w \in L(A) \backslash L(B)$. It is sufficient to show that $w \in L(M)$. Denote c_{in} as the initial configuration of M. Recall that A is normalized (by Lemma 2.4), there is a computation of A on every word. By the definition of transitions of M, there is a computation of M on w. There are three cases:

1. There are no computations of B on w, or there is a computation of B on w but after reading w, the stack of B is nonempty. By transitions of M, we have $c_{in} \xrightarrow{w} \langle (p, 0), \varepsilon \rangle$. Since $w \in L(A)$, $(p, 0) \in F_A \times \{0\}$. Thus, $w \in L(M)$.

2. There is a computation of B on w leading to a configuration (q, ε), where $q \notin F_B$. Because A accepts w, there is a computation of M on w leading to a configuration $\langle (p, q), \varepsilon \rangle$, $(p, q) \in F_A \times (Q_B \setminus F_B)$. Thus, $w \in L(M)$.

3. Where simulating w, the stacking fails. In this case, we have $c_{in} \xrightarrow{w} \langle (p, 0), \varepsilon \rangle$, $p \in F_A$.

□

3.3.2 Completeness

Lemma 3.3. *Let k be the number computed in Lemma 3.1. $L(M(A, B, k+1)) \neq \varnothing$ implies $L(A) \not\subseteq L(B)$.*

Proof. Let $w \in L(M(A, B, k+1))$. There is an accepting computation of $M(A, B, k+1)$ on w. We consider two cases of accepting configurations of M.

Case 1. If $c_{in} \xrightarrow{w} \langle (p, q), \varepsilon \rangle$ where $(p, q) \in F_A \times (Q_B \setminus F_B)$. In this case, there is a computation of B on w leading to the configuration (q, ε). Because $q \notin F_B$, we obtain $w \notin L(B)$. On the other hand, on reading w, A leads to the accepting configuration (p, ε), i.e., $w \in L(A)$. Thus, $w \in L(A) \setminus L(B)$.

Case 2. If $c_{in} \xrightarrow{w} \langle (p, 0), \varepsilon \rangle$ where $p \in F_A$. Consider two subcases. First, if B is blocked at some point on reading w. In this case, there is not a computation of B on w, i.e., $w \notin L(B)$. On the other hand, on reading w, A leads to the configuration (p, ε), $p \in F_A$. Hence $w \in L(A)$. Since B is deterministic, we have $w \in L(A) \setminus L(B)$. The proof is completed. Second, if stacking fails at some point on simulating w. In this case, to prove $L(A) \not\subseteq L(B)$, we assume on the contrary that $L(A) \subseteq L(B)$. We will show a contradiction. Since the stacking fails on reading w, we suppose that $w = w_1 w_2$ such that, after reading w_1 the first time, stacking fail occurs and M is in the control $(p_1, 0)$ with the stack content $(X_t, [v_t])(X_{t-1}, [v_{t-1}]) \cdots (X_1, [v_1])(\$_1, [\$_2])$. Whereas after reading w_1, A is in the configuration $c_2 = (p_1, X_t ... X_1)$ (c_2 is live) and B is in the configuration $c_2' = (p_2, f(v_t v_{t-1} ... v_1))$. There are two subcases which lead to the stacking failure: either $[v_t] = [\varepsilon]$ or $|f(v_t)| \geq 2r + 1$.

- **If $[v_t] = [\varepsilon]$:** we have $t \geq 2$ and $f(v_t ... v_1) \neq \epsilon$ (because, if $t = 1$ then $[v_t]$ must be $[\$_2]$, and if $f(v_t ... v_1) = \epsilon$ then the stack of B is empty and $[v_t] = [\$_2]$). Since $f(v_t ... v_1) \neq \epsilon$ there is at least one $f(v_i) \neq \epsilon$. Select the "nearest" v_j such that $f(v_j) \neq \epsilon$ and $f(v_i) = \epsilon$ for $j + 1 \leq i \leq t$. Consider the time when the level $j + 1$ of the stack is opened. Since $f(v_j) \neq \epsilon$, this means that the rule II(3) or II(4) was used, and the "new" top segment at that time was v'_{j+1} with $|v'_{j+1}| = r$ or $|v'_{j+1}| = r+1$. Since that time, M has not read below level $j+1$. Thus, we have $w_1 = w' w''$, and after reading

w', M is in the configuration $\langle (p'_1, p'_2), (X'_{j+1}, [v'_{j+1}])(X_j, [v_j]) \cdots (\$_1, [\$_2]) \rangle$ encoding the configurations $c_1 = (p'_1, X'_{j+1}X_j...X_1)$ of A, and $c'_1 = (p'_2, f(v'_{j+1}v_j...v_1))$ of B such that: $c_0 \xrightarrow{w'} c_1$ and $c'_0 \xrightarrow{w'} c'_1$ (c_0 and c'_0 are the initial configurations of A and B, respectively). Because $L(A) \subseteq L(B)$ (by assumption) and B is deterministic, $L(c_1) \subseteq L(c'_1)$. On the other hand, we have $c_1 \uparrow (w'')c_2$ and $c'_1 \xrightarrow{w''} c'_2$. Note that these conditions satisfy assumptions of the Key lemma (Lemma 3.1). However, we have $|c'_1| - |c'_2| = |v'_{j+1}| \geq r = k + 1 > k$. This contradicts Lemma 3.1. Hence, the assumption $L(A) \subseteq L(B)$ is wrong. Thus, $L(A) \nsubseteq L(B)$.

- **If $|f(v_t)| \geq 2r + 1$:** Consider the time when the level $t - 1$ of the stack is opened. At that point, one of rules II(1), II(2), II(3), or II(4) was used and the "new" top segment was v'_{t-1} with $|v'_{t-1}| \leq r + 1$. Since that time, M has not read below level $t - 1$. Thus, we have $w_1 = w'w''$, and after reading w', M is in the configuration $\langle (p'_1, p'_2), (X'_{t-1}, [v'_{t-1}]) \cdots (\$_1, [\$_2]) \rangle$ encoding the configurations $c_1 = (p'_1, X'_{t-1}X_{t-2}...X_1)$ of A, and $c'_1 = (p'_2, f(v'_{t-1}v_{t-2}...v_1))$ of B such that: $c_0 \xrightarrow{w'} c_1$ and $c'_0 \xrightarrow{w'} c'_1$. Because $L(A) \subseteq L(B)$ (by assumption) and B is deterministic, $L(c_1) \subseteq L(c'_1)$. In addition, we have $c_1 \uparrow (w'')c_2$ and $c'_1 \xrightarrow{w''} c'_2$. Note that these conditions satisfy assumptions of Lemma 3.1. Now, we can compute:

$$\begin{cases} |c_1| - |c_2| = |X_{t-1}| - |X'_{t-1}| = 1 - 1 = 0 \\ |c'_2| - |c'_1| = |v_t| - |v'_{t-1}| \geq k + 1 > k. \end{cases}$$

  This contradicts Lemma 3.1. Hence $L(A) \nsubseteq L(B)$.

In both cases, we have, if $L(M(A, B, k + 1)) \neq \varnothing$, then $L(A) \nsubseteq L(B)$. The lemma is proved.

□

From Lemmas 3.2 and 3.3, we obtain:

Lemma 3.4. *$L(A) \subseteq L(B)$ if and only if $L(M(A, B, k + 1)) = \varnothing$.*

3.3.3 The Inclusion Problem

Let $A = (Q, \Sigma, \Gamma, Z_0, \Delta, q_0, F)$ be a PDA. The *size* $|A|$ of a PDA A is defined as $|Q| + |\Sigma| + |\Gamma| + \{|pXq\alpha| \mid (p, X) \xrightarrow{a} (q, \alpha) \in \Delta\}$. We obtain the same complexity class as that of the original construction.

Theorem 3.5. *The inclusion problem $L(A) \subseteq L(B)$, where A is a PDA and B is an SPDA, is decidable. Furthermore, the decision procedure has time complexity bounded by $2^{2^{p(h)}}$, where $p(h)$ is a polynomial time in the size of both automata, $h = |A| + |B|$.*

Proof. The decidability follows from Lemma 3.4. We now approximate the size of M. Recall that the emptiness problem can be decided in $O(n^3)$ for any PDA of size n. The stack of M is bounded by $|\Gamma_A| \cdot |\Gamma_B|^{2k+2}$, where k is the number given in Lemma 3.1. The maximum number of control states of M is $|Q_A| \cdot |Q_B| \cdot |\Gamma_B|^{2k+1}$. The number of transitions of M is bounded by $|Q_A|^2 \cdot |Q_B|^2 \cdot |\Sigma| \cdot |\Gamma_A|^3 \cdot |\Gamma_B|^{6k+6}$. Recall that $s = |Q_A| + |Q_B|$, $g = |\Gamma_A| + |\Gamma_B|$. The size of M is bounded by $|M| \leq s^4 g^{6k+6}$. Lemma 3.1 expresses that $k = (d+1)(s+4)g(g+1)^{2(1+2s^2g^2(s^2+4))} + 2d$, where d is the delay of B. Define $h = s + g$, and we see that $k \leq h^{c_1 h^{c_2}}$, for some constants c_1 and c_2. Thus, for some constant c_3, the size of M is bounded by $h^{c_3 h^{c_1 h^{c_2}}}$. Thus, the time complexity of the construction is bounded by $2^{2^{p(h)}}$ for a polynomial $p(h)$. □

3.4 Related Work

It is known that the class of PDAs is not closed under intersection and complementation. Thus, decision problems like inclusion and equivalence are undecidable for PDAs. This situation only slightly improves when considering the subclass of DPDAs (see [9, 72] for an overview). Especially, the equivalence problem is decidable for DPDAs. Despite intensive work throughout the late 1960s and 1970s, this problem remained unsolved until 1997 when Sénizergues announced a positive solution [61]. However, the notation of pushdown configurations seem not rich enough to sustain a proof. Deeper algebraic structure needs to exposed. Stirling [63] proposed another proof for this problem based on a mixture of techniques developed in concurrency theory and language theory.

SPDAs were proposed by S. Greibach and E. Friedman in [34, 32]. It is shown that the acceptance condition of SPDAs does strictly affect decision problems. More precisely, for SPDAs accepting by final control state, the inclusion problem is undecidable [32]. If we consider SPDAs accepting by a final state and an empty stack, it is shown that the language inclusion problem $L(A) \subseteq L(B)$ is decidable for A is an arbitrary PDA, and B is an SPDA [34]. As far as we know, the class of SPDAs is the largest class which enjoys decidability for this inclusion problem. The main results of the inclusion problem $L(A) \subseteq L(B)$, in which A is an arbitrary PDA and B is an SPDA, can be summarized as follows:

- This inclusion problem is undecidable if B accepting by the final state [32].
- This inclusion problem is decidable if B accepting by the final state and the empty stack [34].

Some works related to the inclusion problem of context-free languages have been published recently by Y. Minamide and A. Tozawa [54, 70]. In [54], Minamide and Tozawa

developed two algorithms for deciding the inclusion $L(G_1) \subseteq L(G_2)$ where G_1 is a context-free grammar and G_2 is either an *XML-grammar* or a *regular hedge grammar*. Tozawa and Minamide [70] proved further that these algorithms for XML-grammars and regular hedge grammars are PTIME and 2EXPTIME, respectively. These algorithms were incorporated into the PHP string analyzer and validated several publicly available PHP programs against XHTML DTD. The languages of XML-grammars or regular hedge grammars are subclasses of generalized parenthesis languages. On the other hand, the class of languages of SPDAs contains the class of generalized parenthesis languages [34]. Thus, SPDAs are more expressive than XML-grammars and regular hedge grammars.

Chapter 4

Visibly Pushdown Automata and Its Extensions

Visibly pushdown automata [5] are pushdown automata whose stack behavior (i.e. whether to execute a push, a pop, or no stack operation) is completely determined by the input symbol according to a fixed partition of the input alphabet. As shown in [5, 7], this class of visibly pushdown automata enjoys many good properties similar to those of the class of regular languages, the main reason for this being that each nondeterministic VPA can be transformed into an equivalent deterministic one. As each nondeterministic VPA can be determinized, all problems that concern the accepted language such as universality and inclusion problems are decidable for VPAs. Visibly pushdown automata have turned out to be useful in various context, e.g., as specification formalism for verification for pushdown models [5], and as automaton model for processing XML streams [7, 50, 46].

Visibly pushdown automata with multiple stacks have been considered recently and independently by Carotenuto et al. [17] and Torre et al. [69]. The purpose of these papers is to exploit the notion of visibility further to obtain even richer classes of languages while preserving important closure properties and decidability of verification-related problems such as emptiness and inclusion. The emptiness problems for these extensions, however, are undecidable.

To retain the nice closure properties, Torre et al. [69] consider a subclass, named *k-MVPAs*, of multiple-stack VPAs with restrictions that: *an input word can be divided into at most k phases such that, in every phase, pop actions can occur in at most one stack.* Then, the emptiness problem becomes decidable. Although k-MVPAs are not determinizable, they are closed under Boolean operations [69].

In [17], the approach to gain decidability and nice closure properties is to exclude simultaneous pop operations by introducing an ordering constraint on stacks. This restricted subclass of multi-stack VPAs is called *ordered* n-VPAs (n-OVPAs). For instance, in 2-OVPAs, a pop action on the second stack occurs only after the first stack becomes

empty. Then, the emptiness problem turns out to be decidable in polynomial time. They also claimed determinizability of 2-OVPAs (2-VPAs).

Here we show that the determinization for extensions of VPAs are difficult:

1. First, we show a detailed counter example to refute the claim about the determinizability of 2-OVPAs (consequently, 2-VPAs).

2. Second, we introduce the class of *visibly stack automata* (VSAs) as an extension of VPAs by combining ideas of visibility and stack automata [35]. We also give a counter example to show that VSAs are not determinizable.

4.1 Visibly Pushdown Automata

4.1.1 Definition of Visibly Pushdown Automata

We borrow most of terminology and definitions from [5, 7]. Let Σ be the finite input alphabet, and let $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i$ be a partition of Σ. The intuition behind the partition is: Σ_c is the finite set of *call* (push) symbols, Σ_r is the finite set of *return* (pop) symbols, and Σ_i is the finite set of *internal* symbols.

Definition 4.1. *A* visibly pushdown automaton *(VPA) M over Σ is a tuple $(Q, Q_0, \Gamma, \Delta, F)$ where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, Γ is a finite stack alphabet with a special symbol $\perp$ (representing the* bottom-of-stack*), and $\Delta = \Delta_c \cup \Delta_r \cup \Delta_i$ is the transition function, where $\Delta_c : Q \times \Sigma_c \to Q \times (\Gamma \setminus \{\perp\})$, $\Delta_r : Q \times \Sigma_r \times \Gamma \to Q$, and $\Delta_i : Q \times \Sigma_i \to Q$.*

If $\Delta_c(q, c) = (q', \gamma)$, where $c \in \Sigma_c$ and $\gamma \neq \perp$, there is a *push-transition* from q on input c where on reading c, γ is pushed onto the stack and the control changes from state q to q'; we denote such a transition by $q \xrightarrow{c/+\gamma} q'$. Similarly, if $\Delta_r(q, r, \gamma) = q'$, there is a *pop-transition* from q on input r where γ is read from the top of the stack and popped (if the top of the stack is $\perp$, then it is read but not popped), and the control changes from q to q'; we denote such a transition $q \xrightarrow{r/-\gamma} q'$. If $\Delta_i(q, i) = q'$, there is an *internal-transition* from q on input i where on reading i, the state changes from q to q'; we denote such a transition by $q \xrightarrow{i} q'$. Note that there are no stack operations on internal transitions.

We write St for the set of *stacks* $\{w\perp \mid w \in (\Gamma \setminus \{\perp\})^*\}$. A *configuration* is a pair (q, σ) of $q \in Q$ and $\sigma \in St$. The transition function of a VPA can be used to define how the configuration of the machine changes in a single step: we say $(q, \sigma) \xrightarrow{a} (q', \sigma')$ if one of the following conditions holds:

- If $a \in \Sigma_c$ then there exists $\gamma \in \Gamma$ such that $q \xrightarrow{a/+\gamma} q'$ and $\sigma' = \gamma \cdot \sigma$

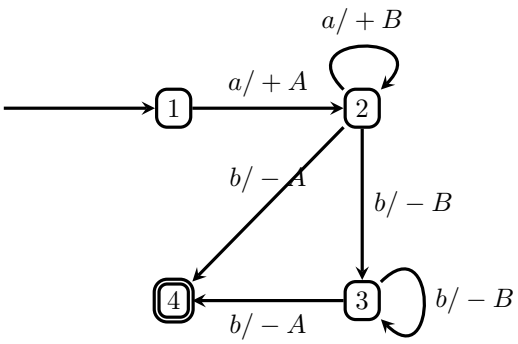

Figure 4.1: VPA M

- If $a \in \Sigma_r$, then there exists $\gamma \in \Gamma$ such that $q \xrightarrow{a/-\gamma} q'$ and either $\sigma = \gamma \cdot \sigma'$, or $\gamma = \perp$ and $\sigma = \sigma' = \perp$
- If $a \in \Sigma_i$, then $q \xrightarrow{a} q'$ and $\sigma = \sigma'$.

Remark 4.1. *In a VPA, it is worth to note that a call (push) transition is insensitive to the top of the stack, and a return (pop) transition is allowed even when the stack is empty (i.e., the top of the stack is the bottom $\perp$). The former requirement is to ensure that a* well-matched *word (Definition 4.3) is stable under concatenation (for determinization), and the latter requirement is needed for completion (i.e., each input symbol at each state has a transition).*

A ((q_0, w_0)-)*run* on a word $u = a_1 \cdots a_n$ is a sequence of configurations $(q_0, w_0) \xrightarrow{a_1} (q_1, w_1) \cdots \xrightarrow{a_n} (q_n, w_n)$, and is denoted by $(q_0, w_0) \xrightarrow{u} (q_n, w_n)$. A word u is accepted by M if there is a run $(q_0, w_0) \xrightarrow{u} (q_n, w_n)$ with $q_0 \in Q_0$, $w_0 = \perp$, and $q_n \in Q_F$. The language $L(M)$ is the set of words accepted by M. The language $L \subseteq \Sigma^*$ is a *visibly pushdown language* (VPL) if there exists a VPA M with $L = L(M)$.

Example 4.1. Consider the VPA M in Figure 4.2, where a is a push and b is a pop. It is easy to see that M recognizes the context-free language $L(M) = \{a^n b^n \mid n \geq 1\}$ with $\Sigma_c = \{a\}$ and $\Sigma_r = \{b\}$.

Remark 4.2. The class of VPL is a super class of parenthesis languages and Dyck languages. On the other hand, let consider the language $L' = \{a^n b a^n \mid n \geq 1\}$. This language can be recognized by a deterministic PDA N in which while reading the input symbol a, N may push into or pop from the stack. However, this language L' cannot be accepted by any VPA. Therefore, the class of VPL is a proper subclass of deterministic context-free languages.

Definition 4.2. *A VPA M is* deterministic *if $|Q_0| = 1$ and for every $q \in Q$:*

- *for every* $a \in \Sigma_i$*, there is at most one transition of the kind* $q \xrightarrow{a} q' \in \Delta_i$*,*
- *for every* $a \in \Sigma_c$*, there is at most one transition of the form* $q \xrightarrow{a/+\gamma} q' \in \Delta_c$*, and*
- *for every* $a \in \Sigma_r, \gamma \in \Gamma$*, there is at most one transition of the form* $q \xrightarrow{a/-\gamma} q' \in \Delta_r$*.*

Remark 4.3. *A VPA is* height deterministic *[45], i.e., the height of the stack is uniquely determined by an input word. Further, since the partition* $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i$ *is universal on VPAs, for an input word* u*, they are* synchronous*, i.e., they have the same height of the stack at the destination of each run of* u*.*

4.1.2 Determinization

This section presents an alternative proof for determinizability of VPA based on congruence relations.

Definition 4.3. *Let* $u \in \Sigma^*$*.*

- u *is* return-matched*, if for each prefix* u' *of* u*, the number of return symbols (*Σ_r*) in* u' *is at most the number of call symbols (*Σ_c*) in* u'*.*
- u *is* call-matched*, if for each suffix* u' *of* u*, the number of call symbols (*Σ_c*) in* u' *is at most the number of return symbols (*Σ_r*) in* u'*.*
- u *is* well-matched *if* u *is return-matched and call-matched.*

The set of all call-, return-, or well-matched words is denoted by $MC(\Sigma)$*,* $MR(\Sigma)$*, or* $WM(\Sigma)$ *respectively.*

Lemma 4.1. *For each* $u \in \Sigma^*$*, there exists a unique decomposition* $u = vc_1v_1 \cdots c_nv_n$ *with* $v \in MC(\Sigma)$*,* $c_i \in \Sigma_c$*, and* $v_i \in WM(\Sigma)$ *for all* $1 \leq i \leq k$*. This decomposition is called well-matched decomposition. When* $n \geq 1$*, the string* v_n *is called the well-matched suffix of* u *and is denoted by* $\mathsf{wms}(u)$*.*

Definition 4.4. *Let* $L \subseteq \Sigma^*$*. We define the relations* $\approx_L$ *on* $WM(\Sigma)$*,* $\equiv^L$ *on* Σ^**, and* $\sim_0^L$ *on* $MC(\Sigma)$ *as follows:*

- $u_1 \approx_L u_2$ *if* $vu_1v' \in L \Leftrightarrow vu_2v' \in L$ *for each* $v, v' \in \Sigma^*$*.*
- $u_1 \equiv^L u_2$ *if* $u_1v \in L \Leftrightarrow u_2v \in L$ *for each* $v \in MR(\Sigma)$*.*
- $u_1 \sim_0^L u_2$ *if* $u_1v \in L \Leftrightarrow u_2v \in L$ *for each* $v \in \Sigma^*$*.*

Let $\approx$ *be a congruence on* $WM(\Sigma)$*,* $\equiv$ *a right congruence on* Σ^**, and* $\sim_0$ *a right congruence on* $MC(\Sigma)$*. We say that* $(\approx, \equiv, \sim_0)$ *is compatible with* L *if they are refinements of the corresponding (right) congruence relations.*

Theorem 4.2 ([7, Theorems 1 and 2])**.** *A language L over the alphabet Σ is a VPL if and only if,*

- $\approx_L$ *is a finite congruence on* $WM(\Sigma)$*,*
- $\equiv^L$ *is a finite right congruence on* Σ^**, and*
- $\sim_0^L$ *is a finite right congruence on* $MC(\Sigma)$*.*

The following definition is a slightly refined version of determinization in[5, Theorem 2].

Definition 4.5. *Let $M = (Q, Q_0, \Gamma, \Delta, F)$ be a VPA, and $(\approx, \equiv, \sim_0)$ a tuple compatible with L. Suppose* `void` *is a fresh symbol. Let $u = vc_1v_1 \cdots c_nv_n$ with $v \in MC(\Sigma)$, $c_i \in \Sigma_c$, and $v_i \in WM(\Sigma)$ be the well-matched decomposition of $u \in \Sigma^*$. We define*

$$[\![u]\!] = \begin{cases} (\texttt{void}, [u]_{\sim_0}) & \text{if } n = 0 \\ ([v_n]_{\approx}, [u]_{\equiv}) & \text{if } n > 0 \end{cases}$$

The determinization $M^d = (Q^d, Q_0^d, \Gamma^d, \Delta^d, F^d)$ of M is defined as the VPA such that $Q_0^d = \{[\![\epsilon]\!]\}$, $Q_0^d \cup \Delta^d(Q^d) \subseteq Q^d$, $\Gamma^d = Q^d \cup \{\bot\}$, $F^d = \{[\![u]\!] \in Q^d \mid u \in L(M)\}$, and for every $u \in Q^d$, $a, a' \in \Sigma_c$, $b \in \Sigma_r$, $c \in \Sigma_i$, and $u' \in \Sigma^*$:

$$[\![u]\!] \xrightarrow{c} [\![uc]\!] \in \Delta^d \qquad [\![u]\!] \xrightarrow{a/+[\![ua]\!]} [\![ua]\!] \in \Delta^d$$

$$[\![u]\!] \xrightarrow{b/-\bot} [\![ub]\!] \in \Delta^d \quad \text{if } n = 0 \qquad [\![u]\!] \xrightarrow{b/-[\![u'a']\!]} [\![u'a'v_nb]\!] \in \Delta^d \quad \text{if } n > 0$$

where $vc_1v_1 \cdots c_nv_n$ is the well-matched decomposition of u.

Note that the original stack alphabet of M is ignored, and during transitions, the last element (before $\bot$) in the stack remains in $\{(\texttt{void}, [u]_{\equiv} \mid u \in MC(\Sigma)\}$, and `void` does not appear elsewhere in the stack. Based on the construction in Definition 4.5, we have:

Theorem 4.3 ([5, Theorem 2])**.** *Let M be a VPA. The VPA M^d is actually deterministic and $L(M) = L(M^d)$. Moreover, if M has n states, one can construct M^d with at most $O(2^{n^2})$ states and with stack alphabet of size $O(2^{n^2})$.*

Remark 4.4. In the determinization, our version is smaller than the original one.

Example 4.2. Consider the VPA M in Figure 4.2 where $\Sigma_c = \{a\}, \Sigma_r = \{b\}, \Sigma_i = \{c\}$. M^d is depicted as in Figure 4.3.

We introduce a practical candidate for $(\approx, \equiv, \sim_0)$ that is obtained from the definition of VPA.

Definition 4.6. *For $u \in \Sigma^*$,*

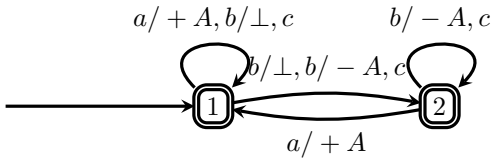

Figure 4.2: VPA M

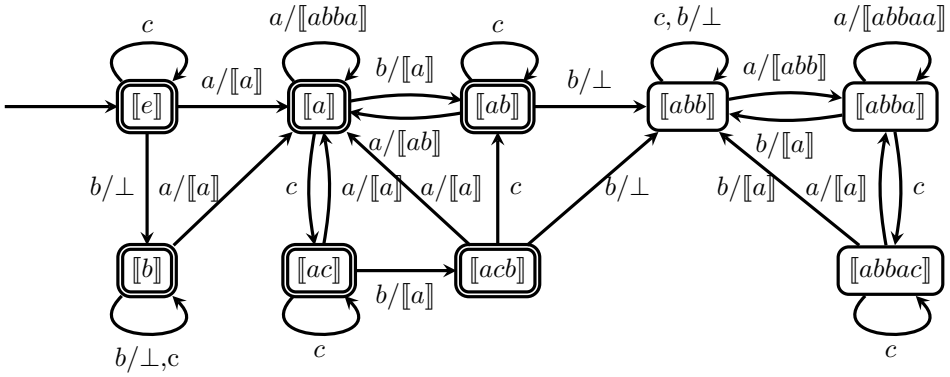

Figure 4.3: Determinized VPA M^d

- $\delta_\bot(u)$ *denotes the set of* $q \in Q$ *satisfying that there exist* $q' \in Q_0$ *and* $w \in St_\Gamma$ *with* $(q', \bot) \xrightarrow{u} (q, w)$. $\delta_\bot(u)$ *is called* reachables *(of* u*).*

- $\Delta(u)$ *denotes the set of* $(q, q') \in Q \times Q'$ *satisfying that there exist* $w, w' \in St_\Gamma$ *with* $(q, w) \xrightarrow{u} (q', w')$. *When* $u \in WM(\Sigma)$, $\Delta(u)$ *is called* summaries *(of* u*).*

Definition 4.7. *Let* $M = (Q, Q_0, \Gamma, \Delta, F)$ *be a VPA. We define relations* $\approx_M$, $\equiv^M$, $\sim_0^M$ *as follows.*

- *For* $u_1, u_2 \in WM(\Sigma)$, $u_1 \approx_M u_2$ *if* $q \xrightarrow{u_1} q' \Leftrightarrow q \xrightarrow{u_2} q'$ *for each* $q, q' \in Q$.

- *For* $u_1, u_2 \in \Sigma^*$, $u_1 \equiv^M u_2$ *if* $\delta_\bot(u_1) = \delta_\bot(u_2)$.

- *For* $u_1, u_2 \in MC(\Sigma)$, $u_1 \sim_0^M u_2$ *if* $\delta_\bot(u_1) = \delta_\bot(u_2)$.

Lemma 4.4. $(\approx_M, \equiv^M, \sim_0^M)$ *is compatible with* $L(M)$.

Note that $\sim_0^M$ coincides with $\equiv^M$ on $MC(\Sigma)$

4.1.3 Closure Properties and Decision Problems

Recall that PDAs are closed under union, but not under intersection and complementation. VPAs are however closed under all these operations.

Theorem 4.5 ([5, Theorem 1]). *The class of VPAs is closed under union, intersection, and complementation.*

Turning to the decidability of decision problems for VPAs, observe that since a VPA is a PDA, emptiness is decidable in time $O(n^3)$ where $n = |Q| + |\Delta|$ is the size of the VPA. Recall that the universality problem for VPAs is to check whether a given VPA M accepts all strings in Σ^*. The inclusion problem is to find whether, given two VPAs M_1 and M_2, $L(M_1) \subseteq L(M_2)$. Though both are undecidable for PDAs, they are decidable for VPAs:

Theorem 4.6 ([5, Theorem 3]). *The universality problem and the inclusion problem for VPAs are EXPTIME-complete.*

4.2 Language Extensions of VPA Are Difficult

4.2.1 2-Visibly Pushdown Automata

We give a brief description of 2-visibly pushdown automata. Readers are referred to [17] for more details.

Definition 4.8 ([17]). *A* 2-pushdown alphabet *is a pair of pushdown alphabets* $\overline{\Sigma} = \langle \overline{\Sigma}^0, \overline{\Sigma}^1 \rangle$, *where* $\overline{\Sigma}^0 = \Sigma_c^0 \cup \Sigma_r^0 \cup \Sigma_i^0$ *and* $\overline{\Sigma}^1 = \Sigma_c^1 \cup \Sigma_r^1 \cup \Sigma_i^1$ *are possibly different partitions of the same input alphabet* Σ. *The intuition behind a 2-pushdown alphabet is explained as follows:*

1. $\overline{\Sigma}^0$ *and* $\overline{\Sigma}^1$ *drive the actions over the first stack and over the second stack, respectively.*
2. *Symbols in* $\overline{\Sigma}$ *belonging to push, pop or internal actions of both* $\overline{\Sigma}^0$ *and* $\overline{\Sigma}^1$ *are simply denoted by* Σ_c, Σ_r, *and* Σ_i, *respectively.*
3. *The input symbols that drive a push action on the first (resp., second) stack and a pop on the second (resp., first) stack are called* synchronized communication *symbols, and are formally denoted as* $\Sigma_{s_1} = \Sigma_c^0 \cap \Sigma_r^1$ *(resp.,* $\Sigma_{s_0} = \Sigma_r^0 \cap \Sigma_c^1$*).*
4. *Let* Σ_{c_i} *(resp.,* Σ_{r_i}*) denote the set of push (resp., pop) symbols for the stack* i *and internal for the other stack, with* $i = 0, 1$.

Definition 4.9 ([17]). *A (nondeterministic)* 2-visibly pushdown automaton *(*2-VPA*) on finite words over a 2-pushdown alphabet* $\overline{\Sigma}$ *is a tuple* $M = \langle Q, Q_0, \Gamma, \bot, \delta, Q_f \rangle$, *where*

1. Q, Q_0, *and* Q_f *are finite sets of states, initial states, and final states, respectively.*

2. Γ *is a finite set of stack symbols,* $\perp \notin \Gamma$ *is the stack bottom symbol (with* $\Gamma_\perp$ *used to denote* $\Gamma \cup \{\perp\}$*).*

3. δ *is the transition relations defined as the union of the following sets, for* $i \in \{0,1\}$*:*

   - $\delta_{c_i} \subseteq (Q \times \Sigma_{c_i} \times Q \times \Gamma)$, $\delta_{r_i} \subseteq (Q \times \Sigma_{r_i} \times \Gamma_\perp \times Q)$
   - $\delta_c \subseteq (Q \times \Sigma_c \times Q \times \Gamma \times \Gamma)$, $\delta_r \subseteq (Q \times \Sigma_r \times \Gamma_\perp \times \Gamma_\perp \times Q)$,
   - $\delta_{s_i} \subseteq (Q \times \Sigma_{s_i} \times \Gamma_\perp \times Q \times \Gamma)$, $\delta_i \subseteq Q \times \Sigma_i \times Q$.

A *configuration* of a 2-VPA M is a triple (q, σ^0, σ^1) where $q \in Q$ and $\sigma^0, \sigma^1 \in \Gamma^*.\{\perp\}$. For an input word $w = a_1 \cdots a_k \in \Sigma^*$, a run of M on w is a sequence $\rho = (q_0, \sigma_0^0, \sigma_0^1) \cdots (q_k, \sigma_k^0, \sigma_k^1)$ where $q_0 \in Q_0$, $\sigma_0^0 = \sigma_0^1 = \perp$, and for all $i \in \{0, \cdots k-1\}$, there are $j, j' \in \{0,1\}, j \neq j'$, such that one of the following conditions holds:

- **[Push]:** $(q_i, a_i, q_{i+1}, \gamma) \in \delta_{c_j}$, then $\sigma_{i+1}^j = \gamma \sigma_i^j$ and $\sigma_{i+1}^{j'} = \sigma_i^{j'}$.
- **[2Push]:** $(q_i, a_i, q_{i+1}, \gamma, \gamma') \in \delta_c$, then $\sigma_{i+1}^j = \gamma \sigma_i^j$ and $\sigma_{i+1}^{j'} = \gamma' \sigma_i^{j'}$.
- **[Pop]:** $(q_i, a_i, \gamma, q_{i+1}) \in \delta_{r_j}$, then either $\gamma = \sigma_i^j = \sigma_{i+1}^j = \perp$, or $\gamma \neq \perp$ and $\sigma_i^j = \gamma \sigma_{i+1}^j$. In both cases $\sigma_{i+1}^{j'} = \sigma_i^{j'}$.
- **[2Pop]:** $(q_i, a_i, \gamma_0, \gamma_1, q_{i+1}) \in \delta_r$, then, for $j \in \{0,1\}$, either $\gamma_j = \sigma_i^j = \sigma_{i+1}^j = \perp$, or $\gamma_j \neq \perp$ and $\sigma_i^j = \sigma_{i+1}^j . \gamma_j$.
- **[Internal]:** $(q_i, a_i, q_{i+1}) \in \delta_i$ then $\sigma_{i+1}^j = \sigma_i^j$.
- **[Sync]:** $(q_i, a_i, \gamma, q_{i+1}, \bar{\gamma}) \in \delta_{s_j}$ then either $\gamma = \sigma_i^j = \sigma_{i+1}^j = \perp$, or $\gamma \neq \perp$ and $\sigma_i^j = \gamma \sigma_{i+1}^j$. In both cases $\sigma_{i+1}^{j'} = \bar{\gamma} \sigma_i^{j'}$.

Definition 4.10. *A run* ρ *is* accepting *if it ends in a final state. A word* w *is* accepting *if there is an accepting run* ρ *of* M *on* w*. The language accepted by* M*, denoted by* $L(M)$*, is the set of all words accepted by* M*. Let* 2-VPL *denote the set of languages recognized by 2-VPAs.*

Remark 4.5. As in the VPA case, 2-VPA input symbols are partitioned in subclasses, each of them triggers a transition belonging to a specific class, i.e., push/pop/local transition, which also selects the operating stacks, the first or the second or both. Moreover, visibility in 2-VPA affects the transfer of information from one stack to the other. Therefore, 2-VPA becomes strictly more expressive than VPA and they also accept some context-sensitive languages (for instance, $\{a^n b^n c^n \mid n \geq 1\}$ [17]) that are not context-free. Unfortunately, this extension does not preserve decidability of the emptiness problem as it can be proved by a reduction from the halting problem over two counter machines (see [17] for details).

Remark 4.6. The class $2-VPL$ is a proper subclass of context-sensitive languages (CSL). However, this class $2-VPL$ is incomparable with the class of context-free languages (CFL), i.e., $2-VPL \nsubseteq CFL$ and $CFL \nsubseteq 2-VPL$. In particular, the non-context-free language $\{a^n b^n c^n \mid n \geq 1\}$ [17] can be accepted by a 2-VPA. On the other hand, the context-free language $\{a^n b a^n \mid n \geq 1\}$ cannot be accepted by any 2-VPA because the symbol a here is used for two kinds of operations: push and pop.

Theorem 4.7 ([17, Theorem 2])**.** *The emptiness problem for 2-VPAs is undecidable.*

In automata-theoretic approach, decidability of the emptiness problem is crucial. Thus, the following restriction on 2-VPA was used to retain the decidability of the emptiness problem.

Definition 4.11. *An* ordered 2-VPA *(*2-OVPA*) M is a 2-VPA in which a pop action on the second stack can occur only if the first stack is empty.*

Theorem 4.8 ([17, Corollary 1])**.** *Given a 2-OVPA M, deciding whether $L(M) \neq \varnothing$ is solvable in $O(n^3)$, where n is the number of states in M.*

Definition 4.12. *A 2-VPA (resp., 2-OVPA) M is deterministic if $|Q_0| = 1$, and for every $q \in Q, a \in \Sigma$, and $\gamma_r, \gamma'_r \in \Gamma_\perp$, there is at most one transition of the form (q, a, q'), (q, a, q', γ), $(q, a, q', \gamma, \gamma')$, $(q, a, \gamma_r, \gamma'_r, q')$, or $(q, a, \gamma_r, q', \gamma')$ belonging to δ.*

Now we are ready to discuss the main question of this section: *How about the determinizability of 2-VPA?*. In [17], the author showed that the class of 2-VPA (and 2-OVPA) is determinizable ([17, Theorem 6]). Their technique is to extend the determinization procedure of Alur and Madhusudan [5] for the cases of 2-VPAs and 2-OVPAs. In the following, we show a detailed counter example to refute their proof. This example is inspired from the example in [69]. Indeed, as Counter Example 1 shows, the determinization procedure in [17] does not properly work for 2-VPAs (2-OVPAs). The reason for failure is that actions on 2 stacks may interleave each other. Thus, two sets of "summary edges" do not correctly maintain reachable states.

Counter-example 1. *Let Σ be a 2-pushdown alphabet in which: $\Sigma_c^0 = \{a\}$, $\Sigma_r^0 = \{c, d, u\}$, $\Sigma_i^0 = \emptyset$, $\Sigma_c^1 = \{b\}$, $\Sigma_r^1 = \{x, y, v\}$, $\Sigma_i^1 = \emptyset$.*

Consider a nondeterministic 2-VPA $M = \langle Q, Q_0, \Gamma, \perp, \delta, Q_f \rangle$, with $Q = \{q_1, \cdots, q_{10}\}$, $Q_0 = \{q_1\}$, $Q_f = \{q_{10}\}$, and $\Gamma = \{A, B\}$. The 2-VPA M is depicted in Figure 4.4. Moreover, it is easy to note that M is a 2-OVPA.

We describe states of M as nodes of a graph. We adopt the following conventions to represent edges: for instance, a push into the first stack (q_i, a, q_j, A) is labeled as $a/+^0 A$, a pop from the second stack (q_i, x, B, q_j) is labeled as $x/-^1 B$. Note that, M accepts the language $L_1 = \{(ab)^i c^j d^{i-j} u x^j y^{i-j} v \mid i \geq 1, 1 \leq j \leq i\}$.

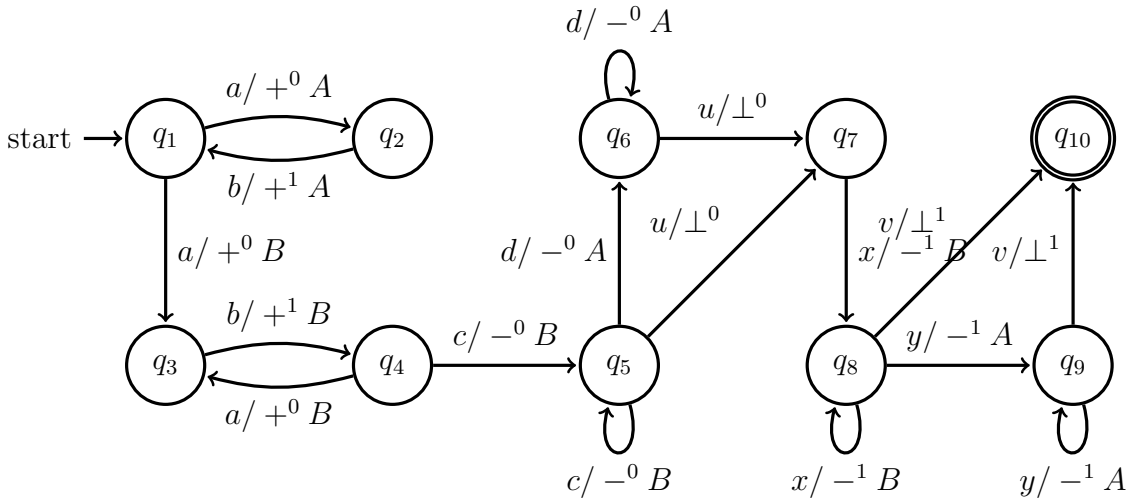

Figure 4.4: A nondeterministic 2-VPA accepting L_1

We prove that M cannot be determinized. Assume that M' is a deterministic 2-VPA accepting L_1. Suppose that after reading $(ab)^i$, M' is in a state, say s, with the same contents on two stacks. Select i larger than the number of states of M', and assume that $s \xrightarrow{c^t d^{i-t}} s_t$ for $0 \leq t \leq i$. Thus, there are some j and k $(0 \leq j < k \leq i)$ such that $s_j \equiv s_k$. On reading $c^t d^{i-t}$, M' pops symbols only from the first stack. Thus, M' reaches the same configuration on $(ab)^i c^j d^{i-j} u$ and on $(ab)^i c^k d^{i-k} u$. By appending the word $x^j y^{i-j} v$, $w = (ab)^i c^j d^{i-j} u x^j y^{i-j} v$ and $w' = (ab)^i c^k d^{i-k} u x^j y^{i-j} v$ lead M' to the same configuration. Since $L(M') = L_1$, M' accepts w and rejects w'. This is a contradiction.

□

4.2.2 Visibly Stack Automata

A *visibly input alphabet* is a finite set of input symbols $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i \cup \Sigma_u \cup \Sigma_d$ that comprises five disjoint finite alphabets, where Σ_c, Σ_r, Σ_i, Σ_u, and Σ_d are finite sets of *push*, *pop*, *internal*, *up*, and *down* actions, respectively. We formally define visibly stack automata as follows:

Definition 4.13. *A* visibly stack automaton *(*VSA*) on finite words over Σ is a tuple $M = \langle Q, Q_0, \Gamma, \uparrow, \perp, \top, \delta, Q_f \rangle$, where*

1. *Q, Q_0, and Q_f are finite sets of states, initial states, and final states, respectively.*
2. *Γ is a finite stack alphabet.*
3. *$\uparrow$, $\perp$, and $\top$ are special symbols not in Γ (where $\uparrow$ is the stack reading head, $\perp$ is the bottom-of-stack symbol, and $\top$ is the top-of-stack symbol). Denote $\Gamma_\perp = \Gamma \cup \{\perp\}$ and $\Gamma_{\perp\top} = \Gamma \cup \{\perp, \top\}$.*

4. $\delta = \delta_c \cup \delta_r \cup \delta_i \cup \delta_u \cup \delta_d$ *is the transition relation:*

   - $\delta_c \subseteq Q \times \Sigma_c \times Q \times \Gamma$ *are push-transitions*
   - $\delta_r \subseteq Q \times \Sigma_r \times \Gamma_\perp \times Q$ *are pop-transitions*
   - $\delta_i \subseteq Q \times \Sigma_i \times Q$ *are internal-transitions*
   - $\delta_d \subseteq Q \times \Sigma_d \times \Gamma_{\perp\top} \times Q$ *are down-transitions*
   - $\delta_u \subseteq Q \times \Sigma_u \times \Gamma_{\perp\top} \times Q$ *are up-transitions*

A stack is a nonempty finite sequence over $\Gamma \cup \{\uparrow\}$ starting at the bottom-of-stack symbol $\perp$, and containing exactly one occurrence of the symbol $\uparrow$. More formally, the set of all possible stacks is $St = \top\Gamma^* \uparrow \Gamma^*.\{\perp\} \cup \uparrow \top\Gamma^*.\{\perp\}$.

A configuration of M is a pair $c = (q, \sigma)$ where $q \in Q$, and $\sigma \in St$. For a word $\alpha = a_1 \cdots a_k \in \Sigma^*$, a run of M on α is a sequence of configurations $\rho = (q_0, \sigma_0) \cdots (q_k, \sigma_k)$, where each $q_0 \in Q_0$, $\sigma_0 = \top \uparrow \perp$, and for every $1 \leq i \leq k$, one of the following conditions holds:

- **[Push]:** If a_i is a push action, then $\sigma_{i-1} = \top \uparrow \gamma u \perp$, and for some $\gamma' \in \Gamma$: $(q_{i-1}, a_i, q_i, \gamma') \in \delta_c$ and $\sigma_i = \top \uparrow \gamma'\gamma u \perp$.
- **[Pop]:** If a_i is a pop action, then for some $\gamma \in \Gamma'$: $\sigma_{i-1} = \top \uparrow \gamma u \perp$, $(q_{i-1}, a_i, \gamma, q_i) \in \delta_r$ and $\sigma_i = \top \uparrow u \perp$.
- **[Internal]:** If a_i is an internal action, then $(q_{i-1}, a_i, q_i) \in \delta_i$ and $\sigma_i = \sigma_{i-1}$.
- **[Down]:** If a_i is a down action and $\sigma_{i-1} = \top u_2\gamma' \uparrow \gamma u_1 \perp$, then $(q_{i-1}, a_i, \gamma, q_i) \in \delta_d$ and $\sigma_i = \top u_2\gamma'\gamma \uparrow u_1 \perp$.
- **[Up]:** If a_i is an up action and $\sigma_{i-1} = \top u_2\gamma \uparrow u_1 \perp$, then $(q_{i-1}, a_i, \gamma, q_i) \in \delta_u$ and $\sigma_i = \top u_2 \uparrow \gamma u_1 \perp$.

A run $\rho = (q_0, \sigma_0) \cdots (q_k, \sigma_k)$ is *accepting* if the last state is a final state. A word $\alpha \in \Sigma^*$ is accepted by a VSA M if there is an accepting run of M on α. The language of M, $L(M)$, is the set of words accepted by M. A language of finite words $L \subseteq \Sigma^*$ is a *visibly stack language* if there is a VSA M over Σ such that $L(M) = L$.

Definition 4.14. *A visibly stack automaton* $M = \langle Q, \Sigma, Q_0, \Gamma, \uparrow, \perp, \top, \delta, Q_f \rangle$ *is said to be deterministic if* $|Q_0| = 1$ *and* $|\{(q, a, q') \in \delta_i\} \cup \{(q, a, q', \gamma') \in \delta_c\} \cup \{(q, a, \gamma, q') \in \delta_r\} \cup \{(q, a, \gamma, q') \in \delta_u\} \cup \{(q, a, \gamma, q') \in \delta_d\}| \leq 1$*, for every* $q \in Q$ *and* $a \in \Sigma$.

Since the emptiness problem for stack automata is decidable [39], the next corollary holds.

Corollary 4.9. *The emptiness problem for VSAs is decidable.*

Based on visibility, the class of VSAs is closed under union and intersection. However, as Counter Example 2 shows, determinization of VSAs fails.

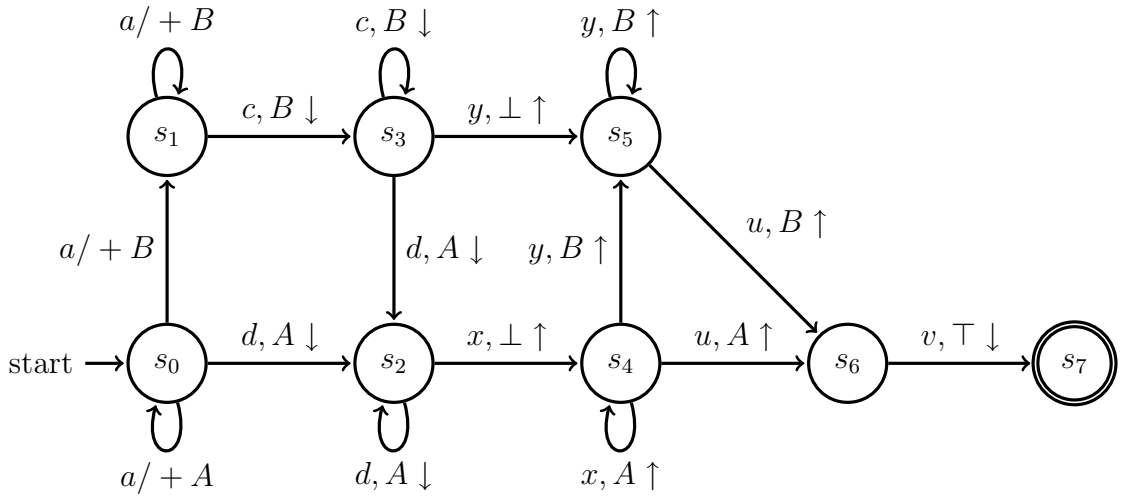

Figure 4.5: A nondeterministic VSA accepting L_2

Counter-example 2. *Let a be a push. Let c, d, v be downs, and x, y, u be ups. We show that no deterministic VSA can recognize the language $L_2 = \{a^i c^j d^{i-j} x^{i-j} y^j uv \mid 0 \leq j \leq i\}$.*

The VSA N in Figure 4.5 accepts L_2. On reading a, N pushes A onto the stack, and nondeterministically switches to a state s_1 where N pushes B onto the stack. Intuitively, this switch corresponds to the guess of what j is. On going down, N checks that the number of "c"s corresponds to the number of "B"s in the stack, and then N starts reading d and checks that the number of "d"s is equal to the number of "A"s. Then, from the bottom of the stack, the stack head goes up checking that the number of "x"s is $i - j$, and the number of "y"s is j.

We prove that N cannot be determinized. Assume that N' is a deterministic VSA accepting L_2. Suppose that after reading a^i, N' is in a state, say s. Select i larger than the number of states of N', and assume that $s \xrightarrow{c^t d^{i-t}} s_t$ for $0 \leq t \leq i$. Thus, there are some j and k $(0 \leq j < k \leq i)$ such that $s_j \equiv s_k$. N' reaches the same configuration on $a^i c^j d^{i-j}$ and on $a^i c^k d^{i-k}$. By appending the word $x^{i-j} y^j uv$, the words $w = a^i c^j d^{i-j} x^{i-j} y^j uv$ and $w' = a^i c^k d^{i-k} x^{i-j} y^j uv$ lead N' to the same configuration. Since $L(N') = L_2$, N' accepts w and rejects w'. This is a contradiction.

□

4.3 Related Work

The most related work is a recent paper of Nowotka and Srba [45] on *height-deterministic pushdown automata*. A PDA is height-deterministic if the stack height is determined solely by the input word; more precisely, a PDA M is height-deterministic if all runs of M on input $w \in \Sigma^*$ lead to configurations of the same stack height. Two height-deterministic PDAs M and N are *synchronized* if their stack heights coincide after reading the same input words. Given two height-deterministic PDAs M and N, the inclusion problem $L(M) \subseteq L(N)$ is decidable only if M and N are synchronized.

In [18], Caucal introduced an extension of Alur and Madhusudan's visibly pushdown languages, and proved that it forms a Boolean algebra. Caucal's class is defined with the help of a notion of synchronization. In particular, he introduced the synchronization of a pushdown automaton by a sequential transducer associating an integer to each input word. The visibly pushdown automata are the automata synchronized by a one-state transducer whose output labels are -1, 0, 1. For each transducer, one can decide whether a pushdown automaton is synchronized. The pushdown automata synchronized by a given transducer accept languages which form an effective Boolean algebra containing the regular languages and included in the deterministic real-time context-free languages.

This chapter refuted the claim about the determinizability of 2-OVPAs (2-VPAs). In addition, we have introduced the class of visibly stack automata (VSAs) and showed that this class of automata is not determinizable. Note that each 2-OVPA (resp., VSA) universally rejects certain words because of violation of the definition. For instance, $a^3b^3xycdyd$ (resp., $a^3cd^2a^5$), under the partition in Counter Example 1 (resp. Counter Example 2), is such a word. This means that 2-OVPAs (resp., VSAs) cannot be complemented. We, however, do not know whether the inclusion problems for 2-OVPAs (resp., VSAs)) are decidable or not, but we conjecture that they are not decidable.

Chapter 5

Checking Universality and Inclusion of Visibly Pushdown Automata

One of the most important properties of VPAs is that nondeterministic VPAs can be determinized, and determinization plays a key role in universality and inclusion checking [5]. However, the determinization is much harder to obtain than in the case of finite automata. In particular, for a nondeterministic VPA with n states, the determinization has a $O(2^{n^2})$ worst case complexity. To check universality for a nondeterministic VPA A over Σ (that is, to check if $L(A) = \Sigma^*$), the classical approach is first to make it complete, determinize it, and then checks for reachability of nonaccepting states of the determinized VPA. To check the inclusion problem $L(A) \subseteq L(B)$, the standard approach computes the complement of B, takes its intersection with A, and then check for emptiness. This is costly as computing the complement necessitates determinization. This explosion is in some sense unavoidable, as the universality and language inclusion problem for VPAs are known to be EXPTIME-complete [5].

During the recent years, a lot of research has been done to implement efficiently operations like complementation [48, 49] and universality or inclusion checking on nondeterministic word, Büchi, or tree automata [77, 24, 14]. The solutions in [77, 24, 14] is so-called *antichain technique*; an antichain is a finite set of incomparable elements. Its idea comes from an analysis of the complementation, which consists of two steps (1) the determinization and (2) the alternation of final states. In a determinization, the subset construction generates determinized states, each of which is the collection of destination states of transitions of a word. A (forward) antichain further reduces it to minimal determinized states only. The idea is that, in either the universality or the inclusion checking, the final step is the emptiness checking, i.e., whether there exists a word reachable to a rejecting determinized state, regardless of which word is an instance. Although the antichain algorithm does not improve the complexity in theory, it is significant in practice. For instance, there had been virtually no implementations of complementing a Buchi au-

tomaton (which is $O(2^{n \log n})$), but one was given by antichains [24] and implemented as ALASKA [78].

In this chapter, we first apply the standard method to check universality and inclusion problems for nondeterministic VPA. The method includes two main steps: determinization and reachability checking for non-accepting configurations. For determinization, we use the Alur-Madhusudan's procedure. For reachability checking, we apply the symbolic technique $\mathcal{P}$-automata [29, 58] to compute the sets of all reachable configurations of a VPA. We implement this standard approach in a prototype tool written in Java 1.5.0/NetBeans 6.0. We test on randomly generated VPA. However, the performance of this method is very low. The program stuck with very mall size of input VPAs.

To improve the standard method, we propose and experimentally evaluate new efficient methods for checking universality and inclusion problems of VPAs: *on-the-fly* method and *antichain-based* method.

- **On-the-fly method.** To improve efficiency, we perform determinization and reachability checking on-the-fly manner. More precisely, we construct determinized VPA and $\mathcal{P}$-automaton simultaneously. For checking universality of nondeterministic VPA M, we first create the initial state of the determinized VPA M^d and a $\mathcal{P}$-automaton A to represent the initial configuration. Second, construct new transitions departing from the initial states, and update the $\mathcal{P}$-automaton A. Then, using new states and transitions of A (which correspond to pairs of the states and topmost stack symbols of M^d), update the determinized VPA M^d, and so on. When a nonaccepting state is added to A, we can stop and report that M is not universal.

- **Antichain-based method.** We extend the antichain-based algorithms [77] to visibly pushdown automata. In particular, as determinization is expensive, we first construct an algorithm for checking universality by keeping determinization implicitly. The main idea is to try to find at least one word not accepted by the VPA. For this sake, we follow the simultaneous technique as in the on-the-fly method. Besides, an ordering over transitions of determinized VPA is introduced to perform a kind of *minimal* symbolic simulation of the $\mathcal{P}$-automaton to cover all runs necessarily leading to non-accepting states. We also give a new algorithmic solution to inclusion checking for VPAs. Again, no explicit determinization is performed. To solve the language-inclusion problem for nondeterministic VPAs, $L(A) \subseteq L(B)$, the main idea is to find at leat one word w accepted by A but not accepted by B, i.e., $w \in L(A) \setminus L(B)$.

To evaluate the proposed algorithms, we have implemented them all in a prototype tool and tested them in a series of experiments. Although the standard approaches (as

well as ours) have the same worst case complexity, our prototype implementation outperforms those approaches where determinization is explicit. Preliminary experiments on randomly generated visibly pushdown automata show a significant improvement of on-the-fly and antichain-based methods compared to the standard method, especially when the universality / the inclusion do not hold. For the cases of universal VPAs, our experimental results show that the antichain-based method is considerably faster than the standard method.

5.1 Checking Universality and Inclusion of Finite Automata

In this section we present algorithms to check universality and inclusion for finite automata. In particular, we first briefly discuss the standard algorithms before entering to the antichain-based algorithms that were proposed recently by Wulf et al. [77].

5.1.1 Standard Methods

Definition 5.1. *A* (nondeterministic) finite automaton *(NFA) over a finite alphabet* Σ *is a tuple* $A = (Q, \Sigma, \Delta,$
$Q_0, F)$*, where* Q *is a set of finite states,* $\Delta \subseteq Q \times \Sigma \times Q$ *is a set of transitions,* $Q_0 \subseteq Q$ *is a set of initial states, and* F *is a set of final states.*

For $(q, a, q') \in \Delta$, we denote it by $q \xrightarrow{a} q'$. The finite automaton A is said to be *deterministic* if $|Q_0| = 1$, and for all $q \in Q$ and $a \in \Sigma$, there is at most one transition $(q, a, q') \in \Delta$. For $U \subseteq Q$, we denote $\Delta(U) = \{q' \mid q \in U, a \in \Sigma, (q, a, q') \in \Delta\}$.

For a word $u = a_1 a_2 \cdots a_n \in \Sigma^*$ and $q \in Q$, a (q-)*run* of A over w is a sequence $q = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \cdots q_{n-1} \xrightarrow{a_n} q_n = q'$, and denoted by $q \xrightarrow{u} q'$. The word w is *accepted* by A if u has a run $q \xrightarrow{u} q'$ such that $q \in Q_0$ and $q' \in F$. Let $L(A)$ denote the set of words accepted by A. Without loss of generality, we avoid ϵ transitions by substituting them with multiple initial states and nondeterminism. For instance, Figure 5.1 is an example of non-deterministic finite automata.

Theorem 5.1 ([72])**.** *Every NFA* A *can be determinized. Furthermore, if* A *has* n *states, then one can construct a deterministic FA* B *with at most* 2^n *states such that* $L(B) = L(A)$*. This technique is called subset construction.*

Solving universality problem using standard method The *universality problem* asks, given a NFA A over the alphabet Σ, if the language of A contains all finite words over Σ, that is, if $L(A) = \Sigma^*$. This problem is fundamental in automata theory, and several important problems in verification reduce polynomial time to this problem. The standard

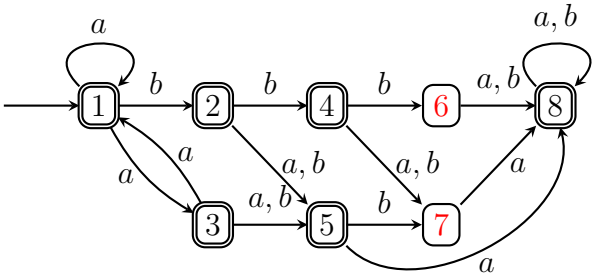

Figure 5.1: Finite automaton A

algorithm for universality of finite automata is to first determinize the automaton using the subset construction, and then check for the reachability of a set containing only non-accepting states. The subset construction may construct a deterministic automaton that is exponentially larger than the original automaton.

Example 5.1. Let us consider the NFA A given in Figure 5.1 where $Q = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $F = \{1, 2, 3, 4, 5, 8\}$. The standard method to check universality of A is illustrated in Figure 5.2.

Solving inclusion problem using standard method Let $A = (Q_A, \Sigma, \Delta_A, Q_A^0, F_A)$ and $B = (Q_B, \Sigma, \Delta_B, Q_B^0, F_B)$ be two NFAs. We want to check whether $L(A) \subseteq L(B)$. The standard approach computes the complement of B, takes its intersection with A, and checks for emptiness. It means that $L(A) \subseteq L(B) \iff L(A) \cap \overline{L(B)} = \varnothing$. This costly as computing the complement necessitates determinization. Here we show how to check inclusion without explicit determinization.

5.1.2 Antichain Methods

In [77], it was shown that explicit determinization via the subset construction can be avoided when solving universality for finite automata. To avoid the subset construction, they proposed the notion of antichains as above, and constructed monotone function on the lattice of the defined antichains. Then, checking universality can be done via forward or backward manners. For forward manner, the least fixed point the defined monotone function is enough to find counterexample words if the finite automaton is not universal. Similarly, for backward manner, the greatest fixed point of the defined monotone function is enough for checking universality. It has shown that the backward algorithm and the forward algorithm are equivalent [77]. In this section, we only present the forward manner to solve universality and inclusion problems for finite automata. Readers are referred to [77] for more details of backward manner.

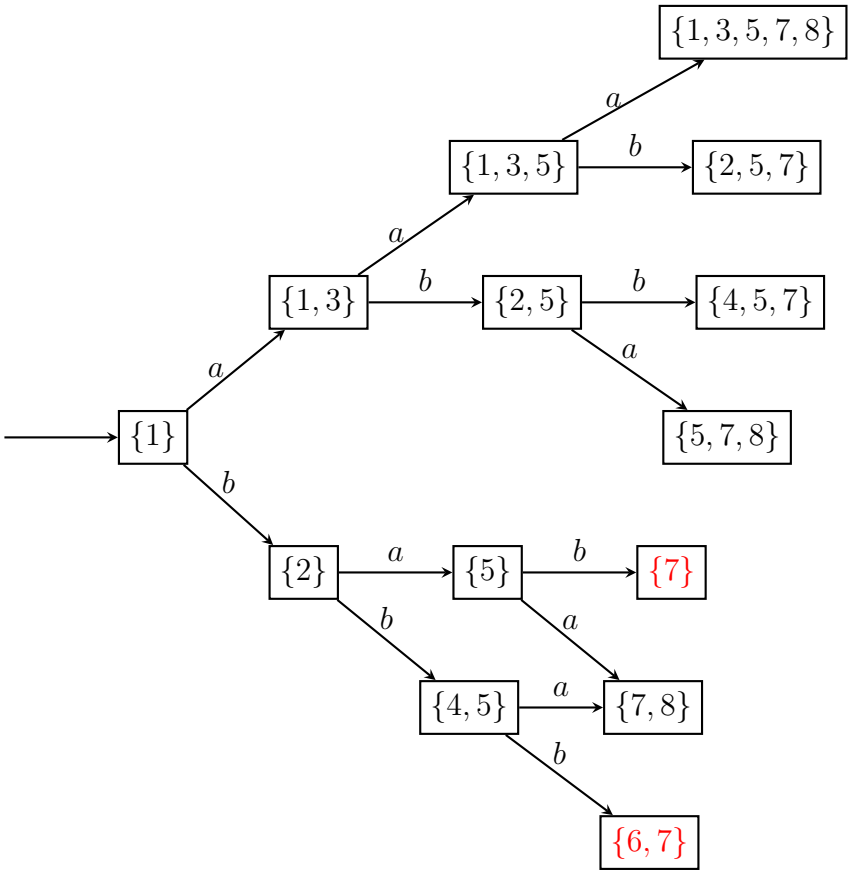

Figure 5.2: Determinization for checking universality of finite automaton A.

We now briefly recall the notion of antichains that were proposed in [77] for solving universality of finite automata. Let Q be a finite set (in [77], Q is a set of states of some finite automaton).

Definition 5.2. An *antichain* over Q is a set $p \subseteq 2^Q$ such that $\forall s, s' \in p : s \not\subseteq s'$ and $s' \not\subseteq s$. Intuitively, p is a set of pairwise incomparable subsets of Q.

Let L be the set of antichains over Q. Based on the subset-inclusion relation, we define the partial orders over L as follows:

Definition 5.3. For two antichains $p, p' \in L$, let $p \sqsubseteq p'$ iff $\forall s' \in p' \cdot \exists s \in p : s \subseteq s'$.

Given a set $p \subseteq 2^Q$ (not necessarily an antichain), a set $s \in p$ is maximal in p iff $\forall s' \in p : s \not\subseteq s'$. Similarly, $s \in p$ is minimal in p iff $\forall s' \in p : s' \not\subseteq s$. Denote $\lceil p \rceil$ (resp. $\lfloor p \rfloor$) for the set of maximal (resp. minimal) elements of p. We are now ready to define the operations over a set of antichains.

Definition 5.4. Given two antichains $p, p' \in L$, the $\sqsubseteq$-lub (least upper bound) is the antichain $p \sqcup p' = \lfloor \{s \cup s' \mid s \in p \wedge s' \in p'\} \rfloor$ and the $\sqsubseteq$-glb (greatest lower bound) is the antichain $p \sqcap p' = \lfloor \{s \mid s \in p \vee s \in p'\} \rfloor$.

Recall that a partially ordered set $(L, \sqsubseteq)$ is a *lattice* if for every two elements $p, p' \in L$ both the least upper bound and the greatest lower bound of $\{p, p'\}$ exist. A lattice is *complete* if for every subset $P \subseteq L$ both the least upper bound and the greatest lower bound of P exist. Based on this definition, the next lemma holds:

Lemma 5.2. *The partial orders $\sqsubseteq$ yields a complete lattice on the set L of antichains:* $\mathsf{Latt} = \langle L, \sqsubseteq, \bigsqcup, \bigsqcap, \emptyset, \{Q\}\rangle$.

The above lattice was used to solve universality for finite automata via forward algorithms. We are going to describe this technique in detail as below:

Using the lattice of antichains to solve universality Given a nondeterministic finite automaton $A = (Q, \Sigma, \Delta, Q_0, F)$, we define the following monotone function on the lattice L of antichains over Q. For an antichain $p \in L$, let

$$\mathsf{Post}(p) = \lfloor \{s' \mid \exists s \in p \cdot \exists a \in \Sigma \ : \ s' = \mathsf{post}_a^A(s)\} \rfloor$$

where $\mathsf{post}_a^A(s) = \{l' \in Q \mid \exists l \in s \ : \ (l, a, l') \in \Delta\}$.

So, a set s' of states belongs to the antichain $\mathsf{Post}(p)$ iff it is minimal and there exists a state set s and a symbol $a \in \Sigma$ such that $s' = \mathsf{post}_a^A(s)$. It is important to note that this Post function is monotone on the lattice of antichains Latt. The computation of Post can be seen as a step of implicit determinization for a NFA. This monotone function can be used to solve the universality problem for nondeterministic finite automata. We start with some detail definitions.

Definition 5.5. For a antichain $p \in L$, we define $\mathsf{Post}_0(p) = p$ and for all $i > 0$, $\mathsf{Post}_i(p) = \mathsf{Post}(\mathsf{Post}_{i-1}(p)) \sqcap p$.

Intuitively, $\mathsf{Post}_i(p)$ contains the $\sqsubseteq$-smallest elements $s \in 2^Q$ into which the NFA can nondeterministically get after processing words of length up to i starting from the states in the second component of the elements of p. Using only the minimal elements is enough as we just need to know whether there is a word on which the given NFA runs exclusively into non-final states. This makes universality checking easier than the standard approach that used explicit determinization.

Note that $\mathsf{Post}_1(p) = \mathsf{Post}(\mathsf{Post}_0(p)) \sqcap p \sqsubseteq p = \mathsf{Post}_0(p)$. Moreover, for $i > 0$, if $\mathsf{Post}_i(p) \sqsubseteq \mathsf{Post}_{i-1}(p)$, then due to the monotonicity of Post, $\mathsf{Post}(\mathsf{Post}_i(p)) \sqsubseteq \mathsf{Post}(\mathsf{Post}_{i-1}(p))$, $\mathsf{Post}(\mathsf{Post}_i(p)) \sqcap p \sqsubseteq \mathsf{Post}(\mathsf{Post}_{i-1}(p)) \sqcap p$, and thus $\mathsf{Post}_{i+1}(p) \sqsubseteq \mathsf{Post}_i(p)$. Altogether, we obtain:

Lemma 5.3. $\forall p \in L \cdot \ \forall i \geq 0 \ : \ \mathsf{Post}_{i+1}(p) \sqsubseteq \mathsf{Post}_i(p)$.

Since we work on a finite lattice, by Tarski's fixed point theorem, this implies that for all p there exists a number fi (abbreviation for fixed point index) such that $\mathsf{Post}_{fi}(p) = \mathsf{Post}_{fi+1}(p)$. We let $\mathsf{Post}^*(p) = \mathsf{Post}_{fi}(p)$.

Let $p_0 = \{Q_0\}$ be the *initial antichain*. Now, we explain the main difference between the antichain approach and the standard approach. Intuitively, $\mathsf{Post}^*(p_0)$ can be seen as a subset of states of the determinized FA. The computation of $\mathsf{Post}^*(p_0)$ can be seen as doing determinization implicit. The computation of this least fixed point can be used to check universality for NFA. This is formalized in the next theorem.

Theorem 5.4 ([77, Theorems 3])**.** *An NFA $A = (Q, \Sigma, \Delta, Q_0, F)$ is not universal iff $\exists s \in \mathsf{Post}^*(p_0)$ such that $s \cap F = \varnothing$.*

The algorithm that consists in computing the least fixed point $\mathsf{Post}^*(p_0)$ from Theorem 5.4 is called the *forward antichain algorithm*. The computation is similar to the subset construction used in the forward determinization of A, with the essential difference that it maintains only sets of states that are *minimal* in the subset-inclusion order.

Remark 5.1. When the automaton is not universal, then $\mathsf{Post}^*(p_0)$ is not fully computed, because we stop the computation as soon as one of the sets in $\mathsf{Post}^*(p_0)$ does not intersect with F.

Algorithm 1: The antichain algorithm for universality

Data: A NFA $A = (Q, \Sigma, Q_0, \Delta, F)$.

Result: universality of A

```
1  begin
2      p_0 ⟵ {Q_0};
3      p ⟵ p_0;
4      if ∃s ∈ p : s ∩ F = ∅ then
5          return False;
6      WorkList ⟵ Post(p) ⊓ p;
7      while (p ≠ WorkList) do
8          p ⟵ WorkList;
9          if ∃s ∈ p : s ∩ F = ∅ then
10             return False;
11         WorkList ⟵ Post(p) ⊓ p;
12     return True;
13 end
```

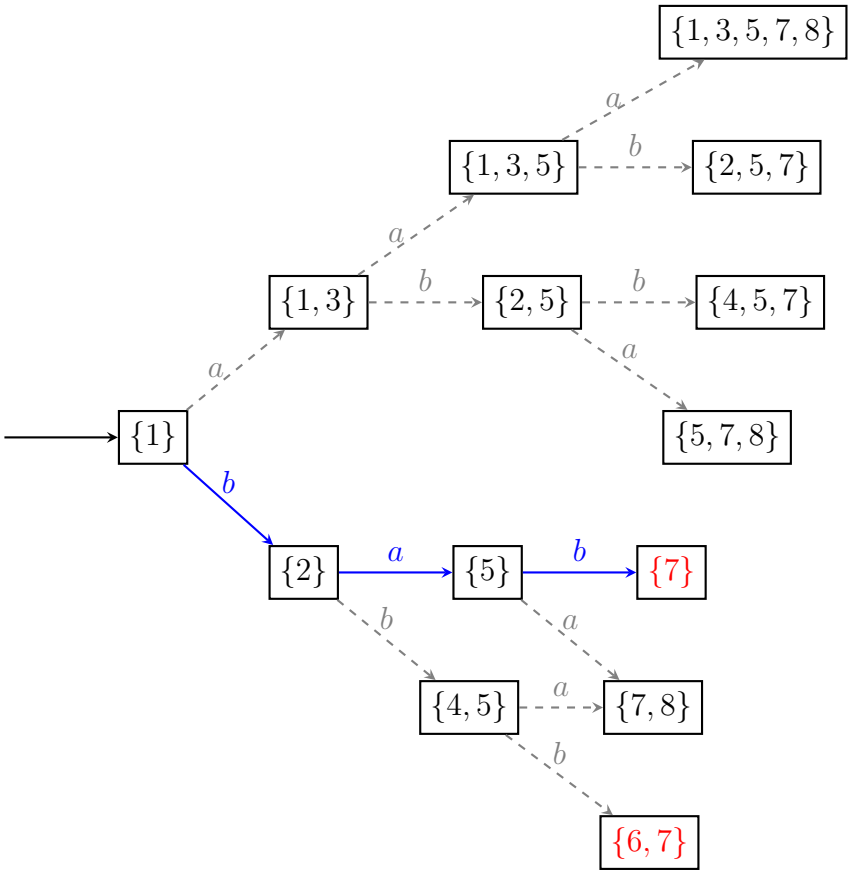

Figure 5.3: Checking universality of finite automaton A via antichains.

Example 5.2. Let us describe this antichain method by checking universality for the NFA A in Figure 5.1. We first have: $p_0 = \{\{1\}\}$, then

- $\mathsf{Post}_1(p_0) = \mathsf{Post}(p_0)) \sqcap p_0 = \{\{1\},\{2\},\{1,3\}\} \sqcap \{\{1\}\} = \lfloor\{\{1\},\{2\},\{1,3\}\}\rfloor = \{\{1\},\{2\}\}$
- $\mathsf{Post}_2(p_0) = \mathsf{Post}(\mathsf{Post}_1(p_0)) \sqcap p_0 = \{\{1\},\{2\},\{1,3\},\{5\},\{4,5\}\} \sqcap \{\{1\}\}$ $= \lfloor\{\{1\},\{2\},\{1,3\},\{5\},\{4,5\}\}\rfloor = \{\{1\},\{2\},\{5\}\}$
- $\mathsf{Post}_3(p_0) = \mathsf{Post}(\mathsf{Post}_2(p_0)) \sqcap p_0 = \{\{1\},\{2\},\{1,3\},\{5\},\{7,8\},\{7\}\} \sqcap \{\{1\}\}$ $= \lfloor\{\{1\},\{2\},\{1,3\},\{5\},\{7,8\},\{7\}\}\rfloor = \{\{1\},\{2\},\{5\},\{7\}\}$

Since $\{7\} \in \mathsf{Post}_3(p_0)$ and $\{7\} \cap \{1,2,3,4,5,8\} = \varnothing$, we obtain that the NFA A is not universal. The computation of $\mathsf{Post}_3(p_0)$ is illustrated in Figure 5.3.

Remark 5.2. There are three main advantages of the antichain method over the standard one as follows. First, the new antichain algorithm keep determinization implicit. Second, the antichain algorithm try to find at least a word not accepted by the automaton and

computes only what is necessary to establish the existence of such a word. Third, antichains of state sets allow us to store only minimal (or, maximal for backward manner) subsets of states for which we know a finite automaton is universal or not.

Using the lattice of antichains to solve inclusion Let $A = (Q_A, \Sigma, \Delta_A, Q_A^0, F_A)$ and $B = (Q_B, \Sigma, \Delta_B, Q_B^0, F_B)$ be two NFAs. We want to check whether $L(A) \subseteq L(B)$. The standard approach computes the complement of B, takes its intersection with A, and checks for emptiness. This costly as computing the complement necessitates determinization. Here we show how to check inclusion without explicit determinization. The language inclusion for finite automata can be checked using an antichain algorithm based on a slightly richer lattice. An *antichain* over $Q_A \times 2^{Q_B}$ is a set $p \in 2^{Q_A \times 2^{Q_B}}$ such that for all $(l_1, s_1), (l_2, s_2) \in p$ with $l_1 = l_2$ and $s_1 \neq s_2$, we have neither $s_1 \subseteq s_2$ nor $s_2 \subseteq s_1$. Given a set $p2^{Q_A \times 2^{Q_B}}$, an element $l, s) \in q$ is *minimal* iff for every s' with $s' \subset s$, we have $(l, s') \notin p$. We denote by $\lfloor p \rfloor$ the set of minimal elements of p.

Definition 5.6. Given two antichains p and p', we define:

- $p \sqsubseteq_l p'$ iff $\forall (l, s') \in p' \cdot \exists (l, s) \in p : s \subseteq s'$,
- $p \sqcup_l p' = \lfloor \{(l, s \cup s') \mid (l, s) \in p \wedge (l, s') \in p'\} \rfloor$,
- $p \sqcap_l p' = \lfloor \{l, s) \mid (l, s) \in p \vee (l, s) \in p'\} \rfloor$,

Let $\mathsf{IPost}_l(p) = \lfloor \{(l', s') \mid \exists a \in \Sigma \cdot \exists (l, s) \in p \ : \ (l, a, l') \in \Delta_A \wedge \mathsf{post}_a^B(s) = s'\} \rfloor$. Let $p_0 = Q_A^0 \times \{Q_B^0\}$ be the initial antichain. Similar to the case for universality checking, the next theorem holds:

Theorem 5.5. *Let A and B be two two finite automata. Then, $L(A) \not\subseteq L(B)$ iff there exists $p = (l, s) \in \mathsf{IPost}^*(p_0)$ such that: $l \in F_A \wedge s \subseteq (Q_B \setminus F_B)$.*

5.2 Checking Universality and Inclusion of Visibly Pushdown Automata

Since the set of configurations or a VPA is infinite, we cannot directly apply the antichain method to compute the least fixed point as for the finite automata case. This is because the computation of the least fixed point for an infinite lattice will not terminate. In this section, we propose new two methods (*on-the-fly* and *antichain-based*) to solve the universality and inclusion problems for visibly pushdown automata. We first briefly recall the standard method in the next subsection.

5.2.1 Standard Methods

Solving universality problem using standard method The *universality problem* asks, given a nondeterministic VPA A over the alphabet Σ, if the language of A contains all finite words over Σ, that is, if $L(A) = \Sigma^*$. The standard algorithm for universality of finite automata is to first determinize the automaton, and then check for the reachability of a nonaccepting states.

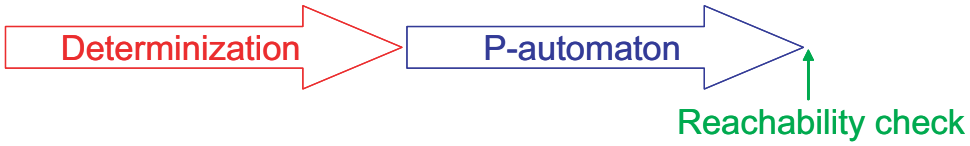

Figure 5.4: Checking Universality of VPA via Standard Method

Step 1: Determinization The first step is to do determinization. As shown in [5], any nondeterministic VPA can be transformed into an equivalent deterministic VPA. The key idea of the determinization procedure is to do a subset construction, but postponing handling push transitions. The push transitions are stored into the stack and simulated at the time of matching pop transitions. The construction has two components: a set of *summary edges* S, that keeps track of what state transitions are possible from a push transition to the corresponding pop transition, and a set of *path edges* R, that keeps track of all possible state reached from initial states. Formally, let $M = (Q, \Gamma, Q_0, \Delta, F)$ be a nondeterministic VPA. We construct an equivalent deterministic VPA $M^d = (Q', \Gamma', Q'_0, \Delta', F')$ as follows:

- $Q' = 2^{Q \times Q} \times 2^Q$,
- $Q'_0 = \{(Id_Q, Q_0)\}$ where $Id_Q = \{(q, q) \mid q \in Q\}$,
- $F' = \{(S, R) \mid R \cap F \neq \varnothing\}$,
- $\Gamma' = Q' \times \Sigma_c$.

The transition relation $\Delta' = \Delta'_i \cup \Delta'_c \cup \Delta'_r$ is given by:

- **Internal:** For every $a \in \Sigma_i$, $(S, R) \xrightarrow{a} (S', R') \in \Delta'_i$ where

$$\begin{cases} S' &= \{(q, q') \mid \exists q'' \in Q : (q, q'') \in S, q'' \xrightarrow{a} q' \in \Delta_i\} \\ R' &= \{q' \mid \exists q \in R : q \xrightarrow{a} q' \in \Delta_i\} \end{cases}$$

- **Push:** For every $a \in \Sigma_c$, $(S, R) \xrightarrow{a/+(S,R,a)} (Id_Q, R') \in \Delta'_c$ where

$$R' = \{q' \mid \exists q \in R : q \xrightarrow{a/+\gamma} q' \in \Delta_c\}$$

- **Pop:** For every $a \in \Sigma_r$,
  - if the stack is empty : $(S, R) \xrightarrow{a/-\perp} (S', R') \in \Delta'_r$ where

$$\begin{cases} S' &= \{(q, q') \mid \exists q'' \in Q : (q, q'') \in S, q'' \xrightarrow{a/-\perp} q' \in \Delta_r\} \\ R' &= \{q' \mid \exists q \in R : q \xrightarrow{a/-\perp} q' \in \Delta_r\} \end{cases}$$

  - otherwise: $(S, R) \xrightarrow{a/-(S',R',a')} (S'', R'') \in \Delta'_r$, where

$$\begin{cases} R'' &= \left\{ q' \;\middle|\; \begin{array}{l} \exists q_1 \in Q, q_2 \in R : (q_1, q_2) \in S, \\ q \xrightarrow{a'/+\gamma} q_1 \in \Delta_c, q_2 \xrightarrow{a/-\gamma} q' \in \Delta_r \end{array} \right\} \\ S'' &= \{(q, q') \mid \exists q_3 \in Q : (q, q_3) \in S', (q_3, q') \in Update\} \\ Update &= \left\{ (q, q') \;\middle|\; \begin{array}{l} \exists q_1 \in Q, q_2 \in R : (q_1, q_2) \in S, \\ q \xrightarrow{a'/+\gamma} q_1 \in \Delta_c, q_2 \xrightarrow{a/-\gamma} q' \in \Delta_r \end{array} \right\} \end{cases}$$

Step 2: Checking Reachability using $\mathcal{P}$-automata Checking reachability configurations of determinized VPA can be done by using $\mathcal{P}$-automata technique, which was presented in Section 2.3. If a non-accepting configuration $c = (q, w)$ $(q \notin F)$ is found, we stop and report that the original VPA is not universal. Otherwise, if all reachable configurations of determinized VPA are accepting configurations, we report that the original VPA is universal. The diagram describes in Figure 5.4 two steps of the standard method.

Solving inclusion problem using standard method Let A and B be two VPAs. We want to check whether $L(A) \subseteq L(B)$. The standard approach computes the complement of B, takes its intersection with A, and checks for emptiness (equivalent to reachability check). It means that $L(A) \subseteq L(B) \iff L(A) \cap \overline{L(B)} = \varnothing$. This costly as computing the complement necessitates determinization. Here we show how to check inclusion without explicit determinization.

5.2.2 On-the-fly Methods

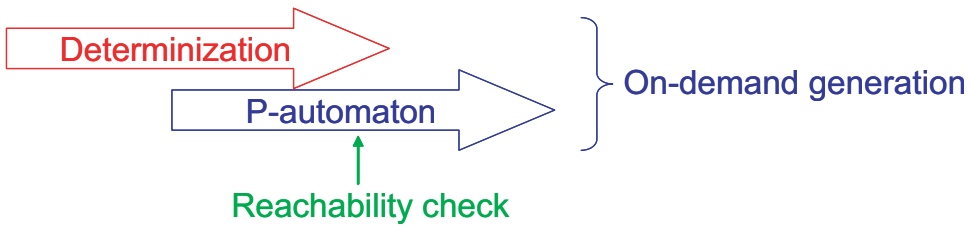

Figure 5.5: Checking Universality of VPA via On-the-fly Method

In this subsection we propose an on-the-fly method to check universality of VPAs by doing determinization and $\mathcal{P}$-automata construction simultaneously.

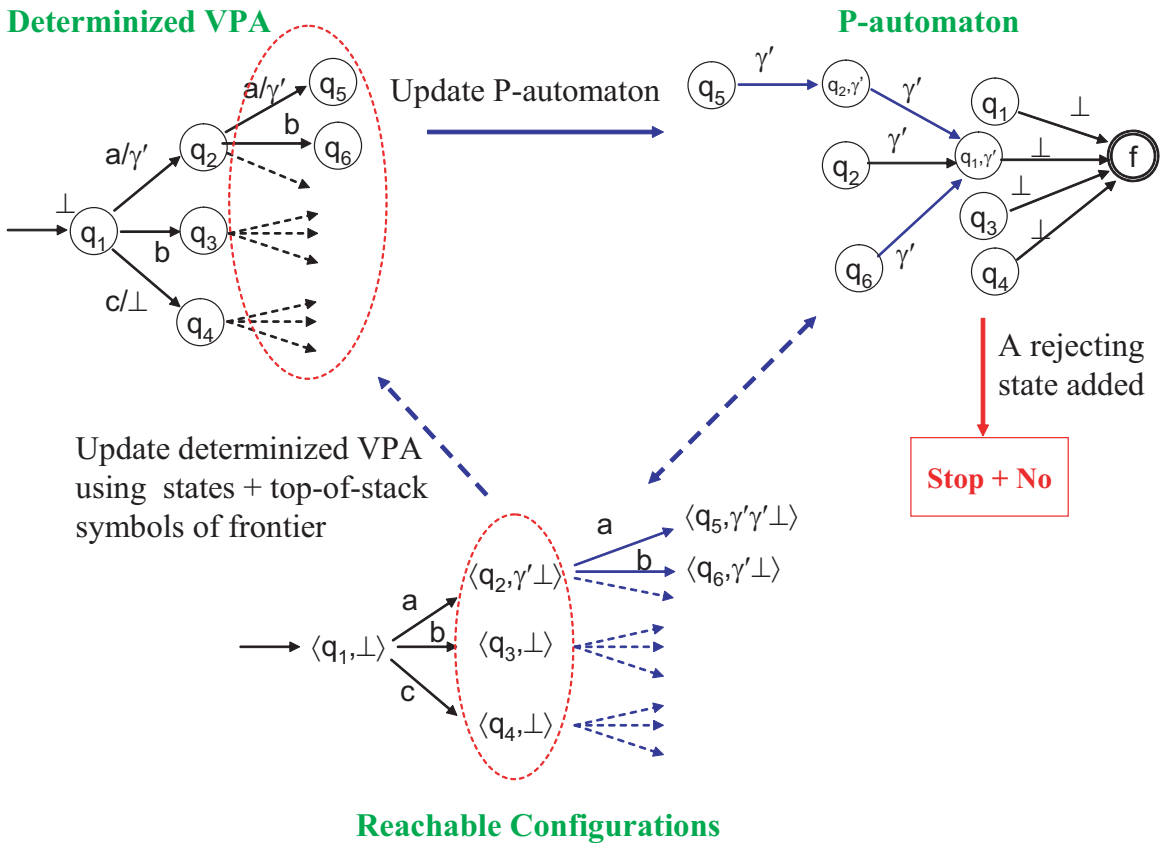

Figure 5.6: Description of the On-the-fly Method

Simultaneous On-the-fly Determinization and $\mathcal{P}$-Automata Construction To improve the efficiency, we perform simultaneously on-the-fly determinization and $\mathcal{P}$-automata construction. There are two interleaving phases in this approach. First, we determinize VPA step by step (iterations). After each step of determinization, we update the $\mathcal{P}$-automaton. Then, using the $\mathcal{P}$-automaton, we perform determinization again, and so on. It is crucial to note that this procedure terminates. This is because the size of M^d is finite, and the $\mathcal{P}$-automaton construction is terminated. However, once a non-accepting state is added to the $\mathcal{P}$-automaton, we stop and report that the VPA is not universal. Let $\mathsf{Conf}(M^d)$ and $\mathsf{Non\text{-}Accepting\text{-}Conf}(M^d)$ denote the sets of reachable and non-accepting configurations of M^d, respectively. We have the following theorem as a criterion for on-the-fly method, and this method is formalized in Algorithm 2.

Theorem 5.6. *Let M be a nondeterministic VPA. The automaton M is not universal iff there exists an non-accepting reachable configuration, i.e.,* $\mathsf{Non\text{-}Accepting\text{-}Conf}(M^d) \cap \mathsf{Conf}(M^d) \neq \varnothing$.

Example 5.3. We illustrate the on-thy-fly algorithm by an example given in Figure 5.6. In the example, we assume that $a \in \Sigma_c$, $b \in \Sigma_i$, and $c \in \Sigma_r$. The process of the algorithms is performed as below:

Algorithm 2: The on-the-fly algorithm for checking university of VPA

Data: A nondeterministic VPA $M = (Q, Q_0, \Gamma, \Delta, F)$

Result: Universality of M

```
1  begin
2      Create the initial state of the determinized VPA M^d;
3      Initiate P-automaton A to present the initial configuration of M^d;
4      A_post* ⟵ A;
5      Create transitions of M^d departing from the initial state;
6      while the set of new transitions of M^d is not empty do
7          Update the P-automaton A_post* using new transitions of M^d;
8          if a rejecting state is added to A_post* then
9              return False;
10         Update M^d using new transitions of A_post*;
11     return True;
12 end
```

1. At the first time, assume that the initial state q_1 of determinized VPA M^d is created.

2. Then, the $\mathcal{P}$-automaton A is constructed which includes two states $\{q_1, f\}$ and one transition $q_1 \xrightarrow{\perp} f$, where f is a unique final state. $\mathcal{P}$-automaton A represents a set of initial configurations $\{(q_1, \perp)\}$ of M^d.

3. Update M^d using A. Suppose that M^d has new states $\{q_2, q_3, q_4\}$; and new transitions $\{q_1 \xrightarrow{a/+\gamma'} q_2, q_1 \xrightarrow{b} q_3, q_1 \xrightarrow{c/-\perp} q_4\}$.

4. Update $\mathcal{P}$-automaton A using new transitions of M^d. A has new states $\{q_2, q_3, q_4, p_{(q_1,\gamma')}\}$ and transitions $\{q_2 \xrightarrow{\gamma'} p_{(q_1,\gamma')}, p_{(q_1,\gamma')} \xrightarrow{\perp} f, q_3 \xrightarrow{\perp} f, q_4 \xrightarrow{\perp} f\}$.

5. We again return to update M^d using new transitions of A, and so on.

On-the-fly algorithm for checking inclusion of VPA Let A and B be two VPAs. We want to check whether $L(A) \subseteq L(B)$. This is equivalent to $L(A \times \overline{B}) = \varnothing$. The on-the-fly approach tries to find if there exists at least a word $w \in L(A) \setminus L(B)$. If such a word w was found, we can conclude that $L(A) \nsubseteq L(B)$. Otherwise, $L(A)$ is a subset of $L(B)$. To do so, similar to the case of universality checking, we perform simultaneously on-the-fly determinization for B and $\mathcal{P}$-automata construction for the product VPA $A \times \overline{B}$. Once a state $(p, q) \in (F_A \times F_{\overline{B}}$ is added to the $\mathcal{P}$-automaton, it means that there is a word $w \in L(A) \setminus L(B)$. We stop and report that $L(A) \nsubseteq L(B)$.

5.2.3 Antichain-based Methods

In this subsection, we improve the on-the-fly method to solve the universality and inclusion problems for nondeterministic VPA. The key ideas of our approach are also based on antichain technique. However, in our algorithms, the antichain technique was performed over the set of transitions of the corresponding $\mathcal{P}$-automaton. Where we only need to keep "minimal" transitions of $\mathcal{P}$-automaton for checking universality and inclusion.

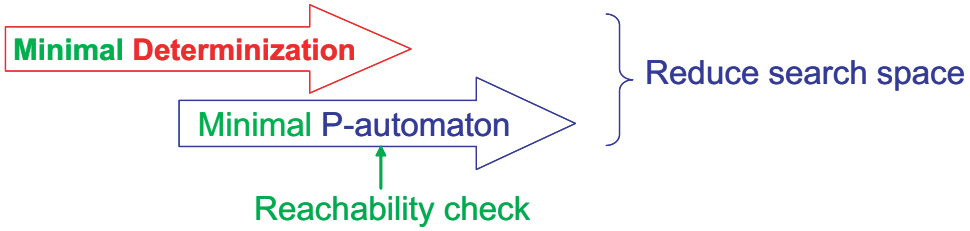

Figure 5.7: Checking Universality of VPA via Antichain-Based Method

Simultaneous Minimal Determinization and Minimal $\mathcal{P}$-Automata Construction We begin with the following observations that play an important role in establishing theoretical background for correctness of our algorithms. For a given nondeterministic VPA M, let M^d be the determinized VPA. We now define an ordering over states and stack symbols of M^d as follows:

Definition 5.7 (Ordering over determinized states and stack symbols)**.**

- Let $q'_1 = (S_1, R_1)$ and $q'_2 = (S_2, R_2)$ be states of M^d. We say $q'_1 \leq q'_2$ if $S_1 \subseteq S_2$ and $R_1 \subseteq R_2$.
- Let $\gamma'_1 = (S_1, R_1, a)$ and $\gamma'_2 = (S_2, R_2, a)$ be stack symbols of M^d. We say $\gamma'_1 \leq \gamma'_2$ if $S_1 \subseteq S_2$ and $R_1 \subseteq R_2$.

With these definitions, we obtain the following lemma:

Lemma 5.7. *Let* $(S_1, R_1) \xrightarrow{a} (S'_1, R'_1)$ *and* $(S_2, R_2) \xrightarrow{a} (S'_2, R'_2)$ *be internal transitions of the determinized VPA* M^d*. We have* $(S'_1, R'_1) \leq (S'_2, R'_2)$ *if* $(S_1, R_1) \leq (S_2, R_2)$*.*

Proof. By the determinization procedure, we have:

- $(S_1, R_1) \xrightarrow{a} (S'_1, R'_1) \in \Delta'_i$ where

$$\begin{cases} S'_1 &= \{(q, q') \mid \exists q'' \in Q : (q, q'') \in S_1, q'' \xrightarrow{a} q' \in \Delta_i\} \\ R'_1 &= \{q' \mid \exists q \in R_1 : q \xrightarrow{a} q' \in \Delta_i\} \end{cases}$$

- $(S_2, R_2) \xrightarrow{a} (S_2', R_2') \in \Delta_i'$ where

$$\begin{cases} S_2' &= \{(q, q') \mid \exists q'' \in Q : (q, q'') \in S_2, q'' \xrightarrow{a} q' \in \Delta_i\} \\ R_2' &= \{q' \mid \exists q \in R_2 : q \xrightarrow{a} q' \in \Delta_i\} \end{cases}$$

Thus, it is easy to verify that $(S_1', R_1') \leq (S_2', R_2')$ if $(S_1, R_1) \leq (S_2, R_2)$. □

Similarly, for the cases of push and pop transitions, the next two lemmas hold:

Lemma 5.8. *Let* $(S_1, R_1) \xrightarrow{a/+(S_1,R_1,a)} (Id_Q, R_1')$ *and* $(S_2, R_2) \xrightarrow{a/+(S_2,R_2,a)} (Id_Q, R_2')$ *be push transitions of the determinized VPA* M^d*. We have* $(Id_Q, R_1') \leq (Id_Q, R_2')$ *if* $(S_1, R_1) \leq (S_2, R_2)$*.*

Lemma 5.9. *Let* $(S_1, R_1) \xrightarrow{a/-(S_1',R_1',a')} (S_1'', R_1'')$ *and* $(S_2, R_2) \xrightarrow{a/-(S_2',R_2',a')} (S_2'', R_2'')$ *be pop transitions of the determinized VPA* M^d*. Assume that* $(S_1', R_1', a') \leq (S_2', R_2', a')$ *and* $(S_1, R_1) \leq (S_2, R_2)$*. Then, we have* $(Id_Q, R_1') \leq (Id_Q, R_2')$*.*

We now extend the ordering in Definition 5.7 to define a ordering over configurations of the determinized VPA M^d.

Definition 5.8. Let $c_1 = (q_1, \gamma_n \cdots \gamma_1 \perp)$ and $c_2 = (q_2, \gamma_n' \cdots \gamma_1' \perp)$ be two configurations of M^d. We say $(q_1, \gamma_n \cdots \gamma_1 \perp) \leq (q_2, \gamma_n' \cdots \gamma_1' \perp)$ iff the following conditions hold:

- $q_1 = (S_1, R_1) \leq (S_2, R_2) = q_2$, and
- $\gamma_i \leq \gamma_i'$ for all $1 \leq i \leq n$.

We now obtain a key theorem as a criterion for minimal generation of the $\mathcal{P}$-automaton for universality solving.

Lemma 5.10 (Key Lemma)**.** *Let* $c_1 = (q_1, \gamma_n \cdots \gamma_1 \perp)$ *and* $c_2 = (q_2, \gamma_n' \cdots \gamma_1' \perp)$ *be configuration of* M^d *such that* $(q_1, \gamma_n \cdots \gamma_1 \perp) \leq (q_2, \gamma_n' \cdots \gamma_1' \perp)$*. For any word* $w = a_1 \cdots a_n \in \Sigma^*$*, if:*

$$(q_1, \gamma_n \cdots \gamma_1 \perp) \xrightarrow{w} (\bar{q}_1, \bar{\gamma}_m \cdots \bar{\gamma}_1 \perp) \text{ and } (q_2, \gamma_n' \cdots \gamma_1' \perp) \xrightarrow{w} (\bar{q}_2, \bar{\gamma}_m' \cdots \bar{\gamma}_1' \perp)$$

Then, $(\bar{q}_1, \bar{\gamma}_n \cdots \bar{\gamma}_1 \perp) \leq (\bar{q}_2, \bar{\gamma}_m' \cdots \bar{\gamma}_1' \perp)$

Proof. We prove this lemma by induction on the length $|w|$ of w. If $|w| = 1$, it means that $w = a$. The proof immediately follows from Lemma 5.7, Lemma 5.8, or 5.9 wrt. the type of input symbol a. Now, assume that the lemma holds for the case $|w| = n$. Again, using induction hypothesis and Lemmas 5.7, 5.8 or 5.9, it is easy to see that this lemma also holds for the case $|w| = n + 1$. The lemma is proved. □

Key Observation: Keep Only Minimal Configurations It is crucial to note that, $L(M) \neq \Sigma^*$, iff there exists an *non-accepting* reachable configuration of M^d. Recall that a configuration $((S, R), \sigma)$ is non-accepting if $R \cap F = \varnothing$. Note that if $((S, R), \sigma) \leq ((S', R'), \sigma')$ and $R' \cap F = \varnothing$, then $R \cap F = \varnothing$. Based on this observation and Lemma 5.10, it is sufficient to compute only minimal reachable configurations and check for the existence of a *non-accepting* configuration. Formally, let $\mathsf{Minimal\text{-}Conf}(M^d)$ denote the set of minimal configurations of M^d, and let $\mathsf{Non\text{-}Accepting\text{-}Conf}(M^d)$ denote the set of non-accepting configurations of M^d. We have the main theorem as below:

Theorem 5.11 (Main Theorem)**.** *Let M be a nondeterministic VPA. The automaton M is not universal iff there exists an non-accepting minimal reachable configuration, i.e.,* $\mathsf{Minimal\text{-}Conf}(M^d) \cap \mathsf{Non\text{-}Accepting\text{-}Conf}(M^d) \neq \varnothing$.

Algorithm 3: The antichain-based algorithm for checking university of VPA

Data: A nondeterministic VPA $M = (Q, Q_0, \Gamma, \Delta, F)$

Result: Universality of M

```
1  begin
2      Create the initial state of the minimal determinized VPA M^md;
3      Initiate P-automaton A to present the initial configuration of M^md;
4      A^min_post* ⟵ A;
5      Create transitions of M^md departing from the initial state;
6      while the set of new transitions of M^md is not empty do
7          Update the P-automaton A^min_post* using new transitions of M^md;
8          if a rejecting state is added to A^min_post* then
9              return False;
10         Update M^md using new transitions of A^min_post*;
11     return True;
12 end
```

Minimization $\mathcal{P}$-automata Let $C_0 = \{((Id_Q, Q_0), \bot)\}$ be the set of initial configurations of M^d. The set of $\mathsf{Minimal\text{-}Conf}(M^d)$ (represented by $A^{min}_{post^*}$) is computed by minimizing the $\mathcal{P}$-automaton A_{post^*} as follows: for two configurations $(q'_1, \gamma'_1\sigma)$ and $(q'_2, \gamma'_2\sigma)$, we only need to compare the states and top-of-stack symbols. To do that, we have the following procedure:

Definition 5.9 (Minimization of $\mathcal{P}$-automata)**.** $A^{min}_{post^*}$ is obtained from A_{post^*} by the following optimization procedure: If $(q'_1, \gamma'_1, p) \in A_{post^*}$ and $(q'_2, \gamma'_2, p) \in A_{post^*}$ such that $q'_1 \leq q'_2 \wedge \gamma'_1 \leq \gamma'_2$, then remove the transition (q'_2, γ'_2, p).

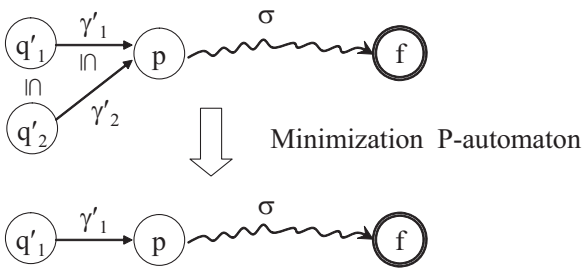

Figure 5.8: Minimization of P-automata

Example 5.4. To describe how the antichain-based algorithm work, we revisit Example 5.3. Without loss of generality, we assume that in the determinized VPA M^d $q_1 \subseteq q_3, q_1 \subseteq q_4$, and $q_2 \subseteq q_6$. With this assumption, the process of antichain-based method for solving universality problem of VPA is illustrated in Figure 5.9.

Antichain-based algorithm for checking inclusion of VPA Similar to the on-the-fly algorithm for inclusion checking, the antichain-based approach tries to find if there exists at least a word $w \in L(A) \setminus L(B)$. If such a word w was found, we conclude that $L(A) \nsubseteq L(B)$. Otherwise, $L(A)$ is a subset of $L(B)$. Here, different from the on-the-fly algorithm, we perform simultaneously minimal determinization for B and minimal $\mathcal{P}$-automata construction for the product VPA $A \times \overline{B}_{min}$, where $\overline{B}_{min}$ is complement of the minimal determinization of B. Once a state $(p, q) \in (F_A \times F_{\overline{B}_{min}}$ is added to the $\mathcal{P}$-automaton, it means that there is a word $w \in L(A) \setminus L(B)$. We stop and report that $L(A) \nsubseteq L(B)$. Otherwise, $L(A) \subseteq L(B)$.

5.3 Implementation and Experiments

To compare the antichain-based algorithm with the standard algorithm, we run our implementations on randomly generated VPAs. The package is implemented in Java 1.5.0 on Windows XP, and all tests are performed on a PC equipped with 1.50 GHz Intel® Core™ Duo Processor L2300 and 1.5 GB of memory. During experiments, we fix the size of the input alphabet to $|\Sigma_c| = |\Sigma_r| = |\Sigma_i| = 2$, the size of the stack alphabet to $|\Gamma| = 3$.

We first set parameters of the test set are (i) the *density of final states* $f = \frac{|F|}{|Q|}$, and (ii) the *transition density* $r = \frac{k_a}{|Q|}$ where k_a is the number of transitions for each input symbol a. For instance, $r = 3$ for 20 states mean 360 transition rules. Timeout is set to 60 seconds for universality checking. Table 5.1 shows results of three algorithms, ANTICHAIN, ON-THE-FLY, and STANDARD random VPAs with $r = 3$ and $f = 1$. Since the random generated VPAs are not complete, almost VPAs are detected not to be universal. We ran tests over 100 instances for each sample point. The results show that ANTICHAIN and ON-THE-FLY are significantly faster than STANDARD.

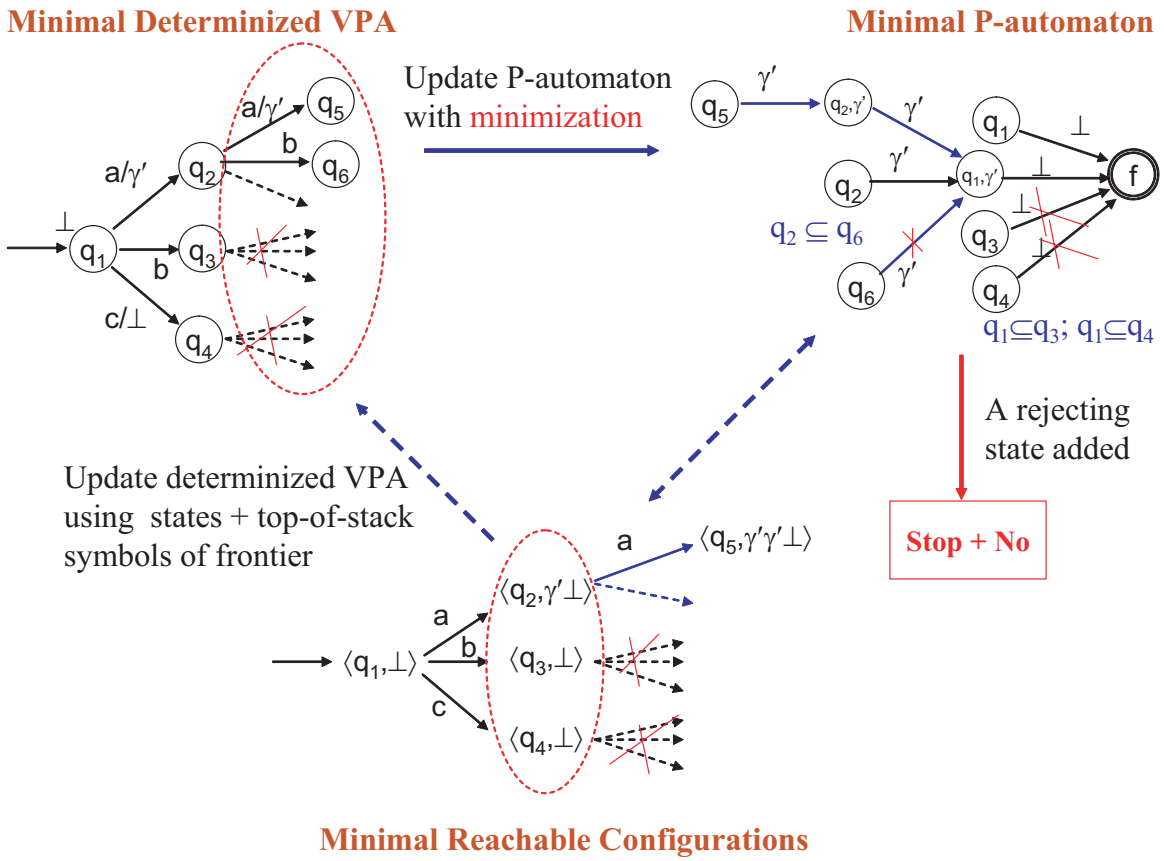

Figure 5.9: Description of the Antichain-based Method

To produce randomly generated complete VPAs, we change parameters of the tests as follows: *density of final states* $f = \frac{|F|}{|Q|}$, and the *density of transitions* $r : Q \times \Sigma \to \mathbb{N}$; $r(q, a)$ depends on not only the input symbol a but also on the state q. This parameter guarantees that randomly generated VPAs are complete. Therefore, the probability of being universal increases. We randomly generate a complete and nondeterminitic (for each state and input) VPA, and further randomly choose final states up to the rate f is satisfied.

- $r(q, a) = 2$ for all $q \in Q$ and $a \in \Sigma_c$,
- $r(q, b) = 6$ for all $q \in Q$ and $b \in \Sigma_r$. This is because the return transitions also depend on the stack symbols, and
- $r(q, c) = 2$ for all $q \in Q$ and $c \in \Sigma_i$.

We see that, for instance, with the above parameters, a VPA with 10 states has 200 transitions. To evaluate the performance of algorithms, we first test for randomly VPAs with 10 states and various density f. Table 5.2 shows results of three algorithms, ANTICHAIN, ON-THE-FLY, and STANDARD for 10 states with various density of final

Table 5.1: checking random VPA with $r = 3$, $f = 1$

	number of states							
ANTICHAIN	10	20	30	40	50	60	70	80
not universal	100	100	98	99	96	98	95	93
	1.18	2.52	3.41	4.54	6.27	9.93	9.05	16.32
timeout (*60* s)	0	0	2	1	4	2	5	7
total time	118	252	455	510	842	1094	1160	1938
	number of states							
ON-THE-FLY	10	20	30	40	50	60	70	80
not universal	100	100	98	99	92	84	84	66
	1.45	3.58	5.30	7.76	8.31	8.60	9.34	11.98
timeout (*60* s)	0	0	2	1	8	16	16	34
total time	145	358	640	829	1245	1683	1744	2831
	number of states							
STANDARD	10	20	30	40	50	60	70	80
not universal	-	-	-	-	-	-	-	-
	-	-	-	-	-	-	-	-
timeout (*60* s)	100	100	100	100	100	100	100	100
total time	6000	6000	6000	6000	6000	6000	6000	6000

states f. We ran with 20 samples for each point, setting timeout to 180 seconds. Results of STANDARD are all timeout. ON-THE-FLY and ANTICHAIN both behave in significantly better ways than those of STANDARD. For universal VPA cases, we see that ANTICHAIN outperforms ON-THE-FLY very much. For not universal VPA cases, the performances of ANTICHAIN and ON-THE-FLY are equivalent.

To evaluate scalability of the algorithms, we set the density of final states $f = 0.6$ and test for various sizes of VPAs. The results are given in Table 5.3. Similarly, we see that ANTICHAIN performance are more efficient than that of ON-THE-FLY.

Similar to the case of universality checking, experimental results for inclusion checking are summarized in Table 5.4. We ran with 25 samples for each point, setting timeout to 300 seconds. The results show that ANTICHAIN algorithm is again significantly faster than ON-THE-FLY and STANDARD algorithms.

Table 5.2: Universality checking for random VPA with 10 states

	density of final states				
ANTICHAIN	0.2	0.4	0.6	0.8	1.0
universal	1	13	17	16	14
	67.38	69.90	51.67	81.50	87.18
not universal	19	5	0	0	0
	0.74	0.79	–	–	–
timeout (*180* s)	0	2	3	4	6
total time	81.51	1227.76	1418.44	2024	2300.52
	density of final states				
ON-THE-FLY	0.2	0.4	0.6	0.8	1.0
universal	0	0	4	1	0
	–	–	102.19	166	–
not universal	19	5	0	0	0
	0.74	0.75	–	–	–
timeout (*180* s)	1	15	16	19	20
total time	194.08	2703.73	3288.76	3586	3600
	density of final states				
STANDARD	0.2	0.4	0.6	0.8	1.0
universal	–	–	–	–	–
	–	–	–	–	–
not universal	–	–	–	–	–
	–	–	–	–	–
timeout (*180* s)	20	20	20	20	20
total time	3600	3600	3600	3600	3600

5.4 Related Work

The concept of antichains is firstly applied to the universality / inclusion checking to finite automata in [77] and extended to tree automata in [14]. The antichain was also applied to attack the universality / inclusion checking of Buchi automata [24], which is $O(2^{n\ log\ n})$. Since Buchi automata are not determinizable, they used an alternating automata technique, and implemented as ALASKA [78].

The known implementation of VPA VPAlib [55] only works for basic operations such as union, intersection, and determinization. However, in their implementation, determinization is not on-the-fly manner, thus it easily stacks with just few states.

Table 5.3: Universality checking for random VPA with $r = 0.6$

	number of states					
ANTICHAIN	5	10	15	20	25	30
universal	10	10	4	2	0	0
	0.97	85.4	102	90.5	–	–
not universal	7	8	13	10	16	9
	0.86	0.75	6.64	0.30	1.70	6.45
timeout (*180* s)	3	2	3	8	4	11
total time	643	1220	1035	1651	747	2038
	number of states					
ON-THE-FLY	5	10	15	20	25	30
universal	8	1	1	1	0	0
	38	59	23	154	–	–
not universal	7	8	13	10	16	9
	0.86	0.75	16.61	0.90	2.34	21.80
timeout (*180* s)	5	11	6	9	4	11
total time	1211	2045	1319	1864	757	2176

In this chapter we tackled this problem by the antichain idea, and proposed an antichain algorithm for checking universality and inclusion among VPAs. Determinization of a VPA is re-formulated in terms of finite congruences, and by combining with the $\mathcal{P}$ automata techniques [28, 58]. Experiments show that it is much more efficient than the standard approach. We stress that our current implementation does not use an sophisticated data structure like BDD. We anticipate that use of BDD might strive for even greater efficiency.

Table 5.4: Checking inclusion with $r(q, a) = 2$, $f = 0.6$

	size of states	
ANTICHAIN	$[1000 - 5]$	$[3000 - 5]$
included	14	6
	60	110
not included	5	13
	6.60	40.50
timeout (*300* s)	6	6
total time	2677	2987

	size of states	
ON-THE-FLY	$[1000 - 5]$	$[3000 - 5]$
included	6	0
	54	–
not included	5	10
	5.89	37.8
timeout (*300* s)	14	15
total time	4554	4878

Chapter 6

Timed Extensions of Visibly Pushdown Automata

Timed automata (TAs) were introduced by Alur and Dill in [2], and have become a standard modeling formalism for real-time systems. A timed automaton is a finite automaton augmented with a finite set of real-valued clocks, in which constraints on the clocks are used to restrict the behaviors of an automaton. The theory of timed automata allows the solution of certain verification problems for real-time systems [2, 37, 15], e.g., reachability and safety properties. These solutions have been implemented in automatic tools such as UPPAAL [1].

However, the general verification problems (i.e., language inclusion) for timed automata is undecidable. Therefore, for the verification purpose, one has to work either with deterministic specifications or with a restricted class of timed automata which has the required closure properties. One such restricted case is the class of *event-clock automata* (ECAs) [3, 25, 26]. The key feature of these automata is that they have a pair of implicit clocks associated with each input symbol. The event clocks record the time elapsed since the last occurrence of the associated symbol, as well as the time that will elapse before the next occurrence of the associated symbol. When an ECA reads a timed word, clock valuations depend only on the input word itself rather than on the choice of nondeterministic transitions. Hence, ECAs are determinizable and closed under Boolean operations.

During the recent years, there has been much extensive research on the inclusion problem for timed automata [42, 27, 16]. In particular, it was shown that the inclusion problem $L(A) \subseteq L(B)$, for timed automata A and B, becomes *decidable* if B has at most *one clock* [42]. The key idea of the proof is to encode this inclusion problem as the reachability problem for well-structured transition systems. However, over infinite timed words, one clock is enough to make the inclusion problem undecidable [1].

[1] `http://www.uppaal.com/`

A *timed pushdown automaton* (TPDA) [13] is a timed automaton augmented with a pushdown stack. Decision problems for TPDAs such as emptiness is decidable [13]. However, the inclusion problem for TPDAs is undecidable, since the corresponding problem is already undecidable for pushdown automata. One, therefore, has to deal with formalism of less expressive power. One such candidate is the class of *visibly* pushdown automata (VPAs) [5], in which the stack pushes and pops are determined explicitly by an input alphabet. VPAs are closed under all Boolean operations, and the inclusion problem for VPAs is decidable. Motivated by real-time software verification, Emmi and Majumdar [27] introduced *timed* visibly pushdown automata (TVPAs) as the timed extension of VPAs. However, for TVPAs A and B, the inclusion problem $L(A) \subseteq L(B)$ is *undecidable* even when B has exactly *one clock* [27].

Inspired by the ideas of ECAs [3] and VPAs [5], we introduce the class of *event-clock visibly pushdown automata* (ECVPAs). The class of ECVPAs is expressive enough to specify common context-free real-time properties such as "if p holds when a procedure is invoked, then the procedure must return within d time units and q must hold at the return state". Besides, the class of ECVPAs is closed under all Boolean operations. Our results are summarized as follows:

1. We show the essence behind the notion of event clocks is that every ECVPA can be translated into an untimed VPA, which interprets timing constraints symbolically, and vice-versa. Therefore, the closure properties and the decidability results of ECVPAs can be reduced to those of VPAs.

2. We use the translation technique to prove that the inclusion problem $L(A) \subseteq L(B)$ for a TVPA A and an ECVPA B is decidable.

3. We show that class of duration automata (DAs) [65] is a special case of ECVPAs. Thus, the inclusion problem for DAs is decidable.

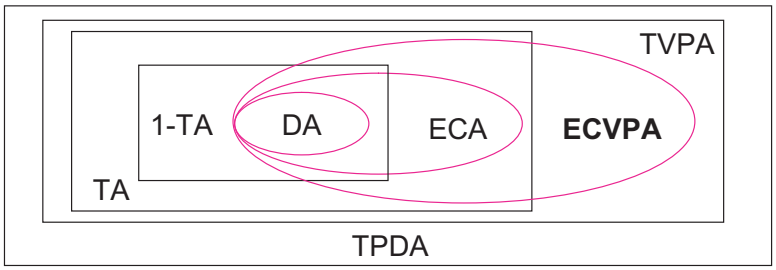

Figure 6.1: Relationships between involved classes of TPDAs

6.1 Event-Clock Visibly Pushdown Automata

6.1.1 Event clocks

In this subsection we give a brief description for *event clocks*. Readers are referred to [3] for more details.

Let R^+ and Q^+ denote the set of non-negative real numbers and non-negative rational numbers, respectively. Given a finite alphabet Σ, a *timed word* $\bar{w}$ over Σ is a finite sequence $(a_0, t_0)(a_1, t_1) \cdots (a_n, t_n)$ of symbols $a_i \in \Sigma$ that are paired with nonnegative real numbers $t_i \in R^+$ such that the sequence $\bar{t} = t_0 t_1 \cdots t_n$ of *time-stamps* is nondecreasing (i.e., $t_i \leq t_{i+1}$ for all $0 \leq i < n$). We denote the timed word $\bar{w}$ by the pair $(\bar{a}, \bar{t})$, where $\bar{a} \in \Sigma^*$ is an *untimed word* over Σ. The set of all finite timed words over Σ is denoted by $T\Sigma^*$. A *timed language* is a set of timed words.

Definition 6.1 (Event Clocks [3])**.** *For each symbol $a \in \Sigma$, we use two implicit clocks x_a (*event-recording*) and y_a (*event-predicting*). Along a timed word, the clock x_a measures the time since the last occurrence of symbol a, and y_a measures the time to the next occurrence of symbol a. If there are no last (resp., next) occurrence of a, the value of x_a (resp., y_a) is* "undefined"*, denoted by $\perp$.*

Remark 6.1. *The notation $\perp$ denotes the "undefined" value, and the* bottom-of-stack *symbol of visibly pushdown automata (defined in the next subsection).*

Let $C_\Sigma = \{x_a | a \in \Sigma\} \cup \{y_a | a \in \Sigma\}$ be the set of event-recording and event-predicting clocks. Define $R^+_\perp = R^+ \cup \{\perp\}$ and $Q^+_\perp = Q^+ \cup \{\perp\}$.

Definition 6.2 (Event-clock valuation [3])**.** *For each timed word $\bar{w} = (a_0, t_0)(a_1, t_1) \cdots (a_n, t_n)$, a* clock valuation *over $\bar{w}$ is a function $\nu^{\bar{w}}_j : C_\Sigma \to R^+_\perp$ which specifies the values of the clocks (in C_Σ) at position j in $\bar{w}$.*

$$\nu^{\bar{w}}_j(x_a) = \begin{cases} t_j - t_i & \text{If } \exists i < j : a_i = a, \textit{and } \forall k : i < k < j \Rightarrow a_k \neq a \\ \perp & \text{otherwise} \end{cases}$$

$$\nu^{\bar{w}}_j(y_a) = \begin{cases} t_{i'} - t_j & \text{If } \exists i' > j : a_{i'} = a, \textit{and } \forall l : j < l < i' \Rightarrow a_l \neq a \\ \perp & \text{otherwise} \end{cases}$$

Figure 6.2, for example, shows the values of $\nu^{\bar{w}}_j$ $(j = 0, 1, 2)$ for the clocks x_a and y_a for the timed word $\bar{w} = (a, 2)(a, 3)(a, 6)(b, 7)(b, 10)$.

Definition 6.3 (Event-clock constraints [3])**.** *1. The* event-clock constraints *compare clock values to $Q^+_\perp$, i.e, to rational constants or to the special value $\perp$. The clock constraints over C_Σ are interpreted with respect to the clock-valuation function ν from C_Σ to $R^+_\perp$: the atom $\perp \leq \perp$ evaluates to* True*, and all other comparisons that involve $\perp$ (e.g., $\perp \geq 3$) evaluate to* False*.*

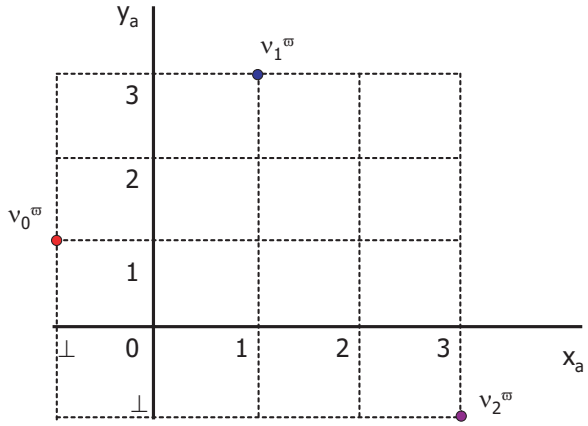

Figure 6.2: Event-clock valuations of x_a and y_a for $\bar{w}$

2. *For simplicity, let* $\Phi(C_\Sigma)$ *denote the set of event-clock constraints over* C_Σ.

3. *For the clock-valuation* ν *and an event-clock constraint* $\varphi \in \Phi_\Sigma$, *we write* $\nu \models \varphi$ *(resp.,* $\nu \not\models \varphi$*) to denote that according to* ν *the constraint* φ *evaluates to* True *(resp.,* False*).*

6.1.2 Event-Clock Visibly Pushdown Automata

A *pushdown alphabet* is a set $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i$ that comprises three *disjoint* finite alphabets in which Σ_c is a finite set of *calls*, Σ_r is a finite set of *returns*, and Σ_i is a finite set of *internal symbols*. We formally define *event-clock visibly pushdown automata* over the pushdown alphabet Σ as follows:

Definition 6.4. *An* event-clock visibly pushdown automaton *(*ECVPA*) on finite timed words over* Σ *is a tuple* $M = \langle Q, \Sigma, Q_0, \Gamma, \Delta, F \rangle$, *where* Q *is a finite set of locations,* $Q_0 \subseteq Q$ *is a finite set of initial locations,* Γ *is a finite stack alphabet that contains a special symbol* $\perp$ *(bottom-of-stack symbol),* $F \subseteq Q$ *is a set of final locations, and* $\Delta = \Delta_c \cup \Delta_r \cup \Delta_i$ *is the transition relation,*

1. $\Delta_c \subseteq Q \times \Sigma_c \times \Phi(C_\Sigma) \times Q \times (\Gamma \backslash \{\perp\})$ *is a push-transition relation*

2. $\Delta_r \subseteq Q \times \Sigma_r \times \Phi(C_\Sigma) \times \Gamma \times Q$ *is a pop-transition relation*

3. $\Delta_i \subseteq Q \times \Sigma_i \times \Phi(C_\Sigma) \times Q$ *is an internal-transition relation.*

The intuition behind the transition relation is briefly explained as follows:

- $(q, a, \varphi, q', \gamma) \in \Delta_c$ is a push-transition, where on reading a when the clock valuation satisfies φ, the symbol γ is pushed onto the stack and the location changes to q'.

- $(q, a, \varphi, \gamma, q') \in \Delta_r$ is a pop-transition, where on reading a when the clock valuation satisfies φ, γ is popped from the stack, the location q changes to q' (if $\gamma = \perp$, it is read but not popped).

- $(q, a, \varphi, q') \in \Delta_i$ is an internal-transition, where the location, on reading a when the clock valuation satisfies φ, the location changes from q to q' without stack operations.

A stack is a nonempty finite sequence from the set $St = \{w\perp \mid w \in (\Gamma \setminus \{\perp\})^*\}$ starting with the top symbol on the left, and ending with the symbol $\perp$ on the right. The *empty stack* is the one that only contains the symbol $\perp$.

Definition 6.5. *A* configuration *of an ECVPA M is a pair (q, σ) where $q \in Q$, and $\sigma \in St$. For a timed word $\bar{w} = (a_0, t_0) \cdots (a_n, t_n)$, a run of M on $\bar{w}$ is a sequence of configurations $\rho = (q_0, \sigma_0) \cdots (q_{n+1}, \sigma_{n+1})$, where $q_i \in Q$, $\sigma_i \in St$, $q_0 \in Q_0, \sigma_0 = \perp$, and for every $0 \leq i \leq n$ one of the following condition holds:*

- **Push:** *If a_i is a call symbol, then for some $\gamma \in \Gamma$, $(q_i, a_i, \varphi_i, q_{i+1}, \gamma) \in \Delta_c$, $\nu_i^{\bar{w}} \models \varphi_i$, and $\sigma_{i+1} = \gamma.\sigma_i$.*

- **Pop:** *If a_i is a return symbol, then for some $\gamma \in \Gamma$, $(q_i, a_i, \varphi_i, \gamma, q_{i+1}) \in \Delta_r$, $\nu_i^{\bar{w}} \models \varphi_i$, and either $\gamma \in \Gamma$ and $\sigma_i = \gamma.\sigma_{i+1}$, or $\gamma = \sigma_i = \sigma_{i+1} = \perp$.*

- **Internal:** *If a_i is an internal symbol, then $(q_i, a_i, \varphi_i, q_{i+1}) \in \Delta_i$, $\nu_i^{\bar{w}} \models \varphi_i$, and $\sigma_{i+1} = \sigma_i$.*

A run ρ is an *accepting* run if it ends in a final location. A timed word $\bar{w}$ is an accepting word if there is an accepting run of M on $\bar{w}$. The language of an ECVPA M, denoted by $L(M)$, is the set of all accepting timed words $\bar{w}$ of M.

Remark 6.2. *An (untimed) visibly pushdown automaton [5] can be seen as an event-clock visibly pushdown automaton that has no clock constraints on transitions. In the rest of this paper, we mention a VPA as an ECVPA without $\Phi(C_\Sigma)$ component in the transitions. Note also that a VPA is* deterministic *if $|Q_0| = 1$ and, for each configuration (q, σ) and $a \in \Sigma$, there are at most one transition from (q, σ) by a. VPAs are determinizable and closed under Boolean operations [5]. In particular, for a nondeterministic VPA with n states, one can construct an equivalent deterministic VPA with $O(2^{n^2})$ states and $O(2^{n^2}.|\Sigma_c|)$ stack symbols. The inclusion problem for VPAs is* EXPTIME*-complete [5].*

We next present some examples of event-clock visibly pushdown automata.

Example 6.1. *It is easy to see that an ECA [3] is an ECVPA that has only internal symbols, i.e, $\Sigma_c = \Sigma_r = \varnothing$. Thus, the class ECAs is a subclass of ECVPAs.*

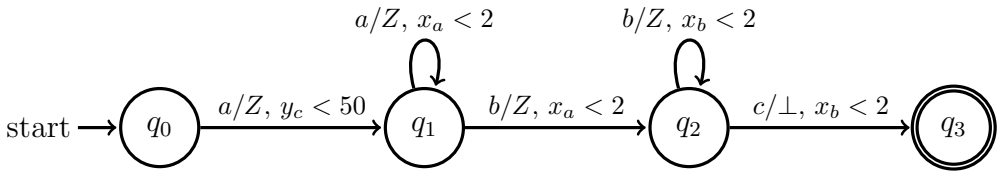

Figure 6.3: Event-clock visibly pushdown automaton M

Example 6.2. Duration automata *(DAs) were studied in [65] for modeling simple component-based real-time systems. A DA is a finite automaton in which each transition must occur in an associated time interval. A duration automaton can be viewed as an one-clock timed automata, where the clock is reset at each transition. The clock valuations of a DA are also explicitly determined by the input timed word. Therefore, the class of DAs can be seen as a special subclass of ECAs, and thus DA is a subclass of ECVPAs.*

Example 6.3. *Let a be a push. Let b, c be pops, and Z be a stack symbol. The ECVPA M of Figure 6.3 uses two event-recording clocks x_a and x_b, and an event-predicting clock y_c. The transitions of M are given as follows:*

- **Push:** $(q_0, a, y_c < 50, q_1, Z)$, $(q_1, a, x_a < 2, q_1, Z)$.
- **Pop:** $(q_1, b, x_a < 2, Z, q_2)$, $(q_2, b, x_b < 2, Z, q_2)$, $(q_2, c, x_b < 2, \perp, q_3)$.

We describe locations of M as nodes of a graph. We adopt the following conventions to represent edges: for instance, a push-transition (q_i, a, ϕ, q_j, Z) is labeled as $a/+Z, \phi$; a pop-transition (q_i, b, ϕ, Z, q_j) is labeled as $b/-Z, \phi$.

The clock constraint $y_c < 50$ that is associated with the edge from q_0 to q_1 ensures that c occurs within 50 time units of the first a. The constraint $x_a < 2$ that is associated with the edge from q_1 to q_2 makes sure that the first b occurs within 2 time units of the last a.

The automaton M accepts the set of input timed words: $L(M) = \{(\bar{\alpha}, \bar{t}) \mid \bar{\alpha} = a^n b^n c, n \in N^+, t_{i+1} < t_i + 2. \forall (1 \leq i \leq 2n); t_{2n+1} - t_1 < 50\}$. This timed language, however, cannot be accepted by any timed automaton [2]. This is because the untimed part $\{a^n b^n c \mid n \in N^+\}$ is a context-free language.

The next example shows that ECVPA and timed automata are incomparable.

Example 6.4. *Consider the timed language:*

$$L = \{(a^n, \bar{t}) \mid n \geq 2, t_j - t_i = 1, \textit{for some } 0 \leq i < j < n\}$$

The language L can be accepted by a nondeterministic one-clock timed automaton (1-TA) A in Figure 6.4. This language, however, cannot be accepted by any ECVPA.

Definition 6.6. An ECVPA $M = \langle Q, \Sigma, Q_0, \Gamma, \delta, F \rangle$, is *deterministic* if $|Q_0| \leq 1$ and for every $q \in Q$, and for every clock valuation ν:

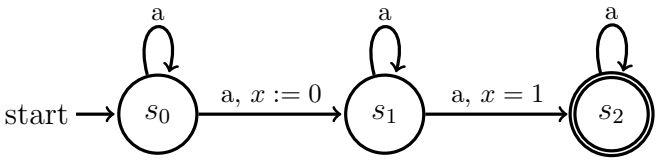

Figure 6.4: One clock timed automaton A

- if $(q, a, \varphi_1, q_1) \in \Delta_i$ and $(q, a, \varphi_2, q_2) \in \Delta_i$, then $\nu \not\models \varphi_1 \wedge \varphi_2$.
- if $(q, b, \varphi_1, q_1, \gamma_1) \in \Delta_c$ and $(q, b, \varphi_2, q_2, \gamma_2) \in \Delta_c$, then $\nu \not\models \varphi_1 \wedge \varphi_2$.
- if $(q, c, \varphi_1, \gamma, q_1) \in \Delta_r$ and $(q, c, \varphi_2, \gamma, q_2) \in \Delta_r$, then $\nu \not\models \varphi_1 \wedge \varphi_2$.

The determinism condition ensures that at each step during a run, the choice of the next transition is uniquely determined by the current location of the ECVPA, the input word, the current stack content, and the current clock-valuation of the ECVPA along the input word. It is easy to check that every deterministic ECVPA has at most one run over any given timed input word.

6.2 Properties of Event-Clock Visibly Pushdown Automata

6.2.1 Untimed/Timed Translation between ECVPA and VPA

Similar to the case for event clock automata [25], we show in this section that an arbitrary ECVPA can be translated into an untimed VPA that interprets timing constraints symbolically and exhibits the same behaviors, and vice-versa.

In particular, for a given ECVPA M, let $B = \{r_0, r_1, ..., r_n\}$ be a finite set of constants appearing in the clock constraints of M. Without loss of generality, let us assume that $0 = r_0 < r_1 < \cdots < r_n$. We define $Intv = \{[\bot, \bot]\} \cup \{[r_i, r_i], (r_i, r_{i+1}) \mid 0 \leq i < n\} \cup \{[r_n, r_n], (r_n, \infty)\}$.

Definition 6.7. *An* interval-based alphabet *over* Σ *is the set* $\Pi = \Sigma \times Intv^{|C_\Sigma|}$. *We have* $|\Pi| = |\Sigma| \times |Intv|^{|C_\Sigma|} = |\Sigma| \times (2r_n + 1)^{|C_\Sigma|}$.

Elements of an interval alphabet are of the form (a, g) with $a \in \Sigma$ and $g : C_\Sigma \to Intv$. The component g is called *guard*, and it is used to represent the timing constraint: $\bigwedge_{x \in C_\Sigma} x \in g(x)$.

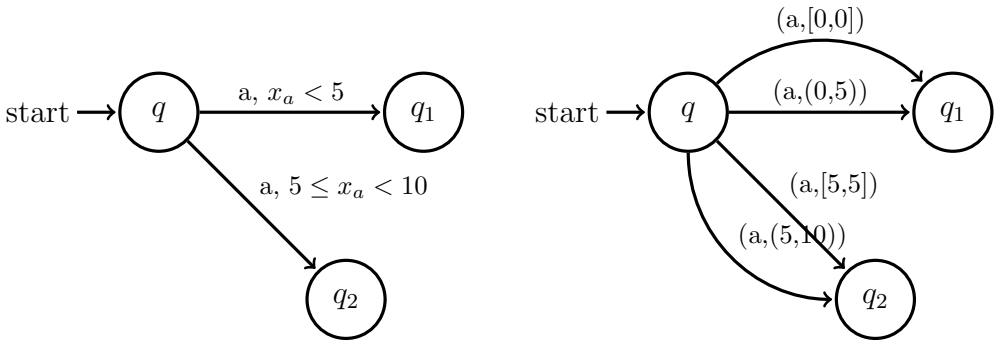

Figure 6.5: Description of untimed translation

Definition 6.8. *Define a function* $tw : \Pi^* \to 2^{T\Sigma^*}$ *where for each* $\alpha = (a_0, g_0) \cdots (a_n, g_n) \in \Pi^*$*, we have:*

$$tw(\alpha) = \{\bar{w} \mid \bar{w} = (a_0, t_0) \cdots (a_n, t_n), \forall x \in C_\Sigma, \forall i (1 \leq i \leq n) : \nu_i^{\bar{w}}(x) \in g_i(x)\}.$$

$tw(\alpha)$ is the set of timed words that "satisfies" α.

The *untimed translation* technique is formalized in the next definition:

Definition 6.9 (Untimed Transformation)**.** *Let* $M = \langle Q, \Sigma, Q_0, \Gamma, \Delta, F\rangle$ *be an ECVPA. We define a VPA* $ut(M) = \langle Q, \Pi, Q_0, \Gamma, \Delta', F\rangle$ *in which for each transition e of M with the input symbol a and the clock constraint φ, there exists a natural number k such that e is translated to transitions of $ut(M)$ as follows:*

- *the interval input symbols are* $(a, g_i) \in \Pi$*,* $i = 1 \cdots k$*, and*
- φ *is equivalent to* $\bigvee_{i=1..k}(\bigwedge_{x \in C_\Sigma} x \in g_i(x))$*.*

Example 6.5. *Consider a transition* $e = (q, a, x_a < 5, q_1)$ *and* $e' = (q, a, 5 \leq x_a < 10, q_2)$ *of M. We have* $Intv = \{[0,0], (0,5), [5,5], (5,10), [10,10], (10,\infty), [\perp, \perp]\}$*. The transition e will be translated to parallel transitions in $ut(M)$ as below:*

$$(q, (a, [0,0]), q_1), \quad (q, (a, (0,5)), q_1), \quad (q, (a, [5,5]), q_2), \quad (q, (a, (5,10)), q_2).$$

The untimed translation is described in Figure 6.5. Note that this translation preserves determinism. Similar to the case for event-clock automata [25], the next lemma holds:

Lemma 6.1. $tw(L(ut(M))) = L(M)$ *for all ECVPAs M. Moreover, if M is a deterministic ECVPA, then $ut(M)$ is a deterministic VPA.*

The reverse of the translation is described in the next definition.

Definition 6.10 (Timed Transformation)**.** *Let* $N = \langle Q, \Pi, Q_0, \Gamma, \Delta', F\rangle$ *be a VPA. We define an ECVPA* $ec(N) = \langle Q, \Sigma, Q_0, \Gamma, \Delta, F\rangle$ *such that each transition of N with the interval input symbol (a, g) is translated to a transition of $ec(N)$ whose input symbol is a and the clock constraint is* $\varphi = \bigwedge_{x \in C_\Sigma} x \in g(x)$*.*

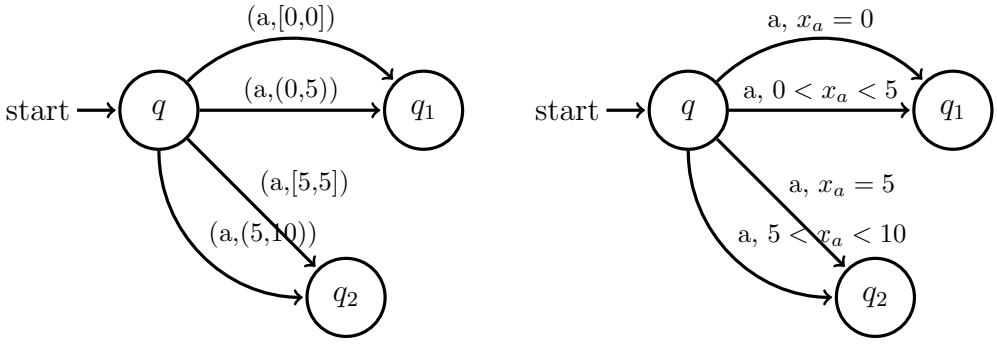

Figure 6.6: Description of timed translation

Example 6.6 (Continued from Example 6.5)**.** *Now, suppose that we want to translate the VPA $ut(M)$ in Example 6.5 to an ECVPA. The transitions of $ut(M)$ are translated back to the following transitions:*

$$(q, a, x_a = 0, q_1), (q, a, 0 < x_a < 5, q_1), (q, a, x_a = 5, q_2), (q, a, 5 < x_a < 10, q_2).$$

Observe that this translation also preserves determinism.

For the timed translation, we get the following lemma:

Lemma 6.2. *$L(ec(N)) = tw(L(N))$ for all VPA N. Moreover, if N is deterministic VPA, then $ec(N)$ is a deterministic ECVPA.*

Proof.

- For $\alpha = (a_0, g_0) \cdots (a_n, g_n) \in L(N)$, let $\rho = (q_0, \sigma_0) \cdots (q_{n+1}, \sigma_{n+1})$ be a run of N on α. If $\bar{w} = (a_0, t_0) \cdots (a_n, t_n) \in tw(\alpha)$, then ρ is also a run of $ec(N)$ on $\bar{w}$. Thus, $tw(L(N)) \subseteq L(ec(N))$.
- Conversely, let $\bar{w} = (a_0, t_0) \cdots (a_n, t_n) \in L(ec(N))$. There is an accepting run $\rho = (q_0, \sigma_0) \cdots (q_{n+1}, \sigma_{n+1})$ of $ec(N)$ on $\bar{w}$, $q_n \in F$. Based on the timed translation, there is an untimed word $\alpha = (a_0, g_0) \cdots (a_n, g_n) \in \Pi^*$ such that $\bar{w} \in tw(\alpha)$, and ρ is also a run of N on α. Thus, $L(ec(N)) \subseteq tw(L(N))$.

□

The next theorem immediately follows from Lemmas 6.1 and 6.2.

Theorem 6.3. *$L(ec(ut(M))) = L(M)$ for all ECVPAs M.*

6.2.2 Closure Properties and Inclusion Problem

From Theorem 6.3 and the decidability results of VPAs [5], the following theorems hold:

Lemma 6.4 (Determinization)**.** *For any nondeterministic ECVPA M, there is a deterministic ECVPA $Det(M)$ such that $L(Det(M)) = L(M)$. Moreover, if M has n locations, we can construct $Det(M)$ with $O(2^{n^2})$ locations and $O(2^{n^2} \cdot |\Sigma_c| \cdot (2r)^{2|\Sigma|})$ stack symbols, where r is the largest constant appearing in the clock constraints of M. The set of clocks of $Det(M)$ coincides with that of M.*

Theorem 6.5 (Closure properties)**.** *The class of ECVPAs is closed under union, intersection, and complementation.*

Theorem 6.6 (Language Inclusion)**.** *The inclusion problem for ECVPAs is* EXPTIME*-complete.*

Proof. Consider two ECVPAs A and B such that each automaton has at most n locations, let m be the size of the input alphabet. Let c be the largest integer constant that appears in the clock constraints. To check whether $L(A) \subseteq L(B)$, we first untimed translate B to a VPA B_1, determinize B_1 to B_2, and then timed translate B_2 to an ECVPA B'. The automaton B' has 2^{n^2+n} locations. Let M be the product of A and B'. The ECVPA M has $n.2^{n^2+n}$ locations, and the integer constants that appear in the clock constraints of M are also bounded by c. Now, we can construct a VPA $ut(M)$, and check for its emptiness. Since checking emptiness of VPA is cubic time proportional to its size, it follows that emptiness of $ut(M)$ can be checked in EXPTIME. The proof of hardness is the same as the corresponding proof for VPA [5].

□

Remark 6.3. *Büchi VPAs are closed under union, intersection, and complementation [5]. By using a similar technique, we also can translate Büchi ECVPA to Büchi VPA, and vice-versa. Therefore, the result of Theorem 6.6 can be extended to the case of Büchi ECVPA.*

6.3 Related Classes of Timed Pushdown Automata

6.3.1 Timed Visibly Pushdown Automata

Let $X = \{x_1, ..., x_n\}$ be a finite set of *clocks*. Define the set $\Phi(X)$ of clock constraints over X by the grammar:

$$\varphi ::= \top \mid x \bowtie c \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2,$$

where $c \in Q^+$, $x \in X$, $\bowtie \ \in \{<, \leq, \geq, >\}$.

For the set of clocks X, a *clock valuation* is a function $\nu : X \to R^+$ which describes the values of each clock $x \in X$ at an instant. For the clock valuation ν and a clock constraint φ, we write $\nu \models \varphi$ to denote that ν satisfies the constraint φ. Given a set of clocks $\lambda \subseteq X$ and a clock valuation ν, let $\nu \downarrow \lambda$ be a clock valuation defined as follows:

$$(\nu \downarrow \lambda)(x) = \begin{cases} 0 \text{ when } x \in \lambda \\ \nu(x) \text{ otherwise} \end{cases} \tag{6.1}$$

Given a clock valuation ν and a time $t \in R^+$, define $(\nu + t)(x) = \nu(x) + t$.

Definition 6.11 (Timed Visibly Pushdown Automaton [27])**.** *A* timed visibly pushdown automaton *(*TVPA*) over* pushdown alphabet Σ *is a tuple* $M = (Q, \Sigma, Q_0, \Gamma, X, \delta, F)$*, where* Q *is a finite set of locations,* $Q_0 \subseteq Q$ *is a finite set of initial locations,* Γ *is a finite stack alphabet that contains* $\perp$*,* $F \subseteq Q$ *is a set of final locations,* X *is a finite set of clocks, and* $\delta = \delta_c \cup \delta_i \cup \delta_r$ *is the transition relation:*

1. $\delta_c \subseteq Q \times \Sigma_c \times \Phi(X) \times Q \times (\Gamma \setminus \{\perp\}) \times 2^X$ *is the push-transition relation*
2. $\delta_r \subseteq Q \times \Sigma_r \times \Gamma \times \Phi(X) \times Q \times 2^X$ *is the pop-transition relation*
3. $\delta_i \subseteq Q \times \Sigma_i \times \Phi(X) \times Q \times 2^X$ *is the internal-transition relation.*

Let ν be a clock valuation. We briefly explain the intuition behind the transitions of a TVPA as follows:

- A *push-transition* $(q, a, \phi, q', \gamma, \lambda)$ is a move on the (call) input symbol a from q to q' where ν satisfies ϕ, the clock valuation is updated from ν to $\nu \downarrow \lambda$, and γ is pushed on the stack.
- A *pop-transition* $(q, a, \gamma, \phi, q', \lambda)$ is a move on the (return) input symbol a and stack symbol γ, from q to q' where ϕ is satisfied and ν is updated to $\nu \downarrow \lambda$, and γ is popped from the stack (if the top of stack is $\perp$, then it is read but not popped).
- An *internal-transition* $(q, a, \phi, q', \lambda) \in \delta$ at clock valuation ν is a move on the (internal) input symbol a from the location q to q' such that $\nu \models \phi$ and the resulting clock valuation $\nu' = \nu \downarrow \lambda$.

Remark 6.4. *There are clock reset conditions* λ *in the transitions of TVPAs. Thus, unlike the case of ECVPAs, the clock valuations of TVPAs not only depend on input timed words but also on transitions of TVPAs.*

Definition 6.12. *A* configuration *of a TVPA* M *is a triple* (q, ν, σ)*, where* $q \in Q$*,* $\sigma \in St$*, and* ν *is a clock valuation. Given a timed word* $\bar{w} = (a_0, t_0) \cdots (a_n, t_n)$*, a* run *of* M *on* $\bar{w}$ *is a sequence of configurations* $\rho = (q_0, \nu_0, \sigma_0) \xrightarrow{(a_0, t_0)} (q_1, \nu_1, \sigma_1) \cdots \xrightarrow{(a_n, t_n)} (q_{n+1}, \nu_{n+1}, \sigma_{n+1})$*, where* $q_0 \in Q_0$*,* $\sigma_0 = \perp$*, and for every* $1 \leq i \leq n$*, one of the following conditions holds:*

1. **Push:** $(q_i, a_i, \phi_i, q_{i+1}, \gamma, \lambda) \in \delta$*,* $\nu_i \models \phi_i$*,* $\nu_{i+1} = (\nu_i \downarrow \lambda) + (t_{i+1} - t_i)$*, and* $\sigma_{i+1} = \gamma\sigma_i$

2. **Pop:** $(q_i, a_i, \gamma, \phi_i, q_{i+1}, \lambda) \in \delta$, $\nu_i \models \phi_i$, $\nu_{i+1} = (\nu_i \downarrow \lambda) + (t_{i+1} - t_i)$, *and* $\gamma\sigma_{i+1} = \sigma_i$ *or* $\gamma = \sigma_{i+1} = \sigma_i = \perp$,

3. **Internal:** $(q_i, a_i, \phi_i, q_{i+1}, \lambda) \in \delta$, $\nu_i \models \phi_i$, $\nu_{i+1} = (\nu_i \downarrow \lambda) + (t_{i+1} - t_i)$, *and* $\sigma_{i+1} = \sigma_i$

A run ρ is an *accepting* run if it ends in a final state. A timed word $\bar{w}$ is *accepting* if there is an accepting run of M on $\bar{w}$. The language $L(M)$ is the set of timed words accepted by M. It is easy to see that TVPAs is a subclass of *timed pushdown automata* (*TPDA*) [13], and a superclass of timed automata [2]. Indeed, a TVPA is a *timed automaton* if $\Sigma_c = \Sigma_r = \varnothing$. Unlike ECVPA, TVPA are not determinizable and not closed under complementation. Moreover, the next theorem was proved by Emmi and Majumdar [27].

Theorem 6.7 ([27, Theorem 3])**.** *The inclusion problem $L(A) \subseteq L(B)$, where A is a TVPA and B is a TVPA with at least* one clock*, is undecidable.*

6.3.2 Translation from ECVPA to TVPA

As shown in [3], every ECA can be translated into a timed automaton that accepts the same timed language. There the basic idea of the translation can be described as follows:

Definition 6.13 (Translating event-clocks to original clocks [3])**.** *An event-recording clock x_a can be seen as an original clock that is reset on a transition e if the input symbol of e is a. For event-predicting clocks, consider a given ECA A and the set of all* atomic event-predicting clock constraints *(denoted by Φ_A) of the form $y_a = \perp$ or $y_a \sim c$, where $\sim \in \{\leq, <, >, \geq\}$. Define a nondeterministic timed automaton B as follows:*

- *The states of the target timed automaton B are the pairs (q, Ψ) with $q \in Q_A$ and $\Psi \subseteq \Phi_A$.*
- *The state (q, Ψ) is an initial state of B iff q is the initial state of A and Ψ does not contain a constraint of the form $y_a \sim c$.*
- *The state (q, Ψ) is a final state of B iff $q \in F_A$ and $\Psi = \{y_a = \perp\}$.*
- *For each $\psi \in \Phi_A$, B has a clock z_ψ.*
- *The automaton B has an edge from the source state (q, Ψ) to the target state (l', Ψ') with the input symbol a, the clock constraint φ, and the reset condition ρ if and only if the following seven conditions are satisfied. Intuitively, a prediction $y_b \sim c$, along a transition in A, on the time difference to the next occurrence of a is replaced in B by a constraint on the clock $z_{(y_b \sim c)}$: the clock $z_{(y_b \sim c)}$ is reset when the prediction is performed, and its value is checked by the constraint $z_{(y_b \sim c)} \sim c$ when the next b occurs.*

1. *The automaton A has an edge of the form (l, l', a, χ).*
2. *The constraint $y_a = \perp$ does not appear in Ψ.*
3. *The constraint φ is the conjunction of all atomic clock constraints of the form $(z_{(y_a \sim c)} \sim c)$ with $(y_a \sim c) \in \Psi$.*
4. *For each input symbol b different from a, if a constraint involving y_b appears in Ψ, then it appears in Ψ' also.*
5. *Each conjunct of χ appears in Ψ' also.*
6. *For each input symbol b and for $\sim$ equal to $>$ or $\geq$, the clock $z_{(y_b \sim c)}$ appears in the reset condition ρ iff the constraint $y_b \sim c$ is a conjunct of χ.*
7. *For each input symbol b and for $\sim$ equal to $<$ or $\leq$, the clock $z_{(y_b \sim c)}$ appears in the reset condition ρ iff the constraint $y_b \sim c$ is a conjunct of χ, and either $b = a$ or the constraint $y_b \sim c$ does not appear in Ψ.*

Then the timed automaton B defines the timed language $L(A)$.

Similarly, by using the above translation of the event clocks to the original clocks, the next lemma holds:

Lemma 6.8. *Every ECVPA can be translated into a TVPA that accepts the same timed language.*

We now arrive at the main theorem of this chapter:

Theorem 6.9 (Language Inclusion)**.** *The inclusion problem $L(A) \subseteq L(B)$, where A is a TVPA and B is an ECVPA, is decidable.*

Proof. $L(A) \subseteq L(B) \iff L(A) \cap \overline{L(B)} = \varnothing$. We first determinize B and then compute its complement $\overline{B}$. Second, translate the ECVPA $\overline{B}$ into a TVPA B'. Note that $L(B') = \overline{L(B)}$. Third, take the intersection of A and B', and check for emptiness of $L(A \cap B')$. Similar to Theorem 6.6, the complexity of this inclusion checking is EXPTIME-complete. □

6.4 Related Work

Ouaknine and Worrell [42] proved that when B has at most one clock, the inclusion problem $L(A) \subseteq L(B)$ can be encoded as the reachability problem for well-structured transition systems [31]. Thus, the inclusion problem for timed automata becomes decidable when B has at most one clock. However, over infinite timed words, one clock is enough to make the inclusion problem undecidable [1], since this problem can be reduced from the space-bounded recurrent-state problem for alternating channel machines.

The closest related works is the paper of Emmi and Majumdar [27]. There they extended the proof technique of [42], and showed that the inclusion problem $L(A) \subseteq L(B)$ is decidable if A is a TPDA, and B is a TA with at most one clock. However, for TVPAs A and B, the inclusion problem is undecidable even B has exactly one clock [27]. In this paper, we have shown that when A is a TVPA and B is an ECVPA, the inclusion problem $L(A) \subseteq L(B)$ become decidable.

A decidable subclass of real-time logic *so-called* EventClockTL, which corresponds to event-clock automata, was presented in [37]. Furthermore, DSouza [25] showed that the class of event-clock automata admit a logical characterization via a monadic second order logic interpreted over timed words. The proof technique is based on the untimed translation, as in Definition 6.9, that transform an event-clock automaton to a finite automaton.

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 considered the inclusion problems for subclasses of PDAs. These problems can be seen as model checking context-free properties for pushdown models. The main contributions of the thesis are:

- In Chapter 3, we presented an improvement on the alternate stacking technique used in Greibach-Friedman's proof of the language inclusion problem $L(A) \subseteq L(B)$, where A is a PDA and B is an SPDA. The original construction encodes everything as stack symbols (in an intricate way), whereas our refinement gives a more direct product construction, and clarifies how alternate stacking works. For our construction, a proof of "liveness" is not needed, and the whole correctness proof for the decision procedure became simpler.

- In Chapter 4, we refuted the claim about the determinizability of 2-OVPAs (2-VPAs) of Carotenuto et al. [17]. In addition, we have introduced the class of visibly stack automata (VSAs) and showed that this class of automata is either not determinizable.

- The universality and inclusion problems for VPA were already known to be EXPTIME-complete [5], but no implementation for these solutions was available. In Chapter 5, we have provided new antichain-based algorithms for these problems. Although the standard approaches (as well as ours) have the same worst case complexity, our prototype implementation outperforms those approaches where determinization is explicit.

- In Chapter 6, we introduced the class of ECVPAs by combining the ideas of ECAs [3] and VPAs [5]. We showed that the class of ECVPAs enjoys good closure properties and decidability results. We also showed that the inclusion problem $L(A) \subseteq L(B)$, where A is a TVPA and B is an ECVPA, is decidable. This provides an algorithm for checking if a TVPA meets a specification that is given as an ECVPA. We hope that the class of ECVPA will be useful for verification of recursive real-time programs.

7.2 Further Research

We have presented antichain-based algorithms for checking universality and inclusion of visibly pushdown automata. This work suggests a number of possibilities for future research.

Implementation with BDD-Based Representations

In our current implementation, the data structures for VPAs are not well optimized. That is the reason why the running time of our tool is not fast. A BDD (Binary Decision Diagram) is a graph representing a Boolean function. The BDD representation has some extremely convenient properties, such as compactness and canonicity, and it allows efficient manipulation. BDDs have successfully been used in a long range of verification techniques, e.g., in automata-theoretic tools such as MONA [1] and ALASKA [2]. It would be interesting to manipulate VPAs using BDD technique. We expect that with BDD-based representation, the running time of our tool will be significantly improved.

Validation of XML Documents

XML is nowadays the de facto standard for electronic data interchange on the Web. An XML document is a text-based linear encoding of tree-structured data. Markups in the text in terms of open- and close-tags are used to create a bracketing structure on the document, and captures the hierarchy of information in the tree. The study of XML has naturally concentrated on its tree representation; for example, document types are represented using the parse-trees generated by DTDs and EDTDs. Also, XML query languages like XPath have modalities like parent and child that refer to the tree edges, and tree automata (over unranked trees) have been used to model and solve decision problems for XML (see [44, 40, 10], for example).

Recent works [50, 46, 53] have shown that the class of VPA is a promising and natural formalism for modeling and validating XML streams. In particular, XML streaming and validating problems can be reduced to the decision problems of VPAs. However, no

[1] `http://www.brics.dk/mona/`

[2] `http://www.antichains.be/alaska/`

implementations for those algorithms are available. It would be interesting to apply our prototype tool to a static analysis tool for XML streams. Moreover, Benedikt et al. [10] recently defined, investigated, and proposed solutions for the XML stream fire-walling problem. Their underlying model are based on finite automata, thus their techniques can only handle non-recursive DTDs with respect to XPath specifications. We expect that their techniques can be extended to deal with the recursive DTDs and wider specifications, if the models are VPAs instead of finite automata.

Bibliography

[1] P. A. Abdulla, J. Deneux, J. Ouaknine, and J. Worrell. Decidability and complexity results for timed automata via channel machines. Proceedings of the *32nd International Colloquium on Automata, Languages, and Programming (ICALP'05)*, LNCS 3580, pp. 1089-1101, Springer-Verlag 2005.

[2] A. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126: pp. 183-235, 1994.

[3] R. Alur, L. Fix, and T. A. Henzinger. Event-clock automata: A determinizable class of timed automata. *Theor. Comp. Sci*, 211: pp.253-273, 1999. A preliminary version appeared in *CAV'94*, LNCS 818, pp. 1-13, Springer-Verlag 1994.

[4] R. Alur and P. Madhusudan. Decision problems for timed automata: a survey. In *School of Formal Method*, LNCS 3185, pp. 1-24, Springer-Verlag 2004.

[5] R. Alur and P. Madhusudan. Visibly pushdown languages. Proceedings of the *36th ACM Symposium on Theory of Computing (STOC'04)*, pp. 202-211, ACM Press 2004.

[6] R. Alur, K. Etessami, P. Madhusudan. A temporal logic of nested calls and returns. Proceedings of the *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, LNCS 2988, pp. 467-481, Springer-Verlag 2004.

[7] R. Alur, V. Kumar, P. Madhusudan, and M. Viswanathan. Congruences for visibly pushdown languages. Proceedings of the *32nd International Colloquium on Automata, Languages, and Programming (ICALP'05)*, LNCS 3580, pp. 1102-1114, Springer-Verlag 2005.

[8] R. Alur, S. Chaudhuri, and P. Madhusudan. A fixpoint calculus for local and global program flows. Proceedings of the *33rd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'06)*, pp. 153-165, ACM Press 2006.

[9] J. M. Autebert, J. Berstel, and L. Boasson: *Context-free languages and push-down automata*. Handbook of Formal Languages, Springer-Verlag 1997.

[10] M. Benedikt, A. Jeffrey, and R. Ley-Wild. Stream firewalling of XML constraints. Proceedings of the *28th ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*, pp. 487-498, ACM Press 2008.

[11] M. Benedikt, W. Fan, and Floris Geerts. XPath satisfiability in the presence of DTDs. *Journal of the ACM*. Vol. 55(2), ACM Press 2008.

[12] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zue. *Bounded Model Checking*. Chapter 3: Advances in Computers, pp. 118-149, Academic Press 2003.

[13] A. Bouajjani, R. Echahed, and R. Robbana. On the automatic verification of systems with continuous variables and unbounded discrete data structures. In *Hybrid Systems II*, LNCS 999, pp. 64-85, Springer-Verlag 1995.

[14] A. Bouajjani, P. Habermehl, L. Holik, T. Touili, and T. Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. Proceeding of the *13th International Conference on Implementation and Application of Automata (CIAA'08)*, LNCS 5148, pp. 57-67, Springer-Verlag 2008.

[15] P. Bouyer and F. Laroussinie. Model checking timed automata. In *Modeling and Verification of Real-Time Systems*, pp. 111-140. ISTE Ltd. - John Wiley & Sons, Ltd., 2008.

[16] P. Bouyer, Kim G. Larsen, and N. Markey. Model checking one-clock priced timed automata. Logical Methods in Computer Science 4, pp. 14-57, 2008.

[17] D. Carotenuto, A. Murano, and A. Peron. 2-Visibly pushdown automata. Proceedings of the *11th International Conference Developments in Language Theory (DLT'07)*, LNCS 4588, pp. 132-144, Springer-Verlag 2007.

[18] D. Caucal. Synchronization of pushdown automata. Proceedings of the *11th International Conference Developments in Language Theory (DLT'07)*, LNCS 4036, pp. 120-132, Springer-Verlag 2006.

[19] H. Comon-Lundh, F. Jacquemard, and N. Perrin. Tree automata with memory, visibility and structural constraints. Proceedings of the *10th International Conference on Foundations of Software Science and Computation Structure (FoSSaCS'07)*, LNCS 4423, pp. 168-182, Springer-Verlag 2007.

[20] E. M. Clarke and E. A. Emerson. The design and synthesis of synchronization skeletons using temporal logic. Proceedings of the *Second Workshop on Logics of Programs*, LNCS 131, pp. 52-71, IBM Watson Research Center, New York, May 1981.

[21] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[22] Z. Dang. Pushown time automata: a binary reachability characterization and safety verification. *Theor. Comp. Sci*, 302(1-3): pp. 93-121, 2003.

[23] Z. Dang, T. Bultan, O. H. Ibarra, and R. A. Kemmerer. Past pushdown timed automata and safety verification. *Theor. Comp. Sci*, 313(1): pp. 57-71, 2004.

[24] L. Doyen and J. F. Raskin. Improved algorithms for the automata-based approach to model checking. Proceedings of the *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, LNCS 4424, pp. 451-465, Springer-Verlag 2007.

[25] D. D'Souza. A logical characterisation of event clock automata. *International Journal of Foundation for Computer Science*, Vol. 14, No. 4, pp. 625-640, 2003.

[26] D. D'Souza and N. Tabareau. On timed automata with input-determined guards. In *FORMATS/FTRTFT'04*, LNCS 3253, pp. 68-83, Springer-Verlag 2004.

[27] M. Emmi and R. Majumdar. Decision problems for the verification of real-time software. Proceedings of the *9th International Workshop on Hybrid Systems : Computation and Control (HSCC'06)*, LNCS 3927, pp. 200-211, Springer-Verlag 2006.

[28] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. Proceedings of the *12th International Conference on Computer Aided Verification (CAV 2000)*, LNCS 1855, pp. 232-247, Springer-Verlag 2000.

[29] J. Esparza, A. Kucera, and S. S. Schwoon. Model checking LTL with regular valuations for pushdown systems. *Information and Computation*, 186(2), pp.355-376, 2003.

[30] A. Finkel. B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. Proceedings of the *2nd International Workshop on Verification of Infinite State Systems (Infinity'97)*, ENTCS, Vol. 9, 1997.

[31] A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere!. *Theor. Comp. Sci*, 256(1-2), pp. 63-92, 2001.

[32] E. P. Friedman and S. Greibach. Superdeterministic PDAs. The method of accepting does affect decision problems. *Journal of Systems and Computer Science*, 19(3), pp. 79-117, 1979.

[33] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. Proceedings of the *15th International Symposium on Protocol Specification, Testing, and Verification (PSTV95)*, pp. 3-18, Chapman-Hall, 1995.

[34] S. Greibach and E. P. Friedman. Subperdeterministic PDAs: A Subcase with a decidable inclusion problem. *Journal of the ACM*, 27(4), pp. 675-700, 1980.

[35] S. Ginsburg, S. Greibach, and M. Harrison. One-way stack automata. *Journal of the ACM*, 14(2), pp. 381-418, 1967.

[36] D. Harel and D. Raz. Deciding mmptiness for stack automata on infinite trees. *Information and Computation*, 113(2), pp. 278-299, 1994.

[37] T. A. Henzinger, J. Raskin, and P. Schobbens. The regular real-time languages. In Proceedings of the *25th International Colloquium on Automata, Languages and Programming (ICALP'98)*, LNCS 1443, pp. 580-598, Springer-Verlag 1998.

[38] G. J. Holzmann. The SPIN model checker: primer and reference manual. Addison-Wesley Professional, 2003.

[39] H. B. Hunt. On the complexity of finite, pushdown, and stack automata. *Mathematical Systems Theory*, Vol. 10, 1976.

[40] D. Olteanu. *Evaluation of XPath Queries against XML streams*. PhD Thesis, Institute for Informatics, Ludwig Maximilian University of Munich, 2004.

[41] M. Ogawa, N. V. Tang, and N. Hirokawa. Antichains for visibly pushdown automata. Submitted to *an International Conference*.

[42] J. Ouaknine and J. Worrell. On the language inclusion problem for timed automata: Closing a decidability gap. Proceedings of the *19th IEEE Symposium Logic in Computer Science (LICS'04)*, pp. 54-63, IEEE Computer Society 2004.

[43] R. McNaughton. Parenthesis grammars. *Journal of the ACM*, 14(3): 490-500, 1967.

[44] F. Neven. Automata, logic, and XML. Proceedings of the *11th International Conference on Computer Science Logic (CSL'02)*, LNCS 2471, pp. 2-26, Springer-Verlag 2002.

[45] D. Nowotka and J. Srba. Height-deterministic pushdown automata. Proceedings of the *32nd International Symposium on Mathematical Foundations of Computer Science (MFCS'07)*, LNCS 4708, pp. 125-134, Springer-Verlag 2007.

[46] V. Kumar, P. Madhusudan, and M. Viswanathan. Visibly pushdown automata for streaming XML. Proceedings of the *16th International World Wide Web Conference (WWW'07)*, pp. 1053-1062, ACM Press 2007.

[47] O. Kupferman, N. Piterman, and M. Vardi. Pushdown specifications. Proceedings of the *9th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'02)*, LNCS 2514, pp. 262-277, Springer-Verlag 2002.

[48] O. Kupferman and M. Y. Vardi. Weak alternating automata are not that weak. Proceedings of the *ACM TOCL'01*, 2(3): 408-429, 2001.

[49] O. Kupferman. Avoiding Determinization. Proceedings of the *21st IEEE Symposium Logic in Computer Science (LICS'06)*, pp. 243-254, IEEE Computer Society 2006.

[50] C. Pitcher. Visibly pushdown expression effects for XML stream processing. Proceedings of the *Workshop on Programming Language Technologies for XML (PLAN-X'05)*, pp. 5-19, 2005.

[51] N. Piterman and M. Vardi. Micro-macro stack systems: A new frontier of elementary decidability for sequential systems. Proceedings of the *18th IEEE Symposium Logic in Computer Science (LICS'03)*, pp.381-390, IEEE Computer Society 2003.

[52] C. Löding, P. Madhusudan, and O. Serre. Visibly pushdown games. Proceedings of the *24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04)*, LNCS 3328, pp. 408-420, Springer-Verlag 2004.

[53] P. Madhusudan and M. Viswanathan. Query automata for nested words. Under submission.

[54] Y. Minamide and A. Tozawa. XML validation for context-rree grammars. Proceedings of the *4th ASIAN Symposium on Programming Languages and Systems (APLAS'06)*, LNCS 4279, pp. 357-373, Springer-Verlag 2006.

[55] D. H. Nguyen and M. Sudholt. VPA-based aspects: better support for AOP over protocols. Proceedings of the *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pp.167-176, IEEE Computer Society 2006.

[56] A. Pnueli. The temporal logic of programs. Proceedings of the *18th IEEE Symposium on Foundations of Computer Science (FOCS'77)*, pp. 46-57, IEEE Computer Society 1977.

[57] J. Raskin and P. Schobbens. State clock logic: a decidable real-time logic. In *Hybrid and Real-time Systems*, LNCS 1201, pp. 33-47, Springer-Verlag 1997.

[58] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, Vol. 58, pp. 206-263, 2005.

[59] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, LNCS 137, pp. 337-351, 1982.

[60] S. Safra. On the complexity of ω-automata. Proceedings of the *29th IEEE Symposium on Foundations of Computer Science (FOCS'88)*, pp. 319-327, IEEE Computer Society 1988.

[61] G. Sénizergues. $L(A) = L(B)$? A simplified decidability proof. *Theore. Comp. Sci*, 281(2), pp. 555-608, 2002.

[62] A. Sistla, E. M. Clarke, N. Francez, and Y. Gurevich. Can message buffers be axiomatized in linear temporal logic. *Information and Control*, Vol. 63(1-2), pp. 88-112, 1984.

[63] C. Stirling. Deciding DPDA equivalence is primitive recursive. Proceedings of the *29th International Colloquium on Automata, Languages and Programming (ICALP'02)*, LNCS 2380, pp. 821-832, Springer-Verlag 2002.

[64] D. Tabakov and M. Y. Vardi. Experimental evaluation of classical automata constructions. Proceedings of the *12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05)*, LNCS 3835, pp. 396-411, Springer-Verlag 2005.

[65] N. V. Tang, D. V. Hung, and M. Ogawa. Modeling urgency in component-based real-time systems. In *Advances in Computer Sciences (ASIAN'06)*, LNCS 4435, pp. 248-255, Springer-Verlag 2007.

[66] N. V. Tang and M. Ogawa. Alternate stacking revisited: Inclusion problem of superdeterministic pushdown automata. *IPSJ Transactions on Programming*, PRO 37, Vol. 1(1), pp. 1-11, June 2008.

[67] N. V. Tang and M. Ogawa. Event-clock visibly pushdown automata. Proceedings of the *35th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'09)*, LNCS 5404, pp. 558-569, Springer-Verlag 2009.

[68] N. V. Tang and M. Ogawa. Determinization for extensions of VPAs Seems Difficult. *Submitted to an International Journal.*

[69] S. L. Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. Proceedings of the *22nd IEEE Symposium on Logic in Computer Science (LICS'07)*, pp. 161-170, IEEE Computer Society 2007.

[70] A. Tozawa and Y. Minamide. Complexity results on balanced context-free languages. Proceedings of the *10th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS'07)*, LNCS 4423, pp. 346-360, Springer-Verlag 2007.

[71] Alan M. Turing. On computable numbers, with an application to the Entscheidungproblem. *Proc. London Math. Society,* 2(42), pp. 230-265, 1936.

[72] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addition-Wesley, 1979.

[73] L. G. Valiant. The equivalence problem for finite-turn pushdown automata. *Information and Control*, Vol. 25: 123-133, 1974.

[74] L. G. Valiant. *Decision procedures for families of deterministic pushdown automata*. PhD Thesis. University of Warwick, 1973.

[75] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. Proceedings of the *1st IEEE Symposium Logic in Computer Science (LICS'86)*, pp. 332-344, IEEE Computer Society 1986.

[76] I. Yaki, Y. Takata, and H. Seki. A static analysis using tree automata for XML access control. Proceedings of the *3rd International Symposium on Automated Technology for Verification and Analysis (ATVA'05)*, LNCS 3707, pp. 234-247, Springer-Verlag 2005.

[77] M. De Wulf, L. Doyen, T. A. Henzinger, and J. F. Raskin. Antichains: A new algorithm for checking universality of finite automata. Proceedings of the *18th International Conference on Computer Aided Verification (CAV'06)*, LNCS 4144, pp. 17-30, Springer-Verlag 2006.

[78] M. De Wulf, L. Doyen, J. F. Raskin, and N. Maquet. Alaska: Antichains of logic, automata and symbolic Kripke structures analysis. Proceedings of the *6th International Symposium on Automated Technology for Verification and Analysis (ATVA'08)*, LNCS 5311, pp. 240-245, Springer-Verlag 2008.

Publications

1. Nguyen Van Tang and Mizuhito Ogawa. Event-Clock Visibly Pushdown Automata. In the *Proceedings of the 35th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'09)*, Špindlerüv Mlýn, Czech Republic. LNCS 5404, pp. 558-569, Springer-Verlag, 2009.

2. Nguyen Van Tang and Mizuhito Ogawa. An Improvement on Decision Procedure for Inclusion Problem of Superdeterministic Pushdown Automata. *IPSJ Transactions on Programming*, Vol. 1(1), pp. 36-46, June 2008.

3. Koichi Kobayashi, Nguyen Van Tang, and Kunihiko Hiraishi. Precomputation Based Approximate Algorithm for Model Predictive Control of Hybrid Systems. In the *Proceedings of the 23rd International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC'08)*, pp. 913-916, July 2008.

4. Nguyen Van Tang, Dang Van Hung, and Mizuhito Ogawa. Modeling Urgency in Component-Based Real-time Systems. In the *Proceedings of the 11th Annual Asian Computing Science Conference (ASIAN'06)*, LNCS 4435, pp. 249-256, Tokyo, Japan, 2006.

5. Mizuhito Ogawa, Nguyen Van Tang, and Nao Hirokawa. Antichains for Visibly Pushdown Automata. *Under submission.*

6. Nguyen Van Tang and Mizuhito Ogawa. Determinization for Extensions of VPA Seems Difficult. *Under submission.*

Index