Antichain algorithm, its theory and applications

By Pakorn Techaveerapong

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Mizuhito Ogawa

September, 2013

Antichain algorithm, its theory and applications

By Pakorn Techaveerapong (1110202)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Mizuhito Ogawa

and approved by
Professor Mizuhito Ogawa
Associate Professor Nao Hirokawa
Associate Professor Kazuhiro Ogata

August, 2013 (Submitted)

Abstract

The antichain algorithm and bounded model checking (BMC) are well-known techniques in finite model checking framework. However, they are less explored for pushdown model checking. In this thesis, we combine them together in order to check universality of visibly pushdown automata (VPAs). VPAs, introduced by Alur and Madhusudan in 2004, is a subclass of pushdown automata whose stack behavior is determined by the partition of input alphabet. VPAs have been applied to be useful in various contexts, e.g., processing XML documents and checking context-sensitive properties. Due to its high complexity, ExpTime-complete, the universality of VPAs is a challenge. We propose two approaches to check universality of VPAs. We implement the proposed approaches in a prototype tool and conduct experiments on randomly generated VPAs. Although the experimental results show that our algorithms are inferior to the other algorithms, we can prove a huge contribution of antichain algorithm in BMC.

Acknowledgments

The author wishes to express his sincere gratitude to his principal advisor Professor Mizuhito Ogawa of Japan Advanced Institute of Science and Technology for giving the author the opportunity and constant support and for his invaluable guidances and suggestions throughout this work.

The author would like to thank his co-advisor Associate Professor Nao Hirokawa of Japan Advanced Institute of Science and Technology for his helpful discussions and suggestions.

The author is grateful to his senior Mr.Suphat Korkiatithaweechai for his helpful assists and suggestions.

The author also wishes to express honest thanks to his family and friends for giving a lot of encouragements and supports.

The author devotes his sincere thanks and appreciation to all of them, and his colleagues. My thanks are also given to all of those whose names have not been mentioned.

Contents

Chapter 1

Introduction

For a finite partially ordered set, an antichain in P is a set of pairwise incomparable elements (i.e., no two different elements in subset of P are related). Antichains are also called Sperner systems in literature [18].

In automata theory, antichains are used to get smaller objects to manipulate and to reduce the number of computation by reducing an explicit determinization step. Recently, antichains have been successfully applied to many decision problems related to automata: universality and inclusion for finite word automata [19, 22], for Büchi automata [21], and for non-deterministic bottom-up tree automata [13], In this research project, we are focusing on uses of antichain for universality checking of visibly pushdown automata [37].

Visibly pushdown automata (VPAs) [5], aka nested word automata [6], are pushdown automata whose stack behavior (i.e., whether to push, pop, or no stack operation) is determined by an input symbol according to a fixed partition of input alphabet. This class of visibly pushdown automata enjoys many good properties similar to those of the class of finite automata. It is closed under all boolean operations. The main reason is that, each non-deterministic VPA can be transformed into an equivalent deterministic one. As a result, VPAs appear to be useful in many contexts of model-checking framework ,e.g., as automaton models for processing XML streams [33, 30], as AOP protocols for component-based systems [32], and as semantics of programming languages [2, 31] and verification [3].

To check universality of a non-deterministic VPA M over the input alphabet Σ (i.e., to check if $L(M) = \Sigma^*$) is EXPTIME-complete. The standard method faces a hardness due to the expensive determinization process because determinization for VPA requires exponential blowup [5].

There are many attempts to tackle the universality problem of VPA. Nguyen [38] proposed a better-than-standard algorithm. This algorithm simultaneously performs an on-demand determinization and reachability checking by $\mathcal{P}$-automaton [12, 27]. Later Nguyen and Ohsaki [37] improved it by introducing antichains over transitions of $\mathcal{P}$-automata, to generate the smallest number of reachable configurations. Bruyere, et al., [14] also proposed an universality checking algorithm with antichains but their algorithm is alternative to Nguyen's. They do not use the regularity of the set of reachable configurations but they observe on the bottom-up unranked trees. In [28], the authors provide a solution for checking universality of VPAs over finite and infinite words. They avoid the determinization and complementation steps, and use Ramsey-based universality checking algorithm. Their algorithm do not seem to use antichains.

Bounded model checking (BMC) [9, 16] is a symbolic counter-example-finding method that searches for a violation of a given property in a model structure. BMC approach looks for counter-examples of bounded length. If no counter-examples are found, enlarge a bound and repeat the search again. Theoretically, BMC may have high-level complexity, but by combining SAT-encoding and recent SAT-solvers, it reduces the space complexity a lot. In practice, BMC often outperforms the BDD-based symbolic model checking [15, 10].

An important issue of BMC is to make it *complete*. Because at such a bounded length, absence of counter-example is still unclear. From this reason, many attempts have been tried to turn BMC into a complete method with the ability to guarantee the absence of counter-example [9, 17]. Typically, the reachability diameter (length of longest shortest path) and the reachability recurrence diameter (length of longest loop-free path) are used to be the completeness threshold of many properties [29].

There is an example of using BMC in model-checking of pushdown automata. In [8], Basler et al. proposed an algorithm for checking reachability of pushdown systems using BMC to compute universal summaries [7].

The original contribution of this research consists in defining two BMC algorithms for checking universality of VPAs. We call them BMCi and BMCp. These algorithms are combinations of antichain and BMC by using antichain to reduce the size of search space, and the number of variables and clauses of propositional formulae.

We ran tests on randomly generated VPAs. Although, the experimental results have shown that our algorithms are still far from the original antichain algorithm in [37], we can see a huge improvement when compare between BMC with antichain and BMC without antichain.

In Chapter 2 we recall basic notions and properties of VPAs, and $\mathcal{P}$-automata technique and correctness proof. Chapter 3 presents an antichain algorithm for checking universality of VPA. Furthermore, the correctness proof of the algorithm is also given. We give some approaches of BMC for VPA and analyze possibilities in Chapter 4. Based on Chapter 4, our BMC approaches for universality checking of VPAs are proposed in Chapter 5. We implement and carry out some relevant experiments, the experimental results and observation are presented in Chapter 6. We discuss some related works in Chapter 7. Finally, we conclude the research and give suggestions for future works in Chapter 8.

Chapter 2

Preliminaries

2.1 Visibly pushdown automata (VPA)

Let Σ be the finite input alphabet, and let $\Sigma = \Sigma_c \cup \Sigma_i \cup \Sigma_r$ be a partition of Σ where Σ_c, Σ_i and Σ_r are the disjoint finite set of call (push) alphabets, internal alphabets and return (pop) alphabets, respectively. Visibly pushdown automata are formally defined as follows:

Definition 1 (VPA. [5])**.** *A visibly pushdown automata (VPA) M over input alphabet Σ is a tuple $(Q, Q_0, \Gamma, F, \Delta)$ where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, Γ is a finite stack symbols with a special symbol $\perp$ (so-called bottom of stack), $F \subseteq Q$ is a set of final states, and $\Delta = \Delta_c \cup \Delta_i \cup \Delta_r$ is the transition relation where $\Delta_c \subseteq Q \times \Sigma_c \times Q \times (\Gamma \setminus \{\perp\})$, $\Delta_i \subseteq Q \times \Sigma_i \times Q$ and $\Delta_r \subseteq Q \times \Sigma_r \times \Gamma \times Q$*

- A transition $(q, c, q', \gamma) \in \Delta_c$ where $c \in \Sigma_c$ and $\gamma \neq \perp$, is a push-transition where on reading c regardless of stack, state changes from q to q' and γ is pushed onto the stack; we denote a transition by $q \xrightarrow{c/+\gamma} q'$.
- A transition $(q, i, q') \in \Delta_i$ where $i \in \Sigma_i$, is a internal-transition where on reading i regardless of stack, state changes from q to q'; we denote a transition by $q \xrightarrow{i} q'$.
- A transition $(q, r, \gamma, q') \in \Delta_r$ where $r \in \Sigma_r$, is a pop-transition where on reading input r and top of stack γ, state changes from q to q' and γ is popped (if top is $\perp$, read but not popped); we denote a transition by $q \xrightarrow{r/-\gamma} q'$ (or $q \xrightarrow{r/\perp} q'$).

A *stack* is a non-empty finite string over Γ ending with the bottom-of-stack symbol $\perp$. We write St for the set of all stacks. St is denoted as $St = \{w\perp \mid w \in (\Gamma \setminus \{\perp\})^*\}$. A configuration is a pair (q, σ) such that $q \in Q$ and $\sigma \in St$. The transition relation of VPA

can be used to define how the configuration of the machine changes in a single step: we say $(q, \sigma) \overset{a}{\hookrightarrow} (q', \sigma')$ if one of the following holds:

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

Acceptance. For a word $w = a_1 \dots a_n$ in Σ^*, a run of M on w is a sequence of configurations $(q_0, \sigma_0) \overset{a_1}{\hookrightarrow} (q_1, \sigma_1) \overset{a_2}{\hookrightarrow} \dots \overset{a_n}{\hookrightarrow} (q_n, \sigma_n)$ where each $q_i \in Q, \sigma_i \in St, q_0 \in Q_0$ and $\sigma_0 = \bot$, and is denoted by $(q_0, \sigma_0) \overset{w}{\hookrightarrow}{}^* (q_n, \sigma_n)$. A run $(q_0, \sigma_0) \overset{w}{\hookrightarrow}{}^* (q_n, \sigma_n)$ is accepting if the last state is a final state, i.e. if $q_n \in F$. A word $w \in \Sigma^*$ is accepted by a VPA M if there is an accepting run of M on w. The language of M, $L(M)$, 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 $L = L(M)$.

Definition 2 (deterministic VPA. [5])**.** *A VPA M is deterministic if $|Q_0| = 1$ and for every state $q \in Q$:*

- *for every $a \in \Sigma_c$, there is at most one stack symbol $\gamma \in \Gamma$ and at most one transition of the form $(q, a, q', \gamma) \in \Delta_c$*
- *for every $a \in \Sigma_i$, there is at most one transition of the form $(q, a, q') \in \Delta_i$*
- *for every $a \in \Sigma_r$ and $\gamma \in \Gamma$, there is at most one transition of the form $(q, a, \gamma, q') \in \Delta_r$*

As shown in [5], any non-deterministic VPA can be transformed into an equivalent deterministic one. The main idea behind the construction is to do a subset construction but store the call transitions and simulate them later, that is at the time of the corresponding pop transition. The construction has two components: a set of *summary edges* S, that keeps track of what transitions are possible from the last push transition to the corresponding pop transition, and a set of *path edges* R, that keeps track of all possible states reached from initial states (see [5] for more detail). However, this construction contains unneccesary information, that is the set of summaries S may contain pairs that are not truly reached from initial states. In [38], Nguyen has modified the algorithm to make every pair (S, R) satisfy $\Pi_2(S) = R$ without disturbing the correctness of the algorithm (where $\Pi_2(S) = \{s' \mid (s, s') \in S\}$ is the projection on the second component). After this modification, the construction is able to keep information as small as possible, that is every pair in S keeps only reachable states and the component R is no longer needed.

Algorithm 1: Determinization for VPA

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

Result: An equivalent determinized VPA $M^d = (Q', Q'_0, \Gamma', F', \Delta')$

begin

$Q' = 2^{Q \times Q}$

$Q'_0 = \{Id_{Q_0}\}$

$\Gamma' = Q' \times \Sigma_c$

$F' = \{S \in Q' \mid \Pi_2(S) \cap F \neq \emptyset\}$

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

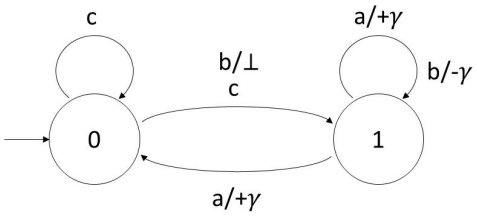
- **Push:** For every $a \in \Sigma_c, S \xrightarrow{a/+(S,a)} Id_{R'} \in \Delta'_c$ where
  $R' = \{q' \mid \exists q \in \Pi_2(S) : q \xrightarrow{a/+\gamma} q' \in \Delta_c\}$
- **Internal:** For every $a \in \Sigma_i, S \xrightarrow{a} S' \in \Delta'_i$ where
  $S' = \{(q, q') \mid \exists q'' \in Q : (q, q'') \in S, q'' \xrightarrow{a} q' \in \Delta_i\}$
- **Pop:** For every $a \in \Sigma_r$,
  - if stack is empty: $S \xrightarrow{a/\perp} S' \in \Delta'_r$ where
    $S' = \{(q, q') \mid \exists q'' \in Q : (q, q'') \in S, q'' \xrightarrow{a/\perp} q' \in \Delta_r\}$
  - if stack is not empty: $S \xrightarrow{a/-(S',a')} S'' \in \Delta'_r$ where
    $update = \{(q, q') \mid \exists q_1, q_2 \in Q : (q_1, q_2) \in S, q \xrightarrow{a'/+\gamma} q_1 \in \Delta_c, q_2 \xrightarrow{a/-\gamma} q' \in \Delta_r\}$
    $S'' = \{(q_3, q') \mid \exists q \in Q : (q_3, q) \in S', (q, q') \in update\}$

end

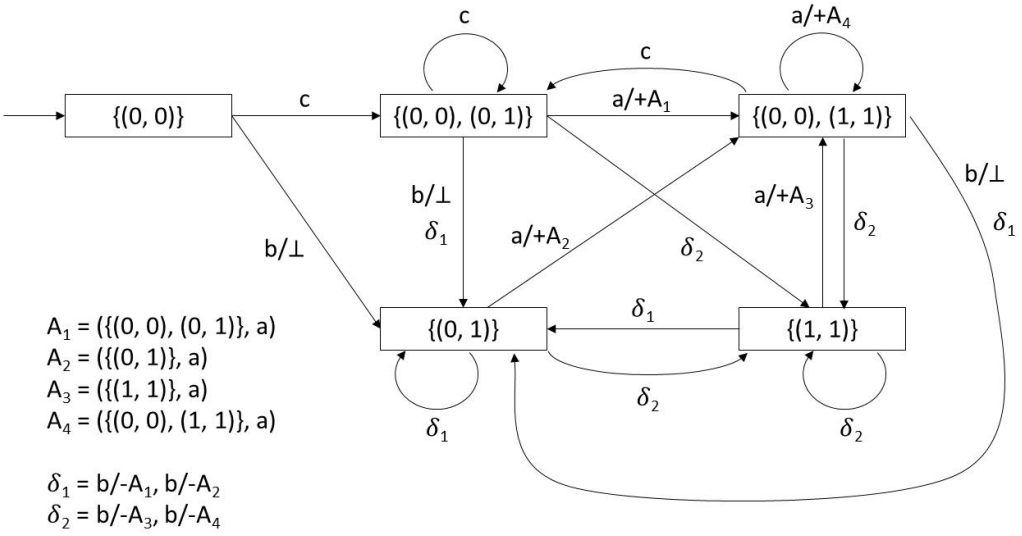
For a finite set X, let $Id_X = \{(q, q) \mid q \in X\}$. Let $M = (Q, Q_0, \Gamma, F, \Delta)$ be any non-deterministic VPA. Let M^d denotes an equivalent deterministic VPA of M such that $L(M) = L(M^d)$. We will construct M^d as in Algorithm 1. Remark that from this point, every time we mention about determinization, it refers to this Algorithm 1.

Theorem 1. *([38]) For any VPA M over Σ, there is a deterministic VPA M^d over Σ such that $L(M^d) = L(M)$. Moreover, if M has n states, we can construct M^d with at most 2^{n^2} states and with stack symbols of size at most $2^{n^2} \cdot |\Sigma_c|$.*

Example 1. *We illustrate an example of determinization in Figure 2.1 where $\Sigma_c = \{a\}$, $\Sigma_r = \{b\}$ and $\Sigma_i = \{c\}$. The unreachable states are omitted, for example, state $\{(1, 0)\}$ and $\{(0, 1), (1, 0)\}$. Note that in M^d some transitions actually are not possible but can be generated from the construction. For example, the transition $\{(0, 0), (1, 1)\} \xrightarrow{b/\perp} \{(0, 1)\}$ would never happen because the configuration $(\{(0, 0), (1, 1)\}, \perp)$ is not reachable.*



(a) A non-deterministic VPA M



(b) A deterministic VPA M^d of M

Figure 2.1: An example of determinization of VPA

2.2 $\mathcal{P}$-automata

The purpose of $\mathcal{P}$-automaton is to keep track of reachable configurations of pushdown system. Bouajjani *et al.* [12] have introduced an efficient symbolic method to compute reachable configurations of a pushdown system. The key of their technique is to use a finite automaton so-called $\mathcal{P}$-automaton to encode a set of infinite configurations of a pushdown system. $\mathcal{P}$-automata are classified into $Post^*$-automata and Pre^*-automata, but we only use $Post^*$-automata. So, all $\mathcal{P}$-automata mentioned in this work are $Post^*$-automata.

Our definition of $\mathcal{P}$-automaton here has been adjusted so that the concepts are easily related to VPA, without disturbing the essential points in [12],

Definition 3 ($\mathcal{P}$-automata)**.** *For a VPA $M = (Q, Q_0, \Gamma, F, \Delta)$ over Σ, a finite automaton $\mathcal{A} = (P, \Gamma, P_0, \{f_{\mathcal{A}}\}, \nabla)$ is a $\mathcal{P}$-automaton for M if*

– *$\mathcal{A}$ uses stack alphabet Γ as the input alphabet*

– $P \subseteq Q \cup \{f_{\mathcal{A}}\}$ *is a finite set of states*

– $P_0 \subseteq Q$ *is a set of (multiple) initial states*

– *the final state is only an unique state* $f_{\mathcal{A}}$

– $\nabla \subseteq (Q \times (\Gamma \setminus \{\bot\}) \times Q) \cup (Q \times \{\bot\} \times \{f_{\mathcal{A}}\})$ *is the set of transitions*

We write $p \overset{\gamma}{\mapsto} p'$ for $(p, \gamma, p') \in \nabla$ and $\mapsto^*$ for the reflexive transitive closure of $\mapsto$. A $\mathcal{P}$-automaton $\mathcal{A}$ accepts a configuration (q, σ) if $q \overset{\sigma}{\mapsto}{}^* f_{\mathcal{A}}$. A set of configurations accepted by $\mathcal{A}$ is denoted by $Conf(\mathcal{A})$.

Definition 4. *Let* $\mathcal{A}$ *and* $\mathcal{A}'$ *be* $\mathcal{P}$*-automata of VPA* M*,*

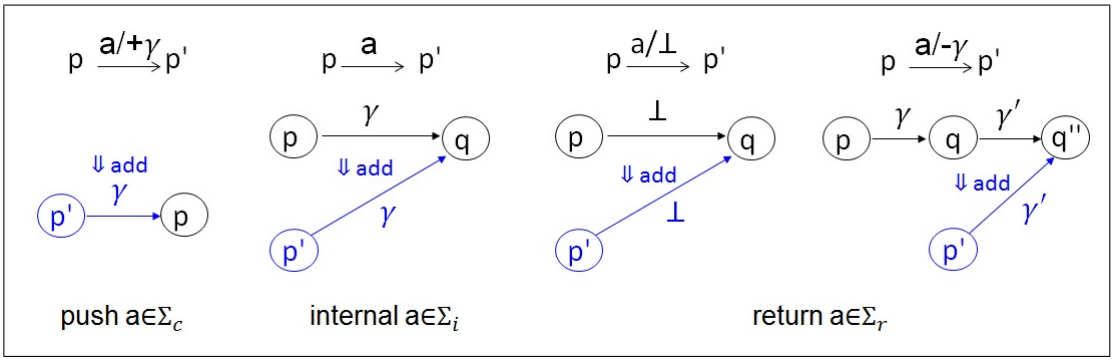
1. *we say* $\mathcal{A} \Rightarrow \mathcal{A}'$ *if* $\mathcal{A}'$ *is obtained from* $\mathcal{A}$ *by one of the following saturation rules:*

$$[c]\ \frac{(P, \Gamma, P_0, \{f_{\mathcal{A}}\}, \nabla)}{(P \cup \{q'\}, \Gamma \cup \{a\}, P_0 \cup \{q'\}, \{f_{\mathcal{A}}\}, \nabla \cup (q', \gamma, q))} \quad \text{if } q \xrightarrow{a/+\gamma} q' \in \Delta_c$$

$$[i]\ \frac{(P, \Gamma, P_0, \{f_{\mathcal{A}}\}, \nabla)}{(P \cup \{q'\}, \Gamma, P_0 \cup \{q'\}, \{f_{\mathcal{A}}\}, \nabla \cup (q', a', q''))} \quad \text{if } q \xrightarrow{a} q' \in \Delta_i \text{ and } q \xrightarrow{a'} q'' \in \nabla$$

$$[r_0]\ \frac{(P, \Gamma, P_0, \{f_{\mathcal{A}}\}, \nabla)}{(P \cup \{q'\}, \Gamma, P_0 \cup \{q'\}, \{f_{\mathcal{A}}\}, \nabla \cup (q', \bot, f_{\mathcal{A}}))} \quad \text{if } q \xrightarrow{a/\bot} q' \in \Delta_r \text{ and } q \xrightarrow{\bot} f_{\mathcal{A}} \in \nabla$$

$$[r]\ \frac{(P, \Gamma, P_0, \{f_{\mathcal{A}}\}, \nabla)}{(P \cup \{q'\}, \Gamma, P_0 \cup \{q'\}, \{f_{\mathcal{A}}\}, \nabla \cup (q', \gamma', q'''))} \quad \text{if } q \xrightarrow{a/-\gamma} q' \in \Delta_r \text{ and } q \xrightarrow{\gamma} q'' \xrightarrow{\gamma'} q''' \in \nabla$$



2. *A* $\mathcal{P}$*-automaton* $\mathcal{A}$ *is saturated if* $\mathcal{A} = \mathcal{A}'$ *whenever* $\mathcal{A} \Rightarrow \mathcal{A}'$*.*

Let $\Rightarrow^*$ denotes the reflexive transitive closure of $\Rightarrow$. Remark that from definition above a saturated $\mathcal{P}$-automaton with $\mathcal{A} \Rightarrow^* \mathcal{A}'$ is always uniquely determined and we denote it by $Post^*(\mathcal{A})$.

For VPA $M = (Q, Q_0, \Gamma, F, \Delta)$, let $\mathcal{C}_0$ be a set of initial configurations of M (i.e., $\mathcal{C}_0 = \{(q_0, \bot) \mid q_0 \in Q_0\}$), $\mathcal{A}_0$ denotes an initial $\mathcal{P}$-automaton that accepting all configurations in $\mathcal{C}_0$. Then $\mathcal{A}_0 = (Q_0 \cup \{f_{\mathcal{A}}\}, \Gamma, Q_0, \{f_{\mathcal{A}}\}, \nabla_0)$ where $\nabla_0 = \{(q_0, \bot, f_{\mathcal{A}}) \mid q_0 \in Q_0\}$.

Let $post^*(\mathcal{C}_0)$ denotes all configurations reachable from $\mathcal{C}_0$ that is set of all reachable configurations of M (i.e., $post^*(\mathcal{C}_0) = \{(q,\sigma) \mid \exists q_0 \in Q_0, \exists w \in \Sigma^*.(q_0,\bot) \overset{w}{\hookrightarrow}^* (q,\sigma)\}$). Given $\mathcal{C}_0$, we can compute $post^*(\mathcal{C}_0)$ by constructing $Post^*(\mathcal{A}_0)$ following Definition 4.

Lemma 1. *Let $M = (Q, Q_0, \Gamma, F, \Delta)$ be a VPA, and let $\mathcal{A}_0$ be a $\mathcal{P}$-automaton accepting $\mathcal{C}_0$. Assume that $p \overset{\sigma}{\mapsto}^* q$ in $Post^*(\mathcal{A}_0)$ and $p \in Q$.*

1. *If $q \in Q$, $(q,\epsilon) \hookrightarrow^* (p,\sigma)$.*
2. *If $q = f_{\mathcal{A}}$, there exists $p' \overset{\sigma'}{\mapsto}^* f_{\mathcal{A}}$ in $\mathcal{A}_0$ and $(p',\sigma') \hookrightarrow^* (p,\sigma)$.*

Proof. Assume the saturation procedure proceeds $\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \ldots$. Then $p \overset{\sigma}{\mapsto}^* q$ is in $\mathcal{A}_i$ for some i. We will prove by induction on steps of i. For $i = 0$, $\mathcal{A}_0 = (Q_0 \cup \{f_{\mathcal{A}}\}, \Gamma, Q_0, \{f_{\mathcal{A}}\}, \nabla_0)$ where $\nabla_0 = \{(q_0, \bot, f_{\mathcal{A}}) \mid q_0 \in Q_0\}$. Thus, only $q = f_{\mathcal{A}}$ and the second statement holds with $p' = p$ and $\sigma' = \sigma$. Assume the above statements hold for every $p \overset{\sigma}{\mapsto}^* q$ in $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$ is constructed by adding new transitions (denoted by $\mapsto_{i+1}$) by one of the saturation rules in Definition 4. We will prove for $p \overset{\sigma}{\mapsto}^* q$ in $\mathcal{A}_{i+1}$.

Assume that $p \overset{\sigma}{\mapsto}^* q$ contains $\mapsto_{i+1}$ k-times, we prove by (nested) induction on k. If $k = 0$, obviously the lemma holds from induction hypothesis. Let $k > 0$ and the leftmost occurence of $\mapsto_{i+1}$ in $p \overset{\sigma}{\mapsto}^* q$ be $p_0 \overset{\delta}{\mapsto} p_0'$. Thus,

$$p \overset{\sigma_1}{\mapsto}^* p_0 \overset{\delta}{\mapsto} p_0' \overset{\sigma_2}{\mapsto}^* q$$

where $\sigma = \sigma_1\delta\sigma_2$ and $p_0' \overset{\sigma_2}{\mapsto}^* q$ contains $\mapsto_{i+1}$ at most $k-1$ times.

We will prove only the first statement of the lemma. The second statement follows similarly. We have four cases according to which rule is applied when construct $\mathcal{A}_i \Rightarrow \mathcal{A}_{i+1}$.

1. Rule $[c]$ is applied.
2. Rule $[i]$ is applied.
3. Rule $[r_0]$ is applied.
4. Rule $[r]$ is applied.

By (both) induction hypothesis on $p_0' \overset{\sigma_2}{\mapsto}^* q$ and $p \overset{\sigma_1}{\mapsto}^* p_0$, we have $(q,\epsilon) \hookrightarrow^* (p_0', \sigma_2)$ and $(p_0,\epsilon) \hookrightarrow^* (p,\sigma_1)$, respectively.

Case 1. By the saturation rule $[c]$, we have $(p_0',\epsilon) \hookrightarrow (p_0,\delta)$. Thus,

$$(q,\epsilon) \hookrightarrow^* (p_0',\sigma_2) \hookrightarrow (p_0,\delta\sigma_2) \hookrightarrow^* (p,\sigma_1\delta\sigma_2)$$

Case 2. By the saturation rule $[i]$, we have $\exists p' \in Q$ that $(p',\gamma) \hookrightarrow (p_0,\gamma)$ and $p' \overset{\gamma}{\mapsto} p_0'$ in $\mathcal{A}_i$. From induction hypothesis, we have $(p_0',\epsilon) \hookrightarrow^* (p',\gamma)$. Thus,

$$(q,\epsilon) \hookrightarrow^* (p_0',\sigma_2) \hookrightarrow^* (p',\gamma\sigma_2) \hookrightarrow (p_0,\gamma\sigma_2) \hookrightarrow^* (p,\sigma_1\gamma\sigma_2)$$

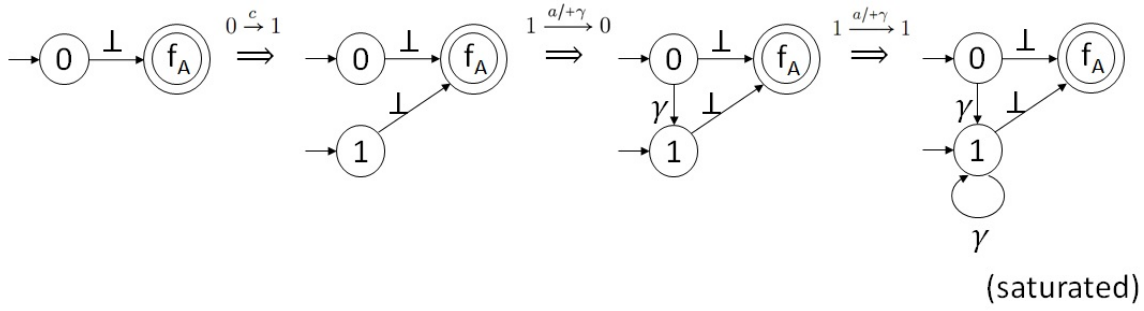


Figure 2.2: An example of $\mathcal{P}$-automaton construction

Case 3. By the saturation rule $[r_0]$, we have $q = p'_0 = f_{\mathcal{A}}, \sigma' = \gamma = \perp, \sigma_2 = \epsilon$. For rule $[r_0]$, the lemma always holds by the second statement.
Case 4. By the saturation rule $[r]$, we have $\exists p', q' \in Q : (p', \gamma) \hookrightarrow (p_0, \epsilon)$, and $p' \overset{\gamma}{\mapsto} q' \overset{\delta}{\mapsto} p'_0$ in $\mathcal{A}_i$. From induction hypothesis, we have $(p'_0, \epsilon) \hookrightarrow (q', \delta)$ and $(q', \epsilon) \hookrightarrow (p', \gamma)$. Thus,

$$(q, \epsilon) \hookrightarrow^* (p'_0, \sigma_2) \hookrightarrow^* (q', \delta\sigma_2) \hookrightarrow^* (p', \gamma\delta\sigma_2) \hookrightarrow (p_0, \delta\sigma_2) \hookrightarrow^* (p, \sigma_1\delta\sigma_2)$$

□

Lemma 1 shows that each accepted configuration of $Post^*(\mathcal{A}_\mathcal{C})$ during the saturation process is in $post^*(\mathcal{C})$ *(soundness)*. On the other hand, saturation rules put immediate successor configurations, and all configurations in $post^*(\mathcal{C})$ are finally accepted by $Post^*(\mathcal{C})$ *(completeness)*.

Theorem 2. $post^*(\mathcal{C}_0) = Conf(Post^*(\mathcal{A}_0))$

Example 2. *Figure 2.2 shows the example of $\mathcal{P}$-automaton construction of VPA M from Figure 2.1(a). However, it saturates before all the transtion rules are applied. The initial configuration of M is: $\mathcal{C}_0 = \{(0, \perp)\}$, and the reachable configurations are: $Post^*(\mathcal{C}_0) = \{(0, \gamma^*\perp), (1, \gamma^*\perp)\}$.*

Chapter 3

Antichain algorithm for universality checking

Recall that a non-deterministic VPA $M = (Q, Q_0, \Gamma, F, \Delta)$, and a determinized VPA M^d of M by Algorithm 1 (Section 2.1) is $(Q', Q'_0, \Gamma', F', \Delta')$.

3.1 Standard method

The standard algorithm for universality of VPA is to first determinize the automaton, and then check for the reachability of a non-accepting state (or non-accepting configuration). All reachable configurations of determinized VPA can be computed by using $\mathcal{P}$-automaton technique to compute $Post^*(\mathcal{C}_0)$. A configuration $c = (q, w)$ is non-accepting (or rejecting) if q is not a final state. When a rejecting configuration is found, we stop and report that the original VPA is not universal. Otherwise, if all reachable configurations of determinized VPA are accepting, we report that the original VPA is universal.

Definition 5. *For determinized VPA M^d, let $ReachableConf(M^d)$ denotes the set of all reachable configurations of M^d and $RejectingConf(M^d)$ denotes the set of rejecting configurations of M^d. Let $q \in Q'$ and $\sigma \in (\Gamma' \setminus \{\bot\})^* \cdot \bot$ be a state and stack of M^d, respectively.*

- $ReachableConf(M^d) = \{(q, \sigma) \mid \exists w \in \Sigma^* : (Id_{Q_0}, \bot) \overset{w}{\hookrightarrow}^* (q, \sigma)\}$.
- $RejectingConf(M^d) = \{(q, \sigma) \mid q \notin F'\}$.

Recall that Id_{Q_0} is an only single initial state of M^d by Algorithm 1 (Section 2.1).

With the above algorithm and definition, we obtain the following theorem:

Theorem 3. *Let M be a non-deterministic VPA. M is not universal iff there exists a rejecting configuration of M^d, i.e., $ReachableConf(M^d) \cap RejectingConf(M^d) \neq \emptyset$.*

3.2 Antichain algorithm

In this section, we present an antichain algorithm to check universality of VPAs [37]. An antichain algorithm is a combination between on-the-fly method and minimization. On-the-fly method is used for finding a rejecting configuration more efficiently than the standard method, and the minimization is used to reduce the number of computations based on a partial ordering over states and stack symbols of M^d.

3.2.1 On-the-fly method

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

Simultaneous on-the-fly determinization and $\mathcal{P}$-automaton construction. To improve the efficiency, we perform simultaneously on-the-fly determinization and $\mathcal{P}$-automaton construction. There are two interleaving phases in this approach. First, we determinize VPA M step by step (iterations). After each step of determinization, we update the $\mathcal{P}$-automaton. Second, using new configurations obtained from newly added transitions of $\mathcal{P}$-automaton, perform the 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 is finally terminated. From standard method and Theorem 3, checking universality of M is finding a rejecting configuration of M^d. In Algorithm 2, we present an on-the-fly way to explore such a rejecting configuration (if there is any).

Example 3. *We illustrate an example of on-the-fly algorithm in Figure 3.1 where $\Sigma_c = \{a\}$, $\Sigma_r = \{b\}$ and $\Sigma_i = \{c\}$.*

The processes of the algorithm is performed as below:

1. At the first time, assume that an initial state of M^d is state q_1. If q_1 is rejecting, report that M is not universal immediately.
2. Then, the $\mathcal{P}$-automaton $\mathcal{A}_{\mathcal{C}_0}$ is constructed with two states $\{q_1, f_{\mathcal{A}}\}$ and only one transition $q_1 \overset{\perp}{\hookrightarrow} f_{\mathcal{A}}$, where $f_{\mathcal{A}}$ is an unique final state. For this time, $\mathcal{A}_{Post^*(\mathcal{C}_0)}$ is representing a set of reachable configurations of M^d $\{(q_1, \perp)\}$.

Algorithm 2: On-the-fly algorithm

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

Result: Universality of M

begin

Create the initial state of determinized VPA M^d

if *the initial state of* M^d *is not a final state* **then**

return *False*;

end

Initiate $\mathcal{P}$-automaton $\mathcal{A}_{\mathcal{C}_0}$ to present the initial configuration of M^d

$\mathcal{A}_{Post^*(\mathcal{C}_0)} \longleftarrow \mathcal{A}_{\mathcal{C}_0}$

Create transitions of M^d departing from the initial state

while *set of new transitions of* M^d *is not empty* **do**

Update $\mathcal{A}_{Post^*(\mathcal{C}_0)}$ using new transitions of M^d

if *a rejecting state is added to* $\mathcal{A}_{Post^*(\mathcal{C}_0)}$ **then**

return *False*;

end

Update transitions of M^d using new configurations from $\mathcal{A}_{Post^*(\mathcal{C}_0)}$

end

return *True*;

end

3. Update M^d departing from $(q_1, \perp)$. Suppose M^d has new transitions $\{q_1 \xrightarrow{a/+(q_1,a)} q_2, q_1 \xrightarrow{b/\perp} q_3, q_1 \xrightarrow{c} q_4\}$.

4. Update $\mathcal{A}_{Post^*(\mathcal{C}_0)}$ using new transitions of M^d. Now, $\mathcal{A}_{Post^*(\mathcal{C}_0)}$ is representing $\{(q_1, \perp), (q_2, (q_1, a)\perp), (q_3, \perp), (q_4, \perp)\}$. If one of q_2, q_3 and q_4 is rejecting, report that M is not universal.

5. We again update M^d using new configurations $\{(q_2, (q_1, a)\perp), (q_3, \perp), (q_4, \perp)\}$ of $\mathcal{A}_{Post^*(\mathcal{C}_0)}$, and so on.

3.2.2 Minimization

Recall that state Q' of M^d belongs to $2^{Q\times Q}$ and stack symbols Γ' belongs to $Q' \times \Sigma_c$. We define an ordering over states and stack symbols as follows:

Definition 6 (Partial ordering over states and stack symbols)**.**

– *Let* S_1 *and* S_2 *be states of* M^d*, we say* $S_1 \leq S_2$ *if* $S_1 \subseteq S_2$*.*

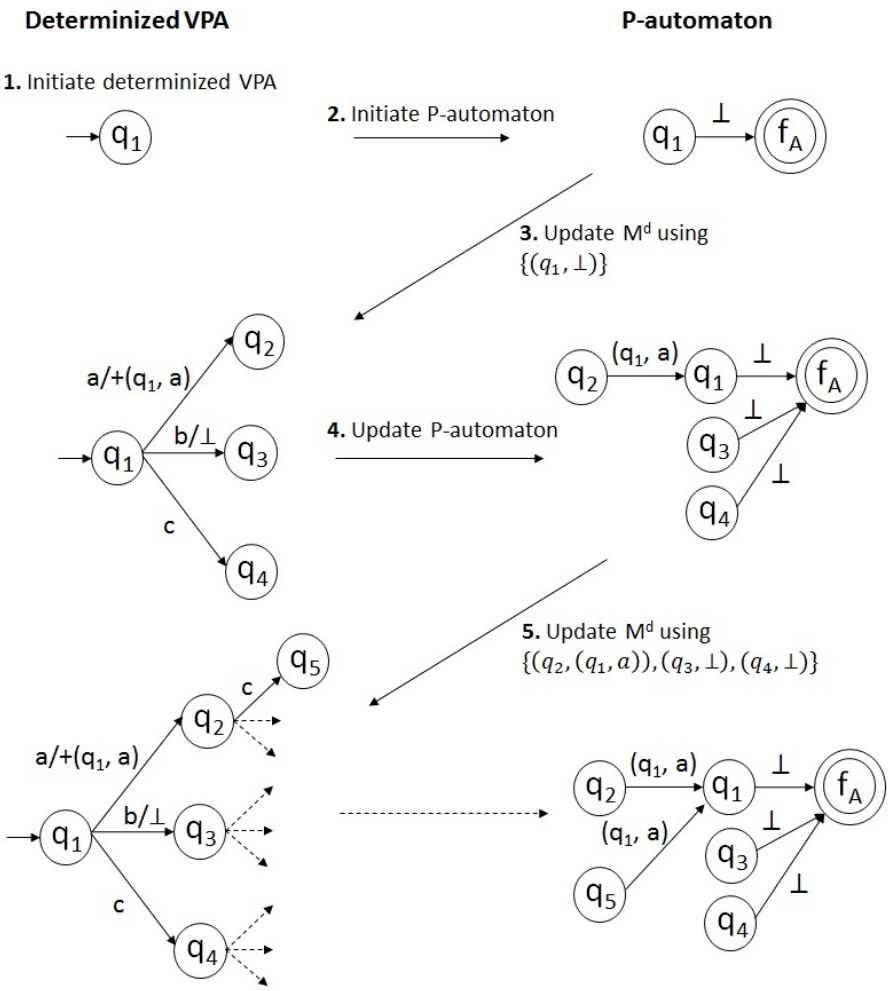


Figure 3.1: An example of on-the-fly algorithm

– *Let* $a_1, a_2 \in \Sigma_c$ *and let* $\gamma_1 = (S_1, a_1)$ *and* $\gamma_2 = (S_2, a_2)$ *be stack symbols of* M^d*, we say* $\gamma_1 \leq \gamma_2$ *if* $S_1 \subseteq S_2$ *and* $a_1 = a_2$*.*

From Definition 6, we extend the ordering to an ordering over configurations of the determinized VPA M^d.

Definition 7 (Partial ordering over configurations)**.** *Let* $c_1 = (S_1, \gamma_k \dots \gamma_1 \perp)$ *and* $c_2 = (S_2, \gamma'_k \dots \gamma'_1 \perp)$ *be configurations of* M^d*. We say* $c_1 \leq c_2$ *if* $S_1 \leq S_2$ *and* $\gamma_i \leq \gamma'_i$ *for all* $1 \leq i \leq n$*.*

Now we begin with establishing theoritical background for minimization in our algorithm.

Lemma 2. *Let* $S_1 \xrightarrow{a/+(S_1,a)} Id_{R'_1}$ *and* $S_2 \xrightarrow{a/+(S_2,a)} Id_{R'_2}$ *be push transitions of* M^d*. If* $S_1 \leq S_2$*, then* $Id_{R'_1} \leq Id_{R'_2}$*.*

Proof. By the determinization, we have:

– $S_1 \xrightarrow{a/+(S_1,a)} Id_{R'_1}$ where $R'_1 = \{q' \mid \exists q \in \Pi_2(S_1) : q \xrightarrow{a/+\gamma} q' \in \Delta_c\}$

– $S_2 \xrightarrow{a/+(S_2,a)} Id_{R'_2}$ where $R'_2 = \{q' \mid \exists q \in \Pi_2(S_2) : q \xrightarrow{a/+\gamma} q' \in \Delta_c\}$

Thus, if $S_1 \leq S_2$ then $R'_1 \leq R'_2$, and obviously $Id_{R'_1} \leq IdR'_2$. □

Lemma 3. *Let $S_1 \xrightarrow{a} S'_1$ and $S_2 \xrightarrow{a} S'_2$ be internal transitions of M^d. If $S_1 \leq S_2$, then $S'_1 \leq S'_2$.*

Proof. By the determinization, we have:

– $S_1 \xrightarrow{a} S'_1$ where $S'_1 = \{(q, q') \mid \exists q'' \in Q : (q, q'') \in S_1, q \xrightarrow{a} q' \in \Delta_i\}$

– $S_2 \xrightarrow{a} S'_2$ where $S'_2 = \{(q, q') \mid \exists q'' \in Q : (q, q'') \in S_2, q \xrightarrow{a} q' \in \Delta_i\}$

Thus, if $S_1 \leq S_2$ then $S'_1 \leq S'_2$. □

Lemma 4. *Let $S_1 \xrightarrow{a/\perp} S'_1$ and $S_2 \xrightarrow{a/\perp} S'_2$ be return transitions with empty stack of M^d. If $S_1 \leq S_2$, then $S'_1 \leq S'_2$.*

Proof. By the determinization, we have:

– $S_1 \xrightarrow{a/\perp} S'_1$ where $S'_1 = \{(q, q') \mid \exists q'' \in Q : (q, q'') \in S_1, q \xrightarrow{a/\perp} q' \in \Delta_r\}$

– $S_2 \xrightarrow{a/\perp} S'_2$ where $S'_2 = \{(q, q') \mid \exists q'' \in Q : (q, q'') \in S_2, q \xrightarrow{a/\perp} q' \in \Delta_r\}$

Thus, if $S_1 \leq S_2$ then $S'_1 \leq S'_2$. □

Lemma 5. *Let $S_1 \xrightarrow{a/-(S'_1,a'_1)} S''_1$ and $S_2 \xrightarrow{a/-(S'_2,a'_2)} S''_2$ be return transitions with non-empty stack of M^d. If $S_1 \leq S_2$ and $(S'_1, a'_1) \leq (S'_2, a'_2)$, then $S''_1 \leq S''_2$.*

Proof. By the determinization, we have:

– $update_1 = \{(q, q') \mid \exists q_1, q_2 \in Q : (q_1, q_2) \in S_1, q \xrightarrow{a'_1/+\gamma} q_1 \in \Delta_c, q_2 \xrightarrow{a/-\gamma} q' \in \Delta_r\}$
$S''_1 = \{(q_3, q') \mid \exists q \in Q : (q_3, q) \in S'_1, (q, q') \in update_1\}$

– $update_2 = \{(q, q') \mid \exists q_1, q_2 \in Q : (q_1, q_2) \in S_2, q \xrightarrow{a'_2/+\gamma} q_1 \in \Delta_c, q_2 \xrightarrow{a/-\gamma} q' \in \Delta_r\}$
$S''_2 = \{(q_3, q') \mid \exists q \in Q : (q_3, q) \in S'_2, (q, q') \in update_2\}$

Thus, if $S_1 \leq S_2$ and $a'_1 = a'_2$, then $update_1 \subseteq update_2$. And plus $S'_1 \leq S'_2$, then $S''_1 \leq S''_2$. □

Then, for configurations:

Lemma 6. *Let $c_1 = (S_1, \gamma_k \dots \gamma_1 \perp)$ and $c_2 = (S_2, \gamma'_k \dots \gamma'_1 \perp)$ be configurations of M^d such that $c_1 \leq c_2$. For any word $w \in \Sigma^*$, if $c_1 \overset{w}{\hookrightarrow}{}^* c'_1$ and $c_2 \overset{w}{\hookrightarrow}{}^* c'_2$, then $c'_1 \leq c'_2$.*

Proof. We prove this lemma by induction on length $|w|$ of w. For $|w| = 1 = a$, the lemma holds immediately from Lemma 2, 3, 4 or 5 with regarding to type of input symbol $|a|$. For

induction case, assume the lemma holds for w. We can prove for wa by using induction hypothesis, and Lemma 2, 3, 4 or 5 with regarding to type of a. □

Lemma 7. *Let c_1 and c_2 be configurations of M^d, If c_2 is a rejecting configuration and $c_1 \leq c_2$, then c_1 is also a rejecting configuration.*

Proof. Let $c_1 = (S_1, \gamma_k \dots \gamma_1 \perp)$ and $c_2 = (S_2, \gamma'_k \dots \gamma'_1 \perp)$ that $c_1 \leq c_2$. Recall that c_2 is a rejecting configuration if $S_2 \notin F'$ where F' is a set of final states of M^d. From Algorithm 1 (Section 2.1), that means $\Pi_2(S_2) \cap F = \emptyset$. Note that if $c_1 \leq c_2$, then $S_1 \leq S_2$. Thus, $\Pi_2(S_2) \cap F = \emptyset$. □

From Lemma 6 and 7, we can conclude that it is sufficient to compute only minimal reachable configurations to check for the existence of a rejecting configuration. Formally, we define:

Definition 8. $MinimalReachableConf(M^d) = \{(s, \sigma) \in ReachableConf(M^d) | \neg\exists (s', \sigma') \in ReachableConf(M^d) : (s', \sigma') \leq (s, \sigma)\}$

Theorem 4. *Let M be a non-deterministic VPA. M is not universal iff there exists a rejecting **minimal** configuration of M^d, i.e., $MinimalReachableConf(M^d) \cap RejectingConf(M^d) \neq \emptyset$.*

Minimal $\mathcal{P}$-automaton. Let $\mathcal{C}_0 = \{(Id_{Q_0}, \perp)\}$ be the initial configuration of M^d. Let $\mathcal{A}_{Post^*(\mathcal{C}_0)}$ be the $\mathcal{P}$-automaton presenting $ReachableConf(M^d)$ and let $\mathcal{A}^{min}_{Post^*(\mathcal{C}_0)}$ be the $\mathcal{P}$-automaton presenting $MinimalReachableConf(M^d)$. $\mathcal{A}^{min}_{Post^*(\mathcal{C}_0)}$ is computed by minimizing the $\mathcal{A}_{Post^*(\mathcal{C}_0)}$ as follows: "at each incremental expansion step" of constructing $\mathcal{A}_{Post^*(\mathcal{C}_0)}$, for two configurations $(S_1, \gamma_1\sigma)$ and $(S_2, \gamma_2\sigma)$, we only need to compare the states (i.e., S_1 and S_2) and top of stack (i.e., γ_1 and γ_2). Assume that $S_1 \leq S_2$ and $\gamma_1 \leq \gamma_2$, then $(S_1, \gamma_1\sigma) \leq (S_2, \gamma_2\sigma)$. So, we only need to keep the "smaller" configuration $(S_1, \gamma_1\sigma)$. We formalize this observation in Algorithm 3.

Finally, an antichain algorithm which combines on-the-fly method and minimization is shown in Algorithm 4. Note that the difference is only in the while-loop, we extract $\mathcal{P}$-automaton to be minimal before continue determinizing VPA.

Theorem 5 (Correctness of Algorithm 4)**.** *Let M be a non-deterministic VPA. Let M^d_i and $\mathcal{A}_i$ be determinized VPA and $\mathcal{P}$-automaton $\mathcal{A}^{min}_{Post^*(\mathcal{C}_0)}$ obtained in the i-th iteration of the while-loop in Algorithm 4. The VPA M is not universal if and only if $Conf(\mathcal{A}_i) \cap RejectingConf(M^d_i) \neq \emptyset$ for some i.*

Proof. If $Conf(\mathcal{A}_i) \cap RejectingConf(M^d_i) \neq \emptyset$ for some i then of course M is not universal. Conversely, if M is not universal, there exists at least one rejecting reachable configuration (S, σ) (i.e., $\Pi_2(S) \cap F \neq \emptyset$). If $(S, \sigma) \in Conf(\mathcal{A}_i)$ for some i, the theorem

Algorithm 3: Extract minimal transitions of $\mathcal{P}$-automaton

Data: A $\mathcal{P}$-automaton $\mathcal{A} = (P, \Gamma, P_0, \{f_\mathcal{A}\}, \nabla)$

Result: A $\mathcal{P}$-automaton $\mathcal{A}^{min} = (P, \Gamma, P_0, \{f_\mathcal{A}\}, \nabla')$

begin

$\nabla' = \nabla$

for *each state* $S \in P$ **do**

if $(S_1, \gamma_1, S) \in \nabla' \wedge (S_2, \gamma_2, S) \in \nabla' \wedge S_1 \leq S_2 \wedge \gamma_1 \leq \gamma_2$ **then**

$\nabla' \leftarrow \nabla' \setminus \{(S_2, \gamma_2, S)\}$

end

end

end

Algorithm 4: Antichain algorithm

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

Result: Universality of M

begin

Create the initial state of determinized VPA M^d

if *the initial state of* M^d *is not a final state* **then**

return *False*;

end

Initiate $\mathcal{P}$-automaton $\mathcal{A}_{\mathcal{C}_0}$ to present the initial configuration of M^d

$\mathcal{A}^{min}_{Post^*(\mathcal{C}_0)} \longleftarrow \mathcal{A}_{\mathcal{C}_0}$

Create transitions of M^d departing from the initial state

while *set of new transitions of* M^d *is not empty* **do**

Update $\mathcal{A}^{min}_{Post^*(\mathcal{C}_0)}$ using new transitions of M^d

if *a rejecting state is added to* $\mathcal{A}^{min}_{Post^*(\mathcal{C}_0)}$ **then**

return *False*;

end

Compute $\mathcal{A}^{min}_{Post^*(\mathcal{C}_0)}$ to be minimal by using Algorithm 3

Update transitions of M^d using new configurations from $\mathcal{A}^{min}_{Post^*(\mathcal{C}_0)}$

end

return *True*;

end

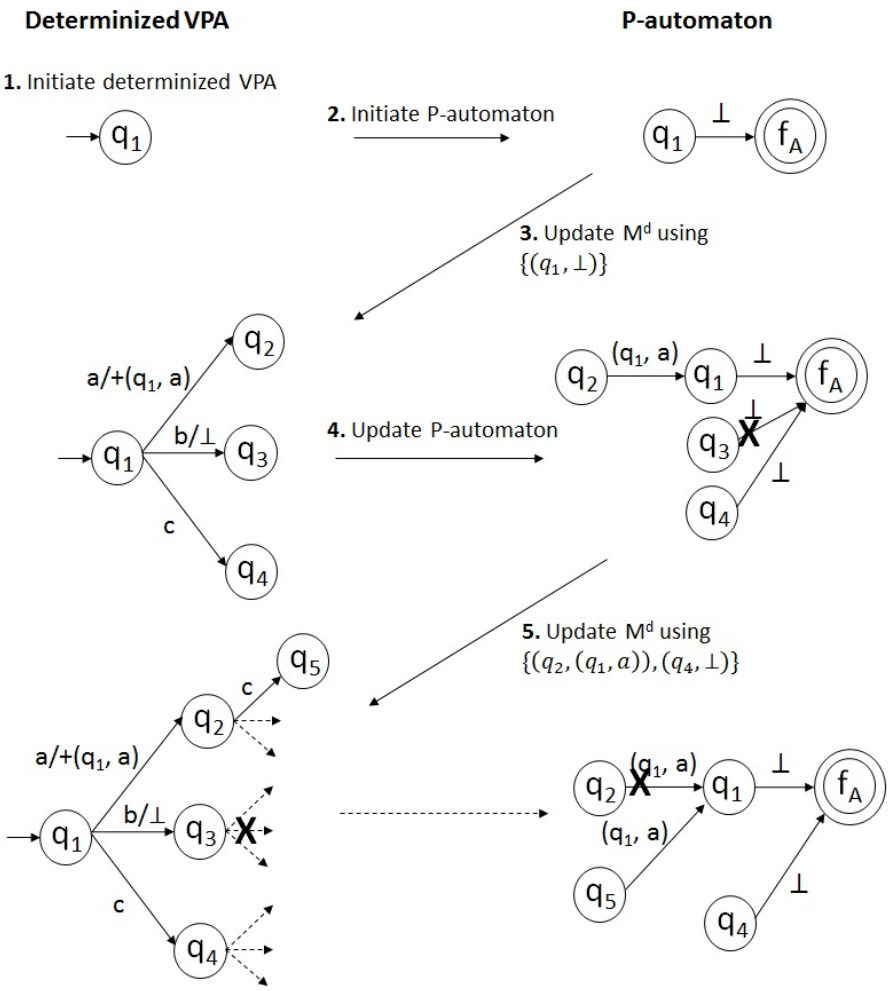


Figure 3.2: An example of antichain algorithm

holds. If $(S, \sigma) \notin Conf(\mathcal{A}_i)$ for any i (within a bound), it means that the configuration (S', σ') that $(S', \sigma') \overset{w}{\hookrightarrow}^* (S, \sigma)$ for some $w \in \Sigma^*$ was taken out from $Conf(\mathcal{A}_j)$ for $j \leq i$ by Algorithm 3. Thus there is at least one reachable configuration $(S'_2, \sigma'_2) \in Conf(\mathcal{A}_j)$ such that $(S'_2, \sigma'_2) \leq (S', \sigma')$ and $(S'_2, \sigma'_2) \overset{w}{\hookrightarrow}^* (S_2, \sigma_2) \in Conf(\mathcal{A}_i)$. From Lemma 6 and 7, we have $(S_2, \sigma_2) \leq (S, \sigma)$ and (S_2, σ_2) is a rejecting configuration. As a result, there exists an i such that $Conf(\mathcal{A}_i) \cap RejectingConf(M_i^d) \neq \emptyset$, the theorem holds. □

Example 4. *To describe how antichain algorithm work, we revisit Example 3. Assume that in determinized VPA M^d $q_1 \leq q_3$ and $q_5 \leq q_2$. With these assumption, after step 4 of updating $\mathcal{P}$-automaton, we remove state and transition of q_3. Plus, we do not need to investigate farther from q_3. And also after q_5 appears, we can remove q_2. The illustration is shown in Figure 3.2.*

Complexity. In the worst case (i.e., when the VPA is universal), the complexity of antichain-base algorithm is the same as the standard method, that is $O(2^{3|Q|^2})$ where

$|Q|$ is a number of state of M (this is because checking emptiness of pushdown system is cubic time of number of state [12]). However, in the case of not universal VPA, the antichain algorithm really outperforms the standard one.

Chapter 4

Bounded model checking for VPA

Bounded model checking (BMC) is a SAT-based technique for symbolic model checking. The main idea of BMC is to avoid the full state space generation and look for a violation of a given property of bounded length. If no violation is found in that length, enlarge the bound and repeat the search again, until either a violation is found, or we can guarantee that no violation is lying beyond that length.

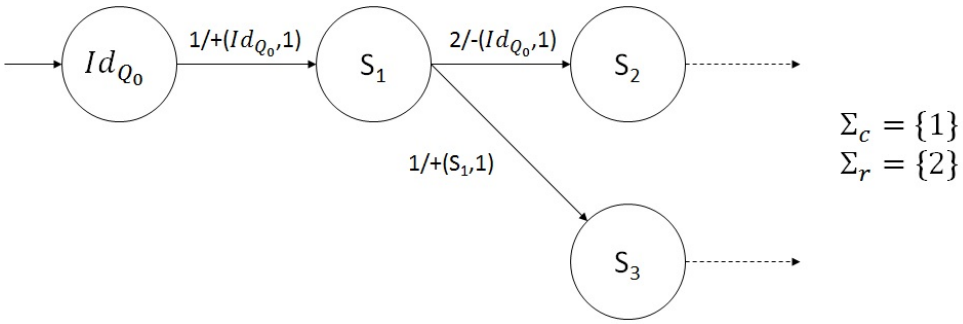
The basic idea of BMC for universality checking of VPA is to unroll the determinized VPA M^d for a fixed number of steps k, and check for universality of that k-step M^d (denoted by M^d_k). If M^d_k is not universal, we can conclude that M^d is also not universal. Otherwise repeat the process with larger bound until $M^d_k = M^d$ or we can guarantee that M^d is universal.

The question is how to unroll M^d to be M^d_k, what of M^d should be represented in M^d_k in order to decide universality, and how to obtain M^d_{k+1} from M^d_k.

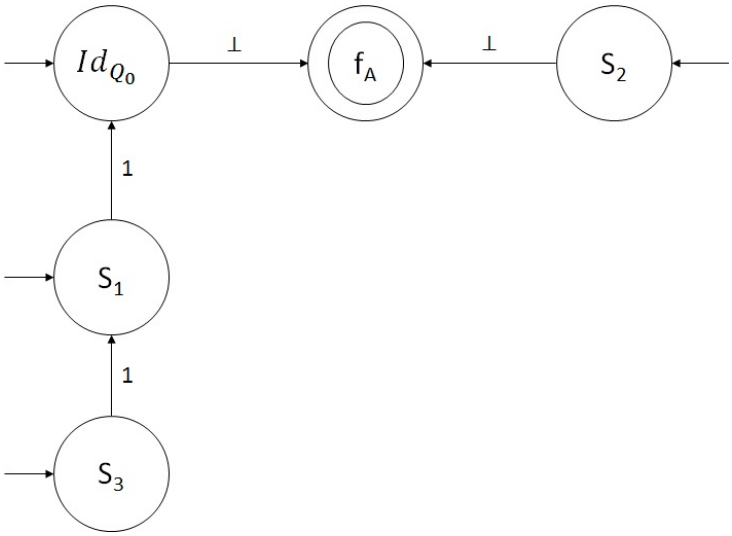
We have considered many options to bound M^d. Each step in bound k can be one step of applying:

1. transition $\rightarrow$ of M^d
2. transition $\mapsto$ of $\mathcal{P}$-automaton of M^d
3. transition $\hookrightarrow$ of configurations
4. transition $\rightharpoonup$ of $\mathcal{P}$-automaton transition (will be defined later)
5. saturation rule $\Rightarrow$ of $\mathcal{P}$-automaton saturation process

In this chapter, we will explain each option and clarify a possibility of each option to be used for BMC for universality checking. Note that our BMC algorithm BMCi is based on the option 4 and BMCp is based on the option 5.



(a) M^d



(b) $\mathcal{P}$-automaton of M^d

Figure 4.1: An example of M^d

4.1 Bounded transitions of M^d ($\rightarrow$)

This is the simplest way to unroll M^d. Like BMC of finite automata, M^d_k represents M^d by reachable states in k steps. Starting from $k = 0$, the starting point is an initial state of M^d Id_{Q_0}. Each step within bound is a transition $\rightarrow$ of Δ'.

Example 5. *From determinized VPA M^d in Figure 4.1(a),*

$$M^d_0 = \{Id_{Q_0}\}, M^d_1 = \{S_1\}, M^d_2 = \{S_2, S_3\}$$

However, unlike finite automata, this bounded method does not work because of at least 2 reasons. (i) M^d_k does not contain any information about stack. From M^d_k we can compute M^d_{k+1} with call and internal transitions but not return transitions. For instance, in Example 5 actually we cannot know whether from M^d_1 we will have S_2 in M^d_2 because we do not know whether stack is $(Id_{Q_0}, 1)$. (ii) To be able to compute M^d_{k+1} from length M^d_k, it requires a full set of transitions. That means we need to compute for a full Δ'

of M^d. If we do that successfully, we can just use $\mathcal{P}$-automaton to check reachability of rejecting configurations like in standard algorithm. It does not make sense to use BMC in the first place.

4.2 Bounded transitions of $\mathcal{P}$-automaton ($\mapsto$)

Recall that the purpose $\mathcal{P}$-automaton is to encode configurations of M^d into a regular expression. Each configuration (q, σ) is accepted by $\mathcal{P}$-automaton if $q \overset{\sigma}{\mapsto}{}^* f_{\mathcal{A}}$. In this method, each step within bound is a *backward* of transition $\mapsto$ of ∇. Starting point M_0^d is $f_{\mathcal{A}}$. For each k, M_k^d represents states that can reach to $f_{\mathcal{A}}$ in k steps of $\mathcal{P}$-automaton. That are states that can reach to stack height k in M^d.

Example 6. *From $\mathcal{P}$-automaton of M^d in Figure 4.1(b),*

$$M_0^d = \{f_{\mathcal{A}}\}, M_1^d = \{Id_{Q_0}, S_2\}, M_2^d = \{S_1\}$$

However, like finite automata, it requires a full set of transitions of $\mathcal{P}$-automaton to perform transitions like that. Again, it requires a full determinization of M^d to construct transitions of $\mathcal{P}$-automaton.

4.3 Bounded transitions of configuration ($\hookrightarrow$)

To keep information of states and stacks, we use M_k^d to represent M^d by reachable configurations instead of reachable states. Starting point is an initial configuration of M^d $(Id_{Q_0}, \bot)$. Each step within bound is a transition $\hookrightarrow$ according to Δ'.

Example 7. *From determinized VPA M^d in Figure 4.1(a),*

$$M_0^d = \{(Id_{Q_0}, \bot)\}, M_1^d = \{(S_1, (Id_{Q_0}, 1)\bot)\}, M_2^d = \{(S_2, \bot), (S_3, (S_1, 1)(Id_{Q_0}, 1)\bot)\}$$

This approach does not require a full set of transitions Δ' like the previous two approaches. This is because if we do know current state and top of stack we can compute next state using determinization algorithm (Algorithm 1 Section 2.1), and can compute stack based on input alphabet. For instance, in Example 7 $M_1^d = \{(S_1, (Id_{Q_0}, 1)\bot)\}$, we can compute for S_2 using S_1 and $(Id_{Q_0}, 1)$ follow Algorithm 1, and pop top of stack out. On the other hand, we can compute S_3 using S_1 (ignore stack), and push $(S_1, 1)$ to stack.

However, even if it is possible to unroll M^d without full computing determinization, this approach is not considered to be used for checking universality of VPAs. Because

configurations of M^d have an uncertain length, stack of M^d can be finitely any height. It is difficult to encode to a propositional formula.

4.4 Bounded transitions of $\mathcal{P}$-automaton transition ($\rightharpoonup$)

Before we talk about transitions of $\mathcal{P}$-automaton transtion ($\rightharpoonup$), we have a crucial observation of $\mathcal{P}$-automaton of determinized VPA.

Remark. From the Definition 3 (Section 2.2) alphabet of $\mathcal{P}$-automaton is equal to the stack alphabet of VPA. The alphabet of $\mathcal{P}$-automaton of M^d must belong to $Q' \times \Sigma_c$. However, it is a crucial observation that every transition of $\mathcal{P}$-automaton, which does not lead to the final state (i.e., $p \xrightarrow{\gamma} p' \in Q'$), the first component of input symbol is always equal to the destination state (i.e., $\exists a' \in \Sigma_c : \gamma = (p', a')$). This is because of behaviors of the determinization process (Algorithm 1 Section 2.1) and $\mathcal{P}$-automaton saturation rules (Definition 4 Section 2.2). Therefore, we can cut out the alphabet of $\mathcal{P}$-automaton of M^d from $Q' \times \Sigma_c$ to just Σ_c. For instance, as shown in Figure 4.1(b), actually the transitions must be $S_1 \overset{(Id_{Q_0},1)}{\mapsto} Id_{Q_0}$ and $S_3 \overset{(S_1,1)}{\mapsto} S_1$

We define transitions $\rightharpoonup$ between transitions of $\mathcal{P}$-automaton of M^d as follow:

Definition 9 (transitions between $\mathcal{P}$-automaton transitions on Σ^*)**.** *For determinized VPA $M^d = (Q', Q'_0, \Gamma', F', \Delta')$ and $\mathcal{P}$-automaton $\mathcal{A} = (P, \Sigma_c, P_0, \{f_\mathcal{A}\}, \nabla)$ accepting configurations of M^d, a set ∇' of transitions of $\mathcal{P}$-automaton transition is given by:*

- [c] *Call transition: For every $a \in \Sigma_c$, $(S \overset{a'}{\mapsto} S'') \overset{a}{\rightharpoonup} (Id_{R'} \overset{a}{\mapsto} S) \in \nabla'$ if $S \xrightarrow{a/+(S,a)} Id_{R'} \in \Delta'_c$*
- [i] *Internal transition: For every $a \in \Sigma_i$, $(S \overset{a'}{\mapsto} S'') \overset{a}{\rightharpoonup} (S' \overset{a'}{\mapsto} S'') \in \nabla'$ if $S \xrightarrow{a} S' \in \Delta'_i$*
- [r_0] *Return transition with empty stack: For every $a \in \Sigma_r$, $(S \overset{\perp}{\mapsto} f_\mathcal{A}) \overset{a}{\rightharpoonup} (S' \overset{\perp}{\mapsto} f_\mathcal{A}) \in \nabla'$ if $S \xrightarrow{a/\perp} S' \in \Delta'_r$*
- [r] *Return transition: For every $a \in \Sigma_r$, $(S \overset{a'}{\mapsto} S'') \overset{a}{\rightharpoonup} (S' \overset{a''}{\mapsto} S''') \in \nabla'$ if $S \xrightarrow{a/-(S'',a')} S' \in \Delta'_r$ and $S \overset{a'}{\mapsto} S'' \overset{a''}{\mapsto} S''' \in \nabla$*

We write $\rightharpoonup^*$ for the reflexive transitive closure of $\rightharpoonup$.

The starting point is an initial transition of $\mathcal{P}$-automaton that is $Id_{Q_0} \overset{\perp}{\mapsto} f_\mathcal{A}$. For each k, M^d_k represents $\mathcal{P}$-automaton transitions after reading input of length k.

Example 8. *From Figure 4.1, the set of transitions of $\mathcal{P}$-automaton transition ∇' is:*

$$\begin{aligned}\nabla' =\{&(Id_{Q_0} \overset{\perp}{\mapsto} f_{\mathcal{A}}) \overset{1}{\rightharpoonup} (S_1 \overset{1}{\mapsto} Id_{Q_0}),\\ &(S_1 \overset{1}{\mapsto} Id_{Q_0}) \overset{2}{\rightharpoonup} (S_2 \overset{\perp}{\mapsto} f_{\mathcal{A}}),\\ &(S_1 \overset{1}{\mapsto} Id_{Q_0}) \overset{1}{\rightharpoonup} (S_3 \overset{1}{\mapsto} S_1)\}\end{aligned}$$

and steps of unrolling M^d are:

$$\begin{aligned}&M_0^d = \{Id_{Q_0} \overset{\perp}{\mapsto} f_{\mathcal{A}}\}, M_1^d = \{S_1 \overset{1}{\mapsto} Id_{Q_0}\},\\ &M_2^d = \{S_2 \overset{\perp}{\mapsto} f_{\mathcal{A}}, S_3 \overset{1}{\mapsto} S_1\}\end{aligned}$$

This method does not require a full determinization of M. Because for each M_k^d we can compute M_{k+1}^d simultaneously with determinization and ∇' construction (Similar to on-the-fly algorithm in Algorithm 2 Section 3.2.1 but construct ∇' instead of $\mathcal{P}$-automaton).

We can check universality of M_k^d by checking whether M_k^d contains a transition $p \overset{a}{\mapsto} p'$ that $p \notin F'$. We will explain in the next chapter why tracking $\mathcal{P}$-automaton transitions can check universality of VPA.

One of our BMC algorithms called BMCi is developed based on this approach.

4.5 Bounded $\mathcal{P}$-automaton saturation process ($\Rightarrow$)

Let $\mathcal{C}_0$ be an initial configuration of M^d and let $\mathcal{A}_0$ be a $\mathcal{P}$-automaton accepting $\mathcal{C}_0$, Assume that the saturation procedure to compute $Post^*(\mathcal{A}_0)$ proceeds $\mathcal{A}_0 \Rightarrow \mathcal{A}_1 \Rightarrow \mathcal{A}_2 \ldots$. For each bound k, M_k^d represents $\mathcal{P}$-automaton of M^d after applying saturation rules k times.

Example 9. *From Figure 4.1,*

$$\begin{aligned}&M_0^d = \{Id_{Q_0} \overset{\perp}{\mapsto} f_{\mathcal{A}}\},\\ &M_1^d = \{Id_{Q_0} \overset{\perp}{\mapsto} f_{\mathcal{A}}, S_1 \overset{1}{\mapsto} S_0\},\\ &M_2^d = \{Id_{Q_0} \overset{\perp}{\mapsto} f_{\mathcal{A}}, S_1 \overset{1}{\mapsto} S_0, S_2 \overset{\perp}{\mapsto} f_{\mathcal{A}}, S_3 \overset{1}{\mapsto} S_1\}\end{aligned}$$

Note that this is different from the previous one. In bounded $\rightharpoonup$, M_k^d represents $\mathcal{P}$-automaton transitions after reading input of length k. In bounded $\Rightarrow$, M_k^d represents $\mathcal{A}_k$, a $\mathcal{P}$-automaton that obtained by applying saturation rules k times.

This method does not require a full determinization of M. Because we can perform determinization and $\mathcal{P}$-automaton construction simultaneously using on-the-fly algorithm (Algorithm 2 Section 3.2.1).

We can check universality of M_k^d by checking whether M_k^d ($= \mathcal{A}_k$) accepts any rejecting configurations, i.e., checking whether there exists a transition $p \overset{a}{\mapsto} p'$ that $p \notin F'$ in M_k^d.

One of our BMC algorithms called BMCp is developed based on this approach.

Chapter 5

BMC and antichains for universality checking of VPA

In the previous chapter, we talked about how to unroll M^d for each bound k. In this chapter, we will talk about our two BMC algorithms, called BMCi and BMCp. BMCi unrolls M^d based on transitions of $\mathcal{P}$-automaton transition ($\rightharpoonup$), and BMCp unrolls M^d based on $\mathcal{P}$-automaton saturation process ($\Rightarrow$).

First of all, we will give an idea of how to check non-universality of VPAs (finding counter-examples) using BMC, and how to apply antichain algorithm to BMC for checking non-universality. Next we will explain our algorithms BMCi and BMCp and show differences of them in non-universality checking. Then we will talk about universality checking, and finally how to apply antichain to universality checking.

5.1 Encoding for non-universality checking of VPAs

In this section, we will first talk about how $\mathcal{P}$-automaton transitions are used to check non-universality. Next we will state the method for encoding transitions of $\mathcal{P}$-automaton transition ($\rightharpoonup$) and non-universality checking to be propositional formulae.

5.1.1 Basic idea

We first clarify why transitions of $\mathcal{P}$-automaton transition ($\rightharpoonup$) relate to non-universality checking. We start with the relation between configurations of M^d and $\mathcal{P}$-automaton transitions. Recall that from the remark mentioned in Section 4.4, the input alphabet of $\mathcal{P}$-automaton of M^d is Σ_c rather than Γ',

Definition 10. *Let $\mathcal{A} = (P, \Sigma_c, P_0, \{f_\mathcal{A}\}, \nabla)$ be a $\mathcal{P}$-automaton of a determinized VPA $M^d = (Q', Q'_0, \Gamma', F', \Delta')$, and let $\sigma = (q_n, a_n) \dots (q_1, a_1)\perp$ be a stack of M^d where $q_i \in Q'$ and $a_i \in \Sigma_c$. For a configuration of M^d $c = (q, \sigma)$, we define $\delta_c = q \overset{a_n}{\mapsto} q'$ as the first transition of accepting path of c in $\mathcal{A}$ (i.e., $q \overset{a_n}{\mapsto} q' \overset{a_{n-1}}{\mapsto} \dots \overset{\perp}{\mapsto} f_\mathcal{A}$).*

Definition 11. *For any word $w \in \Sigma^*$, let $w = a_1 \dots a_k$ and let δ_i are $\mathcal{P}$-automaton transitions for $i \geq 0$.*

- *A run of M^d on w is a sequence of configurations of M^d on w starting from an (unique) initial configuration, i.e., $(q_0, \perp) \overset{a_1}{\hookrightarrow} (q_1, \sigma_1) \dots \overset{a_k}{\hookrightarrow} (q_k, \sigma_k)$ where $q_0 = Id_{Q_0}$, and is denoted by $(q_0, \perp) \overset{w}{\hookrightarrow}{}^* (q_k, \sigma_k)$.*
- *A run of δ on w is a sequence of $\mathcal{P}$-automaton transitions on w obtained by Algorithm 9 (Section 4.4), starting from an (unique) initial $\mathcal{P}$-automaton transition, i.e., $\delta_0 \overset{a_1}{\rightharpoonup} \delta_1 \dots \overset{a_k}{\rightharpoonup} \delta_k$ where $\delta_0 = q_0 \overset{\perp}{\mapsto} f_\mathcal{A}$, and is denoted by $\delta_0 \overset{w}{\rightharpoonup}{}^* \delta_k$.*

Lemma 8. *Let $\delta_i = q_i \overset{a_i}{\mapsto} q'_i \in \nabla$ be transitions of $\mathcal{P}$-automaton of M^d. For input word $w \in \Sigma^*$, let a run of M^d on w be $(q_0, \perp) \overset{w}{\hookrightarrow}{}^* (q_k, \sigma_k)$ and let $\delta_0 \overset{w}{\rightharpoonup}{}^* \delta_k$ be a run of δ on w. If $\delta_0 = \delta_{(q_0, \perp)}$, then $\delta_k = \delta_{(q_k, \sigma_k)}$.*

Proof. We prove it by induction on length of w. For $|w| = 0$, the statement holds immediately. For induction case, assume above statement hold for $|w| = k$, and for wa_{k+1} a run of M^d is extended to $(q_0, \perp) \overset{w}{\hookrightarrow}{}^* (q_k, \sigma_k) \overset{a_{k+1}}{\hookrightarrow} (q_{k+1}, \sigma_{k+1})$, and a run of δ is extended to $\delta_0 \overset{w}{\rightharpoonup}{}^* \delta_k \overset{a_{k+1}}{\rightharpoonup} \delta_{k+1}$. The proof of wa_{k+1} is based on the case of transition of a_{k+1}.

- $a_{k+1} \in \Sigma_c$. Following behaviors of VPA and determinization, we have $(q_k, \sigma_k) \overset{a_{k+1}}{\hookrightarrow} (Id_{q_k}, (q_k, a_{k+1})\sigma_k)$. Following definition 9 (Section 4.4) $[c]$, $\delta_{k+1} = Id_{q_k} \overset{a_{k+1}}{\mapsto} q_k$, and from induction hypothesis, $q_k \overset{\sigma_k}{\mapsto}{}^* f_\mathcal{A}$. Thus, $\delta_{(q_{k+1}, \sigma_{k+1})} = Id_{q_k} \overset{a_{k+1}}{\mapsto} q_k = \delta_{k+1}$
- $a_{k+1} \in \Sigma_i$. Following behaviors of VPA and determinization, we have $(q_k, \sigma_k) \overset{a_{k+1}}{\hookrightarrow} (q_{k+1}, \sigma_k)$. Following definition 9 $[i]$, $\delta_{k+1} = q_{k+1} \overset{a_k}{\mapsto} q'_k$, and from induction hypothesis, $q_k \overset{\sigma_k}{\mapsto}{}^* f_\mathcal{A}$ with the first transition $q_k \overset{a_k}{\mapsto} q'_k$. Thus, δ_{k+1} is the first transition of $q_{k+1} \overset{\sigma_k}{\mapsto}{}^* f_\mathcal{A}$, then $\delta_{k+1} = \delta_{(q_{k+1}, \sigma_k)}$.
- $a_{k+1} \in \Sigma_r$ and stack is empty. Following behaviors of VPA and determinization, we have $(q_k, \perp) \overset{a_{k+1}}{\hookrightarrow} (q_{k+1}, \perp)$. Following definition 9 $[r_0]$, $\delta_{k+1} = q_{k+1} \overset{\perp}{\mapsto} f_\mathcal{A}$. Thus, obviously, $\delta_{k+1} = \delta_{(q_{k+1}, \perp)}$.
- $a_{k+1} \in \Sigma_r$ and stack is not empty. Following behaviors of VPA and determinization, we have $(q_k, (q'', a')\sigma'_k) \overset{a_{k+1}}{\hookrightarrow} (q_{k+1}, \sigma'_k)$. Following definition 9 $[r]$, $\delta_{k+1} = q_{k+1} \overset{a''}{\mapsto} q'_{k+1}$ where $q_k \overset{a'}{\mapsto} q'' \overset{a''}{\mapsto} q'_{k+1}$, and from induction hypothesis, there must be $i < k$ that $q_i = q''$ and $q_i \overset{\sigma_i}{\mapsto}{}^* f_\mathcal{A}$. Thus, $\delta_{k+1} = \delta_{(q_{k+1}, \sigma'_k)}$ □

We say a configuration $c = (q, \sigma)$ of M^d is rejecting if $q \notin F'$, and we say a $\mathcal{P}$-automaton transition $\delta = p \overset{a}{\mapsto} p'$ is rejecting if $p \notin F'$.

Lemma 9. *A configuration c of M^d is rejecting iff δ_c is rejecting*

Proof. Let $c = (q, \sigma)$, from Definition 10 $\delta_c = q \overset{a}{\mapsto} q'$, it is obvious that c is rejecting if and only if δ_c is rejecting □

From Lemma 8 and 9, suppose a run of M^d on w is $(q_0, \bot) \overset{w}{\hookrightarrow}{}^* (q_k, \sigma_k)$ and suppose a run of δ on w is $\delta_0 \overset{w}{\rightharpoonup}{}^* \delta_k$. If $w \notin L(M^d)$, then (q_k, σ_k) is rejecting, and also δ_k is rejecting. We can conclude that we can use a run of δ to monitor a run of M^d for non-universality checking. This is important because a configuration of M^d belongs to $Q' \times ((\Gamma' \setminus \{\bot\})^* \cdot \bot)$ that have an uncertain length but δ belongs to $Q' \times \Sigma_c \times (Q' \cup \{f_\mathcal{A}\})$ that always has a certain length. Therefore, encoding of δ ($\mathcal{P}$-automaton transitions) is a lot easier than encoding configurations directly.

Theorem 6. *M^d is not universal iff there exists $k \in \mathbb{N}$ that $w \in \Sigma^k$ and a run of δ on w is $\delta_0 \overset{w}{\rightharpoonup}{}^* \delta_k$ and δ_k is rejecting*

5.1.2 Encoding of $\mathcal{P}$-automaton transition

Each $\mathcal{P}$-automaton transition $\delta = q \overset{a}{\mapsto} q'$ is encoded to propositional variables as follows:

Definition 12. *Let $n = |Q|$ and $m = \lfloor \log(|\Sigma_c|) \rfloor + 1$. We define three vectors of propositional variables:*

- $\mathfrak{a} = (\mathfrak{a}_{1,1}, \ldots, \mathfrak{a}_{1,n}, \mathfrak{a}_{2,1}, \ldots, \mathfrak{a}_{n,n})$
- $\mathfrak{b} = (\mathfrak{b}_m, \ldots, \mathfrak{b}_1)$
- $\mathfrak{c} = (\mathfrak{c}_f, \mathfrak{c}_{1,1}, \ldots, \mathfrak{c}_{1,n}, \mathfrak{c}_{2,1}, \ldots, \mathfrak{c}_{n,n})$

to represent each transition $q \overset{a}{\mapsto} q'$ of $\mathcal{P}$-automaton of M^d.

In particular, vector $\mathfrak{a}$ represents a source state of transition (i.e., q), vector $\mathfrak{b}$ represents an alphabet of transition (i.e., a), and vector $\mathfrak{c}$ represents a destination state of transition (i.e., q').

Length. Since states of $\mathcal{P}$-automaton of M^d belong to $Q' \cup \{f_\mathcal{A}\}$, size of states $|P| = 2^{n^2} + 1$. But from the definition 3 (Section 2.2) there is always no transition from the final state $f_\mathcal{A}$, length n^2 is enough for vector $\mathfrak{a}$ but length of $\mathfrak{c}$ must be $n^2 + 1$. For vector $\mathfrak{b}$, length $m = \lfloor \log(|\Sigma_c|) \rfloor + 1$ is enough to encode the call alphabet.

Intention. Because $|Q| = n$, one can easily encode the set of states $Q = \{1, \ldots, n\}$. Each state q of M^d is a set of pair of integers (i.e., $q \subseteq Q \times Q$). A vector $\mathfrak{a}$ is a bit representation of set q. Each propositional variable $\mathfrak{a}_{i,j}$ of vector $\mathfrak{a}$ $(1 \leq i, j \leq n)$ represents whether a pair (i, j) is in q, i.e., $\mathfrak{a}_{i,j}$ is true denotes $(i, j) \in q$ and false otherwise. Similarly, $\mathfrak{c}_{i,j}$ represents whether a pair (i, j) is in q'. An additional variable $\mathfrak{c}_f$ is used to represent whether the destination of transition is a final state $f_{\mathcal{A}}$. Either $q' \in Q'$ then $\mathfrak{c}_f$ is false and $\mathfrak{c}_{i,j}$ represent q', or $q' = f_{\mathcal{A}}$ then $\mathfrak{c}_f$ is true and all $\mathfrak{c}_{i,j}$ are false. $\mathfrak{b}_m, \ldots, \mathfrak{b}_1$ are just a binary encoding of Σ_c. We reserve a special value of $\mathfrak{b}$ that is $\mathfrak{b}_i$ are all false for $\perp$.

We use vector $\mathfrak{a}^i, \mathfrak{b}^i, \mathfrak{c}^i$ to encode each δ_i $(0 \leq i \leq k)$ in a run $\delta_0 \rightharpoonup^* \delta_k$. Note that superscript is used to represent iteration index of vectors, and subscript is used to represent index in vectors. For example, a propositional variable $\mathfrak{a}^0_{1,1}$ represents whether a pair $(1, 1)$ is in q_0 $(\delta_0 = q_0 \overset{\perp}{\mapsto} f_{\mathcal{A}})$, and $\mathfrak{a}^2_{1,1}$ represents whether a pair $(1, 1)$ is in q_2 after reading a_2 and a run of δ is $\delta_0 \overset{a_1}{\rightharpoonup} \delta_1 \overset{a_2}{\rightharpoonup} \delta_2$ and $q_2 \overset{a'}{\mapsto} q'_2$.

5.1.3 Encoding of transitions of $\mathcal{P}$-automaton transition

In previous subsection, we presented how to encode each transition of $\mathcal{P}$-automaton. In this subsection, we will present how to encode a run of δ on w $\delta_0 \rightharpoonup^* \delta_k$.

First, we define some notations:

Definition 13. *For vector of propositional variables we define notations:*

- $\neg$ *of vector denotes a conjuction of negative of all indices.*
- $\iff$ *of vector with the same length denotes a conjunction of* $\iff$ *of all indices.*

For example, $\neg\mathfrak{a}$ *denotes* $\neg\mathfrak{a}_{1,1} \wedge \ldots \wedge \neg\mathfrak{a}_{n,n}$ *and* $\mathfrak{a} \iff \mathfrak{a}'$ *denotes* $(\mathfrak{a}_{1,1} \iff \mathfrak{a}'_{1,1}) \wedge \ldots \wedge (\mathfrak{a}_{n,n} \iff \mathfrak{a}'_{n,n})$.

Then, we define a propositional formula:

$$[\mathcal{P}(M^d)]^k := init(\mathfrak{a}^0, \mathfrak{b}^0, \mathfrak{c}^0) \wedge \bigwedge_{l=0}^{k-1} \mathcal{T}((\mathfrak{a}^l, \mathfrak{b}^l, \mathfrak{c}^l), (\mathfrak{a}^{l+1}, \mathfrak{b}^{l+1}, \mathfrak{c}^{l+1}))$$

to represent $\delta_0 \rightharpoonup^* \delta_k$ where $init(\mathfrak{a}^0, \mathfrak{b}^0, \mathfrak{c}^0)$ is an encoding of an initial $\mathcal{P}$-automaton transition, and each $\mathcal{T}((\mathfrak{a}^l, \mathfrak{b}^l, \mathfrak{c}^l), (\mathfrak{a}^{l+1}, \mathfrak{b}^{l+1}, \mathfrak{c}^{l+1}))$ represents a transition $\delta_l \overset{a_{l+1}}{\rightharpoonup} \delta_{l+1}$. Note that we introduced a variable l to run an iteration index of vectors, since i and j will be used to run an index in vectors later.

The initial $\mathcal{P}$-automaton transition δ_0 is $Id_{Q_0} \overset{\perp}{\mapsto} f_{\mathcal{A}}$ that represents an initial configuration

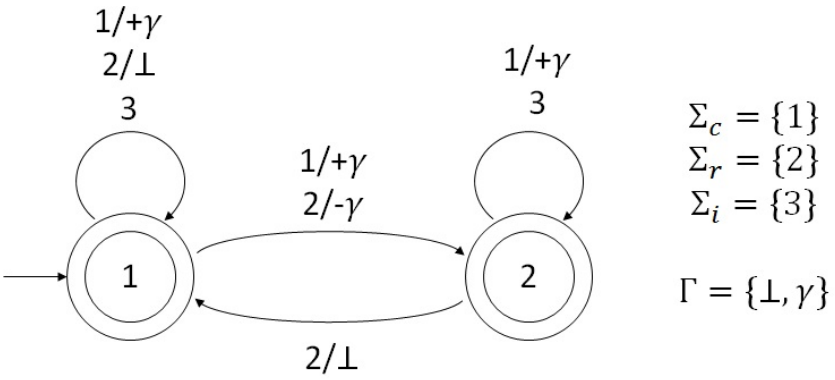


Figure 5.1: An example VPA for checking universality

$(Id_{Q_0}, \perp)$ of M^d. The formula *init* is defined as follow:

$$init(\mathfrak{a}^0, \mathfrak{b}^0, \mathfrak{c}^0) := \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} f_{init}(i,j) \wedge (\neg \mathfrak{b}^0) \wedge (\mathfrak{c}^0 \iff (true, false, \ldots, false))$$

$$f_{init}(i,j) := \begin{cases} \mathfrak{a}^0_{i,j} & \text{if } i = j \text{ and } i \in Q_0 \\ \neg \mathfrak{a}^0_{i,j} & \text{otherwise} \end{cases}$$

The function f_{init} represents an initial state Id_{Q_0} by vector $\mathfrak{a}^0$. Follow the definition, $\mathfrak{a}^0_{i,j}$ is true iff $(i,j) \in Id_{Q_0}$. Variables in vector $\mathfrak{b}^0$ are all false to represent $\perp$. Only $\mathfrak{c}_f$ is true in vector $\mathfrak{c}^0$ to represent $f_{\mathcal{A}}$.

Example 10. *From VPA M in figure 5.1, $|Q| = 2$ so vector $\mathfrak{a}$ has four indices that are $(\mathfrak{a}_{1,1}, \mathfrak{a}_{1,2}, \mathfrak{a}_{2,1}, \mathfrak{a}_{2,2})$, $\mathfrak{b} = (\mathfrak{b}_2, \mathfrak{b}_1)$ and $\mathfrak{c} = (\mathfrak{c}_f, \mathfrak{c}_{1,1}, \mathfrak{c}_{1,2}, \mathfrak{c}_{2,1}, \mathfrak{c}_{2,2})$. Note that since $|\Sigma_c| = 1$, actually we can define vector $\mathfrak{b}$ with only one index, but for better intention and understanding we define $\mathfrak{b}$ with two indices to be able to represent all alphabets $1, 2$ and 3. The initial state of M^d is $\{(1,1)\}$, so the initial transition of $\mathcal{P}$-automaton is $\{(1,1)\} \overset{\perp}{\mapsto} f_{\mathcal{A}}$. The formula $init(\mathfrak{a}^0, \mathfrak{b}^0, \mathfrak{c}^0)$ is $\mathfrak{a}^0_{1,1} \wedge \neg\mathfrak{a}^0_{1,2} \wedge \neg\mathfrak{a}^0_{2,1} \wedge \neg\mathfrak{a}^0_{2,2} \wedge \neg\mathfrak{b}^0_2 \wedge \neg\mathfrak{b}^0_1 \wedge \mathfrak{c}^0_f \wedge \neg\mathfrak{c}^0_{1,1} \wedge \neg\mathfrak{c}^0_{1,2} \wedge \neg\mathfrak{c}^0_{2,1} \wedge \neg\mathfrak{c}^0_{2,2}$ Therefore, $\mathfrak{a}^0, \mathfrak{b}^0$ and $\mathfrak{c}^0$ are $(true, false, false, false), (false, false)$ and $(true, false, false, false, false)$, respectively.*

The formula $\mathcal{T}((\mathfrak{a}^l, \mathfrak{b}^l, \mathfrak{c}^l), (\mathfrak{a}^{l+1}, \mathfrak{b}^{l+1}, \mathfrak{c}^{l+1}))$ represents a transition $\delta_l \overset{a_{l+1}}{\rightharpoonup} \delta_{l+1}$. $\mathcal{T}$ can be divided into four cases based on type of $\rightharpoonup$.

$$\mathcal{T}((\mathfrak{a}^l, \mathfrak{b}^l, \mathfrak{c}^l), (\mathfrak{a}^{l+1}, \mathfrak{b}^{l+1}, \mathfrak{c}^{l+1})) := \mathcal{T}_c \vee \mathcal{T}_i \vee \mathcal{T}_{r_0} \vee \mathcal{T}_r$$

Note that the arguments of $\mathcal{T}_c$, $\mathcal{T}_i$, $\mathcal{T}_{r_0}$ and $\mathcal{T}_r$ are the same as $\mathcal{T}$.

Let $+$ denotes a concatenation of two vectors and for $a \in \Sigma$, let $be(a)$ denotes a binary encoding of a. We define formulae for transitions of $\mathcal{P}$-automaton transition $(\rightharpoonup)$ as follows:

Call transition.

$$\mathcal{T}_c := \bigvee_{a\in\Sigma_c}[\bigwedge_{i=1}^{n}\bigwedge_{j=1}^{n} f_c(i,j) \wedge (\mathfrak{b}^{l+1} \iff be(a))] \wedge (\mathfrak{c}^{l+1} \iff (false) + \mathfrak{a}^l)$$

$$f_c(i,j) := \begin{cases} \mathfrak{a}_{i,j}^{l+1} \iff \bigvee_{j'}\bigvee_{i'=1}^{n} \mathfrak{a}_{i',j'}^{l} & \text{if } i=j \text{ and } \exists j' \in Q : j' \xrightarrow{a/+\gamma} i \in \Delta_c \\ \neg\mathfrak{a}_{i,j}^{l+1} & \text{otherwise} \end{cases}$$

The definition of $\mathcal{T}_c$ is an encoding of the case $[c]$ of $\rightharpoonup$ (Definition 9 Section 4.4). The definition of function f_c is to compute state $Id_{R'}$ from the determinization process. Note that $\mathcal{T}_c$ has a disjunction of call alphabet because case $[c]$ of $\rightharpoonup$ is for a call transition but not for a specific call alphabet. Each $a_{l+1} \in \Sigma_c$ can cause $[c]$ case of $\delta_l \overset{a_{l+1}}{\rightharpoonup} \delta_{l+1}$, and the next $\mathcal{P}$-automaton transition δ_{l+1} depends on a_{l+1}.

Internal transition.

$$\mathcal{T}_i := \bigvee_{a\in\Sigma_i}[\bigwedge_{i=1}^{n}\bigwedge_{j=1}^{n} f_i(i,j)] \wedge (\mathfrak{b}^{l+1} \iff \mathfrak{b}^{l}) \wedge (\mathfrak{c}^{l+1} \iff \mathfrak{c}^{l})$$

$$f_i(i,j) := \begin{cases} \mathfrak{a}_{i,j}^{l+1} \iff \bigvee_{j'} \mathfrak{a}_{i,j'}^{l} & \text{if } \exists j' \in Q : j' \xrightarrow{a} j \in \Delta_i \\ \neg\mathfrak{a}_{i,j}^{l+1} & \text{otherwise} \end{cases}$$

The definition of $\mathcal{T}_i$ is an encoding of the case $[i]$ of $\rightharpoonup$ (Definition 9 Section 4.4). The definition of function f_i is to compute state S' from the determinization process. Follow the definition of case $[i]$ of $\rightharpoonup$, the only vector that changes is vector $\mathfrak{a}$ (of source state in δ). Vectors $\mathfrak{b}^{l+1}$ and $\mathfrak{c}^{l+1}$ are the same as $\mathfrak{b}^{l}$ and $\mathfrak{c}^{l}$, respectively.

Return transition with empty stack.

$$\mathcal{T}_{r_0} := (\neg\mathfrak{b}^{l}) \wedge \bigvee_{a\in\Sigma_r}[\bigwedge_{i=1}^{n}\bigwedge_{j=1}^{n} f_{r_0}(i,j)] \wedge (\neg\mathfrak{b}^{l+1}) \wedge (\mathfrak{c}^{l+1} \iff (true, false, \ldots, false))$$

$$f_{r_0}(i,j) := \begin{cases} \mathfrak{a}_{i,j}^{l+1} \iff \bigvee_{j'} \mathfrak{a}_{i,j'}^{l} & \text{if } \exists j' \in Q : j' \xrightarrow{a/\perp} j \in \Delta_r \\ \neg\mathfrak{a}_{i,j}^{l+1} & \text{otherwise} \end{cases}$$

The definition of $\mathcal{T}_{r_0}$ is an encoding of the case $[r_0]$ of $\rightharpoonup$ (Definition 9 Section 4.4). First of all, $\neg\mathfrak{b}^{l}$ (denotes $\perp$) is a necessary condition to check whether the current stack is empty. Case $[r_0]$ is really similar to case $[i]$. Vector $\mathfrak{a}^{l+1}$ is computed in the same way as case $[i]$, but $\mathfrak{b}^{l+1}$ is all false to denote $\perp$ and only $\mathfrak{c}_f^{l+1}$ in $\mathfrak{c}^{l+1}$ is true to denote $f_{\mathcal{A}}$.

Return transition.

Let vector $\mathfrak{c}'$ be a vector $\mathfrak{c}$ without a variable $\mathfrak{c}_f$.

$$\mathcal{T}_r := \bigvee_{a\in\Sigma_r}\bigvee_{a'\in\Sigma_c}[(\mathfrak{b}^l \iff be(a'))\wedge\bigwedge_{i=1}^{n}\bigwedge_{j=1}^{n} f_r(i,j)]\wedge$$
$$\bigvee_{l'=0}^{l}[(\mathfrak{c}'^l \iff \mathfrak{a}^{l'})\wedge(\mathfrak{b}^{l+1} \iff \mathfrak{b}^{l'})\wedge(\mathfrak{c}^{l+1} \iff \mathfrak{c}^{l'})]$$

$$f_r(i,j) := \begin{cases} \mathfrak{a}_{i,j}^{l+1} \iff \bigvee_{i'}\bigvee_{j''}\bigvee_{j'}[\mathfrak{c}_{i,j''}^{l}\wedge\mathfrak{a}_{i',j'}^{l}] & \text{if } \exists i',j',j''\in Q, \exists\gamma\in\Gamma\setminus\perp: \\ & j''\xrightarrow{a'/+\gamma} i'\in\Delta_c \text{ and } j'\xrightarrow{a/-\gamma} j\in\Delta_r \\ \neg\mathfrak{a}_{i,j}^{l+1} & \text{otherwise} \end{cases}$$

The definition of $\mathcal{T}_r$ is an encoding of the case $[r]$ of $\rightharpoonup$ (Definition 9 Section 4.4). Return transition is the most complicated. First, formula $(\mathfrak{b}^l \iff be(a'))$ is used to check top of stack. $\mathcal{T}_r$ has two disjunctions, one of return alphabet and one of call alphabet which is the top of stack. This is because case $[r]$ does not depend on only alphabet but alphabet and top of stack. The definition of function f_r is to compute state S' from the determinization process. Both current state and top of stack must be considered to compute next state. This result in $\mathfrak{a}^{l+1}$ has to consider both $\mathfrak{a}^l$ and $\mathfrak{c}^l$. The disjunction over l' on the second line is used to find the condition "$S \overset{a'}{\mapsto} S'' \overset{a''}{\mapsto} S''' \in \nabla$" in the previous $\mathcal{P}$-automaton transition δ_0 to current $\mathcal{P}$-automaton transition δ_l.

Example 11. *Recall the same example, from Figure 5.1, the encoding of transition are:*

$$\mathcal{T}_c := (\mathfrak{a}_{1,1}^{l+1} \iff \mathfrak{a}_{1,1}^{l}\vee\mathfrak{a}_{2,1}^{l})\wedge\neg\mathfrak{a}_{1,2}^{l+1}\wedge\neg\mathfrak{a}_{2,1}^{l+1}\wedge(\mathfrak{a}_{2,2}^{l+1} \iff \mathfrak{a}_{1,1}^{l}\vee\mathfrak{a}_{1,2}^{l}\vee\mathfrak{a}_{2,1}^{l}\vee\mathfrak{a}_{2,2}^{l})\wedge$$
$$\neg\mathfrak{b}_{2}^{l+1}\wedge\mathfrak{b}_{1}^{l+1}\wedge\neg\mathfrak{c}_{f}^{l+1}\quad\wedge$$
$$(\mathfrak{c}_{1,1}^{l+1} \iff \mathfrak{a}_{1,1}^{l})\wedge(\mathfrak{c}_{1,2}^{l+1} \iff \mathfrak{a}_{1,2}^{l})\wedge(\mathfrak{c}_{2,1}^{l+1} \iff \mathfrak{a}_{2,1}^{l})\wedge(\mathfrak{c}_{2,2}^{l+1} \iff \mathfrak{a}_{2,2}^{l})$$

$$\mathcal{T}_i := (\mathfrak{a}_{1,1}^{l+1} \iff \mathfrak{a}_{1,1}^{l})\wedge(\mathfrak{a}_{1,2}^{l+1} \iff \mathfrak{a}_{1,2}^{l})\wedge(\mathfrak{a}_{2,1}^{l+1} \iff \mathfrak{a}_{2,1}^{l})\wedge(\mathfrak{a}_{2,2}^{l+1} \iff \mathfrak{a}_{2,2}^{l})\wedge$$
$$(\mathfrak{b}_{2}^{l+1} \iff \mathfrak{b}_{2}^{l})\wedge(\mathfrak{b}_{1}^{l+1} \iff \mathfrak{b}_{1}^{l})\wedge(\mathfrak{c}_{f}^{l+1} \iff \mathfrak{c}_{f}^{l})\quad\wedge$$
$$(\mathfrak{c}_{1,1}^{l+1} \iff \mathfrak{c}_{1,1}^{l})\wedge(\mathfrak{c}_{1,2}^{l+1} \iff \mathfrak{c}_{1,2}^{l})\wedge(\mathfrak{c}_{2,1}^{l+1} \iff \mathfrak{c}_{2,1}^{l})\wedge(\mathfrak{c}_{2,2}^{l+1} \iff \mathfrak{c}_{2,2}^{l})$$

$$\mathcal{T}_{r_0} := \neg\mathfrak{b}_{2}^{l}\wedge\neg\mathfrak{b}_{1}^{l}\wedge$$
$$(\mathfrak{a}_{1,1}^{l+1} \iff \mathfrak{a}_{1,1}^{l}\vee\mathfrak{a}_{1,2}^{l})\wedge\neg\mathfrak{a}_{1,2}^{l+1}\wedge(\mathfrak{a}_{2,1}^{l+1} \iff \mathfrak{a}_{2,1}^{l}\vee\mathfrak{a}_{2,2}^{l})\wedge\neg\mathfrak{a}_{2,2}^{l+1}\wedge$$
$$\neg\mathfrak{b}_{2}^{l+1}\wedge\neg\mathfrak{b}_{1}^{l+1}\wedge\mathfrak{c}_{f}^{l+1}\wedge\neg\mathfrak{c}_{1,1}^{l+1}\wedge\neg\mathfrak{c}_{1,2}^{l+1}\wedge\neg\mathfrak{c}_{2,1}^{l+1}\wedge\neg\mathfrak{c}_{2,2}^{l+1}$$

$$\mathcal{T}_r := (\mathfrak{b}_{2}^{l} \iff false)\wedge(\mathfrak{b}_{1}^{l} \iff true)\wedge$$
$$\neg\mathfrak{a}_{1,1}^{l+1}\wedge(\mathfrak{a}_{1,2}^{l+1} \iff (\mathfrak{a}_{1,1}^{l}\wedge\mathfrak{c}_{1,1}^{l})\vee(\mathfrak{a}_{2,1}^{l}\wedge\mathfrak{c}_{1,1}^{l})\vee(\mathfrak{a}_{2,1}^{l}\wedge\mathfrak{c}_{1,2}^{l}))\wedge$$
$$\neg\mathfrak{a}_{2,1}^{l+1}\wedge(\mathfrak{a}_{2,2}^{l+1} \iff (\mathfrak{a}_{1,1}^{l}\wedge\mathfrak{c}_{2,1}^{l})\vee(\mathfrak{a}_{2,1}^{l}\wedge\mathfrak{c}_{2,1}^{l})\vee(\mathfrak{a}_{2,1}^{l}\wedge\mathfrak{c}_{2,2}^{l}))\wedge$$
$$[\{(\mathfrak{c}_{1,1}^{l} \iff \mathfrak{a}_{1,1}^{0})\wedge(\mathfrak{c}_{1,2}^{l} \iff \mathfrak{a}_{1,2}^{0})\wedge(\mathfrak{c}_{2,1}^{l} \iff \mathfrak{a}_{2,1}^{0})\wedge(\mathfrak{c}_{2,2}^{l} \iff \mathfrak{a}_{2,2}^{0})\wedge$$
$$(\mathfrak{b}_{2}^{l+1} \iff \mathfrak{b}_{2}^{0})\quad\wedge(\mathfrak{b}_{1}^{l+1} \iff \mathfrak{b}_{1}^{0})\wedge(\mathfrak{c}_{f}^{l+1} \iff \mathfrak{c}_{f}^{0})\quad\wedge$$
$$(\mathfrak{c}_{1,1}^{l+1} \iff \mathfrak{c}_{1,1}^{0})\quad\wedge(\mathfrak{c}_{1,2}^{l+1} \iff \mathfrak{c}_{1,2}^{0})\wedge(\mathfrak{c}_{2,1}^{l+1} \iff \mathfrak{c}_{2,1}^{0})\wedge(\mathfrak{c}_{2,2}^{l+1} \iff \mathfrak{c}_{2,2}^{0})\}\vee$$
$$\ldots \textit{(until l' = l)}\ ]$$

where the [ ] bracket in $\mathcal{T}_r$ is the disjunction l' part in the $\mathcal{T}_r$ defined above. Actually, it must repeat the formula in $\{\}$ $l+1$ times from 0 to l.

Recall the vectors $\mathfrak{a}^0, \mathfrak{b}^0, \mathfrak{c}^0$ from Example 10, when compute for $\mathfrak{a}^1, \mathfrak{b}^1, \mathfrak{c}^1$:

- *If $\mathcal{T}_c$ is chosen, then vectors $\mathfrak{a}^1, \mathfrak{b}^1$ and $\mathfrak{c}^1$ will become $(true, false, false, true)$, $(false, true)$ and $(false, true, false, false, false)$ respectively that represent $\mathcal{P}$-automaton transition $\delta_1 = \{(1,1),(2,2)\} \overset{1}{\mapsto} \{(0,0)\}$.*
- *If $\mathcal{T}_i$ is chosen, then vectors $\mathfrak{a}^1, \mathfrak{b}^1$ and $\mathfrak{c}^1$ will become $(true, false, false, false)$, $(false, false)$ and $(true, false, false, false, false)$ respectively that represent $\mathcal{P}$-automaton transition $\delta_1 = \{(0,0)\} \overset{\perp}{\mapsto} f_{\mathcal{A}}$.*
- *If $\mathcal{T}_{r_0}$ is chosen, then vectors $\mathfrak{a}^1, \mathfrak{b}^1$ and $\mathfrak{c}^1$ will become the same as internal case.*
- *$\mathcal{T}_r$ cannot be chosen because $\mathfrak{b}_2^0$ and $\mathfrak{b}_1^0$ are false. If $\mathcal{T}_r$ is chosen, the formula will become unsatisfiable immediately.*

5.1.4 Encoding of non-universality checking

In this subsection, we define a propositional formula that is satisfiable if δ_k from a run $\delta_0 \overset{w}{\rightharpoonup}{}^* \delta_k$ is rejecting. That means the formula is satisfiable if M^d is not universal.

$$[rej(M^d)]^k := \bigwedge_{i=1}^{n} \bigwedge_{j \in F} \neg \mathfrak{a}_{i,j}^k$$

Because set of final states of M^d is $F' = \{S \in Q' \mid \Pi_2(S) \cap F \neq \emptyset\}$. Intuitively, state q of M^d is not a final state if every second component of pair in q is not in F regardless of the first component. Extend to $\mathcal{P}$-automaton transition δ, $\delta = p \overset{a}{\rightharpoonup} p'$ is rejecting if every second component of pair in p is not in F regardless of the first component. Thus, to check whether δ_k is rejecting, we need to check that every variable $\mathfrak{a}_{i,j}^k$ is false, for $j \in F$ regardless of i.

Finally, we sum all formulae above:

$$[notUniversal(M^d)]^k := [\mathcal{P}(M^d)]^k \wedge [rej(M^d)]^k$$

This formula $[notUniversal(M^d)]^k$ is used to check non-universality of VPAs. It is satisfiable if the given VPA is not universal. In particular, there exists $w \in \Sigma^k$ that a run of δ on w, $\delta_0 \overset{w}{\rightharpoonup}{}^* \delta_k$, ends up with a rejecting $\mathcal{P}$-automaton transition.

Note that because we start a bound from $k = 0$, if at some k, $[notUniversal(M^d)]^k$ is satisfiable, that means there exists a counter-example of input length $|w| = k$ and of course that counter-example is the shortest.

Example 12. *Recall the same example again, from figure 5.1, formula* $[rej(M^d)]^k := \neg\mathfrak{a}^k_{1,1} \wedge \neg\mathfrak{a}^k_{1,2} \wedge \neg\mathfrak{a}^k_{2,1} \wedge \neg\mathfrak{a}^k_{2,2}$. *This means the only state that is not a final state of* M^d *is* $q = \emptyset$ *(*$\mathfrak{a}_{i,j}$ *are all false).*

For this VPA, the shortest counter-example is '1212'. The formula $[notUniversal(M^d)]^k$ *is satisfiable if* $k = 4$ *with choice of transition* $\mathcal{T}_c$, $\mathcal{T}_r$, $\mathcal{T}_c$, $\mathcal{T}_r$, *respectively. Vectors* $\mathfrak{a}^4, \mathfrak{b}^4$ *and* $\mathfrak{c}^4$ *become* $(false, false, false, false)$, $(false, false)$, $(true, false, false, false, false)$ *respectively. This means* $\delta_4 = \emptyset \overset{\perp}{\mapsto} f_{\mathcal{A}}$ *which is a rejecting* $\mathcal{P}$*-automaton transition.*

5.2 Antichain for non-universality checking

In this section, we will explain theoritical backgrounds why we can apply antichain to the transition formula $\mathcal{T}$ when checking for non-universality, and this results in replacing some $\iff$ in $\mathcal{T}$ with $\implies$.

First, we define an ordering over $\mathcal{P}$-automaton transition.

Definition 14 (Partial ordering over $\mathcal{P}$-automaton transition of M^d)**.** *Let* $\delta_1 = p_1 \overset{a}{\mapsto} p'_1$ *and* $\delta_2 = p_2 \overset{a}{\mapsto} p'_2$ *be transitions of* $\mathcal{P}$*-automaton of determinized VPA* M^d, *we say* $\delta_1 \leq \delta_2$ *if* $p_1 \leq p_2$ *and* $p'_1 \leq p'_2$ *wrt ordering over states of* M^d *from definition 6 (Section 3.2.2). Note that if* $p'_1 = p'_2 = f_{\mathcal{A}}$, *we also say* $p'_1 \leq p'_2$.

From definition 14, we start with establishing theoritical background.

Lemma 10. *Let* $\delta_1 = p_1 \overset{a}{\mapsto} p'_1$, $\delta'_1 = p'_1 \overset{a'}{\mapsto} p''_1$, $\delta_2 = p_2 \overset{a}{\mapsto} p'_2$, *and* $\delta'_2 = p'_2 \overset{a'}{\mapsto} p''_2$ *be transitions of* $\mathcal{P}$*-automaton of* M^d. *Let* $\delta_1 \overset{a''}{\rightharpoonup} \delta''_1$ *and* $\delta_2 \overset{a''}{\rightharpoonup} \delta''_2$ *be transitions of* $\mathcal{P}$*-automaton transtion follow definition 9 (Section 4.4). If* $\delta_1 \leq \delta_2$ *and* $\delta'_1 \leq \delta'_2$, *then* $\delta''_1 \leq \delta''_2$.

Proof. We have four cases.

- $[c]$ transition: by definition, we have $p_1 = S_1, p_2 = S_2, \delta''_1 = Id_{R'_1} \overset{a''}{\mapsto} S_1$ and $\delta''_2 = Id_{R'_2} \overset{a''}{\mapsto} S_2$ that $S_1 \leq S_2$. From lemma 2 (Section 3.2.2), we have $Id_{R'_1} \leq Id_{R'_2}$. Thus, $\delta''_1 \leq \delta''_2$.
- $[i]$ transition: by definition, we have $p_1 = S_1, p_2 = S_2, \delta''_1 = S'_1 \overset{a}{\mapsto} p'_1$ and $\delta''_2 = S'_2 \overset{a}{\mapsto} p'_2$ that $S_1 \leq S_2$ and $p'_1 \leq p'_2$. From lemma 3 (Section 3.2.2), we have $S'_1 \leq S'_2$. Thus, $\delta''_1 \leq \delta''_2$.
- $[r_0]$ transition: by definition, we have $p_1 = S_1, p_2 = S_2, \delta''_1 = S'_1 \overset{\perp}{\mapsto} f_{\mathcal{A}}$ and $\delta''_2 = S'_2 \overset{\perp}{\mapsto} f_{\mathcal{A}}$ that $S_1 \leq S_2$. From lemma 4 (Section 3.2.2), we have $S'_1 \leq S'_2$. Thus, $\delta''_1 \leq \delta''_2$.

– $[r]$ transition: by definition, we have $p_1 = S_1, p'_1 = S''_1, p_2 = S_2, p'_2 = S''_2$ $\delta''_1 = S'_1 \overset{x'}{\mapsto} p''_1$ and $\delta''_2 = S'_2 \overset{x'}{\mapsto} p''_2$ that $S_1 \leq S_2$, $S''_1 \leq S''_2$ and $p''_1 \leq p''_2$. From lemma 5 (Section 3.2.2), we have $S'_1 \leq S'_2$. Thus, $\delta''_1 \leq \delta''_2$. □

Now, we extend lemma to a run of δ.

Lemma 11. *Let $\delta_0 = q_0 \overset{\perp}{\mapsto} f_{\mathcal{A}}$ be an unique initial $\mathcal{P}$-automaton transition. and let $\delta_0 \overset{w}{\rightharpoonup}{}^* \delta_k$ be a run of δ on w. Suppose $\delta_0 \overset{w}{\rightharpoonup}{}^* \delta'_k$ be a run of δ on w that $\delta_i \leq \delta'_i$ for every $1 \leq i \leq k$. For any word $w' \in \Sigma^*$, if a run of δ on ww' are $\delta_0 \overset{w}{\rightharpoonup}{}^* \delta_k \overset{w'}{\rightharpoonup}{}^* \delta_m$ and $\delta_0 \overset{w}{\rightharpoonup}{}^* \delta'_k \overset{w'}{\rightharpoonup}{}^* \delta'_m$, then $\delta_i \leq \delta'_i$ for every $1 \leq i \leq m$.*

Proof. We prove this lemma by induction on length $|w'|$. For $|w'| = 1$, the lemma holds immediately from lemma 10. For induction case, assume the lemma holds for w, we can prove for wa by using induction hypothesis and lemma 10 wrt type of a. □

Lemma 12. *Let δ_1 and δ_2 are $\mathcal{P}$-automaton transition that $\delta_1 \leq \delta_2$. If δ_2 is rejecting, then δ_1 is also rejecting.*

Proof. Let $\delta_1 = p_1 \overset{a}{\mapsto} p'_1$ and $\delta_2 = p_2 \overset{a}{\mapsto} p'_2$. Recall that δ_2 is rejecting if $\Pi_2(p_2) \cap F = \emptyset$. Note that $p_1 \leq p_2$, so $\Pi_2(p_1) \cap F = \emptyset$. □

From lemma 11 and lemma 12, we apply antichain to encoding of transitions of $\mathcal{P}$-automaton transition, i.e., apply to propositional formula $\mathcal{T}$. We change the logical operator $\iff$ (equivalence) in f_c, f_i, f_{r_0} and f_r to be just $\impliedby$ (implication).

$$\mathcal{T}_c := \bigvee_{a\in\Sigma_c} [\bigwedge_{i=1}^{n}\bigwedge_{j=1}^{n} f_c(i,j) \wedge (\mathfrak{b}^{l+1} \iff be(a))] \wedge (\mathfrak{c}^{l+1} \iff (false) + \mathfrak{a}^{l})$$

$$f_c(i,j) := \begin{cases} \mathfrak{a}_{i,j}^{l+1} \impliedby \bigvee_{j'}\bigvee_{i'=1}^{n} \mathfrak{a}_{i',j'}^{l} & \text{if } i=j \text{ and } \exists j' \in Q : j' \xrightarrow{a/+\gamma} i \in \Delta_c \\ \neg\mathfrak{a}_{i,j}^{l+1} & \text{otherwise} \end{cases}$$

$$\mathcal{T}_i := \bigvee_{a\in\Sigma_i} [\bigwedge_{i=1}^{n}\bigwedge_{j=1}^{n} f_i(i,j)] \wedge (\mathfrak{b}^{l+1} \iff \mathfrak{b}^{l}) \wedge (\mathfrak{c}^{l+1} \iff \mathfrak{c}^{l})$$

$$f_i(i,j) := \begin{cases} \mathfrak{a}_{i,j}^{l+1} \impliedby \bigvee_{j'} \mathfrak{a}_{i,j'}^{l} & \text{if } \exists j' \in Q : j' \xrightarrow{a} j \in \Delta_i \\ \neg\mathfrak{a}_{i,j}^{l+1} & \text{otherwise} \end{cases}$$

$$\mathcal{T}_{r_0} := (\neg\mathfrak{b}^{l}) \wedge \bigvee_{a\in\Sigma_r} [\bigwedge_{i=1}^{n}\bigwedge_{j=1}^{n} f_{r_0}(i,j)] \wedge (\neg\mathfrak{b}^{l+1}) \wedge (\mathfrak{c}^{l+1} \iff (true, false, \ldots, false))$$

$$f_{r_0}(i,j) := \begin{cases} \mathfrak{a}_{i,j}^{l+1} \impliedby \bigvee_{j'} \mathfrak{a}_{i,j'}^{l} & \text{if } \exists j' \in Q : j' \xrightarrow{a/\perp} j \in \Delta_r \\ \neg\mathfrak{a}_{i,j}^{l+1} & \text{otherwise} \end{cases}$$

$$\mathcal{T}_r := \bigvee_{a\in\Sigma_r}\bigvee_{a'\in\Sigma_c} [(\mathfrak{b}^{l} \iff be(a')) \wedge \bigwedge_{i=1}^{n}\bigwedge_{j=1}^{n} f_r(i,j)] \wedge$$
$$\bigvee_{l'=0}^{l} [(\mathfrak{c}'^{l} \iff \mathfrak{a}^{l'}) \wedge (\mathfrak{b}^{l+1} \iff \mathfrak{b}^{l'}) \wedge (\mathfrak{c}^{l+1} \iff \mathfrak{c}^{l'})]$$

$$f_r(i,j) := \begin{cases} \mathfrak{a}_{i,j}^{l+1} \impliedby \bigvee_{i'}\bigvee_{j''}\bigvee_{j'} [\mathfrak{c}_{i,j''}^{l} \wedge \mathfrak{a}_{i',j'}^{l}] & \text{if } \exists i', j', j'' \in Q, \exists \gamma \in \Gamma \setminus \perp : \\ & j'' \xrightarrow{a'/+\gamma} i' \in \Delta_c \text{ and } j' \xrightarrow{a/-\gamma} j \in \Delta_r \\ \neg\mathfrak{a}_{i,j}^{l+1} & \text{otherwise} \end{cases}$$

Remark. This change from $\iff$ to $\impliedby$ of a bit representation of set means that when checking the satisfiability, for each $\delta\ p \overset{a}{\mapsto} p''$, SAT-solver can construct $p' \overset{a}{\mapsto} p''$ that $p \leq p'$ instead. It is crucial to note that this is safe because if the formula is satisfiable that means $p' \overset{a}{\mapsto} p''$ can lead to the counter-example, from lemma 11 and 12 so does $p \overset{a}{\mapsto} p''$. Furthermore, if $p' \overset{a}{\mapsto} p''$ is 'too large' to lead to the counter-example, SAT-solver can construct it smaller and the smallest p' that SAT-solver can construct is actually $p' = p$.

Although this change seems very little, it results in a big decrease of number of variables and clauses when converse all formulae to CNF to be an input to SAT-solver (See Figure 6.1 and 6.2 in Chapter 6 for experimental results).

5.3 BMCi and BMCp for non-universality checking

Recall that BMCi unrolls M^d based on bounded transitions of $\mathcal{P}$-automaton transition ($\rightharpoonup$) but BMCp unrolled M^d based on bounded $\mathcal{P}$-automaton saturation process ($\Rightarrow$).

BMCi

Since BMCi uses $\rightharpoonup$ to unroll M^d, we can use the formula $[\mathcal{P}(M^d)]^k$ to encode transitions. Furthermore, checking non-universality process of BMCi can use the formula $[notUniversal(M^d)]^k$ directly.

We define an algorithm BMCi based on encoding above.

Algorithm 5: BMCi

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

Result: Universality of M

begin

- $k \leftarrow 0$
- Construct $[notUniversal(M^d)]^0 := init(\mathfrak{a}^0, \mathfrak{b}^0, \mathfrak{c}^0) \wedge [rej(M^d)]^0$
- Check for satisfiability of $[notUniversal(M^d)]^0$
- **while** $[notUniversal(M^d)]^k$ *is unsatisfiable* **do**
  - $k \leftarrow k + 1$
  - Construct $[notUniversal(M^d)]^k :=$
  - $init(\mathfrak{a}^0, \mathfrak{b}^0, \mathfrak{c}^0) \wedge \bigwedge_{i=0}^{k-1} \mathcal{T}((\mathfrak{a}^i, \mathfrak{b}^i, \mathfrak{c}^i), (\mathfrak{a}^{i+1}, \mathfrak{b}^{i+1}, \mathfrak{c}^{i+1})) \wedge [rej(M^d)]^k$
  - Check for satisfiability of $[notUniversal(M^d)]^k$
- **end**
- **return** *False*

end

BMCp

BMCp transitions are based on $\mathcal{P}$-automaton saturation process $\mathcal{A}_0 \Rightarrow \ldots \Rightarrow \mathcal{A}_k$ (denoted by $\mathcal{A}_0 \Rightarrow_k^* \mathcal{A}_k$), where $\mathcal{A}_0$ is an unique initial $\mathcal{P}$-automaton of M^d accepting the initial configuration $(Id_{Q_0}, \bot)$, $\mathcal{A}_0 = (\{Id_{Q_0}\}, \Sigma_c, \{Id_{Q_0}\}, \{f_\mathcal{A}\}, \{(Id_{Q_0}, \bot, f_\mathcal{A})\})$.

Theorem 7. *M^d is not universal iff there exists $k \in \mathbb{N}$ that $\mathcal{A}_0 \Rightarrow_k^* \mathcal{A}_k$ and $\mathcal{A}_k$ contains a rejecting state $q \notin F'$*

we define a new propositional formula:

$$[\mathcal{P}'(M^d)]^k := init(\mathfrak{a}^0, \mathfrak{b}^0, \mathfrak{c}^0) \wedge \bigwedge_{l=0}^{k-1} \mathcal{T}'((\mathbb{A}^l, \mathbb{B}^l, \mathbb{C}^l), (\mathfrak{a}^{l+1}, \mathfrak{b}^{l+1}, \mathfrak{c}^{l+1}))$$

to represent $\mathcal{A}_0 \Rightarrow_k^* \mathcal{A}_k$, and a formula for non-universality checking for BMCp is:

$$[notUniversal'(M^d)]^k := [\mathcal{P}'(M^d)]^k \wedge [rej(M^d)]^k$$

The formula *init* is the same as that in BMCi, since the initial $\mathcal{P}$-automaton $\mathcal{A}_0$ contains an initial $\mathcal{P}$-automaton transition only.

The formula $\mathcal{T}'$ that represents a transition $\mathcal{A}_l \Rightarrow \mathcal{A}_{l+1}$, is defined as:

$$\mathcal{T}'((\mathbb{A}^l, \mathbb{B}^l, \mathbb{C}^l), (\mathfrak{a}^{l+1}, \mathfrak{b}^{l+1}, \mathfrak{c}^{l+1})) := union^l(\mathbb{A}^l, \mathbb{B}^l, \mathbb{C}^l) \wedge \mathcal{T}((\mathbb{A}^l, \mathbb{B}^l, \mathbb{C}^l), (\mathfrak{a}^{l+1}, \mathfrak{b}^{l+1}, \mathfrak{c}^{l+1}))$$

Intuitively, we can look at the transition $\delta \rightharpoonup \delta'$ as δ' is an immediate successor of δ, i.e., if $\delta \rightharpoonup \delta'$ and δ is in $\mathcal{A}_i$, then δ' can be in $\mathcal{A}_{i+1}$ (up to choice of saturation rules $[c], [i], [r_0]$ or $[r]$). Therefore, we can extend the formula $\mathcal{T}$ to saturation process $\Rightarrow$ of $\mathcal{P}$-automaton, with one condition that $\mathfrak{a}^{l+1}, \mathfrak{b}^{l+1}, \mathfrak{c}^{l+1}$ must be computed from $(\mathfrak{a}^0, \mathfrak{b}^0, \mathfrak{c}^0), (\mathfrak{a}^1, \mathfrak{b}^1, \mathfrak{c}^1), \ldots, (\mathfrak{a}^l, \mathfrak{b}^l, \mathfrak{c}^l)$ (not only from $\mathfrak{a}^l, \mathfrak{b}^l, \mathfrak{c}^l$ as in $[\mathcal{P}(M^d)]^k$).

We define the formula $union^l$ as:

$$union^l(\mathbb{A}^l, \mathbb{B}^l, \mathbb{C}^l) := \bigvee_{i=0}^{l} [(\mathbb{A}^l \iff \mathfrak{a}^i) \wedge (\mathbb{B}^l \iff \mathfrak{b}^i) \wedge (\mathbb{C}^l \iff \mathfrak{c}^i)]$$

The new vectors $\mathbb{A}^l, \mathbb{B}^l, \mathbb{C}^l$ and formula $union^l$ are used to combine $(\mathfrak{a}^0, \mathfrak{b}^0, \mathfrak{c}^0)$ to $(\mathfrak{a}^l, \mathfrak{b}^l, \mathfrak{c}^l)$ in order to compute $\mathfrak{a}^{l+1}, \mathfrak{b}^{l+1}, \mathfrak{c}^{l+1}$. Intuitively, $\mathbb{A}^l, \mathbb{B}^l, \mathbb{C}^l$ can choose to be $\mathfrak{a}^i, \mathfrak{b}^i, \mathfrak{c}^i$ for $0 \leq i \leq l$ in order to compute $\mathfrak{a}^{l+1}, \mathfrak{b}^{l+1}, \mathfrak{c}^{l+1}$.

Example 13. *Recall VPA from figure 5.1, suppose $k = 2$ the formula $[\mathcal{P}'(M^d)]^k$ is:*

$$\begin{aligned}[\mathcal{P}'(M^d)]^2 := \quad & init^0(\mathfrak{a}^0, \mathfrak{b}^0, \mathfrak{c}^0) \wedge union(\mathbb{A}^0, \mathbb{B}^0, \mathbb{C}^0) \wedge \mathcal{T}((\mathbb{A}^0, \mathbb{B}^0, \mathbb{C}^0), (\mathfrak{a}^1, \mathfrak{b}^1, \mathfrak{c}^1)) \wedge \\ & union^1(\mathbb{A}^1, \mathbb{B}^1, \mathbb{C}^1) \wedge \mathcal{T}((\mathbb{A}^1, \mathbb{B}^1, \mathbb{C}^1), (\mathfrak{a}^2, \mathfrak{b}^2, \mathfrak{c}^2))\end{aligned}$$

where $init^0$ and $init^1$ are:

$$\begin{aligned}union^0(\mathbb{A}^0, \mathbb{B}^0, \mathbb{C}^0) :=& (\mathbb{A}^0 \iff \mathfrak{a}^0) \wedge (\mathbb{B}^0 \iff \mathfrak{b}^0) \wedge (\mathbb{C}^0 \iff \mathfrak{c}^0) \\ union^1(\mathbb{A}^1, \mathbb{B}^1, \mathbb{C}^1) :=& (\mathbb{A}^1 \iff \mathfrak{a}^0) \wedge (\mathbb{B}^1 \iff \mathfrak{b}^0) \wedge (\mathbb{C}^1 \iff \mathfrak{c}^0) \\ & (\mathbb{A}^1 \iff \mathfrak{a}^1) \wedge (\mathbb{B}^1 \iff \mathfrak{b}^1) \wedge (\mathbb{C}^1 \iff \mathfrak{c}^1)\end{aligned}$$

Note that $union^0$ is not neccessary, because $\mathcal{A}_0$ has only one transition $\mathfrak{a}^0, \mathfrak{b}^0, \mathfrak{c}^0$.

Example 14. *We show an example that* $[\mathcal{P}(M^d)]$ *and* $[\mathcal{P}'(M^d)]^k$ *are different. For a VPA* M *in a Figure 5.2, the initial* $\mathcal{P}$*-automaton transition* $\delta_0(=\mathcal{A}_0)$ *is* $\{(1,1)\} \overset{\perp}{\mapsto} f_{\mathcal{A}}$*. A* $\mathcal{P}$*-automaton transition* $\{(1,3)\} \overset{1}{\mapsto} \{(1,3)\}$ *is in* $[\mathcal{P}(M^d)]^5$ *but* $[\mathcal{P}'(M^d)]^4$*. This is because* $\delta_0 \overset{12112}{\rightharpoonup}{}^* (\{(1,3)\} \overset{1}{\mapsto} \{(1,3)\})$ *but* $\{(1,3)\} \overset{1}{\mapsto} \{(1,3)\}$ *appears since* $\mathcal{A}_4$*.* $\{(1,5)\} \overset{\perp}{\mapsto} f_{\mathcal{A}}$ *is in* $[\mathcal{P}(M^d)]^{10}$ *(input is '1211211222') but* $[\mathcal{P}'(M^d)]^6$*, and* $\{(1,5)\} \overset{1}{\mapsto} \{(1,3)\}$ *is in* $[\mathcal{P}(M^d)]^{13}$ *(input is '1211211211222') but also* $[\mathcal{P}'(M^d)]^6$*,*

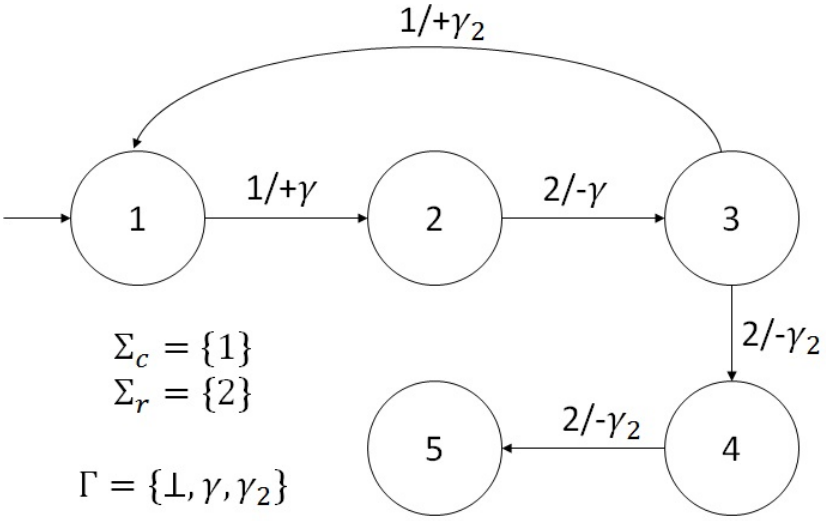


Figure 5.2: An example of VPA that shows differences of $[\mathcal{P}(M^d)]^k$ and $[\mathcal{P}'(M^d)]^k$

5.4 BMCp for universality checking

Remark. It is crucial to note that if there exists a counter-example, BMCi method will eventually find it. The formula $[notUniversal(M^d)]^k$ will be satisfiable at some bound k. But if the given VPA is universal, we need to enlarge bound k until we can confirm that all reachable states of VPA are considered. The smallest k that has this property is called *reachability diameter* [9, 29]. However, for a VPA with an infinite state space (infinite configuration), there is no such finite reachability diameter [8]. Therefore, BMCi is not complete. It will not terminate for universal cases. Therefore, for universality checking only BMCp will be considered.

For a bound k, let a $\mathcal{P}$-automaton saturation process proceeds $\mathcal{A}_0 \Rightarrow^* \mathcal{A}_k$. We can conclude that M^d is universal if no counter-examples are found in $\mathcal{A}_k$ and $\mathcal{A}_k$ is saturated (i.e., $\mathcal{A}_k = \mathcal{A}_{k-1}$).

We define a propositional formula:

$$[notConverged(M^d)]^k := \bigwedge_{i=1}^{k} \bigwedge_{j=0}^{i-1} \neg[(\mathfrak{a}^i \iff \mathfrak{a}^j) \wedge (\mathfrak{b}^i \iff \mathfrak{b}^j) \wedge (\mathfrak{c}^i \iff \mathfrak{c}^j)]$$

This formula is satisfiable if $\mathcal{A}_k$ is not converged, that is $\mathcal{A}_i \neq \mathcal{A}_j$ for every $i \leq k$ and $j \leq i$.

Finally, the formula for universality checking is:

$$[notSaturated(M^d)]^k := [\mathcal{P}'(M^d)]^k \wedge [notConverged(M^d)]^k$$

This formula is satisfiable if $\mathcal{A}_k$ is still not saturated, and unsatisfiable if $\mathcal{A}_k$ is saturated and universal. Note that if $[notSaturated(M^d)]^k$ is unsatisfiable we can conclude that M^d is universal immediately without checking $[notUniversal'(M^d)]^k$, since $\mathcal{A}_k = \mathcal{A}_{k-1}$ and $\mathcal{A}_{k-1}$ does not have any rejecting $\mathcal{P}$-automaton transition (if $\mathcal{A}_{k-1}$ has rejecting $\mathcal{P}$-automaton transitions, we could already conclude that M^d is not universal).

Example 15. *We show an example for saturation (without considering universality). The M^d in Figure 5.2 is saturated in 7 steps with process:*

$$\begin{aligned}
\nabla_0 &= \{((1,1), \bot, f_{\mathcal{A}})\} \\
\nabla_1 &= \nabla_0 \cup \{(\{(2,2)\}, ((1,1),1), \{(1,1)\})\} \\
\nabla_2 &= \nabla_1 \cup \{(\{(1,3)\}, \bot, f_{\mathcal{A}})\} \\
\nabla_3 &= \nabla_2 \cup \{(\{(1,1)\}, ((1,3),1), \{(1,3)\})\} \\
\nabla_4 &= \nabla_3 \cup \{(\{(1,3)\}, ((1,3),1), \{(1,3)\})\} \\
\nabla_5 &= \nabla_4 \cup \{(\{(1,4)\}, \bot, f_{\mathcal{A}}), (\{(1,4)\}, ((1,3),1), \{(1,3)\})\} \\
\nabla_6 &= \nabla_5 \cup \{(\{(1,5)\}, \bot, f_{\mathcal{A}}), (\{(1,5)\}, ((1,3),1), \{(1,3)\})\} \\
\nabla_7 &= \nabla_6
\end{aligned}$$

Thus, when $i = 7$, whatever $\mathfrak{a}^7, \mathfrak{b}^7, \mathfrak{c}^7$ be, it will be equal to one of $(\mathfrak{a}^0, \mathfrak{b}^0, \mathfrak{c}^0) \ldots (\mathfrak{a}^6, \mathfrak{b}^6, \mathfrak{c}^6)$ and $[notConverged(M^d)]^7$ and $[notSaturated(M^d)]^7$ are unsatisfiable.

5.5 Antichain for universality checking

From lemma 6 and 7 (Section 3.2.2), we conclude that it is sufficient to keep only minimal configurations in order to check universality of VPA. In antichain algorithm, we also keep only minimal configurations and remove the larger configurations from $\mathcal{P}$-automaton (algorithm 3 Section 3.2.2). We apply this to BMCp by changing $(\mathfrak{a}^i \iff \mathfrak{a}^j)$ in $[notConverged(M^d)]^k$ to $(\mathfrak{a}^i \implies \mathfrak{a}^j)$.

$$[notConverged(M^d)]^k := \bigwedge_{i=1}^{k} \bigwedge_{j=0}^{i-1} \neg[(\mathfrak{a}^i \implies \mathfrak{a}^j) \wedge (\mathfrak{b}^i \iff \mathfrak{b}^j) \wedge (\mathfrak{c}^i \iff \mathfrak{c}^j)]$$

where $\implies$ of two vectors with the same length denotes conjunction of $\implies$ of all indices (i.e., $(\mathfrak{a}^i_{1,1} \implies \mathfrak{a}^j_{1,1}) \ldots (\mathfrak{a}^i_{n,n} \implies \mathfrak{a}^j_{n,n})$).

Since we use a bit representation to represent sets, a logical operation $\implies$ over two vectors denotes a subset relation of two sets. In particular, $\mathfrak{a}^i \implies \mathfrak{a}^j$ denotes $q^i \subseteq q^j$.

Now, the formula $[notConverged(M^d)]^k$ is unsatisfiable if all $\mathcal{P}$-automaton transitions in $\mathcal{A}_k$ are not minimal. This is similar to an Algorithm 3 (Section 3.2.2). If all configurations that we have to check are not minimal, we can discard them and $\mathcal{P}$-automaton is converged.

Although this change seems very little, it results in a decrease of number of bounds until convergence when checking universal VPAs. (See Figure 6.3 in Chapter 6 for experimental results).

Finally, we define an algorithm BMCp as below:

In conclusion, Differences of BMCi and BMCp are:

- how to bound M^d, BMCi bounds M^d based on transitions of $\mathcal{P}$-automaton transition but BMCp bounds M^d based on $\mathcal{P}$-automaton saturation process.
- encoding of transitions, BMCi uses the formula $[\mathcal{P}(M^d)]^k$ but BMCp uses $[\mathcal{P}'(M^d)]^k$ which has additional vectors and clauses.
- BMCp has to check for satisfiability of formulae two times for each bound k, that are check for universality and for non-universality, but BMCi checks for non-universality only.
- BMCi is not a complete method but BMCp is. BMCi can check for non-universality, but cannot check for universality of VPAs.

Algorithm 6: BMCp

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

Result: Universality of M

begin

$k \leftarrow 0$

Construct $[notUniversal'(M^d)]^0 := init(\mathfrak{a}^0, \mathfrak{b}^0, \mathfrak{c}^0) \wedge [rej(M^d)]^0$

Check for satisfiability of $[notUniversal'(M^d)]^0$

while $[notUniversal'(M^d)]^k$ *is unsatisfiable* **do**

$k \leftarrow k + 1$

Construct $[notSaturated(M^d)]^k := [\mathcal{P}'(M^d)]^k \wedge [notConverged(M^d)]^k$

Check for satisfiability of $[notSaturated(M^d)]^k$

if $[notSaturated(M^d)]^k$ *is unsatisfiable* **then**

return *True*

end

Construct $[notUniversal'(M^d)]^k := [\mathcal{P}'(M^d)]^k \wedge [rej(M^d)]^k$

Check for satisfiability of $[notUniversal'(M^d)]^k$

end

return *False*

end

Chapter 6

Experimental results

We have implemented the algorithms BMCi and BMCp. The packages are implemented in Java 1.7.0 on Windows 7 and we use the latest version of MiniSat [25, 26] (i.e., minisat-2.2.0) for checking satisfiability of the formula. The coperation between Java and MiniSat is that the Java program provides a CNF propositional formula as a text file to be an input for MiniSat according to Algorithm 5 (Section 5.3) and 6 (Section 5.5). Then the Java program calls MiniSat to solve for the satisfiability. After getting the result, either reports whether the given VPA is universal or not universal, or enlarge the bound.

To compare efficiency between BMCi and BMCp, we run our implementations on randomly generated VPA. All tests are performed on a laptop PC, which is equipped with Intel®Core^{TM} i5-2430M 2.4 GHz and 4 GB of memory.

During the experiments, we fix the size of input alphabet to $|\Sigma_c| = |\Sigma_i| = |\Sigma_r| = 2$ and the size of stack symbol to $|\Gamma| = 3$. The density of final states $f = \frac{|F|}{|Q|}$ (resp. density of initial states $i = \frac{|Q_0|}{|Q|}$) is the ratio of number of final states (resp. initial states) over the number of states of VPA.

Experiment1: non-universal r-random VPAs

We first set the parameters of the tests as follows:

Definition 15 (r-random)**.** *An r-random VPA is a VPA that is generated by: first randomly generated transitions labeled with a input symbol (and stack symbol if the input symbol is in return alphabet), then for each transition randomly select two states to be source and destination of the transition. The density of transitions* $r = \frac{|\Delta|}{|Q|}$ *is average number of transitions for each state.*

	number of states					
BMCi w/ antichain	5	10	15	20	25	30
number of success	50	50	49	44	39	33
number of timeout	0	0	1	6	11	17
total time(sec)	5	79	54	158	474	442
	number of states					
BMCi w/o antichain	5	10	15	20	25	30
number of success	50	49	49	41	23	17
number of timeout	0	1	1	9	27	33
total time(sec)	6	45	219	354	447	378
	number of states					
BMCp w/ antichain	5	10	15	20	25	30
number of success	50	49	49	43	33	30
number of timeout	0	1	1	7	17	20
total time(sec)	9	50	110	228	501	637
	number of states					
BMCp w/ antichain	5	10	15	20	25	30
number of success	50	48	48	40	21	16
number of timeout	0	2	2	10	29	34
total time(sec)	11	82	327	439	499	512

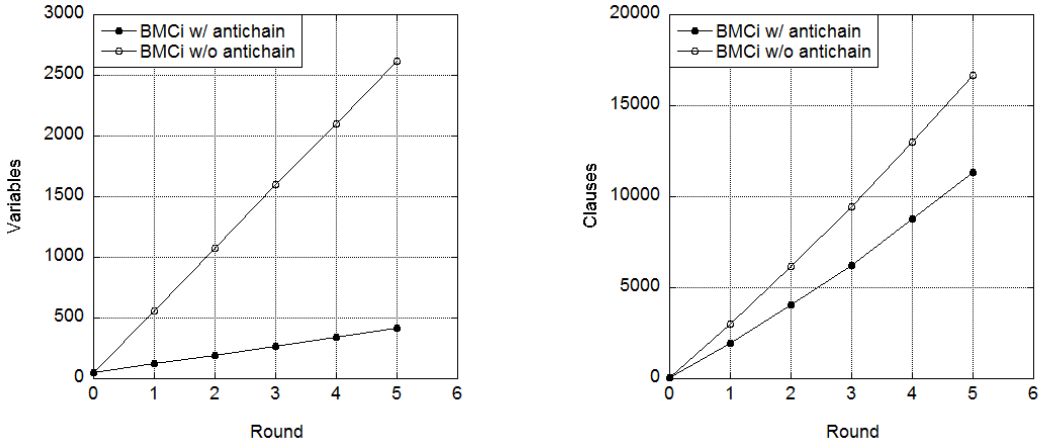
Table 6.1: Universality checking for not universal VPA generated by r-random (50 automata for each sample)

We ran our tests on r-random VPAs. We set the density of final states and initial state $f = i = 0.6$, and the density of transitions $r = 20$. This makes the number of transitions $|\Delta| = 20 \times |Q|$. We have tried VPA sizes from 5 to 30. We generated 50 VPAs for each sample point and the timeout is set to 60 seconds. The results are shown in table 6.1.

We found that all VPAs are not universal, since r-random VPAs are almost always not universal from random graph theory. The experimental results show that BMCi with antichain algorithm is the best among 4 algorithms. Note that this is not a fair comparison for BMCp, since BMCp has to check for universal before checking non-universal. However, even if BMCp has to check for universal before checking non-universal, BMCp with antichain algorithm is still better than BMCi without antichain algorithm.

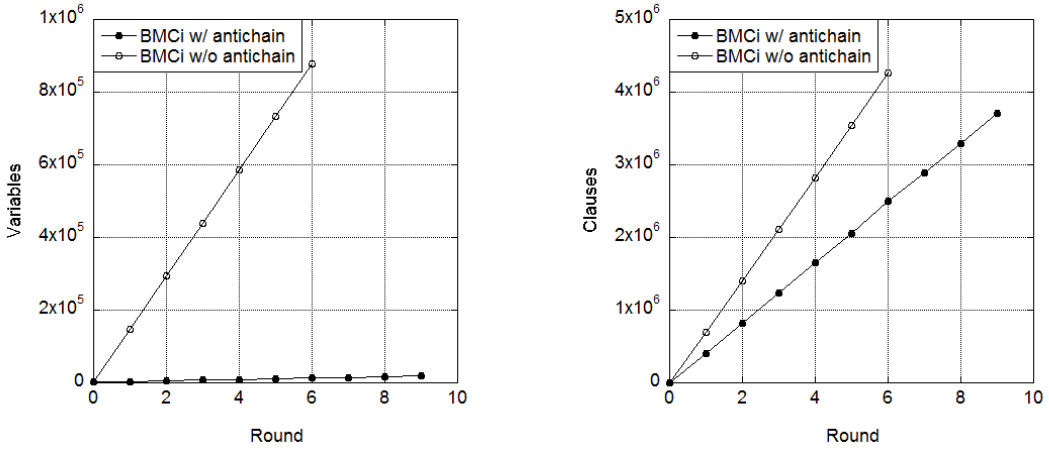
Figure 6.1 shows benefits of antichain algorithm in reducing the number of variables and clauses. This is from testing instances with size 5. The maximal benefit appears in testing

instances with size 30, as shown in Figure 6.2. The results show that in maximal case (6th round of size 30 instances) antichain algorithm can reduce number of variables to only 1.5 percent (approx. 0.8 million variables removed), and number of clauses decreases to 58 percent (approx. 1.8 million clauses removed). All of the removed variables and clauses are from conversion-to-CNF procedure [35] (using Tseitin conversion [36]). Since antichain change the transition formula from $\iff$ to $\implies$ (Section 5.2), the number of fresh variables and clauses introduced by the conversion procedure decreases a lot.



(a) number of variables (b) number of clauses

Figure 6.1: number of variables and clauses from test cases r-random of size 5



(a) number of variables (b) number of clauses

Figure 6.2: number of variables and clauses from test cases r-random of size 30

Experiment2: universal r-regular random VPAs

We define an r-regular random VPAs in order to test the universal cases as below:

Definition 16 (r-regular random)**.** *An r-regular random VPA is generated by: first generate states, add transitions labeled with every input symbol (and stack symbol in case of return alphabet) to every state, and randomly generate destination for each transition. The density of transitions r depends on state and input symbol, $r(q, a)$ is a number of transitions from state q reading input symbol a for $a \in \Sigma_c$ and $a \in \Sigma_i$, and $r(q, a, \gamma)$ is a number of transitions from state q reading input symbol a and top of stack is γ for $a \in \Sigma_r$. Intuitively, for VPAs generated with r-regular random, every state must have r transitions with every input symbol (and stack symbol in case of return alphabet).*

During the experiments, we set the density of final states and initial states $f = i = 1$, and $r(q, a) = r(q, b, \gamma) = r(q, c) = 2$ for each $a \in \Sigma_c, b \in \Sigma_r, c \in \Sigma_i$ and $\gamma \in \Gamma$. This means each state q has totally 20 transitions from it.

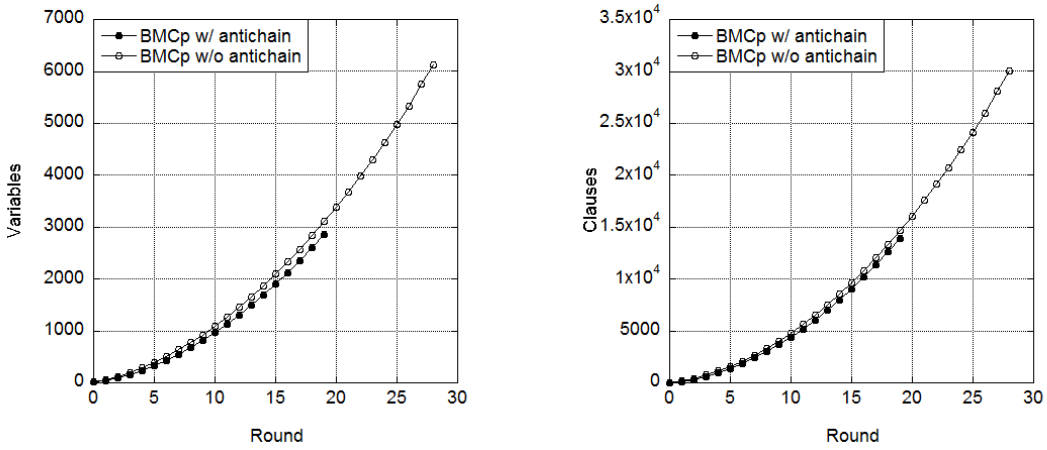
VPAs generated by r-regular random parameters is always universal, since every state is a final state and has transitions for every input symbol and stack symbol.

Table 6.2 shows experimental results on 50 r-regular random VPAs with the timeout set to 300 seconds. We tested only BMCp, because BMCi is not complete. Unfortunately, our current BMCp implementations can solve instances of only 2 and 3 states. Nevertheless, we can see a benefit of antichain algorithm clearly. BMCp with antichain algorithm outperforms BMCp without antichain algorithm.

	number of states	
BMCp w/ antichain	2	3
number of success	46	14
number of timeout	4	36
total time(sec)	56	1062
	number of states	
BMCp w/o antichain	2	3
number of success	36	0
number of timeout	14	0
total time(sec)	257	-

Table 6.2: Universality checking for universal VPA generated by r-regular random (50 automata for each sample)

BMC with antichain algorithm is faster than BMC without antichain algorithm from the



(a) number of variables (b) number of clauses

Figure 6.3: number of variables and clauses from test cases r-regular random of size 2

following two reasons:

- Antichain applied to transitions (Section 5.2) makes the number of variables and clauses generated by CNF conversion fewer. This result enables BMC to search with deeper bound. As shown in Figure 6.1 and 6.2, the number of the bound for BMC with antichain until timeout is greater than that without antichian.
- Antichain applied to saturation checking (Section 5.5) makes $\mathcal{P}$-automaton convergence faster. As shown in Figure 6.3, the number of the bound for BMC with antichain until convergence is less than that without antichain. Note that the number of variables and clauses are not much different, since the number of states and transitions of VPAs are few.

Remark. Figure 6.4 refers to the most recent results of other tools (Fig. 2 of [14]). Despite differences in test cases and machine, we have to admit that our current algorithms and implementations are still behind.

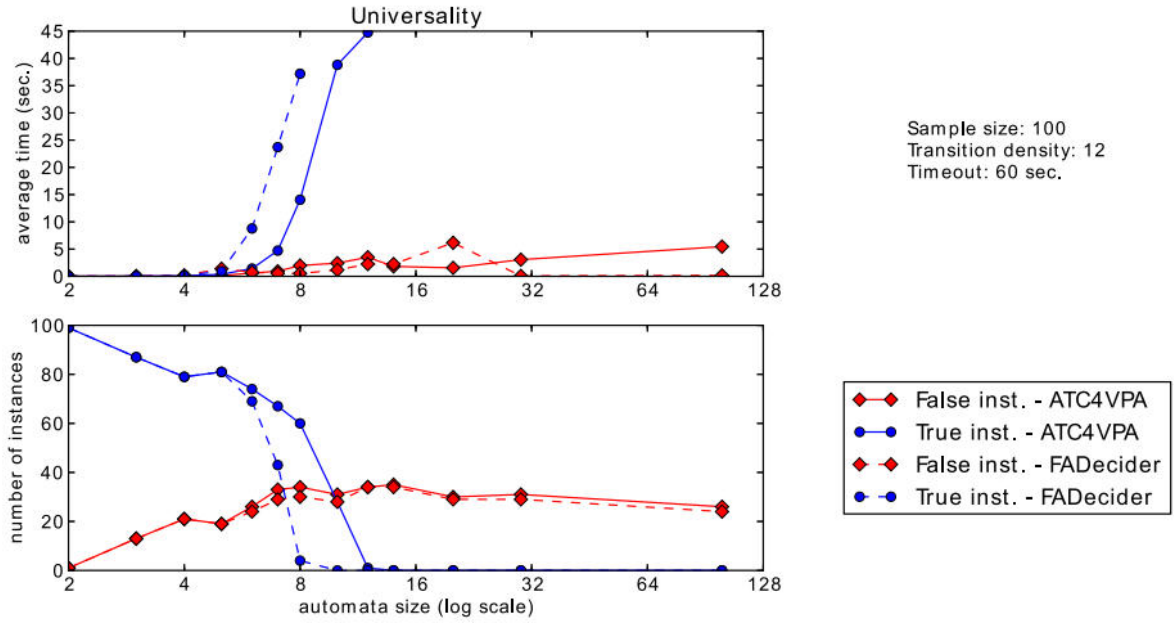


Figure 6.4: Results of other tools (Fig. 2 of [14])

Chapter 7

Related work

The concept of antichains is firstly applied to the universality and inclusion checking of finite automata and emptiness checking of alternating finite automata in [19]. However, their tool called ALASKA [20] seems not being actively developed anymore. Later the concept of antichains is extened to tree automata [13] and Büchi automata [21]. Recently, antichain is also applied to equivalence checking of finite automata [11].

Visibly pushdown automata were firstly proposed by Alur and Madhusudan in [5] and investigated in [4]. Before that it seems this type of automata was studied under the name of input driven automata [39, 24] . The visibly pushdown class nicely extends the closure property of finite automata. This is mainly because VPA can be performed determinization.

The universality checking of VPA requires $O(2^{n^2})$ because of determinization procedure [38]. The first known implementation VPAlib [32] only works with just few states. Currently, there are several tools solving this problem, e.g.,

- **OpenNWA** a nested-word automaton library that provides the standard boolean operations [23],
- **VPAchecker** a package based on VPAlib, deciding universality and inclusion of VPAs by using on-the-fly and antichain algorithm [37],
- **FADecider** a package for deciding universality and inclusion checking of VPAs by using Ramsey-based methods [28],
- **ATC4VPA** a package for deciding universality and inclusion checking of VPAs by encoding VPAs as an trees acceptors and using antichain algorithm [14].

From above tools, only VPAchecker and ATC4VPA are only tools that using antichain.

However, VPAchecker seems to have some bugs reported in [14].

BMC was introduced by Biere [9] as a SAT-based efficient technique for finding bugs. It became a well-known model checking method shortly after its introduction. The BMC technique that is most related to our work, is BMC for checking reachability of pushdown system [8] using summarization [34, 7].

Chapter 8

Conclusion and future works

In our article, we proposed two algorithms to tackle the universality checking of VPAs. These algorithms combine antichain and BMC technique in order to overcome state space explosion and optimize size of search space. We unrolled determinization procedure by transitions of $\mathcal{P}$-automaton transition based on input word length and $\mathcal{P}$-automaton saturation process. We implemented our algorithms and evaluated with existing antichain algorithm. Although experiments show that our current algorithm implementations are slower. They show that an antichain algorithm boosts BMC than a standard BMC without antichain algorithm. This is the first attempt of BMC technique for the universality checking of VPAs, and we expect future improvements of CNF encoding.

This work suggests several future works. We first plan to continue working on BMC approach. We should emphasize that we need to improve our algorithms as well as implementations to be able to check larger examples. At this moment, the encoding of determinized VPA and $\mathcal{P}$-automaton are rather naive (e.g., encoding set by bit representation). That would be a reason why our implementations are not fast. It would be interesting to explore a more compact encoding or a more effective method to unroll VPA. We also would like to extend our algorithms to inclusion checking of VPAs. Another challenge would be to consider combination of antichain and simulation, since this combination is successful for finite automata and tree automata [1]. Simulation can help antichain to reduce a large portion of unneccessary search space. The last would be applications to practical case studies (rather than randomly generated). XML processing and/or program verification will be such examples.

Bibliography

[1] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. When simulation meets antichains. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 158–174. Springer, 2010.

[2] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):786–818, 2005.

[3] Rajeev Alur, Kousha Etessami, and P Madhusudan. A temporal logic of nested calls and returns. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 467–481. Springer, 2004.

[4] Rajeev Alur, Viraj Kumar, P Madhusudan, and Mahesh Viswanathan. Congruences for visibly pushdown languages. In *Automata, Languages and Programming*, pages 1102–1114. Springer, 2005.

[5] Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211. ACM, 2004.

[6] Rajeev Alur and Parthasarathy Madhusudan. Adding nesting structure to words. *Journal of the ACM (JACM)*, 56(3):16, 2009.

[7] Gérard Basler, Daniel Kroening, and Georg Weissenbacher. Sat-based summarization for boolean programs. In *Model Checking Software*, pages 131–148. Springer, 2007.

[8] Gérard Basler, Daniel Kroening, and Georg Weissenbacher. A complete bounded model checking algorithm for pushdown systems. In *Hardware and Software: Verification and Testing*, pages 202–217. Springer, 2008.

[9] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999.

[10] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[11] Filippo Bonchi and Damien Pous. Checking nfa equivalence with bisimulations up to congruence. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 457–468. ACM, 2013.

[12] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR'97: Concurrency Theory*, pages 135–150. Springer, 1997.

[13] Ahmed Bouajjani, Peter Habermehl, Lukáš Holík, Tayssir Touili, and Tomáš Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In *Implementation and Applications of Automata*, pages 57–67. Springer, 2008.

[14] Véronique Bruyere, Marc Ducobu, and Olivier Gauwin. Visibly pushdown automata: universality and inclusion via antichains. In *Language and Automata Theory and Applications*, pages 190–201. Springer, 2013.

[15] Gianpiero Cabodi, Paolo Camurati, and Stefano Quer. Can bdds compete with sat solvers on bounded model checking? In *Proceedings of the 39th annual Design Automation Conference*, pages 117–122. ACM, 2002.

[16] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[17] Edmund M Clarke, E Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11):74–84, 2009.

[18] Louis Comtet. *Advanced Combinatorics: The art of finite and infinite expansions*. Springer, 1974.

[19] Martin De Wulf, Laurent Doyen, Thomas A Henzinger, and J-F Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Computer Aided Verification*, pages 17–30. Springer, 2006.

[20] Martin De Wulf, Laurent Doyen, Nicolas Maquet, and Jean-Francois Raskin. Alaska. In *Automated Technology for Verification and Analysis*, pages 240–245. Springer, 2008.

[21] Laurent Doyen and Jean-Francois Raskin. Improved algorithms for the automata-based approach to model-checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 451–465. Springer, 2007.

[22] Laurent Doyen and Jean-Francois Raskin. Antichain algorithms for finite automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 2–22. Springer, 2010.

[23] Evan Driscoll, Aditya Thakur, and Thomas Reps. Opennwa: A nested-word automaton library. In *Computer Aided Verification*, pages 665–671. Springer, 2012.

[24] Patrick W Dymond. Input-driven languages are in log n depth. *Information processing letters*, 26(5):247–250, 1988.

[25] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2004.

[26] Niklas Een and Niklas Sörensson. Minisat: A sat solver with conflict-clause minimization. *Sat*, 5, 2005.

[27] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification*, pages 232–247. Springer, 2000.

[28] Oliver Friedmann, Felix Klaedtke, and Martin Lange. Ramsey goes visibly pushdown. Technical report, Tech. rep, 2012.

[29] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In *Verification, Model Checking, and Abstract Interpretation*, pages 298–309. Springer, 2003.

[30] Viraj Kumar, Parthasarathy Madhusudan, and Mahesh Viswanathan. Visibly pushdown automata for streaming xml. In *Proceedings of the 16th international conference on World Wide Web*, pages 1053–1062. ACM, 2007.

[31] Christof Löding, Carsten Lutz, and Olivier Serre. Propositional dynamic logic with recursive programs. *Journal of Logic and Algebraic Programming*, 73(1):51–69, 2007.

[32] Dong Ha Nguyen and Mario Sudholt. Vpa-based aspects: better support for aop over protocols. In *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, pages 167–176. IEEE, 2006.

[33] Corin Pitcher. Visibly pushdown expression effects for xml stream processing. *Programming Language Technologies for*, 1060:1–14, 2005.

[34] M Pnueli. Two approaches to interprocedural data flow analysis. *Program flow analysis: theory and applications*, pages 189–234, 1981.

[35] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 74. Prentice hall Englewood Cliffs, 1995.

[36] Grigori S Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2(115-125):10–13, 1968.

[37] Tang Van Nguyen and Hitoshi Ohsaki. On model checking for visibly pushdown automata. In *Language and Automata Theory and Applications*, pages 408–419. Springer, 2012.

[38] Nguyen Van Tang. A tighter bound for the determinization of visibly pushdown automata. *arXiv preprint arXiv:0911.3275*, 2009.

[39] Burchard von Braunmühl and Rutger Verbeek. Input-driven languages are recognized in log n space. In *Foundations of Computation Theory*, pages 40–51. Springer, 1983.