Minimization, Learning, and Conformance Testing of Boolean Programs

Viraj Kumar *, P. Madhusudan, and Mahesh Viswanathan **

University of Illinois at Urbana-Champaign, Urbana, IL, USA,
{kumar, madhu, vmahesh}@cs.uiuc.edu

Abstract. Boolean programs with recursion are convenient abstractions of sequential imperative programs, and can be represented as recursive state machines (RSMs) or pushdown automata. Motivated by the special structure of RSMs, we define a notion of modular visibly pushdown automata (modular VPA) and show that for the class of languages accepted by such automata, unique minimal modular VPA exist. This yields an efficient *approximate* minimization theorem that minimizes RSMs to within a factor of k of the minimal RSM, where k is the maximum number of parameters in any module. Using the congruence defined for minimization, we show an active learning algorithm (with a minimally adequate teacher) for context free languages in terms of modular VPAs. We also present an algorithm that constructs complete test suites for Boolean program specifications. Finally, we apply our results on learning and test generation to perform model checking of black-box Boolean programs.

1 Introduction

The abstraction-based approach to model-checking is based on building finite models, say using predicates over variables, and subjecting the finite models to systematic state-space exploration [10]. Recursion of control in programs leads to models with recursion, which can be captured using pushdown automata. The model of recursive state machines (RSMs) [1] is an alternate model, which is equivalent in power but whose notation is closer to programs.

The class of visibly pushdown languages is a subclass of context-free languages, defined as those languages that can be accepted by pushdown automata whose action on the stack is determined by the letter the automaton reads. Given that a model of a program is naturally visibly pushdown (since we can make calls and returns to modules visible), visibly pushdown languages are a tighter model for Boolean programs. The class of visibly pushdown languages enjoys closure and decidability properties, making several problems like model-checking pushdown program models against visibly pushdown specifications decidable [3, 5].

In this paper we reap more benefits from the visibly pushdown modeling of programs, by showing that pushdown program models can be *minimized*, can be

* Supported by DARPA/AFOSR MURI award F49620-02-1-0325
** Supported by NSF CCF 04-29639 and NSF CCF 04-48178

learnt and *tested for conformance*, and subject to *black-box checking*, paralleling results for finite-state models. We now outline these results.

In a recent paper [4], we showed that visibly pushdown languages have a *congruence* based characterization. However, this congruence does not yield minimal visibly pushdown automata, and in fact, unique minimal visibly pushdown automata do not exist in general. The main reason why the minimization result fails is that when implementing functions in the automata model there are two choices available. One option is to have function modules that "compute" the value for multiple (or all the) parameters, and then let the caller decide which result to pick when the function returns. The second option is for the function to only "compute" the answer to the specific parameter with which it was called.

In [4], we showed a minimization result for a special class of models. We looked at visibly pushdown machines with a modular structure (similar to recursive state machines) which have the additional property that modules, when called, compute the answers to all parameters and let the caller decide the right answer on return. This results in *modular*, *single-entry* (i.e., the state the machine enters on function calls is the same, no matter what the parameter is) machines. We showed that for any visibly pushdown language there is a unique minimal modular single-entry machine.

The restriction to single-entry machines is awkward. First they do not correspond to program models, as programs typically do not compute answers to all parameters on function calls. Second, combining the computation for multiple parameters can result in requiring a lot more memory, which in the context of automata corresponds to larger number of states.

The first contribution of this paper is a minimization result of a variant of modular VPAs that has multiple entry points in each module, corresponding to the multiple parameters. This variant is inspired by the recursive state machine model in two ways: (a) the parameters passed to modules are explicit and visible, and (b) we demand that when a module is called, the state *but not the parameter* is pushed onto the stack. Requiring that the parameter not be pushed onto the stack is crucial in achieving a unique minimization result; since the program does not "remember" the parameter it called the module with, it cannot choose the result for a parameter from a combined result. Thus, we get minimal program models that are more faithful to the semantics of programming languages. Technically, if we allow automata models that are not complete (i.e., certain transitions being disabled from certain states) then it is possible to encode the parameter in the calling state. Thus our minimization result only applies to complete models. However, we also show that any incomplete recursive state machine model for a program can be translated into a *canonical*, complete, recursive state machine model which is at most k times larger than the incomplete model, where k is the maximum number of parameters in any module. This results in an approximate minimization procedure for incomplete RSMs that transforms a deterministic RSM in polynomial time into one whose size is at most k times the size of the minimal deterministic RSM.

Next, we look at the problem of learning modular VPA models for context free languages. The learning model that we consider is one where the learning algorithm is allowed to interact with a knowledgeable teacher who answers two types of queries: *membership* queries, where the learner can ask whether a string belongs to the target language, and *equivalence* queries, where the learner can ask whether a hypothesis machine does indeed recognize exactly the target language. Learning algorithms identifying machine models for formal languages in such a learning framework have recently been extensively used in formal verification in a variety of contexts (see [8, 2, 12, 7, 21, 14, 25] for some examples). However, all these applications use algorithms that learn finite state models based on the algorithm originally proposed by Angluin [9]. The reason for this is because known learning algorithms apply only to very limited push-down models: Chomsky Normal Form grammars with known non-terminals [9] (which corresponds to knowing all the states of a pushdown model and discovering only the transitions), and deterministic one-counter machines [11].

Our main result in the context of the learning problem is that we can learn the smallest complete, deterministic, modular VPA for a language in time which is polynomial in the length of the longest counter-example provided by the teacher, and the size of the smallest machine model. The algorithm is based on the congruence based characterization of the minimum machine that we present in this paper[1].

We would like to contrast this learning algorithm with the implicit one suggested by the results of [5, 22]. The results in [5] show that associated with every visibly pushdown language is the tree language of *stack trees* which is regular. Using Sakakibara's algorithm [22] one could learn the deterministic bottom-up tree automaton accepting the language of stack trees, and convert that to obtain a visibly pushdown automaton for the language using the results of [5]. There are two downsides to using this approach. First, the resulting VPA is non-deterministic, and one would need to pay the exponential cost in obtaining a deterministic machine. Second, even the non-deterministic VPA obtained thus has an awkward structure, as it may not be modular, or have one entry for each parameter, that we expect of program models.

The number of membership and equivalence queries made by our learning algorithm has the same dependence on the size of the minimal machine and length of the longest counter-example as Angluin's algorithm for learning finite state machines. However, in the case of regular languages, it is possible for a *cooperative teacher* to present counter-examples that are linear in the size of the smallest deterministic finite automaton accepting the language. For modular VPAs this is not the case; one can construct examples where the shortest counter-example is exponential in the size of the smallest modular VPA recognizing the language. However, we observe that the counter-examples (even if long) are

[1] The learning, conformance testing, and black-box checking algorithms in this paper can also be adapted to the congruence presented in [4] to construct single-entry VPA models. We present results using the congruence presented in this paper as we believe that multiple entry modular VPAs are a more natural and succinct model.

highly structured, and can be succinctly represented using an equation system. Our learning algorithm can be shown to have the same running time even when the teacher presents such succinct counter-examples, thus yielding a polynomial learning algorithm for such cooperative teachers.

We can also PAC learn modular-VPAs with membership queries. The PAC learning with membership queries model [24, 9] is a weaker learning framework, where the equivalence queries are replaced by an oracle that samples strings (based on any fixed probability distribution) and labels them as either belonging to the language or not; the learning algorithm is required to identify the concept "approximately" in polynomial time, using the sampling oracle, with "high probability". We can show that one can PAC learn modular VPAs provided one has an oracle that samples strings represented succinctly using the equational representation. Because of lack of space we do not outline the PAC learning algorithm, but the extension to this framework is standard based on our results on learning with a knowledgeable teacher.

Next, we study the problem of *conformance testing* modular VPAs. In this framework, one is given a black-box implementation, whose internal transition structure is assumed to be unknown. The specification is another machine, but one whose transition structure is fully known. The objective in conformance testing is to construct a sequence of test inputs (based on the specification) such that if the implementation does not "conform" to the specification, then the implementation gives a different output than the specification on the test. Typically the notion of "conformance" is taken to be language equivalence, though weaker notions such as ioco have also been explored [23]. Since Moore's seminal work [20], there have been many algorithms to generate such test sequences; major results are summarized in [16, 13, 19, 18] [2]. These algorithms construct complete suites (i.e., guaranteed to catch all buggy implementations) when both the specification and the implementation are known to be finite state machines. Further, these algorithms also assume that an a priori bound on the number of states of the implementation is known. We extend these results on conformance testing to the case when the specification and implementation are modeled as complete modular VPAs. The size of our test suite and the running time to construct the test suite depend on the number of states in the unknown black-box implementation, and the construction of the test suite relies on our characterization of the minimal modular VPA recognizing a language.

Finally, we show how we can apply our results to verify third-party programs. *Black Box Checking* [21] is a framework to model check unknown systems, by first learning the model of the system and then model checking the constructed model. This framework has been applied to construct finite state models, using Angluin's learning algorithm and conformance testing algorithms for finite models. Our extension to learning and testing boolean programs, allows one to extend this framework to verify recursive systems.

[2] The references here only talk about algorithms to construct complete test suites, which is the focus of this paper. There is also extensive work on constructing incomplete test suites that catch all bugs in the limit.

An alternative formulation of visibly pushdown automata is *nested word automata* [6], which are *finite* automata (without stack) on words endowed with a nesting relation (corresponding to the nesting relation defined between calls and their matching returns). A nested word automaton can decide the state at a return based on the previous state and the state before its matching call. This model already has the implicit restriction that at a call the module and parameter cannot be "pushed", and hence conforms to the restriction we introduce in this paper. Consequently, all results in this paper also hold for appropriately defined *modular* nested word automata.

The paper is organized as follows. We first introduce the model of modular VPAs and RSMs, along with useful definitions and notation. In Section 3 we present our results on the existence of unique, minimal, complete modular VPAs, and show how these results can be used to construct approximately minimal RSM models. After this we focus our attention exclusively on complete machines. Our learning algorithm is presented in Section 4, while our conformance testing results are presented in Section 5. Finally, we conclude in Section 6 by showing how these results can be combined to perform black-box checking.

2 Preliminaries

In this section, we define modular VPAs, and introduce notation that we will use in the rest of the paper.

We model Boolean programs as modular VPAs by modeling each module as a finite-state machine that also allows calls to and returns from other modules: modules representing different procedures are modeled separately, the usage of stack is implicit in that when a call to a module occurs, the local state of the module is pushed into the stack automatically, but neither the name of the called module nor the parameter passed is stored in the stack.

Let us fix M, a finite set of *modules*, with $m_0 \in M$ as the *initial module*. For each $m \in M$, let us fix a nonempty finite set of *parameters* P_m, with $P_{m_0} = \{p_0\}$.

A *call* c is a pair (m, p) where $m \in M \setminus \{m_0\}$ and $p \in P_m$, and denotes the action calling the module m with parameter p (we won't allow the initial module to be called except at the beginning, and hence (m_0, p_0) will not be a call). Let Σ_{call} denote the set of all calls. Let us also fix a finite set of internal actions Σ_{int}, and let $\Sigma_{\text{ret}} = \{r\}$ be the alphabet of returns, containing the unique symbol r. Let $\widehat{\Sigma} = (\Sigma_{\text{call}}, \Sigma_{\text{int}}, \Sigma_{\text{ret}})$ and let $\Sigma = \Sigma_{\text{call}} \cup \Sigma_{\text{int}} \cup \Sigma_{\text{ret}}$.

Definition 1 (Modular VPAs). *A* modular VPA *over* $\langle M, \{P_m\}_{m \in M}, m_0, \widehat{\Sigma} \rangle$ *is a tuple* $(\{Q_m, \{q_m^p\}_{p \in P_m}, \delta_m\}_{m \in M}, F)$ *where for each* $m \in M$

- Q_m *is a finite set of states. We assume that for* $m \neq m'$, $Q_m \cap Q_{m'} = \emptyset$. *Let* $Q = \bigcup_{m \in M} Q_m$ *denote the set of all states.*
- *For each parameter* $p \in P_m$, q_m^p *is a state associated with* p; *we will call this the* entry *associated with the call* (m, p).
  (Note that we do not insist that q_m^p *be different from* $q_m^{p'}$, *when* $p \neq p'$.*)*
- $F \subseteq Q_0$ *is the set of final states.*

– $\delta_m = \langle \delta^m_{\text{call}}, \delta^m_{\text{int}}, \delta^m_{\text{ret}} \rangle$ *is a triple of transition relations, one for calls, one for internals and one for returns, where*
  - $\delta^m_{\text{call}} \subseteq \{(q,(n,p),q^p_n) \mid q \in Q_m, (n,p) \in \Sigma_{\text{call}}\}$;
  - $\delta^m_{\text{int}} \subseteq \{(q,a,q') \mid q,q' \in Q_m, a \in \Sigma_{\text{int}}\}$;
  - $\delta^m_{\text{ret}} \subseteq \{(q,q',q'') \mid q',q'' \in Q_m, q \in Q\}$;

Notation: We write $q \xrightarrow{(n,p)} q^p_n$ to mean $(q,(n,p),q^p_n) \in \delta^m_{\text{call}}$, $q \xrightarrow{a} q'$ to mean $(q,a,q') \in \delta^m_{\text{int}}$, and $q \xrightarrow{q'} q''$ to mean $(q,q',q'') \in \delta^m_{\text{ret}}$.

Semantics: A *stack* is a finite sequence over Q; let the set of all stacks be $St = Q^*$. A *configuration* is any pair (q,σ) where $q \in Q$, and $\sigma \in St$. Let *Conf* denote the set of all configurations, along with the special configuration c_0.

The configuration graph of a modular VPA is (V,E) where $V = Conf$ and E is the smallest set of Σ-labeled edges that satisfies:

(Initial edge) $c_0 \xrightarrow{(m_0,p_0)} (q^{p_0}_{m_0}, \epsilon) \in E$.

(Internal edges) If $(q,\sigma) \in V$ $(q \in Q_m)$ and $(q,a,q') \in \delta^m_{\text{int}}$, then $(q,\sigma) \xrightarrow{a} (q',\sigma) \in E$.

(Call edges) If $(q,\sigma) \in V$ and $q \xrightarrow{(m,p)} q^p_m$, then $(q,\sigma) \xrightarrow{(m,p)} (q^p_m, \sigma q) \in E$.

(Return edges) If $(q,\sigma q') \in V$ $(q' \in Q_m)$, and $(q,q',q'') \in \delta^m_{\text{ret}}$, then $(q,\sigma q') \xrightarrow{r} (q'',\sigma) \in E$.
(Note that q'' and q' belong to the same module m.)

A *run* of A on a word u is a path in the configuration graph on u. Let $\rho : Conf \times \Sigma^* \to 2^{Conf}$ be the function where $\rho(conf,u)$ is the set of configurations reached at the end of all runs from *conf* on u in the configuration graph. An *accepting run* of A on u is a run from the initial configuration c_0 that ends in a configuration whose state is in the final set F. A word u is *accepted* by A if there is an accepting run of A on u, i.e. if $\rho(c_0,u) \cap (F \times St) \neq \emptyset$. The *language* of A, $L(A)$, is defined as the set of words $u \in \Sigma^*$ accepted by A.

Let WM be the set of well-matched words over $\widehat{\Sigma}$, i.e, the set of all words generated by the grammar: $S \to cSrS$ (for each $c \in \Sigma_{\text{call}}$), $S \to aS$ (for each $a \in \Sigma_{\text{int}}$), and $S \to \epsilon$. We will denote by $w, w', w_i, \ldots$ words in WM. Note that a modular VPA accepts only words that are in $\{(m_0,p_0)\}.WM$ (since the final states are in module m_0, and the initial symbol (m_0,p_0) is not considered a call).

A word u *reaches* state q in A if $(q,\sigma) \in \rho(c_0,u)$ for some $\sigma \in St$. Note that if q belongs to module m, then $u = u_1(m,p)w$ for some $p \in P_m$ and $w \in WM$. We say that $(m,p)w$ is an *access string* for state q in A.

A (complete) modular VPA is said to be *deterministic* if its transition relation is deterministic, i.e. for each $m \in M$:

– $\forall q \in Q_m, a \in \Sigma_{\text{int}}$, there is at most one q' such that $(q,a,q') \in \delta^m_{\text{int}}$; and
– $\forall q \in Q, q' \in Q_m$, there is exactly one q'' such that $(q,q',q'') \in \delta^m_{\text{ret}}$.

Note that transitions on calls are always deterministic since the target state is always the unique entry state associated with the call.

A modular VPA is said to be *complete* if a transition of every label is enabled from every state, i.e. for each $m \in M$,

- for each $q \in Q_m$ and $(n,p) \in \Sigma_{\text{call}}$, $(q,(n,p),q_n^p) \in \delta_{\text{call}}^m$;
- for each $q \in Q_m$ and $a \in \Sigma_{\text{int}}$, $\exists q'$ such that $(q,a,q') \in \delta_{\text{int}}^m$; and
- for each $q \in Q$ and $q' \in Q_m$, $\exists q''$ such that $(q,q',q'') \in \delta_{\text{ret}}^m$.

A *recursive state machine* (RSM) is a modular VPA with no final states set and where every word that has a run on it can be completed to a well-matched word. More precisely, the language defined by an RSM R is the set of words u such that there is a path in the configuration graph from the initial configuration, and we require that for every $u \in L(R)$, there is some word $w \in (\{(m_0,p_0)\}.WM) \cap L(R)$, such that u is a prefix of w.

Let MR be the set of all words with "matched-returns", i.e. where every return has a matching call, i.e. $MR = \{u \in \Sigma^* \mid \exists v \in \Sigma^*, uv \in WM\}$. It is easy to see then that the language of an RSM consists of words in $(m_0,p_0).MR$.

The *size* of a modular VPA (or RSM) is the number of states in it; when we refer to minimization, we mean minimizing the number of states.

The definition of modular VPAs above has been chosen carefully with final states only in the initial module, and disallowing calls to the initial module. Note that if we did allow final states in non-initial modules, then complete VPAs are less powerful than incomplete ones. For example, if $u(m,p)$ is accepted by a complete VPA, then $u'(m,p)$ is also accepted by it. An incomplete VPA can disallow the call after u' and hence reject $u'(m,p)$. However, incomplete VPAs are too ill behaved in the sense that we can encode parameters into the state being pushed at a call in an incomplete VPA, leading minimization results to fail. The focus on complete VPAs is a subtle and tricky restriction that allows our minimization result to go through.

Section 4 and Section 5 will consider only complete modular VPAs, and show the learning and conformance testing results for them. In the latter half of Section 3, we show how to handle (incomplete) RSMs by using the results for complete machines.

3 Minimization of VPAs and RSMs

Minimization of complete modular VPAs: In this section, we will show that for any *complete* modular VPA A, there exists a unique minimal (with respect to number of states) deterministic modular VPA that accepts the same language as A does. As a corollary, it will follow that deterministic complete modular VPAs are as powerful as non-deterministic complete ones.

Lemma 1. *For any complete modular VPA A, there exists a unique minimal complete modular VPA A' such that $L(A') = L(A)$. Further, given a complete deterministic modular VPA A, the unique minimal deterministic modular VPA equivalent to it can be constructed in polynomial time.*

Proof. (sketch)
Let $A = (\{Q_m, \{q_m^p\}_{p \in P_m}, \delta_m\}_{m \in M}, F)$ and let $L(A) = L$. For every $m \in M$, we define an equivalence relation $\sim_m$ on $P_m \times WM$ which depends on L (and not on A) as: $(p_1, w_1) \sim_m (p_2, w_2)$ **iff** $\forall u, v \in \Sigma^*$

$$u(m, p_1)w_1v \in L \text{ iff } u(m, p_2)w_2v \in L$$

Note that $\sim_m$ is a congruence in the sense that if $(p_1, w_1) \sim_m (p_2, w_2)$, then for any well-matched word w, $(p_1, w_1w) \sim_m (p_2, w_2w)$.

Let $[(p, w)]_m$ denote the equivalence class of (p, w) with respect to $\sim_m$. It can be shown that $\sim_m$ has at most $2^{|Q_m|}$ equivalence classes. These equivalence classes correspond to states of the unique minimal complete deterministic modular VPA. The details of the construction and complexity, and the proof of minimality can be found in [17]. □

Let A be a complete modular VPA. For distinct states q_1, q_2 in module m of A with access strings $(m, p_1)w_1$ and $(m, p_2)w_2$ respectively, a pair of strings (u, v) is a *distinguishing test* for $\{q_1, q_2\}$ if exactly one of $u(m, p_1)w_1v$ and $u(m, p_2)w_2v$ is in $L(A)$. By the above theorem, for a *minimal* complete modular VPA A, there is a set D of distinguishing tests such that for every module m and distinct states q_1, q_2 in module m of A, there is a distinguishing test $(u, v) \in D$ for $\{q_1, q_2\}$. We call such a set D a *complete* set of distinguishing tests.

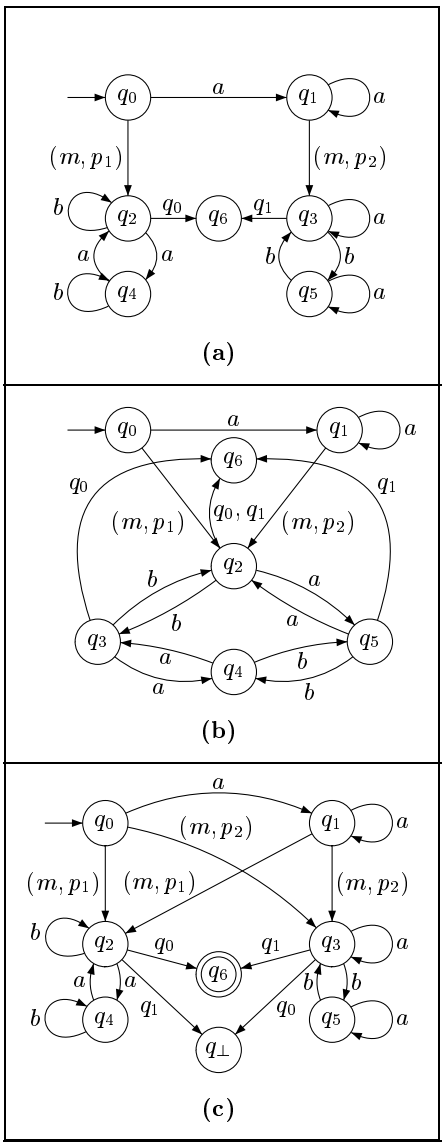

Fig. 1. (a) and (b): Two non-isomorphic minimum-state RSMs; (c) completing the RSM

Minimization of recursive state machines: Figure 1 shows two non-isomorphic (incomplete) RSMs that use minimal number of states and accept the same language. The call with parameter p_1 checks if there are an even number of a's (from the call to the return) while the parameter p_2 checks if there are an even number of b's. The first machine processes the parameters separately, while the second machine processes both parameters and lets the caller choose

the appropriate result. However, if we restrict to complete machines, then we can complete the first machine by enabling all calls from q_0 and q_1, to get a modular VPA that accepts the language $L' = \{w \in WM \mid \forall v \preceq w, v \in L\}$ [3], where L is the language accepted by the RSM (see Fig. 1(c); all edges are not drawn). However, the second automaton cannot be transformed this way: if we enable the call (m, p_2) from q_0, then upon returning from a call, we would not know whether the module was called with p_1 or p_2, and hence cannot accept the right language.

Our strategy for minimizing RSMs is to translate an RSM into a complete modular VPA, minimize it, and translate it back to an RSM. This results in an RSM whose size is at most a factor k of the minimal size possible, where k is bound by the maximum number of parameters in any module of the RSM.

Lemma 2. *Let $R = (\{Q_m, \{q_m^p\}_{p \in P_m}, \delta_m\}_{m \in M})$ be an RSM and let k be the maximum number of parameters for any module. Then there exists a complete modular VPA A such that $L(A) = \{w \in WM \mid \forall v \preceq w, v \in L(R)\}$. Further, the size of A is at most k times R, and A is deterministic if R is deterministic.*

Lemma 3. *Let R be an RSM, and let $\hat{A}$ be the complete minimal deterministic automaton such that $L(\hat{A}) = \{w \in WM \mid \forall v \preceq w, v \in L(R)\}$. Then there exists a deterministic RSM R' with at most the number of states in $\hat{A}$, such that $L(R') = L(R)$.*

Using the lemmas above and Lemma 1, we can show:

Theorem 1. *Given a deterministic RSM R, we can compute in polynomial time an RSM $\hat{R}$ that accepts the same language, such that if R' is any RSM accepting $L(R)$, then $\hat{R}$ has at most k times the number of states R' has.*

Proof. Given R, we complete it (using Lemma 2), minimize it (using Lemma 1), and using Lemma 3, build an incomplete RSM $\hat{R}$ (all this takes polynomial time). If R' is another RSM accepting the same language as R does, then its completion results in the same language as the completion of R, and is at most k times size of R'. Since $\hat{R}$ was obtained using incompletion of a minimal machine (and the incompletion process only removes states), the result follows.

4 Learning complete modular VPAs

We will now consider the problem of exactly learning a target context free language L (over $\langle M, \{P_m\}_{m \in M}, m_0, \widehat{\Sigma}\rangle$) by constructing a complete, modular VPA for L from examples of strings in L and those not in L. In our learning model, we will assume that the learning algorithm is interacting with a knowledgeable teacher (often called a *minimally adequate teacher*) who assists the learner in identifying L. We can think of the teacher as an oracle answering two types of queries

[3] $\preceq$ denotes the prefix relation on words.

Membership Query The learning algorithm may select any string x and ask whether x is a member of L.

Equivalence Query In such a query, the algorithm submits a hypothesis RSM $\widehat{A}$. If $L = L(\widehat{A})$ then the teacher informs the learning algorithm that it has correctly identified the target language. Otherwise, in response to the query, the learner receives a *counter-example* word $(m_0, p_0)x$ where x is a well-matched string and $(m_0, p_0)x \in (L \setminus L(\widehat{A})) \cup (L(\widehat{A}) \setminus L)$. No assumptions are made about how the counter-example is chosen. In particular the counter-example x maybe picked adversarially.

Our goal is to design an algorithm that identifies L in time which is polynomial in the size of the smallest modular VPA recognizing L and the length of the longest counter-example presented to it. The algorithm that we present is very similar to the learning algorithm for regular languages due to Angluin [9]. However our presentation is closer in spirit to the algorithm due to Kearns and Vazirani [15].

4.1 Overview of algorithm

Let A be the smallest, complete, deterministic, modular VPA that recognizes the target language L and let $size(A)$ be the number of states of A. Recall from Lemma 1, that the states of A correspond to the equivalence classes of $\sim_m$. The main idea behind the learning algorithm will be to progressively identify the equivalence classes of $\sim_m$; the construction of the VPA A from $\sim_m$ will be the same as that outlined in Lemma 1 (see [17]).

The learning algorithm will proceed in phases. During the execution, the algorithm will maintain equivalence relations (not necessarily congruences) $\equiv_m$ on $P_m \times WM$ such that if $(p_1, w_1) \sim_m (p_2, w_2)$ then $(p_1, w_1) \equiv_m (p_2, w_2)$. In other words, $\sim_m$ will always be a refinement of $\equiv_m$. The algorithm will also ensure that if it knows $(m_0, p_0)w_1 \in L$ and $(m_0, p_0)w_2 \notin L$, then $(p_0, w_1) \not\equiv_{m_0} (p_0, w_2)$. Further the equivalence $\equiv_m$ itself will be maintained implicitly using a data structure called a *classification forest*, such that deciding if $(p_1, w_1) \equiv_m (p_2, w_2)$ is efficient; this is formally stated next. A classification forest is very similar to a classification tree, introduced by Vazirani and Kearns. Readers unfamiliar with the Vazirani-Kearns data structure are referred to [17].

Proposition 1. *Given (p_1, w_1) and (p_2, w_2), $(p_1, w_1) \equiv_m (p_2, w_2)$ can be decided using $O(size(A))$ membership queries.*

In addition to maintaining the equivalence relation $\equiv_m$, the algorithm will maintain a representative (p, w) for each equivalence class $[(p, w)]_m$ of $\equiv_m$. In what follows we will denote the representative of $[(p, w)]_m$ by $\text{rep}([(p, w)]_m)$. In particular, the algorithm will ensure that (p_0, ϵ) is always among the representatives. In each phase of the algorithm, these representatives will be used to construct a hypothesis machine $\widehat{A}$. A module m will have one state corresponding to each representative $\text{rep}([(p, w)]_m)$. The transitions are naturally determined by the relation $\equiv_m$ as follows. On a call symbol (m, p), every state has a transition to

the state $\text{rep}([(p,\epsilon)]_m)$. On an internal symbol a, a state $(p,w) = \text{rep}([(p,w)]_m)$ has a transition to the state $\text{rep}([(p,wa)]_m)$. Finally on a return with $(p_1,w_1) = \text{rep}([(p_1,w_1)]_{m_1})$ on top of the stack, the state $(p_2,w_2) = \text{rep}([(p_2,w_2)]_{m_2})$ has a transition to the state $\text{rep}([(p_1,w_1(m_2,p_2)w_2r)]_{m_1})$. Observe that since $\equiv_m$ is not necessarily a congruence, the machine $\widehat{A}$ depends on the representatives chosen. Finally, by using special data structures, this machine can be constructed efficiently from $\equiv_m$ and the representatives (details are in [17]).

In each phase, the algorithm will construct the hypothesis machine $\widehat{A}$ based on the current $\equiv_m$ and representatives. It will then ask an equivalence query with the machine $\widehat{A}$. If the query has a positive answer, the learning algorithm will stop and one can show that in this case $\equiv_m = \sim_m$ and that $\widehat{A}$ is exactly the machine A. On the other hand the equivalence oracle may present a counter example string w. The algorithm will process this counter example to refine $\equiv_m$ to discover a new equivalence class of $\sim_m$. The details of how the counter-example is processed is similar to Angluin's algorithm and is skipped in the interests of space; the interested reader is referred to [17].

The overall algorithm is thus as follows. The algorithm starts with a hypothesis machine, where each module has exactly one state; thus $(p_1,w_1) \equiv_m (p_2,w_2)$ for any p_1,p_2,w_1,w_2. In each phase the algorithm asks an equivalence query with the current hypothesis, and uses the answer to refine the equivalence $\equiv_m$ by identifying one more equivalence class of $\sim_m$. This process repeats until the algorithm has identified all the equivalence classes of $\sim_m$. This algorithm can be implemented efficiently and this is the main theorem of this section.

Theorem 2. *Let L be a language accepted by a complete, deterministic VPA and let A be the smallest modular VPA accepting L. The learning algorithm identifies A by making at most $size(A)$ calls to the equivalence oracle, and $O(size(A)(size(A)+n))$ calls to the membership oracle, where n is the length of the longest counter-example returned by the equivalence oracle.*

4.2 Cooperative Teacher

The running time of the learning algorithm presented in the previous section has the same dependence on the size of the minimal machine and the length of counter-examples, as the learning algorithm for regular languages. However, there is one important difference. For regular languages, a *cooperative* teacher can always find a counter-example of length at most $size(A)$ in response to an equivalence query, yielding a polynomial running time in the presence of cooperative teachers. This is, however not the case for VPAs as the shortest counter-example in response to an equivalence query maybe as long as $2^{size(A)}$. Thus, even if the counter-examples are guaranteed to be the shortest possible, the learning algorithm for VPAs will not run in polynomial time.

There is, however, one form of cooperative teacher who can assist in learning the target VPA fast. Observe that even though the shortest counter-example maybe exponentially long, it is typically highly structured and has a very small, succinct representation. Consider an equation system $\{x_i = t_i\}_{i\geq 1}^{k}$, where x_i is a

variable and t_i is a well-matched string over $\Sigma \cup \{x_1, \ldots x_{i-1}\}$. The variable x_k in such an equation system represents a string over WM that can be obtained by progressively solving for x_i for increasing values of i, by replacing solutions for $x_1, \ldots x_{i-1}$. It can be shown that there is an equation system of size at most $size(A)$ that represents a counter-example to any equivalence query. Further, given a counter-example represented by an equation system (instead of explicitly), we can process the counter example using linearly many (in the size of the equation system) membership queries to discover a new state in the hypothesis machine. The details are a straightforward extension of the ideas already presented, and are skipped in the interests of space.

5 Conformance testing

We now describe the setting for conformance testing. We are given a *specification machine* $\mathcal{S}$ and a "black-box" *implementation machine* $\mathcal{I}$ that are both deterministic complete modular VPAs over $\langle M, \{P_m\}_{m \in M}, \Sigma_{\text{int}}, \Sigma_{\text{ret}} \rangle$. The task is to test whether or not $\mathcal{I}$ is equivalent to $\mathcal{S}$, i.e. $L(\mathcal{I}) = L(\mathcal{S})$. In order to achieve this, we make the following assumptions:

1. $\mathcal{S}$ is minimized and has n states;
2. $\mathcal{I}$ is equivalent to a deterministic complete modular VPA that has at most N states;
3. $\mathcal{I}$ does not change during the testing experiment.

Note that assumption 1 can be made with no loss of generality, since the specification $\mathcal{S}$ is known, and hence we can assume it is minimized. Assumption 2 is necessary in order to guarantee that every state of the implementation is explored. The need for assumption 3 is obvious.

A *sample* over Σ is a pair (T^+, T^-), where T^+, T^- are finite subsets of Σ^*. A modular VPA A is *consistent* with sample (T^+, T^-) if $T^+ \subseteq L(A)$ and $T^- \subseteq \overline{L(A)}$.

Definition 2. *A* conformance test *for $(\mathcal{S}, \mathcal{I})$ is a sample (T^+, T^-) over Σ such that $\mathcal{S}$ is consistent with (T^+, T^-) and, for any $\mathcal{I}$ satisfying the above assumptions, $\mathcal{I}$ is consistent with (T^+, T^-) if and only if $L(\mathcal{I}) = L(\mathcal{S})$.*

Let $Q_{\mathcal{S}}$ (the states of $\mathcal{S}$) be $\{q_1, q_2, \ldots, q_n\}$, with access strings $(m_1, p_1)w_1$, $(m_2, p_2)w_2$, ..., $(m_n, p_n)w_n$ respectively, and let the set of final states of $\mathcal{S}$ be $F_{\mathcal{S}}$. Assume without loss of generality that the access string for every entry state q_m^p of $\mathcal{S}$ is (m, p), and that $q_1 = q_{m_0}^{p_0}$. Let $Q_{\mathcal{I}}$ (the set of states of $\mathcal{I}$) be $\{\hat{q}_1, \hat{q}_2, \ldots, \hat{q}_N\}$, let $\hat{q}_1 = \hat{q}_{m_0}^{p_0}$, and let the set of final states of $\mathcal{I}$ be $F_{\mathcal{I}}$.

Since $\mathcal{S}$ is minimized and has n states, for $\mathcal{I}$ to be equivalent to $\mathcal{S}$ it is necessary for $\mathcal{I}$ to have at least n distinct states. Using the fact that $\mathcal{S}$, being minimized, has a complete set of distinguishing tests, we construct a sample (T_0^+, T_0^-) such that any modular VPA consistent with it has at least n states. Let D be a complete set of distinguishing tests for $\mathcal{S}$. Hence, for every distinct pair of states q_i, q_j in module m, there is a distinguishing test

$(u_{ij}, v_{ij}) \in D$ for $\{q_i, q_j\}$. For every $i = 1, \ldots, n$, let $D_i = \bigcup_j \{(u_{ij}, v_{ij})\}$. Define $T_0 = \bigcup_{i=1}^{n} \{u(m_i, p_i)w_i v \mid (u, v) \in D_i\}$. Let $T_0^+ = T_0 \cap L(\mathcal{S})$ and $T_0^- = T_0 \setminus L(\mathcal{S})$. The following lemma is easy to prove.

Lemma 4. *If $\mathcal{I}$ is consistent with (T_0^+, T_0^-), then*

1. *for every $i \neq j$, $(m_i, p_i)w_i$ and $(m_j, p_j)w_j$ are access strings for* distinct *states of $\mathcal{I}$ (hence $N \geq n$)*
2. *there are access strings $\{x_i\}_{i=1}^{N}$ for all states of $\mathcal{I}$, where $x_i = (m_i, p_i)w_i$ for $i = 1, \ldots, n$ and for $i > n$, x_i is of one of the following forms: $x_i = ya$, where $a \in \Sigma_{\text{int}}$; or $x_i = yzr$, where $y, z \in \{x_1, x_2, \ldots, x_{i-1}\}$ and $z \neq x_1$.*

Note that every access string x_i of $\mathcal{I}$ is of the form $(m, p)w$ for some $m \in M, p \in P_m, w \in WM$. Assume without loss of generality that for each i, x_i is an access string for $\hat{q}_i$. If $\mathcal{I}$ is equivalent to $\mathcal{S}$, it is necessary that for each i, x_i is an access string of a final state of $\mathcal{I}$ if and only if x_i is an access string of a final state in $\mathcal{S}$. We define a sample (T_1^+, T_1^-) such that $\mathcal{I}$ is consistent with this sample if this condition holds.

Define $h : Q_{\mathcal{I}} \to Q_{\mathcal{S}}$ as follows: $h(\hat{q}_i) = q_j$ iff x_i is an access string for q_j in $\mathcal{S}$. Define $T_1 = \{x_i \mid i = 1, \ldots, N\}$. Let $T_1^+ = T_1 \cap L(\mathcal{S})$ and $T_1^- = T_1 \setminus L(\mathcal{S})$. We immediately have the following lemma:

Lemma 5. *If $\mathcal{I}$ is consistent with (T_1^+, T_1^-), then for every $1 \leq i \leq n$, $\hat{q}_i \in F_{\mathcal{I}}$ iff $h(\hat{q}_i) \in F_{\mathcal{S}}$.*

Our goal is to design a sample (T^+, T^-) such that if $\mathcal{I}$ is consistent with it, then $L(\mathcal{I}) = L(\mathcal{S})$. In view of Lemma 5, it is enough to construct a sample such that if $\mathcal{I}$ is consistent with it, then for every $u \in MR$, $h(\hat{q}_i) \xrightarrow{u}_{\mathcal{S}} h(\hat{q}_j)$ whenever $\hat{q}_i \xrightarrow{u}_{\mathcal{I}} \hat{q}_j$. Define

$$T_2 = \bigcup_{i=1}^{n} \{ux_i av \mid a \in \Sigma_{\text{int}}, (u, v) \in D_j \text{ where } h(\hat{q}_i) \xrightarrow{a}_{\mathcal{S}} q_j\}$$
$$T_3 = \bigcup_{i,j=1}^{n} \{ux_j x_i rv \mid (u, v) \in D_k \text{ where } h(\hat{q}_i) \xrightarrow{h(\hat{q}_j)}_{\mathcal{S}} q_k\}$$

It is not hard to see that if $\mathcal{I}$ is consistent with $(T_2 \cap L(\mathcal{S}), T_2 \setminus L(\mathcal{S}))$, then for every $a \in \Sigma_{\text{int}}$, $h(\hat{q}_i) \xrightarrow{a}_{\mathcal{S}} h(\hat{q}_j)$ whenever $\hat{q}_i \xrightarrow{a}_{\mathcal{I}} \hat{q}_j$. Similarly, it can be show that if $\mathcal{I}$ is consistent with $(T_3 \cap L(\mathcal{S}), T_3 \setminus L(\mathcal{S}))$, then $h(\hat{q}_i) \xrightarrow{h(\hat{q}_j)}_{\mathcal{S}} h(\hat{q}_k)$ whenever $\hat{q}_i \xrightarrow{\hat{q}_j}_{\mathcal{I}} \hat{q}_k$. Finally, since we had assumed that the access string for each entry state q_m^p of $\mathcal{S}$ was (m, p) and $x_j = (m, p)$ for some $1 \leq j \leq n$, it follows that $h(\hat{q}_m^p) = q_m^p$. Hence, $h(\hat{q}_i) \xrightarrow{(m,p)}_{\mathcal{S}} h(\hat{q}_m^p)$ whenever $\hat{q}_i \xrightarrow{(m,p)}_{\mathcal{I}} \hat{q}_m^p$. The following theorem now follows.

Theorem 3. *Let $T = T_0 \cup T_1 \cup T_2 \cup T_3$. If $\mathcal{I}$ is consistent with $(T \cap L(\mathcal{S}), T \setminus L(\mathcal{S}))$, then $L(\mathcal{I}) = L(\mathcal{S})$*

Proof. By the above observations, for any string $u \in MR$, it follows by induction on the length of u that $h(\hat{q}_i) \xrightarrow{u}_{\mathcal{S}} h(\hat{q}_j)$ whenever $\hat{q}_i \xrightarrow{u}_{\mathcal{I}} \hat{q}_j$. Now Lemma 5 implies that $L(\mathcal{I}) = L(\mathcal{S})$. □

By the above Theorem, a conformance test (T^+, T^-) for $(\mathcal{S}, \mathcal{I})$ can be constructed given a complete set of distinguishing tests D for $\mathcal{S}$, and a set of access strings for all states of $\mathcal{I}$. We show how these requirements can be met.

Constructing a complete set of distinguishing tests

Lemma 6. *If $\mathcal{S}$ is a minimized deterministic complete modular VPA, a complete set of distinguishing tests D can be constructed effectively.*

The proof of the above lemma is presented in the full version [17]. Let $\Omega = \Sigma \cup \{x_i\}_{i=1}^{n}$. The following lemma is a simple corollary to Lemma 6.

Lemma 7. *A complete set of distinguishing tests D for $\mathcal{S}$ can be represented as $\binom{n}{2}$ strings in Ω^*, each of length $O(n^2)$, where n is the number of states of $\mathcal{S}$.*

Constructing access strings

Let Ω be as defined above, and let $\Omega' = \Omega \cup \{x_{n+1}, \ldots, x_N\}$. By Lemma 4, if $\mathcal{I}$ is consistent with (T_0^+, T_0^-), there is a system of $N-n$ equations, each representable by $O(1)$ symbols in Ω', describing the set of access strings for all states in $\mathcal{I}$. There are at most $\left(N|\Sigma| + N^2\right)^{N-n}$ such systems of equations, at least one of which describes a correct set of access strings for $\mathcal{I}$. Assuming $|\Sigma|$ is a constant, a set of access strings for $\mathcal{I}$ can be represented in $O(n \log n + N^{2(N-n)} \log N)$ space.

6 Black Box Checking

Our learning algorithm, along with our algorithm to generate conformance tests, can be used in a powerful way to model check black-box programs whose structure is unknown. Black-box checking was introduced in [21], and in this framework one assumes that while the structure of the system is unknown, it can be simulated to see if it exhibits certain behaviors. The main idea is to use a machine learning algorithm to construct a model of the program and then use the constructed machine model for verification. Our learning algorithm requires a teacher to answer both membership and equivalence queries. So in order to use our learning algorithm to construct a model of the program, we will need to find a way to answer these queries. Membership queries correspond to whether a certain sequence of steps is executed by the system; thus they can be answered by simulating the system. Equivalence queries are handled by constructing a conformance test. We assume that an *a priori* upper bound on the size of the model of the program is known. When the learning algorithm builds a hypothesis machine, we construct a conformance test using the hypothesis as the specification and the program as the implementation. If the program behaves the same way as the constructed hypothesis, then we have constructed a faithful model of the program. On the other hand, if the program differs from the hypothesis, then the conformance test gives us the counter-example needed for the learning algorithm to refine its hypothesis. Thus, using the learning and testing algorithms presented here, we can perform black-box checking of recursive programs.

References

1. R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis. Analysis of recursive state machines. *TOPLAS*, 27(4):786–818, 2005.
2. R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL*, pages 98–109. ACM Press, 2005.
3. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS*, LNCS 2988, pages 467–481. Springer, 2004.
4. R. Alur, V. Kumar, P. Madhusudan, and M. Viswanathan. Congruences for visibly pushdown languages. In *ICALP*, LNCS 3580, pages 1102–1114. Springer, 2005.
5. R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pages 202–211. ACM Press, 2004.
6. R. Alur and P. Madhusudan. Adding nesting structure to words. In *DLT*, LNCS 4036, pages 1–13, 2006.
7. R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *CAV*, LNCS 3576, pages 548–562. Springer, 2005.
8. G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, 2002.
9. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.
10. T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, LNCS 1885, pages 113–130. Springer, 2000.
11. P. Berman and R. Roos. Learning one–counter languages in polynomial time. In *FOCS*, pages 61–67. IEEE, 1987.
12. J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, LNCS 2619, pages 331–346, 2003.
13. A. Friedman and P. Menon. *Fault Detection in Digital Circuits*. PrenticeHall, Inc., Englewood Cliffs, New Jersey, 1971.
14. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *TACAS*, LNCS 2280, pages 357–370. Springer-Verlag, 2002.
15. M. Kearns and U. Vazirani. *An introduction to computational learning theory*. MIT Press, 1994.
16. Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, New York, 1978.
17. V. Kumar, P. Madhusudan, and M. Viswanathan. Minimization, learning, and conformance testing of Boolean programs. Technical Report UIUCDCS-R-2006-2736, University of Illinois at Urbana-Champaign, June 2006.
18. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.
19. R. Linn and M. Üyar. *Conformance testing methodologies and architectures for OSI protocols*. IEEE Computer Society Press, 1995.
20. E. F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies*, pages 129–153, Princeton University Press, Princeton, NJ, 1956.
21. D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. *J. Autom. Lang. Comb.*, 7(2):225–246, 2001.
22. Y. Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Inf. Comput.*, 97(1):23–60, 1992.
23. J. Tretmans. A formal approach to conformance testing. In *Protocol Test Systems*, volume C-19 of *IFIP Trans.*, pages 257–276. North-Holland, 1994.
24. L. Valiant. A theory of the learnable. *Comm. of the ACM*, 27(11):1134–1142, 1984.
25. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Actively learning to verify safety for FIFO automata. In *FSTTCS*, LNCS 3328, pages 494–505, 2004.