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. Recursive state machines (RSM) serve as machine models for Boolean programs and are semantically equivalent to pushdown automata. While pushdown automata cannot be minimized, motivated by the special structure of RSMs, we define a notion of modular VPA and show that for the class of languages accepted by such automata, unique minimal modular VPA exist. Using this we obtain *approximate* minimization theorems for RSMs, where we show we can construct RSMs that are at most k times the minimal RSM, where k is the maximum number of parameters in a module. Our characterization of the minimum RSM leads to 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 for 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 [1, 2]. Recursion of control in programs leads to models with recursion, which can be captured using pushdown automata. The model of recursive state machines [3] is an alternate model, which is equivalent in power but whose notation is closer to programs.

The class of visibly pushdown languages is defined as a subclass of context-free languages, as the 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. Moreover, 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 [4, 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

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 [6], 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 minimally visibly pushdown automata do not exist in general. The main reason why the minimization result fails is because 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 [6], we showed a minimization result for a special class of models. We looked at visibly pushown 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 natural program models, as programs typically do not compute answers to all parameters on function calls. Second, combining the computation for multiple parameters can often result 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 models for programs 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.

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 [7–14] for some examples). However, all these applications use algorithms that learn finite state models based on the algorithm originally proposed by Angluin [15]. 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 [15] (which corresponds to knowing all the states of a pushdown model and discovering only the transitions), and deterministic one-counter machines [16, 17].

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. We would like to contrast this learning algorithm with the implicit one suggested by the results of [5, 18]. The results in [5] show that associated with every visibly pushdown language is the tree language of *stack trees* which is regular. Thus, using Sakakibara's algorithm [18] 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 down-sides 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 there is one important difference. 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 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. Moreover, this result can be used to obtain a PAC learning algorithm for modular-VPAs. The PAC learning model [19] is a weaker learning framework, where the knowledgeable teacher is 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. 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 [20]. Since Moore's seminal work [21], there have been many algorithms to generate such test sequences; major results are summarized in [22–25] [1]. 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 a prior 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.

While our results on learning and conformance testing are of independent interest since they extend previous results for finite state machines to pushdown models, they can be combined in a useful manner to model check black-box programs. *Black Box Checking* [11] is a framework to model check unknown systems, by first learning the model of the system and then model checking the constructed model. Whenever the learning algorithm asks a membership query for a string (i.e., whether a certain behavior is exhibited by the system), we simulate the black-box system on the behavior. When the learning algorithm wants to ask an equivalence query, the black-box checking algorithm constructs a conformance test based on the hypothesis, and checks whether the system "conforms" to the hypothesis. If it does, then we have model of the system which can be model checked. Otherwise, the conformance test is the witness to the inequivalence of the model and the system. This framework of black-box checking has been applied to construct finite state models, based on Angluin's algorithm

[1] 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. But it is impossible to survey this huge body of work.

and conformance testing for finite state machines. Our results on learning and conformance testing can be combined in the same manner to now perform black box checking of systems by constructing pushdown models.

The rest of 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. Then (in Section 4) we show how our minimization results can be used to construct approximately minimal RSM models for incomplete models. After this we focus our attention exclusively on complete machines. Our learning algorithm is presented in Section 5, while our conformance testing results are presented in Section 6. Finally we conclude (Section 7) by showing how these results can be combined to perform black-box checking.

2 Preliminaries

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

We will 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 seperately, 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* $p \neq p'$*.)*
- $\delta_m = \langle \delta_{\text{call}}^m, \delta_{\text{int}}^m, \delta_{\text{ret}}^m \rangle$ *is a triple of transition relations, one for calls, one for internals and one for returns, where*
  - $\delta_{\text{call}}^m \subseteq \{(q, (n, p), q_n^p) \mid q \in Q_m, (n, p) \in \Sigma_{\text{call}}\}$;
  - $\delta_{\text{int}}^m \subseteq \{(q, a, q') \mid q, q' \in Q_m, a \in \Sigma_{\text{int}}\}$;

- $\delta^m_{\text{ret}} \subseteq \{(q, q', q'') \mid q', q'' \in Q_m, q \in Q\}$;

– $F \subseteq Q_0$ *is the set of final states.*

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 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 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.

Sections 3, 5, and 6 will consider only complete modular VPAs, and show the minimization, learning and conformance testing results for them. In Section 4 we show how to handle (incomplete) RSMs by using the results for complete machines. In particular, we show how RSMs can be minimized up to a factor based on the number of parameters in each module.

3 Minimization of complete modular VPAs

In this section, we will show that for any *complete* modular VPA A, there exists a unique minimal 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.

Theorem 1. *If A is a complete modular VPA, then there exists a unique minimal complete modular VPA A' such that $L(A') = L(A)$.*

Proof. 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:

$$(p_1,w_1) \sim_m (p_2,w_2) \;\; \textbf{iff} \;\; \forall u,v \in \Sigma^*.\; u(m,p_1)w_1v \in L \text{ iff } u(m,p_2)w_2v \in L$$

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, $(p_1,w_1w) \sim_m (p_2,w_2w)$.

Now, $\sim_m$ has only finitely many equivalence classes. We show in fact that it has at most $2^{|Q_m|}$ equivalence classes (note that although $\sim_m$ is based only on L, we bound the number of classes using A). For $(p, w) \in P_m \times WM$, let $R_{p,m}$ be the set of states reached by A starting in q_m^p and running on w. Then it is easy to observe that if $R_{p_1,w_1} = R_{p_2,w_2}$, then $(p_1, w_1) \sim_m (p_2, w_2)$, which gives the bound on the number of classes.

Define the modular VPA $\hat{A} = (\{\hat{Q}_m, \{\hat{q}_m^p\}_{p \in P_m}, \hat{\delta}_m\}_{m \in M}, \hat{q}_0, \hat{F})$ as follows: for each $m \in M$, $\hat{Q}_m = \{[(p, w)]_m \mid p \in P_m, w \in WM\}$, and for each $p \in P_m$, $\hat{q}_m^p = [(p, \epsilon)]_m$, and the transition relation $\hat{\delta}_m = \langle \hat{\delta}_{\text{call}}^m, \hat{\delta}_{\text{int}}^m, \hat{\delta}_{\text{ret}}^m \rangle$ is defined as:

1. **call:** $\hat{\delta}_{\text{call}}^m([(p, w)]_m, (m', p')) = [(p', \epsilon)]_{m'}$ for all $(m', p') \in \Sigma_{\text{call}}$. In other words $\hat{q}_m^p = [p, \epsilon]_m$.
2. **internal:** $\hat{\delta}_{\text{int}}^m([(p, w)]_m, a) = [(p, wa)]_m$, for all $a \in \Sigma_{\text{int}}$
3. **return:** $\hat{\delta}_{\text{ret}}^m([(p', w')]_{m'}, [(p, w)]_m) = [(p, w(m', p')w'r)]_m$.

The final states are $\hat{F} = \{[(p_0, w)]_{m_0} \mid (m_0, p_0)w \in L\}$

The internal transitions are well-defined since $\sim_m$ relations are congruences with respect to well-matched words. Similarly, it can easily be shown that the return transitions are also well defined. It is easy to establish the invariant that on any input $u = (m_0, p_0)w_0(m_1, p_1)w_1(m_2, p_2)w_2 \ldots (m_n, p_n)w_n$, $\hat{A}$ reaches the unique configuration

$$\big([(p_n, w_n)]_{m_n}, \quad [(p_0, w_0)]_{m_0}[(p_1, w_1)]_{m_1}[(p_2, w_2)]_{m_2} \ldots [(p_{n-1}, w_{n-1})]_{m_{n-1}}\big)$$

It then follows that $\hat{A}$ accepts L.

The above shows that every complete modular VPA can be determinized. Now, let A be a deterministic complete modular VPA accepting L. We will show that there is a homomorphism from the modules of A to those of $\hat{A}$, which would prove $\hat{A}$ is the unique minimal deterministic complete modular VPA accepting A.

For each $m \in M$, we define the equivalence relation $\approx_m$ on $(P_m \times WM)$ as follows:

$$(p_1, w_1) \approx_m (p_2, w_2) \ \textbf{iff} \ \rho((q_m^{p_1}, \epsilon), w_1) = \rho((q_m^{p_2}, \epsilon), w_2)$$

Since the configuration reached after reading a well-matched word will have its stack empty, $\approx_m$ clearly has finite index for every $m \in M$.

Next, we show that for any $m \in M$, $\approx_m$ refines $\sim_m$. Let $(p_1, w_1) \approx_m (p_2, w_2)$. Let $u \in \Sigma^*$. Since $\rho((q_m^{p_1}, \epsilon), w_1) = \rho((q_m^{p_2}, \epsilon), w_2)$, it follows that $\rho(c_0, u(m, p_1)w_1) = \rho(c_0, u(m, p_2)w_2)$ (where c_0 is the special initial configuration). Hence for any v, $u(m, p)w_1v \in L$ iff $u(m, p)w_2v \in L$, and hence $(p, w_1) \sim_m (p, w_2)$.

Thus, for each $m \in M$, there is a well-defined function h_m such that for all $p \in P_m$ and $w \in WM$, $h_m([(p, w)]_{\approx_m}) = [(p, w)]_{\sim_m}$. Let h be the union of all $h_m, m \in M$. It is easy to verify that h is a homomorphism from A to $\hat{A}$, and hence $\hat{A}$ is minimal. $\square$

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.

4 Minimization of recursive state machines

While the notion of complete modular VPAs leads to unique minimal machines, languages accepted by RSMs do not have unique minimal models (see Figure 1 in Appendix A).

However, using minimization of complete modular VPAs, we can minimize recursive state machines up to a factor k of the minimal size possible, where k is bound by the maximum number of parameters in any module of the RSM. First, let us show that recursive state machines can be faithfully modeled as *complete* modular VPAs.

Lemma 1. *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 $L(A) = \{w \in WM \mid \forall v \preceq w, v \in L(R)\}$.* [2] *Further, the size of A is at most k times R, and A is deterministic if R is deterministic.*

Lemma 2. *Let R be an RSM, and let $\hat{A}$ be the complete minimal deterministic automaton such that $L(A) = \{w \in WM \mid \forall v \preceq w, v \in L(R)\}$. Then there exists an RSM R' with the at most the number of states in A, such that $L(R') = L(R)$.*

Proofs of the above lemmas are deferred to Appendix A.

Using the results of the previous section and the lemmas above, we can show:

Theorem 2. *Given an 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. Take R and complete it, minimize it, and then incomplete it to get $\hat{R}$. 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.

5 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

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

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 who answers two types of queries from the learner

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* well-matched string $x \in (L \setminus L(\widehat{A})) \cup (L(\widehat{A}) \setminus L)$. 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 [15]. However our presentation is closer in spirit to the algorithm due to Kearns and Vazirani [26]. There are two reasons for following the Kearns-Vazirani approach. First, the resulting algorithm is more efficient in terms of running time and space. But more importantly, the Kearns-Vazirani based approach is easier to understand and makes the connections to a congruence based characterization of modular VPAs explicit.

5.1 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 Theorem 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 Theorem 1.

The learning algorithm will proceed in phases. During the execution, the algorithm will maintain equivalence relations (not necessarily a congruence) $\equiv_m$ 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 $(m_0, p_0)w_1 \in L$ 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 the Appendix.

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 $\text{rep}([(p_1, w_1(m_2, p_2)w_2 r)]_{m_1})$. 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; the details are in the Appendix.

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. For prefix $x = y(m, p)w_1$ of w, where (m, p) is the last unmatched call in x, the machine $\widehat{A}$ reaches the state $\rho(c_0, x)$ while the machine A reaches a state corresponding to the equivalence class of (p, w_1) with respect to $\sim_m$. Now since exactly one out A and $\widehat{A}$ accepts w, and strings in L are not equivalent (w.r.t $\equiv_{m_0}$) to strings not in L, we know that $(p_0, w') \not\equiv_{m_0} (p_0, w)$, where $(p_0, w') = \rho(c_0, w)$. Further since (p_0, ϵ) is a state of module m_0, we know that there must be a shortest prefix $x = y(m, p)w_1$ (of w) such that $\rho(c_0, y) \not\equiv_m (p, w_1)$. Let $x = za$. The main observation is that because the machines A and $\widehat{A}$ are deterministic, the string z identifies a new equivalence class of $\sim_m$ that was merged under the relation $\equiv_m$ to class associated with the state $\rho(c_0, z)$. The algorithm therefore adds z to the representatives, refines the equivalence relation $\equiv_m$ to ensure that $\rho(c_0, z)$ and z belong to different classes, and proceeds to the next phase. The way the equivalence relation $\equiv_m$ is refined depends crucially on the way it is represented, and guarantees that $\sim_m$ will continue to be a refinement of $\equiv_m$.

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 3. *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.

5.2 Cooperative Teacher

In the previous section, we showed that the running time of the learning algorithm for modular VPAs depends on the length of the counter-example returned in response to an equivalence query. In fact, the dependency on the length of the counter-example is exactly the same as in the case of learning regular languages. However, there is one important difference.

For the case of regular languages, a *cooperative* teacher can always find a counter-example of length at most $size(A)$ in response to an equivalence query. Thus, the overall running time of the algorithm is guaranteed to be polynomial in $size(A)$ provided the equivalence oracle is helpful. This is, however not the case for VPAs. The shortest counter-example that one may provide in response to an equivalence query maybe as long as $2^{size(A)}$. Thus, even with a cooperative teacher, the algorithm in the previous section will not run in time which is polynomial in $size(A)$.

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 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.

6 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 can 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 $L(\mathcal{I}) = L(\mathcal{S})$.*

In the context of finite-state automata with output, a conformance test is defined as a *single* input sequence x such that $L(\mathcal{I}) = L(\mathcal{S})$ if and only if $\mathcal{I}$ and $\mathcal{S}$ produce identical outputs on input x. Further, it is necessary to assume that the underlying directed graphs of $\mathcal{S}$ and $\mathcal{I}$ are *strongly connected*, in order to ensure that every state can be explored. In the context of modular VPAs, some states may only be reachable together with certain stack configurations. Hence, it is necessary to assume that the underlying (infinite) configuration graph is strongly connected. To simplify matters, we assume that we can "reset" both the specification $\mathcal{S}$ and the implementation $\mathcal{I}$ to their respective initial configurations. This decomposes the single input into a set of tests, which we call a sample.

Let $Q_{\mathcal{S}}$ (the states of $\mathcal{S}$) be $\{q_1, q_2, \ldots, q_n\}$, with access strings $(m_1, p_1)w_1$, $(m_2, p_2)w_2$, $\ldots$, $(m_n, p_n)w_n$ 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 3. *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 4. *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 4, 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

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

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 4. *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 4 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.

6.1 Constructing a complete set of distinguishing tests

Lemma 5. *If $\mathcal{S}$ is a minimized deterministic complete modular VPA with at most n states, a complete set of distinguishing tests D can be constructed effectively.*

The proof of the above lemma is deferred to Appendix C. Let $\Omega = \Sigma \cup \{x_i\}_{i=1}^{n}$. The following lemma is a simple corollary to Lemma 5.

Lemma 6. *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 $\mathcal{S}$.*

6.2 Constructing access strings

Let Ω be as defined above, and let $\Omega' = \Omega \cup \{x_{n+1}, \ldots, x_N\}$. By Lemma 3, 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.

7 Black Box Checking

Our learning algorithm, along with our algorithm to generate conformance tests, can be used in a powerful way to model checking black-box programs whose structure is unknown. Black-box checking was first introduced in [11], 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. One 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. S. Graf, H. Saidi: Construction of abstract state graphs with PVS. In: Proc. 9th International Conference on Computer Aided Verification (CAV'97). Volume 1254., Springer Verlag (1997) 72–83
2. Ball, T., Rajamani, S.: Bebop: A symbolic model checker for boolean programs. In: SPIN 2000 Workshop on Model Checking of Software. LNCS 1885. Springer (2000) 113–130
3. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T.W., Yannakakis, M.: Analysis of recursive state machines. ACM Trans. Program. Lang. Syst. **27** (2005) 786–818

4. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: TACAS. (2004) 467–481
5. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing, New York, NY, USA, ACM Press (2004) 202–211
6. Alur, R., Kumar, V., Madhusudan, P., Viswanathan, M.: Congruences for visibly pushdown languages. In: ICALP. (2005) 1102–1114
7. Ammons, G., Bodik, R., Larus, J.R.: Mining specifications. SIGPLAN Not. **37** (2002) 4–16
8. Alur, R., Cerný, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: POPL. (2005) 98–109
9. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S.: Learning assumptions for compositional verification. In: TACAS. (2003) 331–346
10. Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: CAV. (2005) 548–562
11. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: FORTE XII / PSTV XIX '99: Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX), Deventer, The Netherlands, The Netherlands, Kluwer, B.V. (1999) 225–240
12. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. In: TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, London, UK, Springer-Verlag (2002) 357–370
13. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Actively learning to verify safety for fifo automata. In: 24th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04). Volume 3328 of Lecture Notes in Computer Science., Springer (2004) 494–505
14. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Using language inference to verify omega-regular properties. In: 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05). Volume 3440 of Lecture Notes in Computer Science., Springer (2005) 45–60
15. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75** (1987) 87–106
16. Berman, P., Roos, R.: Learning one–counter languages in polynomial time. In: Foundations of Computer Science. (1987) 61–67
17. Fahmy, A.F., Roos, R.: Efficient learning of real time one-counter automata. In Jantke, K.P., Shinohara, T., Zeugmann, T., eds.: Proceedings of the 6th International Workshop on Algorithmic Learning Theory ALT'95. Volume 997., Berlin, Springer (1995) 25–40
18. Sakakibara, Y.: Efficient learning of context-free grammars from positive structural examples. Inf. Comput. **97** (1992) 23–60
19. Valiant, L.: A theory of the learnable. Communications of the ACM **27** (1984) 1134–1142
20. Tretmans, J.: A formal approach to conformance testing. In: Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI, North-Holland (1994) 257–276
21. Moore, E.F.: Gedanken-experiments on sequential machines. In: Automata Studies, Princeton University Press, Princeton, NJ (1956) 129–153

22. Kohavi, Z.: Switching and Finite Automata Theory. McGraw-Hill, New York (1978)
23. Friedman, A., Menon, P.: Fault Detection in Digital Circuits. PrenticeHall, Inc., Englewood Cliffs, New Jersey (1971)
24. Linn, R., Üyar, M.: Conformance testing methodologies and architectures for OSI protocols. IEEE Computer Society Press (1995)
25. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - A survey. In: Proceedings of the IEEE. Volume 84. (1996) 1090–1126
26. Kearns, M., Vazirani, U.: An introduction to computational learning theory. MIT Press (1994)

A Minimization of recursive state machines

Lemma 1. *Let $M\ R = (\{Q_m, \{q_m^p\}_{p \in P_m}, \delta_m\}_{m \in M})$ 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 $L(A) = \{w \in WM \mid \forall v \preceq w, v \in L(R)\}$.* [3] *Further, the size of A is at most k times R, and A is deterministic if R is deterministic.*

Proof. Build an automaton by making one copy of $q \in Q_m$ for each parameter p: let the state-space hence for each m hence be $Q_m \times P_m$. Now, the entry state for (m, p) will be (q_m^p, p), and hence all entry states are *different*. Intuitively, the state in the called modules remember the parameter that was passed to the module when it was called. Change return edges so as to return to appropriate states (i.e. on popping (q, p), the return edge must go to a state of the form (q', p)). This transformation does not change the language of the RSM.

Now, let us complete the RSM. Introduce a dead-state $dead_m$ for each $m \in M$. whenever there is an internal or return edge missing, throw in an edge that goes to the appropriate dead-state of the module. Note that on calls, we *cannot* go to the dead state, but must go to the entry of the module called. However, since the new automaton will accept only well-matched words, this is alright and we will transition to a reject state when the call returns and hence effectively disable the call. Finally, if the call to (m', p') is disallowed from (q, p), then remove all transitions from states (q', p') in module m' that pop (q, p) and throw in a return transition that pops (q, p) and goes to state $dead_m$. Note that if the call-edge from (q, p) is taken, then at the corresponding return the automaton would reach the dead-state and hence reject all runs on well-matched words that take this call-transition.

It is easy to see that the resulting automaton satisfies the requirements of the lemma.

Lemma 2. *Let R be an RSM, and let $\hat{A}$ be the complete minimal deterministic automaton such that $L(A) = \{w \in WM \mid \forall v \preceq w, v \in L(R)\}$. Then there exists an RSM R' with the at most the number of states in A, such that $L(R') = L(R)$.*

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

Proof. We will remove edges from A in order to construct R'. The removal of edges may make some states unreachable, and these states can of course be removed as well from R'. We will assume that the states in A are of the form $[p, w]_m$, as defined in the minimization construction in the last section.

Construct R' by removing all *useless* transitions. A transition is useless if there is no word in $L(A)$ that is accepted such that the run on the word uses the transition. We will show that $L(R') = L(R)$.

$L(R) \subseteq L(R')$ is clear, as if $u \in L(R)$, then by definition of RSMs, we know that there is a v such that $uv \in WM \cap L(R)$, and hence $uv \in L(A)$. The run of A on uv will make sure that all transitions that constitute the run on u are useful. Hence none of them will be removed and R' will have a run on u.

For the converse, assume the contrary, i.e. there is a word in $L(R')$ that is not in $L(R)$. Choose such a minimal word, that is a word such that all its prefixes are in $L(R)$. We will consider cases depending on the last letter in this word. Let $u(m, p)w \in L(R)$ and $u(m, p)wa \in L(R') \setminus L(R)$,

Internal/Call: $a \in \Sigma_{\text{int}} \cup \Sigma_c$

> A (and R') on the word $u(m, p)w$ reaches a configuration of the form $([p, w]_m, \sigma)$. Since the transition from $[p, w]_m$ on a is present in R', it must be useful. Hence there must be a word $u'(m, p')w'av' \in L(A)$, such that R' on $u'(m, p')w'$ reaches the same state $[p, w]_m$. But then this state must be the same as $[p', w']_m$. So we know $(p, w) \sim_m (p', w')$.
> Since $u'(m, p')w'av' \in L(A)$, it follows that $u'(m, p)wav' \in L(A)$, and hence $u'(m, p)wa \in L(R)$. But if RSM R accepts $u'(m, p)wa$ and accepts $u(m, p)$, then it must accept $u(m, p)wa$ (after the run on $u(m, p)$ in R, we can follow the run for wa as in the run for $u'(m, p)wa$, since wa does not have any unmatched returns and hence doesn't look at the stack. This refutes the assumption.

Return: $a = r$

> Since $a = r$, we know that $(m, p) \neq (m_0, p_0)$. Now, let the state reached in A after u be q. The state A reaches reading $u(m, p)w$ is $[p, w]_m$. Since $u(m, p)wr \in L(R')$, we know that the transition on the final r from the state $[p, w]_m$ popping q is useful. Hence there must be a word $u'(m, p')w'rv' \in L(A)$ such that R' on u' reaches q and on $u'(m, p')w'$ reaches the state $[p, w]_m$, which must be the same state as $[(p', w')_m]$. So $(p, w) \sim_m (p', w')$.
> Since $u'(m, p')w'rv' \in L(A)$, it follows that $u'(m, p)wrv' \in L(A)$, and hence $u'(m, p)wr \in L(R)$. But if RSM R accepts $u'(m, p)wr$ and accepts $u(m, p)$, then it must accept $u(m, p)wr$ (after the run on $u(m, p)$ in R, we can follow the run for w as in the run for $u'(m, p)wa$, and then execute r since the return edge popping q is indeed enabled). This again refutes the assumption.

Figure 1 shows two non-isomorphic RSMs that are minimal and accept the same language.

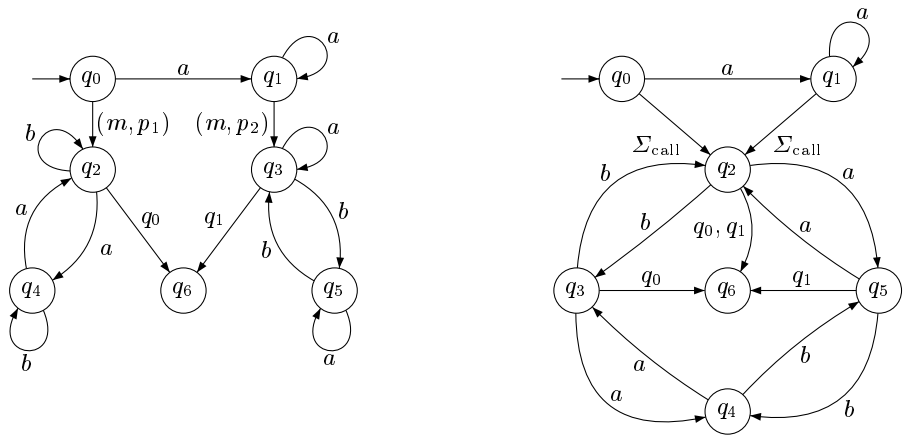

Fig. 1. Two non-isomorphic minimum-state RSMs

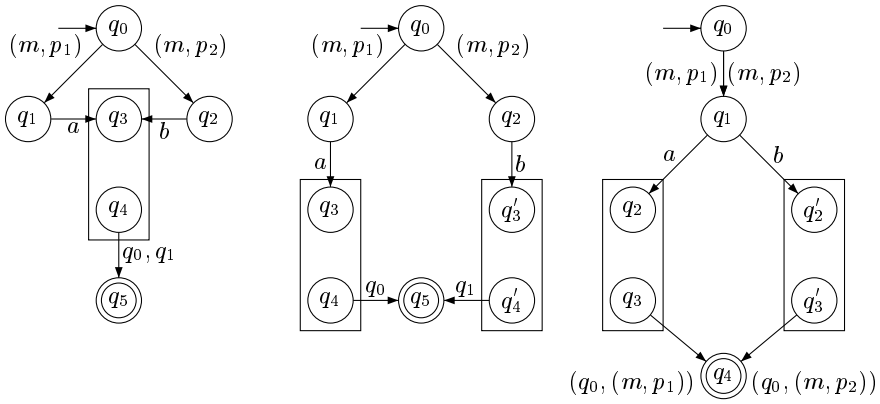

Fig. 2. Two non-isomorphic minimum-state RSMs

Figure 2 shows that there is a language L for which the smallest modular VPA accepting L (shown on the left) is smaller than any k-module SEVPA accepting L.

B The Learning Algorithm for VPAs

We will now describe in detail the learning algorithm for modular VPAs. Recall that the problem we are investigating is that 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 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 who answers two types of queries from the learner

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* well-matched string $x \in (L \setminus L(\widehat{A})) \cup (L(\widehat{A}) \setminus L)$. 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 [15]. However our presentation is closer in spirit to the algorithm due to Kearns and Vazirani [26]. There are two reasons for following the Kearns-Vazirani approach. First, the resulting algorithm is more efficient in terms of running time and space. But more importantly, the Kearns-Vazirani based approach is easier to understand and makes the connections to a congruence based characterization of modular VPAs explicit.

B.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 Theorem 1, that the states of A correspond to the equivalence classes of $\sim_m$. Recall that we said that the learning algorithm proceeds in phases, where in each phase it constructs a hypothesis machine, asks an equivalence query, and based on the counter-example returned in response to the equivalence query it changes the hypothesis machine for the next phase. The hypothesis machine is constructed based on two pieces of information that the learning algorithm maintains. First it maintains an equivalence relation $\equiv_m$ which is always guaranteed to be coarser than $\sim_m$. Second, it maintains representatives for each equivalence class of $\equiv_m$. In the next section, we will outline how $\equiv_m$ and the representatives are dynamically maintained, thereby completely describing the learning algorithm in detail.

B.2 Details of the algorithm

As outlined before, the algorithm maintains an equivalence relation $\equiv_m$ and a set of representatives of each equivalence class in a special data structure called *classification forest*. Let $S = \bigcup_{m \in M} S_m$ be the representatives. So S_m is a collection of pairs (p, w) which are representatives of the current equivalence relation $\equiv_m$. We can also thing of S_m be the states of A that have been discovered

so far. In order to witness the fact that the elements of S_m correspond to different equivalence classes of $\sim_m$, the algorithm also maintains a set of *distinguishing tests* $D_m = \bigcup_{m \in M} D_m$. Thus, D_m consists of pairs (u, v). Each such pair is said to distinguish (p_1, w_1) and (p_2, w_2) if exactly one out of $u(m, p_1)w_1v$ and $u(m, p_2)w_2v$ is in the language L. The representatives S and distinguishing tests D are maintained in a data structure called *classification forest*, which together with the operation of *sifting*, identifies the equivalence relation $\equiv_m$.

Classification Forest. A forest $\mathcal{F} = \{T_m\}_{m \in M}$ is a collection of binary trees T_m, one for each module m. Each internal node in T_m is labeled by a test from D_m and each leaf is labeled by a string S_m. The tree T_m is constructed as follows. The root is labeled by some test $d = (u, v) \in D_m$. We will place in the left subtree all $(p, w) \in S_m$ such that $u(m, p)wv \notin L$ and in the right subtree all $(p, w) \in S_m$ such that $u(m, p)wv \in L$. We recurse this construction at the left and right children until each $(p, w) \in S_m$ is at its own leaf. Observe that a pair $(p_1, w_1), (p_2, w_2) \in S_m$ is distinguished by the test labeling their least common ancestor. Our algorithm will dynamically maintain this forest $\mathcal{F}$ as it proceeds.

Sifting. We will now describe an operation called *sifting* that will play an important role in our algorithm. Consider $(p, w) \in P_m \times WM$. We can "sift" (p, w) down the tree T_m in the classification forest $\mathcal{F}$ by making a series of membership queries as follows. Starting at the root, if we are at an internal node labeled by test (u, v) then we move to left child if $u(m, p)wv \notin L$ and otherwise we move to the right child. We continue recursively in this fashion until we reach a leaf and we return the label (p', w') of reached leaf. Thus, $\mathit{sift}((p, w), T_m)$ returns a string $(p', w') \in S_m$ such that (p, w) agrees with (p', w') on the tests in D_m. The number of membership queries made by $\mathit{sift}((p, w), T_m)$ is equal to the depth of tree T_m which is at most $|S_m|$

Defining the equivalence $\equiv_m$. The classification forest data structure and the sift operation naturally define an equivalence relation $\equiv_m$ as follows.

$$(p_1, w_1) \equiv_m (p_2, w_2) \text{ iff } \mathit{sift}((p_1, w_1), T_m) = \mathit{sift}((p_2, w_2), T_m)$$

The main feature of this equivalence relation is that $\sim_m$ is a refinement of $\equiv_m$. This is formally stated and proved next.

Proposition 2. *Let $\mathcal{F} = \{T_m\}_{m \in M}$ be a classification forest, and let $(p_1, w_1), (p_2, w_2) \in P_m \times WM$. If $(p_1, w_1) \sim_m (p_2, w_2)$ then $\mathit{sift}((p_1, w_1), T_m) = \mathit{sift}((p_2, w_2), T_m)$*

Proof. The proposition follows from the definition of $\sim_m$ in the proof of Theorem 1 that (p_1, w_1) and (p_2, w_2) are equivalent if they behave identically on all distinguishing tests (of that kind that appear in D). Thus, they will follow the same path in T_m and reach the same leaf node.

Thus the goal of the algorithm is to refine $\equiv_m$ until it is the same as $\sim_m$. Our representation allows one to also decide equivalence very efficiently.

Proposition 3. *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.*

Proof. Follows from the observation that sifting $O(|S_m|)$ time. Since S_m has exactly one representative per equivalence class of $\equiv_m$, which is coarser than $\sim_m$, the proposition holds.

Constructing Hypothesis $\widehat{A}$. A classification forest $\mathcal{F} = \{T_m\}_{m \in M}$ naturally defines a modular VPA $\widehat{A}$. Each state of $\widehat{A}$ will correspond to an equivalence class of $\equiv_m$, with the representative of the class from S_m being used to define the transitions. Formally, $\widehat{A} = (\{Q_m, \{q_m^p\}_{p \in P_m}, \delta_m\}_{m \in M}, F)$ where

- For each $(p, w) \in S_m$, we will have state $(p, w) \in Q_m$ of $\widehat{A}$
- $q_m^p = \mathit{sift}((p, \epsilon), T_m)$
- For a state $(p, w) \in Q_m$, $\delta_{\text{call}}^m((p, w), (m', p')) = \mathit{sift}((p', \epsilon), T_{m'})$
- $\delta_{\text{int}}^m((p, w), a) = \mathit{sift}((p, wa), T_m)$, where $a \in \Sigma_{\text{int}}$
- $\delta_{\text{ret}}^m((p, w), (p', w')) = \mathit{sift}((p', w'(m, p)wr), T_{m'})$, where $(p', w') \in Q_{m'}$
- $F = Q_{m_0} \cap L$. We will ensure (ϵ, ϵ) is the distinguishing test labeling the root of T_{m_0}. Thus, F is just all the access strings in the right subtree of T_{m_0}.

Observe that the transition structure of $\widehat{A}$ can be very different from that of A and $\widehat{A}$ might accept a totally different language. Further $\widehat{A}$ can be constructed by making at most $O(|\mathcal{F}||\Sigma|)$ calls to the membership oracle.

Initial Classification Forest. The initial classification forest $\mathcal{F} = \{T_m\}_{m \in M}$ is such that T_m for every m is tree with only one node (which is also the leaf). For $m \neq m_0$, the node is labeled by the string (p, ϵ), where p is some parameter in P_m (the choice of which p does not matter). In T_{m_0} the node is labeled with (p_0, ϵ). The machine $\widehat{A}$ associated with this machine has exactly one state in each module, which serves as the entry point for every parameter. All internal transitions will be self loops, while on returns the machine will go to the unique state of the calling module.

Processing Counter-examples Having defined the initial classification forest and hypothesis, the only thing left for us to outline is how the algorithm uses the counter-examples obtained in response to an equivalence query to discover a new state of A and maintain the classification forest. As mentioned earlier, we will ensure that (except for the initial forest) the root of T_{m_0} will be labeled by test (ϵ, ϵ). This ensures that no state of $\widehat{A}$ at any point during the learning ever corresponds to a subset of states containing both an accepting state and a non-accepting state of A. In addition, S_{m_0} will have (p_0, ϵ) as a string. This means the initial state of A will be known.

Before describing the procedure for processing a counter-example formally, let us look at the intuition behind it. Recall that $\rho^A(c_0, x)$ denotes that state reached by machine A after reading x. Similarly, let us denote by $\rho^{\widehat{A}}(c_0, x)$ the label of the state reached by $\widehat{A}$ after reading x. Consider w a counter-example

to the equivalence of A and $\widehat{A}$. Let us look at the execution of A and $\widehat{A}$ on w. Since exactly one out of $\widehat{A}$ and A accept w and because in the partition induced by a classification forest an accepting state and non-accepting state are never equivalent, it must be the case that $\rho^A(c_0, w)$ and $\rho^{\widehat{A}}(c_0, w)$ belong to different equivalence classes. Further, we know that the initial state of A is a known state, and so the execution of A and $\widehat{A}$ on w begins in the same equivalence class. Thus, it must be the case that there is a shortest prefix y of w such that $\rho^A(c_0, y)$ and $\rho^{\widehat{A}}(c_0, y)$ belong to different subsets in the partition induced by $\mathcal{F}$. Let $y = xa$ for some $a \in \Sigma$. Now since the machines A and $\widehat{A}$ are deterministic, even though $\rho^A(c_0, x)$ and $\rho^{\widehat{A}}(c_0, x)$ are equivalent, it must be the case that $\rho^A(c_0, x)$ and $\rho^{\widehat{A}}(c_0, x)$ are different states of A. Thus we have discovered a new state of A whose access string is y. A distinguishing test for $\rho^A(c_0, x)$ and $\rho^{\widehat{A}}(c_0, x)$ can also be constructed based on the test that distinguishes $\rho^A(c_0, y)$ and $\rho^{\widehat{A}}(c_0, y)$.

To describe the counter-example processing precisely, we also need to consider the boundary cases of when T_m for a module is a single leaf node (as is the case initially) and that introduces some additional cases to consider. For a string $x = x_1 x_2 \cdots x_n$ ($x_i \in \Sigma$) and $1 \leq i \leq j \leq n$, we will denote by $x[i, j]$ the substring $x_i x_{i+1} \cdots x_j$. The recursive procedure *update*() defined in Figure B.2 processes the counter-example w. We assume that it takes as arguments the counter-example w, indices i and j such that $w[i, j] \in WM$ is the substring we need to process, and module m inside which we examine the execution $w[i, j]$. Initially we call $update(w, 1, |w|, m_0)$. A loose analysis shows that a counter-example can be processed by making $O(|w|)$ membership queries.

$update(w, i, j, m)$
  If T_m is tree with a single leaf node then
    Replace the leaf labeled by (p, ϵ) by root labeled $(w[1, i-1], w[j, n])$
    with two children labeled with (p, ϵ) and $w[i, j-1]$
  Else
    let k_1 be the smallest number such that $w[i+1, k_1] \in WM$ and $sift(T_m, w[i, k_1]) \neq \rho^{\widehat{A}}(c_0, w[i, k_1])$
    let $w[i+1, k_2]$ be the longest WM strict prefix of $w[i+1, k_1]$; so $sift(T_m, w[i, k_2]) = \rho^{\widehat{A}}(c_0, w[i, k_2])$
    If $k_2 = k_1 - 1$ (i.e., $w_{k_1} \in \Sigma_{\text{int}}$) then
      Replace the node labeled $\rho^{\widehat{A}}(c_0, w[i, k_2])$ with an internal node and two leaves.
      The leaves are labeled by $\rho^{\widehat{A}}(c_0, w[i, k_2])$ and $w[i, k_2]$.
      The distinguishing test labeling the new internal node is $(u, w_{k_1} v)$ where
      (u, v) is the test that distinguishes $sift(T_m, w[i, k_1])$ from $\rho^{\widehat{A}}(c_0, w[i, k_1])$
    Else
      Let $w[k_2 + 1, k_1]$ be of the form $(m', p)w'r$
      Call $update(w, k_2 + 1, k_1, m')$

Fig. 3. Procedure to update classification forest

Overall Algorithm. The algorithm starts by constructing the initial classification forest $\mathcal{F}$ and the associated hypothesis $\widehat{A}$. It then makes an equivalence query on $\widehat{A}$. The counter-example returned is then used to update the forest, construct a new machine, and make another call to the equivalence oracle on the new machine. This process is repeated until the equivalence query succeeds.

Complexity Analysis. The main loop of the algorithm (that makes calls to the equivalence oracle) will be executed $size(A)$ number of times because in each iteration we discover a new state of A. Further at any point $|\mathcal{F}|$ is at most $O(size(A))$. Thus each iteration of the main loop makes at most $O(size(A) + n)$ calls to the membership oracle, where n is the length of longest counter-example returned by the equivalence oracle. This the total running time is $O(size(A)(size(A) + n))$.

Theorem 5. *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 R 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.*

C Conformance Testing

Lemma 5. *If $\mathcal{S}$ is a minimized deterministic complete modular VPA with at most n states, a complete set of distinguishing tests D can be constructed effectively.*

Proof. Note that (u, v) is a distinguishing test for distinct states q_i, q_j in the same module m of $\mathcal{S}$ iff one of the following conditions holds:

1. $v = \epsilon$, in which case $m = m_0$, $u = \epsilon$ and exactly one of $\{q_i, q_j\}$ is in $F_{\mathcal{S}}$; or
2. $v = av'$ for some $a \in \Sigma_{\text{int}}$, in which case (u, v') is a distinguishing test for $\{q_k, q_l\}$, where $q_i \xrightarrow{a} q_k$ and $q_j \xrightarrow{a} q_l$; or
3. $v = (m', p)wrv'$ for some $(m', p) \in \Sigma_{\text{call}}$ and $w \in WM$, in which case there is a state $q \in Q_{m'}$ $(m' \neq m_0)$ with access string $(m', p)w$ such that $q \xrightarrow{q_i} q_k$ and $q \xrightarrow{q_j} q_l$, and (u, v') is a distinguishing test for $\{q_k, q_l\}$; or
4. $v = rv'$ and $u = u'(m', p)w$, in which case there is a state $q \in Q_{m'}$ $(m' \neq m_0)$ with access string $(m', p)w$ such that $q_i \xrightarrow{q} q_k$ and $q_j \xrightarrow{q} q_l$, and (u', v') is a distinguishing test for $\{q_k, q_l\}$.

We will use the above characterization of distinguishing tests to construct D. For every $m \in M$ and $p \in P_m$, let

$$W_{m,p} = \{q \in Q_m \mid \exists w \in WM \text{ such that } (m, p)w \text{ is an access string for } q\}$$

Define the directed graph $G_{\mathcal{S}} = (V, E)$ where the vertex set $V = \bigcup_{m \in M} Q_m \times Q_m$, and the edge set E is defined as the smallest set such that $(q_i, q_j) \rightarrow (q_k, q_l) \in E$ whenever $q_i \in W_{m,p_i}$, $q_j \in W_{m,p_j}$, $q_k \in W_{m',p_k}$, $q_l \in W_{m',p_l}$ and one of the following conditions holds:

Type 1 edge: $m = m'$ and $q_i \xrightarrow{a} q_k$ and $q_j \xrightarrow{a} q_l$ for some $a \in \Sigma_{\text{int}}$; or
Type 2 edge: $m = m'$ and $\exists q \in W_{m'',p}$ such that $q \xrightarrow{q_i} q_k$ and $q \xrightarrow{q_j} q_l$; or
Type 3 edge: $p_k = p_l = p'$ and $\exists q \in W_{m',p'}$ such that $q_i \xrightarrow{q} q_k$ and $q_j \xrightarrow{q} q_l$.

Note that for every $q_i, q_j \in Q_{m_0}$ such that $q_i \in F_{\mathcal{S}}$ and $q_j \notin F_{\mathcal{S}}$, (ϵ, ϵ) is a distinguishing test for $\{q_i, q_j\}$. Further, if (u, v) is a distinguishing test for $\{q_k, q_l\}$ and $e = (q_i, q_j) \to (q_k, q_l)$ is an edge in E, then

1. (u, av) is a distinguishing test for $\{q_i, q_j\}$, if e is an edge of type 1 above;
2. $(u, (m'', p)w_{p'',q}rv)$ is a distinguishing test for $\{q_i, q_j\}$, if e is an edge of type 2 above;
3. $(u(m', p')w_{p',q}, rv)$ is a distinguishing test for $\{q_i, q_j\}$, if e is an edge of type 3 above.

Hence, a distinguishing test for $\{q_i, q_j\}$ can be found effectively by determining a path from (q_i, q_j) to a vertex (q_k, q_l) in $G_{\mathcal{S}}$ such that exactly one of $\{q_k, q_l\}$ is in $F_{\mathcal{S}}$. Determining such a path for every pair of distinct states within each module yields a complete set of distinguishing tests D.