CONFORMANCE TESTING AND ERROR EXPLANATION
FOR SOFTWARE MODELS

BY

VIRAJ KUMAR

B.S., St. Stephen’s College, Delhi, India, 1998
M.S., Indian Institute of Technology, Powai, India, 2000
M.S., University of Illinois at Urbana-Champaign, 2004

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois
Abstract

The ever-increasing reliance on digital systems has dramatically increased the emphasis on the reliability of the software controlling them. Consequently, techniques known as formal methods have been developed to mathematically verify correctness of software, and to fix bugs that are found. In general, verification is a hard problem, because of the inherent complexity of abstractions that are used to model programs. For many classes of systems, the verification problem is in fact undecidable (there is no algorithm that can always give the correct answer for arbitrary inputs). Despite these negative results, the urgent need for software validation has fueled research aimed at identifying the largest possible classes for which the problem is not only decidable, but tractable.

The first part of this thesis focuses on an important category of software verification problems known as conformance testing. Given a specification of correctness, the problem is to devise a test suite (known as a conformance test) that will pass precisely those program implementations that conform to the specification. We tackle this problem for finite-state models which capture non-recursive programs, and also for pushdown models which capture recursive program behavior.

The conformance test must be constructed without knowledge of the program implementation details. For finite-state models, this essentially reduces the problem to exploring an unknown directed graph. The hardness of this latter problem is a well-studied consequence of so-called combination locks, that inhibit exploration algorithms. Our key observation is that the assumptions leading directly to the hardness of conformance testing can, in many cases, be relaxed sufficiently to make the problem tractable. To show this, we establish an interesting connection between three quantities: the number of faults in the program implementation $P$, the size of certain cuts in the directed graph $G_P$ corresponding to $P$, and the time needed to explore $G_P$ via random walks. This result, which we believe is interesting in a far more general context as well, enables us to extend the applicability of conformance testing to a much wider class of program models.

For pushdown models, several complications arise because the most well-studied kinds of pushdown automata do not share several key decidability properties enjoyed by finite automata. We focus our attention on the recently pro-
posed sub-class of \textit{visibly pushdown automata} (VPA) that are powerful enough to capture recursive program behavior and also possess the requisite decidability properties. In order to extend conformance testing algorithms for finite automata to VPAs, we first present a congruence-based characterization of the class of languages accepted by VPAs (similar to the Myhill-Nerode theorem for regular languages). This result has several applications, and in this thesis we apply it to construct conformance tests for VPA program models.

While conformance testing is concerned with detecting faults in a program model, \textit{error explanation} addresses the question of how to correct faults. By its very nature, error explanation is an informal process that admits many heuristics. As a result, there has been very little effort to characterize the complexity of the error explanation problem. Because this is a vital part of any verification endeavor, we address these unanswered questions in the latter half of this thesis.

We consider two of the most popular heuristics employed in error-explanation algorithms, and analyze their complexity as a function of the program model. We establish an interesting connection between error explanation according to one heuristic, and the problem of determining the smallest deterministic finite automaton that is consistent with a given sample of positively and negatively labeled inputs. By appealing to a well-known hardness result for this latter problem, we are able to show that this approach to error explanation is likely to be intractable. On the other hand, we prove that error explanation based on the second heuristic is tractable for several models that capture program behavior.
To my family, for their continuous love and support.
Acknowledgments

I owe a great deal of thanks to my adviser, Mahesh Viswanathan, for his tremendous patience and support. I would also like to thank my committee members, Michael Loui, P. Madhusudan and Leonard Pitt, who generously offered their time, insight and support on numerous occasions. Thanks also to Craig Zilles and Mehdi Harandi for several opportunities as an instructor. Not only has this experience enhanced my teaching abilities, it has broadened my research interests and has allowed me to explore the intersection between these facets of my academic profession.
# Table of Contents

List of Figures ................................................................. ix
List of Abbreviations ........................................................ x

## Chapter 1 Introduction ....................................................... 1
  1.1 Motivation ................................................................. 1
  1.2 Conformance Testing ................................................... 3
    1.2.1 Related Work ...................................................... 3
    1.2.2 Our Contributions ................................................ 4
  1.3 Error Explanation ..................................................... 7
    1.3.1 Related Work ...................................................... 7
  1.4 Publications ............................................................ 8

## Chapter 2 Preliminaries .................................................... 10
  2.1 Non-recursive Program Models ........................................ 10
    2.1.1 Finite State Machines ........................................... 10
    2.1.2 Extended Finite State Automata ................................ 11
  2.2 Recursive Program Models ............................................ 12
    2.2.1 Pushdown Machines .............................................. 12
    2.2.2 Visibly Pushdown Automata .................................... 13

## Chapter 3 Conformance Testing for Non-recursive Program Models .......... 16
  3.1 Introduction ............................................................ 16
    3.1.1 Our Results ...................................................... 18
    3.1.2 Directed Graphs, Walks and Cuts ................................ 19
  3.2 Random Walks on Digraphs ............................................ 20
  3.3 Conformance Testing of FSMs .......................................... 24
  3.4 Testing One Black Box ................................................ 28
    3.4.1 Algorithm for $r > d \min(n, \Delta)$ .......................... 28
    3.4.2 Algorithm for $r \leq d \min(n, \Delta)$ ........................ 29
  3.5 Lower Bounds .......................................................... 30
  3.6 Checking sequences ................................................... 35
  3.7 Discussion ............................................................. 35

## Chapter 4 Congruences for Visibly Pushdown Languages .................. 37
  4.1 Introduction ............................................................ 37
  4.2 Congruence based characterization of VPLs .......................... 39
    4.2.1 Well-matched visibly pushdown languages ..................... 39
    4.2.2 General visibly pushdown languages ........................... 40
  4.3 $k$-module single-entry visibly pushdown automata .................. 43
    4.3.1 Minimization Algorithm ........................................ 47
  4.4 Conclusions ............................................................ 51
Chapter 5  Conformance Testing for Recursive Program Models  52
  5.1 Introduction .............................................. 52
  5.2 Preliminaries ............................................. 53
  5.3 Minimization of complete modular VPAs .................... 55
  5.4 Conformance testing ..................................... 57
    5.4.1 Constructing a complete set of distinguishing tests .... 60
    5.4.2 Constructing access strings ........................... 61
  5.5 Conclusions .............................................. 61

Chapter 6  Error Explanation ................................. 62
  6.1 Introduction .............................................. 62
  6.2 Preliminaries ............................................. 65
  6.3 Computing the Minimum Edit Set .......................... 66
    6.3.1 NP-completeness ..................................... 66
    6.3.2 Inapproximability Result ............................. 69
  6.4 Computing the Closest-output distance ...................... 72
    6.4.1 Upper bound for FSMs and PDMs ....................... 72
    6.4.2 Hardness results for EFSA representation ............. 74
  6.5 Conclusions .............................................. 75

References ..................................................... 77

Author’s Biography ........................................... 82
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Construction of Extended Cut Paths</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Algorithm for testing a specific Black-Box when $r &gt; d \min(n, \Delta)$</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Algorithm for testing a specific black-box when $r \leq d \min(n, \Delta)$</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>Specification machine $S$ for $r &gt; d$</td>
<td>32</td>
</tr>
<tr>
<td>3.5</td>
<td>Implementation machine $I(i, j, X)$</td>
<td>32</td>
</tr>
<tr>
<td>3.6</td>
<td>Implementation machine $I(j, X)$</td>
<td>33</td>
</tr>
<tr>
<td>3.7</td>
<td>Specification machine $S$ for $r \leq d \min{n, \Delta}$</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Two non-isomorphic minimum-state VPAs</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>Algorithm for computing a minimum-state single-entry VPA</td>
<td>47</td>
</tr>
<tr>
<td>6.1</td>
<td>A graph $G$ and the associated FSM $M$</td>
<td>67</td>
</tr>
<tr>
<td>6.2</td>
<td>A solution FSM for the example graph $G$</td>
<td>68</td>
</tr>
<tr>
<td>6.3</td>
<td>The EFSA for $n = 2$</td>
<td>74</td>
</tr>
</tbody>
</table>
List of Abbreviations

FSM  Finite State Machine.

VPA  Visibly Pushdown Automata.

VPL  Visibly Pushdown Language.

PDM  Pushdown Machine.
Chapter 1

Introduction

1.1 Motivation

The last few decades have seen an ever-increasing reliance on digital systems, a trend that is almost certain to intensify even further in the future. The increased dependence on such systems puts an increased emphasis on the dependability of software: errors can cause serious or even catastrophic failures, leading to losses in time, money and lives [66, 39]. In view of these pressing concerns, a large proportion of the time and money invested in software development is devoted to validation.

A host of techniques collectively known as formal methods have been developed in order to provide mathematically precise correctness guarantees for software. To begin with, a model of the program is created, which can be subjected to mathematical analysis. The choice of model depends on the behavioral sophistication of the code, and can range from simple finite automata models [36] to infinite-state models equipped with powerful constructs such as counters, stacks, clocks, etc. The choice of model is critical: simpler models yield efficient algorithms but may not accurately capture program behavior, whereas complex models can be accurate but invariably lead to inefficient algorithms. Next, a precise formulation of correctness is established in the form of a formal specification. For a complex system, this is usually a non-trivial task, and the specification may be imprecise, incomplete, or even conflicting. In the next phase, a formal method for deciding whether or not the program model satisfies the specification is provided. These methods can be divided into two general categories: proof-based methods and state-exploration methods. In the former approach, the designer attempts to construct a mathematical proof that the model meets its specification. These techniques often require human ingenuity to guide the proof search, and therefore cannot be fully automated. In contrast, state-exploration methods (also known as algorithmic verification or model checking [20]) typically restrict the model to have a finite number of states, and perform a fully automated exploration of the state-space. Should the program model fail to satisfy the specification, the final step is to determine the cause of the error and correct it. Even if an incorrect behavior pattern is discovered, it is invariably a non-trivial task to determine how to fix it. This
is particularly true for complex systems, because input sequences that cause incorrect behavior tend to be extremely lengthy, thereby obfuscating the cause of the error (i.e., the bug).

Proof-based and state-exploration methods rarely scale to models for real-world programs due to the well-known state-space explosion problem. Thus from a practical point of view, this phase is typically replaced by testing. The goal of testing is simply to discover bugs (if any) in the given program by executing it on a carefully chosen test suite. Testing has two primary advantages over the above methods: (i) there is no need to construct a program model since tests can be conducted directly on the program, and (ii) the testing approach yields a practical tradeoff between accuracy and speed: short test suites may not detect all errors but are useful when time is constrained, whereas complete tests can be much longer but can detect all bugs.

All too often test suites are constructed manually, a laborious and expensive task that is prone to errors and bias. In fact it is estimated that improvements in testing infrastructure alone could save one-third of the costs associated with software failures [66]. Our focus, therefore, will be on algorithmic techniques to generate test suites automatically from the correctness specification. In addition, one may also have access to the details of the program model. A host of techniques have been developed that can exploit the additional structural information of the program model to produce shorter tests.

As we have seen above, however, it can be difficult to choose an appropriate model for the program. In addition, there are sometimes good reasons to ignore known details of the implementation. Today, nearly all complex programming projects are subdivided into stages, and these are developed incrementally. Ideally, designers like to generate test suites early in the design process, so that these test suites can be re-used with each new iteration of the code. In such cases, even when details of the code are accessible, it is desirable to ignore those details, to avoid biasing the test suite. Another important case arises when one is interested in validating only a specific aspect of the implementation. In such cases, the overall program behavior may be hard to capture by a simple model but its restriction to behaviors of interest may be much simpler. In large systems, it is usually infeasible to extract code corresponding to this restriction (and, in any case, such a process is itself susceptible to errors). Thus, once again, one needs to assume that the details of the program model are unknown. Yet another important instance occurs in the context of certifying off-the-shelf software components [69]. In this scenario, one is interested in using certain commercially developed software components (for example, device drivers), in order to save time and effort in the development process. Such software components are usually delivered as machine-executable code, whose licenses forbid decompilation back to source code. In order to verify that such components pass muster, it is necessary to model them as “black boxes”, whose internal structure is unknown. In a “black box” program model, one is only permitted to feed
the program inputs and observe its outputs. The task is to devise a test suite, formally called a checking sequence, that will pass precisely those implementations of the program that satisfy the (known) specification. Checking sequences are also known as conformance tests, and the problem is therefore called the conformance testing problem [41, 51, 70]. The computational complexity of the problem is determined by the length of the conformance test. The first part of this thesis will focus on the development of efficient algorithms for constructing conformance tests.

In any software validation endeavor, the final, and perhaps most critical, step is to correct the errors (or bugs) determined by the testing process. The existence of a test input on which the program fails to produce the correct output can theoretically be used to identify errors. Unfortunately, as programs grow in complexity, the sheer length of tests makes it difficult to correctly identify bugs. Intuitively, counter-examples are merely symptoms of error and, as such, do not generally capture the cause of error succinctly. Because of the vital importance of debugging, the problem of error explanation has recently received considerable attention from the research community. Error explanation is an intrinsically informal process that admits many heuristic approaches which cannot be justified formally. Most current approaches to this problem rely on two broad philosophical themes for justification. First, in order to explain something (like an error), one has to identify its “causes” [50]. And second is Occam’s principle, which states that a “simpler” explanation is to be always preferred between two competing theories.

Unfortunately, there has been very little effort to characterize the computational complexity of these methods. However, since error explanation plays a such an important role in the final debugging phase of the verification process, we feel that it is necessary to understand the conditions under which these heuristics yield practical algorithms for error explanation. We address these important unanswered questions in the second half of this thesis.

1.2 Conformance Testing

1.2.1 Related Work

Conformance testing is an important problem, and has been investigated thoroughly over many decades. Most of the results in the area have been obtained under the assumption that both the specification and the (unknown) program model can be represented by finite-state machines [53, 29, 65, 38, 30, 22, 48, 7, 63, 42, 58, 70] (FSMs). The advantage of FSMs is their simplicity, which has led to a precise understanding of the conditions under which conformance testing is tractable for such models. The results indicate that the complexity of conformance testing increases exponentially with the number of extra states that the program model has over the specification model. This is discouraging.
news on two fronts. First, because implementations tend to be more detailed than specifications, it is likely that program models contain many more states than their specification models. Hence, conformance testing quickly becomes infeasible for most programs of interest. Second, the existence of an intractability result for even the simple finite-state model immediately rules out application of this method to more sophisticated models that are typically necessary to capture program behavior, specifically recursive behavior. Recursive program models augment the finite set of states with an unbounded stack that captures function calls (push operations) and returns (pop operations), and include recursive state machines (RSMs) [6], pushdown machines (PDMs) [36] and, more recently, visibly pushdown automata (VPAs) [10].

1.2.2 Our Contributions

In this thesis, we explore the conformance testing problem and its variants in the context of both recursive and non-recursive program models. For non-recursive FSM models, we study a new kind of trade-off between efficient generation of valid test suites and exploration of the state-space. Unlike other automatic test generation techniques, the quality of our test-suites can be tuned by a parameter to the algorithm, and offers provable reliability guarantees. In the context of recursive program models, we extend our results to more powerful visibly pushdown automata (VPA) specification models that capture recursive program behavior. We now detail the ideas behind these two approaches. We believe that our extensions to the applicability of conformance testing are an interesting and important contribution, both from a theoretical and a practical point of view.

Conformance Testing for Non-recursive Program Models

We first describe a natural generalization of the conformance testing problem. Recall that the original problem asks whether or not a black-box program is correct with respect to the given specification. We now relax this problem to the setting where, in addition, we are guaranteed that the implementation is either correct, or else has multiple faults.

At first glance, it might seem unreasonable to consider such a relaxation of the conformance testing problem. Implementations, if they are faulty, usually have a few subtle errors instead of several. Nonetheless, we believe that the relaxed problem is interesting for the following reasons. First, it is a natural mathematical generalization, and one which has been fruitfully studied in the context of property testing* ([31, 4, 3, 2, 14, 32, 18, 26, 57, 25, 5]; see [61] for a survey). Second, as mentioned before, precisely formulating correctness as a specification is usually impossible, and hence it may be unnecessary to

---

*Our approach is inspired by property testing [62, 61], but is in fact different in several important ways.
demonstrate precise adherence to the specification. Third, although we can guarantee to detect faults (with high probability) only when the number of faults is large, we believe our tests will be useful in practice because they have a non-zero probability of detecting every faulty program (including those with very few faults). Thus our tests can be used as a heuristic that applies in all situations. Finally, our approach provides the debugging team with a hierarchy of test suites of increasing precision to choose from, based on practical time constraints imposed on the testing process by product release times.

In this generalized setting, the task is to construct an approximate checking sequence, i.e., an input sequence that will pass those implementations of the program that satisfy the specification, and fail those implementations that have several faults. Note that the behavior of programs with few faults on the approximate checking sequence is left unspecified.

**Our Results**

We study the problem of constructing approximate checking sequences for FSM program models. Recall that the complexity of the original conformance testing problem grows exponentially in the number of extra states that the program model has over the specification model. More precisely, if both the specification and the implementation are modeled as deterministic finite-state machines (FSMs) over input alphabet $\Sigma$, where the minimized FSM for the specification has $n$ states, and the implementation has $n + \Delta$ states, then the shortest checking sequence has length $\Omega(n^3 |\Sigma|^{\Delta+1})$ [68]. The exponential factor, $|\Sigma|^\Delta$, originates from the well-known complexity of exploring an unknown directed graph (due to the existence of so-called combination locks). Unknown directed graphs (i.e., directed graphs with unlabeled vertices and edges) are usually explored via random walks [54]. We demonstrate the existence of an interesting connection between the number of faults in the program model $P$, the size of certain cuts$^1$ in the directed graph associated with $P$, and the time needed to explore this directed graph via a random walk. We prove that the complexity of exploring unknown directed graphs decreases as the size of such cuts increases. More precisely, we have the following key lemma:

**Lemma 1 ([45]).** If a directed graph $G$ has $n + \Delta$ vertices, each with maximum degree $d$, and $U, V$ are sets of vertices such that the size of the smallest $(U, V)$-cut is $r$ ($r < d$), then a random walk from a vertex in $U$ of length $\frac{\Delta}{r} \left( \frac{\Delta^2}{r} \right)^\Delta$ contains a vertex in $V$, with high probability.

In the underlying graph of an FSM, the value of $d$ in the above lemma is $|\Sigma|$. Hence, the exponential factor obtained from Lemma 1 is $\left( \frac{\Delta^2}{r} \right)^\Delta$ (which decreases as $r$ increases), as opposed to the $|\Sigma|^\Delta$ factor for general directed graphs.

$^1$A cut between sets of vertices $U$ and $V$ in a graph is a set of edges whose removal eliminates all paths from vertices in $U$ to vertices in $V$. The size of a cut is the number of edges it has.
We believe that this result will also be of independent interest in the context of model checking. Armed with Lemma 1, we are able to show that existing randomized algorithms for constructing checking sequences can be augmented in a simple manner to construct approximate checking sequences of near-optimal length.

The near-optimality of our result gives us a fairly precise characterization of the conditions under which conformance testing can be performed efficiently in this more relaxed framework. When the program model $P$ is guaranteed to either have several faults or none, we are able to prove that $P$ can have more states than the specification without adversely affecting the complexity of conformance testing. This immediately extends the applicability of conformance testing under the assumption of finite-state models, by lifting the highly restrictive assumption that the model $P$ has no more states than the given specification model.

**Conformance Testing for Recursive Program Models**

Pushdown machines (PDMs) are a convenient abstraction of sequential computation in typical programming languages with nested, potentially recursive, calls to program modules. Consequently, the conformance testing problem formulation can be extended naturally for pushdown models. The class of context-free languages associated with PDMs, however, lacks many of the robust closure and decidability properties of regular languages. This prevents straightforward extensions of existing conformance testing algorithms for finite-state models to the more general context. The class of visibly pushdown languages (VPDs) 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. Pushdown automata with this restriction are called visibly pushdown automata (VPAs). 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 software 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 [8, 10]. Our contribution in this thesis is to show that conformance testing, too, can be extended to VPA models.

**Our Results**

We present two new VPA models (as powerful as regular VPAs) that capture the modular nature of recursive programs: single-entry and multiple-entry modular VPAs $^1$. Unlike regular languages, VPLs do not have unique minimal recognizers

---

$^1$Precise definitions are deferred to Chapter 2
in general. In contrast, we are able to show that every well-matched\textsuperscript{5} VPL can be recognized by a unique minimum-state modular VPA. Furthermore, we present an efficient minimization algorithm for single-entry modular VPAs.

**Theorem 1.** Given a single-entry (respectively, multiple-entry) modular VPA $M$ accepting a well-matched language $L$, there is a unique single-entry (respectively, multiple-entry) modular VPA $\tilde{M}$ that accepts $L$. If $M$ is a single-entry modular VPA of size $n$, $\tilde{M}$ can be computed in time $O(n^3)$.

We then tackle the problem of conformance testing for modular VPAs. The size of our conformance test suite and the running time to construct the test suite depend on the number of states in the unknown black-box implementation $I$, and the construction of the test suite relies on our characterization of the minimal modular VPA recognizing a language.

**Theorem 2** ([9, 44]). If $S$ is a minimized multiple-entry modular VPA specification with $n$ states, and $I$ is a multiple-entry modular VPA implementation with $n + \Delta$ states, a conformance test for $(S, I)$ can be constructed, and can be represented in $O((n^4 + (n + \Delta)^{2\Delta})\log(n + \Delta))$ space.

As in the case for FSMs, the conformance test can be constructed efficiently when the implementation $I$ has no more states than the specification $S$, but requires an exponential amount of time otherwise.

### 1.3 Error Explanation

#### 1.3.1 Related Work

Because of the vital importance of debugging, there has recently been considerable research effort [72, 73, 40, 60, 11, 34, 33] directed towards automating the process of error explanation (or localizing errors or isolating error causes), to assist in the debugging process by identifying possible causes for the faulty behavior. Error explanation tools are now featured in model checkers such as SLAM [13, 11] and Java PathFinder (JPF) [16, 34]. The algorithms developed for these error explanation tools are based on sophisticated use of SAT solvers and model checkers [71, 67, 73, 72, 60, 11, 34, 33].

Two popular heuristics have been widely and successfully used in debugging. The approaches differ primarily in what they choose to be the “causal theory” for errors. The first heuristic relies on the observation that program changes which result in a system that no longer exhibits the offending error trace identify possible causes for the error [71, 67, 15, 73, 72]; in accordance with Occam’s principle, one tries to find the minimum number of changes. The second, more popular approach [60, 11, 34, 33] relies on the intuition that differences between

\textsuperscript{5}A language $L$ is a well-matched VPL if for every string in $L$, each return symbol is preceded by a matching call symbol.
correct and faulty runs of the system shed considerable light on the sources of errors. This approach tries to find correct runs exhibited by the system that closely match the error trace. They then infer the causes of the error from the correct executions and the given error-trace.

Our Results

Explaining errors using the first heuristic discussed above involves computing the smallest edit set, i.e., a set of program changes that eliminates the exhibition of the error. We prove that computing the smallest edit set is intractable, even when the program model is extremely simplistic (i.e., a finite state machine). In particular, we have the following result:

**Theorem 3 ([43]).** Computing the size of the smallest edit set for an FSM is NP-complete.

Furthermore, by reducing the problem to the minimum consistent DFA (deterministic finite automaton) problem [59], we show that the smallest edit set cannot even be approximated efficiently:

**Theorem 4 ([43]).** For every fixed positive integer $k$, there is no polynomial time algorithm to compute an edit set of size opt$^k$ for an FSM $M$, where opt is the size of the smallest edit set for $M$ (unless P = NP).

Thus, our results provide justification for both the relative unpopularity of this approach, and for the necessity of sophisticated SAT solvers that have been employed whenever error-explanation is based on this heuristic. For error-explanation using the second heuristic (namely, finding correct program runs that closely match the error trace), we are able to provide polynomial upper bounds on the running time, even for very general program models such as nondeterministic pushdown automata. For program models that can compactly represent exponentially many states (e.g., using boolean variables), we show that error explanation with this heuristic is once again intractable. However, the intractability in this case stems from the state space explosion problem, and is not intrinsic to the heuristic itself.

1.4 Publications

The research work conducted in the course of this thesis has appeared at the following conferences.

1. A preliminary version of our conformance testing results for non-recursive program models appeared as *Conformance Testing in the Presence of Multiple Faults*, at the ACM-SIAM Symposium on Discrete Algorithms (SODA), 2005 [45].
2. Preliminary versions of our conformance testing results for recursive program models appeared in two papers: (i) *Congruences for Visibly Pushdown Languages*, at the International Colloquium on Automata, Languages and Programming (ICALP), 2005 [9] (this work was conducted in collaboration with R. Alur and P. Madhusudan); and (ii) *Minimization, Learning, and Conformance Testing of Boolean Programs*, at the International Conference on Concurrency Theory (CONCUR), 2006 [44] (this work was conducted in collaboration with P. Madhusudan).

3. A preliminary version of our error explanation results appeared as *On the Complexity of Error Explanation*, at the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), 2005 (this work was conducted in collaboration with N. Kumar).
Chapter 2

Preliminaries

In this chapter, we will specify the formal program models for which the problems of conformance testing and error explanation will be studied. We begin by defining four program models of interest which fall into two categories: recursive program models that consist of a finite set of states together with an unbounded stack, and non-recursive program models that do not possess a stack. Each model is characterized as either an automaton, i.e., a language recognizer that simply accepts or rejects strings, or a (Mealy) machine that produces outputs on each transition. In general, machine models will be used for problems where we demonstrate tractable solutions, and hence our results also apply to automata variants of these models. Conversely, we will demonstrate intractability results for automata models, which will then extend to the more general machine models.

2.1 Non-recursive Program Models

We define two non-recursive program models: FSM (Finite State Machines, i.e., Mealy machines) and EFSA (Extended Finite State Automata). The latter are finite automata equipped with a finite set of boolean variables that can be manipulated on each transition.

2.1.1 Finite State Machines

Definition 1. An FSM is a deterministic finite state Mealy machine $M = (Q, q_0, \delta, \lambda)$ with finite input alphabet $\Sigma$ and finite output alphabet $\Omega$ where $Q$ is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function and $\lambda : Q \times \Sigma \rightarrow \Omega$ is the output function. For $w \in \Sigma^*$, $M(w)$ will denote the string in $\Omega^*$ generated by $M$ on input $w$. We will denote $\delta(q, a) = (q', \alpha)$ by the shorthand $q \xrightarrow{a/\alpha}_M q'$.

For technical reasons, we will sometimes leave the initial state $q_0$ of an FSM unspecified. We will use the notation $M = (Q, \delta, \lambda)$ to denote such an FSM.

The transition function $\delta$ and output function $\lambda$ of an FSM $M$ can be extended to a mapping from $Q \times \Sigma^*$ to $Q$ and to $\Omega^*$ respectively in the usual way.
We say that $M$ is strongly connected if for every pair of states $q, q' \in Q$, $\exists w \in \Sigma^*$ such that $\delta(q, w) = q'$. We say that $M$ is minimal if, for every pair of distinct states $q, q' \in Q$ there is some $w \in \Sigma^*$ such that $\lambda(q, w) \neq \lambda(q', w)$.

**Definition 2.** A separating family for $M$ is a collection of sets $\{Z_q \mid q \in Q\}$ such that for every pair of distinct states $q, q' \in Q$ there is an input string $w$ such that $\lambda(q, w) \neq \lambda(q', w)$, and $w$ is a common prefix of some string in $Z_q$ and some string in $Z_{q'}$.

**Proposition 1 (Yannakakis-Lee [70]).** Every minimal FSM $M$ has a separating family $\{Z_q\}$ such that for each $q \in Q$, $|Z_q| \leq |Q| - 1$ and each $w \in Z_q$ has length at most $|Q| - 1$.

### 2.1.2 Extended Finite State Automata

Let $X$ be a finite set of Boolean variables, and let $\mathcal{A}$ be the set of all possible assignments to variables in $X$, i.e., $\mathcal{A} = \{0, 1\}^X$. Let $B$ be the set of all finite Boolean expressions involving the variables in $X$ and the constants $\top$ (true) and $\bot$ (false).

**Definition 3.** An EFSA is a tuple $M = (Q, q_0, F, \nu_0, g, \delta)$ where:

1. $Q$ is a finite set of states, $F \subseteq Q$ is the set of final (accepting) states;
2. $q_0 \in Q$ is the initial state, $\nu_0 \in \mathcal{A}$ is the initial assignment of variables in $X$;
3. $g : Q \times \Sigma \rightarrow B$ is the guard function that determines when transitions are enabled;
4. $\delta : Q \times \Sigma \rightarrow Q \times \mathcal{A} \times \mathcal{A}$ is the transition function.

Executions of an EFSA are defined as follows. The initial state is $q_0$ and the initial assignment to the variables is $\nu_0$. If the current state is $q$, the current assignment is $\nu$, the input symbol is $a$ and $\delta(q, a) = (q', \nu_T, \nu_F)$, then

- if $g(q, a) = \bot$, no transition is possible;
- if $g(q, a) \neq \bot$, the next state is $q'$ and the next assignment is $\nu'$, where $\nu' = \nu_T$ if the boolean expression $g(q, a)$ is satisfied by the assignment $\nu$, and $\nu' = \nu_F$ otherwise.

We use the shorthand notation $(q, a) \xrightarrow{\nu_T} q'$ to represent such a transition. We use the notation $(q, a) \rightarrow q'$ when the new assignment is identical to the current assignment $\nu$. If $\nu \in \mathcal{A}$, we use the notation $\nu[x_{i} := b]$ to denote the assignment in $\mathcal{A}$ that is identical to $\nu$ except that $x_i$ is set to $b$. We use the notation $L(M) \subseteq \Sigma^*$ to denote the set of strings accepted by $M$. 

11
Remark 1. Note that any FSM $M = (Q, q_0, \delta, \lambda)$ can be modeled as an EFSA $M'$ over the input alphabet $\Sigma \times \Omega$ and the empty set of boolean variables, where the guard function for any pair $(q, (a, \alpha))$ is $\top$ whenever $M$ produces the output $\alpha$ on input $a$ from state $q$, and $\bot$ otherwise.

2.2 Recursive Program Models

We define two recursive program models: PDM (Pushdown Machines, i.e., nondeterministic finite state Mealy machines equipped with a stack) and VPA (Visibly Pushdown Automata). The latter are deterministic pushdown automata where the input alphabet is partitioned into call (or push) symbols on which the automaton must push onto the stack, return (or pop) symbols on which the automaton must pop its stack, and internal symbols on which the automaton must not modify its stack.

2.2.1 Pushdown Machines

We formally define pushdown machines that produce outputs on each transition. We restrict ourselves slightly to nondeterministic pushdown machines which, on every transition, push or pop at most one symbol from the stack, and consume exactly one input symbol. A (nondeterministic) PDM with finite input alphabet $\Sigma$ and finite output alphabet $\Omega$ is a tuple $M = (Q, Q_0, \Gamma, \delta)$ where

1. $Q$ is a finite set of states, $Q_0 \subseteq Q$ is the set of initial states;
2. $\Gamma = \Gamma' \cup \{\bot\}$ is a finite stack alphabet, $\bot$ is the bottom-of-stack symbol;
3. $\delta = \delta_{\text{call}} \cup \delta_{\text{set}} \cup \delta_{\text{int}}$ is the transition relation, where $\delta_{\text{call}} \subseteq (Q \times \Sigma \times Q \times \Gamma' \times \Omega)$ is the set of call (push) transitions, $\delta_{\text{set}} \subseteq (Q \times \Sigma \times \Gamma' \times Q \times \Omega)$ is the set of return (pop) transitions, and $\delta_{\text{int}} \subseteq (Q \times \Sigma \times Q \times \Omega)$ is the set of internal transitions.

A transition $(q, a, q', \gamma, \alpha) \in \delta_{\text{call}}$ is a call-transition where on reading $a$, $\gamma$ is pushed onto the stack, $\alpha$ is output and the state changes from $q$ to $q'$. Similarly, $(q, a, \gamma, q', \alpha) \in \delta_{\text{set}}$ is a return-transition where on reading $a$, if $\gamma \neq \bot$ is the top of the stack, the symbol $\gamma$ is popped from the top of the stack, $\alpha$ is output and the state changes from $q$ to $q'$. If the top of the stack is $\bot$, no return-transition is possible. On internal transitions $(q, a, q', \alpha) \in \delta_{\text{int}}$, the stack does not change and $\alpha$ is output while the state changes from $q$ to $q'$. Note that PDMs need not be deterministic.

A stack is a non-empty finite string over $\Gamma'$ ending in the bottom-of-stack symbol $\bot$. The set of all stacks is denoted as $St = \Gamma'^* \cdot \{\bot\}$. A configuration is a pair $(q, s)$ such that $q$ is a state and $s \in St$. We say that a run $r$ exists from configuration $(q_1, s_1)$ on input $w = a_1 a_2 \ldots a_k \in \Sigma^*$ if there are configurations $(q_1, s_1), (q_2, s_2), \ldots, (q_k, s_k)$ such that, for every $j \in [k]$, $(q_j, s_j) \in Q \times St$ and one of the following transitions $t_j$ exists in $\delta$:  

12
1. \( t_j = (q_j, a_j, q_{j+1}, \gamma_j, \alpha_j) \in \delta_{\text{call}} \) such that \( \gamma_j \in \Gamma' \) and \( s_{j+1} = \gamma s_j \)

2. \( t_j = (q_j, a_j, q_{j+1}, \alpha_j) \in \delta_{\text{ret}} \) such that \( \gamma \in \Gamma' \) and \( s_j = \gamma s_{j+1} \)

3. \( t_j = (q_j, a_j, q_{j+1}, \alpha_j) \in \delta_{\text{int}} \) and \( s_{j+1} = s_j \).

In this case, we say that the sequence of transitions \( T = t_1 t_2 \ldots t_k \) is consistent with the run \( r \).

We say that \( w \in \Sigma^* \) is a valid input from state \( q \in Q \) if there is a run \( r \) from \((q, \bot)\) on input \( w \). We say that \( w \in \Sigma^* \) is a well-matched input from state \( q \in Q \) if there is a run \( r \) from \((q, \bot)\) on input \( w \) such that for some sequence \( \overline{r} \) of transitions consistent with \( r \), the number of push-transitions in \( \overline{r} \) is equal to the number of pop-transitions in \( \overline{r} \). If the output string produced by the sequence \( \overline{r} \) is \( \phi \) and the destination of the final transition in \( \overline{r} \) is \( q' \), we use the notation \( q \xrightarrow{w/\phi} M q' \) to denote the fact that \( M \) produces output \( \phi \) on the well-matched input \( w \) from \( q \), and the state changes from \( q \) to \( q' \).

We say that \( M \) can reach the state \( q \) and can produce the output \( \alpha_1 \alpha_2 \ldots \alpha_k \) on input \( w \) if \( w \) is a valid input from some state \( q \in Q_0 \) and for some sequence of transitions \( t_1 t_2 \ldots t_k \) consistent with a run from \((q, \bot)\) on input \( w \), the output of \( t_j \) is \( \alpha_j \) for every \( j \in [k] \) and the destination of \( t_k \) is \( q' \). Let \( M(w) \) denote the set of all strings that \( M \) can produce on input \( w \).

Remark 2. Note that an FSM \( M = (Q, q_0, \delta, \lambda) \) is a PDM with several restrictions:

1. There is a unique initial state \( q_0 \).

2. The stack alphabet \( \Gamma = \emptyset \). As a consequence, the set of call transitions \( \delta_{\text{call}} \) and return transitions \( \delta_{\text{ret}} \) are both empty.

3. The transition relation \( \delta_{\text{int}} = \{ (q, a, q', \alpha) \mid \delta(q, a) = q' \text{ and } \lambda(q, a) = \alpha \} \)
   is deterministic since both \( \delta \) and \( \lambda \) are deterministic.

### 2.2.2 Visibly Pushdown Automata

Visibly Pushdown Automata (VPAs) are restricted pushdown automata, the latter being automata variants of PDMs. Unlike the case of pushdown automata, it turns out that deterministic VPAs are as powerful as a non-deterministic VPAs [10]. In light of this, we will consider only deterministic VPAs. A visibly pushdown automaton (VPA) on finite strings over \( \widehat{\Sigma} = (\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \Sigma_{\text{int}}) \) is a tuple \( M = (Q, q_0, \Gamma, \delta, Q_F) \) where \( Q \) is a finite set of states, \( q_0 \in Q \) is the initial state, \( \Gamma = \Gamma' \cup \{ \bot \} \) is a finite stack alphabet that contains a special bottom-of-stack symbol \( \bot, \delta = \delta_{\text{call}} \cup \delta_{\text{ret}} \cup \delta_{\text{int}} \) is the transition function, where \( \delta_{\text{call}} : Q \times \Sigma_{\text{call}} \to Q \times \Gamma', \delta_{\text{ret}} : Q \times \Sigma_{\text{ret}} \times \Gamma \to Q \), and \( \delta_{\text{int}} : Q \times \Sigma_{\text{int}} \to Q \), and \( Q_F \subseteq Q \) is a set of final states.
Call (push), return (pop) and internal transitions are defined as natural deterministic counterparts of our earlier PDM definitions, with the only significant difference being the behavior on popping the bottom-of-stack symbol $\bot$. If $\delta_{\text{call}}(q, c) = (q', \gamma)$, where $c \in \Sigma_{\text{call}}$ and $\gamma \neq \bot$, there is a call-transition from $q$ on input $c$ where on reading $c$, $\gamma$ is pushed onto the stack and the control changes from state $q$ to $q'$; we denote such a transition by $q \xrightarrow{c} (q', \gamma)$. Similarly, if $\delta_{\text{ret}}(q, r, \gamma) = q'$, there is a return-transition from $q$ on input $r$ where $\gamma$ is read from the top of the stack and popped (if the top of the stack is $\bot$, then it is read but not popped), and the control changes from $q$ to $q'$; we denote such a transition by $q \xrightarrow{r/\gamma} (q', \gamma)$. If $\delta_{\text{int}}(q, i) = q'$, there is an internal-transition from $q$ on input $i$ where on reading $i$, the state changes from $q$ to $q'$; we denote such a transition by $q \xrightarrow{i} (q', \gamma)$. Note that there are no stack operations on internal transitions.

We define the set of all stacks $St = \Gamma^* \cdot \bot$ and the set of all configurations $Q \times St$ exactly as for PDMs. The notion of VPA acceptance once again is a natural counterpart of the notion of strings produced by a PDM on a certain input. Once again, the transition function of a VPA can be used to describe how the configuration of the machine changes in a single step: we say $\delta((q, s), a) = (q', s')$ if one of the following holds:

1. If $a \in \Sigma_{\text{call}}$ then there exists $\gamma \in \Gamma'$ such that $\delta_{\text{call}}(q, a) = (q', \gamma)$ and $s' = \gamma \cdot s$.
2. If $a \in \Sigma_{\text{ret}}$, then there exists $\gamma \in \Gamma'$ such that $\delta_{\text{ret}}(q, a, \gamma) = q'$ and either $\gamma \neq \bot$ and $s = \gamma \cdot s'$, or $\gamma = \bot$ and $s = s' = \bot$.
3. If $a \in \Sigma_{\text{int}}$ is an internal action, then $\delta_{\text{int}}(q, a) = q'$ and $s' = s$.

The transitive closure of the single-step transition function, which we also denote by $\delta$, can be easily defined in the standard inductive manner. For a stack $s \in St$, we define the function $\delta_s : Q \times \Sigma^* \to Q$ as $\delta_s(q, u) = q'$ whenever $\delta((q, s), u) = (q', s')$ for some $s' \in St$.

A string $u \in \Sigma^*$ is accepted by VPA $M$ if $\delta_{\bot}(q_0, u) \in Q_F$. The language of $M$, $L(M)$, is the set of strings accepted by $M$.

**Visibly Pushdown Languages**

A language over finite strings $L \subseteq \Sigma^*$ is a visibly pushdown language (VPL) with respect to $\hat{\Sigma}$ (a $\hat{\Sigma}$-VPL) if there is a VPA $M$ over $\hat{\Sigma}$ such that $L(M) = L$. Since VPAs can push onto (respectively pop from) the stack only when on symbols from $\Sigma_{\text{call}}$ (respectively $\Sigma_{\text{ret}}$), the notion of well-matched strings is considerably more structured in this context.

Let $MR(\hat{\Sigma})$ denote the set of all strings where every return has a matched call before it, i.e., $u \in MR(\hat{\Sigma})$ if for every prefix $u'$ of $u$, the number of return

\footnote{We abuse notation and use $\delta$ for both the transition function of the automaton and the single step transition function on configurations.}
symbols in \( u' \) is at most the number of call symbols in \( u' \). Similarly, let \( MC(\hat{\Sigma}) \) denote the set of all strings where every call has a matching return after it, i.e., \( u \in MC(\hat{\Sigma}) \) if for every suffix \( u' \) of \( u \), the number of call symbols in \( u' \) is at most the number of return symbols in \( u' \). The set of well-matched strings over \( \hat{\Sigma} \) is \( WM(\hat{\Sigma}) = MR(\hat{\Sigma}) \cap MC(\hat{\Sigma}) \). A \( \hat{\Sigma} \)-VPL \( L \) is said to be well-matched if \( L \subseteq WM(\hat{\Sigma}) \).

**Remark 3.** For every \( w \in WM(\hat{\Sigma}) \), there is a unique matching between call and return symbols such that every call-symbol always precedes its matching return-symbol and the substring \( u' \) between a matching pair of call and return symbols is a well-matched string.

**A Note on Notation**

For a non-negative integer \( n \), we will use the notation \([n]\) to denote the set \( \{1, 2, \ldots, n\} \).
Chapter 3

Conformance Testing for Non-recursive Program Models

3.1 Introduction

Conformance testing is an important problem in verification, where one tries to determine if the implementation of a program adheres to the prescribed specification. In this chapter, we will focus on the conformance testing problem for the most typical specification model, namely a deterministic finite state Mealy machine (FSM) that produces outputs whenever it makes a state transition. The implementation is assumed to be a deterministic “black-box” machine whose internal state transition structure is assumed to be unknown. However, one can test the implementation by applying input symbols and observing the outputs it produces.

Since Moore’s seminal work [53] that introduced the framework of testing finite state machines, there is an extensive literature on the problem of testing such machines [29, 65, 38, 30, 22, 48, 7], and the fault detection problem [63, 42, 58] in particular. Major results are summarized in [41, 28, 51] (see [49] for a recent survey). A number of algorithms for conformance testing in special cases have been proposed: the D-method based on distinguishing sequences [35] (a distinguishing sequence is a sequence of input symbol that helps identify the initial state based on the output produced in response); the U-method based on UIO sequences [63] (a UIO sequence helps check if the initial state was correct); the T-method based on transition tours [55]; checking sequence based on reliable resets [19] are some examples. In an influential paper, Yannakakis and Lee [70] relaxed some of the special constraints imposed by previous algorithms and gave a randomized construction of a polynomially long checking sequence.

All previous algorithms, including the Lee and Yannakakis algorithm, made some common assumptions about the specification and implementation machines. The first is that the specification is a minimal finite state machine and that the specification and implementation are strongly connected, i.e., for any pair of states in the specification (respectively, implementation), there is an input sequence that transfers the machine from the first state to the second state. The second more restrictive assumption is that the implementation does not have more states than the specification. This second assumption is made is

Typically this is not a serious constraint since specifications and implementations often have a reset transition, which guarantees strong connectivity.
because in order to detect faulty implementations that have more states than the specification, the length of the checking sequence needs to be exponential in the number of extra states in the implementation [68]. However, the second assumption severely restricts the applicability of these methods in practice. This is because implementations (being more detailed) typically have many more states than the specification.

In this chapter we investigate whether this restriction on the size of the implementation can be relaxed, and at the same time obtain polynomially long testing sequences that can provide useful information about the presence of faults in the implementation. More specifically, as in property testing [62, 61], we study the conformance testing problem in a “promise” setting, namely, one where the implementations with additional states are promised to either have multiple faults or be completely correct. Once again, like in property testing, the number of faults in an implementation is measured by the Hamming distance to the closest correct implementation †. We call a sequence of inputs a \((r, \Delta)-\text{approximate checking sequence}\) if all implementations with at most \(\Delta\) extra states and at least \(r\) faults give a different output than the specification on this sequence, while correct implementations give the same output as the specification. We present a randomized algorithm that given \(r\) and \(\Delta\), constructs a \((r, \Delta)-\text{approximate checking sequence}\) with high probability. It must be noted that the probability of error is over the random choices made by the algorithm and not on a particular distribution on the implementation machines. Secondly the probability of error can be made as small as needed by increasing the length of the checking sequence.

Investigating conformance testing in the promise setting might seem unreasonable in the light of the fact that implementations usually have few subtle errors, rather than many. Nonetheless, we believe that the results presented here are interesting for the following reasons. First, this relaxation of the fault detection problem is a natural mathematical generalization, and one which has been fruitfully studied in the context of property testing ([31, 4, 3, 2, 14, 32, 18, 26, 57, 25, 5]; see [61] for a survey). In addition, apart from its theoretical interest, we believe our constructions will be useful in practice. This is primarily because although we can prove the success (with high probability) of our tests only on machines with at least \(r\) faults, the tests have a non-zero probability of detecting every faulty machine (including those with less than \(r\) faults). Thus our algorithm’s output can be used as a heuristic test that applies in all situations. Second, our algorithm provides the debugging team a hierarchy of test suites of increasing precision to choose from based on practical time constraints imposed on the testing process by product release times.

†There are however important differences between the problem considered here and the property testing setting. First, we are not solving a decision problem. Second, we measure the absolute Hamming distance, as opposed to the distance relative to the size of the input. And finally, we measure complexity in terms of the length of the test suite generated and not the sample complexity.
3.1.1 Our Results

We present two randomized algorithms. The first algorithm constructs an \((r, \Delta)\)-approximate checking sequence to detect implementations with a large number of faults; more precisely when \(r > d \min(n, \Delta)\), where \(d\) is the size of the input alphabet and \(n\) is the size of the specification. This algorithm is essentially identical to that of Yannakakis and Lee [70]. Our contribution here lies in showing that this algorithm works in this new setting by observing that the presence of a large number of errors allows one to search for faults in a small neighbourhood, and thus ignore the fact that the implementation has extra states. The second algorithm constructs approximate checking sequences for the case when the number of errors is small, i.e., when \(r \leq d \min(n, \Delta)\). This algorithm relies on performing a random walk to reach a target set \(T\) among the “unknown” extra states \(X\) of the implementation. In general, exploration via random walks takes time exponential in \(|X|\) due to the existence of so-called combination-lock subgraphs \(^1\). The hardness of exploring digraphs via random walks has also been characterized in terms of spectral and connectivity properties [23, 47, 52]. For special classes of digraphs, better bounds can be obtained, e.g., polynomial-time upper bounds exist when the digraph is symmetric (i.e., undirected) [54, 64], and when the indegree of each vertex is equal to the outdegree \([1]\). Good bounds can also be obtained for digraphs with high expansion \(^2\) [52]. However, these results do not directly apply to the digraphs we consider here. To prove the correctness of our second algorithm, we derive bounds on the probability that a random walk on a directed graph eventually reaches a target set \(T\) as a function of the cut-size of \(T\), i.e., the number of edges that need to be removed to disconnect \(T\) from an initial set of vertices. We believe that our results on random walks are of independent interest and may help design provably (time and space) efficient algorithms for verifying safety properties [20] based on random walks. We also present lower bounds for the problems considered here. Our lower bounds demonstrate that our algorithms for constructing tests are in fact very close to being optimal.

The rest of the chapter is organized as follows. Our results on random walks are presented in Section 3.2. Section 3.3 describes the setting for the fault detection experiment and formally defines approximate checking sequences. Our randomized algorithms for constructing tests for detecting a single faulty machine are presented in Section 3.4; these algorithms are used in Section 3.6 to construct approximate checking sequences. Lower bounds are outlined in Section 3.5. Finally in Section 3.7 we discuss the upper and lower bounds presented here, and determine bounds on \(\Delta\) and \(r\) when the checking sequence

---

\(^1\)In particular, \(T\) is reachable only via a particular sequence of edges (called the “combination”) of length \(|X| - |T|\), which can be as large as \(\Omega(|X|)\).

\(^2\)The expansion of a digraph \(G = (V, E)\) is the minimum value of \(\frac{1}{|V|/2} \max \{|N^+(S)|, |N^-(S)|\}\) taken over all subsets \(S\) of \(V\) for which \(0 < |S| \leq |V|/2\), where \(N^+(S) = \{v \in V \setminus S \mid \{u, v\} \in E \text{ and } u \in S\}\) and \(N^-(S) = \{v \in V \setminus S \mid \{v, u\} \in E \text{ and } u \in S\}\).
can be polynomially long.

### 3.1.2 Directed Graphs, Walks and Cuts

A *labeled directed graph* (or digraph) \( G = (V, E, \Sigma) \) is a directed graph with vertex set \( V \), label set \( \Sigma \) and edge function \( E : V \times \Sigma \rightarrow V \). We use the notation \( u \xrightarrow{\sigma} v \) to denote \( E(u, \sigma) = v \). When the digraph \( G \) is clear from the context, we abbreviate this to \( u \xrightarrow{\sigma} v \). Note that if \( M = (Q, q_0, \delta, \lambda) \) is an FSM, then \( G_M = (Q, \delta, \Sigma) \) is a labeled digraph.

A *walk* of length \( k \) in \( G = (V, E, \Sigma) \) is a sequence \( ((t_1, \sigma_1), (t_2, \sigma_2), \ldots, (t_k, \sigma_k)) \in (V \times \Sigma)^k \) such that \( t_i \xrightarrow{\sigma_i} t_{i+1} \) for every \( i \in [k-1] \). Note that a walk is completely specified by its initial state \( t_1 \) and the string of edge labels \( \sigma_1 \sigma_2 \ldots \sigma_k \in \Sigma^k \). A *random walk* in \( G = (V, E, \Sigma) \) of length \( k \) starting from a set \( U \subseteq V \) is a walk \(((t_1, \sigma_1), \ldots, (t_k, \sigma_k))\) where the initial vertex \( t_1 \) is chosen uniformly at random from \( U \), and the string of edge labels \( \sigma_1 \sigma_2 \ldots \sigma_k \) is chosen uniformly at random from \( \Sigma^k \). We say that \( t_i \) is the \( i \)th vertex and \((t_i, \sigma_i)\) is the \( i \)th edge in the random walk \( R \).

A *path* is a walk \(((t_1, \sigma_1), \ldots, (t_k, \sigma_k))\) such that \( t_i \neq t_j \) for every \( i \neq j \). Two paths \(((t_1, \sigma_1), \ldots, (t_k, \sigma_k))\) and \(((t'_1, \sigma'_1), \ldots, (t'_k, \sigma'_k))\) are *edge-disjoint* if there is no pair \((i, j)\) such that \((t_i, \sigma_i) = (t'_j, \sigma'_j)\).

Let \( U \subseteq V \) and \( X \subseteq V \times \Sigma \). A set \( C \subseteq V \times \Sigma \) is called a *(U,X)-cut* in \( G = (V, E, \Sigma) \) if every path \(((t_1, \sigma_1), \ldots, (t_k, \sigma_k))\) in \( G \) such that \( t_1 \in U \) and \((t_k, \sigma_k) \in X \) contains some edge \((t_i, \sigma_i) \in C \). A set of *(U,X)-cut-paths* \( P \) is a set of edge-disjoint paths such that each \( p = ((t_1, \sigma_1), \ldots, (t_k, \sigma_k)) \in P \) has the following properties:

1. \( t_1 \in U \) and \( t_i \notin U \) for each \( i = 2, \ldots, k \);  
2. \( (t_k, \sigma_k) \in X \) and \( (t_i, \sigma_i) \notin X \) for each \( i = 1, \ldots, k-1 \).

We say that \( C \) is a *minimum* *(U,X)-cut* if \( C \) is a *(U,X)-cut* and there is no *(U,X)-cut* of cardinality strictly less than \( |C| \). The following Lemma guarantees the existence of \( |C| \) edge-disjoint *(U,X)-cut-paths*, where \( C \) is a minimum *(U,X)-cut*.

**Lemma 2.** Let \( G = (V, E, \Sigma) \) be a labeled digraph. If \( U \subseteq V \), \( X \subseteq V \times \Sigma \) and \( C \) is a minimum *(U,X)-cut* in \( G \), then there is a set \( P_C \) of \(|C|\) edge-disjoint paths in \( G \) such that each \( p = ((t_1, \sigma_1), \ldots, (t_k, \sigma_k)) \in P_C \) has the following properties:

1. \( t_1 \in U \) and \( t_i \notin U \) for each \( i = 2, \ldots, k \);  
2. \( (t_k, \sigma_k) \in X \) and \( (t_i, \sigma_i) \notin X \) for each \( i = 1, \ldots, k-1 \).

**Proof:** Consider the *flow network* \( G' \) obtained from \( G \) by adding a new *source* vertex \( s \) with infinite-capacity edges to every vertex in \( U \), a new *sink* vertex \( t \) with all edges in \( X \) as infinite-capacity incoming edges, and unit capacities on
all other edges. By the construction, note that $C$ is a minimum $s-t$ cut in the flow network $G'$. By the Maxflow-Mincut theorem [27], there is an integer flow of value $|C|$ in $G'$. The flow-carrying edges of $G$ in $G'$ form $|C|$ edge-disjoint paths with the required properties. \hfill \Box

### 3.2 Random Walks on Digraphs

In this section, we present our technical results on random walks on directed graphs. Lemma 2 guarantees the existence of several edge-disjoint cut-paths. Our first goal is to show that many of these paths are not very long.

**Lemma 3.** Let $G = (V, E, \Sigma)$ be a labeled digraph such that $|V| \leq n + \Delta$ and $|\Sigma| = d$. Further, suppose $U \subseteq V$, $|U| = n$ and $X \subseteq (V \setminus U) \times \Sigma$ such that every $(U, X)$-cut in $G$ has size at least $r > 0$, where $r \leq d \min(n, \Delta)$. We consider two cases: (i) $r$ is $o(d)$, and (ii) there is some constant $c > 1$ such that $r \geq \frac{c}{\Delta}$. Define

$$
\rho = \begin{cases} 
  r & \text{if } r \text{ is } o(d) \\
  r(1 - \frac{1}{r}) & \text{otherwise}
\end{cases}
$$

$$
\lambda = \begin{cases} 
  1 + \Delta & \text{if } r \text{ is } o(d) \\
  1 + \frac{cd\Delta}{r} & \text{otherwise}
\end{cases}
$$

Then there are at least $\rho$ edge-disjoint $(U, X)$-cut-paths each of length at most $\lambda$ satisfying the conditions of Lemma 2.

**Proof:** The claim is clearly true when $r$ is $o(d)$, since the size of a minimum $(U, X)$-cut is at least $r$ and hence by Lemma 2 there are at least $r = \rho$ cut-paths, each of length at most $1 + \Delta = \lambda$.

Suppose there is some constant $c > 1$ such that $r \geq \frac{c}{\Delta}$. Lemma 2 guarantees the existence of $r$ edge-disjoint $(U, X)$-cut-paths. Suppose $N$ of these have length at least $2 + \frac{cd\Delta}{r}$. Since all cut-paths are edge disjoint, the total number of edges other than the first in all such cut-paths is at least $N(1 + \frac{cd\Delta}{r})$. Now $N(1 + \frac{cd\Delta}{r}) \leq d\Delta$ since every edge other than the first in a cut-path is of the form $(t, \sigma)$ where $t \notin U$. Hence, there are at most $\frac{c}{\Delta}$ cut-paths of length at least $2 + \frac{cd\Delta}{r}$. Therefore, in this case too, there are at least $(1 - \frac{1}{r})r = \rho$ cut-paths, each of length at most $1 + \frac{cd\Delta}{r} = \lambda$. \hfill \Box

We are now ready to prove our main lemma about random walks in directed graphs.

**Lemma 4.** Let $G = (V, E, \Sigma)$ be a labeled digraph such that $|V| \leq n + \Delta$ and $|\Sigma| = d$. Further, suppose $U \subseteq V$, $|U| = n$ and $X \subseteq (V \setminus U) \times \Sigma$ such that every $(U, X)$-cut in $G$ has size at least $r > 0$, where $r \leq d \min(n, \Delta)$. Let $P$ be the probability that the last edge of a random walk $R$ in $G$ beginning from $U$ of length $l$ (where $l$ is chosen uniformly at random from $\{2, \ldots, \lambda\}$) is in $X$. Then

$$
P \geq \frac{\rho^{l-1}}{\lambda^{l-1}} \left( \frac{d}{4\Delta^2} \right)^{\lambda-1},
$$

where $\rho$ and $\lambda$ are as defined in Lemma 3.
Proof: Let \( p_1, p_2, \ldots, p_r \) be the \( \rho \) edge-disjoint cut-paths of length at most \( \lambda \) guaranteed by Lemma 3.

Consider first the case when the length of each cut-path is exactly \( \lambda \). For each \( i = 0, 1, \ldots, \lambda - 1 \), let \( L_i \) be the set of \((i + 1)\)th vertices of these cut-paths (note that \( L_0 \subseteq U \)) and let \( P_i \) be the probability that each of the first \((i + 1)\) edges of the random walk \( R \) is an edge in some cut-path.

Since \(|L_0| \leq |U| = n\), it follows by the definition of \( R \) that \( P_0 \geq \frac{\rho}{m'} \) and \( P_1 \geq \left( \frac{\rho}{d} \right) \frac{P_0}{m' - 1} \). We now compute lower bounds for \( P_2, \ldots, P_{\lambda - 1} \).

For each \( t \in [\lambda - 2] \), let \( L_t = \{t_1, t_2, \ldots, t_m\} \) and \( L_{t+1} = \{t'_1, t'_2, \ldots, t'_m\} \), where \( m, m' \leq \Delta \). Further, let

1. there are \( y_{ij} \) edges belonging to cut-paths from \( t_j \) to \( t'_i \). Let \( x_j = \sum_{i=1}^{m'} y_{ij} \) for every \( j \in [m] \).
2. there are \( x'_i \) out-edges belonging to cut-paths from \( t'_i \) for every \( i \in [m'] \).
3. for each \( j \in [m] \), the probability that the \((t + 1)\)th vertex of \( R \) is \( t_j \) is \( p_j \).
4. for each \( i \in [m'] \), the probability that the \((t + 2)\)th vertex of \( R \) is \( t'_i \) is \( p'_i \).

Then \( P_t = \frac{1}{m} \sum_{j=1}^{m} p_j x_j \) and \( P_{t+1} = \frac{1}{m'} \sum_{i=1}^{m'} p'_i x'_i \).

We know that:

\[
\sum_{j=1}^{m} x_j = \sum_{i=1}^{m'} x'_i \tag{3.1}
\]

\[
\sum_{j=1}^{m} y_{ij} = x'_i \tag{3.2}
\]

\[
x_j = \sum_{i=1}^{m'} y_{ij} \tag{3.3}
\]

\[
\sum_{j=1}^{m} x_j \geq \rho \tag{3.4}
\]

By the definition of the random walk \( R \), we have for every \( i \in [m'] \)

\[
p'_i \geq \sum_{j=1}^{m} \frac{p_j y_{ij}}{d} \tag{3.5}
\]

We first claim that for all \( t \in [\lambda - 2] \) and for all \( m, m' \leq \Delta \):

\[
P_{t+1} \geq \frac{\rho}{dmm'} P_t \tag{3.6}
\]

We prove Equation 3.6 by induction on \( m \).

Base case \( m = 1 \): Let \( x' = \frac{1}{m'} \sum_{i=1}^{m'} x'_i \). Simplifying the expression \( \sum_{i=1}^{m'} (x'_i - \frac{1}{m'} \sum_{i=1}^{m'} x'_i) \),
\[ (x')^2 \geq 0 \] yields the fact that \[ \sum_{i=1}^{m'} (x'_i)^2 \geq \frac{1}{m'} \left( \sum_{i=1}^{m'} x'_i \right)^2. \] Now,

\[
\sum_{i=1}^{m'} \left( \sum_{j=1}^{m} p_j y_{ij} \right) x'_i = p_1 \sum_{i=1}^{m'} (x'_i)^2 \quad \text{by Equation 3.2}
\]

\[
\geq \frac{\rho}{m'} \left( \sum_{i=1}^{m'} x'_i \right)^2 \quad \text{by the above fact}
\]

\[
\geq \frac{\rho}{1/m'} \sum_{j=1}^{m} p_j x_j \quad \text{by Equation 3.4 and Equation 3.1}
\]

This proves the base case.

**Inductive step m > 1:** Without loss of generality, suppose \( x_m \geq x_j \) for all \( j \in [m] \). It follows from Equation 3.4 that

\[
x_m \geq \frac{\rho}{m}
\] (3.7)

Let

\[
Y_{ij} = \begin{cases} 
  y_{ij} & \text{if } j < m - 1 \\
  y_{im-1} + y_{im} & \text{otherwise}
\end{cases}
\]

and

\[
X_j = \begin{cases} 
  x_j & \text{if } j < m - 1 \\
  x_{m-1} + x_m & \text{otherwise}
\end{cases}
\]

Note that, as in Equation 3.2, for every \( i \in [m'] \)

\[
\sum_{j=1}^{m-1} Y_{ij} = x'_i
\] (3.8)

and

\[
\sum_{j=1}^{m-1} X_j = \sum_{j=1}^{m} x_j
\] (3.9)

We have

\[
\sum_{i=1}^{m'} \left( \sum_{j=1}^{m} p_j y_{ij} \right) x'_i = \sum_{i=1}^{m'} \left( \sum_{j=1}^{m} p_j y_{ij} \right) x'_i + p_m \sum_{i=1}^{m'} y_{im} x'_i - p_{m-1} \sum_{i=1}^{m'} y_{im} x'_i \quad \text{by definition of } Y_{ij}
\]

\[
\geq \frac{\rho}{(m-1)m'} \sum_{j=1}^{m-1} p_j X_j + (p_m - p_{m-1}) \sum_{i=1}^{m'} y_{im} x'_i \quad \text{by the ind. hyp. and Equations 3.8, 3.9}
\]

\[
\geq \frac{\rho}{(m-1)m'} \sum_{j=1}^{m-1} p_j X_j + (p_m - p_{m-1}) \frac{x_m^2}{m}- \quad \text{by the above fact } (y_{im} \leq x'_i), \text{ Equation 3.3}
\]

\[
\geq \frac{\rho}{mm'} \sum_{j=1}^{m} p_j X_j + \frac{\rho}{mm'} (p_m - p_{m-1}) x_m \quad \text{by Equation 3.7}
\]

\[
= \frac{\rho}{mm'} \left( \sum_{j=1}^{m} p_j x_j + p_m x_m \right) \quad \text{by definition of } X_j
\]

\[
= \frac{\rho}{mm'} \sum_{j=1}^{m} p_j x_j
\]

Hence by induction, Equation 3.6 holds. Thus we have

\[
P_{\lambda-1} \geq \frac{\rho}{nd} \left( \frac{\rho}{d} \right)^{\lambda-1} \frac{1}{\prod_{i=1}^{\lambda-1} |L_i|^2}
\] (3.10)

Now consider the general case. For every \( i \in [\lambda - 1] \), let \( L'_i \) be the set
Figure 3.1. Construction of Extended Cut Paths

of $(i + 1)^{th}$ vertices of $p_1, \ldots, p_\rho$. Note that $|L'_i| \leq \Delta$ for every $i$. For every $i \in [\lambda-1]$, let $n_i$ be the number of cut-paths among $p_1, \ldots, p_\rho$ of length at most $i$ (note that $n_1 = 0$ since every cut-path has length at least 2). Let $L''_i$ be a set of $\max(1, \frac{\lambda}{\rho})$ new vertices. Note that $|L''_i| \leq \max(1, \frac{\lambda}{\rho}) \leq \Delta$ for every $i$.

Let $L_0 = U$ and for every $i \in [\lambda-1]$, let $L_i = L'_i \cup L''_i$. Let $L = \bigcup_{i=1}^{\lambda-1} L_i$.

We claim that there is a digraph $G' = (V', E', \Sigma)$ such that

1. $V' = V \cup L \cup \{s\}$, where $s$ is a new vertex not in $V \cup L$;

2. for every $p = p_1, \ldots, p_\rho$, if $p = (t_0, \sigma_0 \ldots \sigma_k)$ then there is a path $p' = (t_0, \sigma_0 \ldots \sigma_k \sigma_{k+1} \ldots \sigma_\lambda)$ in $G'$ such that for every $i > k$, $t_i \in L''_i$, and for each $i \neq j$, $p'_i$ and $p'_j$ are edge-disjoint; and

3. for every $(t, \sigma) \in (V \times \Sigma) \setminus X$, $E'(t, \sigma) = E(t, \sigma)$.

This is easy to see, since $G'$ can be obtained from $G$ by adding the new vertices (in the sets $L''_i$) and edges by “extending” the cut-paths $p_1, \ldots, p_\rho$ in the appropriate manner as shown in Figure 3.1. We call the paths $p'_1, \ldots, p'_\rho$ extended cut-paths in $G'$. Let $X' \subseteq V' \times \Sigma$ be the set of last edges of all extended cut-paths. Thus, there are $\rho$ (extended) cut-paths from $U$ to $X'$ in $G'$, each of length $\lambda$. Every extended cut-path $p'$ in $G'$ has a prefix $p$ of length $k$ ($2 \leq k \leq \lambda$) that is a cut-path in $G$.

Let $P'$ be the probability that each edge of a random walk in $G'$ beginning from $U$ of length $\lambda$ is an edge of some extended cut-path in $G'$. Using the above inequality 3.10 and noting that $|L_i| \leq 2\Delta$ for every $i$, we have

$$P' = P_\lambda \geq \frac{\rho}{\lambda d} \left( \frac{\rho}{4d\Delta^2} \right)^{\lambda-1}$$

The probability that the last edge of a prefix of length $l$ (where $l$ is chosen uniformly at random from $\{2, \ldots, \lambda\}$) of such a random walk is the last edge of some cut-path is $\frac{p'}{\lambda-1}$. Clearly, $P' \geq \frac{p'}{\lambda-1}$, and hence the lemma holds. □
3.3 Conformance Testing of FSMs

We now describe the setting for conformance testing. We are given a specification machine \( S \) and a “black-box” implementation \( I \). Both \( I \) and \( S \) are assumed to be FSMs with input alphabet \( \Sigma \) and output alphabet \( \Omega \). In order to test if \( I \) is equivalent to \( S \) (defined later) we make some assumptions:

1. \( S = (Q, \delta, \lambda) \) is a minimal, strongly connected FSM;
2. \( Q = \{ q_1, q_2, \ldots, q_n \} \);
3. \( I = (Q', \delta', \lambda') \) does not change during the testing experiment;
4. \( I \) is a strongly connected FSM;
5. \( |Q'| \leq n + \Delta \) for some \( \Delta > 0 \).

The assumption of strong connectivity of \( S \) is needed because otherwise the test may not be able to visit all the states of \( S \) and thereby guarantee the equivalence of \( I \) to \( S \); typically this is not a serious constraint since specifications often have a reset transition. The requirement that \( S \) be minimal can be easily met by minimizing the specification, if it is not already minimal. The need for assumption (3) is obvious. The requirement that \( I \) be strongly connected is a technical requirement based on ensuring that the homing sequence applied before the test leads \( I \) to the right state (see discussion below). If we are guaranteed that \( I \) starts in a known state, we can drop this requirement. Finally, in order to construct the checking sequence we need to know an upper bound on the number of states in \( I \); without this knowledge constructing the checking sequence is impossible.

Under the assumptions discussed above, we would like to test if the implementation is equivalent to the specification or has multiple faults. We first define the notion of equivalence (trace) that we use.

**Definition 4.** Let \( S = (Q, \delta, \lambda) \) and \( I = (Q', \delta', \lambda') \) be two FSMs. A state \( q \in Q \) is said to be equivalent to \( q' \in Q' \) (denoted by \( q \simeq q' \)) iff for every \( w \in \Sigma^* \), \( \lambda(q, w) = \lambda'(q, w) \). \( S \) is said to be equivalent to \( I \) (denoted as \( S \simeq I \)) if and only if some state \( q \in Q \) is equivalent to some state \( q' \in Q' \).

**Remark 4.** Observe that if \( S \) is strongly connected and \( S \simeq I \) then for every \( q \in Q \), there is a \( q' \in Q' \) such that \( q \simeq q' \). A similar statement about the states of \( I \) holds if \( I \) is strongly connected.

Note that our notion of a specification does not include an initial state. If the specification includes a designated initial state then the notion of equivalence will require, in addition, that the initial state of the implementation be equivalent. Thus, in such a situation, the problem consists of two parts: (1) verify that the black box \( I \) is equivalent to \( S \), and (2) verify that \( I \) starts in a
state that is equivalent to the initial state of \( S \), (say) \( q_1 \). The second problem, which is called the state verification problem, requires the existence of a sequence of inputs (called unique input-output (UIO) sequence) that distinguishes the starting state \( q_1 \) from all other states based on the outputs produced on the UIO sequence. UIO sequences need not exist for all minimal strongly connected FSMs, and so a test may not exist if our specification includes an initial state.

Thus, \( I \) may be equivalent to \( S \) but start in an unknown state. The first part of a fault detection experiment, therefore, applies what is called a homing sequence (see [41]) that brings \( I \) to a known state \( q'_1 \), which is then the start state of the second part of the experiment. We require the implementation to be strongly connected to ensure that if \( I \) is equivalent to \( S \) then the homing sequence will work correctly. If \( I \) is known to start in a state equivalent to some state in the specification, then we do not require \( I \) to be strongly connected. Since homing sequences have been well understood for decades, the key challenge in designing a fault detection experiment is therefore to check if \( I \) is equivalent to \( S \), and this is the problem we address here.

**Definition 5.** We say that \( I \) is at distance \( r \) from \( S \) if \( D \subseteq Q' \times \Sigma \) with \( |D| = r \), is a set of smallest size for which there is a FSM \( J = (Q'', \delta'', \lambda''') \) such that

1. \( Q'' = Q' \),
2. \((\delta'', \lambda''')\) and \((\delta', \lambda')\) differ only on the set \( D \), and
3. \( S \) is equivalent to \( J \).

Note, that \( J \) need not be strongly connected in the above definition. Also observe that if \( I \) is at distance \( r \geq 1 \) from \( S \), then \( S \) is not equivalent to \( I \).

**Definition 6.** An \((r, \Delta)\)-approximate checking sequence for an \( n \) state FSM \( S \) is an input sequence \( w \) such that every FSM \( I \) with at most \( n + \Delta \) states that is at distance \( r \) or more from \( S \) produces on input \( w \) a different output than that produced by \( S \) starting from state \( q_1 \), and every FSM \( I \) with at most \( n + \Delta \) states that is equivalent to \( S \) produces on input \( w \) the same output that \( S \) produces starting from state \( q_1 \).

Observe, that \((1, 0)\)-approximate checking sequence is usually simply called a checking sequence in the literature; such a sequence checks whether the implementation (with no additional states) is indeed isomorphic to the specification.

A configuration is a pair of states \((q, q') \in Q \times Q'\); it is used to denote the states of \( S \) and \( I \) during some stage in the conformance testing experiment. Recall, that Proposition 1 says that every minimal FSM has a separating family. Let \( Z = \{ Z_i \mid i \in [n] \} \) be a separating family for \( S \) (we use the shorthand notation \( Z_i \) to denote \( Z_{q_i} \)). We say that state \( q' \in Q' \) is similar to state \( q_i \in Q \) (which we denote by \( q' \sim q_i \)) if for every \( w \in Z_i \), \( \lambda(q_i, w) = \lambda'(q', w) \). We define the similarity function \( \pi : Q' \rightarrow Q \cup \{ \perp \} \) as \( \pi(q') = q_i \) if \( q' \sim q_i \), and \( \pi(q') = \perp \)
otherwise. Note that \( \pi(q') \) is well defined since a state \( q' \in Q' \) can be similar to at most one state \( q_i \in Q \) (since \( S \) is minimal). We define the \textit{error set} \( X \) w.r.t. \( S \) and \( I \) as \( X = X_1 \cup X_2 \cup X_3 \subseteq (Q' \times \Sigma) \) where \( X_1 \) and \( X_2 \) are the “transfer-errors”:

\[
X_1 = \{ (q', a) \mid \pi(\delta(q, a)) = \perp \} \\
X_2 = \{ (q', a) \mid \pi(q') \neq \perp, \pi(\delta'(q', a)) \neq \delta(\pi(q'), a) \}
\]

and \( X_3 \) is the set of “output-errors”:

\[
X_3 = \{ (q', a) \mid \pi(q') \neq \perp \text{ and } \lambda'(q', a) \neq \lambda(\pi(q'), a) \}
\]

If \( U \subseteq Q' \), a set \( C \subseteq Q' \times \Sigma \) is called a \((U, X)\)-cut in \( I \) if \( C \) is a \((U, X)\)-cut in the labeled digraph \( G_T = (Q', \delta', \Sigma) \).

Note that \( S \) is equivalent to \( I \) if and only if \( X = \emptyset \), i.e., there are no erroneous transitions. If it turns out that \( S \) and \( I \) are inequivalent, the task is to identify some erroneous edge. The simplest way to accomplish this is to search for such an edge in the “current vicinity”. We now formalize this intuition. For every \( i \in [n] \) let \( T_i \) be some fixed (directed) breadth-first search (BFS) tree in \( G_S = (Q, \delta, \Sigma) \) with root \( q_i \). For every \( j \in [n] \), let \( w_{ij} \) be the (unique) string such that \( (q_i, w_{ij}) \) is the unique path from \( q_i \) to \( q_j \) in \( T_i \). Thus \( \delta(q_i, w_{ij}) = q_j \) for every \( i, j \in [n] \). Note that \( w_{ii} \) is the empty string, for every \( i \in [n] \). We call the set \( V(q_i, q') = \{ q'_{ij} \mid q_{ij} = \delta'(q', w_{ij}), j \in [n] \} \) the vicinity of \((q_i, q') \) in \( Q \times Q' \).

We say that \( V(q_i, q') \) is \textit{safe} if \((V(q_i, q') \times \Sigma) \cap X = \emptyset \). Otherwise, \( V(q_i, q') \) is called \textit{unsafe}.

\textbf{Remark 5.} For every prefix \( w \) of \( w_{ij} \), there is some \( k \in [n] \) such that \( w = w_{ik} \) and hence \( \delta'(q', w) = q'_{ik} \in V(q_i, q') \).

We now observe a crucial fact about safe vicinities, namely that, all the states in a safe vicinity are similar to corresponding states in the specification. This is the content of the next lemma which is proved before the main result of this section.

\textbf{Lemma 5.} Let \((q_i, q') \in Q \times Q' \) such that \( \pi(q') = q_i \) and \( V(q_i, q') = \{ q'_{ij} \mid j \in [n] \} \) is safe. Then for every \( j \in [n] \), \( \pi(q'_{ij}) = q_i \).

\textbf{Proof:} The proof is by induction on \( |w_{ij}| \), the length of \( w_{ij} \) (recall that \( \delta'(q', w_{ij}) = q'_{ij} \)). The base case, when \( |w_{ij}| = 0 \), is trivially true since in this case \( j = i \) and it is given that \( \pi(q'_{ij}) = \pi(q') = q_i \).

Suppose \( w_{ij} = w \cdot a \) for some \( w \in \Sigma^* \) and \( a \in \Sigma \). By Remark 5, there is some \( k \in [n] \) such that \( w = w_{ik} \) and \( \delta'(q', w) = q'_{ik} \). By the inductive hypothesis, \( \pi(q'_{ik}) = q_k \). Since \( V(q_i, q') \) is safe, it follows that \( \pi(\delta'(q'_{ik}, a)) = \delta(q_k, a) \), i.e., \( \pi(q'_{ij}) = q_j \). The proof is now complete by induction. \( \square \)
Cuts in the graph $G_I$ and the distance of $I$ from $S$ can be related, and cuts play a crucial role in our algorithms. This relationship is formally stated in the next lemma, which is the main result of this section.

**Lemma 6.** Suppose $I$ is at distance $r$ from $S$ and $(q_i, q'_i) \in Q \times Q'$ is a configuration such that $\pi(q'_i) = q_i$ and $V(q_i, q'_i)$ is safe. Then, for every $(V(q_i, q'_i), X)$-cut $C$ in $I$, $|C| \geq r$.

**Proof:** Let $Q'' = Q'$. Define $\delta'' : Q'' \times \Sigma \to Q''$ and $\lambda'' : Q'' \times \Sigma \to \Omega$ as follows:

$$
\delta''(q', a) = \begin{cases} 
\delta(q', a) & \text{if } (q, a) \notin C \text{ or } \pi(q') = \bot \\
d_{ij} & \text{otherwise, where } \delta(\pi(q'), a) = q_j
\end{cases}
$$

$$
\lambda''(q', a) = \begin{cases} 
\lambda(q', a) & \text{if } (q', a) \notin C \text{ or } \pi(q') = \bot \\
\lambda(\pi(q'), a) & \text{otherwise}
\end{cases}
$$

Let $J = (Q'', \delta'', \lambda'')$. We prove that $J$ is equivalent to $S$ by showing that $q' \simeq q_i$. In particular, we claim that for every $w \in \Sigma^*$, $\pi(\delta''(q', w)) = \delta(q_i, w)$, and $\lambda''(q', w) = \lambda(q_i, w)$. The proof is by induction on $|w|$, the length of $w$.

The base case, when $|w| = 0$, is trivially true since $\pi(q') = q_i$. Suppose $w = w' \cdot a$ for some $w' \in \Sigma^*$ and $a \in \Sigma$. Let $q'' = \delta''(q', w')$ and $q = \delta(q_i, w')$.

By the inductive hypothesis, we have $\pi(q'') = q$ and $\lambda''(q', w') = \lambda(q_i, w')$.

Thus, to prove the inductive step, we need to show that $\pi(\delta''(q', a)) = \delta(q, a)$ and $\lambda''(q', a) = \lambda(q, a)$. There are two cases:

**Case (q', a) \in C:** Since $\pi(q') = q \neq \bot$, we have $\delta''(q', a) = d_{ij}$, where $j$ is such that $\delta(q, a) = q_j$ and $\lambda''(q', a) = \lambda(q, a)$. By Lemma 5, $\pi(d_{ij}) = q_j$ and hence $\pi(\delta''(q', a)) = \delta(q, a)$ and $\lambda''(q', a) = \lambda(q, a)$.

**Case (q', a) \notin C:** We first show that $(q', a) \notin X$. It suffices to show that for every $q_{ij} \in V(q_i, q')$ and every $w \in \Sigma^*$, the path $(q_{ij}, w)$ in $J$ does not contain an edge in $X$. Suppose, for the sake of contradiction, that there is a path $p = (q_{i,j}, a_1 \ldots a_k)$ in $J$ which corresponds to the sequence $((q_{i,j}, a_1), \ldots, (q_k, a_k)) \in (Q' \times \Sigma)^k$ such that:

1. $q_i = q_{i,j} \in V(q_i, q')$ and $d_m \notin V(q_i, q') \forall 2 \leq m \leq k$
2. $(d_k, a_k) \in X$ and $(d_m, a_m) \notin X \forall 1 \leq m < k$

We show by induction that for all $m \in [k]$, $\pi(d_m) \neq \bot$ and $(d_{m+1}, a_{m+1}) \notin C$ (when $m > 1$). The base case ($m = 1$) is true since $\pi(d_1) = \pi(d_{ij}) = q_i$ by Lemma 5. For the inductive step, suppose that $\pi(d_m) \neq \bot$. Since $d_{m+1} \notin V(q_i, q')$, it follows from the definition of $\delta''$ that $(d_m, a_m) \notin C$ and hence $\delta''(d_m, a_m) = \delta(d_m, a_m)$. Since $(d_m, a_m) \notin X$, $\pi(\delta''(d_m, a_m)) \neq \bot$. Further since $d_{m+1} = \delta''(d_m, a_m)$, we have $\pi(d_{m+1}) \neq \bot$ as desired.
Thus, the sequence $((q_1,a_1), \ldots, (q_k,a_k))$ in fact corresponds to a path $(q_1,a_1 \ldots a_k)$ in $\mathcal{I}$ such that $q_i \in V(q_i, q')$, $(q_m, a_m) \notin C$ for every $1 \leq m < k$ and $(q_k, a_k) \in X$. But this is impossible, since $C$ is a $(V(q_i, q'), X)$-cut in $\mathcal{I}$. Hence, $(q'', a) \notin X$.

Now, since $(q'', a) \notin C$, we have $\delta''(q'', a) = \delta'(q', a)$. Since $(q', a) \notin X$, we have $\pi(\delta'(q', a)) = \delta(q, a)$ and $\lambda''(q'', a) = \lambda(q, a)$. This proves the inductive step.

Hence $\mathcal{J}$ is equivalent to $\mathcal{S}$. But this implies that $\mathcal{I}$ is at distance at most $|C| < r$ from $\mathcal{S}$, a contradiction. Hence, $|C| \geq r$. □

**Corollary 1.** Suppose $\mathcal{I}$ is at distance $r$ from $\mathcal{S}$ and $(q_i, q') \in Q \times Q'$ is a configuration such that $\pi(q') = q_i$ and $V(q, q')$ is safe. Then $|X| \geq r$ and $|V(q_i, q') \times \Sigma| \geq r$

**Proof:** Clearly, the sets $X$ and $(V(q_i, q') \times \Sigma)$ are both $(V(q_i, q'), X)$-cuts in $\mathcal{I}$, so Lemma 6 applies. □

### 3.4 Testing One Black Box

Suppose we are given a specific black box implementation machine $\mathcal{I}$ which we want test for conformance with specification $\mathcal{S}$. If $\mathcal{I}$ is subjected to a deterministic experiment, then unless the test sequence $w$ is a $(r, \Delta)$-approximate checking sequence, we cannot say $\mathcal{I}$ has less than $r$ errors if $\mathcal{I}$ passes the test. This is because if $w$ misses even a single FSM with more than $r$ faults, $\mathcal{I}$ could potentially be this FSM. Thus, for deterministic algorithms, testing a single implementation $\mathcal{I}$ does not differ from testing all possible machines. However, if we allow the fault detection experiment to be randomized then testing a single machine to achieve a certain level of confidence may require a shorter test. The algorithms we present in this section produce tests for a single black box. These constructions will be later used in Section 3.6 to obtain $(r, \Delta)$-approximate checking sequences. In this section we first obtain an algorithm for the case when $r$ is large, i.e., $r > d \min(n, \Delta)$, where $d = |\Sigma|$, and then consider the case when $r \leq d \min(n, \Delta)$.

#### 3.4.1 Algorithm for $r > d \min(n, \Delta)$

The algorithm for constructing the $(r, \Delta)$-approximate checking sequence is given in Figure 3.2. The algorithm is essentially similar to the one proposed by Yannakakis and Lee [70] and simply tests transitions in the specification at random.

**Lemma 7.** If $\mathcal{I}$ has at most $n + \Delta$ states and is at distance $r > d \min(n, \Delta)$ from $\mathcal{S}$ then, in each iteration of algorithm in Figure 3.2, the probability that the outputs of $\mathcal{S}$ and $\mathcal{I}$ differ is at least $\frac{1}{2amz}$ where $z = \max_{q \in Q} |Z_q|$.
**Input:** A FSM $S$ initially in state $q_i$, a separating family $\{Z_q\}$ of sequences for the states of $S$, and a black box FSM $I$.

**Output:** A random test sequence to be applied to $S$ and $I$.

**Method:** Repeat $k$ times: Apply one of the following 2 tests uniformly at random (u.a.r.)

**Test 1:** If $S$ is currently in state $q_i$, choose a separating sequence $w$ from $Z_i$ u.a.r. and apply it to $S$ and $I$.

**Test 2:** If $S$ is currently in state $q_i$, choose $q_j \in Q$ u.a.r. and $a \in \Sigma$ u.a.r. If $\delta(q_j,a) = q'$, choose a separating sequence $w$ from $Z_{q'}$ u.a.r. Apply the sequence $w_a w$ to $S$ and $I$.

**Figure 3.2.** Algorithm for testing a specific Black-Box when $r > d \min(n, \Delta)$

**Proof:** Let the configuration at the start of any iteration be $(q_i, q')$. There are two possible cases:

**Case $\pi(q') \neq q$:** In this case test 1 can detect an error, and the probability that an error is discovered in the current iteration is at least $\frac{1}{|Z_q|} \geq \frac{1}{2d}$.

**Case $\pi(q') = q$:** We claim that $V(q_i, q')$ is unsafe. Suppose, for the sake of contradiction, that $V(q_i, q')$ is safe. We have $|V(q_i, q')| = n$. By Corollary 1, $|V(q_i, q') \times \Sigma| = dn \geq r$. Now if $n \leq \Delta$, we obtain a contradiction since $r > d \min(n, \Delta)$. Therefore suppose that $\Delta < n$ and hence $r > d \Delta$. By Corollary 1, we have $|X| \geq r > d \Delta$. Also, the sets $(V(q_i, q') \times \Sigma)$ and $X$ are disjoint, since $V(q_i, q')$ is safe by assumption. Hence, $|Q' \times \Sigma| \geq |V(q_i, q') \times \Sigma| + |X| > nd + d \Delta = (n + \Delta)d$. Hence, $|Q'| > n + \Delta$, which contradicts the fact that $I$ has at most $n + \Delta$ states.

Hence, $V(q_i, q')$ is unsafe. In this case test 2 can detect an error, and the probability that an error is discovered in the current iteration is at least $\frac{1}{ndz}$. Hence, the lemma holds.

**Theorem 5.** Let $I$ be a FSM with at most $n + \Delta$ states and which is at distance at least $r$ from $S$. For any $\epsilon > 0$, after $k = 2 \log(\frac{1}{\epsilon})dnz$ iterations, the probability that the algorithm in Figure 3.2 detects that $I$ is faulty is at least $1 - \epsilon$.

The theorem follows from Lemma 7 and a standard application of Chernoff bounds [54].

**3.4.2 Algorithm for $r \leq d \min(n, \Delta)$**

We now consider the case when $r \leq d \min(n, \Delta)$, where $d = |\Sigma|$. When $r$ is $o(d)$, let $\rho = r$ and $\lambda = 1 + \Delta$, and when there is a constant $c > 1$ such that $r \geq \frac{c}{\lambda}$, let $\rho = r(1 - \frac{1}{c})$ and $\lambda = 1 + \frac{c\Delta}{r}$. The algorithm for testing a single black-box is given in Figure 3.3.

**Lemma 8.** If $I$ has at most $n + \Delta$ states and is at distance $r$ from $S$ then, in each iteration of the algorithm in Figure 3.3, the probability that the outputs of $S$ and $I$ differ is at least $\frac{1}{sdz} \min \left\{ 1, \frac{r}{\lambda - 1} \left( \frac{\rho}{1 + \Delta} \right)^{\lambda - 1} \right\}$, where $z = \max_{q \in Q} |Z_q|

**Proof:** Let the configuration at the start of any iteration be $(q_i, q')$. There are three possible cases:
Input: A FSM $S$ initially in state $q_1$, a separating family $\{Z_q\}$ of sequences for the states of $S$, and a black box FSM $T$.

Output: A random test sequence to be applied to $S$ and $T$.

Method: Repeat $k$ times: Apply one of the following 3 tests uniformly at random (u.a.r.)

Test 1: If $S$ is currently in state $q_i$, choose a separating sequence $w$ from $Z_{q_i}$ u.a.r. and apply it to $S$ and $T$.

Test 2: If $S$ is currently in state $q_i$, choose $q_j \in Q$ u.a.r. and $a \in \Sigma$ u.a.r. If $\delta_S(q_j, a) = q'$, choose a separating sequence $w$ from $Z_{q'}$ u.a.r. Apply the sequence $w_{q_i}aw$ to $S$ and $T$.

Test 3: If $S$ is currently in state $q_i$, choose $q_j \in Q$ u.a.r., $l \in \{2, \ldots, \lambda\}$ u.a.r. and $w \in \Sigma^l$ u.a.r. If $\delta(q_j, w) = q'$, choose a separating sequence $w'$ from $Z_{q'}$ u.a.r. Apply the sequence $w_{q_i}wuw'$ to $S$ and $T$.

Figure 3.3. Algorithm for testing a specific black-box when $r \leq \min(n, \Delta)$

Case $\pi(q') \neq q_i$: In this case test 1 can detect an error, and the probability that an error is discovered in the current iteration is at least $\frac{1}{2} \left[ \frac{1}{\log(l-1)} \right] = \frac{1}{\log(l-1)}$.

Case $V(q_i, q')$ is not safe: Then, there is a state $q_j$ (possibly $q_j = q_i$) such that for some $a \in \Sigma$, $\pi(\delta(q', w_{q_j}a)) \neq \delta(q_i, w_{q_j}a)$ or $\lambda'(q', w_{q_j}a) \neq \lambda(q_i, w_{q_j}a)$. In this case test 2 can detect an error, and the probability that an error is discovered in the current iteration is at least $\frac{1}{2} \left[ \frac{1}{\log(l-1)} \right] = \frac{1}{\log(l-1)}$.

Case $\pi(q') = q_i$ and $V(q_i, q')$ is safe: Note that choosing $w_{q_j}$ and $w$ as described corresponds to a random walk of length $|w|$ in $G_S$ starting from $V(q_i, q')$.

Since $|V(q_i, q')| = n$ and the size of the $(V(q_i, q'), X)$-cut in $T$ (and hence in $G_T$) is at least $r$, it follows from Lemma 4, the probability that test 3 discovers an error is at least $\frac{1}{2} \left[ \frac{1}{\log(l-1)} \right] = \frac{1}{\log(l-1)}$. Hence, the lemma holds.

Theorem 6. Let $T$ be a FSM with at most $n + \Delta$ states and which is at distance at least $r$ from $S$. For any $\epsilon > 0$, the algorithm in Figure 3.3 detects that $T$ is faulty with probability at least $1 - \epsilon$ after $k = 3dnz \log(\frac{1}{\epsilon}) \max\left\{1, \frac{1}{2} \left[ \frac{1}{\log(l-1)} \right] \right\}$

Theorem follows from the observations in Lemma 8. The length of the test sequence output by the algorithm is

$$O\left( dnz(n + \Delta) \max\left\{1, \frac{1}{2} \left[ \frac{1}{\log(l-1)} \right] \right\} \right)$$

when $r$ is $o(d)$

$$O\left( dnz(n + \Delta) \max\left\{1, \frac{1}{2} \left[ \frac{1}{\log(l-1)} \right] \right\} \right)$$

otherwise

3.5 Lower Bounds

In this section, we present lower bounds for the problem of checking if a single black box conforms to the specification or has multiple faults.

Theorem 7. For every $n$, $d > 5$, there is a specification FSM $S$ with $d$ inputs and $n$ states such that for every randomized test of length $L(n, \Delta, r)$ there is FSM $T$ with at most $n + \Delta$ states at distance at least $r$ from $S$ that will pass the
test with probability at least $1 - \epsilon$. The value of $L(n, \Delta, r)$ for different values of $n$, $\Delta$ and $r$ are given as follows.

1. If $\Delta = 1$ and $r > d\Delta$, $L(n, \Delta, r) = \epsilon \frac{(d-5)(n+4)n^2}{32}$

2. If there is a constant $c > 1$ such that \( \frac{d-3}{c} \leq r \leq \frac{d-5}{c} \Delta \), then $L(n, \Delta, r) = \epsilon(n-1)^{\frac{d-3}{c} - \frac{d-5}{c} \Delta}$

3. If $r = o(d)$ then $L(n, \Delta, r) = \epsilon(n-1)^{\Delta \left( \frac{d-3}{c} \right)^{1+\Delta}}$

**Proof:** The lower bounds are proved by generalizing the “combinatorial lock” construction used by Vasilevskii [68] to construct the specification machine and a family of implementation machines that defeat short tests. The proof is divided into two parts. The first part provides bounds for the first case in Theorem 7 while the second part provides the bounds for the other two cases.

**Case when $r > d\Delta$ and $\Delta = 1$**

Let $\Sigma = \{0, 1, 2, a_1, a_2, \ldots, a_{d-3}\}$. Let $\Omega = \{0, 1\}$. Let $\Delta = 1$ and $r = d\Delta + 1$.

We will describe an $n$-state minimal FSM $S$ and a family $I$ of at most $n + \Delta$-state (in fact exactly $n$-state) FSMs such that:

1. $S$ has reset symbol 0, and each $I \in I$ implements this reset reliably

2. each $I \in I$ is at distance $d + 1$ from $S$ (i.e., $r > d\Delta$ where $\Delta = 1$)

3. for every randomized test sequence $C$ which has length $\epsilon \frac{(d-5)(n+4)n^2}{32}$, $\exists I \in I$ that passes the test with probability at least $1 - \epsilon$.

Let $S = (Q, \delta, \lambda)$ be the FSM shown in Figure 3.4. Formally, $Q = \{q_1, \ldots, q_n\}$; on input 0, every state makes a transition to $q_1$; on input 1 $q_1$ makes a transition to $q_2$, on input 2 $q_1$ makes a transition to $q_{2+1}$, on every input $\sigma \neq 0, 1, 2$ $q_1$ makes a transition to $q_n$; for every $i = 2, \ldots, \frac{n}{2} - 1$, on input 1 $q_i$ and $q_{2+i-1}$ make transitions to $q_{i+1}$ and $q_{2+i}$ respectively, and on input $\sigma \neq 0, 1, q_i$ and $q_{2+i-1}$ both make transitions to $q_n$; on input $\sigma \neq 0$ $q_{2+i-1}$, $q_{n-1}$ and $q_n$ all make transitions to $q_n$. All outputs are 0 except for the transitions $\lambda(q_{2+i}, 2) = 1$ and $\lambda(q_{n-1}, 2) = 1$. Let $X'$ be the set of all subsets of $\{a_1, \ldots, a_{d-3}\}$ of size $\frac{d-1}{2}$.

For every $X \in X'$ let $\overline{X} = \{2\} \cup \{a_k \in \Sigma \mid a_k \notin X\}$. For every $2 \leq i, j \leq \frac{n}{2}$ and $X \in X'$ let $I(i, j, X)$ be the FSM that is shown in Figure 3.5. It is identical to $S$ in every respect except for some transitions out of $q_i$ and $q_{2+i-1}$. On every input $\sigma \in X$, $q_i$ makes a transition to $q_j$, and on every input $\sigma \in \overline{X}$, $q_i$ makes a transition to $q_n$. Similarly, on every input $\sigma \in X$, $q_{2+i-1}$ makes a transition to $q_{2+i-1}$, and on every input $\sigma \in \overline{X}$, $q_{2+i-1}$ makes a transition to $q_n$. For every $2 \leq i, j \leq \frac{n}{2}$ and every $X \in \mathcal{X}$ let $U(i, j, X)$ be defined as the set of all strings of the form

- $01^{(i-1)x1(n/2-j)0}2$ and
- $021^{(i-2)x1(n/2-j)0}$ where $x \in X$, if $i < j$
\[ 01^{(i-1)}(x1^{i-j})^*y1^{(n/2-j)2} \text{ and} \]
\[ 021^{(i-2)}(x1^{i-j})^*y1^{(n/2-j)2} \] where \( x, y \in X \), if \( i \geq j \)

Note that for any \((i_1, j_1, X_1) \neq (i_2, j_2, X_2)\), no two strings in \( U(i_1, j_1, X_1) \) and \( U(i_2, j_2, X_2) \) overlap. The following lemma is an easy consequence of the definitions of \( S \) and \( I \).

**Lemma 9.** For every \( 2 \leq i, j \leq \frac{n}{2} \) and \( X \in X \), if a test sequence \( C \in \Sigma^* \) does not contain a sequence from \( U(i, j, X) \), then the FSM \( I(i, j, X) \) will pass the test \( C \).

Let \( C \in \Sigma^* \). Consider any \( 2 \leq i, j \leq \frac{n}{2} \). Let \( U(i, j) = \bigcup_{X \in \mathcal{X}} U(i, j, X) \). Note that the length of the shortest string in \( U(i, j) \) is \( \frac{n}{2} + 2 + i - j \). If \( C \) does not contain at least \( (d - 3) - \frac{d+1}{2} + 1 = \frac{d-5}{2} \) distinct strings from \( U(i, j) \), then there is some \( X \in \mathcal{X} \) such that \( C \) does not contain any string from \( U(i, j, X) \), and hence \( I(i, j, X) \) will pass the test \( C \). If every FSM \( I(i, j, X) \) passes the test \( C \) with probability at least \( \epsilon \), then the expected length of \( C \) is at least

\[
\epsilon \frac{(d - 5)(n + 4)n^2}{2} \sum_{i=1}^{\frac{n}{2}} \sum_{j=1}^{\frac{n}{2}} \left( \frac{n}{2} + 2 + i - j \right) = \epsilon \frac{(d - 5)(n + 4)n^2}{32} \]
Case when $r \leq d \min\{n, \Delta\}$

Let $\Sigma = \{0, 1, a_1, a_2, \ldots, a_{d-3}\}$. Let $\Omega = \{0, 1\}$. Let

$$L = \begin{cases} e(n-1) \frac{(d-3)\Delta}{r} 2\Delta \log \frac{d\Delta}{\epsilon} & \text{when } r \text{ is } o(d) \\ e(n-1) \frac{(d-3)\Delta}{r} \left( \frac{d\Delta}{\epsilon} \right)^c & \text{when } \frac{d}{r} \leq \frac{d\Delta}{\epsilon}, c > 1 \end{cases}$$

We will describe an $n$-state minimal FSM $S$ and a family $I$ of $n + \Delta$-state FSMS such that:

1. $S$ has reset symbol 0, and each $I \in I$ implements this reset reliably
2. each $I \in I$ is at distance $r$ from $S$ ($r \leq d\Delta$)
3. for every randomized test sequence $C$ of length $L$, $\exists I \in I$ that passes the test with probability at least $1 - \epsilon$.

Let $S = (Q, \delta, \lambda)$ be the machine shown in Figure 3.7. Here $Q = \{q_1, \ldots, q_n\}$; on an input $\sigma \neq 0, 1$ every state makes a transition to $q_n$. On input 1, every state $q_i$, for $i \neq n$, moves to the next state $q_{i+1}$, and $q_n$ moves to itself. On input 0 (reset), all states move to $q_1$. All outputs are 0, except for the transition $\lambda(q_{n-1}, 2) = 1$.

Let $m$ be a positive integer such that $1 \leq m \leq r$. Partition the set $\{a_1, \ldots, a_{d-3}\}$ into sets of size $\frac{d\Delta}{m}$ as follows: for each $i = 1, 2, \ldots, (d-3)\frac{d\Delta}{r}$ let $A_i = \{a_{(i-1)\frac{d\Delta}{m}+1}, \ldots, a_{i\frac{d\Delta}{m}}\}$. 
Let $A = \{A_i\}_i$. Let $k = \frac{\lambda}{m}$.
For each $\mathbf{X} = (X_1, X_2, \ldots, X_{k+1}) \in A^{k+1}$ we define

$$\Sigma_{\mathbf{X}} = \{w = \sigma_1 \ldots \sigma_{k+1} \in \Sigma^{k+1} | \sigma_i \in X_i \ \forall i = 1, \ldots, k+1\}$$

For each $j = 1, \ldots, n - 1$ and each $\mathbf{X} = (X_1, \ldots, X_{k+1}) \in A^{k+1}$ we define the FSM $\mathcal{I}(j, \mathbf{X}) = (Q', \sigma', \lambda')$ (shown in Figure 3.6) as follows:

- $Q' = Q \cup (\bigcup_{i=1}^{m} \{q_{i1}, q_{i2}, \ldots, q_{ii}\})$
- Every state makes a transition to $q_1$ on input 0 (reset).
- On input 1, every state $q_i$, $i \neq n$, moves to $q_{i+1}$ and $q_n$ moves to itself.
- For $i > m$, on every input $\sigma \neq 0, 1$, $q_i$ moves to $q_n$
- For $i = 1, \ldots, m$, on input $\sigma \neq 0, 1$, $q_i$ moves to $q_{i1}$ if $\sigma \in X_1$, and to $q_n$ otherwise.
- For $i = 1, \ldots, m$; $j = 1, \ldots, k - 1$, on input $\sigma \neq 0$ $q_{ij}$ moves to $q_{ij+1}$ if $\sigma \in X_{j+1}$, and to $q_n$ otherwise.
- For $i = 1, \ldots, m$, on input $\sigma \neq 0$ $q_{ik}$ moves to $q_j$ if $\sigma \in X_{k+1}$, and to $q_n$ otherwise.
- All outputs are 0, except for the transition $\lambda'(q_{n-1}, 2) = 1$.

For every $j = 1, \ldots, m$ and every $\mathbf{X} \in A^{k+1}$ let $U(j, \mathbf{X})$ be the union of all sets of strings of the following form: $0^{i_1}(w1^{i_2})^{*}w1^{n-1-j}2$, where $w \in \Sigma_{\mathbf{X}}$, $0 \leq i_1, i_2 \leq m - 1$ and $i_2 + j \leq m$.

For every $j = m + 1, \ldots, n - 1$ and every $\mathbf{X} \in A^{k+1}$ let $U(j, \mathbf{X})$ be the union of all sets of strings of the following form: $0^{i}w1^{n-1-j}2$, where $0 \leq i \leq m - 1$.

**Lemma 10.** For any $j = 1, \ldots, n - 1$ and $\mathbf{X} \in A^{k+1}$, if a test sequence $C$ does not contain a sequence from $U(j, \mathbf{X})$, then the FSM $\mathcal{I}(j, \mathbf{X})$ will pass the test $C$.

The shortest string in $U(j, \mathbf{X})$ has length at least $n + k - j + 2 > k$. Also, note that for distinct pairs $(j_1, \mathbf{X}_1)$, $(j_2, \mathbf{X}_2)$, no sequence in $U(j_1, \mathbf{X}_1)$ overlaps with a sequence in $U(j_2, \mathbf{X}_2)$. Let $F_C = \{ (j, \mathbf{X}) | \mathcal{I}(j, \mathbf{X}) \text{ fails the test} \}$. Therefore, $|C| > k|F_C|$. If every FSM $\mathcal{I}(j, \mathbf{X})$ fails the randomized test with probability at least $\epsilon$, then the expected length of the test is

$$\sum_C \text{Prob}[C]|C| \geq \sum_C \text{Prob}[C]|F_C|$$

$$= k \sum_{j, \mathbf{X}} \text{Prob}[\mathcal{I}(j, \mathbf{X}) \text{ fails the test}]$$

$$\geq \epsilon k(n - 1)|A^{k+1}|$$

$$= \epsilon k(n - 1) \left( \frac{(d-3)\Delta}{p_d} \right)^{k+1}$$
When there is a constant $c > 1$ such that $\frac{d-3}{cr} \leq r \leq \frac{(d-3)\Delta}{cr}$, we choose $k = \frac{(d-3)\Delta}{cr}$ and we get

$$|C| > \epsilon(n-1) \frac{(d-3)\Delta}{cr} e^{1 + \frac{(d-3)\Delta}{r}}$$

When $r$ is $o(d)$, we choose $k = \Delta$ and we get

$$|C| > \epsilon(n-1) \Delta \left( \frac{d-3}{r} \right)^{1+\Delta}$$

This completes the proof of the theorem. \(\square\)

### 3.6 Checking sequences

By running our algorithms for more iterations, we can obtain a randomized construction of an $(r, \Delta)$-approximate checking sequence. Let $N$ be the number of faulty machines, i.e., the number of FSMs with at most $n + \Delta$ states that are not equivalent to $S$. Clearly $N$ is at most $\Delta(|\Omega|(n + \Delta))^d(n+\Delta)$, since this is the total number of FSMs with at most $n + \Delta$ states. Note that $|\Omega| \leq d(n + \Delta)$ since this is the maximum number of transitions possible in any FSM with at most $n + \Delta$ states.

For any $\theta > 0$, if we let $\epsilon = \theta / N$ in Theorem 5 and Theorem 6, the probability that the resulting sequence detects all faulty machines is at least $1 - \theta$, yielding an $(r, \Delta)$-approximate checking sequence. We thus obtain the following theorems:

**Theorem 8.** When $r > d \min(n, \Delta)$, for any $\theta > 0$, taking $k = O(\left( \frac{d^2 \log n \max(1, m(d, r))}{\max(1, m(d, r))} \right))$, the algorithm in Figure 3.2 generates a sequence of length $O(\left( \frac{d^2 n \log(\frac{|\Omega|(n + \Delta)}{r})}{\max(1, m(d, r))} \right))$ that is an $(r, \Delta)$-approximate checking sequence with probability at least $1 - \theta$.

**Theorem 9.** When $r \leq d \min(n, \Delta)$, for any $\theta > 0$ the algorithm in Figure 3.3 generates a sequence of length $O(\left( \frac{d^2 n \log(\frac{|\Omega|(n + \Delta)}{r}) \max(1, m(d, r))}{\max(1, m(d, r))} \right))$ that is a $(r, \Delta)$-approximate checking sequence with probability at least $1 - \theta$, where

$$m(d, \Delta, r) = \begin{cases} 2^{O((\Delta \log \frac{\Delta}{r})^2)} & \text{if } r = o(d) \\ \frac{1}{\Delta} 2^{O((\Delta \log \Delta)^2)} & \text{otherwise} \end{cases}$$

### 3.7 Discussion

In this section we determine bounds on the number of extra states in the implementation that can be tolerated, and still obtain polynomially long approximate checking sequences. Based on the lower bounds derived in Section 3.5, we determine the size of $\Delta$ that will force any checking sequence to be super-polynomial. The results from Section 3.6, help us obtain upper bounds on $\Delta$ for which our algorithms can construct polynomially long approximate checking sequence. In
what follows, we consider the upper and lower bounds on $\Delta$ in three cases, depending on the value of $r$; $L(n, \Delta)$ is used to denote the length of a checking sequence.

**Case** $r > d \min(n, \Delta)$: Both the lower bound and the upper bound on the length of checking sequences are polynomial in $n + \Delta$ for every $\Delta$.

**Case** $r$ is $o(d)$: If $\Delta$ is $\omega\left(\frac{\log n}{\log(1/r)}\right)$, then the lower bound on $L(n, \Delta)$ is super-polynomial. We can guarantee a polynomial upper bound on $L(n, \Delta)$ when $\Delta$ is $O\left(\frac{\log n}{\log \log n + \log (d/r)}\right)$. Thus, our guarantee is best-possible when $r$ is $O\left(\frac{d}{\log n}\right)$, and worse by a factor of at most $\log \log n$ otherwise.

**Case** $\frac{d}{e} \leq r \leq d \min(n, \Delta)$ for some constant $e > 1$: If $\Delta$ is $\omega\left(\frac{r}{e} \log n\right)$, then the lower bound on $L(n, \Delta)$ is super-polynomial. We can guarantee a polynomial upper bound on $L(n, \Delta)$ when $\Delta$ is $O\left(\frac{r}{d \log \log n + \log (r/d)}\right)$. Thus, our guarantee is worse than best-possible by at least a factor of $\log \log n$ and at most a factor of $\log n$ (since $r \leq d \min(n, \Delta)$).

These observations demonstrate that our upper bounds and lower bounds are only separated in the case when $r$ is small, and even in this case the separation is at most by a factor of $\log n$. Trying to close this gap is an interesting open problem.
Chapter 4

Congruences for Visibly Pushdown Languages

4.1 Introduction

In this chapter, we lay the necessary foundations in order to tackle the conformance testing problem for the visibly pushdown automata (VPA) program model in Chapter 5. Notice that the algorithms presented for FSMs (Figure 3.2 and Figure 3.3) exploit the existence of a separating family $Z_q$ of sequences that allows states of the machine to be distinguished. The existence of separating families for canonical (minimum-state) FSMs is a consequence of the Myhill-Nerode congruence-based characterization of regular languages. In order to extend these algorithms to VPAs, it is therefore important to come up with parallel notions of canonical VPAs, and to obtain language-theoretic characterizations for VPLs. The purpose of this chapter, therefore, is to address these issues.

Our main result in this chapter is a characterization of the class of VPLs in terms of congruences on strings. It is well known that the syntactic congruence, which is defined as $w_1 \approx w_2$ when for every $u,v$, $uw_1v \in L$ if and only if $uw_2v \in L$, has finite index precisely for languages $L$ that are regular. We show that for VPLs $L$, when we restrict our attention to well-matched words $w_1$ and $w_2$ (i.e., words where every push transition has a corresponding pop transition and vice versa), the syntactic congruence has finite index. Moreover, for languages consisting only of well-matched words, if the syntactic congruence on well-matched words has finite index then the language is a VPL. For languages containing strings that are not well-matched, we need some additional conditions only because no congruence on well-matched words can saturate such a language. Our characterization of VPLs is a natural generalization of the Myhill-Nerode theorem for regular languages—when restricted to languages that do not require any push or pop operations, our congruence coincides with the right congruence defined by Myhill and Nerode [56, 36].

One important consequence of the congruence based characterization of regular (word) languages and regular tree languages is that for any regular language there is a unique minimum state deterministic automaton recognizing the language, which can also be constructed efficiently [36, 37]. For VPLs, however, we show that in general there is no unique minimum state recognizer. Thus, while
our characterization yields the construction of a canonical deterministic acceptor for VPLs, it may not in general be minimal. An implicit consequence of the results in [10] is that VPLs have canonical deterministic pushdown automata. It is shown in [10] that with any language $L$, a language of trees called stack trees, can be associated such that $L$ is a VPL exactly when the corresponding set of stack trees form a regular tree language. The unique minimal bottom-up tree automaton accepting the language of stack trees can then be translated to a canonical deterministic visibly pushdown automaton. However, since bottom-up tree automata can only be translated into deterministic pushdown automata with exponentially more states, the implicit construction in [10] does not result in necessarily small deterministic VPAs.

Visibly pushdown automata are a natural model for programs with recursive procedure calls and finite data types. Such programs are called Boolean programs in the literature on software model checking [12]. When modeling a program as a visibly pushdown automaton, the natural structure the model assumes is one where the machine’s states are partitioned into $k$ modules, one for each procedure in the program. As one expects, these modules are such that from a state in a module, a well-matched sequence of calls and returns to other modules results in a state of the same module. Moreover, if the programs modeled are such that the calls to modules have no input parameters (or if a function is modeled separately for each possible value of its input parameters), then the visibly pushdown automaton assumes additional structure, namely that every call results in going to a unique state in the module corresponding to the call. We call such structured VPAs $k$-module single-entry VPAs ($k$-SEVPAs). They correspond to the model of recursive state machines with a single entry per module [6].

Though visibly pushdown languages in general do not have unique minimum-state recognizers, partitioning the calls into the modules they correspond to fixes enough additional structure that there is a minimum-state $k$-SEVPA that respects the partition and accepts the language. More precisely, we show that for any partition of the call-alphabet into $k$-sets, there is a unique minimum-state $k$-SEVPA accepting any well-matched VPL $L$. If $k = 0$ (that is, there are no calls), the result is equivalent to the Myhill-Nerode theorem for regular languages. The characterization of this unique minimal $k$-SEVPA is done via a set of $k+1$ congruences on words. We also present an algorithm which, given any deterministic $k$-SEVPA accepting a well-matched language, minimizes it in $O(n^3)$ time, where $n$ is the size of the original machine.

The rest of the chapter is organized as follows. Our main result characterizing visibly pushdown languages in terms of language theoretic congruences is presented in Section 4.2. We also show that VPLs, in general, do not have unique minimum state deterministic recognizers. In Section 4.3, we define the notion of how partitions on calls define $k$-module single-entry VPAs and prove that every (well-matched) VPL has a unique minimal $k$-SEVPA accepting it.
We also present an example of a family of languages for which the minimal
1-module machine is super polynomial in the size of the smallest visibly push-
down automaton recognizing it. Conclusions and open problems are presented
in Section 5.5.

4.2 Congruence based characterization of VPLS

In this section we present a congruence based characterization of when a lan-
guage over \( \hat{\Sigma} \) is a visibly pushdown language. Before presenting the charac-
terization for general VPLS, we first consider the case of VPLS that have only
well-matched words.

A note on notation

Let \( \hat{\Sigma} = (\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \Sigma_{\text{int}}) \) be a pushdown alphabet and let \( \Sigma = \Sigma_{\text{call}} \cup \Sigma_{\text{ret}} \cup \Sigma_{\text{int}} \). We will use \( u, v, u_1, \ldots \) for strings in \( \Sigma^* \), \( c, c_1, c_i, \ldots \) for elements of \( \Sigma_{\text{call}} \),
\( r, r_1, r_i, \ldots \) for elements of \( \Sigma_{\text{ret}} \), and \( i, i_1, i_j, \ldots \) for elements of \( \Sigma_{\text{int}} \).

4.2.1 Well-matched visibly pushdown languages

For a language \( L \) over the pushdown alphabet \( \hat{\Sigma} = (\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \Sigma_{\text{int}}) \), consider
the following congruence on well-matched words:

\[
  w_1 \approx w_2 \iff \forall u, v \in \Sigma^*, \; uw_1v \in L \iff uw_2v \in L
\]

Recall that this is the standard syntactic congruence restricted to well-matched
words over \( \hat{\Sigma} \). For example, if \( \hat{\Sigma} = (\{c\}, \{r\}, \emptyset) \) and \( L = \{c^n r^n \mid n \geq 0\} \), then
there are only two equivalence classes that \( \approx \) defines: \( \{c^n r^n \mid n \geq 0\} \) and the
complement of this set with respect to \( WM(\hat{\Sigma}) \).

Analogous to the case of regular languages, the finiteness of the number of
equivalence classes of the syntactic congruence (on well-matched words) provides
a precise characterization of well-matched VPLS.

Theorem 10. \( L \) is a well-matched \( \hat{\Sigma} \)-VPL iff \( \approx \) (as defined above) has finitely
many equivalence classes.

Proof: Suppose \( L \) is a \( \hat{\Sigma} \)-VPL and \( M = (Q, q_0, \Gamma, \delta, Q_F) \) is a VPA over \( \hat{\Sigma} \)
with (unique) initial state \( q_0 \) such that \( L(M) = L \). Every well-matched string
\( w \) defines a function \( f_w : Q \rightarrow Q \) as follows: \( f_w(q) = \delta_\bot(q, w) \). Define the
following equivalence on well-matched strings:

\[
  w_1 \approx_M w_2 \iff f_{w_1} = f_{w_2}
\]

Observe that \( \approx_M \) has finitely many equivalence classes (bounded by \(|Q|^{|Q|}) \). We
will show that \( \approx_M \) is a refinement of \( \approx \), thus establishing that \( \approx \) is also of finite
index. Consider \( w_1 \approx_M w_2 \). Then for any \( u, v \in \Sigma^* \), we know
\[
\delta((q_0, \bot), uw_1v) = \delta(\delta((q_0, \bot), u), w_1), v) = \delta(\delta((q_0, \bot), u), w_2), v) \text{ since } f_{w_1} = f_{w_2} = \delta((q_0, \bot), uw_2v)
\]
Hence \( uw_1v \in L \) iff \( uw_2v \in L \), and so \( w_1 \approx w_2 \). Thus \( \approx_M \) is a refinement of \( \approx \). Observe that this proof does not rely on \( L \) being a well-matched language.

To prove the converse, consider a language \( L \) such that \( \approx \) is of finite index. We construct a deterministic (but incomplete\(^*\)) VPA that recognizes \( L \) and whose states are the equivalence classes of \( \approx \). Consider a string with no unmatched returns \( u = w_1c_1w_2c_2\cdots c_kw_{k+1} \in MR(\hat{\Sigma}) \), where \( c_1, \ldots, c_k \) are the unmatched call symbols in \( u \), and \( w_1, \ldots, w_{k+1} \) are well-matched strings between the unmatched call symbols. The automaton we construct will maintain the following invariant: after reading the string \( u \in MR(\hat{\Sigma}) \), the state of the machine will be \([w_{k+1}]_\approx\) and the stack will be \([(w_k]_\approx, c_k), \ldots, (\emptyset, c_1)\bot\).

The formal construction of VPA \( M = (Q, \emptyset, \Gamma, \delta, Q_F) \) is as follows: \( Q = \{[w]_\approx \mid w \in WM(\hat{\Sigma})\} \), \( \emptyset = \{\bot\} \cup (Q \times \Sigma_{call}) \), and \( Q_F = \{[w]_\approx \mid w \in L\} \).

The transition function \( \delta \) is defined as follows.

- \([w]_\approx \xrightarrow{\epsilon} [w]_\approx \) for every \( \epsilon \in \Sigma_{int} \)
- \([w]_\approx \xrightarrow{\epsilon / [w]_\approx, c} [c]_\approx \) for every \( c \in \Sigma_{call} \)
- \([w]_\approx \xrightarrow{r / [w]_\approx, c} [w']_\approx \) for every \( r \in \Sigma_{ret} \)

The above machine has no pop transitions when \( \bot \) is the only symbol on the stack. Observe that the definitions of \( Q_F \) and \( \delta \) are sound because \( \approx \) saturates \( L \) \(^1\) and \( \approx \) is a congruence with respect to well-matched words. Further, it is easy to verify that the above invariant is maintained. Thus, after reading a \textit{well-matched} word \( w \), the automaton will be in the state \([w]_\approx\) and hence \( L = L(M) \cap WM(\hat{\Sigma}) \). Since \( WM(\hat{\Sigma}) \) is a \( \text{VPL} \), and \( \text{VPLs} \) are closed under intersection, the result follows.

\( \Box \)

### 4.2.2 General visibly pushdown languages

For visibly pushdown languages that are not necessarily well-matched, \( \approx \) being of finite index is not sufficient. This is because \( \approx \) is no longer a congruence that saturates the \( \text{VPL} \). We need to define two additional congruences on strings—one that will capture the behavior of a state when the stack has only \( \bot \), and one that will capture the behavior when the stack has more than one element.

---

\(^*\) A VPA is incomplete if the transition function \( \delta \) is not total. An incomplete VPA can be easily modified to yield a VPA with at most one extra "dead" state to which all undefined transitions go.

\(^1\) We say that \( \approx \) saturates \( L \) iff either \([w]_\approx \cap L = \emptyset\) or \([w]_\approx \subseteq L\), for every equivalence class \([w]_\approx\). This property clearly holds when we choose both \( u \) and \( v \) to be the empty string; now for any two strings \( w_1, w_2 \in [w]_\approx \), \( w_1 \in L \) iff \( w_2 \in L \).
The reason we need to distinguish the cases of the stack having only \( \bot \) and that of the stack having additional elements, is because symbols in \( \Sigma_{rel} \) behave differently. In the first case, elements of \( \Sigma_{rel} \) are like internal actions which leave the stack unchanged, and in the second case they result in the stack being popped.

For a language \( L \) over \( \widehat{\Sigma} \), define the following congruences.

For \( u_1, u_2 \in \Sigma^* \), \( u_1 \equiv u_2 \) iff for all \( v \) in \( MR(\widehat{\Sigma}) \), \( u_1v \in L \) iff \( u_2v \in L \)

For \( u_1, u_2 \in MC(\widehat{\Sigma}), u_1 \sim_0 u_2 \) iff for all \( v \) in \( \Sigma^* \), \( u_1v \in L \) iff \( u_2v \in L \)

Intuitively, the congruence \( \equiv \) says that the two strings \( u_1 \) and \( u_2 \) cannot be distinguished by strings \( v \in MR(\widehat{\Sigma}) \) that do not examine the stacks reached on \( u_1 \) and \( u_2 \). The congruence \( \sim_0 \) is defined only on strings where every call is matched. Thus, after reading such a word, every VPA will only have \( \bot \) on the stack. Starting from such configurations, as was observed earlier, return symbols behave like internal actions, and the congruence is the usual Myhill-Nerode right congruence. We now present the main theorem of this chapter.

**Theorem 11.** \( L \) is a \( \widehat{\Sigma}-VPL \) iff \( \sim, \equiv \) and \( \sim_0 \) all have finite index.

**Proof:** For a \( VPL \) \( L \), let \( M = (Q, q_0, \Gamma, \delta, Q_F) \) be a VPA recognizing \( L \). In the proof of Theorem 10, we already showed that \( \sim \) has finite index. Define the following two equivalences over words in \( \Sigma^* \):

\[
\begin{align*}
u_1 \equiv^M u_2 \text{ iff } & \delta_\bot(q_0, u_1) = \delta_\bot(q_0, u_2) \\
u_1 \sim_0^M u_2 \text{ iff } & \delta_\bot(q_0, u_1) = \delta_\bot(q_0, u_2)
\end{align*}
\]

A similar argument to the one presented in Theorem 10 shows that \( \equiv^M \) refines \( \equiv \), and \( \sim_0^M \) refines \( \sim_0 \) when restricted to \( MC(\widehat{\Sigma}) \). Hence, both \( \equiv \) and \( \sim_0 \) have finitely many equivalence classes.

For the converse, we show that \( L \) is a \( VPL \) by once again constructing a VPA \( M \) whose states are equivalence classes of the congruences we have defined, but the construction is a bit more involved. The main intuition behind the construction is to ensure that the following invariant is maintained after \( M \) has read a string \( u \in \Sigma^* \):

- If \( u \in MC(\widehat{\Sigma}) \) then the state of \( M \) is \( [u]_{\sim_0} \) and the stack is \( \bot \).
- If \( u = w_1 w_2 \cdots c_k w_k \), where \( v \in MC(\widehat{\Sigma}) \), each \( w_j \in WM(\widehat{\Sigma}) \), and each \( c_j \in \Sigma_{call} \), then \( M \) is in state \( ([u]_\equiv, [w_k]_{\sim}) \) and the stack is \( ([w_{k-1}]_{\sim_0}, c_k) \cdots ([w_1]_{\sim_0}, c_2) ([u]_{\sim_0}, c_1) \bot \).

The formal construction of \( M \) is as follows. \( M = (Q, q_0, \Gamma, \delta, Q_F) \) where \( Q = \{[u]_{\sim_0} \mid u \in MC(\widehat{\Sigma})\} \cup \{([u]_\equiv, [w]_{\sim}) \mid u \in \Sigma^*, w \in WM(\widehat{\Sigma})\}; q_0 = [e]_{\sim_0}; \Gamma = Q \times \Sigma_{call} \cup \{ \bot \}; Q_F = \{[u]_{\sim_0} \mid u \in L\} \cup \{([u]_\equiv, [w]_{\sim}) \mid u \in L\}; \) and \( \delta \) is defined as follows:
\[ [u]_{\sim_0} \overset{i}{\rightarrow} [u]_{\sim_0} \text{ for every } i \in \Sigma_{\text{int}} \]

\[ [u]_{\sim_0} \overset{c/([u]_{\sim_0}, c)}{\rightarrow} ([uc]_{\sim}, [e]_{\sim}) \text{ for every } c \in \Sigma_{\text{call}} \]

\[ [u]_{\sim_0} \overset{r/j}{\rightarrow} [ur]_{\sim_0} \text{ for every } r \in \Sigma_{\text{ret}} \]

\[ ([u]_{\sim}, [w]_{\sim}) \overset{i}{\rightarrow} ([u]_{\sim}, [w']_{\sim}) \text{ for every } i \in \Sigma_{\text{int}} \]

\[ ([u]_{\sim}, [w]_{\sim}) \overset{c/([w], [w']_{\sim}, c)}{\rightarrow} ([uc]_{\sim}, [e]_{\sim}) \text{ for every } c \in \Sigma_{\text{call}} \]

\[ ([u]_{\sim}, [w]_{\sim}) \overset{r/j([w'], [w]_{\sim})}{\rightarrow} [u'cur]_{\sim_0} \text{ for every } r \in \Sigma_{\text{ret}} \]

\[ ([u]_{\sim}, [w]_{\sim}) \overset{r/j([w']_{\sim}, [w']_{\sim})}{\rightarrow} ([u'cur]_{\sim}, [w'cur]_{\sim}) \text{ for every } r \in \Sigma_{\text{ret}} \]

\[ \square \]

**Remark 6.** Note that in the case where \( \Sigma_{\text{call}} = \Sigma_{\text{ret}} = \emptyset \) (i.e., for regular languages), the machine \( M \) constructed in Theorem 11 is the unique minimum-state automaton for \( L \) because the only reachable states will be of the form \([w]_{\sim_0}\), where \( w \in \Sigma^* \).

Despite the above remark, the VPA \( M \) constructed in Theorem 11 need not be a minimum-state \( \hat{\Sigma} \)-VPA accepting \( L \). Furthermore,

**Proposition 2.** There are VPAs that have no unique minimum-state VPA accepting them.

To illustrate the above proposition, consider the VPAs in Figure 4.1. Let \( \hat{\Sigma} = (\{c_1, c_2\}, \{r\}, \{a, b\}) \). Let \( L = c_1L_1 + c_2L_2r \), where \( L_1 \) is the regular language over \( \{a, b\} \) such that the number of \( a \)'s is even, and \( L_2 \) is the regular language over \( \{a, b\} \) such that the number of \( b \)'s is even. The figure shows two non-isomorphic minimum-state \( \hat{\Sigma} \)-VPAs accepting \( L \). In both machines, the initial state is \( q_0 \) and the following transitions have been omitted in the figure for readability: in both machines, every call-transition not shown is of the form \( q \overset{c_1/2}{\rightarrow} q_2 \), and every other transition not shown goes to state \( q_2 \).
Notice that the first machine consists of two distinct “modules”, one recognizing $L_1$ and one recognizing $L_2$, and the call symbol $c_1$ or $c_2$ determines which module is “invoked”. In contrast, the second machine consists of a single recognizer for both $L_1$ and $L_2$, and this module is invoked regardless of the call symbol. As this example illustrates, it is not clear when splitting the task of recognition into distinct modules reduces the total number of states in the VPA. In the following section, we consider a restricted class of VPAs for which the partition of the VPA into modules has already been provided and the call-symbol determines which module is to be invoked. The task then is to minimize the number of states of the automaton, while preserving the given partition of states into modules.

4.3 $k$-module single-entry visibly pushdown automata

In this section we show that the class of well-matched VPLs have unique minimum-state $k$-module single-entry automata ($k$-SEVPA). As mentioned in the introduction, these automata are motivated by models of programs with finite datatypes, and are similar to single-entry recursive state machines [6].

$k$-SEVPAs. Let $\{\Sigma_{\text{call}}^j\}_{j=1}^k$ be a partition of $\Sigma_{\text{call}}$. A VPA $M = (Q, q_0, \Gamma, \delta, Q_F)$ is a $k$-module single-entry VPA with respect to $\{\Sigma_{\text{call}}^j\}_{j=1}^k$ if there is a partition $\{Q_j\}_{j=0}^k$ of $Q$ and distinguished states $q_j \in Q_j$ for every $j = 1, \ldots, k$ such that:

1. $Q_F \subseteq Q_0, q_0 \in Q_0$;
2. $\Gamma = \{\bot\} \cup (Q \times \Sigma_{\text{call}})$;
3. if $q \xrightarrow{\bot} q'$ for some $i \in \Sigma_{\text{int}}$, then $\exists j. q, q' \in Q_j$;
4. if $q \xrightarrow{c/(i, c)} q'$ for some $c \in \Sigma_{\text{call}}^j$, then $q' = q_j$;
5. if $q' \xrightarrow{c/(i, c)} q''$ for some $c \in \Sigma_{\text{call}}$, then $\exists j. q, q'' \in Q_j$.

Intuitively, $Q_0$ is the base module (corresponding to the ‘main’ module of a program), and the transition relation is such that a call leads to a unique state in the module corresponding to the call (in models of programs, this state is the initial control state of the function called), and upon return will return to the calling module. Such automata are no less expressive than VPAs; as Theorem 12 below shows, for any partition of call symbols, any well-matched VPL is accepted by some $k$-SEVPA.

We use the abbreviation $k$-SEVPA for such machines and explicitly denote them as

$$M = ((Q, q_0, \Gamma, \delta, Q_F), \{\Sigma_{\text{call}}^1, \ldots, \Sigma_{\text{call}}^k\}, \{Q_0, \ldots, Q_k\}, \{q_1, \ldots, q_k\})$$
For example, for the first VPA in Figure 4.1, given the partition \( \Sigma_{\text{call}} = \{ \{ c_1 \}, \{ c_2 \} \} \), there is a partition of \( Q \) as \( \{ Q_0, Q_1, Q_2 \} \), where \( Q_0 = \{ q_0, q_3 \}, Q_1 = \{ q_1, q_3 \} \), and \( Q_2 = \{ q_2, q_4 \} \) witnessing the fact that this is a 2-SEVPA. Similarly, for the second VPA, given the partition \( \Sigma_{\text{call}} = \{ \{ c_1, c_2 \} \} \), there is a partition of \( Q \) as \( \{ Q_0, Q_1 \} \), where \( Q_0 = \{ q_0, q_3 \} \) and \( Q_1 = \{ q_1, q_2, q_3, q_4 \} \) establishing that it is a 1-SEVPA.

**Remark 7.** The automaton constructed in Theorem 10 is a 1-SEVPA. However, in general, this VPA will be much bigger than the smallest 1-SEVPA. The reason for this is similar to the reason why the finite automaton constructed for regular languages from the syntactic congruence is much larger than that obtained from the syntactic right congruence. Thus, in order to characterize the minimal \( k \)-SEVPAs, we need a new congruence that partitions words like the Myhill-Nerode right congruence does for regular languages.

In our construction of the minimal \( k \)-SEVPA for a language \( L \), we will use \( k+1 \) congruences. To model the case when the stack only has \( \bot \), we will use \( \sim_0 \) (defined in Section 4.2.2 on strings with matched calls). For the case when the stack has additional symbols, we need \( k \) new congruences that use the fact that the states of the machine are partitioned into \( k \) modules identified by the call symbol. Given a \( k \)-SEVPA \( M = (M', \{ \Sigma_{\text{call}} \}^k \}_{j=1}^k, \{ Q_j \}^k_{j=0}, \{ q_j \}^k_{j=1}) \) accepting a language \( L \) over \( \hat{\Sigma} \), define the following congruences on well-matched strings:

For every \( j = 1, \ldots, k \),

\[
    w_1 \sim_j w_2 \text{ iff for all } u, v \text{ in } \Sigma^* \text{ and for all } c \text{ in } \Sigma_{\text{call}}^j, ucw_1v \in L \text{ iff } ucw_2v \in L
\]

Since \( \sim_j \)'s will be used to define states when the stack has more than just \( \bot \), when defining the equivalence we only need to consider contexts where there is an unmatched call. We are ready to present the main theorem of this section.

**Theorem 12.** For any well-matched \( \hat{\Sigma} \)-VPL \( L \) and any partition \( \{ \Sigma_{\text{call}} \}^k_{j=1} \) of \( \Sigma_{\text{call}} \), there is a unique (up to isomorphism) minimum-state \( k \)-SEVPA for \( L \) with respect to this partition.

**Proof:** We first show that given any partition \( \{ \Sigma_{\text{call}} \}^k_{j=1} \) of \( \Sigma_{\text{call}} \), there is a \( k \)-SEVPA \( M \) that recognizes \( L \). We construct \( M \) using the equivalences \( \{ \sim_j \}^k_{j=1} \) and \( \sim_0 \) (defined in Section 4.2.2 on strings with matched calls). We then show that this machine \( M \) is the unique minimum-state \( k \)-SEVPA that recognizes \( L \). The construction of \( M \) relies on the observation that \( \sim_0 \) and \( \sim_j \)'s all have finite index if \( L \) is a VPL. From Theorems 10 and 11, we know that when \( L \) is a VPL, \( \sim_0 \) and \( \approx \) are of finite index. Since \( \approx \) is a refinement of \( \sim_j \) for every \( j \), it follows that all \( \sim_j \)'s are also of finite index.

The formal construction of \( M = ((Q, q_0, \Gamma, \delta, Q_F), \{ \Sigma_{\text{call}} \}^k_{j=1}, \{ Q_j \}^k_{j=0}, \{ q_j \}^k_{j=1}) \) is: \( Q_0 = \{ [u]_{\sim_0} \mid u \in MC(\hat{\Sigma}) \} \), and for every \( j = 1, \ldots, k \), \( Q_j = \{ [w]_{\sim_j} \mid w \in WM(\hat{\Sigma}) \} \). For every \( j \geq 0 \), \( q_j = [e]_{\sim_j} \), and \( Q_F = \{ [u]_{\sim_0} \mid u \in L \} \). The transition function \( \delta \) is given as follows.
For every $i \in \Sigma_{\text{init}}$ and $j \geq 0$, $[u]_{\sim_j} \xrightarrow{i} [uv]_{\sim_j}$

For every $c \in \Sigma_{\text{call}}^j$ and $j \geq 0$, $[u]_{\sim_j} \xrightarrow{c/[u]_{\sim_j}, c} [e]_{\sim_j}$

For every $r \in \Sigma_{\text{ret}}$ and $j, j' \geq 0$, $[w]_{\sim_j} \xrightarrow{r/[w]_{\sim_j}, r} [ucwr]_{\sim_j}$

For every $r \in \Sigma_{\text{ret}}$, $[u]_{\sim_0} \xrightarrow{r} [ur]_{\sim_0}$

$Q_F$ is well-defined because $\sim_0$ is an equivalence that saturates $L$. The transition function is consistent because $u_1 i \sim_j u_2 i$ whenever $u_1 \sim_j u_2$ and $i \in \Sigma_{\text{init}}$, and because when $u_1 \sim_j u_2$ and $w_1 \sim_j w_2$, $u_1 cw_1 r \sim_j u_2 cw_2 r$ for every $j > 0, j' \geq 0$ and $c \in \Sigma_{\text{call}}^j, r \in \Sigma_{\text{ret}}$. Thus, the above machine is well defined.

Further observe that the following invariant is maintained during the execution: after reading a string $u$

- If $u \in MC(\hat{\Sigma})$ then the state of $M$ is $[u]_{\sim_0}$ and the stack is $\bot$.
- If $u = w_1 w_2 \ldots w_l$, where $v \in MC(\hat{\Sigma})$, each $c_j \in \Sigma_{\text{call}}$, and each $w_j \in WM(\hat{\Sigma})$, then the state of $M$ is $[w]_{\sim_m}$, and the stack is $(\rho([w]_{\sim_m}, c_1), \ldots, \rho([w]_{\sim_m}, c_l))^{\top}$.

Hence, if a string reaches a final state, we are guaranteed that the stack only has $\bot$, and recognizes $L$.

Consider any $k$-SEVPA $M' = ((Q', q_0', \Gamma', \delta', Q'_F), \{\Sigma_{\text{call}}^j\}_{j=1}^k, \{Q'_j\}_{j=0}^k, \{q'_j\}_{j=0}^k)$ recognizing $L$. We show that $M$ is the unique minimum-state $k$-SEVPA by demonstrating a homomorphism from $M'$ to $M$. In other words, we construct an onto function $f : \bigcup_{j \geq 0} Q'_j \to \bigcup_{j \geq 0} Q_j$ having the following properties.

1. $f(q'_j) = q_j$ for every $j \geq 0$
2. For any $i \in \Sigma_{\text{init}}$, if $p' \xrightarrow{i} M', q'$ then $f(p') \xrightarrow{i} M f(q')$
3. For any $c \in \Sigma_{\text{call}}$, if $p' \xrightarrow{c/[p', c]} M', q'$ then $f(p') \xrightarrow{c/[f(p'), c]} M f(q')$
4. For any $r \in \Sigma_{\text{ret}}$, if $p' \xrightarrow{r/[p', r]} M', q'$ then $f(p') \xrightarrow{r/[f(p'), r]} M f(q')$

Thus, we will be able to conclude that $| \cup Q'_j | \geq | \cup Q_j |$, and if $| \cup Q'_j | = | \cup Q_j |$ then $f$ witnesses an isomorphism between $M$ and $M'$.

The homomorphism $f$ from $M'$ to $M$ is defined as follows:

$$f(q') = \begin{cases} [u]_{\sim_0} & \text{if } \exists u \in MC(\hat{\Sigma}). \delta'(q'_0, u) = q' \\ [w]_{\sim_j} & \text{if } \exists u \in \Sigma^*, c \in \Sigma_{\text{call}}^j, w \in WM(\hat{\Sigma}). \delta'(q'_0, ucw) = q' \end{cases}$$

Note that $f(q'_j) = [e]_{\sim_j}$ for every $0 \leq j \leq k$. Observe that $f$ maps states of $Q'_j$ to the equivalence classes of $\sim_j$ for every $0 \leq j \leq k$. We need to show that $f$ is well defined, i.e., $f$ is indeed a function and does not map a state of $M'$ to two different states of $M$. This follows from the following lemma.
Lemma 11. If $u_1,u_2 \in MC(\hat{\Sigma})$ are such that $\delta_\parallel(q_0,u_1) = \delta_\parallel(q_0,u_2)$, then $u_1 \sim_0 u_2$. In addition, for well-matched strings $w_1$ and $w_2$ and every $j = 1,\ldots,k$, if $\delta_\parallel(q_j,w_1) = \delta_\parallel(q_j,w_2)$ then $w_1 \sim_j w_2$.

The proof of the above lemma is similar to the proofs in Theorems 10 and 11 where we show our congruences to have finite index. Thus, $f$ is indeed a function. Further, $f$ is clearly onto. Also, $f$ preserves initial state and distinguished states $q_j$, by definition. It preserves the transitions of $M'$ because $\sim_0$ and $\sim_j$’s are congruences. This completes the proof that there is a unique minimum-state $k$-SEVPA.

While the above theorem shows that each (well-matched) VPL has a unique $k$-SEVPA with respect to a given partition of $\Sigma_{\text{call}}$, the constructed machine may be much bigger than the smallest VPA recognizing the language because in a $k$-SEVPA, each module is constrained to have a unique “entry” (an entry is the destination of a push-transition). The presence of multiple entries can greatly reduce the size of the VPA as the following proposition shows.

Proposition 3. For every positive integer $n$, there is a family of well-matched VPLs $L_n$ such that the smallest VPA recognizing $L_n$ has at most $O(n^2)$ states, while the smallest 1-SEVPA recognizing $L_n$ has at least $n\sqrt{n}$ states.

Proof: Let $\hat{\Sigma} = (\Sigma_{\text{call}}, \Sigma_{\text{ret}}, \Sigma_{\text{int}})$ where $\Sigma_{\text{call}} = \{a_0,a_1\}$, $\Sigma_{\text{ret}} = \{r\}$ and $\Sigma_{\text{int}} = \{i\}$. For every suitably large positive integer $n$, we define a VPL $L_n$ as follows:

1. Let $m = \lceil \sqrt{n} \rceil$. Fix a set $p_0,p_1,\ldots,p_{m-1}$ of $m$ primes in the range $(n, 2n]$.

(Such a set must exist provided $n$ is sufficiently large)

2. Let $b = \lceil \log m \rceil$. Every string $u \in (a_0 + a_1)^b$ uniquely defines an integer $k(u)$ in the range $0,\ldots,2^b - 1$

3. Let $L_n = \{ui^* r^b \mid u \in (a_0 + a_1)^b$ and $0 \leq k(u) \leq m$ and $s \equiv 1 (\text{mod } p_{k(u)})\}$

We claim that, for every $n$, there is a VPA $M_n$ with at most $n^2$ states accepting $L_n$. For every $k = 0,\ldots,m-1$, let $A_k$ be a $p_k$-state finite automaton accepting the regular language $\{i^s \mid s \equiv 1 (\text{mod } p_k)\}$. $M_n$ can now easily be constructed so that after reading a string $u \in (a_0 + a_1)^b$, the appropriate automaton $A_{k(u)}$ is called, and if the subsequent input string $i^s$ is accepted by $A_{k(u)}$, the machine $M_n$ accepts if the input ends with the string $r^b$. Since each $p_k \leq 2n$, it is easy to verify that $M_n$ has at most $n^2$ states.

We claim that $\sim_{L_n}$ has at least $n\sqrt{n}$ equivalence classes. Let $p = \prod_{k=0}^{m-1} p_k \geq n\sqrt{n}$. For every pair of integers $s,t$ such that $0 \leq s \neq t \leq p-1$, there is a $j \in \{0,\ldots,m-1\}$ such that $p_j$ divides at most one of the integers $\{s,t\}$. Hence, there is a positive integer $x$ such that $s + x \equiv 1 (\text{mod } p_j)$ and $t + x \neq 1 (\text{mod } p_j)$. Thus, for $u \in (a_0 + a_1)^b$ such that $k(u) = j$, $u(i^s)r^b \in L_n$ but $u(i^t)r^b \notin L_n$, i.e., $i^t \not\sim_{L_n} i^s$. Thus, $\sim_{L_n}$ has at least $p \geq n\sqrt{n}$ equivalence classes. \(\square\)
Input: $k$-SEVPA $M = (Q, q_0, \Gamma, \delta, Q_F)$

Output: Minimum-state $k$-SEVPA $M'$ such that $L(M') = L(M)$

Method:

1. Remove all unreachable states; partition the states into $\{Q_j\}_{j=0}^k$.
2. Let $R := \text{Reach}(G_M, \{q_0\} \times Q_F \times \overline{Q_F})$.
3. For every $j \geq 0$ and every $q, q' \in Q_0$ such that $(q_0, q, q') \notin R$, merge states $q, q'$ of $M$.
4. Output the resulting $k$-SEVPA.

Figure 4.2. Algorithm for computing a minimum-state single-entry VPA

4.3.1 Minimization Algorithm

As the following theorem states, there is an efficient algorithm to minimize $k$-SEVPAs.

Theorem 13. Given a $k$-SEVPA $M$ with respect to a partition $\{\Sigma_{\text{call}}^j\}_{j=1}^k$ of $\Sigma_{\text{call}}$ accepting a well-matched language $L$, the unique minimum-state $k$-SEVPA with respect to $\{\Sigma_{\text{call}}^j\}_j$ that accepts $L$ can be computed in time $O(n^3)$, where $n$ is the size $M$.

Proof: If $M = (Q, q_0, \Gamma, \delta, Q_F)$ with states partitioned into $\{Q_j\}_{j=0}^k$ is a $k$-SEVPA accepting a well-matched VP1, we construct the unique minimal $k$-SEVPA equivalent to it in polynomial time.

Consider the given $k$-SEVPA and the homomorphism $f$ defined in the proof of Theorem 12 from this $k$-SEVPA to the minimal $k$-SEVPA. If $q_1, q_2 \in Q_0$ with $f(q_1) \neq f(q_2)$, then they map to two different $\sim_0$ classes $f(q_1) = [w_1]_{\sim_0}$ and $f(q_2) = [w_2]_{\sim_0}$, where $w_1, w_2 \in WM(\overline{\Sigma})$. Hence, there must be some well-matched word $w$ such that $w_1w \in L$ and $w_2w \notin L$. We call such a $w$ a $\sim_0$-witness for $(q_1, q_2)$. That is, we say $w \in WM(\overline{\Sigma})$ is a $\sim_0$-witness for $(q_1, q_2) \in Q_0 \times Q_0$ if $w_1w \in L(M)$ and $w_2w \notin L(M)$ for some (and hence every) $w_1, w_2 \in WM(\overline{\Sigma})$ such that $\delta_\perp(q_0, w_1) = q_1$ and $\delta_\perp(q_0, w_2) = q_2$.

Similarly, if $q_1, q_2 \in Q_j$ and $f(q_1) = [w_1]_{\sim_j}$, $f(q_2) = [w_2]_{\sim_j}$, for some $w_1, w_2 \in WM(\overline{\Sigma})$, and $[w_1]_{\sim_j} \neq [w_2]_{\sim_j}$, then there must be $uc \in MR(\overline{\Sigma}), \Sigma_{\text{call}}^j$, $v \in \Sigma^*$ such that $ucw_1v \in L(M)$ and $ucw_2v \notin L(M)$. We call the pair $(uc, v)$ a $\sim_j$-witness for the pair $(q_1, q_2)$; i.e., $(uc, v)$ is a $\sim_j$-witness for $(q_1, q_2)$ if for some (and hence every) pair of words $w_1$ and $w_2$, with $\delta_\perp(q_j, w_1) = q_1$ and $\delta_\perp(q_j, w_2) = q_2$, $ucw_1v \in L$ and $ucw_2v \notin L$.

The overall algorithm is given in Figure 4.2. The algorithm constructs a graph such that finding whether $(q_1, q_2)$ has $\sim_0$- and $\sim_j$-witnesses reduce to checking reachability properties on this graph. We first remove the set of all unreachable states and then compute the set $R$ of all nodes from which some
node in \( \{q_0\} \times Q_F \times Q_F \) is reachable. This graph has the property that \( R \) contains \((q_0, q_1, q_2)\) iff \((q_1, q_2)\) has a \( \sim_0 \)-witness and contains \((q_j, q_1, q_2)\) iff \((q_1, q_2)\) has a \( \sim_j \)-witness. The \( \sim_j \)-equivalent (and the \( \sim_0 \)-equivalent) states are then merged to give the required minimal automaton.

The graph \( G_M \): Given the \( k \)-SEVPA \( M \), we define the directed graph \( G_M = (V, E) \) where \( V = \bigcup_{j=0}^{k} (\{q_j\} \times Q_j \times Q_j) \) and \( E \) is defined as follows. There are six types of edges:

1. \( \forall q_1, q_2 \in Q_0, (q_0, q_1, q_2) \rightarrow E (q_0, q_1', q_2) \) whenever \( \exists i \in \Sigma_{\text{int}} \) such that \( q_1 \xrightarrow{i} q_1' \) and \( q_2 \xrightarrow{i} q_2' \);

2. \( \forall q_1, q_2 \in Q_0, (q_0, q_1, q_2) \rightarrow E (q_0, q_1', q_2') \) whenever \( \exists e \in \Sigma^*_{\text{call}}, \exists r \in \Sigma_{\text{ret}}, \exists q \in Q_j \) such that \( q_1 \xrightarrow{e/\gamma} q_j, q_2 \xrightarrow{e/\gamma_2} q_j, q \xrightarrow{r/\gamma} q_1' \) and \( q \xrightarrow{r/\gamma_2} q_2' \);

3. \( \forall q_1, q_2 \in Q_j, (q_j, q_1, q_2) \rightarrow E (q_0, q_1', q_2) \) whenever \( \exists q \in Q_0, \exists e \in \Sigma^*_{\text{call}}, \exists r \in \Sigma_{\text{ret}} \) such that \( q \xrightarrow{e/\gamma} q_j, q_1 \xrightarrow{r/\gamma} q_1' \) and \( q_2 \xrightarrow{r/\gamma_2} q_2' \);

4. \( \forall q_1, q_2 \in Q_j, (q_j, q_1, q_2) \rightarrow E (q_j, q_1', q_2') \) whenever \( \exists i \in \Sigma_{\text{int}} \) such that \( q_1 \xrightarrow{i} q_1' \) and \( q_2 \xrightarrow{i} q_2' \);

5. \( \forall q_1, q_2 \in Q_j, (q_j, q_1, q_2) \rightarrow E (q_j, q_1', q_2') \) whenever \( \exists q \in Q_j, \exists e \in \Sigma^*_{\text{call}}, \exists r \in \Sigma_{\text{ret}} \) such that \( q \xrightarrow{e/\gamma} q_j, q_1 \xrightarrow{r/\gamma} q_1' \) and \( q_2 \xrightarrow{r/\gamma_2} q_2' \);

6. \( \forall q_1, q_2 \in Q_j, (q_j, q_1, q_2) \rightarrow E (q_j, q_1', q_2') \) whenever \( \exists q \in Q_j, \exists e \in \Sigma^*_{\text{call}}, \exists r \in \Sigma_{\text{ret}} \) such that \( q \xrightarrow{e/\gamma} q_j, q_2 \xrightarrow{e/\gamma_2} q_j, q \xrightarrow{r/\gamma} q_1' \) and \( q \xrightarrow{r/\gamma_2} q_2' \).

Intuitively, the rules can be understood as reducing checking witnesses for a pair \((q_1, q_2)\) to that of checking witnesses for other pairs \((q_1', q_2')\). Intuitively, the triple \((q_0, q_1, q_2)\) stands for the assertion that \((q_1, q_2)\) has a \( \sim_0 \)-witness, while \((q_j, q_1, q_2)\) stand for the assertion that \((q_1, q_2)\) has a \( \sim_j \)-witness. Transition (1) above, for example says that \((q_1, q_2)\) has a \( \sim_0 \)-witness if they have a \((q_1', q_2')\) witness, where \( q_1' \) and \( q_2' \) are the destination states of \( q_1 \) and \( q_2 \), respectively, on an internal action \( i \). The other transitions encode other ways to reduce the length of the witnesses to be discovered, and if we reach a node in \( \{q_0\} \times Q_F \times Q_F \), we can stop as we know that \( \epsilon \) is a \( \sim_0 \)-witness for any pair in \( Q_F \times Q_F \). The rules for finding \( \sim_j \)-witnesses are similar.

Before formally proving correctness of the algorithm, we first prove a few useful lemmas about witnesses.

**Lemma 12.** For every \( j \geq 0 \) and every \((q_1, q_2) \in Q_j \times Q_j\), if \((q_j, q_1, q_2) \in R\) then \((q_j, q_1, q_2)\) has a \( \sim_j \)-witness.

**Proof:** We prove the lemma by showing that, for each edge in \( G_M \) from vertex \( x \) to vertex \( y \), a witness for \( x \) can be inferred from a witness for \( y \) depending on the type of edge. In particular, if \( w \) is a \( \sim_0 \)-witness for \((q_0, q_1', q_2')\), then

1. for edges of type 1, \( iw \) is a \( \sim_0 \)-witness for \((q_0, q_1, q_2)\);
2. for edges of type 2, \( cw_irw \) is a \( \sim_0 \)-witness for \((q_0, q_1, q_2)\);

3. for edges of type 3, \((w_ic,rw)\) is a \( \sim_j \)-witness for \((q_j, q_1, q_2)\).

Similarly, if \((uc', v)\) is a \( \sim_j \)-witness for \((q_{j'}, q'_1, q'_2)\), then

1. for edges of type 4, \((uc', iv)\) is a \( \sim_j \)-witness for \((q_{j'}, q_1, q_2)\);

2. for edges of type 5, \((uc'w_ic, rv)\) is a \( \sim_j \)-witness for \((q_j, q_1, q_2)\);

3. for edges of type 6, \((uc', cw_irv)\) is a \( \sim_j \)-witness for \((q_{j'}, q_1, q_2)\).

\[ \square \]

**Lemma 13.** For every \( q_1, q_2 \in Q_0 \), if \((q_1, q_2)\) has a \( \sim_0 \)-witness then \((q_0, q_1, q_2)\) \( \in \) \( R \).

**Proof:** Let \( w \) be a \( \sim_0 \)-witness for \((q_1, q_2)\). We prove by induction on \(|w|\) (the length of \( w \)) that \((q_0, q_1, q_2)\) \( \in \) \( R \). Let \( w_1, w_2 \in WM(\hat{\Sigma}) \) such that \( \delta_\perp(q_0, w_1) = q_1 \) and \( \delta_\perp(q_0, w_2) = q_2 \).

**Base case:** \(|w| = 0\). In this case \( w_1 \in L(M) \) hence \( q_1 \in Q_F \), and \( w_2 \notin L(M) \) hence \( q_2 \notin Q_F \). By definition, \((q_0, q_1, q_2) \in W \subseteq \) \( R \).

**Inductive step:** There are two subcases.

**Case 1:** Suppose \( w = iw'\) for some \( i \in \Sigma_{\text{int}} \). Let \( q_1 \xrightarrow{i} q'_1 \) and \( q_2 \xrightarrow{i} q'_2 \). By definition, \((q_0, q_1, q_2) \rightarrow_E (q_0, q'_1, q'_2)\) and hence \((q_0, q_1, q_2)\) \( \in \) \( R \) if \((q_0, q'_1, q'_2)\) \( \in \) \( R \). We now establish that \((q_0, q'_1, q'_2)\) \( \in \) \( R \). We have \( \delta_\perp(q_0, w_1) = \delta_\perp(q_0, w_1, i) = \delta_\perp(q_1, i) = q'_1 \) and similarly \( \delta_\perp(q_0, w_2) = q'_2 \). Hence \( q'_1, q'_2 \in Q_0 \). Further, \( w_1w' = w_1w \in L(M) \) and \( w_2w' \notin L(M) \). Hence, \( w' \) is a \( \sim_0 \)-witness for \((q_0, q'_1, q'_2)\), and since \(|w'| < |w|\), it follows by induction that \((q_0, q'_1, q'_2)\) \( \in \) \( R \).

**Case 2:** Suppose \( w = cw_irw''\) for some \( c \in \Sigma^i_{\text{call}}, r \in \Sigma_{\text{ret}}, w', w'' \in WM(\hat{\Sigma})\). Let \( \delta_\perp(q_j, w') = q \) and suppose \( q_1 \xrightarrow{\epsilon / r} q_j, q_2 \xrightarrow{\epsilon / r} q_j, q \xrightarrow{r / r} q'_1 \) and \( q \xrightarrow{r / r} q'_2 \). Then, by definition, \((q_0, q_1, q_2) \rightarrow_E (q_0, q'_1, q'_2)\). Using a similar argument as before, it can be shown that \( q'_1, q'_2 \in Q_0 \) and \( w'' \) is a \( \sim_0 \)-witness for \((q_0, q'_1, q'_2)\), and hence by induction \((q_0, q'_1, q'_2)\) \( \in \) \( R \). Hence \((q_1, q_2)\).

This completes the proof. \[ \square \]

**Lemma 14.** For every \( q_1, q_2 \in Q_j \), if \((q_1, q_2)\) has a \( \sim_j \)-witness then \((q_j, q_1, q_2)\) \( \in \) \( R \).

**Proof:** Let \((uc, v)\) be a \( \sim_j \)-witness for \((q_1, q_2)\). We prove by induction on \(|ucv|\) (the length of \( ucv \)) that \((q_j, q_1, q_2)\) \( \in \) \( R \). Let \( w_1, w_2 \in WM(\hat{\Sigma}) \) such that \( \delta_\perp(q_j, w_1) = q_1 \) and \( \delta_\perp(q_j, w_2) = q_2 \).

**Base case:** \(|ucv| = |u|+2\), \( u \in WM(\hat{\Sigma}) \). In this case we must have \( v = r \) for some \( r \in \Sigma_{\text{ret}} \). Let \( q'_1 = \delta_\perp(q_0, ucwr) \) and \( q'_2 = \delta_\perp(q_0, ucwr) \). Since \( ucwr \in L(M) \), \( q'_1 \in Q_F \) and since \( ucwr \notin L(M) \), \( q'_2 \notin Q_F \). Hence \((q_0, q'_1, q'_2)\) \( \in \) \( W \subseteq \) \( R \).

Further, since \( \delta_\perp(q_j, w_1) = q_1 \), \( \delta_\perp(q_j, w_2) = q_2 \) and \( \delta_\perp(q_0, u) \xrightarrow{\epsilon / r} q_j \) for some \( \gamma \in \Gamma \), it follows that \( q_1 \xrightarrow{r / \gamma} q'_1 \) and \( q_2 \xrightarrow{r / \gamma} q'_2 \), and hence \((q_j, q_1, q_2) \rightarrow_E (q_0, q'_1, q'_2)\). Hence \((q_j, q_1, q_2)\) \( \in \) \( R \).
Inductive step: There are three subcases.

**Case 1:** Suppose \( v = iv' \) for some \( i \in \Sigma_{\text{init}} \). Let \( q_1 \xrightarrow{i} q'_1 \) and \( q_2 \xrightarrow{i} q'_2 \). By definition, \( (q_j, v_1, v_2) \rightarrow_E (q_j, q'_1, q'_2) \). It can easily be established that \( q'_1, q'_2 \in Q_j \) and that \( (uc, v') \) is a \( \sim_j \)-witness for \( (q_j, q'_1, q'_2) \), and hence by the inductive hypothesis \( (q_j, q'_1, q'_2) \in R \).

**Case 2:** Suppose \( uc = u'v \) for some \( u \in W(M) \), \( v = rv' \) for some \( r \in \Sigma_{\text{init}} \). Let \( \delta_\Delta(q_0, u'v) = q_1 \) and \( \delta_\Delta(q_0, u'vuvw) = q_2 \). Then, \( \exists \gamma \in \Gamma \) such that \( q_1 \xrightarrow{r} q_j \) and \( q_2 \xrightarrow{r} q_j \). Hence, by definition, \( (q_j, q_1, q_2) \rightarrow_E (q_j, q'_1, q'_2) \). It can be shown that \( q'_1, q'_2 \in Q_j \) and that \( (uc, v') \) is a \( \sim_j \)-witness for \( (q_j, q'_1, q'_2) \), and hence by the inductive hypothesis, \( (q_j, q'_1, q'_2) \in R \). Hence, \( (q_j, q_1, q_2) \in R \).

**Case 3:** Suppose \( v = v'wv' \) for some \( v \in \Sigma_{\text{init}} \), \( w \in W(M) \). Let \( \delta_\Delta(q_0, uvw) = q'_1 \) and \( \delta_\Delta(q_0, ucvwv) = q'_2 \). Then, by definition, \( (q_j, q_1, q_2) \rightarrow_E (q_j, q'_1, q'_2) \). It can be shown that \( q'_1, q'_2 \in Q_j \) and that \( (uc, v') \) is a \( \sim_j \)-witness for \( (q_j, q'_1, q'_2) \), and hence by the inductive hypothesis, \( (q_j, q'_1, q'_2) \in R \). Hence, \( (q_j, q_1, q_2) \in R \).

This completes the proof. \( \square \)

**Lemma 15.** Let \( f \) be the homomorphism defined in Theorem 12. Then for every \( j \geq 0 \) and every \( q_1, q_2 \in Q_j \), there is a \( \sim_j \)-witness for \( (q_1, q_2) \) if and only if \( f(q_1) \neq f(q_2) \).

**Proof:** Suppose \( q_1, q_2 \in Q_0 \) and \( (q_1, q_2) \) has a \( \sim_0 \)-witness. Let \( w_1, w_2 \in W(M) \) such that \( \delta_\Delta(q_0, w_1) = q_1 \) and \( \delta_\Delta(q_0, w_2) = q_2 \). Let \( w \in W(M) \) such that \( w_1w \in L(M) \) and \( w_2w \notin L(M) \). Since \( w_1 \in \text{prefix}(L(M)) \), \( f(q_1) = [w_1]_{\sim_0} \). If \( f(q_1) = f(q_2) \), then \( w_1 \sim_0 w_2 \), but this is impossible since \( w_1w \in L(M) \) whereas \( w_2w \notin L(M) \). Hence, \( f(q_1) \neq f(q_2) \).

Suppose \( q_1, q_2 \in Q_j \) (where \( j > 0 \)) and \( (q_1, q_2) \) has a \( \sim_j \)-witness. Let \( w_1, w_2 \in W(M) \), \( w = wrv \) such that \( \delta_\Delta(q_0, uvw) = q_1 \), \( \delta_\Delta(q_0, ucvwv) = q_2 \), \( ucvw \in W(M) \) and \( ucvw \notin L(M) \). Then \( f(q_1) = [w_1]_{\sim_j} \) and \( f(q_2) = [w_2]_{\sim_j} \) but \( w_1 \neq w_2 \) since \( ucvw \notin L(M) \) whereas \( ucvw \notin L(M) \). Hence, \( f(q_1) \neq f(q_2) \).

Conversely, suppose \( f(q_1) \neq f(q_2) \) and we show that there is a \( \sim_j \)-witness for \( (q_1, q_2) \) or \( (q_2, q_1) \). By definition, we only need to consider the cases when \( q_1 \) and \( q_2 \) both belong to \( Q_j \) for some \( j \geq 0 \).

Suppose \( q_1, q_2 \in Q_0 \) and \( f(q_1) \neq f(q_2) \). Let \( w_1, w_2 \in W(M) \) such that \( \delta_\Delta(q_0, w_1) = q_1 \) and \( \delta_\Delta(q_0, w_2) = q_2 \). We have \( f(q_1) = [w_1]_{\sim_0} \neq [w_2]_{\sim_0} = f(q_2) \), and hence \( \exists w \in W(M) \) such that exactly one of \( \{w_1w, w_2w\} \) is in \( L(M) \). Hence, \( w \) is a \( \sim_0 \)-witness for \( (q_1, q_2) \) or \( (q_2, q_1) \).

Suppose \( q_1, q_2 \in Q_j \) (where \( j > 0 \)) and \( f(q_1) \neq f(q_2) \). Let \( w_1, w_2 \in W(M) \) such that \( \delta_\Delta(q_0, w_1) = q_1 \) and \( \delta_\Delta(q_0, w_2) = q_2 \). We have \( f(q_1) = [w_1]_{\sim} \neq [w_2]_{\sim} = f(q_2) \), and hence \( \exists c \in \Sigma_{\text{call}}, v \in \Sigma^* \) such that exactly one of \( \{ucv_1w, ucv_2w\} \) is in \( L(M) \). Hence, \( (uc, v) \) is a \( \sim_j \)-witness for \( (q_1, q_2) \) or \( (q_2, q_1) \). \( \square \)
We are now ready to prove correctness of our algorithm. The algorithm merges \((q_1, q_2) \in Q_j \times Q_j\) such that \((q_j, q_1, q_2) \notin R\). We first show that 
\((q_0, q_1, q_2) \in R \iff f(q_1) \neq f(q_2)\) (recall that \(f\) is the homomorphism defined in Theorem 12). To do this, we first observe that \((q_0, q_1, q_2) \in R \iff (q_1, q_2)\) has a \(\sim_0\)-witness, and for \(j > 0\), \((q_j, q_1, q_2) \in R \iff (q_1, q_2)\) has a \(\sim_j\)-witness (which follows from Lemmas 12, 13 and 14). Next we observe that \((q_1, q_2)\) has a \(\sim_j\)-witness if \(f(q_1) \neq f(q_2)\), by Lemma 15. It is easy to see that if \(f(q_1) = f(q_2)\), then \(q_1\) and \(q_2\) can be merged to yield a well-defined deterministic \(k\)-SEVPA \(M'\) which accepts the same language as \(L\). Hence, the \(k\)-SEVPA \(M'\) obtained by merging all pairs of states of \(M\) that do not have a witness is a well-defined deterministic \(k\)-SEVPA accepting the language \(L\). Further, \(M'\) is a \(k\)-SEVPA with respect to \(\{\Sigma_{call}^j\}_{j=1}^k\) and has at most as many states as the minimal \(k\) SEVPA for this partition (by Lemma 15). Hence, \(M'\) is in fact a minimum-state normalized \(k\)-SEVPA for \(M\).

This completes the proof of correctness of the algorithm.

Turning to the complexity of the algorithm, we now show that it runs in \(O(n^3)\) time, where \(n\) is the size of the given VPA. We assume that \(|\Sigma|\) is bounded above by a constant. The sets \(Q_j\) can be constructed in \(O(n^3)\) time using the algorithm in [24]. The size of the graph \(G_M\) is clearly \(O(n^3)\) (notice that \(G_M\) has \(O(n^2)\) nodes, and there are \(O(n^3)\) edges of types 2, 3, 5 and 6), and hence the set \(R\) can be computed in \(O(n^3)\) time. The remaining steps can clearly be completed in \(O(n^2)\) time. Thus, the algorithm runs in \(O(n^3)\) time. \(\square\)

### 4.4 Conclusions

We presented a characterization of \(\textbf{VPLS}\) in terms of congruences on strings of finite index and gave constructions of canonical automata recognizing visibly pushdown languages. We showed that while \(\textbf{VPLS}\) in general do not have unique minimum-state deterministic recognizers, the class of well-matched \(\textbf{VPLS}\) do have unique minimal \(k\)-module single-entry deterministic visibly pushdown automata (\(k\)-SEVPAs) for any fixed partition of the call symbols.

We also presented a minimization algorithm for \(k\)-SEVPAs that runs in time \(O(n^3)\). The computational complexity of the problem of constructing the smallest \(k\)-SEVPA given any visibly pushdown automaton (not necessarily \(k\)-module) is open, and would be interesting to investigate.
Chapter 5

Conformance Testing for Recursive Program Models

5.1 Introduction

In Chapter 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 minimally 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 “compute” only the answer to the specific parameter with which it was called.

In Chapter 4, we also presented a minimization result for a special class of VPA with a modular structure 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 (k-SEVPA).

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 result in requiring a lot more memory, which in the context of automata corresponds to larger number of states.

In this chapter, we first present a minimization result for a variant of modular VPAs that has multiple entry points in each module, corresponding to the multiple parameters. This variant is novel 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 applies only to complete models*.

Next we present the main contribution of this chapter: we tackle the problem of conformance testing for complete modular VPAs. The size of our conformance 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.

This chapter is organized as follows. We first introduce the model of modular VPAs, along with useful definitions and notation. In Section 5.3 we present our results on the existence of unique, minimal, complete, modular VPAs. Our conformance testing results are presented in Section 5.4.

5.2 Preliminaries

In this section, we define modular VPAs, and introduce notation is specific to this chapter. We model 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 $\Sigma_{\text{call}}$ denote the set of all calls. Let us also fix a finite set of internal actions $\Sigma_{\text{int}}$, and let $\Sigma_{\text{ret}} = \{ r \}$ be the alphabet of returns, containing the unique symbol $r$. Let $\hat{\Sigma} = (\Sigma_{\text{call}}, \Sigma_{\text{int}}, \Sigma_{\text{ret}})$ and let $\Sigma = \Sigma_{\text{call}} \cup \Sigma_{\text{int}} \cup \Sigma_{\text{ret}}$. We now define (nondeterministic) modular VPAs.

**Definition 7 (Modular VPAs).** A modular VPA over $(M, \{P_m\}_{m \in M}, m_0, \hat{\Sigma})$ is a tuple

\[
(\{Q_m, \{q^p_m\}_{p \in P_m}, \delta_m\}_{m \in M}, F)
\]

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.

*However, we have shown [44] that any incomplete machine model for a program can be translated into a canonical, complete, machine model which is at most $k$ times larger than the incomplete model, where $k$ is the maximum number of parameters in any module. We can therefore obtain an approximate minimization procedure for incomplete models [see [44] for further details].

53
• For each parameter \( p \in P_m \), \( q^p_m \) 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 \( q^p_m \) be different from \( q^p_m^\prime \), when \( p \neq p^\prime \).)

• \( \delta_m = (\delta^m_{\text{call}}, \delta^m_{\text{int}}, \delta^m_{\text{return}}) \) is a triple of transition relations, one for calls, one for internals and one for returns, where

\[
\delta^m_{\text{call}} \subseteq \{ (q, (n, p), q^p_n) \mid q \in Q_m, (n, p) \in \Sigma_{\text{call}} \};
\]

\[
\delta^m_{\text{int}} \subseteq \{ (q, a, q^\prime) \mid q, q^\prime \in Q_m, a \in \Sigma_{\text{int}} \};
\]

\[
\delta^m_{\text{return}} \subseteq \{ (q, q^\prime, q^\prime\prime) \mid q^\prime, q^\prime\prime \in Q_m, q \in Q \};
\]

• \( F \subseteq Q_0 \) is the set of final states.

**Notation**

We write \( q \xrightarrow{(n,p)} q^p_n \) to mean \((q, (n, p), q^p_n) \in \delta^m_{\text{call}}, q \xrightarrow{a} q^\prime \) to mean \((q, a, q^\prime) \in \delta^m_{\text{int}}, \) and \( q \xrightarrow{d} q^\prime \) to mean \((q, q^\prime, q^\prime\prime) \in \delta^m_{\text{return}}.

**Semantics**

In order to cleanly handle minor technical issues that arise when VPAs are restricted to push only the state onto the stack, modular VPAs require a departure from the standard VPA semantics used thus far. A stack, now, is a finite string over \( Q \); let the set of all stacks be \( St = Q^* \). A configuration is any pair \((q, \sigma)\) 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 \( \Sigma \)-labeled edges that satisfies:

**Initial edge** the edge \( c_0 \xrightarrow{(m_0,p_0)} (q^p_{m_0}, \epsilon) \) is in \( E \).

**Internal edges** If \((q, \sigma) \in V \) \((q \in Q_m) \) and \((q, a, q^\prime) \in \delta^m_{\text{int}}, \) then the edge \( (q, \sigma) \xrightarrow{a} (q^\prime, \sigma) \) is in \( E \).

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

**Return edges** If \((q, \sigma q^\prime) \in V \) \((q' \in Q_m) \), and \((q, q', q^\prime\prime) \in \delta^m_{\text{return}}, \) then the edge \( (q, \sigma q^\prime) \xrightarrow{\Delta} (q^\prime\prime, \sigma) \) is in \( E \).

(Note that \( q^\prime\prime \) and \( q^\prime \) belong to the same module \( m \).)

A run of \( A \) on a word \( u \) is a path in the configuration graph on \( u \). Let \( \rho : Conf \times \Sigma^* \rightarrow 2^{Conf} \) 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 \( A \).

As before, \( WM \) will denote the set of well-matched words over \( \hat{\Sigma} \). We will denote by \( w, w', w, \ldots \) words in \( WM \). Note that a modular VPA accepts only words that are in \( \{(m_0, \rho_0)\} \). \( WM \) (since the final states are in module \( m_0 \), and the initial symbol \( (m_0, \rho_0) \) 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 \( m \in M \):

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

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 \( (n, p) \in \Sigma_{call}, (q, (n, p), q_p^m) \in \delta^m_{call} \);
- for each \( q \in Q_m \) and \( a \in \Sigma_{int}, \exists q' \) such that \( (q, a, q') \in \delta^m_{int} \); and
- for each \( q \in Q \) and \( q' \in Q_m, \exists q'' \) such that \( (q, q', q'') \in \delta^m_{int} \).

The size of a modular VPA is the number of states in it; when we refer to minimization of modular VPAs, we minimize 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 for every \( u', u'(m, p) \) 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 restriction that allows our minimization result to go through.

### 5.3 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 complete 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 14.** If $A$ is a complete modular VPA, then there exists a unique
minimal deterministic complete modular VPA $A'$ such that $L(A') = L(A)$.

**Proof:** As in Theorem 12, our proof will construct the minimal automaton $A'$
whose states will correspond to equivalence classes of a suitably chosen equi-
nivalence relation. Let $A = (\{Q_m, \{q^p_m\}_{p \in P_m}, \delta_m\}_{m \in M}, F)$ 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 $u, v \in \Sigma^*$

$$u(m, p_1)w_1 v \in L \text{ iff } u(m, p_2)w_2 v \in L$$

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 $w$, $(p_1, w_1a) \sim_m (p_2, w_2a)$.

Let $[([p, w])_m]$ denote the equivalence class of $(p, w)$ with respect to $\sim_m$. We
now show that $\sim_m$ has only finitely many equivalence (at most $2^{3R+1}$) equi-
nivalence 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 $R_{p, w}$ be the set of states
reached by $A$ starting in $q^p_m$ 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 $\hat{A} = (\{\hat{Q}_m, \{\hat{q}^p_m\}_{p \in P_m}, \hat{\delta}_m\}_{m \in M}, \hat{q}_0, \hat{F})$ as follows:
for each $m \in M$, $\hat{Q}_m = \{[([p, w])_m] | p \in P_m, w \in WM\}$, and for each $p \in P_m$,
$\hat{q}^p_m = [([p, \epsilon])_m]$, and the transition relation $\hat{\delta}_m = (\hat{\delta}_{\text{call}}, \hat{\delta}_{\text{int}}, \hat{\delta}_{\text{return}})$ is defined as:

1. **call:** $\hat{\delta}_{\text{call}}([([p, w])_m, (m', p')]) = [([p', \epsilon])_m]$ for all $(m', p') \in \Sigma_{\text{call}}$. In other
   words $\hat{q}^p_m = [([p, \epsilon])_m]$.
2. **internal:** $\hat{\delta}^m_{\text{int}}([([p, w])_m, a]) = [([p, aw])_m]$, for all $a \in \Sigma_{\text{int}}$
3. **return:** $\hat{\delta}^m_{\text{return}}([([p', w'])_m, ([p, w])_m]) = [([p, w(m', p')w'])_m]$. The final states are $\hat{F} = \{[([p_0, w])_{m_0} | (m_0, p_0)w \in L\}$

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

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

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 $A$. Note that $A$ and $\hat{A}$ share the same set of modules $M$.

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

$$(p_1, w_1) \approx_m (p_2, w_2) \text{ iff } \rho((q^p_m, e), w_1) = \rho((q^p_m, e), w_2)$$

Since the configuration reached after reading a well-matched word will have an empty stack, $\approx_m$ 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 $\rho((q^p_m, e), w_1) = \rho((q^p_m, e), w_2)$, 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.

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.

### 5.4 Conformance testing

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

1. $S$ is minimized and has $n$ states;

2. $I$ is equivalent to a deterministic complete modular VPA that has at most $N$ states;

3. $I$ does not change during the testing experiment.

Note that assumption 1 can be made with no loss of generality, since the specification $S$ is known, and hence we can assume it is minimized. Assump-
tion 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 $\Sigma$ is a pair $(T^+, T^-)$, where $T^+, T^-$ are finite subsets of $\Sigma^*$. A modular VPA $A$ is consistent with sample $(T^+, T^-)$ if $T^+ \subseteq L(A)$ and $T^- \subseteq \overline{L(A)}$.

**Definition 8.** A conformance test for $(S, I)$ is a sample $(T^+, T^-)$ over $\Sigma$ such that $S$ is consistent with $(T^+, T^-)$ and, for any $I$ satisfying the above assumptions, $I$ is consistent with $(T^+, T^-)$ if and only if $L(I) = L(S)$.

Let the set of states of $S$ be $Q_S = \{q_1, q_2, \ldots, q_n\}$, and for each $i \in [n]$ let $(m_i, p_i)w_i$ be an access string for $q_i$. Further assume without loss of generality that the access string for every state $q^*_m \in Q_S$ is $(m, p)$, and that $q_i = q^*_m$. Let $F_S$ be the final states of $S$.

Let the set of states of $I$ be $Q_I = \{\hat{q}_1, \hat{q}_2, \ldots, \hat{q}_N\}$, let $\hat{q}_1 = \hat{q}^*_m$, and let $F_I$ be the set of final states of $I$.

Since $S$ is minimized and has $n$ states, for $I$ to be equivalent to $S$ it is necessary that the access strings $x_1, \ldots, x_n$ lead to distinct states in $I$. Therefore without loss of generality, let us suppose that for each $i \in [n]$, $x_i$ is an access string for $\hat{q}_i$. Hence we observe that the number of states in $I$, namely $N$, must be at least the number of states in $S$, namely $n$.

In the case when $N > n$, we require access strings for the remaining states $\hat{q}_{n+1}, \ldots, \hat{q}_N$. Since $I$ and $S$ share the same set of modules and parameters, and since we already have an access string $(m, p)$ for each entry state $\hat{q}^*_m$ of $I$, all remaining states are not entry states for modules. Any such state $\hat{q}$ in module $m$ must be accessed by a string of the form $(m, p)w$ where $p \in F_m$ and $w$ is a non-empty well-matched string (thus $w$ cannot end with a call symbol).

Hence we can always select each such access string $x_q$ so that exactly one of the following conditions are satisfied:

- $w = w' \alpha$ where $\alpha \in \Sigma_{\text{int}}$ and $y = (m, p)w'$ is an existing access string; or
- $w = w'(m', p')w''r$ where $y = (m, p)w'$ and $y = (m', p')w''$ are existing access strings.

With these observations in place, we immediately have the following lemma:

$n$ distinct states. Using the fact that $S$, 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.

**Lemma 16.** Let $S$ be minimized with $n$ states, and with access strings $x_1, \ldots, x_n$. Let $D$ be a complete set of distinguishing tests for the minimized automaton $S$ i.e., 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 $i \in [n]$, let $D_i = \bigcup_{j \neq i} \{(u_{ij}, v_{ij})\}$. Define $T_0 = \bigcup_{i=1}^n \{(u(m_i, p_i)w)_{i+1} | (u, v) \in D_i\}$, $T_0^+ = T_0 \cap L(S)$ and $T_0^- = T_0 \setminus L(S)$.
If $I$ is consistent with $(T^+_0, T^-_0)$, then $x_1, \ldots, x_n, x_{n+1}, \ldots, x_N$ are access strings for all states of $I$, where for each $i > n$, $x_i$ is of one of the following forms: $x_i = ya$, where $a \in \Sigma\text{int}_i$; 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 $I$ 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 $q_i$. If $I$ is equivalent to $S$, it is necessary that for each $i$, $x_i$ is an access string of a final state of $I$ if and only if $x_i$ is an access string of a final state in $S$. We define a sample $(T^+_1, T^-_1)$ such that $I$ is consistent with this sample if this condition holds.

Define $h : Q_I \to Q_S$ as follows: $h(q_i) = q_j$ if $x_i$ is an access string for $q_j$ in $S$. Define $T_1 = \{x_i | i = 1, \ldots, N\}$. Let $T^+_1 = T_1 \cap L(S)$ and $T^-_1 = T_1 \setminus L(S)$. We immediately have the following lemma:

**Lemma 17.** If $I$ is consistent with $(T^+_1, T^-_1)$, then for every $1 \leq i \leq n$, $q_i \in F_I$ iff $h(q_i) \in F_S$.

Our goal is to design a sample $(T^+, T^-)$ such that if $I$ is consistent with it, then $L(I) = L(S)$. In view of Lemma 17, it is enough to construct a sample such that if $I$ is consistent with it, then for every $u \in MR$, $h(q_i) \overset{u}{\Rightarrow}_S h(q_j)$ whenever $q_i \overset{u}{\Rightarrow}_I q_j$. Define

\[
T_2 = \bigcup_{i=1}^{n} \{ux_iav | a \in \Sigma\text{int}_i, (u, v) \in D_j \text{ where } h(q_i) \overset{a}{\Rightarrow}_S q_j\}
\]

\[
T_3 = \bigcup_{i,j=1}^{n} \{ux_iavv | (u, v) \in D_k \text{ where } h(q_i) \overset{h(q_j)}{\Rightarrow}_S q_k\}
\]

It is not hard to see that if $I$ is consistent with $(T_2 \cap L(S))$, then for every $a \in \Sigma\text{int}_i$, $h(q_i) \overset{a}{\Rightarrow}_S h(q_j)$ whenever $q_i \overset{a}{\Rightarrow}_I q_j$. Similarly, it can be shown that if $I$ is consistent with $(T_3 \cap L(S))$, then $h(q_i) \overset{h(q_j)}{\Rightarrow}_S h(q_k)$ whenever $q_i \overset{h(q_j)}{\Rightarrow}_I q_k$. Finally, since we had assumed that the access string for each entry state $q^e_m$ of $S$ was $(m, p)$ and $x_j = (m, p)$ for some $1 \leq j \leq n$, it follows that $h(q^e_m) = q^e_m$. Hence, $h(q_i) \overset{(m, p)}{\Rightarrow}_S h(q^e_m)$ whenever $q_i \overset{(m, p)}{\Rightarrow}_I q^e_m$. The following Theorem now follows.

**Theorem 15.** Let $T = T_0 \cup T_1 \cup T_2 \cup T_3$. If $I$ is consistent with $(T \cap L(S), T \setminus L(S))$, then $L(I) = L(S)$.

**Proof:** By the above observations, for any string $u \in MR$, it follows by induction on the length of $u$ that $h(q_i) \overset{u}{\Rightarrow}_S h(q_j)$ whenever $q_i \overset{u}{\Rightarrow}_I q_j$. Now Lemma 17 implies that $L(I) = L(S)$.

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

**Lemma 18.** If $S$ 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 $S$ 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 $F_S$; 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. $v = (m', p)wv'$ 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 $q \xrightarrow{w} q_k$ and $q \xrightarrow{w} q_l$, 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 $q_i \xrightarrow{r} q_k$ and $q_j \xrightarrow{r} q_l$, 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 $G_S = (V, E)$ 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 $q_i \in W_{m_0, p}$, $q_j \in W_{m_0, p}$, $q_k \in W_{m', p}$, $q_l \in W_{m', p}$, 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 $q \xrightarrow{a} q_k$ and $q \xrightarrow{a} q_l$; or

**Type 3 edge:** $p_k = p_l = p'$ and $\exists q \in W_{m', p'}$ such that $q_i \xrightarrow{a} q_k$ and $q_j \xrightarrow{a} q_l$.

Note that for every $q_i, q_j \in Q_{m_0}$ such that $q_i \in F_S$ and $q_j \notin F_S$, $(e, e)$ is a distinguishing test for $\{q_i, q_j\}$. Further, if $(u, v)$ is a distinguishing test for $\{q_k, q_l\}$ and $e = (q_i, q_j) \rightarrow (q_k, q_l)$ 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. $(u, (m', p)w'v, rv)$ is a distinguishing test for $\{q_i, q_j\}$, if $e$ is an edge of type 2 above;
3. $(u(m', p)w'v, rv)$ 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 \( G_S \) such that exactly one of \( \{q_k, q_l\} \) is in \( F_S \). Determining such a path for every pair of distinct states within each module yields a complete set of distinguishing tests \( D \). \( \Box \)

Let \( \Omega = \Sigma \cup \{x_i\}_{i=1}^n \). The following lemma is a simple corollary to Lemma 18.

**Lemma 19.** A complete set of distinguishing tests \( D \) for \( S \) can be represented as \( \binom{n}{2} \) strings in \( \Omega \), each of length \( O(n^2) \), where \( n \) is the number of states of \( S \).

### 5.4.2 Constructing access strings

Let \( \Omega \) be as defined above, and let \( \Omega = \Sigma \cup \{x_{n+1}, \ldots, x_N\} \). By Lemma 16, if \( I \) is consistent with \( (T_0^+, T_0^-) \), there is a system of \( N - n \) equations (of the form described in Lemma 16), each representable by \( O(1) \) symbols in \( \Omega \), describing the set of access strings for all states in \( I \). There are at most \( (N|\Sigma| + N^2)^{N-n} \) such systems of equations, at least one of which describes a correct set of access strings for \( I \). Assuming \( |\Sigma| \) is a constant, a set of access strings for \( I \) can be represented in \( O(n \log n + N^{2(N-n)} \log N) \) space.

### 5.5 Conclusions

Our constructions of visibly pushdown automata based on congruences can, in general, result in automata with exponentially more states than a smallest deterministic visibly pushdown automaton recognizing the language. A characterization and construction of visibly pushdown automata that are at most polynomial in the size of the smallest automaton recognizing a language is an interesting open problem. Building on our work, Chervet and Walukiewicz [17] recently introduced a promising new class of VPAs known as *Block* VPAs (BVPA), together with a more general minimization result for this class. The minimization requires that an additional parameter, namely a suitable partition the call alphabet \( \Sigma_{\text{call}} \), be fixed. Unfortunately, it remains unclear how an appropriate partition can be determined in general.
Chapter 6

Error Explanation

6.1 Introduction

State-exploration methods, such as model checking [20], are popular techniques for automated verification of software and hardware systems. One of the principal reasons for the widespread use of model checking is the ability of the model checker to produce a witness to the violation of a safety property in the form of an error trace (counter-example). While counter-examples are useful in debugging a system, the error traces can be very lengthy, and they indicate only the symptom of the error. Locating the cause of the error (or the bug) is often an onerous task even with a detailed counter-example. Recently, considerable research effort [72, 73, 40, 60, 11, 34, 33] has been directed towards automating the process of error explanation (or localizing errors or isolating error causes) to assist in the debugging process by identifying possible causes for the faulty behavior. Error explanation tools are now featured in model checkers such as SLAM [13, 11] and Java PathFinder (JPF) [16, 34].

Error explanation is an intrinsically informal process that admits many heuristic approaches which cannot be justified formally. Most current approaches to this problem rely on two broad philosophical themes for justification. First, in order to explain something (like an error), one has to identify its “causes” [50]. And second is Occam’s principle, which states that a “simpler” explanation is to be always preferred between two competing theories. The different approaches to error explanation primarily differ in what they choose to be the “causal theory” for errors. Two popular heuristics have been widely and successfully used in debugging. The first one relies on the observation that program changes which result in a system that no longer exhibits the offending error trace identify possible causes for the error [71, 67, 15, 73, 72]; in accordance with Occam’s principle, one tries to find the minimum number of changes. The second, more popular approach [60, 11, 34, 33] relies on the intuition that differences between correct and faulty runs of the system shed considerable light on the sources of errors. This approach tries to find correct runs exhibited by the system that closely match the error trace. They then infer the causes of the error from the correct executions and the given error-trace.

While algorithms for these heuristics have been developed based on sophis-
ticated use of SAT solvers and model checkers [71, 67, 73, 72, 60, 11, 34, 33], there has been very little effort to study the computational complexity of these methods. In the this chapter, we study the computational complexity of applying the above mentioned heuristics to three commonly used abstractions to model systems *. The first and least expressive model we look at is that of Finite State (Mealy) Machines (FSMs) [36], which are deterministic finite machines that produce outputs when transiting from one state to another. We consider Mealy machines and not their Moore counterparts because they are a generalization. The second model we consider are Extended Finite State Automata (EFSAs), which are finite state machines equipped with a finite number of boolean variables which are manipulated in each transition. The third model we examine is that of (nondeterministic) Pushdown Machines (PDM), which have been widely used as a model of software programs in model checking †. Once again we consider a generalization of PDMs that produce outputs.

The two approaches to error explanation yield three distinct notions of the ‘smallest distance’ between the given program and a correct program, which we now describe. The precise definitions of these distances depends on the representation used, and will be presented in Section 6.2. We investigate the complexity of computing these distances when the program is abstractly represented by one of the three computation models.

**Minimum edit set** Let $M$ be an abstract representation of the program, and let $\Sigma$ be a fixed input alphabet and $\Omega$ be a fixed output alphabet. For an input string $w \in \Sigma^*$ and an output string $\phi \in \Omega^*$ of the same length as $w$, an *edit set* is a set of transitions of $M$ that can be changed so that the resulting program $M'$ produces output $\phi$ on input $w$. A *minimum edit set* $X_M(w, \phi)$ is an edit set of smallest size. Notice that the set of states cannot be changed.

**Closest-output distance** Let $M$ be an abstract representation of the restriction of the program to correct runs. For an output string $\phi \in \Omega^*$, the *closest-output distance* $d_M(\phi)$ is the smallest Hamming distance between $\phi$ and a string $\phi' \in \Omega^*$ that can be produced by $M$.

**Closest-input distance** Let $M$ be an abstract representation of the restriction of the program to correct runs. For an input string $w$, the *closest-input distance* $d_M(w)$ is the smallest Hamming distance between $w$ and a string $w'$ that is in the language of the machine represented by $M$.

**Remark 8.** Note that the latter two distances are defined when the restriction of the program to correct runs is represented by one of the computation models

---

*We make no effort to judge the practical usefulness of these error explanation approaches. The interested reader is referred to the papers cited here for examples where these heuristics have been effective in debugging. †In software model checking the more commonplace model is actually a boolean program with a stack, which is a combination of EFSAs and PDMs. While we do not explicitly consider this model, our results have consequences for doing error explanation for such boolean programs, which we point out.
we consider. The representations we consider are commonly used to model programs, and for most correctness properties of interest (e.g., those expressed in linear temporal logic) the subset of executions of the system satisfying such properties can be expressed using the same kind of abstraction.

**Summary of results** Our results can be summarized as follows. The problem of determining the size of a minimum edit set for given input/output strings and a program represented as a Mealy machine is NP-complete. In addition, there is no polynomial time algorithm for computing an edit set whose size is within a polynomial factor of the size of a minimum edit set unless $P = NP$. A couple of points are in order about these results. First, the intractability of this error explanation method for extended finite state machines and PDMs (and boolean programs) follows as a corollary because they are generalizations of the simple FSM model. Second, since we prove these results for a deterministic model, we can also conclude the intractability of this problem for nondeterministic models, which are typically encountered in practice.

We provide a more positive answer for the second error explanation approach. When the set of correct executions of a system can be represented by traces of pushdown machines, we present a polynomial time algorithm based on dynamic programming to determine the closest-output distance for a given output string. Since finite state machines are a special case of pushdown machines, this upper bound applies to them as well. However, when the set of correct traces is represented by an extended finite state machine, the results are radically different. Not only is the problem of computing the closest-input distance NP-complete in this case, but we show that it is unlikely that polynomial factor approximation algorithms to compute the closest-input distance in polynomial time exist. Note that the typical model for programs used in model checking is boolean programs, which can be modeled by PDMs with boolean variables. Since this model is a generalization of extended finite state machines, our lower bounds imply that the second error explanation method will be intractable for boolean programs as well.

Note also that for the purposes of error explanation, we are interested not just in computing the above distance measures, but rather in computing the closest correct execution. However, the results on computing the above distances have direct consequences to error explanation. The intractability of computing the closest input distance for extended finite state machines implies that finding the closest correct trace is also intractable. Further, the dynamic programming based algorithm that we present for computing the closest-output distance for PDMs (and FSMs) can be easily modified in a standard manner, to yield not just the distance but also the closest correct trace to the given error trace.

The rest of this chapter is organized as follows. In Section 6.2 we provide the formal definitions of the problems whose complexity we investigate. Section 6.3 provides the hardness results on computing the minimum edit set. Section 6.4
provides a polynomial time algorithm to compute the closest output distance and a hardness result for the problem of computing the closest input distance. Finally, we present our conclusions in Section 6.5.

6.2 Preliminaries

As mentioned in the introduction, we are interested in the examining the complexity of two heuristics for error explanation. The first heuristic tries to find the smallest number of program transformations that eliminate the error trace. This problem is related to computing what we call a *minimum edit set* of a program, which we define formally below.

**Minimum edit set** We define a minimum edit set when the program $M$ is represented as a PDM $(Q, Q_0, \Gamma, \delta)$. Given a well-matched input $w \in \Sigma^*$ and a string $\phi \in \Omega^*$ of equal length, an *edit set* is a set $X \subseteq \delta_{\text{call}} \cup \delta_{\text{ret}} \cup \delta_{\text{int}}$ such that there is a PDM $M' = (Q, Q_0, \Gamma, \delta')$ for which $\delta \Delta \delta' = X$ (i.e., $\delta$ and $\delta'$ differ only on the set $X$) and $\phi \in M'(w)$. A *minimum edit set* $X_M(w, \phi)$ is a smallest edit set $X$.

**Remark 9.** Although we have defined the minimum edit set only for PDMs, we could easily define it for the other models we consider as well. We shall show that this problem is intractable for FSMs, which are specialized PDMs (by Remark 2). Hence the intractability result applies to PDMs as well.

The second heuristic tries to find the closest correct computation to the given error trace. This heuristic is related to computing the *closest-output distance* that we define below.

**Closest-output distance** We define the closest-output distance when the correct executions of the program are represented as a PDM $M$. Given a string $\phi \in \Omega^*$, the *closest-output distance* $d_M(\phi)$ is the smallest non-negative integer $d$ for which there is a string $\phi' \in M(w)$ for some $w \in \Sigma^*$ such that the Hamming-distance between $\phi$ and $\phi'$ is $d$.

**Remark 10.** We will present a polynomial time algorithm for the problem of computing the closest-output distance, when the correct executions are modeled as a PDM (and an FSM). Therefore, we formally define this problem for a model that has outputs as well, which is more general than a model without outputs. Again we formally define this problem only for PDMs, which is a most general model for which this upper bound applies. Also, for error explanation we would actually be interested in computing the closest correct computation and not just the distance, and we outline how this can be done in Section 6.4.
Finally, we show that applying the second heuristic when the correct traces are modeled as an extended finite state machine is difficult. We do this by showing that computing another distance measure is difficult; since we are proving a lower bound, this measure is defined for models without outputs.

**Closest-input distance** We define the closest-input distance when the correct executions of the program are represented as an EFSA $M$. Given a string $w \in \Sigma^*$, the closest-input distance $d_M(w)$ is the smallest non-negative integer $d$ for which there is a string $w' \in L(M)$ such that the Hamming-distance between $w$ and $w'$ is $d$.

### 6.3 Computing the Minimum Edit Set

#### 6.3.1 NP-completeness

Let $M = (Q, q_0, \delta, \lambda)$ be an FSM. We consider the decision version of computing the size of $X_M(w, \phi)$, i.e., given a non-negative integer $k$, decide whether or not $|X_M(w, \phi)| \leq k$. Clearly, this problem is in NP: we guess an edit set $X$ of size $k$ and guess the changes to $\delta$ to be made on the set $X$. We then verify if $M'(w) = \phi$ for the resulting FSM $M'$.

We now show that the decision version of the problem is NP-hard, even when the sizes of the input and output alphabets ($\Sigma$ and $\Omega$) are bounded by a constant. We will reduce the **Hamiltonian-Cycle** problem to our problem. Given a directed graph $G$, we construct an FSM $M$ and input/output strings $w$ and $\phi$ such that any edit set must contain a certain set of transitions of $M$.

The key idea is to show that this set of transitions is “small” if and only if $G$ has a Hamiltonian cycle.

**Reduction** The undirected **Hamiltonian-Cycle** problem for graphs with at most one edge between any pair of vertices and with degree bounded by a constant is NP-complete (see the reduction from 3-CNF in [21]). Since undirected graphs are a special case of directed graphs, **Hamiltonian-Cycle** is NP-complete for digraphs with outdegree bounded above by a constant $d$ and with at most one edge between any ordered pair of vertices. Let $G = (V, E)$ be any such digraph, where $V = \{v_1, v_2, \ldots, v_n\}$ and for every $i \in [n]$, there is a non-negative integer $m_i \leq d$ and a permutation $\pi_i$ of $(1, 2, \ldots, n)$ such that $(v_i, v_{\pi_i(k)}) \in E$ iff $k \leq m_i$.

Let $\Sigma = \{\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_d\}$ and let $\Omega = \{x, y, z\}$. We construct an $n$-state FSM $M = (Q, q_0, \delta, \lambda)$, where $Q = V$, $q_0 = v_1$ and for every $i \in [n]$:

1. $v_i \xrightarrow{\sigma_0/z} M v_i$
2. $v_i \xrightarrow{\sigma_k/x} M v_{\pi_i(k)}$ whenever $k \in [m_i]$, and
3. $v_i \xrightarrow{\sigma_k/x} M v_i$ whenever $m_i < k \leq d$. 

66
Figure 6.1. A graph $G$ and the associated FSM $M$

Figure 6.1 depicts an example graph and the FSM $M$ with $d = 2$.

We use the notation $a^k$ to denote $a$ repeated $k$ times. Let $\phi_1 = yx^{n-1}$, and for $i = 2, \ldots, n$ let $\phi_i = x^{n-i+1}yx^{i-2}$. Further, let

\[ w_d = (\sigma_0 \sigma_1 \sigma_0^d)(\sigma_0 \sigma_2 \sigma_0^d) \cdots (\sigma_0 \sigma_{d-1} \sigma_0^d) \]

\[ w = \sigma_0^k w_d (\sigma_0 w_d)^{n-1} \]

\[ \phi = \phi_1 \phi_1 (yx \phi_1)^d (yx \phi_2)^d \cdots (yx \phi_n)^d \]

**Lemma 20.** $|X_M(w, \phi)| \leq dn$ iff $G$ has a Hamiltonian cycle.

**Proof:** Suppose $G$ has a Hamiltonian cycle $C$. We obtain an FSM $M'$ satisfying $M'(w) = \phi$ by modifying $dn$ transitions of $M$ as follows:

Since $C$ is a Hamiltonian cycle in $G$, for every $v_i$ there is exactly one $v_j$ such that $(v_j, v_i) \in C$. For every such $v_i$, replace the transition $v_i \xrightarrow{\sigma_0/z} v_i$ with the transition $v_i \xrightarrow{\sigma_0/y} v_j$ (if $v_i = v_1$), and with the transition $v_i \xrightarrow{\sigma_0/x} v_j$ (if $v_i \neq v_1$). This accounts for $n$ changes.

Further, by definition of $M$, for each such $(v_j, v_i) \in C$ there is exactly one transition of the form $v_j \xrightarrow{\sigma_1/z} v_i$ in $M$, and hence $d-1$ transitions of the form $v_j \xrightarrow{\sigma_1/x} v_i$ where $v_i \neq v_1$. Replace each such transition with the transition $v_j \xrightarrow{\sigma_1/x} v_i$. This accounts for $(d-1)n$ changes. Hence, $M'$ differs from $M$ in $dn$ transitions. It is easy to verify that $M'(w) = \phi$.

Conversely, suppose $G$ is not Hamiltonian. Consider an FSM $M' = (Q, q_0, \delta', \lambda')$ such that $M'(w) = \phi$. We claim that $M'$ satisfies the following properties for every $i, j \in [n]$:

1. The edges corresponding to the $\sigma_0$-labeled transitions of $M'$ form a Hamiltonian cycle, i.e., the sequence of states reached in $M'$ from $v_i$ on input $\sigma_0, \sigma_0^2, \ldots, \sigma_0^n$ is a permutation of the set of states $Q$ and the state reached in $M'$ from $v_i$ on input $\sigma_0^n$ is $v_i$.

2. $v_i \xrightarrow{\sigma_i/x} v'_i$ for some $v'_i \in Q$, where $\alpha = y$ if $i = 1$ and $\alpha = x$ otherwise.
3. If $v_i \xrightarrow{\sigma_0/\alpha} M' v_j$ for some $\alpha \in \{x, y\}$, then $v_j \xrightarrow{\sigma_k/\beta} M' v_i$ for every $k \in [d]$.

To see this, consider the prefix $\sigma_0^{2^n}$ of $w$ and the corresponding prefix $\phi_1 \phi_1 = y x^{n-1} y x^{n-1}$ of $\phi$. For every $i \in [n]$, let $v_{\pi(i)}$ be the state reached in $M'$ from $v_1$ on input $\sigma_0^{n-1}$ (note that $\pi(1) = 1$). On reaching state $v_{\pi(i)}$, we note that the output produced by $M'$ on the subsequent input $\sigma_0^n$ is $\phi_i$. Since, for distinct $i, j$ we have $\phi_i \neq \phi_j$, $(\pi(1), \ldots, \pi(n))$ must be a permutation of $(1, \ldots, n)$. Also, the state $v$ reached in $M'$ on input $\sigma_0$ from $v_{\pi(n)}$ produces output $\phi_1$ on subsequent input $\sigma_0^n$. Since $M'$ has $n$ states, $v$ must in fact be $v_1$. In other words, we have the following cycle in $M'$ obtained by following the edges labeled by $\sigma_0$: $v_1 \xrightarrow{\sigma_0/\alpha} M' v_{\pi(2)} \xrightarrow{\sigma_0/\beta} M' v_{\pi(3)} \xrightarrow{\sigma_0/\gamma} M' \ldots \xrightarrow{\sigma_0/\delta} M' v_{\pi(n)} \xrightarrow{\sigma_0/\epsilon} M' v_1$. Hence $M'$ satisfies the first two properties.

As observed above, from each state $v_{\pi(i)}$, $M'$ produces the unique output $\phi_i$ on input $\sigma_0^n$ and returns to state $v_{\pi(i)}$. Hence the output produced by $M'$ on input $\sigma_0^n$ can be used to identify the current state in $M'$. The string $w_d$ defined above consists of $d$ substrings of the form $\sigma_0 \sigma_k \sigma_0^n$, where $k \in [d]$. Since $M'(w) = \phi$, it follows that for every $i \in [n]$ and every $k \in [d]$, $M'$ goes from state $v_{\pi(i)}$ to $v_{\pi(i)}$ on input $\sigma_0 \sigma_k$ with output of the form $\alpha x$ (where $\alpha \in \{x, y\}$).

Hence, $M'$ also satisfies the third property. For the example presented in Figure 6.1, a solution FSM $M'$ must be as depicted in Figure 6.2.

Since $M'$ satisfies the first and third properties, for every $i \in [n]$ and every $k \in [d]$: $v_{\pi(i+1)} \xrightarrow{\sigma_k/\epsilon} M' v_{\pi(i)}$ (where $\pi(n + 1) = \pi(1)$). Now $G = (V, E)$ is not Hamiltonian, so there is at least one $j \in \{1, \ldots, n\}$ such that $(v_{\pi(j+1)}, v_{\pi(j)}) \notin E$. Thus, there is no transition of the form $v_{\pi(j+1)} \xrightarrow{\sigma_k/\epsilon} M v_{\pi(j)}$ in $M$. Also, for
every $i \neq j$, there is at most one transition of the form $v_{\pi(i+1)} \xrightarrow{\sigma_i} v_{\pi(i)}$ in $M$. Hence, $M'$ differs from $M$ in at least $d + (d - 1)(n - 1)$ transitions.

Furthermore, since $M'$ satisfies the second property, it clearly differs from $M$ on the $n$ $\sigma_0$-labeled transitions. Thus, $M'$ differs from $M$ in at least $dn + 1$ transitions. Hence, $|X_M(w, \phi)| > dn$. \hfill \square

This completes the proof of NP-completeness for the decision version of computing $|X_M(w, \phi)|$ when $M$ is an FSM.

**Remark 11.** Note that FSMs are restricted versions of EFSAs (as observed in Remark 1) and PDMs. It is easy to show that the decision version of computing $|X_M(w, \phi)|$ is also in NP when $M$ is an EFSA or a PDM. Hence, the decision version of computing $|X_M(w, \phi)|$ is also NP-complete for these models.

### 6.3.2 Inapproximability Result

Given an FSM $M = (Q, q_0, \delta, \lambda)$ and equal length strings $w \in \Sigma^*$ and $\phi \in \Omega^*$, we prove that if the minimum edit set has size $d$, then for every positive constant $k$ there is no polynomial time algorithm to construct an FSM $M' = (Q, q_0, \delta', \lambda')$ such that $M'(w) = \phi$ and $(\delta, \lambda)$ and $(\delta', \lambda')$ differ on a set of size at most $d^k$, unless P = NP.

Our proof is in three steps. We first prove a variant of a result by Pitt and Warmuth [39]: Given a positive integer $k$ and a set of input/output pairs of strings $P$ for which the smallest FSM consistent with $P$ (i.e., an FSM which produces output $\phi$ on input $w$ for every pair $(w, \phi) \in P$) has $n$ states, there is no efficient algorithm to construct an FSM consistent with $P$ having at most $n^k$ states (unless P = NP). Next, given such a set of pairs $P$, we carefully construct an FSM $M$ and a single input/output pair $(w, \phi)$ such that the minimum edit set $X_M(w, \phi)$ has size $\Theta(n^k)$. Finally, we show that any edit set $X$ can be efficiently modified to yield an FSM with $|X|$ states that is consistent with $P$. We put these three results together to complete our proof.

**Definition 9.** A deterministic finite state automaton (FSA) $M = (Q, q_0, \delta, F)$ is the automaton variant of an FSM, where $F \subseteq Q$ is the set of final (accepting) states. Given a finite set $\Sigma$ and two finite sets $\text{POS}, \text{NEG} \subseteq \Sigma^*$, an FSA $M$ is said to be consistent with $(\text{POS}, \text{NEG})$ if $M$ accepts all strings in $\text{POS}$ and rejects all strings in $\text{NEG}$.

Under the assumption that P $\neq$ NP, Pitt and Warmuth [39] prove the following inapproximability result for the minimum consistent FSA problem.

**Theorem 16 (Pitt-Warmuth).** Given a finite set $\Sigma$ such that $|\Sigma| \geq 2$, and two finite sets of strings $\text{POS}, \text{NEG} \subseteq \Sigma^*$, for any positive integer $k$ there is no polynomial time algorithm to find an FSA with at most $n^k$ states, where $n$ is the number of states in the smallest FSA that is consistent with $(\text{POS}, \text{NEG})$ unless P = NP.
Using the result above we prove the following:

**Lemma 21.** Given a finite set of input/output pairs \( P \) in \( \Sigma^* \times \Omega^* \), let \( n \) be the minimum number such that there is an FSM \( M \) with \( n \) states that is consistent with \( P \) (i.e., for every pair \((w, \phi) \in P, M(w) = \phi\)). Then, assuming \( P \neq \text{NP} \), for any positive constant \( k \), there is no polynomial-time algorithm to find a consistent FSM \( M' \) with at most \( n^k \) states. This result holds even if \(|\Sigma|\) and \(|\Omega|\) are bounded by suitable constants.

**Proof:** We reduce the minimum consistent FSA problem to the above problem. Consider an instance of the minimum consistent FSA problem with input alphabet \( \Sigma \) and sets \( \text{POS}, \text{NEG} \subseteq \Sigma^* \). Let \( \Sigma' = \Sigma \cup \{c\} \) where \( c \notin \Sigma \) and \( \Omega = \{0, 1, 2\} \). Consider the set of input output pairs \( P = P^+ \cup P^- \), where

\[
P^+ = \{(w, \phi) \mid w = u'c, w' \in \text{POS and } \phi = 2^{w|1|}\}
\]

\[
P^- = \{(w, \phi) \mid w = u'c, w' \in \text{NEG and } \phi = 2^{w|0|}\}
\]

If there is an FSM \( M \) over input alphabet \( \Sigma' \) with \( n \) states consistent with the set of pairs in \( P \) then we can construct a FSA \( M' \) over input alphabet \( \Sigma \) with \( n \) states, where we neglect the output symbols and label states accept or reject by the output produced on the input \( c \) from that state. Clearly this FSA is consistent with \( (\text{POS,NEG}) \). Conversely, if there is a FSA consistent with \( (\text{POS,NEG}) \) then we can produce an FSM \( M \) over input alphabet \( \Sigma' \) that is consistent with \( P \) as follows: the output on all inputs in \( \Sigma \) is 2, and the transition from every state on input \( c \) leads to the same state, with output 1 if the state is an accepting state of the FSA and with output 0 otherwise. Clearly the FSA produced is consistent with \( P \).

\( \square \)

For the rest of this section, we fix an input alphabet \( \Sigma \) (with \(|\Sigma| = c \), a constant), a finite output alphabet \( \Omega \), and a finite set \( P = \{(w_1, \phi_1), \ldots, (w_k, \phi_k)\} \) of input/output pairs of strings over \( \Sigma^* \times \Omega^* \), with \(|w_i| = |\phi_i|\) for all \( i \in [k] \). A necessary and sufficient condition for the existence of an FSM consistent with the pairs in \( P \) is the following: For any \( w \in \Sigma^* \), if \( w \) is a prefix of both \( w_i \) and \( w_j \), then the prefixes of length \(|w|\) of \( \phi_i \) and \( \phi_j \) must be identical. If this condition is satisfied, it is easy to construct an FSM consistent with \( P \) which has at most \( m = (\sum_{i=1}^{k} |w_i|) + 1 \) states. As a corollary, if the smallest FSM consistent with \( P \) has \( n \) states, then \( n \leq m \).

Let \( a_0, a_1 \notin \Sigma \) and let \( \alpha_1, \alpha_2, \alpha_3 \notin \Omega \). Let \( \Sigma' = \Sigma \cup \{a_0, a_1\}, \Omega' = \Omega \cup \{\alpha_1, \alpha_2, \alpha_3\} \) and \( Q = \{q_1, q_2, \ldots, q_m\} \). We construct an FSM \( M = (Q, q_1, \delta, \lambda) \) over input alphabet \( \Sigma' \) and output alphabet \( \Omega' \) such that for every \( q \in Q \), \( q \overset{a_i}{\longrightarrow} \Phi \) \( q_i \) and for every \( a \in \Sigma \), \( q \overset{a_0}{\longrightarrow} M q_i \) and further, for some fixed permutation \( \pi \) of \((1, 2, \ldots, m)\) such that \( \pi(1) = 1 \),

\[
q_{\pi(1)} \overset{a_0/\alpha_1}{\longrightarrow} M q_{\pi(2)} \overset{a_0/\alpha_1}{\longrightarrow} M q_{\pi(3)} \overset{a_0/\alpha_1}{\longrightarrow} \cdots \overset{a_0/\alpha_1}{\longrightarrow} M q_{\pi(n)} \overset{a_0/\alpha_1}{\longrightarrow} M q_{\pi(1)}
\]

i.e., the transitions on input \( a_0 \) form a Hamiltonian cycle, with output \( \alpha_1 \) for every transition other than the transition from the initial state \( q_1 = q_{\pi(1)} \), for
which the output is $\alpha_2$. Let $\phi'_i = \alpha_2^{\alpha_1^{m-1}}$ and for every $i = 2, \ldots, m$ let $\phi'_i = \alpha_2^{\alpha_1^{m-1} \alpha_2 \alpha_1^{i-2}}$. Consider

$$w = \alpha_2^{m} [(\alpha_2^{\alpha_1^{m-1}})(\alpha_2^{\alpha_1^{m-1}})(\alpha_2^{\alpha_1^{m-1}}) \cdots (\alpha_2^{\alpha_1^{m-1}})](w_1 a_1)(w_2 a_1) \cdots (w_k a_1)$$

$$\phi = \phi'_1 (\alpha_2 \alpha_1 \phi'_1)(\alpha_2 \alpha_1 \alpha_1 \phi'_1) \cdots (\alpha_2 \alpha_1 \alpha_1 \cdots \alpha_1 \alpha_1 \phi'_1)$$

We claim that $|X_M(w, \phi)|$ is $\Theta(n)$, where $n$ is the number of states in the smallest FSM consistent with $P$. The proof of this claim follows from the following two lemmas:

**Lemma 22.** $|X_M(w, \phi)| \leq cn$.

**Proof:** Notice that $\alpha_2^{2m}$ is a prefix of $w$ and the corresponding prefix of $\phi$ is $\phi'_1 \phi'_1 = \alpha_2^{\alpha_1^{m-1} \alpha_2 \alpha_1^{m-1}}$. By an argument similar to the one used in the reduction of Subsection 6.3.1, we can prove that any FSM $M' = (Q, q_1, \delta', \lambda')$ such that $M'(w) = \phi$ must have the structure of a Hamiltonian cycle on the input $a_0$, the output on $a_0$ is $\alpha_2$ from the initial state $q_1$, and $\alpha_1$ from every other state; furthermore, all transitions of $M'$ on input $a_1$ from any state must go to the initial state $q_1$ with output $\alpha_1$. Notice that these properties are true of $M$. Hence, $M$ only needs to be modified so that it produces output $\phi_i$ on input $w_i$ for each $i \in [k]$.

As remarked earlier, $m \geq n$. Since there is an $n$-state FSM consistent with $P$, it is possible to select an $n$-state subset $V$ of $Q$ and modify only the $\Sigma$-transitions from states in $V$ to obtain an FSM $M'$ such that $M'(w_i) = \phi_i$ for every $i \in [k]$. It now immediately follows that $M'(w) = \phi$. Hence, $|X_M(w, \phi)| \leq |\Sigma| n = cn$. □

**Lemma 23.** $|X_M(w, \phi)| \geq n$.

**Proof:** As remarked earlier, $m \geq n$. Let $|X_M(w, \phi)| = d$ and suppose we have made $d$ changes to $M$, yielding $M'$ such that $M'(w) = \phi$ and hence, for every $i \in [k]$, $M'(w_i) = \phi_i$. Thus, there are at most $d$ states $q$ for which at least one transition of the form $q \xrightarrow{a/\alpha_3} M' q$ ($\sigma \in \Sigma$) has been changed. Hence, there are at least $m - d$ states in $M'$ such that $q \xrightarrow{a/\alpha_3} M' q$ for every $\sigma \in \Sigma$. We claim that we can discard all such states $q$ and modify the resulting FSM to obtain an FSM over input alphabet $\Sigma$ and output alphabet $\Omega$ that is consistent with $P$.

Consider any state $q$ such that $q \xrightarrow{a/\alpha_3} M' q$ for every $a \in \Sigma$. Thus on an input $w_i$, $i \in [k]$, if we can ever reach $q$, it must be the last state, since the output from $q$ on any input symbol in $\Sigma$ is $\alpha_3$ and $\phi_i$ does not contain $\alpha_3$. Clearly $q$ cannot be the start state of $M'$. So $q$ can be discarded from $M'$ and it is still possible to construct an FSM $M''$ such that $M''(w_i) = \phi_i$ for every $i \in [k]$ (all edges into $q$ can be sent to some other state).

This process can be repeated for all such states $q$, resulting in an FSM $\hat{M}$ with at most $d$ states such that $\hat{M}(w_i) = \phi_i$ for every $i \in [k]$. Now, by discarding all $a_0$ and $a_1$ transitions from $\hat{M}$, we obtain an FSM over input alphabet $\Sigma$ and
output alphabet \( \Omega \) that is consistent with \( P \), which has \( d = |X_M(w, \phi)| \) states. Hence, \( |X_M(w, \phi)| \geq n \).

Recalling that \( c = |\Sigma| \) is a constant, we conclude that \( |X_M(w, \phi)| \) is \( \Theta(n) \).

Suppose there is a positive integer \( k \) and a polynomial time algorithm to compute an FSM \( M' \) that differs from \( M \) in at most \( |X_M(w, \phi)|^k \) transitions for which \( M'(w) = \phi \). Using an argument similar to the one in Lemma 23, we can modify \( M' \) in polynomial time by discarding at least \( m - |X_M(w, \phi)|^k \) states and the \( a_0 \) and \( a_1 \) transitions to obtain a \( \Theta(n^k) \)-state FSM with input alphabet \( \Sigma \) and output alphabet \( \Omega \) that is consistent with \( P \). By Lemma 21, this would imply that \( P = \text{NP} \). Thus we have the following

**Theorem 17.** Given an FSM \( M \) and equal length strings \( w \in \Sigma^* \) and \( \phi \in \Omega^* \), for any positive integer \( k \) there is no polynomial time algorithm to compute an edit set of size \( |X_M(w, \phi)|^k \) unless \( P = \text{NP} \).

**Remark 12.** By observing that FSMs are restricted versions of EFSAs (Remark 1) and PDMs (Remark 2), we obtain similar inapproximability results when the program is represented using either of these abstractions.

### 6.4 Computing the Closest-output distance

#### 6.4.1 Upper bound for FSMs and PDMs

We will prove a polynomial upper bound on the time to compute the closest-output distance when the correct executions of a program are represented as a PDM. The closest-output distance can be expressed using two simple recurrences, and our algorithm uses a straightforward dynamic programming approach to solve these recurrences.

Let \( \phi \in \Omega^* \) and let \( M = (Q, Q_0, \Gamma, \delta) \) be a PDM. We consider the problem of computing \( d_M(\phi) \). Let the length of \( \phi \), denoted by \(|\phi|\), be \( L \). For every \( i, j \in [L], i < j \), let \( \phi(i, j) \) denote the substring of \( \phi \) from the \( i \)-th to the \( j \)-th position and let \( \phi(i) \) denote the \( i \)-th letter of \( \phi \) (i.e., \( \phi(i) = \phi(i, i) \)).

For every \( q \in Q \) and every \( i \in [L] \), let \( P(q,i) \) denote the Hamming distance of the closest string to \( \phi(i,L) \) that can be produced by \( M \) on a some valid input \( w \) starting from state \( q \). Also, for every \( q, q' \in Q \) and every \( 1 \leq i \leq j \leq L \), let \( B(q,q',i,j) \) denote the Hamming distance of the closest string to \( \phi(i,j) \) that can be produced by \( M \) on some well-matched input \( w \) such that the state changes from \( q \) to \( q' \). By definition, \( d_M(\phi) = \min_{q_0 \in Q_0} P(q_0,1) \).

Let \( [\alpha_1 \neq \alpha_2] \) be 1 if \( \alpha_1 \neq \alpha_2 \) and 0 otherwise. For notational convenience, let \( P(q,i) = 0 \) if \( i > L \) and let \( B(q,q',i+1,i) = 0 \) if \( q = q' \), and \( B(q,q',i+1,i) = \infty \) otherwise. We observe that if \( w \) is a well-matched string from state \( q \), then \( w \) must be of one of the following two forms:

1. \( w = aw_1 \) where \( (q,a,q_1,\alpha) \in \delta_{\text{int}} \) for some \( q_1 \in Q \) and \( \alpha \in \Omega \), and \( w_1 \) is a well-matched string from \( q_1 \);
2. \( w = aw_1 ...w_{k-1}ax_1x_2...x_{k-1}w_{k} \) where \( x_1, ..., x_{k-1} \in \Sigma \), and \( x_{k-1} \neq \epsilon \), and the states for the \( i \)-th part of the string are given by \( \delta_{\text{int}}(q,a,q_{i+1},\alpha_{i+1}) \) for every \( q_1, \alpha_1 \) such that \( (q,a,q_{i},\alpha_{i}) \in \delta_{\text{int}} \) and \( (q_{i+1},\alpha_{i+1}) \in \delta_{\text{int}}(q_1,\alpha_{i}) \).
2.  \( w = aw_1a'w_2 \) where \( (q, a, q_1, \gamma, \alpha) \in \delta_{\text{call}}, q_1 \overset{w_1}{\xrightarrow{\gamma}} q_2 \) and \((q_2, a', \gamma, q_3, \alpha') \in \delta_{\text{ret}} \) for some \( q_1, q_2, q_3 \in Q, \gamma \in \Gamma, \alpha, \alpha' \in \Omega \) and \( \phi' \in \Omega^* \), and \( w_1 \) is a well-matched string from \( q_1 \) and \( w_2 \) is a well-matched string from \( q_2 \).

Note that \( w \) can be of the latter form only if \(|w| \geq 2 \). Thus, for every \( 1 \leq i \leq j \leq L, B(q, q', i, j) = b_1 \) if \( j - i < 2 \) and \( B(q, q', i, j) = \min(b_1, b_2) \) otherwise, where

\[
b_1 = \min_{\alpha \neq \phi(i)} \alpha + B(q_1, q_2, i + 1, j)
\]
minimum over all \( a, q_1, \alpha \) such that \((q, a, q_1, \alpha) \in \delta_{\text{min}} \)

\[
b_2 = \min_{\alpha \neq \phi(i)} \alpha + B(q_1, q_2, i + 1, k) + |\alpha' \neq \phi(k+1)| + B(q_3, q_4, k + 2, j)
\]
minimum over all \( \alpha, q_1, q_2, \alpha', q_3, \gamma, k \) such that \( i < k < j \) and \((q, a, q_1, \alpha, \gamma) \in \delta_{\text{call}} \), \((q_2, a', \gamma, \alpha') \in \delta_{\text{ret}} \) for some \( a, a' \in \Sigma, \gamma \in \Gamma', q_1, q_2, q_3 \in Q \)

We also observe that any valid input \( w \) from state \( q \) must be of the following three forms:

1.  \( w = aw_1 \) where \( (a, q_1, \gamma, \alpha) \in \delta_{\text{call}} \) for some \( a \in \Sigma, q_1 \in Q, \gamma \in \Gamma', \alpha \in \Omega \), and \( w_1 \) is a valid input from \( q_1 \); or

2.  \( w = w_1w_2 \) where \( w_1 \) is a well-matched string from \( q, q \overset{w_1}{\xrightarrow{\phi'}} q_1 \) for some \( \phi' \in \Omega^* \) and \( q_1 \in Q \), and \( w_2 \) is a valid input from \( q_1 \); or

3.  \( w \) is a well-matched string from \( q \).

Thus, for every \( i \in [L], P(q, i) = \min(p_1, p_2, p_3) \) where

\[
p_1 = \min_{a \in \Sigma, q \in Q, \gamma \in \Gamma, \alpha \in \Omega} \{ [\alpha \neq \phi(i)] + P(q_1, i + 1) \} \quad \text{for} \quad \delta_{\text{call}}
\]

\[
p_2 = \min_{i \leq k \leq L, q_1, \alpha \in \Omega} B(q_1, q_1, i, k) + P(q_1, k + 1)
\]

\[
p_3 = \min_{q \in V} B(q', q_1, i, L)
\]

The required value \( \min_{q_0 \in Q_0} P(q_0, 1) \) can easily be computed using dynamic programming in \( O(|\Sigma|^2 \cdot |\Gamma| \cdot |V| \cdot L^3) \) time, which is polynomial in the size of the input. By observing that an FSM is a special case of a PDM, we obtain the following

**Theorem 18.** There is a polynomial time algorithm for computing the closest executions of the program are represented as a PDM or an FSM.

**Remark 13.** Note that the polynomial-time dynamic programming algorithm to compute \( d_M(\phi) \) can easily be modified to compute a string \( w \in \Sigma^* \) such that the Hamming distance between \( \phi \) and some string \( \phi' \in M(w) \) is \( d_M(\phi) \).

**Remark 14.** Most programs are infinite-state systems and need to be abstracted to form PDMs (or extended finite state machines); consequently these abstractions describe more than just the correct executions of the program. Hence, the input string computed by the above dynamic programming algorithm may not be valid, i.e., it may not correspond to a correct execution of the program. While we do not know of a general technique to compute the closest-output distance when inputs are constrained to be valid, the following technique
can be used in practice: Using the procedure by Lawler [46], the above dynamic programming algorithm can be used to compute input strings corresponding to the closest-output distance, the second-closest-output distance, ..., the k-th closest-output distance, in time polynomial in k and the size of the input. These input strings can be examined in order, and the first valid input string can then be chosen.

6.4.2 Hardness results for EFSA representation

Given an EFSA $M$ and a string $w \in \Sigma^*$, we show that the decision version of computing $d_M(w)$ is NP-complete and further, there is no polynomial-time algorithm to compute $d_M(w)$ to within any given polynomial factor (unless $P = NP$). Briefly, given a SAT formula $\psi$ over a set $X$ of $n$ variables, we construct an EFSA $M$ and a string $w \in \Sigma^*$ such that every string $w' \in L_M$ identifies an assignment $v_{w'}$ of the variables in $X$, and if $w'$ is “close” to $w$, then $v_{w'}$ satisfies $\psi$. The results follow from the NP-completeness of SAT.

Let $\Sigma = \{0, 1\}$, let $X$ be a set of $n$ boolean variables, and let $\psi$ be any SAT formula over variables in $X$. For every positive integer $k$, let $N(k) = n^k + 1$ and construct the following EFSA $M_k = (Q, q_0, F, v_0, g, \delta)$:

1. $Q = \{\emptyset, q_1, \ldots, q_n\} \cup \{q_1', \ldots, q_{N(k)}\} \cup \{q_1'', \ldots, q_{N(k)}''\}$, $F = \{q_{N(k)}, q_{N(k)}''\}$;
2. $g(q_i, j) = \top$ for $i = 0, \ldots, n$ and $j = 0, 1$;
3. $g(q_i', j) = \neg \psi$ and $g(q_i'', j) = \psi$ for $i \in [N(k)]$ and $j = 0, 1$;
4. $(q_{i-1}, 0) \ x_i:=0 \ q_i$ and $(q_{i-1}, 1) \ x_i:=1 \ q_i$ for $i \in [n]$;
5. $(q_n, 0) \rightarrow q_i''$ and $(q_n, 1) \rightarrow q_i'$;
6. $(q_i', 0) \rightarrow q_{i+1}'', (q_i'', 1) \rightarrow q_i'$, $(q_i, 0) \rightarrow q_i', (q_i, 1) \rightarrow q_{i+1}'$ for $i \in [N(k) - 1]$;
7. $(q_{N(k)}, j) \rightarrow q_{N(k)}''$ and $(q_{N(k)}', j) \rightarrow q_{N(k)}'$ for $j = 0, 1$;
8. the initial state is $q_0$, and the initial assignment $v_0$ sets all variables to 0.
Figure 6.3 shows the EFSA for \( n = 2 \) variables. The transition arrows are labeled by the input symbol 0 or 1. The change in assignment (if any) is given after the input symbol while the enabling condition is given before the input symbol if it is not always true.

Note that the size of \( M_k \) is polynomial in the size of the inputs. Let \( w = 0^{n+\lceil N(k) \rceil} \). By the construction of \( M_k \), every \( w' \in L(M_k) \) of length \( n + N(k) \) is either of the form \( w''0^{N(k)} \) or \( w''1^{N(k)} \) where \( w'' \in \Sigma^n \). Note that \( w'' \) uniquely determines an assignment \( v_{w''} \) of the variables of \( X \). It is immediately clear from the construction of \( M_k \) that the Hamming distance between \( w \) and \( w' \) is at most \( n \) if \( v_{w''} \) satisfies \( \psi \), and is at least \( N(k) \) otherwise.

The decision problem \( d_{M_k}(w) \leq n \) is clearly in NP: we guess a string \( w' \) of length equal to \( w \) and verify in polynomial time if \( w' \in L(M_k) \) and the Hamming distance between \( w' \) and \( w \) is at most \( n \). If so, then as argued above, the prefix \( w'' \) of length \( n \) of \( w' \) uniquely determines a satisfying assignment to the SAT formula \( \psi \). Since \( \psi \) was an arbitrary SAT formula, we conclude that this decision problem is NP-complete.

We now show that, unless \( P = NP \), there is no polynomial time algorithm for computing \( d_{M_k}(w) \) to within a polynomial approximation factor. Suppose, for the sake of contradiction, that such an algorithm \( A \) exists. Let \( A(M_k, w) \) denote the output of \( A \) for the input \( M_k \) and \( w \) defined above. Since \( d_{M_k}(w) \leq n \) iff \( \psi \) is satisfiable, it follows that \( A(M_k, w) < N(k) \) iff \( \psi \) is satisfiable. Thus, no such polynomial-time algorithm \( A \) exists unless \( SAT \in P \). Hence, we have the following

**Theorem 19.** Let \( M \) be an EFSA and let \( w \in \Sigma^* \) such that \( d_M(w) = d \). For any positive integer \( k \), there is no polynomial time algorithm to decide whether or not \( d_M(w) \leq \lceil k \rceil \), unless \( P = NP \).

**Remark 15.** The typical model for programs used in model checking is boolean programs, which can be seen as PDMs with boolean variables. Since this model is a generalization of extended finite state machines, our hardness result applies to boolean programs as well.

**Remark 16.** As mentioned in Remark 14, programs may be abstracted as extended finite state machines, and thereby describe more than just the correct executions of the program. The above hardness result for extended FSMs clearly extends to this more general case as well.

### 6.5 Conclusions

We have proved upper and lower bounds for two popular heuristics used in automated error explanation for various models encountered in formal verification. Based on our observations, we can draw two important conclusions. First, our lower bounds provide justification for algorithms based on SAT solvers that
have been proposed in the literature. These algorithms are likely to be the most efficient that one can hope to design. Second, since the problem of determining the minimum edit set is intractable even for Mealy machines, it is unlikely that error explanation tools based on this heuristic will scale up to large software programs. On the other hand, the closest correct trace to a counter-example can be computed efficiently for PDMs (and hence for finite state models as well). The intractability of this problem for extended finite state machines is a consequence of the well-known state space explosion problem, and does not seem intrinsic to the heuristic itself.
References


79


Author’s Biography

Viraj Kumar received his B.S. in Mathematics from St. Stephen’s College, Delhi, India in 1998. He completed an M.S. in Applied Statistics and Informatics from the Indian Institute of Technology Bombay, India in 2000, and an M.S. in Computer Science from the University of Illinois at Urbana-Champaign in 2004. At the University of Illinois at Urbana-Champaign, he was twice ranked on the *Incomplete Lists of Teachers Ranked as Excellent*, and was awarded the *Outstanding Teaching Assistant Award* by the Department of Computer Science in 2007.