Model Checking Open Probabilistic Systems
Using Hierarchical Probabilistic Automata

BY
Yue Ben
B.S., Tsinghua University, Beijing, China, 2010
M.Eng., Tsinghua University, Beijing, China, 2012

THESIS
Submitted as 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 Chicago, 2016

Chicago, Illinois

Defense Committee:
A. Prasad Sistla, Chair and Advisor
Ugo Buy
Venkat Venkatakrishnan
Milos Zefran, Electrical and Computer Engineering
Mahesh Viswanathan, University of Illinois at Urbana-Champaign
ACKNOWLEDGMENT

First of all, I would like to express my deepest respect and gratitude to my advisor, Professor A. Prasad Sistla, for his countless guidance, support and advice throughout my PhD study. He is an intellectual, scholarly and wise advisor. He has been like a father to me, not only showing me how to acquire scientific knowledge and research skills, but also sharing valuable life experience. He is better than the best advisor I could ever imagine.

Secondly, I am highly indebted to our coworkers - Professor Mahesh Viswanathan and Professor Rohit Chadha. Without their diligent work and precious contribution we would not be able to make current research progress.

Thirdly, I would like to express thanks to my committee members - Professor Ugo Buy, Professor Venkat Venkatakrishnan, and Professor Milos Zefran, Professor Mahesh Viswanathan again - for their precious suggestions on my research and support in my defense.

Next, I would like to express thanks to my family. With my husband Xing by my side, I always have home and support; with my family missing me from far away, I will never get lost.

My appreciations also go to other people who have helped me in the past few years, including many faculties and staff in our department.

Finally, our research covered in this thesis was funded by the NSF grants 1319754 and 1138277.

YB
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2 BACKGROUND</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Concurrent Systems</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Model Checking Probabilistic Systems</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Reliability v.s. Robustness</td>
<td>7</td>
</tr>
<tr>
<td>3 PRELIMINARIES</td>
<td>8</td>
</tr>
<tr>
<td>3.1 Probabilistic Automata</td>
<td>8</td>
</tr>
<tr>
<td>3.1.1 Underlying Directed Graph of PA</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Hierarchical Probabilistic Automata</td>
<td>11</td>
</tr>
<tr>
<td>3.3 Witness Sets</td>
<td>12</td>
</tr>
<tr>
<td>3.3.1 Backward Traversal</td>
<td>13</td>
</tr>
<tr>
<td>4 VERIFICATION ALGORITHMS</td>
<td>15</td>
</tr>
<tr>
<td>4.1 Backward Algorithm</td>
<td>15</td>
</tr>
<tr>
<td>4.1.1 Big O Analysis of Backward Algorithm</td>
<td>17</td>
</tr>
<tr>
<td>4.2 Forward Algorithm</td>
<td>17</td>
</tr>
<tr>
<td>4.2.1 Big O Analysis of Forward Algorithm</td>
<td>22</td>
</tr>
<tr>
<td>4.3 Forward vs. Backward</td>
<td>23</td>
</tr>
<tr>
<td>4.4 Robustness</td>
<td>24</td>
</tr>
<tr>
<td>4.5 Verifying Acyclic HPA</td>
<td>26</td>
</tr>
<tr>
<td>4.5.1 NP-Completeness</td>
<td>26</td>
</tr>
<tr>
<td>4.5.2 Other EXPTIME Verification Algorithms</td>
<td>28</td>
</tr>
<tr>
<td>4.5.2.1 Verification By Construction</td>
<td>28</td>
</tr>
<tr>
<td>4.5.2.2 Verification By Language</td>
<td>31</td>
</tr>
<tr>
<td>5 IMPLEMENTATION</td>
<td>32</td>
</tr>
<tr>
<td>5.1 PA Validation</td>
<td>32</td>
</tr>
<tr>
<td>5.2 PA Reduction</td>
<td>33</td>
</tr>
<tr>
<td>5.3 HPA validation</td>
<td>33</td>
</tr>
<tr>
<td>5.4 HPA Leveling</td>
<td>34</td>
</tr>
<tr>
<td>5.4.1 Deciding Minimum k for HPA</td>
<td>35</td>
</tr>
<tr>
<td>5.4.1.1 Implementation</td>
<td>36</td>
</tr>
<tr>
<td>5.4.2 Assigning Highest Levels</td>
<td>37</td>
</tr>
<tr>
<td>5.4.3 Assigning Lower Levels</td>
<td>38</td>
</tr>
<tr>
<td>5.5 Obtaining Witness Sets for Finite Acceptance</td>
<td>39</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>6 MODEL CHECKING 1-HPBA</strong></td>
<td></td>
</tr>
<tr>
<td>6.1 Observations on 1-HPBA</td>
<td>41</td>
</tr>
<tr>
<td>6.2 Flag Construction</td>
<td>43</td>
</tr>
<tr>
<td>6.2.1 Flag Construction Using Vectors</td>
<td>43</td>
</tr>
<tr>
<td>6.2.2 Flag Construction Using Sets</td>
<td>45</td>
</tr>
<tr>
<td>6.3 Framework to Find Candidate Witness Sets</td>
<td>48</td>
</tr>
<tr>
<td>6.3.1 Bottom Up Method</td>
<td>48</td>
</tr>
<tr>
<td>6.3.2 Top Down Method</td>
<td>50</td>
</tr>
<tr>
<td>6.4 Summary on Model Checking 1-HPBA</td>
<td>51</td>
</tr>
<tr>
<td><strong>7 MODEL ABSTRACTION AND DOMAIN APPLICATIONS</strong></td>
<td></td>
</tr>
<tr>
<td>7.1 Problem Modeling</td>
<td>52</td>
</tr>
<tr>
<td>7.2 Exception Handling Block</td>
<td>56</td>
</tr>
<tr>
<td>7.3 Domain Applications</td>
<td>57</td>
</tr>
<tr>
<td>7.3.1 Web Applications</td>
<td>58</td>
</tr>
<tr>
<td>7.3.2 Java Exception Handling</td>
<td>59</td>
</tr>
<tr>
<td>7.3.3 Miscellaneous Applications</td>
<td>62</td>
</tr>
<tr>
<td><strong>8 HIPAM - HIERARCHICAL PROBABILISTIC AUTOMATA MODEL CHECKER</strong></td>
<td>64</td>
</tr>
<tr>
<td>8.1 HPA Model Abstraction</td>
<td>66</td>
</tr>
<tr>
<td>8.1.1 HPA Generation in HiPAM</td>
<td>66</td>
</tr>
<tr>
<td>8.1.2 HPA Generation from PRISM Output</td>
<td>72</td>
</tr>
<tr>
<td>8.2 HPA Analysis and Verification</td>
<td>73</td>
</tr>
<tr>
<td>8.3 Experiments</td>
<td>74</td>
</tr>
<tr>
<td>8.3.1 Experiment Limitations</td>
<td>76</td>
</tr>
<tr>
<td><strong>9 EXTRACTING FAILURE SPECIFICATIONS</strong></td>
<td>78</td>
</tr>
<tr>
<td>9.1 Process and Object Modeling</td>
<td>79</td>
</tr>
<tr>
<td>9.1.1 Process Modeling</td>
<td>79</td>
</tr>
<tr>
<td>9.1.2 Object Modeling</td>
<td>82</td>
</tr>
<tr>
<td>9.2 Extracting and Applying Failure Specification</td>
<td>84</td>
</tr>
<tr>
<td>9.3 From Process Models to Probabilistic Automata</td>
<td>86</td>
</tr>
<tr>
<td>9.3.1 Transformation Rules</td>
<td>87</td>
</tr>
<tr>
<td>9.3.2 State Space Analysis</td>
<td>89</td>
</tr>
<tr>
<td>9.4 Summary on Extracting Failure Specifications</td>
<td>89</td>
</tr>
<tr>
<td><strong>10 UNDECIDABILITY FOR 2-HPA</strong></td>
<td>91</td>
</tr>
<tr>
<td><strong>11 CONCLUSION</strong></td>
<td>101</td>
</tr>
<tr>
<td><strong>CITED LITERATURE</strong></td>
<td>102</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>PAGE</td>
</tr>
<tr>
<td>--------------</td>
<td>------</td>
</tr>
<tr>
<td>VITA</td>
<td>106</td>
</tr>
<tr>
<td>APPENDICES</td>
<td>107</td>
</tr>
</tbody>
</table>


LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>RULES OF TRANSFORMING PROCESS MODELS TO PA. . .</td>
</tr>
<tr>
<td>II</td>
<td>PROBABILITY ANALYSIS FOR UNDECIDABILITY PROOF. .</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>DESCRIPTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>An HPA Example (Partial)</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>Types of Good Witness Sets</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>Witness Set Backward Traversal</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>Robustness Illustration</td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td>An Acyclic 1-HPA Example</td>
<td>26</td>
</tr>
<tr>
<td>6</td>
<td>Model Checking Acyclic HPA Step 1</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>Model Checking Acyclic HPA Step 2</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>HPA Leveling Illustrations. Two Options for State 4</td>
<td>34</td>
</tr>
<tr>
<td>9</td>
<td>Deciding Minimum Level for HPA</td>
<td>36</td>
</tr>
<tr>
<td>10</td>
<td>Deciding Minimum k for HPA</td>
<td>37</td>
</tr>
<tr>
<td>11</td>
<td>HPA Leveling Results Examples with More Q(_0) States</td>
<td>38</td>
</tr>
<tr>
<td>12</td>
<td>Irregular Language Acceptance in PBA</td>
<td>40</td>
</tr>
<tr>
<td>13</td>
<td>Semi-Terminal SCC Example</td>
<td>42</td>
</tr>
<tr>
<td>14</td>
<td>Flag Construction Using Vectors</td>
<td>45</td>
</tr>
<tr>
<td>15</td>
<td>Flag Construction Using Sets</td>
<td>45</td>
</tr>
<tr>
<td>16</td>
<td>Color Inheritance in Flag Construction Using Sets</td>
<td>47</td>
</tr>
<tr>
<td>17</td>
<td>Adding Exception Handling: PA Examples</td>
<td>56</td>
</tr>
<tr>
<td>18</td>
<td>Java Exception Handling Example (Partial)</td>
<td>60</td>
</tr>
<tr>
<td>19</td>
<td>Java Exception Handling Example: Division with External Input</td>
<td>62</td>
</tr>
</tbody>
</table>

viii
## LIST OF FIGURES (Continued)

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Automated Trading System Illustration.</td>
</tr>
<tr>
<td>21</td>
<td>File System Illustration.</td>
</tr>
<tr>
<td>22</td>
<td>HiPAM System Architecture.</td>
</tr>
<tr>
<td>23</td>
<td>HiPAM UI for HPA Generation.</td>
</tr>
<tr>
<td>24</td>
<td>HiPAM UI for HPA Generation from PRISM output.</td>
</tr>
<tr>
<td>25</td>
<td>HiPAM UI for HPA Analysis and Verification.</td>
</tr>
<tr>
<td>26</td>
<td>HPA Generation: Single Session $g_0$.</td>
</tr>
<tr>
<td>27</td>
<td>HPA Generation: Intermediate Process $g_{12}$.</td>
</tr>
<tr>
<td>28</td>
<td>HPA Generation: Failure Specification $f$.</td>
</tr>
<tr>
<td>29</td>
<td>HPA Generation: Intermediate Process $g_{21f}$.</td>
</tr>
<tr>
<td>30</td>
<td>HPA Generation: Property Specification $g_p$.</td>
</tr>
<tr>
<td>31</td>
<td>HPA Generation: Intermediate Process $g_{21fp}$.</td>
</tr>
<tr>
<td>32</td>
<td>HPA Generation: Intermediate Process $g_{21fpNoT}$.</td>
</tr>
<tr>
<td>33</td>
<td>HPA Generation: Final HPA - the Determinized $g_{21fpNoT}$.</td>
</tr>
<tr>
<td>34</td>
<td>Experiment Results for Robustness.</td>
</tr>
<tr>
<td>35</td>
<td>Experiment Results for Language Emptiness.</td>
</tr>
<tr>
<td>36</td>
<td>Memory Consumption Example.</td>
</tr>
<tr>
<td>37</td>
<td>Petri Net Demonstration.</td>
</tr>
<tr>
<td>38</td>
<td>An oWfDN Example: Buyer-Seller.</td>
</tr>
<tr>
<td>39</td>
<td>A BPMN Example: Claim Processing.</td>
</tr>
<tr>
<td>40</td>
<td>BPMN Gateway Specifications (Selected).</td>
</tr>
<tr>
<td>FIGURE</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>41</td>
<td>The OLC Graph for the Claim Processing Example.</td>
</tr>
<tr>
<td>42</td>
<td>The oOLC Graphs for the Buyer-Seller Example.</td>
</tr>
<tr>
<td>43</td>
<td>The BPMN Model with Exception Handling for the Claim Processing Example.</td>
</tr>
<tr>
<td>44</td>
<td>The oWfdN Model with Exception Handling for the Buyer-Seller Example.</td>
</tr>
<tr>
<td>45</td>
<td>The PA Model for the Claim Processing Example.</td>
</tr>
<tr>
<td>46</td>
<td>A Serialization Example for Transforming BPMN to PA.</td>
</tr>
<tr>
<td>47</td>
<td>The Procedure of Model Checking Business Processes.</td>
</tr>
<tr>
<td>48</td>
<td>HPA for Undecidability Proof.</td>
</tr>
</tbody>
</table>
## LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSA</td>
<td>Finite-State Automaton/Automata</td>
</tr>
<tr>
<td>PA</td>
<td>Probabilistic Automaton/Automata</td>
</tr>
<tr>
<td>HPA</td>
<td>Hierarchical PA</td>
</tr>
<tr>
<td>PFA</td>
<td>PA with Finite Acceptance</td>
</tr>
<tr>
<td>PBA</td>
<td>PA with Büchi Acceptance</td>
</tr>
<tr>
<td>HPFA</td>
<td>HPA with Finite Acceptance</td>
</tr>
<tr>
<td>HPBA</td>
<td>HPA with Büchi Acceptance</td>
</tr>
<tr>
<td>SCC</td>
<td>Strongly Connected Component</td>
</tr>
<tr>
<td>BPMN</td>
<td>Business Process Model and Notation</td>
</tr>
<tr>
<td>PN</td>
<td>Petri Net</td>
</tr>
<tr>
<td>WiN</td>
<td>Workflow Net</td>
</tr>
<tr>
<td>oWiN</td>
<td>open Workflow Net</td>
</tr>
<tr>
<td>WfDN</td>
<td>Workflow Net with Data</td>
</tr>
<tr>
<td>OLC</td>
<td>Object Life Cycle model(s)</td>
</tr>
</tbody>
</table>


SUMMARY

In this report we will systematically introduce using 1-HPA for model checking failure-prone open concurrent systems with respect to the non-extremal threshold language emptiness decision problem and the universal robustness problem. We have proved such problems are undecidable for 2-HPA but decidable for 1-HPA. We have developed two \textbf{EXPTIME} algorithms - backward algorithm and forward algorithm, and a tool - HiPAM, to model check 1-HPA, which can be obtained via combining open concurrent systems and properties expressed as deterministic automata. Properties are defined on system executions as either safety properties or non-safety properties. For some domain areas such as web applications, we have designed and implemented complete model abstraction techniques. When data models can be extracted from system models, such as in the domain of business process management, we have outlined a methodology to obtain failure specifications more precisely. Other contributions include verifying acyclic 1-HPA and proving that their language non-emptiness decision problem is \textbf{NP}-complete in general.

\textbf{Keywords}: Model Checking, Open Probabilistic System, Hierarchical Probabilistic Automata, Decidability, Emptiness Problem, Universality Problem
CHAPTER 1

INTRODUCTION

Many modern computer systems are interactive in nature. They accept inputs from the environment and process them in a timely manner. A large class of them employ multiple processors/servers to process many requests simultaneously. Server-end web applications (in which the inputs to the system are submitted as user forms), safety-critical cyber-physical systems (where the user commands are some of the inputs), file systems are some examples of such systems. These systems are failure prone. In a fault-tolerant or fail-safe design, once a server crashes, the other working servers take over the remaining session tasks from the failed server and complete them, while running their own sessions at the same time. In such situation, ensuring correct functioning of such systems is of critical importance, so we propose fundamental research on static techniques for formal verification of correctness of such open probabilistic systems.

If we model failures probabilistically, we can verify if a system satisfies a correctness property with a minimum probability on all input sequences even if a failure occurs. We assume that the probabilities of failures depend on the system states as well as the input requests, then we can employ Open Probabilistic Transition Systems (OPTSes) to model such failure-prone open systems. In general, an OPTS takes inputs from the environment and makes transitions to different states according to a probability distribution which may depend on the input as well as the current state. We consider systems abstracted as finite-state OPTSes, and properties
specified by deterministic automata on system computations. The problem of checking whether a system, specified by an OPTS $T$, satisfies a correctness specification given by an automaton $B$ with a probability greater than a given value $x$, reduces to the problem of checking whether the Probabilistic Automaton (PA) $T \times B$, accepts all input sequences with probability greater than $x$. Therefore, checking correctness reduces to checking universality of the language accepted by a PA (1; 2; 3; 4).

When we use the probabilistic behavior of the OPTS only for modeling failures as described above, we get hierarchical OPTSes. A hierarchical OPTS is one in which its states are stratified into $k+1$ levels, say $0, 1, \cdots, k$, (for some integer $k \geq 0$) such that, from any state, on an input, at most one transition goes to a state at the same level and all other transitions go to higher levels, and the initial state is at level 0. Such an OPTS is called a $k$-OPTS. Intuitively, it captures the situation when up to $k$ processors can fail probabilistically in some arbitrary sequence. The level of a state denotes the number of failures that occurred before reaching the state. Now checking correctness of a $k$-OPTS reduces to checking universality of the language accepted by an HPA (Hierarchical Probabilistic Automata).

HPA (5) are a subclass of Probabilistic Automata (PA). Essentially HPA are PA with state space stratified into multiple levels with restriction that from each state on each input, at most one transition goes to same-level states, and all others go to higher levels. We have proved in (6) that the decision problem of checking non-extremal threshold language emptiness is undecidable for general PA and HPA with more than two levels, but is decidable for 1-HPA in EXPTIME and is shown to be PSPACE-hard (5). When we model failures in open non-probabilistic
concurrent systems probabilistically as described earlier, with the restriction that at most one
failure occurs to a running session, the resulting systems can be modeled as 1-HPA.

Given a 1-HPA, we can apply one of the two EXPTIME algorithms - backward algorithm
and forward algorithm - to solve the decision problem of language emptiness. Besides, the
algorithms can also apply to a universal problem - the maximum probability that system $T$
always satisfies property $B$, which we call robustness. We have also developed a tool HiPAM
(Hierarchical Probabilistic Automata Model Checker) to provide an environment where open
concurrent systems with failures are modeled as PA and verified against safety properties (7).
All these contents will be covered in detail in this report.

The remaining contents of this report will be arranged as follows: firstly, background and
related works will be introduced in Chapter 2 and preliminaries in Chapter 3; secondly, the
verification algorithms for 1-HPA will be introduced in Chapter 4, implementation in Chapter 5
while implementation for HPA with Büchi acceptance (HPBA) separately introduced in Chapter
6; thirdly, a detailed explanation of the process of model abstraction is covered in Chapter
7, including introduction on a few application domains; next, our model checker HiPAM and
experimental results will be presented in Chapter 8; additionally, we will introduce some idea for
extracting more accurate failure specifications from the business process management domain
in Chapter 9, and introduce the enriched and refined undecidability result for 2-HPA in Chapter
10; finally, summary and future work will be covered in Chapter 11.

Note some major results have been published in our previous papers (5; 6). Unpublished
novel works in the report include: verification results for acyclic 1-HPA; algorithms and im-
plementations on verifying HPBA, which introduces the possibility of verification against non-safety property directly; new application domains exploration; techniques for extracting failure specification in the business process management domain; greatly expanded and improved model checker, with enhancement on obtaining witness sets and implementing verification algorithms, and more functionalities added including additional HPA leveling options, HPBA support and so on; and refined proof for undecidability of checking 2-HPA language emptiness.
CHAPTER 2

BACKGROUND

In this chapter we will introduce some background knowledge and related works.

2.1 Concurrent Systems

While both distribution and concurrency imply composition of multiple units, distributed unites (8; 9; 10) are usually asynchronous and they communicate by passing messages under the instruction of a global protocol/rule, but concurrent units are generally considered to be synchronous sharing a global clock. Our research can formally verify concurrent systems modeled as PA; what’s more, many so-called distributed systems aiming at achieving a global goal can also have concurrent units, where our research can also apply.

Concurrent units often run in parallel, or at least their running time overlap, and they might compete using shared resources. Because concurrent units can interact with each other at run time, the state space of the overall system can be extremely large and causing the state explosion problem. There is no special order among concurrent units, so even if they are executed out-of-order or in partial order, the overall system will yield the same result. Therefore partial order reduction can be applied to concurrent systems naturally, as what we have done in our research (see Section 8.1.1), so as to reduce the state spaces. The concurrent use of shared resources can lead to issues such as deadlocks and resource starvation, which we disallow for now but plan to support in the future.
Some mathematical models support concurrent computation naturally, such as Petri Net (PN) and process algebra. They are broadly used in domains where model expressiveness is a main consideration, such as business process management. In Chapter 9 we will relate PN-based business process modeling with our work.

There is a broadly used tool Spin (11; 12) for the formal verification of multi-threaded software applications. It targets specially at distributed systems and provides optimization techniques like partial order reduction and optionally BDD-like state storage. But Spin does not have good support for probabilistic systems like our tool HiPAM does.

2.2 Model Checking Probabilistic Systems

Markovian processes are broadly used for modeling probabilistic systems. They are basically transition systems where at each state the next state is decided by a transition matrix independently. For Petri Net there is also an extended version called Stochastic Petri Net (SPN) supporting stochastic behavior, whose semantics is essentially a Markovian process. Markov Decision Process (MDP) is a type of extended Markovian process where at each step there are a few actions/schedulers to choose from, with each action there will be an associated transition matrix and a reward value on each transition. MDP is essentially a discrete-time stochastic control process. Its syntax looks similar to PA with input symbols (Section 3.1), but unlike a PA state with only partial profile restricted by the input symbols, a MDP state/step always has a complete profile and universally quantifies over all schedulers, so it works for closed systems but not for open systems.
There is a popular probabilistic model checker called PRISM (13) using mostly Markovian models. It checks probabilistic systems against properties modeled as temporal logics. In our PA-based model checker HiPAM we provide an interface with PRISM (see Section 8.1.2) making use of its modeling syntax to abstract larger models easily.

PRISM has an extension PRISM-games (14), which is a model checker for turn-based, perfect-information stochastic multi-player games. It does support reasoning about probabilities and precise bounds; however, it disallows synchronization, unlike our HIPAM model checker.

2.3 Reliability v.s. Robustness

One related field of study is Software Reliability Engineering (SRE) (15; 16; 17) where reliability is defined as the probability that a initially well-behaved system will function as intended for a specified period of time under specified conditions. Software reliability largely depends on environment and actual usage quantified as operational profiles, so reliability engineering puts much focus on acquiring the operational profiles during software requirement analysis. Failure models are often specified probabilistically using density/intensity/rate functions (failure quantities against execution time), and the methodology is more statistics in practice than formal model checking.
CHAPTER 3

PRELIMINARIES

We assume reader has understanding of Transition System (TS), Finite-State Automata (FSA), and regular languages, \( \omega \)-regular languages. The closed unit interval will be denoted by [0, 1] and the open unit interval by (0, 1). Given any finite set \( S \), \(|S|\) denotes the cardinality/size of \( S \), that is, the number of elements in \( S \).

For a finite sequence/string/word \( \kappa = s_1s_2\cdots \) over a set \( S \) of symbols, \( \kappa[i] \) denotes the element \( s_i \in S \) of the sequence, \(|\kappa|\) denotes the length of \( \kappa \), and \( \text{Pref}(\kappa) \) denotes the complete set of prefixes of \( \kappa \). \( S \) is also called alphabet, and we consider only finite alphabets. In accordance with common understanding, \( S^* \) denotes the set of all finite sequences over \( S \), and \( S^+ \) denotes the set of all finite non-empty sequences over \( S \), that is, \( S^+ = S^* \cup \{\epsilon\} \), where \( \epsilon \) denotes the empty sequence. What’s more, \( S^\omega \) denotes the set of all infinite sequences over \( S \). A language of finite (or infinite) words is a subset of \( S^* \) (or \( S^\omega \)).

3.1 Probabilistic Automata

A probabilistic automaton (PA) \((1; 2)\) over alphabet \( \Sigma \) is a tuple \( A = (Q, q_0, \delta, Q_f) \), where:

- \( Q \) is a set of states/nodes;
- \( q_0 \in Q \) is the single initial state;
• $\delta : Q \times \Sigma \times Q \to [0,1]$ is the transition relation, such that for all $q, q' \in Q$ and $a \in \Sigma$, $\delta(q, a, q')$ is a rational number on a triple $(q, a, q')$, and $\sum_{q' \in Q} \delta(q, a, q') = 1$; triple $(q, a, q')$ implies a transition $(q, q')$ if $\delta(q, a, q') > 0$;

• $Q_f \subseteq Q$ is an acceptance condition, also called the set of final/accepting states.

A state $q$ is absorbing/terminal/sinking if every transition from $q$ goes back to $q$ itself.

The transition function $\delta$ of PA $A$ on input $a \in \Sigma$ can be seen as a square matrix $\delta_a$ of order $|Q|$ with the rows labeled by “current” state, columns labeled by “next state” and the entry $\delta_a(q, q')$ equal to $\delta(q, a, q')$. Given a word $u = a_0a_1 \cdots a_n \in \Sigma^+$, $\delta_u$ is the product matrix $\delta_{a_0} \delta_{a_1} \cdots \delta_{a_n}$. For the empty word $\epsilon$ we take $\delta_\epsilon$ to be the identity matrix. Finally for any $Q_0 \subseteq Q$, we say that $\delta_u(q, Q_0) = \sum_{q' \in Q_0} \delta_u(q, q')$. Given a word $u \in \Sigma^+$, the successor of a state $q \in Q$ on $u$ (also known as $q$’s $u$-successor) is the set $\text{post}(q, u) = \{q' \mid \delta_u(q, q') > 0\}$, and the predecessor of $q$ on $u$ (also known as $q$’s $u$-predecessor) is the set $\text{pre}(q, u) = \{q' \mid \delta_u(q', q) > 0\}$.

For a set $C \subseteq Q$, its $u$-successor is the set $\text{post}(C, u) = \cup_{q \in C} \text{post}(q, u)$, and its $u$-predecessor is the set $\text{pre}(C, u) = \cup_{q \in C} \text{pre}(q, u)$.

Intuitively, a PA $A$ starts at the initial state $q_0$. If after reading an input sequence $a_0, a_1, \cdots, a_t$ $A$ is at state $q$, then it will move to state $q'$ on symbol $a_{i+1}$ with probability $\delta_{a_{i+1}}(q, q')$. A run/path $\rho \in Q^*$ (or $\rho \in Q^\omega$) of $A$ starting in a state $q \in Q$ on a finite input sequence $\kappa \in \Sigma^*$ (or an infinite input sequence $\kappa \in \Sigma^\omega$) is a state sequence such that

1A probabilistic system with this restriction is stochastic; also PA is deterministic in the sense that at every state $q \in Q$ on every input $a \in \Sigma$, there is a valid transition relation defined. This condition may be relaxed as $\sum_{q' \in Q} \delta(q, a, q') = 0$ or 1 later when we introduce reduction to PA.
\( \rho[0] = q_0, |\rho| = 1 + |\kappa|, \) and for each \( i \geq 0, \delta_{\kappa[i]}(\rho[i], \rho[i + 1]) > 0 \). State \( \rho[j] \) is said \textit{reachable} from state \( \rho[i] \) where \( j > i \). We say that \( \kappa \) is accepted and \( \rho \) is an \textit{accepting} run if \( \rho \) satisfies \( \mathcal{A} \)'s acceptance condition. The probability of an accepting run is the product of all probabilities on the transitions in the run, and the probability of \( \mathcal{A} \) accepting an input sequence is the sum of the probabilities of all accepting runs on the sequence.

**Finite Acceptance.** A run \( \rho \) on a finite input sequence is accepted iff \( \rho[|\rho| - 1] \in Q_f \), i.e., the last state on \( \rho \) is an accepting state. A PA defined with finite acceptance is called a Probabilistic Finite Automaton (PFA), and PFA behave very similarly to Markov chains.

**Büchi Acceptance.** Büchi acceptance (18) defines languages over infinite sequences. Formally, a Büchi acceptance condition of PA \( \mathcal{A} \) is defined on a set of final states \( Q_f \subseteq Q \), and a run \( \rho \) of \( \mathcal{A} \) on an infinite word \( \alpha \in \Sigma^\omega \) is said to be \textit{accepting} if \( \rho \) passes states in \( Q_f \) infinitely often. A PA defined with Büchi acceptance is called a Probabilistic Büchi Automaton (PBA).

The language \( L_{f > x}(\mathcal{A}) \) (or \( L_{f \geq x}(\mathcal{A}) \)) of a PFA \( \mathcal{A} \) denotes the set of those finite sequences accepted by \( \mathcal{A} \) with probabilities greater than (or greater than or equal to) \( x \), where \( x \in [0, 1] \) is a probability threshold; similarly, \( L_{b > x}(\mathcal{A}) \) and \( L_{b \geq x}(\mathcal{A}) \) are defined for PBA.

Unless otherwise stated, we assume PA are finite-state, that is, the set \( Q \) of states is finite. For default notations throughout the report, \( n \) will be used as the size of the state space, i.e. the number of states, \( m \) as the number of transitions, and \( s \) as the size of alphabet.

**3.1.1 Underlying Directed Graph of PA**

The states and transitions of a PA \( \mathcal{A} = (Q, q_0, \delta, Q_f) \) essentially construct a directed graph \( G_{\mathcal{A}} = (Q, E) \) with \( Q \) as its state space and \( E = \{(q, q') \mid \exists a \in \Sigma, \delta(q, a, q') > 0\} \) is the set of
transitions. A strongly connected component (SCC) of $G_A$ is a maximal subset of $Q$ such that every two nodes in $G_A$ are mutually reachable; the component graph of $G_A$ is the directed graph $F_A = (C, E)$ where $C$ is the set of SCCs of $G_A$, and $(C, D) \in E$ iff $C \neq D$ and $\exists q \in C, q' \in D$ such that $(q, q') \in E$. Define $C_0 \in C$ where $q_0 \in C_0$ as the initial SCC of $F_A$. Under reachability assumption, every node in $F_A$ is reachable from $C_0$.

It is known that $F_A$ is acyclic, i.e., has no cycles. Using standard graph algorithms, $G_A$ and $F_A$ can be constructed in time $O(n + m)$.

SCCs and component graph will be used several times in this report when introducing graph-based algorithms for PA. In such cases, we may treat a PA as its underlying directed graph for simplicity, and use terms such as “SCCs of a PA” to imply “SCCs of the underlying directed graph of a PA”, use “the component graph of a PA” to imply “the component graph of the underlying directed graph of a PA”.

### 3.2 Hierarchical Probabilistic Automata

For integer $k > 0$, a $k$-hierarchical probabilistic automaton (HPA) (5; 6) is a PA $A = (Q, q_0, \delta, Q_f)$ over alphabet $\Sigma$ such that $Q$ can be partitioned into $k + 1$ sets $Q_0, Q_1, \ldots, Q_k$ satisfying the following properties:

- $q_0 \in Q_0$;
- $\forall i, 0 \leq i \leq k$ and every $q \in Q_i$ and $a \in \Sigma$, $|\text{post}(q, a) \cap Q_i| \leq 1$; and,
- $\forall i, 0 < i \leq k$, $q \in Q_i$ and $a \in \Sigma$, $\text{post}(q, a) \cap Q_j = \emptyset$ for every $j < i$.

Hierarchical Probabilistic Finite Automata (HPFA) and Hierarchical Probabilistic Büchi Automata (HPBA)(19; 18) are defined as HPA with finite acceptance and Büchi acceptance respectively.
For any k-HPA $\mathcal{A}$, for $0 \leq i \leq k$, we call elements of $Q_i$ as level-i states of $\mathcal{A}$. It’s easy to see that every transition between two $Q_k$ states has probability 1. If from one state $q \in Q_i$ of $\mathcal{A}$ on an input $a$ there are more than transitions with probabilities greater than 0, then $\mathcal{A}$ must have highest level $k > i$. Take the HPA on alphabet $\Sigma = \{A, B, C\}$ given in Figure 1 as an example. At most one of $s_1$ and $t_1$ is on the same level with $s_0$, so at least one of $s_1$ and $t_1$ is on a higher level than $s_0$. Note unspecified input transitions from each state go to the absorbing state $t_3$ with probability 1.

![Figure 1. An HPA Example (Partial).]

We have proved that the non-extremal threshold language emptiness problem and the universality problem are decidable for 1-HPA(5), later we will show the problems are undecidable for 2-HPA in Chapter 10. For the rest of the report, while presenting verification techniques, HPA implies 1-HPA by default.

### 3.3 Witness Sets

Given a 1-HPA $\mathcal{A} = (Q, q_0, \delta, Q_f)$, a witness set is $W \subseteq Q$ which has at most one $Q_0$ state; $q_W$ denotes this $Q_0$ state if it exists. Therefore, there are two types to witness sets: witness sets with no $Q_0$ state (WS0), and those with exactly one $Q_0$ state (WS1).

A set $W \subseteq Q$ of states is good if there exists a sequence $u$ such that $u$ is accepted with probability 1 from every state $q \in W$. Intuitively, a good witness set is defined to “witness”
the acceptance of some input sequence, which can be accepted by at most one run on $Q_0$ and all other runs on $Q_1$ in a 1-HPA.

We are interested in four types of witness sets with different “goodness”: $X$ contains witness sets whose $Q_1$ parts are good, $Y$ is the set of good WS1, and $X'$ is the set of WS1 whose $Q_1$ parts are good, $Y'$ is the set of all good witness sets. These good witness sets play an important role in our verification algorithms. Their relationships are shown in Figure 2.

\[ X' = \{ W | W \cap Q_1 \text{ is good}, |W \cap Q_0| = 1 \} \]

“WS1, i.e. good WS0 plus any $Q_0$”

\[ Y = X' \cap Y' = \{ W | W \text{ is good}, |W \cap Q_0| = 1 \} \]

“good WS1”

\[ Y' = \{ W | W \text{ is good} \} \]

“good WS, i.e. good WS0 and good WS1”

$X = X' \cup Y' = \{ W | W \cap Q_1 \text{ is good} \}$

“good WS0 and WS1”

Figure 2. Types of Good Witness Sets.

3.3.1 Backward Traversal

Here we will introduce an important property of good witness set, and later it will be used in obtaining witness sets for our verification algorithms.

From a known good set of states, we can traverse backward on each input symbol and get other witness sets incrementally. Take a PA $A = (Q, q_0, \delta, Q_f)$ over alphabet $\Sigma$ defined with finite acceptance as an example. Any set $F \subseteq Q_f$ containing at most one $Q_0$ state constructs a good witness set. If we start from $Q_f$ and traverse backward on a finite input sequence $u$, as presented in Figure 3, we will reach a set $V = \text{pre}(Q_f, u)$ composed of three parts:

1. a good WS0, i.e. $\text{pre}(Q_f, u) \cap Q_1$, which is obtained using only probability 1 transitions, and is same as $\text{pre}(Q_f \cap Q_1, u) \cap Q_1$;
2. superQ₀, i.e. the subset of \( \text{pre}(Q₁, u) \cap Q₀ \) obtained using only probability 1 transitions;

3. goodQ₀, i.e. the subset of \( \text{pre}(Q₁, u) \cap Q₀ \) obtained using transitions with probabilities greater than 0 but less than 1.

So a good WS₀ on \( Q₁ \) is associated with a set superQ₀ on \( Q₀ \) and a set goodQ₀ on \( Q₀ \).

Keep traversing backward from \( V \) on symbol \( a \in \Sigma \), we get another set \( W = \text{pre}(V, a) \), while \( W \)'s good WS₀ is obtained from \( V \)'s good WS₀ on probability 1 transitions, \( W \)'s superQ₀ is obtained from \( V \)'s good WS₀ and superQ₀ on probability 1 transitions, and \( W \)'s goodQ₀ is the remaining part of \( \text{pre}(V, a) \).

Figure 3. Witness Set Backward Traversal.

This backward traversal algorithm will get us only desired witness sets: a good WS₀ with one of its superQ₀ nodes constructs a good WS₁ in \( \mathcal{Y} \), and a good WS₀ with one of its goodQ₀ nodes constructs a WS₁ in \( \mathcal{X}' - \mathcal{Y} \). It has a polynomial time complexity with respect to the number of good witness sets obtained on some accepted input string.
CHAPTER 4

VERIFICATION ALGORITHMS

Given a valid 1-HPA, we can decide its language emptiness (non-emptiness) problem given a probability threshold value $x$, and/or its robustness with defined precision. In this chapter we will introduce two verification algorithms that have been published in our previous papers (5) and (6) - backward algorithm and forward algorithm. Both algorithms are based on the calculation of good witness sets for the HPA, which is covered in Section 5.5 and Chapter 6. More details on applying the algorithms will be covered in Chapter 5 and Chapter 8.

Let $x \in [0, 1]$ be a rational threshold of size at most $r$. It is shown in (5) that for an HPA $A$, $L_x(A) \neq \emptyset$ iff there is a finite word $u$ and a good non-empty witness set $W$, such that $\delta_u(q_0, W) > x$ and $|u|$ is less than or equal to a threshold $L = 4rn8^n$.

4.1 Backward Algorithm

Backward algorithm was first introduced in our paper (5) and then improved in our later paper (6). Recall $\mathcal{X}'$ is the set of all $W \in \mathcal{X}$ such that $W \cap Q_0 \neq \emptyset$, $\mathcal{Y}'$ is the set of all good witness sets. A function $\text{Prob}(\cdot, \cdot)$ which maps each pair $(W, i)$, where $W \in \mathcal{X}'$ and $i \geq 0$, to a probability value, is defined as follows: $\text{Prob}(W, i) = \max(\delta_u(q_W, V) | u \in \Sigma^*, V \in \mathcal{Y}', \text{post}(W \cap Q_1, u) \subseteq V, |u| \leq i)$ whereas $\max(\emptyset) = 0$.

\footnotetext[1]{A rational number $s$ has size $r$ iff $s = \frac{m}{n}$ where $m, n$ are integers, and the binary representation of $m$ and $n$ has at most $r$ bits.}
Prob(W, i) denotes the maximum probability of reaching any set $V \in Y'$ using an input of length at most $i$, from the state $q_W$. If $\text{Prob}(W, i) > x$, there is an input accepted from $q_W$ with probability $> x$ (because from $q_W$ using an input of length at most $i$, we can reach a good witness or semi-witness set $V$, with probability $> x$, and then we can use any sequence that is accepted from all states in $V$ with probability 1; there exists at least one such sequence because $V \in \mathcal{X}'$). Clearly, $\text{Prob}(W, i)$ is monotonically non-decreasing with increasing values of $i$.

Since $L_{\geq x}(A) \neq \emptyset$ iff $\text{Prob}([q_0], L) > x$ ((5)), the following inductive algorithm is proposed to compute $\text{Prob}(\cdot, \cdot)$.

\[
\text{Prob}(W, 1) = \max\{\delta_a(q_W, V) | a \in \Sigma, V \in Y', \text{post}(W \cap Q_1, a) \subseteq V\};
\]

\[
\text{Prob}(W, i + 1) = \max(\{\text{Prob}(W, i)\} \cup \{\delta_a(q_W, q_V) \text{Prob}(V, i) + \delta_a(q_W, V \cap Q_1) | a \in \Sigma, V \in \mathcal{X}', \text{post}(W \cap Q_1, a) \subseteq V\}) \quad (4.1)
\]

Backward algorithm works as follows. For increasing values of $i = 1, \ldots, L$, it computes $\text{Prob}(W, i)$ for each $W \in \mathcal{X}'$. The algorithm terminates at the smallest $i > 1$ such that one of the following conditions is satisfied: (a) $\text{Prob}([q_0], i) > x$, (b) for each $W \in \mathcal{X}'$, $\text{Prob}(W, i) = \text{Prob}(W, i - 1)$, (c) $i = L$ and $\text{Prob}([q_0], i) \leq x$. It is not difficult to see that if the convergence condition (b) is satisfied for a particular value $i$, then for all $j \geq i$, $\text{Prob}(W, j) = \text{Prob}(W, i)$, i.e., the values given by $\text{Prob}(\cdot)$ do not change. On termination, if condition (a) is satisfied then the algorithm gives “yes” answer, otherwise it gives a “no” answer. We refer to condition (a) as positive termination, condition (b) and (c) as negative termination conditions.

Backward algorithm is in $\text{EXPTIME}$ in the worst case. However, in case it converges and terminates early, it may take much less time than the theoretically worst time. The following theorem states the correctness of the algorithm.
Theorem 4.1 (Backward Termination). Backward algorithm is correct, that is, it answers “yes”, iff $L_{>x}(A) \neq \emptyset$.

4.1.1 Big O Analysis of Backward Algorithm

Recall the formula for $\text{Prob}(W,i)$ in Equation 4.1. When we fix $W$, the set of a-successors (i.e. $\{V\}$ in the equation above) are fixed for fixed input $a$, thus it is calculated only once and then stored. To find the a-successors for $W$, we need to scan through all witness set in WS and find the $\{V\}$. So totally this takes time $w*(m+n)$. In the worst case, backward algorithm takes time:

- $w \times \ldots$ for each witness set $W$
- $s \times \ldots$ for each input of $W$
- $w \times (m+n) \ldots$ scan WS to decide $W$’s a-successors, i.e. $\{V\}$ at the first run
- $+L_b \times \ldots$ for each increasing length of input until algorithm terminates
- $w \times \ldots$ for each witness set $W$
- $s \times \ldots$ for each input of $W$
- $w \times \ldots$ for each a-successor
  
  $m \ldots$ to calculate $\delta_a(q_U,q_V)\text{Prob}(V,i) + \delta_a(q_U,V \cap Q_1)$ etc.

$$= O((L_b m + m + n)sw^2)$$

4.2 Forward Algorithm

Forward algorithm was first introduced in our paper (6). It is based on a quantity $\text{val}(C,x,u)$ defined for a set $C \subseteq Q_1$ of states in HPA $A$, a threshold $x \in (0,1)$, and a finite word $u$. The
intuition of $val$ is, starting from $q_0$, after the input string $u$, the probability that the automaton is in some state in $C$ is $\delta_u(q_0, C)$; this means an additional probability of $(x - \delta_u(q_0, C))$ is needed to cross the threshold $x$; this additional probability can only come from the probability remaining at level 0, which is $\delta_u(q_0, Q_0)$. Thus, $val(C, x, u)$, defined as the ratio of the above additional probability to that of $\delta_u(q_0, Q_0)$, is the fraction of $\delta_u(q_0, Q_0)$ that still needs to move to $C$ so that the probability of reaching the accepting states at the end exceeds the threshold $x$. Formally, we have the following definition.

$$val(C, x, u) = \begin{cases} x - \frac{\delta_u(q_0, C)}{\delta_u(q_0, Q_0)} & \text{if } \delta_u(q_0, Q_0) \neq 0 \\ +\infty & \text{if } \delta_u(q_0, C) < x, \delta_u(q_0, Q_0) = 0 \\ 0 & \text{if } \delta_u(q_0, C) = x, \delta_u(q_0, Q_0) = 0 \\ -\infty & \text{if } \delta_u(q_0, C) > x, \delta_u(q_0, Q_0) = 0 \end{cases}. \quad (4.2)$$

From the definition of the function $val$, and using algebraic simplifications, the following properties are easily proved.

**Lemma 4.2.** For $u, v \in \Sigma^+$, if $C, D \subseteq Q_1$, $q, q' \in Q_0$ be such that $C = \text{pre}(D, v) \cap Q_1, \{q\} = \text{post}(q_0, u), \{q'\} = \text{post}(q, v)$, $x' = \delta_v(q, D), z' = \delta_v(q, q'), y' = 1 - z'$ and $y', z' > 0$, then the following hold:

1. $val(D, x, uv) = \frac{val(C, x, u) - x'}{y'}$, and

2. If $C = D$ (i.e., post($C, v$) = $C$) and $q' = q$, then for any integer $p \geq 0$, $val(C, x, uv^p) = \frac{z'}{y'} + \left(\frac{1}{z'}\right)^p(val(C, x, u) - \frac{z'}{y'})$. 


Proof. Let \( u, v, C, D, q, q', x', y' \) and \( z' \) be as given in the statement of the lemma. Part (1) of the lemma is seen as follows. From the definition, we have \( \text{val}(D, x, uv) = \frac{x - \delta_{uv}(q_0, D)}{\delta_u(q_0, q) \cdot x'} \). Clearly, 
\[
\delta_{uv}(q_0, D) = \delta_u(q_0, C) + \delta_u(q_0, q) \cdot x'.
\]
Substituting the right hand of the above equation for \( \delta_{uv}(q_0, D) \) in the equation for \( \text{val}(D, x, uv) \) and performing some simplifications, we get 
\[
\text{val}(D, x, uv) = \frac{x - \delta_u(q_0, C)}{\delta_u(q_0, q) \cdot z'} - \frac{x'}{z'},
\]
which is seen to be same as the equation given in part (1). Part (2) of the lemma is proved by induction. For the base \( p = 0 \) and in this case the part (2) holds, since the right hand side expression of the equation simplifies to \( \text{val}(C, x, u) \). For the induction step, assume part (2) holds for all \( p \leq k \). Now, using part (1), we see that 
\[
\text{val}(C, x, uv^{k+1}) = \frac{1}{z'}(\text{val}(C, x, uv^k) - x')
\]
Using the induction hypothesis and substituting for \( \text{val}(C, x, uv^k) \), after some simplification, we get the equation of part (2) for \( p = k + 1 \). \( \square \)

The \( \text{val} \) values play an important role in deciding whether a word \( \kappa \) is accepted by an HPA. It has been shown in (5) that \( \kappa \) is accepted with probability \( x \) iff \( \kappa \) can be divided into strings \( u, \kappa' \), i.e., \( \kappa = u\kappa' \), and there is a witness \( W \) such that \( \kappa' \) is definitely accepted from \( W \) and one of the following conditions:

- \( W \subseteq Q_1, \text{val}(W, x, u) < 0 \)
- \( W \cap Q_0 \neq \emptyset \) and \( 0 \leq \text{val}(W \cap Q_1, x, u) < 1 \).

Now, to check the existence of a string \( \kappa \) satisfying the above property, we define another quantity \( \text{minval}(W, i) \) for each \( W \in \mathcal{X} \) and \( i \geq 0 \) as follows. Intuitively, this is the minimum of
the values given by \( \text{val} \) over all strings of length at most \( i \). Formally, we have \( \text{minval}(W, i) = \min(\text{val}(W \cap Q_1, x, u) | |u| \leq i) \) where \( \min(\emptyset) = +\infty \).

Recall definitions of witness sets in Section 3.3. It is easily proved that \( L_{>x}(A) \neq \emptyset \) iff for some \( i \geq 0 \), either \( \exists W \in \mathcal{X} : \text{minval}(W, i) < 0 \) or \( \exists W \in \mathcal{Y} : 0 \leq \text{minval}(W, i) < 1 \).

Now, for any \( W \in \mathcal{X}, a \in \Sigma, \) and \( q \in Q_0 \), let \( W_{a,q} = (\text{pre}(W, a) \cap Q_1) \cup \{q\} \in \mathcal{X} \). To calculate the values \( \text{minval}(W, i) \) for \( W \in \mathcal{X} \) and \( i \geq 0 \), we present the following algorithm.

### Algorithm

\[
\begin{align*}
\text{minval}(W, 0) &= \begin{cases} 
\text{x} & \text{if } q_0 \in W; \\
+\infty & \text{else}.
\end{cases} \\
\text{minval}(W, i) &= \begin{cases} 
-\infty & \text{if } \exists a \in \Sigma, q \in Q_0 \text{ such that } \\
0 & \text{if } 0 \leq \text{minval}(W_{a,q}, i-1) < \delta_a(q, W); \\
\text{minval}(W, i-1) & \text{else.}
\end{cases}
\end{align*}
\]

(4.3)

For \( i > 0 \) and \( W \cap Q_0 = \emptyset \),

\[
\begin{align*}
\text{minval}(W, i) &= \min(\text{minval}(W, i-1), \\
&\min(\text{minval}(W_{a,q}, i-1) - \delta_a(q, W \cap Q_1) \\
&\delta_a(q, q_W) | a \in \Sigma, q \in Q_0, \delta_a(q, q_W) > 0)).
\end{align*}
\]
By induction on \(i\), it can be shown the above algorithm computes \(\minval(\cdot, \cdot)\) correctly, i.e., the computed values agree with the definition of \(\minval(\cdot, \cdot)\) given earlier. Clearly, \(\minval(W, i)\) is monotonically non-increasing with increasing values of \(i\).

Let \(w = |X|\), then \(w \leq 2^n \leq L\). The algorithm computes the values of \(\minval(W, i)\), for each witness set \(W \in \mathcal{X}\), in increasing values of \(i = 0, \cdots\) until one of the following conditions is satisfied: (a) \((\exists W \in \mathcal{X} : \minval(W, i) < 0)\) or \((\exists W \in \mathcal{Y} : 0 \leq \minval(W, i) < 1)\); (b) \(i > 0\) and \(\forall W \in \mathcal{X}, \minval(W, i) = \minval(W, i - 1)\); (c) \(i = w\). Observe that once condition (b) holds, the values of \(\minval(W, i)\) do not change from that point onwards, i.e., they reach a fixed point.

The algorithm terminates for the smallest integer \(i\) such that either of the conditions (a) or (b) hold. If at termination, condition (a) holds then it answers “yes”; if (a) does not hold but (b) holds then it answers “no”; if both (a) and (b) do not hold, but (c) holds then it will answer “yes”. Now we have the following theorem.

**Theorem 4.3 (Correctness of Forward Algorithm).** _Forward algorithm will definitely terminate after at most \(w\) iterations providing the correct answer, i.e., it outputs “yes” iff \(L_{>\chi}(A) \neq \emptyset\)._ 

**Proof.** Clearly, the algorithm terminates within at most \(w\) iterations and outputs an “yes” or “no” answer. It is enough if we prove that it outputs the correct answer. Assume that the algorithm outputs an “yes” answer, then either condition (a) is satisfied, clearly implying \(L_{>\chi}(A) \neq \emptyset\), or condition (c) is satisfied, i.e., \(i = w\) at termination and neither of conditions (a), (b) is satisfied. Let \(F_0 = \{W \in \mathcal{X} | q_0 \in W\}\). For \(j > 0\), let \(F_j = \{W \in \mathcal{X} | \minval(W, j) < \minval(W, j - 1)\}\). It is easily seen that for each \(j > 0, F_j \neq \emptyset\), otherwise the fixed point condition (b) would have been satisfied before the \(w^{th}\) iteration. Also for each \(W \in F_j, W \cap Q_0 \neq \emptyset\) and there exists \(V \in F_{j-1}\)
and $a \in \Sigma$ such that $\minval(W, j) = \frac{\minval(V_{j-1}, W) - \delta_a(q, W \cap Q_1)}{\delta_a(q, W \cap Q_0)}$. Since $F_W \not= \emptyset$, the above property would imply that there exists $W_0 \in F_0$ and for each $0 < j \leq w$, there exists $W_j \in F_j$ and $a_j \in \Sigma$ such that $\minval(W_{j-1}, j-1) - \delta_a(q, W_j \cap Q_1) \leq \delta_a(q, W_{j-1} \cap Q_0)$ where $q = q_{W_{j-1}}$. Let $\kappa_j = (a_1, \ldots, a_j)$ for $j > 0$. Then, by simple induction on $j$, it is seen that $\minval(W_j, j) = \val(W_j \cap Q_1, x, \kappa_j)$.

Now, using Pigeon Hole principle, we see there exist integers $0 \leq j < k \leq w$ such that $W_j = W_k$. Let $V = W_j$ and $q = q_{W_j}$. Clearly $\minval(V, k) < \minval(V, j)$. Now, let $u = \kappa_j$ as given above, and $v = (a_{j+1}, a_{j+2}, \ldots, a_k) \in \Sigma^*$. Let $x' = \delta_v(q, V \cap Q_1)$, $z' = \delta_v(q, q)$ and $y' = 1 - z'$. From the earlier observation, $\minval(V, j) = \val(V \cap Q_1, x, u)$ and $\minval(V, k) = \val(V \cap Q_1, x, uv)$. Using property (1) of Lemma 4.2, it is seen that $\minval(V, j) - \minval(V, k) = y' \frac{x'}{y'} > \minval(V, j)$. Observe that $|u| = j, |v| = k - j$. Now, for an integer $p > 0$, let $n_p = j + p(k - j)$. By considering the string $uv^p$ and using part (2) of lemma 4.2, we see that $\minval(V, n_p) = \rho + \left(\frac{1}{2}\right)^p(\minval(V, j) - \rho)$ where $\rho = \frac{x'}{y'}$. Since $\minval(V, j) - \rho < 0$ and $z' < 1$, we see that $\minval(V, n_p) < 0$ for sufficiently large $p > 0$ and $L_{>x} (\mathcal{A}) \not= \emptyset$.

Now, assume the algorithm terminated after $j$ iterations and outputs a “no” answer, which means condition (b) is satisfied but condition (a) is not. Clearly, condition (a) is not satisfied for all $i \leq j$. Condition (b) implies for all $i \geq j$ and for all $W \in \mathcal{X}$, $\minval(W, i) = \minval(W, j)$. Hence for all $i \geq j$, condition (a) is not satisfied, so $L_{>x} (\mathcal{A}) = \emptyset$.

### 4.2.1 Big O Analysis of Forward Algorithm

Recall the formula for $\Val(W, i)$ in Equation Equation 4.3. When we fix $W$, $W_{a, q} = \{r \in Q_1 | \delta_a(r, W) > 0\} \cup \{q\}$ is unique for fixed $(a, q)$, thus it can be calculated once and then stored. Note the level 1 states in $W_{a, q}$ are obtained by searching for those $r$’s on level 1 with
a probability greater 0 (exactly probability 1 since on level 1) transferring to $W$ on $a$, this is exactly the same with how we generated witness sets WS using backward traversal, therefore $W_{a,q}$ is already in WS. So the time spent to calculate $W_{a,q}$ is roughly $O(|Q_1| \cdot m)$ for deciding the level 1 states in it, and $O(|Q_1|)$ to locate its unique ID in WS if WS is implemented in data structures like trie, thus totally $O(|Q_1| \cdot m + |SWS| \cdot |Q_1|)$. Therefore in the worst case, forward algorithm takes time:

$$wx\ldots\text{for each witness set } W$$
$$([Q_1] \times m + |Q_1|)\ldots\text{for calculating } W_{a,q} \text{ at the first run}$$
$$+L_f \times \ldots\text{for each increasing length of input until algorithm terminates}$$
$$wx\ldots\text{for each witness set } W$$
$$m \times \ldots\text{the number of } <a,q> \text{ is at most } |\text{in-transitions of } W| \leq m$$
$$m \times \ldots\text{to calculate } \delta_a(q, W \cap Q_1) \text{ and } \delta_a(q, W \cap Q_0) \text{ etc. w.r.t. } W_{a,q}$$

$$= O((mn + L_f m^2)w)$$

4.3 Forward vs. Backward

Both forward and backward algorithms can be utilized to solve the decidability problem. Suppose the HPA $A$ has $n$ states, $m$ transitions, $w = |\mathcal{X}|$ and the size of alphabet is $s$. Forward algorithm has worst time complexity $O((mn + L_f m^2)w)$, while backward algorithm runs in $O((L_b m + n + m)sw^2)$, where $L_f$ and $L_b$ denote the number of iterations before the respective algorithms terminate. Clearly $L_f \leq w$, while $L_b \leq L$. Since $w = O(2^n)$ and $L > w$ is also exponential in $n$, both algorithms run in time exponential in $n$ in the worst case. However,
the above complexities show that forward algorithm is quadratic in \( w \), while that of backward algorithm is cubic in \( w \), considering the dependence of \( L_f, L_b \) on \( w \). Thus, forward algorithm has better worst case complexity. Our experimental results, given in Section 8.3, show that forward algorithm has better average time complexity than backward algorithm. Forward algorithm often terminates much faster than backward algorithm in cases where \( L_{>x}(A) = \emptyset \) and the latter terminates on condition (c). However, backward algorithm can terminate sooner in some non-empty cases, where the maximum \( \text{Prob}(\cdot, \cdot) \) value is purely decided by good witness sets without \( Q_0 \) states. In such cases, backward algorithm is benefiting from the truth that good witness sets are generated backward from accepting states, and effort is saved in calculating \( \text{Prob}(\cdot, i) \) values for increasing \( i \). What’s more, forward algorithm critically depends on \( x \), i.e., the definition of \( \text{minval}(\cdot, \cdot) \) function depends on \( x \), while the \( \text{Prob}(\cdot, \cdot) \) used in backward algorithm is independent of \( x \).

### 4.4 Robustness

When failure-prone systems are modeled as HPA, an equally important notion is robustness. Intuitively, this is the maximum probability that a system does not violate desired correctness property. If we model the incorrectness of an open system under failures as a 1-HPA, we can define the robustness of an HPA \( A \) as \( (1 - y) \) where \( y \) is the least upper bound of the set of values \( \{ x \mid L_{=x}(A) \neq \emptyset \} \). Thus the robustness value is the greatest lower bound of the set of values where \( A \) rejects some input string, i.e. \( \{ z \mid z \text{ is the probability of rejection of some input string by } A \} \), as illustrated in Figure 4.
The value of $y$, and hence the robustness, can be found within some accuracy by using binary search repeatedly employing forward algorithm for various values of $x$. More specifically, we can compute value of $y$ to be within a sub-interval $I$ of the unit interval $[0, 1]$, by using binary search and repeatedly invoking forward algorithm by giving an appropriate threshold in each iteration. The accuracy of the result is given by the length of the returned interval $I$. If the binary search is done for $k$ iterations then the accuracy will be $\Delta = 2^{-k}$. Equivalently, an accuracy of $\Delta$ can be guaranteed by iterating the binary search for $k = \log_2\left(\frac{1}{\Delta}\right)$ number of times. If forward algorithm decides $L_{>u}(A)$ is empty, $L_{>l}(A)$ is non-empty, and $(ub - lb) \leq \Delta$ where $\Delta$ is the user-specified accuracy, it terminates and reports a robust interval of $[1 - ub, 1 - lb]$.

Although backward algorithm is less efficient for the verification problem, it can be used to compute the exact value of the robustness in some cases. Suppose backward algorithm reaches a fixed point after $k$ iterations, then $y = \text{Prob}([q_0], k)$ where $q_0$ is the initial state of $A$. In this case, robustness has the exact value $(1 - y)$.

Further introduction on robustness can also be found in Section 7.1.
4.5 Verifying Acyclic HPA

An *acyclic* automaton $A = (Q, q_0, \delta, Q_f)$ has no loops other than absorbing states - accepting state(s) in $Q_f$ and rejecting/error state(s) (also called *deadlocks*\(^1\)) in $Q - Q_f$. It has a structure like Directed Acyclic Graph (DAG) except for the self-loops on absorbing states. Its depth $d$ - the distance between $q_0$ and any absorbing state - is at most $n - k - 1$ if it’s a $k$-HPA. An acyclic 1-HPA example is given in Figure 5, where self loops imply absorbing states. It’s not difficult to see that all accepting states can be merged into one accepting state, all error states can be merged into one error state, and such merging will create an equivalent automaton.

![Figure 5. An Acyclic 1-HPA Example.](image)

4.5.1 NP-Completeness

The problem of checking emptiness for 1-HPA is shown to be in EXPTIME by our results. This has been shown to be PSPACE-hard in (6). We show that for acyclic 1-HPA and also for any acyclic HPA, it is only NP-complete.

\(^1\)Strictly speaking, anticipated failures lead to an “error state”, while unanticipated failures will lead to a “deadlock”. Here we mix the use of the two terms without causing misunderstanding.
Given an acyclic HPA $A$ and a probability value $x$ such that $0 \leq x \leq 1$, it is easy to see that there is an input string that is accepted with probability $p$, iff there is a finite input string $u$ of length $\leq n$, where $n$ is the number of states in $A$, accepted with probability $p$. A non-deterministic algorithm, running in polynomial time, guesses a string $u$ whose length is at most $n$, and then computes its acceptance probability, and accepts it iff the probability of $A$ accepting $u$ is $> x$. It is not difficult to see that the probability of acceptance of $u$ can be computed in time polynomial of $n$ and $r$, where $n$ is the number of states in $A$ and $r$ is the number of bits required to represent the transition probabilities of $A$, i.e., each transition probability of $A$ is represented as a fraction $\frac{y}{z}$, where $y$ and $z$ are integers whose binary representation has at most $r$ bits. This is so because, all the intermediate probabilities computed are fractions of the form $\frac{y}{z}$, where $y$ and $z$ are integers whose binary representation has at most $O(nr)$ number of bits.

Now, we show the above problem to be NP-hard by reducing the 3-CNF satisfiability (i.e. 3SAT, a well-known NP-complete problem). Consider a 3-CNF formula $f$ given by $C_1 \land C_2 \land \cdots \land C_n$. Let $x_1, x_2, \cdots, x_m$ be the variables appearing in $f$. We construct an acyclic HPA $A$ whose input alphabet is the set $\{a, x_1, \neg x_1, \cdots, x_m, \neg x_m\}$; that is, the input alphabet has a special symbol $a$, together with literals corresponding to the variables appearing in $f$.

First we construct $n+1$ deterministic automata $A_0, A_1, \cdots, A_n$. The automaton $A_0$ ensures that the input string starts with $a$, followed by $n$ symbols which are literals corresponding to the $n$ variables $x_1, \cdots, x_n$ in that order. More specifically, it ensures that the first symbol is $a$ and the $(i+1)$-st symbol is either $x_i$ or $\neg x_i$, for $i = 1, \cdots, m$. If this is not satisfied, it goes to a rejecting state, otherwise it goes to an accepting state after the $(m+1)$-st symbol. Intuitively,
\( A_0 \) ensures that the \( m \) input symbols coming after \( a \) denote an assignment of values to the variables \( x_1, \ldots, x_m \). The automaton \( A_j \), for \( 1 \leq j \leq n \), checks that the input assignment given by the first \((n + 1)\)-st symbols satisfies clause \( C_j \). If \( C_j \) is \( y_{j,1} \lor y_{j,2} \lor y_{j,3} \), then \( A_j \) goes to an accepting state if at least one of \( y_{j,1}, y_{j,2} \) and \( y_{j,3} \) appear in the first \( n + 1 \) symbols; otherwise it goes to a rejecting state. Also, \( A_j \) goes to a rejecting state on the input symbol \( a \). It is easy to see that the formula \( f \) is satisfiable iff there is at least one input string that is accepted by all the deterministic automata \( A_0, A_1, \ldots, A_n \).

The HPA \( A \) is constructed from \( A_0, A_1, \ldots, A_n \) as follows. From its initial state on input \( a \) it has transitions with probability \( 1/(n + 1) \) to the initial states of \( A_0, A_1, \ldots, A_n \). On all other input symbols, from its initial state, \( A \) goes to the rejecting state. All the accepting and rejecting states in all the automata are absorbing states. It can easily be seen that the formula \( f \) is satisfiable iff \( A \) accepts at least one input string with probability 1, i.e., with probability greater than \((n/n + 1)\).

4.5.2 Other EXPTIME Verification Algorithms

Given a deterministic acyclic 1-HPA, other than forward algorithm and backward algorithm, we introduce briefly a few other EXPTIME algorithms here. They can also be applied to general PA that are decidable for the language emptiness problem.

4.5.2.1 Verification By Construction

We construct a product automaton by straight-forward reachability analysis.
• Step 1: Reduction (optional). Remove all states and transitions that are not on a path from $q_0$ to an accepting state. This can be done in linear time using an algorithm like SCC construction. For example, $g_0$ in Figure 5 is reduced to $g_1$ in Figure 6.

Figure 6. Model Checking Acyclic HPA Step 1.

• Step 2: Power-set construction. From the reduced PA $A = (Q, q_0, \delta, Q_f)$ on input alphabet $\Sigma$, we construct the power-set graph $A_P = (Q_P, q_{0P}, \delta_P, Q_{fP})$, where the state space is $Q_P = 2^{Q \times [0,1]}$, the initial state is $q_{0P} = \{(q_0, 1)\}$, the transition relation is $\delta_P(\{(q, pr)\}, a \in \Sigma) = \{(q', pr \times \delta(q, a, q')) | \delta(q, a, q') > 0\}$ in $Q_P \times \Sigma \rightarrow Q_P$, and the set of final states are those constructed with only final states in $Q_f$ of $A$ and they are all absorbing.

The construction can be done breadth first, or level by level more precisely. All set states from previous levels in $A_P$ can be discarded if the language information is not needed. Or else on each input string there will be at most one run.

Whenever we reach a final state, we sum up all probabilities to obtain the probability for $A$ to accept the input string. If the sum probability is 1, we stop immediately; or else we keep construction until all current states are final.
For example, $g_2$ as in Figure 7 is the constructed automaton for $g_1$ in Figure 6. States and transitions in dashed lines are only for demonstration; they will not be generated because of the existence of state $\{(2 : 1)\}$. Note in $g_2$, number of nodes on level $i \leq$ number of nodes on level $(i - 1)$ times $s$. In each node, number of elements $\leq (n - 1)$.

![Figure 7. Model Checking Acyclic HPA Step 2.](image)

The time complexity of this algorithm is $O(n s^d)$ in the worst case. While checking language emptiness with a given threshold, the algorithm may terminate early yielding a non-empty result, like in the example given above.

Note Step 1 is not irremovable as it may not improve Step 2 significantly, but it is inexpensive compared to Step 2, and it does make it easier to decide when to end construction and reduce the size of constructed graph in Step 2. Another optional reduction we can do in Step 1 is to merge non-distinguishable states, similar to that in deterministic finite-state automata minimization.
Using Hopcroft’s algorithm (20), the worst-case time complexity is $O(ns \log(n))$, where $n$ and $s$ is the sizes of state space and input alphabet respectively.

4.5.2.2 Verification By Language

Other than storing graph structure, we can store accepted strings as an alternative method, calculate the probability of accepting each string and then find the maximum non-emptiness threshold. To do this, define the probability of accepting input string $\alpha$ from state $q \in Q$ as $P_{q,\alpha}$. Then the maximum non-emptiness is $\max_{\alpha} \{P_{q_0,\alpha}\}$ where $\alpha$ is an accepted input string. We start from states in $Q$ with minimum out-degree and go backward to calculate for each state the set of all accepted strings and their acceptance probabilities until we reach $q_0$. When all successor nodes of a node has been taken care of, calculate accepted strings and probabilities for the node using the formula $P_{p,\alpha} = \sum_q \delta_{a(p, q)}P_{q,\alpha}$ where $q \in \text{post}(p, a)$. This requires string comparison and optionally sorting. This algorithm has worst time complexity $O(s^d)$ since that is the number of possible accepted runs.
CHAPTER 5

IMPLEMENTATION

In this chapter we will introduce our implementations on processing PA and prepare 1-HPA for verification. Some ideas were published in previous paper (6) but largely expanded here.

5.1 PA Validation

Since we have posted several restrictions on the type of HPA we can handle, when a PA is provided, we first need to execute a series of validations. If restrictions are violated, there are two types of messages reported - ERROR and WARNING. When an ERROR message is returned, the given PA is discarded; on the other hand, a WARNING message can be ignored and the input PA can still be processed, but allowing some chances of inaccurate results. Reporting more ERROR messages instead of WARNING messages can enhance robustness, but can also reduce usability, so a balance is needed in implementation.

- Initial node: There should be exactly one initial state in the given PA. Initial states can be identified either by the special state proposition “INITIAL”, or zero incoming degree (i.e. no incoming edges to a state). If there is no initial state specified, report ERROR. If there is more than one initial state defined with proposition “INITIAL”, the last one will be recognized as the initial state. When reading a PA from PRISM outputs, we use either the last state with label “INITIAL” or the first state with zero incoming degree as the initial state.
• Final state(s): without final states, no input strings will be accepted, and the language accepted by the given PA is always empty. In this case, a WARNING message is returned.

• Reachability: If not every state is reachable from the initial state, either an ERROR message or a WARNING message can be reported.

• Determinism: If there is not a transition relation defined for every (state, input) pair, the given PA is not deterministic. Then it will be determinized by generating and leading every missing transition to a preserved “ERROR” state.

Besides the restrictions posted above, principle examinations are also done while loading the PA source files, such as checking if a transition probability is valid, if the probabilities sum up to one in each transition distribution, whether a transition relation from a state on an input is duplicate and so on.

5.2 PA Reduction

Given a PA \( A \) over input alphabet \( \Sigma \), we can reduce its size by removing: 1) states (and related transitions) unreachable from the initial state; 2) states (and related transitions) that are not on a path from the initial state to any valid accepting state.

Such reductions will not reduce the verification problem size tremendously, since these removed components are irrelevant to generating witness sets. However, they can assist locating irrelevant components in system models.

5.3 HPA validation

In this section, we will introduce how to check if a given PA \( A = (Q, q_0, \delta, Q_f) \) over input alphabet \( \Sigma \) is an HPA, assuming every node in \( Q \) is reachable from \( q_0 \).
As introduced in Section 3.1.1, we construct the component graph $F_A = (C, E)$ of $A$. A SCC node $C \in C$ is said to be conflicting iff $\exists a \in \Sigma$ and $\exists q, q_1, q_2 \in C$ such that $q_1 \neq q_2$ and $\delta(q, a, q_1) > 0$ and $\delta(q, a, q_2) > 0$.

**Theorem 5.1.** A PA $A$ is an HPA iff there are no conflicting nodes in $C$, i.e., there are no conflicting SCCs in $G_A$.

This theorem is trivial based on the observation that all nodes in each SCC node $C \in C$ are on the same level, thus its proof is omitted.

### 5.4 HPA Leveling

Leveling an HPA is essentially partitioning or classifying its states into different levels. The minimum $k$ is fixed for an $k$-HPA, but there can be various leveling algorithms guaranteeing this $k$. For example, in the 1-HPA given in Figure 8, it’s fixed that states $\{0, 1\} \subset Q_0$ and states $\{2, 3\} \subset Q_1$, but one of states 4 and 5 can be on $Q_0$ instead of both on $Q_1$. The difference is whether to push more states on $Q_1$ or to leave more states on $Q_0$. This does affect the generation of good witness sets: depending on the specific instances, either leveling option can get the better result, i.e. fewer witness sets thus leading to less time consumption. However, the leveling will not affect the correctness of verification results.

![Figure 8. HPA Leveling Illustrations. Two Options for State 4.](image-url)
5.4.1 Deciding Minimum $k$ for HPA

As introduced in paper (6), for an HPA $A = (Q, q_0, \delta, Q_f)$, we let $\text{Min}_\text{level}(A)$ be the minimum $k$ such that $A$ is $k$-HPA, and introduce the following algorithm to compute this value. For any $Q' \subseteq Q$ and any $q \in Q$, $q$ is deterministic with respect to $Q'$, if for each $a \in \Sigma$, there is at most one state $q' \in Q'$ such that $\delta(q, a, q') > 0$.

We construct the component graph $F_A = (C, E)$ of the underlying directed graph $G_A$ of $A$. An SCC $C \in C$ is deterministic with respect to $Q'$, if every state in $C$ is deterministic with respect to $Q'$. For any sub-graph $H$ of $F_A$, let $\text{States}(H)$ be the union of all SCCs $C$ in $H$, and $\text{TD}(H) \subseteq \text{States}(H)$ be the set of all SCCs $C$ in $H$ such that all nodes in $H$, which are reachable from $C$ (including $C$), are deterministic with respect to $\text{States}(H)$. Note that all terminal nodes of $H$ are in $\text{TD}(H)$. Let $\text{Level}_\text{seq}(A)$ be a maximum length sequence $(H_0, H_1, \ldots, H_\ell)$ of non-empty subgraphs of $F_A$ such that $H_0 = F_A$, for each $i, 0 \leq i < \ell$, $H_{i+1}$ is the subgraph of $H_i$ obtained by deleting all nodes in $\text{TD}(H_i)$. Since $\text{Level}_\text{seq}(A)$ is a sequence of the maximum length, it is easy to see that every node in $H_\ell$ is a deterministic SCC with respect to $\text{States}(H_\ell)$.

Now, we have the following theorem.

**Theorem 5.2.** If $\text{Level}_\text{seq}(A) = (H_0, \ldots, H_\ell)$ for HPA $A$ then $\text{Min}_\text{level}(A) = \ell$.

This can be easily proved using contradiction, thus the proof is omitted. Using standard graph algorithms, computing $\text{Level}_\text{seq}(A)$, $\text{Min}_\text{level}(A)$ and partitioning the states of $A$ into different levels can be done in time $O(\text{Min}_\text{level}(A) \cdot (n + m))$. Furthermore, if we partition $A$ as a $\ell$-HPA, then $Q_i = \text{TD}(H_{\ell-i})$ for $0 \leq i \leq \ell$ is one applicable partitioning. For example, the HPA in Figure 8 will be partitioned as presented in Figure 9.
5.4.1.1 Implementation

To implement the algorithm introduced above on the component graph, we assign a value \( k_{\text{Max}} \) as the maximum/highest possible level to each SCC, then calculate minimum \( k \) based on \( k_{\text{Max}} \) values.

Step 1. Assign \( k_{\text{Max}} = |\mathcal{C}| - 1 \) to each terminal SCC.

Step 2. Go backward and decide the \( k_{\text{Max}} \) value \( k_{\text{Max}}^C \) for every remaining SCC \( C \in \mathcal{C} \) whose successor SCCs (excluding itself) in \( \cup_{a \in \Sigma} \text{post}(C, a) \subseteq \mathcal{C} \) have all been assigned \( k_{\text{Max}} \) values. Find the maximum \( k_{\text{Max}}^C \) for \( C \) that satisfies the conditions below.

- \( k_{\text{Max}}^C \leq k_{\text{Max}}^D \) where \( D \) is \( C \)'s successor on any input.
- If for any \( a \in \Sigma \), \( C \in \text{post}(C, a) \), then \( k_{\text{Max}}^C < k_{\text{Max}}^D \) where \( D \in \text{post}(C, a) \) and \( D \neq C \).
- Let \( k_{\text{MaxMin}}_{C,a} = \min(k_{\text{Max}}^D \mid D \in \text{post}(C, a), a \in \Sigma) \) be the minimum \( k_{\text{Max}} \) value for \( C \)'s \( a \)-successors, if more than one successor in \( \text{post}(C, a) \) has this minimum value, then \( k_{\text{Max}}^C < k_{\text{MaxMin}}_{C,a} \).

Figure 9. Deciding Minimum Level for HPA.
Step 3. The minimum \( k \) for \( \mathcal{A} \) is the difference between the maximum and the minimum of the kMax values of all SCCs in \( F_\mathcal{A} \). That is, \( k = \text{kMaxMax} - \text{kMaxMin} \) where \( \text{kMaxMax} = \max\{\text{kMax}_C \mid C \in \mathcal{C}\} \) and \( \text{kMaxMin} = \min\{\text{kMax}_C \mid C \in \mathcal{C}\} \).

Take the HPA \( \mathcal{A} \) in Figure 8 as an example. Each node of \( \mathcal{A} \) is in a different SCC, so the figure can represent the component graph \( F_\mathcal{A} \) at the same time. We decide its minimum \( k \) step by step as presented in Figure 10, and \( \mathcal{A} \) turns out to be a 1-HPA.

![Figure 10. Deciding Minimum k for HPA.](image)

5.4.2 Assigning Highest Levels

As introduced previously, all nodes in the same SCC are on the same level. So in assigning levels to states in an HPA, we first level SCCs in its component graph, then level each state the same level as its SCC.

Given a set of different leveling partitions \( \{\{Q - Q_1, Q_1\}, \{Q - Q'_1, Q'_1\}, \ldots\} \) on the same HPA \( \mathcal{A} \), there exists one partition with most \( Q_1 \) states: \( \{Q - Q_1 \cup Q'_1 \cup \cdots, Q_1 \cup Q'_1 \cup \cdots\} \). This is easy to prove using set theories, and it shows that there is a canonical way to represent an HPA. To achieve this, we simply use the kMax values calculated in deciding minimum \( k \) (see Section 5.4.1): for each SCC \( C \), assign \( (\text{kMax}_C - \text{kMaxMin}) \) as the level of \( C \).

For the HPA in Figure 8, assigning levels with most \( Q_1 \) states will lead to a partition of \( Q_0 = \{0, 1\} \) and \( Q_1 = \{2, 3, 4, 5, 6\} \), same as that presented in Figure 9.
5.4.3 Assigning Lower Levels

Below we will introduce an algorithm to assign lower levels to SCCs (and thus states). It is similar to that of deciding $k\text{Max}$ values in Section 5.4.1, but going on the opposite direction.

Step 1. Assign level 0 to the initial SCC.

Step 2. Go forward and decide the level $\text{level}_C$ for every remaining SCC $C \in \mathcal{C}$ whose predecessor SCCs (excluding itself) in $\bigcup_{a \in \Sigma} \text{pre}(C, a) \subseteq C$ have all been assigned levels. Find the minimum $\text{level}_C$ for $C$ that satisfies the conditions below.

- $\text{level}_C \geq \text{level}_D$ where $D$ is $C$’s predecessor on any input.
- If $|\text{post}(C, a)| > 1$ for $a \in \Sigma$, choose the one SCC with the minimum $k\text{Max}$ value in $\text{post}(C, a)$ and assign it the same level as $C$, any other SCC in $\text{post}(C, a)$ is on a higher level. Specially, if $C \in \text{post}(C, a)$, then the chosen one has to be $C$; else if there are more than one SCC in $\text{post}(C, a)$ with the minimum $k\text{Max}$ value, randomly choose one from them, and this is where different leveling results come from.

Take the HPA $\mathcal{A}$ in Figure 8 again as an example. Two different leveling results are shown in Figure 11, and there are even more possible leveling results not shown.

Figure 11. HPA Leveling Results Examples with More $Q_0$ States.
5.5 Obtaining Witness Sets for Finite Acceptance

Assume \( A = (Q, q_0, \delta, Q_f) \) is a 1-HPA. In section 3.3, we defined the various types of witness sets and witness set backward traversal. For finite acceptance, we start from the set \( Q_f \) of final states and traverse backward. This gives us all necessary witness sets for backward algorithm which is based on set predecessors. For Forward algorithm, missing set successors will be generated on the air. So it’s possible for the same HPA forward algorithm generates more witness sets than backward algorithm does.

Equivalently, we construct a standard non-deterministic automaton \( A^{-1} \) which is a reversal of \( A \), i.e., \( A^{-1} \) has the same set of states as \( A \); there is a transition from \( q \) to \( q' \) on input \( a \) iff \( \delta(q', a, q) > 0 \); the set of its initial states is \( Q_f \). The algorithm is similar to that of determinizing \( A^{-1} \) using standard subset-construction, and its complexity is \( O(wsmn) \) where \( n = |Q| \), \( m \) is the number of transitions, \( s = |\Sigma| \) and \( w = |X| \).
CHAPTER 6

MODEL CHECKING 1-HPBA

Given a 1-HPBA $\mathcal{A} = (Q, q_0, \delta, Q_f)$ over alphabet $\Sigma$, we introduce how to obtain its good witness sets and apply model checking in this chapter. We assume all states in $Q$ are reachable from $q_0$. Therefore, the language $L_{b \geq 0}^b(\mathcal{A})$ recognized by $\mathcal{A}$ is non-empty, iff there exists a final state lying on a directed cycle (or so-called lasso) with all transition probabilities equal to 1. That is, $L_{b \geq 0}^b(\mathcal{A}) \neq \emptyset$ iff there exists a non-trivial Strongly Connected Component (SCC) in $\mathcal{A}$ which contains a final state and is constructed using only probability-1 transitions. We call such SCCs as final SCCs, or FSCCs for short. To decide this extreme-threshold emptiness problem, a simple linear-time searching algorithm will work.

Observe that every accepting run eventually ends in one FSCC. More than one lasso may be involved in one accepting run but they are all in the same SCC, and there is at least one final state in the involved lasso(s). Note that with multiple lassos irregular languages can be expressed, such as the FSCC given in Figure 12. This FSCC $S = \{q_1, q_2, q_3\}$ has final states $\{q_2, q_3\}$ and two lassos $C_1 = \{q_1, q_2\}$ and $C_2 = \{q_1, q_3\}$ accepting different languages. By concatenating different numbers of $C_1$ and $C_2$ in arbitrary order and constructing a string that is not ultimately periodic, this FSCC can accept irregular language.

![Figure 12. Irregular Language Acceptance in PBA.](image-url)
For non-extremal threshold $x$ with value $0 < x \leq 1$, the problem of checking whether $L^b_{\vartriangleright x}(A) = \emptyset$ where $\vartriangleright \in \{<, \leq\}$ is undecidable for general PBA but decidable for 1-HPBA. To decide this, similar to 1-HPFA, we can also use verification algorithms introduced in Section 4, but we will need to obtain witness sets differently from that introduced in Section 5.5.

Observing that only final states in FSCCs contribute in accepting input strings in PBA, we assume that every final state belongs to a FSCC in this chapter.

6.1 Observations on 1-HPBA

There are several fundamental observations on 1-HPBA.

**Theorem 6.1 (States in FSCCs).** Each FSCC state constructs a good B"uchi witness set.

This is trivial thus proof is omitted.

If a SCC has no out transitions, that is, every state in this SCC transits to a state in the same SCC with probability 1 on any input, we call it a **terminal SCC**.

**Theorem 6.2 (Terminal FSCCs).** All states in terminal FSCCs construct one good witness set in an HPBA.

**Proof.** Prove by construction. In each terminal FSCC $C_i$, we can find an input string $\alpha_i$ passing every state (including a final state) deterministically in the SCC. Since all terminal FSCCs transit within themselves on any input string, we can connect each such $\alpha_i$ for each terminal $C_i$ in any order, the constructed string will be one commonly accepted input string among all these terminal FSCCs, thus they together construct a good witness set. \qed
The good witness set constructed above may not be maximal. These terminal FSCC states together with other FSCC states can construct larger witness sets.

In a FSCC $S$, if $\exists \Sigma' \subseteq \Sigma$, such that $\forall a \in \Sigma', \forall q \in S$, $\delta(q, a) \in S$, we call $S$ as semi-terminal FSCC. To obtain this set $\Sigma'$ is trivial, as $\Sigma'$ is $\Sigma$ minus each input symbol $a$, such that there exists an out-transition from a state in $S$ to a state outside $S$ on $a$. We also remove all transitions within $S$ on $a$, and then we can decide if $S$ is still strongly connected on $\Sigma'$, if that is the case, $S$ is a semi-terminal FSCC.

**Theorem 6.3** (Semi-Terminal FSCCs). *Each semi-terminal FSCC is a good witness set.*

*Proof.* Semi-terminal, again, is a sufficient but not necessary condition for good witness sets. “$\Rightarrow$” is easily proved by construction, while “$\Leftarrow$” can be proved by giving a counter example, such as the one given in Figure 13. FSCC $S=\{q1, q2, q3, q4\}$ has final states $q2$ and $q4$, and $\Sigma=\{a, b\}$. With $\Sigma'=\{a\}$, $S$ is still a good witness set, but no longer a SCC.

![Figure 13. Semi-Terminal SCC Example.](image)

From known good Büchi witness sets, we can do backward traversal (recall Section 3.3.1) to obtain all other desired witness sets. For HPFA we start from the set of $Q_f$, but for HPBA the challenge remains in obtaining all maximum good sets to begin with.
6.2 Flag Construction

Given 1-HPBA $A = (Q, q_0, \delta, Q_f)$ over input alphabet $\Sigma$, we employ the algorithm of flag construction to decide if a set $W$ of state makes a good witness set. The basic idea is to search for accepting lassos passing $W$ - this is achieved by starting from $W$, searching breath-first for lassos and meanwhile flagging final states at each cardinality of set nodes along the path. There are two versions of this algorithm, based on two different data structures selected.

6.2.1 Flag Construction Using Vectors

Using vectors/arrays to denote candidate sets is more straightforward, since cardinality matters in flagging. Given a candidate witness set $W = \{w_0, \ldots, w_{k-1}\} \subseteq 2^Q$ of size/width/cardinality $k$, we attach one flag bit to each cardinality in $W$ to mark final states. $W$ is a good witness set iff on every cardinality there exists a run of $A$ passing some final state on the same input string and comes back to $W$.

We construct a PA $A' = (Q', q'_0, \delta')$ on $\Sigma' = \Sigma \cup \{\epsilon\}$ as follows:

- $Q' \subseteq C \times \{0, 1\}$ where $C \subseteq Q$ is the vector state space, and the $\{0, 1\}$ domain flags final states on each cardinality;
- $q'_0 = < (w_0, b_0), \ldots, (w_{k-1}, b_{k-1}) > \in Q'$ is the initial vector state - the candidate $W$ with flags, where $b_i = 1$ iff $w_i$ is in a FSCC of $A$;
- $\delta' = Q' \times \Sigma' \rightarrow Q'$ is the transition relation with only probability 1 transitions. For $q = < \ldots, (w_i, b_i), \ldots >, q' = < \ldots, (w'_i, b'_i), \ldots > \in Q'$ and input $a \in \Sigma$, we have $\delta'_a(q) = q'$ iff $\forall i = 0, 1, \ldots, k-1, \delta_a(w_i, w'_i) = 1$ in $A$, $w_i$ and $w'_i$ are in one FSCC of $A$, and $b'_i = 1$ iff $b_i = 1$.
or $w'_i \in Q_f$, otherwise $b'_i = 0$.

Additionally, if $\forall i, b_i = 1$ in $q$, a reset transition is defined on $\epsilon$ as $\delta_\epsilon(<\ldots,(w_i,1),\ldots>)$

$= <\ldots,(w_i,b_i),\ldots>$ where $b_i$ is the initial flag value depending only on $w_i$.

One open question is to decide the set $C$, which will be covered in Section 6.3. Ideally, $C$ shall be the minimal state space for checking whether $W$ is a good witness set.

Note that every vector state in $Q'$ is of the size $k$. If on some input $a$, more than one state in vector state $q$ goes to the same state, then a transition on $a$ at $q$ is undefined in $A'$.

A set $\{\ldots,w_i,\ldots\}$ with at most one $Q_0$ state obtained from a vector state $<\ldots,(w_i,b_i),\ldots>$ is a good Büchi witness set of $A$ iff the vector state is in a lasso of $A'$ containing an $\epsilon$-transition. One example is given in subgraph (1) of Figure 14 where the set $\{q_1,q_2,q_3\}$ makes a good witness set of $A$. These lassos can be easily obtained by running lineal-time graph searching algorithms on $A'$. Observe that all states on one cardinality of such a lasso are from the same SCC of $A$, and further more, they are on a non-trivial directed circle containing at least one $Q_f$ state, which can be easily proved using contradiction.

An alternative of this flag construction method using vectors can be achieved by removing the flag domain from vector states but still tracking whether there is at least one final state on each cardinality of a lasso. An example is given in subgraph (2) of Figure 14. Using this method, we can also obtain partially good witness set where only a subset of all cardinalities are good together.
6.2.2 Flag Construction Using Sets

For the flag construction example given in Figure 14 above, if we use set states instead of vector states, we will have a much smaller state space in the constructed graph, as shown in Figure 15 below. The challenge is to track down final states on each cardinality.

Figure 15. Flag Construction Using Sets.

Given a candidate witness set $W = \{w_0, \ldots, w_{k-1}\} \subseteq 2^Q$ of size/width/cardinality $k$, we first construct a PA $A'' = (Q'', q_0'', \delta'')$ on $\Sigma$ as follows:

- $Q'' \subseteq 2^C$ where $C \subseteq Q$ is the set state space, and every set state in $Q''$ has same size $k$;
- $q_0'' = \{w_0, \ldots, w_{k-1}\} \in Q''$ is the initial set state - the candidate set $W$;
• $\delta'' = \mathcal{Q}'' \times \Sigma \rightarrow \mathcal{Q}''$ is the transition relation with only probability 1 transitions. For $q = \{\ldots, w_i, \ldots\}$ in $\mathcal{Q}''$ and input $a \in \Sigma$, $\delta''_a(q) = \{\ldots, w'_i, \ldots\}$ iff $\forall i = 0, 1, \ldots, k-1$, $\delta(w_i, a, w'_i) = 1$ and $w_i$ and $w'_i$ are from the same FSCC in $\mathcal{A}$.

The selection of set $C \subseteq \mathcal{Q}$ here is same as that in flag construction using vectors for the same $W$. Every good witness set constructed by states in $C$ of $\mathcal{A}$ has to be in a non-trivial SCC of $\mathcal{A}''$. So after $\mathcal{A}''$ is constructed, for each non-trivial SCC $\{S_1, S_2, \ldots\} \subseteq \mathcal{Q}''$ of $\mathcal{A}''$, we track final states by assigning colors. We start from any set node of this SCC, say $S_1=\{w_{1,0}, \ldots, w_{1,k-1}\}$, and color each $w_{1,i}$ with a unique color $i$. We then explore every node and all its out-transitions within the SCC using breadth-first searching and color the next nodes along the path. Every transition $\delta''_a(S_i)=S_j$ within this SCC - that is, $S_i$ and $S_j$ are both in this SCC - will be visited exactly once, and colors will be inherited by nodes in $S_j$ from nodes in $S_i$. If a node state is assigned more than one color, then these colors are treated as same, implying “twisting” of multiple connected cardinalities in this SCC.

Take the transition in Figure 16 as an example. Initially set state $S_1=\{w_1, w_2, w_3\}$ is assigned colors as $\{(w_1, \{1\}), (w_2, \{2\}), (w_3, \{3\})\}$. State $S_2=\{w_4, w_5, w_6\}$ inherited colors from transition $\delta''_a(S_1)=S_2$ as $\{(w_4, \{1\}), (w_5, \{2\}), (w_6, \{3\})\}$. After exploring transition $\delta''_b(S_1)=S_2$, we have $S_2$ colored as $\{(w_4, \{1, 2\}), (w_5, \{1, 2\}), (w_6, \{3\})\}$, implying the equivalence of colors 1 and 2.
Meanwhile, we maintain a set of “final colors” containing all colors that were assigned to a final state of $A$. At the end of the SCC exploration, in each set state of the SCC, every node state with a final color together - if containing at most one $Q_0$ state - will construct a good witness set. Similar to flag construction using vectors, each cardinality with a final color are also from the same FSCC of $A$, and further more, they are on a non-trivial directed circle containing at least one final state.

Comparing the two flag construction methods using vector and using sets, both have lineal-time complexity with respect to the number of candidate witness sets. However, using sets can save state space, as for each set state of cardinality $w$ there can potentially be $w!$ corresponding vector states. What’s more, we can find good witness set from a partially-good candidate using sets, but not while using vectors.
6.3 Framework to Find Candidate Witness Sets

Now that given a candidate set we know how to decide whether it is a good witness set using flag construction, we need to obtain the original candidate witness sets when initially a 1-HPBA is given. To do this we introduce two alternative frameworks in this section.

Keep in mind that to ensure the correctness of the verification algorithms based on good witness sets, we must find all maximal good witness sets. Some non-maximal ones can also be included, but they are not critical in calculating the values of those quantities used in the verification algorithms.

6.3.1 Bottom Up Method

Observe that any subset of a good witness set is a good witness set, although it may not be a maximal one. Therefore we can start from the smallest good Büchi witness sets and keep generating larger ones by merging smaller ones. By merging we mean union and flag construction using either vectors or sets (see Section 6.2), involving only probability-1 transitions within same FSCCs. If and only if two good Büchi witness sets have common accepted language, we say they can merge. Merging is a binary relation. It’s symmetric, reflexive, but not transitive. If two good witness sets A and B can not merge, then any good Büchi witness set overriding A cannot merge with any good Büchi witness set overriding B. Given three good Büchi witness sets A, B, C, even if we can merge A and B, merge B and C, and merge C and A, we may not be able to merge A, B, C all together.

Based on the observations above, we design a bottom-up framework as below.

Step 1. Initially, each FSCC state constructs a size-1 good Büchi witness set. Add them to WS.
Step 2. We keep merging good Büchi witness sets of size 1 with size 1, 2, \ldots, k, \ldots to get good Büchi witness sets of size 2, 3, \ldots, k + 1, \ldots until no good Büchi witness set of a greater size can be obtained. Record all new witness sets.

**Complexity.** Given 1-HPBA $A = (Q, q_0, \delta, Q_f)$, let the number of size-k good witness sets obtained is $w_k$, then $w_1 \geq 1$ is the number of FSCC states, and $w_{i+1} \leq w_1 w_i \leq w_1^{i+1}$. Let the algorithm’s total running time be $T_{bu}$, $M$ be the size of the largest good witness set obtained, and $w$ be the total number of good Büchi witness sets obtained, then we have $w = w_1 + w_2 + \ldots + w_M$, and $T_{bu} \leq w_1 + w_1 w_1 + \ldots + w_1 w_{M-1} \leq w_1 w \leq nw$ where $n$ is the size of the state space of $A$. That is, $T_{bu} = O(nw)$ is in polynomial time of $n$ and $w$.

Using this method we record every subset of a good witness set, and this is the major shortcoming of this method: there are too many non-maximal witness sets. We can do some tweets to improve the method a little bit such as the following.

- If there is one size-(k + 1) good Büchi witness set, there are at least $k + 1$ size-k good Büchi witness sets. Therefore, if the number of size-k good Büchi witness sets is less than $k + 1$, we know $k$ is the largest possible good witness set and no need to test on $k + 1$.

- Record for each FSCC state its acceptable inputs - on which there is a transition leading back to the same FSCC, and the set of its potential merge-able FSCC states, which are obtained and recorded when merging size-1. Such information can be used to reduce the state space when building higher levels.

- For each good Büchi witness set, we union the commonly acceptable inputs among all states as CI, and the sets of merge-able states as PAS. For every pair of good Büchi
witness sets, only if each good Büchi witness set is a subset of the other good Büchi
witness set’s PAS can they possibly merge, and we only need to consider the intersection
of their acceptable inputs when testing merging.

6.3.2 Top Down Method

Top-down method is very similar to flag construction using sets (Section 6.2.2). We start
from the initial set state containing all FSCC nodes of 1-HPBA \( \mathcal{A} = (Q, q_0, \delta, Q_f) \), and construct
a PA with state space \( 2^{FSCC} \) where not all set states are of the same cardinality and FSCC
denotes the set of all FSCC states in \( \mathcal{A} \). In constructing the graph, from a set state \( q = \{w_1, \ldots, w_k\} \in 2^{FSCC} \) on input \( a \in \Sigma \), the generated new set state will be \( q' = \{w'_i | \exists w_i \in q, \delta(w_i, a, w'_i) = 1, w_i \text{ and } w'_i \text{ are from the same FSCC of } \mathcal{A}\} \). It’s possible more than
one node in \( q \) goes to the same node on \( a \) in \( q' \), and it’s also possible some node in \( q \) has no
satisfying \( \delta \)-transition on \( a \) in \( \mathcal{A} \), therefore the size of \( q' \) can be strictly less than or equal to
that of \( q \). In this case, we will add the new node \( q' \) to the graph, but not the transition, since
it will not contribute in detecting SCCs.

After building the complete graph, we detect SCCs and do coloring using the same method-
ology as that in flag construction using sets to find good witness sets.

We basically expand the graph using breadth-first, and the sizes of newly generated set
states are non-increasing. So an improvement will be, we push further “level by level”: once we
have obtained all set states of current greatest size \( k \), we pause and construct component graph
to find and record good witness sets; then we discard all size-\( k \) set states and their transitions,
keep building on size \( k - 1 \).
**Complexity.** Theoretically, the top-down method is of $O(2^Q)$ time complexity in the worst case, but in practice it could hugely reduce the number of non-maximal good witness sets, thus leading to much better average time efficiency.

6.4 **Summary on Model Checking 1-HPBA**

To sum up, the complete procedure to obtain a 1-HPBA’s desired witness sets for verification is as below.

**Step 1 Validate and prepare.** Execute fundamental validity checks including reachability, determinism, final states existence and so on. Then we obtain all SCCs using a standard graph algorithm and mark FSCCS, after which we will also “clean” the set $Q_f$ of final states in $\mathcal{A}$ to contain only final states in FSCCs. Next we assign minimum leveling to SCCs and states as described in Section 5.4.

**Step 2 Starting from FSCCs, do Flag Construction using Top Down or Bottom Up methods to obtain “base” witness sets.**

**Step 3 Traverse backward from obtained base witness sets, and then pair $Q_0$ states with $Q_1$ witness sets to obtain all other good witness sets.**

Future works can be done on improving the bottom-up framework by reducing the number of trials. For example, if we do not increase level sizes one by one, but instead, treat mergable obtained on level 1 as a relationship, construct a graph and find maximum clique for trial. This could reduce the number of trials largely, but this improvement itself is also at least NP-hard.
CHAPTER 7

MODEL ABSTRACTION AND DOMAIN APPLICATIONS

Theoretically, our HPA-based verification techniques can be applied to any open probabilistic systems that can be partitioned into two layers. For safety-critical or highly-available failure-prone systems, the formal verification becomes even more important. Failures can occur in all types of systems, caused by various reasons from design faults to hardware crashes. The uncertainty of failures can be described as randomization, and modeled with probability distribution. In this chapter we introduce how we model failure-prone systems using PA. We will also introduce several safety-critical domains to present HPA model checking application.

7.1 Problem Modeling

As introduced in our paper (6), we consider concurrent systems deployed on multiple processors. They take inputs from the environment and consume inputs by going through a sequence of states. However, they are subjected to processor failures. Failures can happen independently to single processors. When failures occur, exception handling is executed and usually the load on the failed processor, i.e., the processes running on it, are transferred to the remaining processors. Thus after occurrence of a failure, the computation of the system changes. We model

\footnote{Strictly speaking, “fault/defect/bug”, “error/mistake”, and “failure” are different terms. First it could be (human) errors that cause faults (15); then a fault in the system leads to an error system state, which results in a subsequent system failure (21). Here in this report these terms will somehow be used mixed without causing misunderstanding.}
the failures with probability distributions and everything else as deterministic (with probability 1), and model the behavior as Open Probabilistic Transition Systems (OPTS).

An Open Probabilistic Transition System $T$ is a 6-tuple $(S, \Sigma, \eta, s_0, P, \phi)$ where $S$ is a set of states, $\Sigma$ is an input alphabet, $\eta : S \times \Sigma \times S \rightarrow [0, 1]$ is the transition relation so that for all $s, s' \in S$ and $a \in \Sigma$, $\eta(s, a, s')$ is a rational number and $\sum_{s' \in Q} \eta(s, a, s') = 1$, and $s_0 \in S$ is the initial state, $P$ is a set of atomic propositions and $\phi : S \rightarrow 2^P$ is a function assigning a set of atomic propositions to each state. We assume that $S, P$ are finite sets. Observe that $S, \eta, s_0$ and $\Sigma$ are similar to the corresponding components of a PA. However, $T$ has additional information given by $P$ and $\phi$ that label the states with atomic propositions. For a given OPTS $T$ as specified above, we can define a probability space $(S^\omega, E, \psi)$ where $S^\omega$ is the set of infinite sequences of states, $E$ is the set of measurable subsets of $S^\omega$ which is the $\sigma$-algebra generated by cylinders of the form $uS^\omega$ where $u \in S^*$, and $\psi$ is a probability function defined on it. As pointed out earlier, we use OPTSes to model failure-prone open concurrent systems. An input to $T$ is an infinite sequence $\beta \in \Sigma^\omega$. A computation $\sigma$ of $T$ on input $\beta$ is an infinite sequence of states $(s_0, \cdots, s_i, \cdots)$ starting from the initial state $s_0$ such that $\eta(s_i, \beta[i], s_{i+1}) > 0$ for all $i \geq 0$. We let $C(T, \beta)$ denote the set of computations of $T$ on $\beta$. For a computation $\sigma$ as given above, we let $\phi(\sigma)$ to be the sequence $(\phi(s_0), \phi(s_1), \cdots)$.

We consider the problem of verifying OPTSes against properties specified by deterministic automata. The inputs to a property automaton are elements of $2^P$. Formally a correctness specification $B$ for a OPTS $T$ as given above is a deterministic finite-state safety automaton $B = (R, \delta_1, r_0, F_1)$ where $R$ is a finite set of automaton states, $\delta_1 : R \times 2^P \rightarrow R$ is the next
state function such that \( \delta_1(\text{error}, c) = \text{error} \) for all \( c \in 2^P \), \( r_0 \in R \) is the initial state and \( F_1 = R - \{\text{error}\} \) where \( \text{error} \) is the unique absorbing state in \( R \) called the \textit{error} state, making \( B \) a Büchi automaton. As usual, we define a run \( \rho \) of \( B \) on an input \( t = (t_0, \cdots) \in (2^P)^\omega \) to be an infinite sequence of states \((r_0, r_1, \cdots)\) starting from the initial state \( r_0 \) such that \( r_{i+1} \in \delta_1(r_i, t_i) \).

Now, we define the probability of satisfaction of the property \( B \) by the \( \text{OPTS} \) \( T \) on an input sequence \( \beta \in \Sigma^\omega \), denoted by \( \Pr(T, \beta, B) \), to be the probability given by \( \psi(D) \) where \( D \) is the set of all computations \( \sigma \) of \( T \) on the input \( \beta \) such that \( \phi(\sigma) \) is accepted by \( B \). Note that \( 1 - \Pr(T, \beta, B) \) denotes the probability that \( T \) does not satisfy the property \( B \) on the input \( \beta \). The \textit{verification problem} for \( \text{OPTSes} \) is - given \( T, B \) as above and a rational \( x \in [0, 1] \) - determine if \( \Pr(T, \beta, B) \geq x \) for all \( \beta \in \Sigma^\omega \).

Now, we transform the above verification problem to checking emptiness problem for PA. Given \( T, B, x \) as specified above, we construct a PA, \( A(T, B) \), over the set of input symbols \( \Sigma \), which is a product of \( T, B \), and is defined as follows: \( A(T, B) = (Q, q_0, \delta, F) \) where \( Q = S \times R \), \( q_0 = (s_0, r_0) \), \( F = S \times \{\text{error}\} \) and \( \delta \) is defined as follows: for any \( s, s' \in S \) and \( r, r' \in R \), \( a \in \Sigma \), \( \delta((s, r), a, (s', r')) = \eta(s, a, s') \) if \( r' = \delta_1(r, \phi(s)) \), and is 0 otherwise. The following theorem can be easily shown.

\textbf{Theorem 7.1.} For any \( \text{OPTS} \) \( T = (S, \Sigma, \eta, s_0, P, \phi) \) and property automaton \( B = (R, \delta_1, r_0, F) \) over \( 2^P \) and rational \( x \in [0, 1] \), for all \( \beta \in \Sigma^\omega \), \( T \) satisfies the specification \( B \) with probability \( \geq x \) (i.e., \( \Pr(T, \beta, B) \geq x \)) iff for all \( u \in \Sigma^* \) the PA \( A(T, B) \) accepts \( u \) with probability \( \leq 1 - x \) iff \( L_{>1-x}(A(T, B)) = \emptyset \).
We say that an OPTS $T$ is a $k$-OPTS ($k > 0$) if its set of states $S$ can be partitioned into $k + 1$ sets $S_0, S_1, \ldots, S_k$ satisfying the following conditions: (a) $s_0 \in S_0$; (b) for every integer $i \in [0, k]$, every $s \in S_i$, and every $a \in \Sigma$, there is at most one state $s' \in S_i$ such that $\eta(s, a, s') > 0$; (c) for every $i, j$ such that $0 \leq j < i \leq k$, for every $s \in S_i, s' \in S_j$, $\eta(s, a, s') = 0$.

$k$-OPTSes can be used to model web applications with at most $k$ processor failures. Obviously, if $T$ is a $k$-OPTS then $A(T, B)$ is a $k$-HPA. If $T$ is a 1-OPST, using the above theorem, we can verify that $\Pr(T, \beta, B) \geq x$ for all $\beta \in \Sigma^\omega$ by checking $L_{1-x}(A(T, B)) = \emptyset$. The latter property can be checked using the algorithms given in Section 4. We can define the robustness of $T$ with respect to the correctness specification $B$ to be the greatest lower bound of the set $\{\Pr(T, \beta, B) | \beta \in \Sigma^\omega\}$. This value can be computed as the robustness of $A(T, B)$ which can be computed as presented in Section 4.4.

Note there is an alternative way to model the problem based on the property specification. What’s described above is that we specify the desired correctness safety property on the computations of $T$ as a Büchi automaton $B$. To obtain $A(T, B)$, we incorporate $T$ with the complement $\bar{B}$ of $B$, which is a deterministic automaton that specifies the incorrectness property. Since safety properties (7) can be violated by a finite system execution, $\bar{B}$ accepting the complement of the language accepted by $B$ is defined on finite strings and $A(T, B)$ is defined with finite acceptance. Alternatively, we can define the incorrectness property as $B$ which is a non-safety property, and then construct a PA from $T$ and $B$ directly on infinite strings as a Büchi automaton.
What’s more, if the correctness property $B$ is a safety property as stated above, the incorrectness property $\bar{B}$ is defined on finite strings; but if $B$ is a non-safety property, $\bar{B}$ is defined on infinite strings. The finally composed PA should have the same acceptance type as that of the property automaton. So extra caution is needed in abstracting a system model in reality.

7.2 Exception Handling Block

An important assumption about the failure-prone systems we study is that there will be no more failure in a system if one failure has already occurred on the same control flow. It’s a broadly-used assumption since any failed parts are ideally isolated and failure damages are not expected to propagate. Here this assumption is to guarantee the generated HPA have at most two levels, that is, it’s a 1-HPA with decidable language emptiness.

To achieve this, we can add isolated redundancy to the system model. After a failure point introduces branches on some input in the flow, we split the transition absorbing this input, make a copy of the reachable part from failure point in the original model, and attach it to the failed branch while ensuring the failed branch will not interfere with normal flow. Simple PA demonstrations are given below in Figure 17, with one possible failure defined on input symbol $a$ with probability $f$. More examples are given in Chapter 9.

Figure 17. Adding Exception Handling: PA Examples.
In a modeling language with predicates or variables, such as the PRISM modeling language (13) for Markov Decision Process (MDP), we can achieve the “one-failure” assumption by adding an extra variable to “lock” the flow status. We can export the system state spaces in PRISM and convert them into PA using our HiPAM tool (see Section 8.1.2).

One typical PRISM code segment is presented below. The variable $s$ indicates system state, and $f$ indicates failure status. Other transition statements than these two remain same for both normal flow and exception handling flow.

\[
\begin{align*}
[a] & \quad s=0 \land f=false \rightarrow \text{//no failure ever occurred} \\
& \quad 2/3:(s'=1) \land (f'=false) \text{//normal flow} \\
& \quad + 1/3:(s'=1) \land (f'=true); \text{//failed branch} \\
[a] & \quad s=0 \land f=true \rightarrow \text{//useless if no path from s=1 to s=0} \\
& \quad (s'=1); \text{//f'=true}
\end{align*}
\]

7.3 Domain Applications

High-availability or safety-critical systems often require such designs (22; 23) that even if some component(s) failed (i.e. partial failure), the overall systems would still maintain functionality and operate at normal performance (or with minimized performance degradation) to avoid loss or dangerous situations, given that initially the systems are functioning properly. This property is called fault tolerance for computing systems.

Fault-tolerant computing systems are called robust if they somehow continue execution despite errors; they are fail-safe (or fail-secure, or fail-gracefully, or fail-soft in computing) (24) if systems function at a reduced level of performance or fail completely after failures, but still maintain harmless (or at least a minimum of harm) to other devices or personnel. The principles (25) of fault tolerance include no interruption after failure and isolating failed components to prevent propagation of the failures. To achieve fault tolerance at runtime, we can anticipate (or
observe system runs to conclude) exceptional conditions, refine system models and avoid/remove faulty behaviors at design phase - similar to the idea of manually refining model abstraction under the guidance of counter-examples (26). This could add expensive latencies to system time efficiency, so an alternative solution would be using duplications/redundancies to take over failed part(s) - called fail-over, which sacrifices space consumption instead. In any case, it would be useful if a system can reserve a safe mode (or safe point or checkpoint) right on or before failure point.

If we model failures probabilistically, we can obtain failure specifications by observation at runtime, and apply HPA-based techniques to model check the problems at design phase to obtain system robustness and counter examples if any. Some application examples will be presented below.

7.3.1 Web Applications

In this section we describe how we abstract HPA models of web applications. The procedure is about the same as that is described in Section 7.1 and that we have introduced in paper (6). We consider only server-side web applications implemented on multiple servers. Assume a server accepts and processes a single session at a time. Concurrent sessions are executed by different servers. Each server is abstracted as a labeled deterministic finite-state automaton whose inputs are user-submitted forms; these include initiating a new session and ending the current session. These automata each have a special input symbol T that denotes elapse of one unit time (of course, we use discrete time and assume synchronized clocks). The states of these automata are labeled with atomic propositions denoting their properties.
Next the process automata are composed and synchronized on the T symbol, during which a failure specification on a single session is incorporated. Only one server can fail, and its failure probabilities for each (state, input) pair are specified in the failure model. After a failure, the current session of the failed server is taken over by the good server until completed. During this period, the good server is executing its own session as well as that of the failed processor. The order in which these should be processed is also specified as part of the failure model. Once a good server completes the session of the failed server and its own current session, it will continue to accept one new session at a time and process it.

The property to be verified can be specified as a deterministic finite-state automaton on the executions of the failed session. Then composed session automata with failure is composed with the property to get an OPTS. Finally we abstract away the T transitions as these are not the actual user inputs. Using the approach given in Section 7.1, we obtain a 1-HPA.

Web service is our main application domain, and more details on how this model abstraction can be implemented including a complete example will be introduced in Section 8.1.1.

7.3.2 Java Exception Handling

In the Java programming language (27), an exception is a failure event that occurs during a program execution and disrupts the normal flow - control flow instructed by program statements plus data flow constructed of variable values. Such runtime exceptions can occur anywhere and numerously in one program. Instead of leaving program crashing uncontrolled when exceptions occur, Java programmers can predict exceptions, and then try to catch or throw them as graceful exits when they truly occur, so as to prevent data corruption. If a predefined exception is
caught while executing a function, commands/statements in the catch clause will be executed immediately, after which clause the program will continue executing the remaining code of this function. Note that the statement that triggered the exception will not contribute to data flow. On the other hand, exceptions can also be thrown out so that they will be taken care of by the outside function that called the current function, and the remaining code of the current function will not be executed.

If we model exception handling probabilistically, an exception can be specified as a distribution with two branches - either triggered (caught or thrown out) or not. The exception handling is an alternative flow to the normal program flow, thus generating a higher level in the generated HPA model. Figure 18 shows a simplified 1-HPA model obtained when verifying the correctness of a Java program which involves web connection and database connection.

Figure 18. Java Exception Handling Example (Partial).

Sample source code for this program in a single Java function is partially presented below.

```java
try {
    //web connection
    URL url = new URL("http://JavaExample.com/");
    URLConnection con = url.openConnection();
    con.connect();

    //read input from web page
    BufferedReader in = new BufferedReader(
            new InputStreamReader(con.getInputStream()));
    String line = in.readLine();
    ...
```
Another Java program for division calculation is presented below, which follows a *fail-dead* (or *fail-stop*) design and every exception will cause the program to end immediately. It can be modeled as 1-HPA presented in Figure 19. Note this program is partially open with respect to input values, and it's closed after the values of both $n[0]$ and $n[1]$ have been obtained.

```java
public static void division_input_failDead() {
    java.util.Scanner in = new java.util.Scanner(System.in);
    int[] n = new int[2];
    int i = 0;
    while (i < 2) {
        try { n[i] = Integer.parseInt(in.nextLine()); } 
        catch (NumberFormatException e1) {
            e1.printStackTrace(); break;
        }
        if(i!=1){ ++i; continue; }
        try{
            System.out.println(n[0]+"/"+n[1] + "=" + n[0]/n[1]); return;
        } catch (ArithmeticException e2) {
            //n[1]=0
            e2.printStackTrace(); break;
        }
    }
}
```
To sum up, when we model program exception handling, we can model source files directly as presented in the examples above. This is essentially adding failure probabilistically to the program’s highest-level Abstract Syntax Trees (AST) or Control-Flow Graph (CFG), and then transforming it to PA. Related tools include AST View in Eclipse for Java file, DMS for C programs and so on. Alternatively, there exist many compilers using code generation techniques that can parse source files into lower-level representation before building CFGs. There are also tools that explore the run-time state space of the program and build a reachability graph based on that, such as GRaphs for Object-Oriented VERification (GROOVE). Failures can also be introduced on such abstracted graphs to obtain PA models as long as failures and properties are properly specified.

7.3.3 Miscellaneous Applications

1-HPA techniques can be applied to many other application domains than those listed above. For example, in an automated trading systems as shown in Figure 20, trade orders generated from trading models will be preprocessed through a series of steps before getting submitted to exchange, during which traders can make changes and interfere anywhere by providing commands, making the preprocessing system an open system. Many such processes
are deployed on a distributed server system. Exceptions can happen anywhere. It’s essential to model check this procedure properly and ensure data integrity to avoid financial loss.

![Diagram of Automated Trading System Illustration](image)

**Figure 20. Automated Trading System Illustration.**

HPA model checking can also be applied in designing of file systems in computers. A file system is used to control data storage and retrieval. It’s the bridge between user applications and data storage devices such as hard disk drive, flash memory and so on. A simple file retrieval example is given in Figure 21, while file system processing a file retrieval request submitted by user applications or operating system, failures can occur if there is hardware failure in the storage system, or power shortage, or another interrupting command is received, etc. If the file system provides file access via a network connection, it can be even more failure-prone. In such cases, reliability become extremely important design considerations. If we model failures probabilistically, we can model the problem using HPA and check its robustness quantitatively.

![Diagram of File System Illustration](image)

**Figure 21. File System Illustration.**

There are many other domains we can apply HPA-based model checking, such as storage device test, robotics robustness test and so on. We will not list them all here.
HiPAM (28) is a model checker we developed for model abstraction and verification using HPA-based technologies. It is written in Java, and it is platform-independent. The architecture of HiPAM is shown in Figure 22.

As presented in Figure 22, HiPAM provides three different options:

1. user specifies an open finite-state concurrent system using the “HPA Generation” panel (Figure 23);
2. user specifies the system in PRISM using “HPA Generation from PRISM output” panel (Figure 24), and transforms the PRISM compiler output to HPA;

Figure 23. HiPAM UI for HPA Generation.

Figure 24. HiPAM UI for HPA Generation from PRISM output.
3. user loads an HPA and checks the emptiness of its language and/or its robustness by using the “HPA Analysis and Verification” panel (Figure 25).

![HiPAM UI for HPA Analysis and Verification.](image)

Figure 25. HiPAM UI for HPA Analysis and Verification.

All these options will be explained in this chapter.

### 8.1 HPA Model Abstraction

As introduced in Section 7.1, in abstracting HPA models from an open concurrent system to be checked, we combine specifications of normal processes, failure and property. Here we will introduce our implementation of this procedure in detail.

#### 8.1.1 HPA Generation in HiPAM

When the HiPAM panel “HPA Generation” presented in Figure 23, i.e. the first option, is used for abstracting HPA models, we assume that all processes/sessions are identical except
for naming, and this is typical among multi-thread systems (8; 9). Intuitively, each session is assumed to be deployed on a different server and all servers run on synchronous clocks. The generic open process representing a session is specified as process \( g_0 \) (see example in Figure 26), which has only probability-1 transitions.

![Figure 26. HPA Generation: Single Session \( g_0 \).](image)

To be simple, HiPAM implements two-session systems. From \( g_0 \), it creates two different instances \( g_1 \) and \( g_2 \). Except for the symbol \( T \) which denotes passage of one discrete time unit, the input symbols of \( g_1 \) and \( g_2 \) are disjoint and each corresponds to an external user input in one of the two sessions. \( g_1 \) and \( g_2 \) synchronize on the \( T \) symbol only. This model can easily be extended to arbitrary number of sessions and servers.

\( g_1 \) and \( g_2 \) are composed in the traditional way, with partial order reduction to reduce state space. The composed automaton \( g_{12} \) is shown in Figure 27, where \( g_1 \) has priority in partial order reduction. Every transition in \( g_{12} \) has probability 1.
HiPAM then takes a failure specification \( f \) (example in Figure 28) and incorporates \( f \) on \( g_1 \) to denote the occurrence of failure. We assume failure can occur to only one server, and we will assume that \( g_1 \) is the ongoing session of that failed server below. \( f \) specifies the probabilistic transitions on each input and state combination of \( g_1 \).

After failure occurs, we require the remaining working server to take over the partially completed session from the failed server before starting new sessions, and we provide two options here: either one process completes first then the other; or they each take one step interleavingly and execute one external input each. For example, if the defined after-failure priority is \( g_2 > g_1 \), then either 1) \( g_2 \) completes current session first, then \( g_1 \) completes current session, and finally \( g_2 \) can start new sessions, and this generates \( g_{21f} \) (Figure 29); or, 2) \( g_2 \) then \( g_1 \) will execute one external input alternatively, then \( g_2 \) can start new sessions, and this generates \( g_{21i} \). Note if there is no failure occurrence, \( g_1 \) can start new sessions as well.
As shown in Figure 29, the occurrence of failure has pushed some system states to a higher level, and the resulting composed system with failure is a 1-HPA. If in single process specification $g_0$ starting new session is allowed without failure, in $g_{21f}$ there could be more than one session following the first $g_1$ and $g_2$ sessions.

HiPAM also takes as inputs the property specification $g_p$ (Figure 30), which is an automaton defined over the executions of $g_1$. Note in defining $g_0$ and $g_p$, we are assuming the set of states absorbing T symbols and the set of states absorbing external inputs are disjoint, so as to
guarantee the correct timing logic. Any violation of this disjoint rule will fail the abstraction process. Therefore, $g_0$ and $g_p$ are not strictly deterministic according to our definition, as not a transition is defined on every input at each state. However, all defined transitions are given probability 1, and all missing transitions are assumed to go to an error absorbing state.

Next, the composition system with failure is composed with $g_p$ (Figure 31), after which the transitions on $T$ are absorbed into those of other input symbols (Figure 32). The resulting automaton is an HPA whose input symbols are those of $g_1$ and $g_2$ excluding the $T$ symbol. The final states of this HPA are those composite states whose component corresponding to the state of $g_p$ is a final state of $g_p$. 

Figure 30. HPA Generation: Property Specification $g_p$. 


Finally, HiPAM determinizes the obtained HPA (Figure 33) and outputs it. There are three
file formats available for storing HPA models: HPA files (.HPA, .txt or plain file) for HPA
reloading in verification, plain files for debugging, and FAT files for graphically presentation in an external tool called FAT (29).

Figure 33. HPA Generation: Final HPA - the Determinized $g_{21fpNoT}$.

8.1.2 HPA Generation from PRISM Output

Under the second option, user can specify the open concurrent system to be verified using PRISM as a MDP Model (see Chapter 2). In a single PRISM model, user can specify multiple session processes - $g_1$ with failure and one or more others without failure, and the property $g_p$ defined over the executions of $g_1$. Then user can use PRISM’s embedded function to compose
all processes to obtain the composed system as a PA. The composed PA is checked to ensure
that it is an HPA, otherwise an error message is thrown.

Using PRISM one can specify systems of different sizes easily. The most straight-forward
way is to make various numbers of duplicates of the sessions without failure. Depending on
the system to be modeled, concurrent processes can not only synchronize on the special $T$
symbol as defined in 8.1.1, but also asynchronously communicate via global shared variables.
However, using the PRISM modeling language, it’s hardly guaranteed that states processing
external inputs and states processing time consumption are disjoint, implying less precise control
over the composed model than using HiPAM modeling. What’s more, in HiPAM we provide
partial order reduction to reduce the problem size, while in PRISM, only symmetry reduction
is provided for fully component-symmetric asynchronous systems.

8.2 HPA Analysis and Verification

The implementation of HPA analysis and verification in HiPAM has been introduced in
Chapter 5 and Chapter 6. Here we give an example output on using HiPAM to model check a
valid 1-HPA. To decide the language emptiness or robustness of an HPA, simply load HPA file
in HiPAM, input parameters and click corresponding buttons. The log below is for the 1-HPA
generated in Section 8.1.1. It’s very small so CPU time spent is often zero. For non-empty
results there will also be an accepted trace example. However, to improve time efficiency, our
verifications using accumulating algorithms can terminate early and record an incomplete trace.

PA is a 1-HPA with 10 states.
$q_0 = 0$
$F(size 3)= [7, 8, 9]$
$Alphabet(size 5)= [2Sa, 1Sb, 2Sb, 2Snew, 1Sa]$
$F_0(size 0)= [];
F1(size 3) = F - F0.
Q0(3) = [0, 2, 4]
Q1(size 7) = all others.
WSs (WS0 size 8, WS1 size 1) //WS file name omitted

*Robustness Result:
L_{UpperBound} = 85899345920

BKD: R = 3 / 5 at L = 2, time = 0 ms;
FWD: R = (19 / 32, 77 / 128] at precision 1 / 100, time = 16 ms;

*Language Emptiness Result:
BKD: empty for x=9 / 10 at L=2
CPU Time: 0 ms.

Note: BKD Convergence

*Language Emptiness Result:
FWD: empty for x=9 / 10 at L=1
CPU Time: 0 ms.

Note: Forward Convergence

*Language Emptiness Result:
BKD: non-empty for x=1 / 10 at L=1
Trace Example: 0-1Sa->[3]
CPU Time: 0 ms.

*Language Emptiness Result:
FWD: non-empty for x=1 / 10 at L=1
Trace Example: [i=1][3]<-[ ]-[i=0]0
CPU Time: 0 ms.

Note: Val Termination

8.3 Experiments

In the experiments, we used examples obtained from server-side web applications using options 1 and 2 described in Section 8.1. In these examples we defined incorrectness property states that normal real-time requirement for session one is violated because of failure, and the obtained HPA are HPFA.

Results are shown in Figure 34 and Figure 35, presenting the time efficiency on robustness calculation and language emptiness checking respectively. Each data point represents one HPA obtained using a different combination of process and failure definition, and then leveled using one of the two strategies described in Section 5.4.
Figure 34. Experiment Results for Robustness.

Figure 35. Experiment Results for Language Emptiness.
To calculate robustness (Figure 34), forward algorithm can be faster for large systems, but backward algorithm can compute the exact robustness in most cases. For language emptiness (Figure 35) where each test case is verified against thresholds \((0.1, 0.3, 0.5, 0.7, 0.9)\), forward algorithm generally outperforms backward algorithm by orders of magnitude, but backward algorithm can perform better in case of non-empty result.

The difference between verifying HPFA and verifying HPBA exists only in obtaining witness sets. So in this section we included only data collected from verifying HPFA for which we have large system models.

8.3.1 Experiment Limitations

For the main performance indicator, we are using standard JDK utility for measuring CPU time to indicate time efficiency. CPU time is the total time spent using a CPU, and it’s user time plus system time, where user time is the time spent running program code, and system time is the time spent running OS code on behalf of the program (such as for I/O). The returned time value is of nanoseconds precision, but accuracies are often a few milliseconds.

Our programming platform is Eclipse Mars.2, on a test platform of Win8.1Pro64, CPU i7-5500u, JRE8u101, where HiPAM generated witness sets for HPA with up to a few hundred thousand states in a few minutes. However, in model checking such large HPA the memory limit 4GB was reached. In implementing the verification algorithms, we first compute and store some of the values that are repeatedly used in the algorithm, such as the sets \(W_{a,q}\) in the forward algorithm, for each \(W \in \mathcal{X}, q \in Q\) and \(a \in \Sigma\). This way we are sacrificing space efficiency to get better time efficiency. For a system model with 16713 good witness sets, the
memory consumption during robustness calculation is partially shown in Figure 36, and the peak memory usage is about 1900MB, in which about 500MB is consumed by Eclipse itself.

Another restriction on the programming side of the implementation is on the precision of probability values. We are using a fraction data structure to calculate and store probabilities, where denominators and nominators are stored as signed integers. For huge HPA with long runs there could be the integer overflow exception in math calculation, for example, running backward algorithm on some empty language emptiness results. In such cases we obtain double values of fractions as approximations when applicable, or else ERROR messages will be reported.
CHAPTER 9

EXTRACTING FAILURE SPECIFICATIONS

For a deterministic system modeled as $A = (Q, q_0, \delta, Q_f)$ with input alphabet $\Sigma$, a failure specification/model $f$ is a valid probability distribution defined either as $(Q, \Sigma)$ pairs, or purely on $\Sigma$. That is, $f \in (Q, \Sigma) \rightarrow [0,1]$, or $f \in \Sigma \rightarrow [0,1]$. Here we assume the valid failure profitability domain is $(0,1]$ instead of $[0,1]$, since probability 0 implies no failure at all.

In a partially observable system where system states are invisible and only variable value changes are observable, it’s most often the case $f \in \Sigma \rightarrow (0,1]$, and by observation we can define more reliable failure distributions depending solely on data flows. Therefore, one methodology to obtain precise failure specifications here is to jump between control flow with data and data flow, where we obtain failure probability distributions in each data’s own flow graph and then apply back to control flow graph.

In the area of business process management (30; 31) including workflow management (32; 33), there have been mature results on extracting data flows from the complete business processes by following a complete set of transformation rules (34). Also in (35) such data extraction has been extended for open models in collaborative environments (36), where collaborating processes are independently modeled, asynchronous and communicating by sending messages under agreed protocols. Although we can not guarantee in generating the original process model using the generated data models, it’s good enough for us to preserve key points of data processing.
during transformation, so that we can apply failure models from obtained data models back to
the original process model precisely.

Therefore, here we present how we can add exact failure models to certain process models,
and then transform them to probabilistic automata for model checking.

9.1 Process and Object Modeling

9.1.1 Process Modeling

There exist numerous models for business processes, from industrial standards to academics-
preferred algebra, whether with data operations modeling or not, their key control-flow struc-
tures are about the same.

A Petri Net (PN) as presented in Figure 37 is a tuple \( < P, T, F > \), where \( P \) is a finite
set of places/states, \( T \) is a finite set of transitions/events/tasks/activities/actions, \( P \cap T = \emptyset \)
and \( P \cup T \neq \emptyset \), and \( F \subset (P \times T) \cup (T \times P) \) is a set of arcs (or flow relations). Initially, there is one token
in the start place for one process instance. A transition will (and must) be triggered/fired iff
there is at least one token in every incoming place of the transition. After triggering a transition,
each previous/in place absorbs one token, and each next/out place gains one token.

A Workflow Net (WFN) is a sound PN with only one source/start place, one sink/end place,
and guaranteed reachability of all places (except start place) and transitions from the start
place. For each active workflow instance, there is one token in the start place at the beginning.

![Figure 37. Petri Net Demonstration.](image-url)
of time, so WfN can express concurrent instances running simultaneously. A Workflow Net with Data (WfDN) (37) is an extended WfN where data operations (e.g. read, write and destroy) are defined inside transitions. An open Workflow Net (oWfN) is another extended WfN in a collaborative environment where border/boundary/interface places and transitions are defined to specify collaboration among processes. In studying collaborative processes independently as open systems, we assume compliance is guaranteed, that is, all processes comply with agreed protocol, such that boundary input and output needs will all be satisfied.

An oWfN can also be extended with data to an open Workflow Net with Data (oWfDN). One example is given in Figure 38, presenting a buyer-seller interaction scenario (35) with two data objects - quote form q and order form o.

Figure 38. An oWfDN Example: Buyer-Seller.
A Business Process Model and Notation (BPMN) (38) is similar to a Petri Net on capturing the control flow of activities/actions by consuming tokens, but a BPMN can also capture the effects of business activities on data by specifying the input and output states of data objects for each activity. BPMN models can have interfaces and interact with one another, so they are often used in an embedded way. In this report we will look at simplified BPMNs in the sense that we keep only basic structural modeling elements and object-based conditions.

A simple BPMN example is demonstrated in Figure 39 for a claim processing scenario (34).

Gateway is a control flow structure in BPMN defined on activities as a diverging/splitting (one-to-many) or converging/merging (many-to-one) relationship. A valid BPMN model has no zero output flows, so at each gateway there will be at least one activated activity. Other than the XOR gateway shown in the example, there are two more core gateways defined in the BPMN 2.0 standard (38). Event gateways behave similarly to XOR gateways, except that
its conditions are often defined as the occurrence of specified events, while XOR gateways are mostly conditioned on objects states. Since we are interested in data objects, we will not look at event gateways here. Another core gateway is parallel gateway (or AND gateway), which forces execution of all outgoing activities (called fork) and synchronization of all incoming activities (called join). It’s commonly used in pair, that is, a join is used in a combination with a fork (using AND-AND gateways). Their notations and explanations are shown in Figure 40.

![Figure 40. BPMN Gateway Specifications (Selected).]

9.1.2 Object Modeling

While process models introduced above are mainly structured around control flows, an Object Lifecycle (OLC) model is data-oriented and demonstrating the complete lifecycle of one data object. It’s essentially a Finite State Machine (FSM) focusing on the dynamics of one object and modeling allowed object state transitions. That is, with each state uniquely defined in the graph, how the object is changed from one state to another by activities. Process models and object lifecycles represent overlapping behavior. An OLC example (34) is given in Figure 41, and it is extracted from the claim processing scenario given in Figure 39 by following a set of standard transformation rules: object creation, state change, and final consumption.

The OLC model can also be extended with boundaries as open Object Lifecycle (oOLC) (35), such as the examples in Figure 42 for the data object q extracted from the oWfDN model.
of the buyer-seller example in Figure 38. In the extraction process, internal states can be automatically obtained using transformation rules, but boundary states need to be manually regulated to ensure compliance with protocol.

Figure 41. The OLC Graph for the Claim Processing Example.

Figure 42. The oOLC Graphs for the Buyer-Seller Example.
### 9.2 Extracting and Applying Failure Specification

While observing a real life-cycle of a data object on the fly, an exception may be caught, i.e. the object failed to change from some status as expected. In practice, this failure probability can be defined as the number of failed observations divided by the total number of observations in numerous independently conducted repeatable experiments, and the confidence of accuracy will depend on the size of observation population, sampling strategy and so on.

After obtaining this failure probability \( f \) for object \( o \) at status \( o[f] \), we get the failure specification defined over data object. Then we may be able to add an exception handling block (see Section 7.2) directly to the process model by the following:

1. locate the activity (activities) \( e \) (e’s) that will update \( o \) from status \( o[f] \) to another status;
2. split \( e \)’s each input transition \( t_{c_{in}} \) on \( o[f] \) into two: one \( (t_{c_{in0}}) \) in the original process model with probability \((1-f)\), the other \( (t_{c_{in1}}) \) leading to the exception handling block with probability \( f \);
3. make a copy of the affected partial process model that comes after the failure point, and attach it to the exception handling block after transition \( t_{c_{in1}} \) properly.

For example, object Claim in Figure 41 may have some chance to fail evolving at status Claim[Granted]. Looking at the process model in Figure 39, it’s the Carry out payment activity that will first update Claim from Granted, thus we split the transition from Prepare settlement on Claim[Granted] to Carry out payment and add exception handling branch using approaches described in Chapter 7. The updated BPMN model is presented in Figure 43. Note
it’s extended with stochastic branches for presentation purpose, and it’s not strictly falling within the BPMN syntax.

Figure 43. The BPMN Model with Exception Handling for the Claim Processing Example.

Take the oOLCs in Figure 42 as another example. Suppose failure may occur at status \( q_{\text{accepted}} \) in the seller’s data flow oOLC\((S,q)\). The failure specification can also be obtained by observation and statistics, as described above. Back at the seller’s open process model oWFDN\((s)\) in Figure 38, the arc connecting the in-place and activity \( t_3 \) will be split into an or-split and linked to exception handling block, as presented below in Figure 44. Again this stochastic oWFDN model is just for presentation purpose and it’s not strictly an oWFDN.
Figure 44. The oWfdN Model with Exception Handling for the Buyer-Seller Example.

Note this failure specification procedure is just one applicable design for automatic implementation using computers. In reality, further investigate can be made by business owners into the actual activities that would be responsible for the data failures and add exception handling.

9.3 From Process Models to Probabilistic Automata

Note either workflow or business process, their modeling techniques introduced above share very similar control structures based on Petri-net and token consumption. Therefore, in transforming a process model to a PA, we will use BPMN terms for illustration, but same assumptions and rules apple to WfN as well.

We assume the original process models to be studied only have gateways of either XOR-gateways or AND-gateways. Input and output conditions of every activity or gateway are defined over data object states. XOR-gateways have mutually exclusive conditions, such that different instances have consistent control flow structure. AND-AND gateways are used in
pair to present concurrent and synchronous branches. Activities to be synchronized by AND-gateways should have different output data states.

While Petri net is expressible using higher dimensional automata (39), which is basically presenting all possible serialization combinations of the concurrent activities, it is more expressive than plain automata. So in our transformation, we are taking measures to remove concurrency but meanwhile preserve the same language emptiness results in model checking.

9.3.1 Transformation Rules

The basic rules to transform stochastic BPMN to PA are introduced in Table I. Since PA models do not support concurrency, we serialize all concurrent branches in the BPMN models in transformation. There could be various serialization results, but according to the theory of partial order reduction, whichever one shall work.

TABLE I

<table>
<thead>
<tr>
<th>BPMN</th>
<th>PA with Finite Acceptance</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>initial state</td>
</tr>
<tr>
<td>end</td>
<td>absorbing state (manually select accepting state)</td>
</tr>
<tr>
<td>action</td>
<td>state</td>
</tr>
<tr>
<td>object</td>
<td>input variable (object status as input value)</td>
</tr>
<tr>
<td>XOR-gateway</td>
<td>(removed)</td>
</tr>
<tr>
<td>AND-gateway</td>
<td>serialized states</td>
</tr>
</tbody>
</table>
Following the rules above, we get a PA model in Figure 45 for the claim processing example in Figure 43. In this example, specifying failure and transformation to PA model can be done in either order. But generally, we recommend transformation before adding failure. Note the accepting states can also be customized according to desired system property.

Figure 45. The PA Model for the Claim Processing Example.

A concurrent BPMN example is given in Figure 46. Its control flow can be transformed into the PA sequence $e_1, \{e_2, e_7\}, \{e_3, \{e_4, e_5\}, e_6\}$, with $e_4$ and $e_5$, the $\{e_2\}$-branch and the $\{e_3\}$-branch exchangeable.

Figure 46. A Serialization Example for Transforming BPMN to PA.
9.3.2 State Space Analysis

Using PRISM, failures can be introduced to processes models directly, then the whole state space is output and transformed to PA in HiPAM. This method can be useful especially if failure is found in concurrent process branches.

Alternatively, we can make use of reachability analysis in transforming process models. Take Petri Net as an example, we can use a vector to denote the token counts of all places or activation status of all activities (35), and rephrase the Petri Net as its reachability graph, then transform this graph to PA, which is trivial.

9.4 Summary on Extracting Failure Specifications

To sum up, we design the following procedure in Figure 47 to specify failure for business process models and transform them to PA for model checking.

Figure 47. The Procedure of Model Checking Business Processes.
Such Petri-net process models also have the problem of state explosion especially with concurrent activities. Applying reduction rules (40; 41) can help reduce the size of problems and handle bigger examples. But the reduction rate varies among different examples.
CHAPTER 10

UNDECIDABILITY FOR 2-HPA

We now demonstrate that the emptiness problem for 2-HPA is undecidable. This result was first introduced in our previous paper (6) and it is expanded and explained in more detail here.

Theorem 10.1. Given a 2-HPA \(A\), \(a \in \{f, b, m\}\), and \(\triangleright \in \{>, \geq\}\), the problem of determining if \(L_{>\, 2}^a(A) = \emptyset\) is undecidable.

Proof. We will modify the approach given in (18) where the non-emptiness problem for \(k\)-HPA was shown to be undecidable for \(k \geq 6\). The undecidability is established by reducing the halting problem of deterministic 2-counter machines to the non-emptiness problem of 2-HPA.

Let \(T\) be a deterministic 2-counter machine with control states \(Q\) and a special halting state \(q_h\). Without loss of generality, we make the following assumptions about \(T\): the operation \(T\) does on the counters (including zero and non-zero tests) is uniquely determined by it’s control state; each transition of \(T\) changes at most one counter as determined by the control state; and the initial counter values are 0. Recall that a configuration of such a machine is of the form \((q, a^i, b^j)\), where \(q \in Q\) is the current control state, and \(a^i (b^j)\) is the unary representation of the value stored in the first counter (second counter, respectively). We will refer to the first counter as the “a-counter” (since it is encoded using ‘a’ symbols) and the second counter as the “b-counter” (since it is encoded using ‘b’ symbols).
We will construct a 2-HPA $A_T$ with the property that its language is non-empty iff $T$ halts.

The input alphabet $\Sigma$ of $A_T$ will have 6 symbols - ‘,’ ‘(’, ‘)’, ‘a’, ‘b’, and ‘τ’. $A_T$ will have the following properties.

- The only words, $\rho$, accepted with non-zero probability are those that contain the symbol $\tau$, i.e., $\rho$ is of the form $\alpha\tau\beta$ where $\alpha \in (\Sigma \setminus \{\tau\})^*$ and $\beta \in \Sigma^* \cup \Sigma^\omega$.

- Consider $\rho = \alpha\tau\beta$, where $\alpha \in (\Sigma \setminus \{\tau\})^*$. If $\alpha$ is of the form $\alpha_1\alpha_2$, where $\alpha_1$ the (unique) halting computation of $T$ then $\rho$ will be accepted with probability $> \frac{1}{2}$. On the other hand, if $\alpha$ is either a proper prefix of a computation of $T$, or is invalid (i.e., either $\alpha$ is not of the right format or has an invalid transition), then $\rho$ is accepted with probability $< \frac{1}{2}$.

If we successfully ensure the above properties, then we would have $T$ halts iff $L^a_{\geq \frac{1}{2}}(A_T) \neq \emptyset$, for $a \in \{f, b, m\}$, and $\geq \in \{>, \geq\}$.

To implement the above ideas, the HPA $A_T$ on an input $\rho$ needs to check the following conditions.

1. $\rho$ is of the form $\alpha\tau\beta$, where $\alpha \in (\Sigma \setminus \{\tau\})^*$.

2. $\alpha$ is of the right format, i.e., it is a sequence of tuples of the form $(q, a^i, b^j)$.

3. The first configuration of $\alpha$ is the initial configuration of $T$.

4. The control state in successive configurations in $\alpha$ is in accordance with the transitions of $T$.

5. Successive counter values in the sequence follow because of a valid transition of $T$.

6. In the last configuration of $\alpha$, $T$ is in the halting state $q_h$. 
All the above checks, except (5), can be accomplished using a deterministic finite automaton (which is a 0-HPA). Thus, there is a deterministic finite automaton $B_T$ with a special, absorbing accept state such that the input words $\rho$ that cause $B_T$ to reach this unique accepting state are exactly those that satisfy all the above conditions, except possibly (5). Checking condition (5) will be accomplished by a 2-HPA $C_T$, which will be introduced next. Before presenting the details of $C_T$, we point out that the desired automaton $A_T$ is one that synchronously executes $B_T$ and $C_T$ on the input, and accepts when both accept. Notice that $A_T$ will be a 2-HPA, since $B_T$ is a deterministic finite automaton, and $C_T$ is a 2-HPA.

We now describe the automaton $C_T$. We will not give a formal, tuple-based definition of automaton $C_T$ because it will obfuscate the ideas behind the construction. The set of control states of $C_T$ will include $Q$ (the control states of $T$) and a unique, absorbing accepting state $\text{accept}$, and a unique, absorbing rejecting state $\text{reject}$. To outline the ideas behind the construction of $C_T$, we will only consider its behavior on inputs $\rho$ that satisfy conditions (1), (2), (3), (4), and (6), i.e., $\rho$ is a sequence of configurations $(q_0, a^{i_0}, b^{j_0})(q_1, a^{i_1}, b^{j_1})\cdots$, where each control state $q_{i+1}$ follows correctly from $q_i$ according to the transition determined by $q_i$; this is because $B_T$ ensures that any input that violates (1), (2), (3), (4) or (6) will be rejected with probability 1 by $A_T$ (independent of $C_T$’s behavior). On such a well formed input $\rho$, we will describe how $C_T$ processes each configuration $(q, a^i, b^j)$ in $\rho$ as it is encountered. After reading ‘(q’ of the configuration $(q, a^i, b^j)$, the “main” thread of $C$’s computation will be in state $q$. If $q$ is the unique halting state $q_h$ of $T$, $C_T$ will transit to $\text{accept}$ with probability 1 on reading ‘,’ and stay there.
On the other hand, if $q$ is not the halting state, $C_T$’s behavior is shown in Figure 48. To reduce clutter, not all transition are shown in Figure 48. Transitions from state $q \neq q_h$ in $C_T$. Inputs and associated probabilities are shown on each transition separated by “:” symbol. Missing transitions are as follows: from every state (except accept) there is a $\tau$ transitions to reject with probability 1; transition on an input symbol (not $\tau$) not shown from a state (e.g., on $a$ from $q$) are assumed to be self-loops with probability 1. Transition structure from states $s_1, s_2, s_3$ is not shown and is “similar” to that from $r_1, r_2, r_3$. $u$ is the control state of $T$ after taking the transition enabled from control state $q$. Some of the transitions are labeled by regular expressions; they can be implemented by having appropriate intermediate states and transitions.

Figure 48. HPA for Undecidability Proof.
We now explain the rationale behind Figure 48. Intuitively, on reading ‘,’ $C_T$ will probabilistically choose to do one of the following.

**Accept** Accept the input without reading any further, by transitioning to state $\text{accept}$. This happens with probability $\frac{63}{256}$.

**Reject** Reject the input without reading any further, by transitioning to state $\text{reject}$. This also happens with probability $\frac{63}{256}$.

**Skip** “Ignore” the configuration $(q, a^i, b^j)$ and potentially check the correctness of counter values in the future. This is depicted by the computation(s) going through state $q'$. In this computation, as it reads the counters $a$ and $b$, there is a chance that $C_T$ will abandon reading more of the input, and transition to $\text{accept}$ or $\text{reject}$ states. Let $u$ be the control state of $T$ after taking the transition enabled at $q$. If the input is neither accepted nor rejected while reading $(q, a^i, b^j)$ in this mode, then $C_T$ moves to state $u$. The computation proceeds again by making a choice between Accept/Reject/Skip/Check phases from $u$.

**Check** Check that the $a$-counter (or $b$-counter, choice made probabilistically) of the next configuration is consistent with the transition taken from $(q, a^i, b^j)$, the current configuration being processed. Checking the value of the $a$-counters is accomplished by the paths going through the $r_i$ states, while checking the $b$-counters is depicted by the computations through the $s_i$ states, which is not shown completely in Figure 48. We will explain these computations in greater detail below.
Before explaining how the “Check” phase of $C_T$ proceeds, it is important to make a couple of observation about the transitions shown in Figure 48. First, suppose $C_T$ chooses one among Accept, Reject or Skip options. Then the probability that $C_T$ is in state accept after reading the configuration $(q, a^i, b^j)$ is the same as the probability that it is in state reject. Thus, neither acceptance nor rejection probabilities are biased, unless $C_T$ chooses the Check phase. Second, it is useful to clarify the structure of the states shown in Figure 48. The states $q'$, $r_i$, and $s_i$ (and intermediate states needed to implement transition labeled by regular expressions in Figure 48) are different for each state $q \in Q$. The states $q, u \in Q$ and $q'$ are level 0 states; $r_i$ and $s_i$ (and other missing intermediate states) are level 1 states; and accept and reject are level 2 states. Thus, $C_T$ is a 2-HPA.

Now let us explain how the “Check” phase proceeds. First the ideas behind checking the correctness of $b$-counter values are the same as those to check the $a$-counter values. Thus, we will only explain how $a$-counter values are checked (the transition structure involving the states $r_i$). Next, recall that, based on the state $q$, $T$ might choose either increment, decrement, or leave unchanged the $a$-counter. If the configuration immediately after $(q, a^i, b^j)$ in input $\rho$ is $(u, a^k, b^l)$, this requires checking $i + 1 = k$ (increment), $i = k + 1$ (decrement), or $i = k$ (unchanged). Hence, the challenge is primarily to check the equality of two numbers probabilistically; if we can check the “unchanged” case, then checking the increment or decrement cases only involves checking the unchanged case when 1 is added to either $i$ or $k$. So in what follows we will only explain how $C_T$ checks the unchanged case.
Let us fix the configurations \((q, a^i, b^j)\) and \((u, a^k, b^\ell)\) and let us assume that the goal in the Check phase is to check that \(i = k\). Checking if \(i = k\) requires “counting” the number of \(a\)s. But since \(C_T\) has only finite memory this cannot be carried out explicitly. Instead \(C_T\) will use coin tosses to *implicitly* count these numbers in terms of the probability of reaching various states; this general principle has been exploited previously (42; 18). Intuitively, \(C_T\) probabilistically chooses to do one of the following.

**Count i and k** If \(C_T\) goes through state \(r_1\) then it implicitly counts both \(i\) and \(k\). In state \(r_1\), on every \(a\) symbol read, it probabilistically chooses to either stay in \(r_1\) (with probability \(\frac{1}{2}\)), or go to accept or reject (with probability \(\frac{1}{4}\) each). After all the \(a\)s are read, it skips all the symbols until the start of the \(a\)s in the next configuration, and moves to state \(r_4\) with probability 1. In state \(r_4\), with each \(a\) read, it chooses to either stay in \(r_4\), go to accept or go to reject with the same probabilities as from state \(r_1\). Once all the \(a\)s have been read, it goes to state accept with probability 1 when it reads ‘,’.

**Count i** If \(C_T\) goes through state \(r_2\) then implicitly counts only \(i\). With every \(a\) read, \(C_T\) stays in \(r_2\) with probability \(\frac{1}{4}\), moves to accept or reject with probability \(\frac{3}{8}\) each. When all the \(a\)s in the first configuration are read, \(C_T\) moves to reject with probability 1 when ‘,’ is read, skipping the rest of the input.

**Count k** \(C_T\) implicitly counts \(k\) when it moves to state \(r_3\). It skips the encoding of the \(a\) and b-counters in the first configuration, and the control state of the next configuration, and goes to state \(r_5\) with probability 1, when the beginning of the encoding of the \(a\)-counter in the next configuration is reached. At this point, it counts \(k\) in \(r_5\) in the same way as
i was counted in $r_2$; with every a symbol read, it either stays in $r_5$ with probability $\frac{1}{8}$, or moves to accept or reject with probability $\frac{3}{8}$ each. After all the a$s have been read, it moves to reject with probability 1 when \textquoteleft;,' is encountered.

To understand how i and k are implicitly being counted through these probabilistic transitions, we need to analyze the probabilities of acceptance and rejection. This analysis will also help establish the correctness of our construction.

Let $p_{\text{accept}}$ and $p_{\text{reject}}$ be the probabilities of reaching accept and reject states, respectively, when choosing each of the 3 options — Count i and k (state $r_1$), Count i (state $r_2$), and Count k (state $r_3$). These values are shown in Table II.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
 & $p_{\text{accept}}$ & $p_{\text{reject}}$ & sum \\
\hline
Count i and k (state $r_1$) & $\frac{1}{16} (1 + 2^{-i-k})$ & $\frac{1}{16} (1 - 2^{-i-k})$ & $\frac{1}{8}$ \\
\hline
Count i (state $r_2$) & $\frac{1}{32} (1 - 2^{-2i})$ & $\frac{1}{32} (1 + 2^{-2i})$ & $\frac{1}{16}$ \\
\hline
Count k (state $r_3$) & $\frac{1}{32} (1 - 2^{-2k})$ & $\frac{1}{32} (1 + 2^{-2k})$ & $\frac{1}{16}$ \\
\hline
\end{tabular}
\caption{Probability Analysis for Undecidability Proof.}
\end{table}

These values are derived as follows. Observe that $C_T$ is going to be in the states $r_1$ and $r_4$, respectively, when receiving input a symbols of the first counter in the current and next configurations, after taking the $r_1$ branch from state q. The probabilities of taking the $r_1$ branch and then be in the states $r_1$, accept and reject, respectively, at the end of the first counter of the current configuration, are easily seen to be $\frac{1}{8} \cdot 2^{-i}$, $\frac{1}{16} (1 - 2^{-i})$ and $\frac{1}{16} (1 - 2^{-i})$. The probabilities of being in the states $r_4$, accept and reject, respectively, after passing through both $r_1$ and $r_4$
, at the end of the first counter of the next configuration, are easily seen to be \( \frac{1}{8} \cdot 2^{-(l+k)} \), \( \frac{1}{16} 2^{-i} (1 - 2^{-k}) \) and \( \frac{1}{16} 2^{-i} (1 - 2^{-k}) \). Putting the above observations together and observing that the remaining probability of being in \( r_4 \) is transferred to the accept state after ‘,’ is read the first counter of the next configuration, we get the probabilities shown in the first row of Table II.

Observe that \( C_T \) is also in state \( r_2 \) with non-zero probability when the symbols of the first counter of the current configuration are received. The probabilities of taking the \( r_2 \) branch and then be in state \( r_2 \), accept and reject, respectively, at the end of the first counter of the current configuration are \( \frac{1}{16} 4^{-i} \), \( \frac{1}{32} (1 - 4^{-i}) \) and \( \frac{1}{32} (1 - 4^{-i}) \). Using these probability values and observing that the probability of being in \( r_2 \) is transferred to the reject state immediately after ‘,’ is read after the first counter of the current configuration, we get the probability values shown in the second row of Table II. By symmetric argument and by observing that, when in state \( r_3 \), \( C_T \) is receiving the symbols of the first counter of the next configuration, we get the probability values given in the third row of Table II.

From this, we see that

\[
p_{\text{reject}} - p_{\text{accept}} = \frac{1}{16} (2^{-2i} + 2^{-2k} - 2 \cdot 2^{-(l+k)}) = \frac{1}{16} (2^{-i} - 2^{-k})^2 \geq 0.
\]

If \( i = k \), then \( p_{\text{reject}} - p_{\text{accept}} = 0 \). Now consider the case when \( i \neq k \). Assume with out loss of generality that \( i > k \). In this case,

\[
p_{\text{reject}} - p_{\text{accept}} \geq \frac{1}{16} (2^{-k} - 2^{-k-1})^2 = \frac{1}{16} 2^{-2(k+1)} = \frac{1}{64} \cdot 2^{-2k}.
\]
Symmetrically, when $k > i$, we have $p_{\text{reject}} - p_{\text{accept}} \geq \frac{1}{51} \cdot 2^{-2i}$. As observed before, the probabilities of going to the accept and reject states from $q$, without going through $r_1, r_2, r_3$ or $s_1, s_2, s_3$, are equal. Now the probability that $C_T$ is at a level 0 state (i.e., such as $q, q'$) after the current configuration is $\frac{1}{128} \cdot (\frac{1}{128})^{(i+j)}$ where $i, j$ are the values of the first and second counters in the current configuration, which is $= \frac{1}{128} \cdot 2^{-(7i+7j)}$. This value is less than $\frac{1}{51} \cdot 2^{-2k}$ when $i > k$, and is less than $\frac{1}{51} \cdot 2^{-2i}$ when $k > i$. (The same reasoning is used to establish this property when we consider the second counter values using the values $j, \ell$). This shows that $p_{\text{reject}} - p_{\text{accept}}$ is greater the probability that $C_T$ is at a level 0 state plus the probabilities that $C_T$ is in any of the intermediate level 1 states which go to the accepting state if the following configuration has a halt state in an illegal computation. From this, we see that even if we get an illegal computation that ends with a halting state, that computation will be accepted with probability $< \frac{1}{2}$. On the other hand a legal halting computation will be accepted with probability $> \frac{1}{2}$. $\square$
CHAPTER 11

CONCLUSION

In this report we systematically introduced using 1-HPA for model checking failure-prone open concurrent systems with respect to the non-extremal threshold language emptiness decision problem and the universal robustness problem. We proved such problems are undecidable for 2-HPA but decidable for 1-HPA. We developed two \textbf{EXPTIME} algorithms - backward algorithm and forward algorithm, and a tool - HiPAM, to model check 1-HPA, which is obtained combining open probabilistic systems and properties expressed as deterministic automata. Properties are defined on system executions as either safety properties or non-safety properties. For some domain applications such as web applications, we designed and implemented complete model abstraction techniques. When data models can be extracted from system models, such as in the domain of business process management, we introduced a methodology on how to obtain failure specification more precisely. Other contributions include verifying acyclic 1-HPA and proving the emptiness problem for acyclic HPA in general is \textbf{NP}-complete.

Future work can be done on improving the HPA modeling technique, by providing support for internal inputs and shared variables. To solve the state explosion problem in concurrent systems, symmetry reduction and symbolic model checking are being investigated. What’s more, properties expressed by temporal logics are to be supported in the future.


   http://www.bpmn.org/.

39. van Glabbeek, R. J.: CONCUR’99 Concurrency Theory: 10th International Conference  
   Eindhoven, The Netherlands, August 24—27, 1999 Proceedings, chapter Petri  
   Nets, Configuration Structures and Higher Dimensional Automata, pages 21–27.  


41. Sha, W., Zhang, L., Ben, Y., and Lu, Y.: Compliance verification using a joint model of  
   open workflow net and global calculus. In Proceedings of the IADIS International  

42. Condon, A. and Lipton, R. J.: On the complexity of space bounded interactive proofs  
   (extended abstract). In Symp. on Foundations of Computer Science, pages 462–  
   467, 1989.
VITA

Yue Ben was born in Heilongjiang, China. After completing high-school study in Daqing No.1 Middle School in summer 2006, she was admitted to School of Software, Tsinghua University located in Beijing, China. There she obtained a Bachelor’s degree on Computer Software in July 2010, and a Master’s degree on Software Engineering in June 2012. In August 2012, she began her study in the PhD program in the Department of Computer Science at University of Illinois at Chicago, United States.
This Agreement between Yue Ben ("You") and Springer ("Springer") consists of your license details and the terms and conditions provided by Springer and Copyright Clearance Center.

**License Number** 3940420327642  
**License date** Sep 01, 2016  
**Licensed Content Publisher** Springer  
**Licensed Content Publication** Springer eBook  
**Licensed Content Title** Decidable and Expressive Classes of Probabilistic Automata  
**Licensed Content Author** Rohit Chadha  
**Licensed Content Date** Jan 1, 2015  
**Type of Use** Thesis/Dissertation  
**Portion** Full text  
**Number of copies** 1  
**Author of this Springer article** Yes and you are a contributor of the new work  

**Order reference number**  
**Title of your thesis / dissertation** Model Checking Open Probabilistic Systems Using Hierarchical Probabilistic Automata  
**Expected completion date** Sep 2016  
**Estimated size(pages)** 120  
**Requestor Location** Yue Ben  
130 Descanso Dr  
Unit 168  
SAN JOSE, CA 95134  
United States  
Attn: Yue Ben  

**Billing Type** Invoice  
**Billing Address** Yue Ben  
130 Descanso Dr  
Unit 168  
SAN JOSE, CA 95134  
United States  
Attn: Yue Ben  

**Total** 0.00 USD

**Terms and Conditions**

Introduction  
The publisher for this copyrighted material is Springer. By clicking "accept" in connection with completing this licensing transaction, you agree that the following terms and conditions apply to this transaction (along with the Billing and Payment terms and conditions established by Copyright Clearance Center, Inc. ("CCC"), at the time that you opened your Rightslink account and that are available at any time at [http://myaccount.copyright.com](http://myaccount.copyright.com)).
Limited License
With reference to your request to reuse material on which Springer controls the copyright, permission is granted for the use indicated in your enquiry under the following conditions:
- Licenses are for one-time use only with a maximum distribution equal to the number stated in your request.
- Springer material represents original material which does not carry references to other sources. If the material in question appears with a credit to another source, this permission is not valid and authorization has to be obtained from the original copyright holder.
- This permission
  • is non-exclusive
  • is only valid if no personal rights, trademarks, or competitive products are infringed.
  • explicitly excludes the right for derivatives.
- Springer does not supply original artwork or content.
- According to the format which you have selected, the following conditions apply accordingly:
  - Print and Electronic: This License include use in electronic form provided it is password protected, on intranet, or CD-Rom/DVD or E-book/E-journal. It may not be republished in electronic open access.
  - Print: This License excludes use in electronic form.
  - Electronic: This License only pertains to use in electronic form provided it is password protected, on intranet, or CD-Rom/DVD or E-book/E-journal. It may not be republished in electronic open access.
For any electronic use not mentioned, please contact Springer at permissions.springer@spiglobal.com.
- Although Springer controls the copyright to the material and is entitled to negotiate on rights, this license is only valid subject to courtesy information to the author (address is given in the article/chapter).
- If you are an STM Signatory or your work will be published by an STM Signatory and you are requesting to reuse figures/tables/illustrations or single text extracts, permission is granted according to STM Permissions Guidelines: http://www.stm-assoc.org/permissions-guidelines/
For any electronic use not mentioned in the Guidelines, please contact Springer at permissions.springer@spiglobal.com. If you request to reuse more content than stipulated in the STM Permissions Guidelines, you will be charged a permission fee for the excess content.
Permission is valid upon payment of the fee as indicated in the licensing process. If permission is granted free of charge on this occasion, that does not prejudice any rights we might have to charge for reproduction of our copyrighted material in the future.
- If your request is for reuse in a Thesis, permission is granted free of charge under the following conditions:
  This license is valid for one-time use only for the purpose of defending your thesis and with a maximum of 100 extra copies in paper. If the thesis is going to be published, permission needs to be reobtained.
- includes use in an electronic form, provided it is an author-created version of the thesis on his/her own website and his/her university’s repository, including UMI (according to the definition on the Sherpa website: http://www.sherpa.ac.uk/romeo/);
- is subject to courtesy information to the co-author or corresponding author.
Geographic Rights: Scope
Licenses may be exercised anywhere in the world.
Altering/Modifying Material: Not Permitted
Figures, tables, and illustrations may be altered minimally to serve your work. You may not alter or modify text in any manner. Abbreviations, additions, deletions and/or any other alterations shall be made only with prior written authorization of the author(s).
Reservation of Rights
Springer reserves all rights not specifically granted in the combination of (i) the license details provided by you and accepted in the course of this licensing transaction and (ii) these terms and conditions and (iii) CCC's Billing and Payment terms and conditions.

License Contingent on Payment
While you may exercise the rights licensed immediately upon issuance of the license at the end of the licensing process for the transaction, provided that you have disclosed complete and accurate details of your proposed use, no license is finally effective unless and until full payment is received from you (either by Springer or by CCC) as provided in CCC's Billing and Payment terms and conditions. If full payment is not received by the date due, then any license preliminarily granted shall be deemed automatically revoked and shall be void as if never granted. Further, in the event that you breach any of these terms and conditions or any of CCC's Billing and Payment terms and conditions, the license is automatically revoked and shall be void as if never granted. Use of materials as described in a revoked license, as well as any use of the materials beyond the scope of an unrevoked license, may constitute copyright infringement and Springer reserves the right to take any and all action to protect its copyright in the materials.

Copyright Notice: Disclaimer
You must include the following copyright and permission notice in connection with any reproduction of the licensed material:
"Springer book/journal title, chapter/article title, volume, year of publication, page, name(s) of author(s), (original copyright notice as given in the publication in which the material was originally published) "With permission of Springer"
In case of use of a graph or illustration, the caption of the graph or illustration must be included, as it is indicated in the original publication.

Warranties: None
Springer makes no representations or warranties with respect to the licensed material and adopts on its own behalf the limitations and disclaimers established by CCC on its behalf in its Billing and Payment terms and conditions for this licensing transaction.

Indemnity
You hereby indemnify and agree to hold harmless Springer and CCC, and their respective officers, directors, employees and agents, from and against any and all claims arising out of your use of the licensed material other than as specifically authorized pursuant to this license.

No Transfer of License
This license is personal to you and may not be sublicensed, assigned, or transferred by you without Springer's written permission.

No Amendment Except in Writing
This license may not be amended except in a writing signed by both parties (or, in the case of Springer, by CCC on Springer's behalf).

Objection to Contrary Terms
Springer hereby objects to any terms contained in any purchase order, acknowledgment, check endorsement or other writing prepared by you, which terms are inconsistent with these terms and conditions or CCC's Billing and Payment terms and conditions. These terms and conditions, together with CCC's Billing and Payment terms and conditions (which are incorporated herein), comprise the entire agreement between you and Springer (and CCC) concerning this licensing transaction. In the event of any conflict between your obligations established by these terms and conditions and those established by CCC's Billing and Payment terms and conditions, these terms and conditions shall control.

Jurisdiction
All disputes that may arise in connection with this present License, or the breach thereof, shall be settled exclusively by arbitration, to be held in the Federal Republic of Germany, in accordance with German law.

Other conditions:
Questions? custumercare@copyright.com or +1-855-239-3415 (toll free in the US) or +1-978-646-2777.
This Agreement between Yue Ben ("You") and Springer ("Springer") consists of your license details and the terms and conditions provided by Springer and Copyright Clearance Center.

License Number                   3940420497890
License date                     Sep 01, 2016
Licensed Content Publisher       Springer
Licensed Content Publication     Springer eBook
Licensed Content Title           Model Checking Failure-Prone Open Systems Using Probabilistic Automata
Licensed Content Author          Yue Ben
Licensed Content Date            Jan 1, 2015
Type of Use                     Thesis/Dissertation
Portion                         Full text
Number of copies                1
Author of this Springer article  Yes and you are a contributor of the new work
Order reference number
Title of your thesis / dissertation           Model Checking Open Probabilistic Systems Using Hierarchical Probabilistic Automata
Expected completion date         Sep 2016
Estimated size(pages)            120
Requestor Location              Yue Ben
                                130 Descanso Dr
                                Unit 168
                                SAN JOSE, CA 95134
                                United States
                                Attn: Yue Ben
Billing Type                    Invoice
Billing Address                 Yue Ben
                                130 Descanso Dr
                                Unit 168
                                SAN JOSE, CA 95134
                                United States
                                Attn: Yue Ben
Total                           0.00 USD

Terms and Conditions

Introduction
The publisher for this copyrighted material is Springer. By clicking "accept" in connection with completing this licensing transaction, you agree that the following terms and conditions apply to this transaction (along with the Billing and Payment terms and conditions...
established by Copyright Clearance Center, Inc. ("CCC"), at the time that you opened your Rightslink account and that are available at any time at http://myaccount.copyright.com).

Limited License

With reference to your request to reuse material on which Springer controls the copyright, permission is granted for the use indicated in your enquiry under the following conditions:
- Licenses are for one-time use only with a maximum distribution equal to the number stated in your request.
- Springer material represents original material which does not carry references to other sources. If the material in question appears with a credit to another source, this permission is not valid and authorization has to be obtained from the original copyright holder.
- This permission
  - is non-exclusive
  - is only valid if no personal rights, trademarks, or competitive products are infringed.
  - explicitly excludes the right for derivatives.
  - Springer does not supply original artwork or content.
- According to the format which you have selected, the following conditions apply accordingly:
  - **Print and Electronic**: This License include use in electronic form provided it is password protected, on intranet, or CD-Rom/DVD or E-book/E-journal. It may not be republished in electronic open access.
  - **Print**: This License excludes use in electronic form.
  - **Electronic**: This License only pertains to use in electronic form provided it is password protected, on intranet, or CD-Rom/DVD or E-book/E-journal. It may not be republished in electronic open access.

For any electronic use not mentioned, please contact Springer at permissions.springer@spi-global.com.
- Although Springer controls the copyright to the material and is entitled to negotiate on rights, this license is only valid subject to courtesy information to the author (address is given in the article/chapter).
- If you are an STM Signatory oryour work will be published by an STM Signatory and you are requesting to reuse figures/tables/illustrations or single text extracts, permission is granted according to STM Permissions Guidelines: http://www.stm-assoc.org/permissions-guidelines/

For any electronic use not mentioned in the Guidelines, please contact Springer at permissions.springer@spi-global.com. If you request to reuse more content than stipulated in the STM Permissions Guidelines, you will be charged a permission fee for the excess content.

Permission is valid upon payment of the fee as indicated in the licensing process. If permission is granted free of charge on this occasion, that does not prejudice any rights we might have to charge for reproduction of our copyrighted material in the future.

- If your request is for reuse in a Thesis, permission is granted free of charge under the following conditions:
  - This license is valid for one-time use only for the purpose of defending your thesis and with a maximum of 100 extra copies in paper. If the thesis is going to be published, permission needs to be reobtained.
  - includes use in an electronic form, provided it is an author-created version of the thesis on his/her own website and his/her university’s repository, including UMI (according to the definition on the Sherpa website: http://www.sherpa.ac.uk/romeo/);
  - is subject to courtesy information to the co-author or corresponding author.

Geographic Rights: Scope

Licenses may be exercised anywhere in the world.

Altering/Modifying Material: Not Permitted

Figures, tables, and illustrations may be altered minimally to serve your work. You may not
alter or modify text in any manner. Abbreviations, additions, deletions and/or any other alterations shall be made only with prior written authorization of the author(s).

Reservation of Rights
Springer reserves all rights not specifically granted in the combination of (i) the license details provided by you and accepted in the course of this licensing transaction and (ii) these terms and conditions and (iii) CCC's Billing and Payment terms and conditions.

License Contingent on Payment
While you may exercise the rights licensed immediately upon issuance of the license at the end of the licensing process for the transaction, provided that you have disclosed complete and accurate details of your proposed use, no license is finally effective unless and until full payment is received from you (either by Springer or by CCC) as provided in CCC's Billing and Payment terms and conditions. If full payment is not received by the date due, then any license preliminarily granted shall be deemed automatically revoked and shall be void as if never granted. Further, in the event that you breach any of these terms and conditions or any of CCC's Billing and Payment terms and conditions, the license is automatically revoked and shall be void as if never granted. Use of materials as described in a revoked license, as well as any use of the materials beyond the scope of an unrevoked license, may constitute copyright infringement and Springer reserves the right to take any and all action to protect its copyright in the materials.

Copyright Notice: Disclaimer
You must include the following copyright and permission notice in connection with any reproduction of the licensed material:
"Springer book/journal title, chapter/article title, volume, year of publication, page, name(s) of author(s), (original copyright notice as given in the publication in which the material was originally published) "With permission of Springer"
In case of use of a graph or illustration, the caption of the graph or illustration must be included, as it is indicated in the original publication.

Warranties: None
Springer makes no representations or warranties with respect to the licensed material and adopts on its own behalf the limitations and disclaimers established by CCC on its behalf in its Billing and Payment terms and conditions for this licensing transaction.

Indemnity
You hereby indemnify and agree to hold harmless Springer and CCC, and their respective officers, directors, employees and agents, from and against any and all claims arising out of your use of the licensed material other than as specifically authorized pursuant to this license.

No Transfer of License
This license is personal to you and may not be sublicensed, assigned, or transferred by you without Springer's written permission.

No Amendment Except in Writing
This license may not be amended except in a writing signed by both parties (or, in the case of Springer, by CCC on Springer's behalf).

Objection to Contrary Terms
Springer hereby objects to any terms contained in any purchase order, acknowledgment, check endorsement or other writing prepared by you, which terms are inconsistent with these terms and conditions or CCC's Billing and Payment terms and conditions. These terms and conditions, together with CCC's Billing and Payment terms and conditions (which are incorporated herein), comprise the entire agreement between you and Springer (and CCC) concerning this licensing transaction. In the event of any conflict between your obligations established by these terms and conditions and those established by CCC's Billing and Payment terms and conditions, these terms and conditions shall control.

Jurisdiction
All disputes that may arise in connection with this present License, or the breach thereof,
shall be settled exclusively by arbitration, to be held in the Federal Republic of Germany, in accordance with German law.

**Other conditions:**
V 12AUG2015

Questions? [customercare@copyright.com](mailto:customercare@copyright.com) or +1-855-239-3415 (toll free in the US) or +1-978-646-2777.