HPA Demo V1.0 Technical Report

Yue (Cindy) Ben*

March 26, 2015

The HPA Demo tool is developed to present our research on Model Checking Failure Prone Open Systems Using Probabilistic Automata. Systems are modeled as Hierarchical Probabilistic Automata (HPA), and verified against our Forward and Backward algorithms.

Project Website: https://github.com/benyue/HPA/

Please visit the project website for any download, such as the executable JAR file, source code, reference paper, etc..

1 Installation

There are several options to install the HPA Demo tool.

Download JAR Download "HPA.jar" file, which is the single executable file of the HPA demo tool. Then double-click the jar file to execute it.

Download jnlp Download "HPA.jnlp" file, which will connect to Internet and download the latest HPA.

Online Access To run the app from web browser without downloading it, please click Launch on the following webpage (IE browser recommended):
http://www.cs.uic.edu/ yben/tools/HPA/dist/launch.html

Compile Source Code In case the provided Jar file does not work on your computer, you may want to compile the Java source code yourself. Please refer to the project website mentioned above and especially the Source Code folder.

Make sure you have Java Runtime Environment (JRE) installed, with a recommended version of 1.7.0 or higher. Hint: your operating system will very possibly block the Jar program from running, since it requires access to your local files to load and save HPA files. Please adjust your Java security settings accordingly[1]. You may have to add "http://www.cs.uic.edu" in the exception website list.

[0]University of Illinois at Chicago. Email: yben2@uic.edu

[1]Reference: http://java.com/en/download/help/jcp_security.xml

2 Quick Start

There are mainly three parts in the tool: Part 1 interacts with user to create a 1-HPA; Part 2 is for verification and solving robustness problem; and Part 3 is a log window.

Part 1 is NOT necessary if you cares only about verification. In that case, go to Part 2 directly!

If you wanna try generating HPA, please refer to Section 4.1.4 for more details especially on defining the failure.

To get started, simply follow steps presented in the figure below.

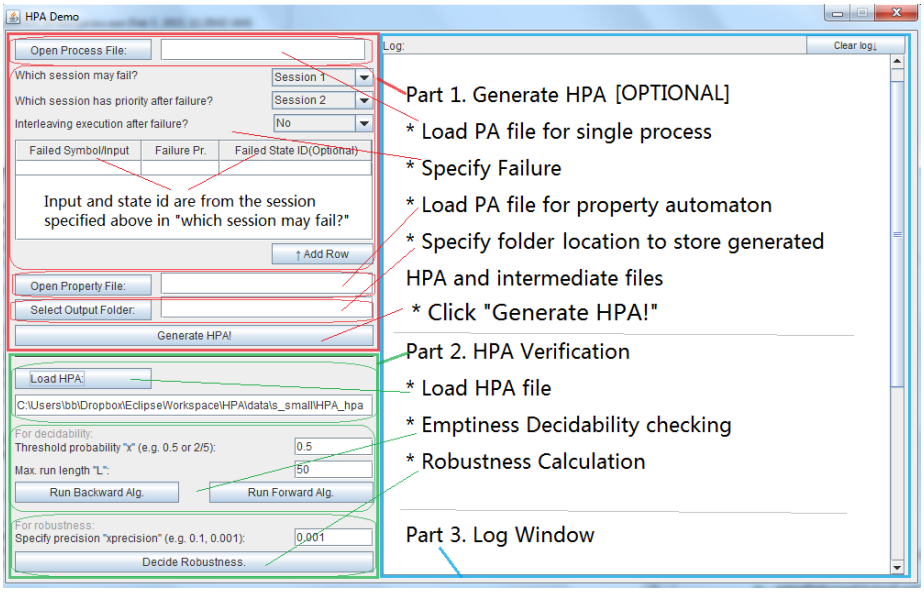

Figure 1: HPA Demo User Interface

Examples can be found under the folder ”examples” on the project website. Or you can write your own HPA files for Part 2 (or PA files for Part 1) following corresponding file formates introduced later in this manual.

3 A Complete HPA Verification Example

Here we give a complete example including generating HPA and all its intermediate files, and verification over the generated HPA as well. The example is named as ”s_small” and can be found in the ”examples” given on the project website.

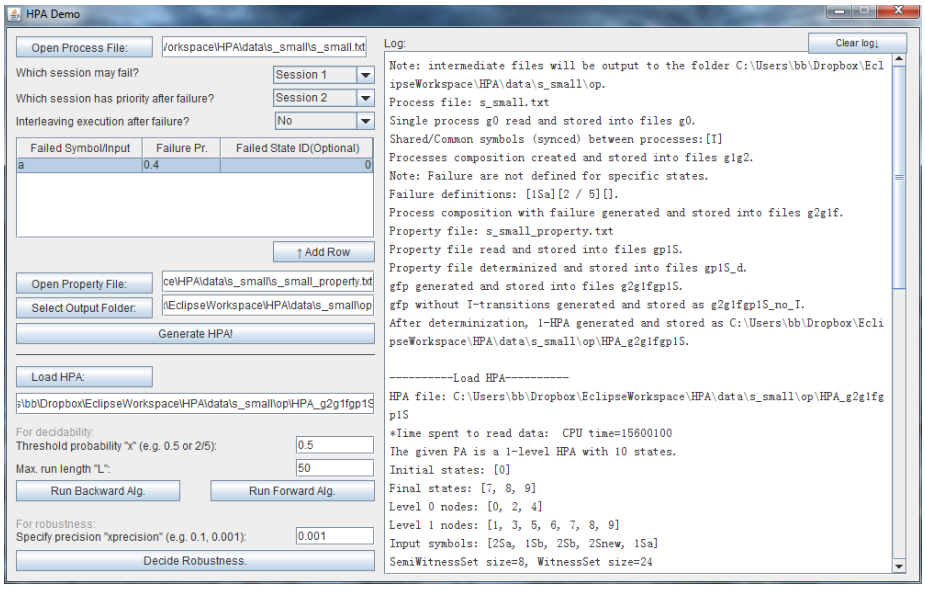

Figure 2: HPA Example: User Interface

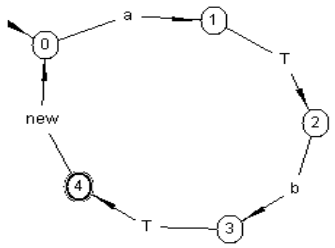

Figure 3: HPA Example: User Input Single Session g_0. The state with a right arrow is the initial state. Double circle denotes final states. On edges are the input symbols. Probabilities are not presented in the graph.

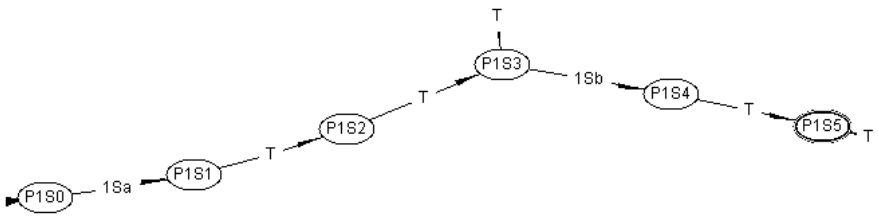

Figure 4: HPA Example: User Input property g_p. The property automaton is loaded then determinized by the program.

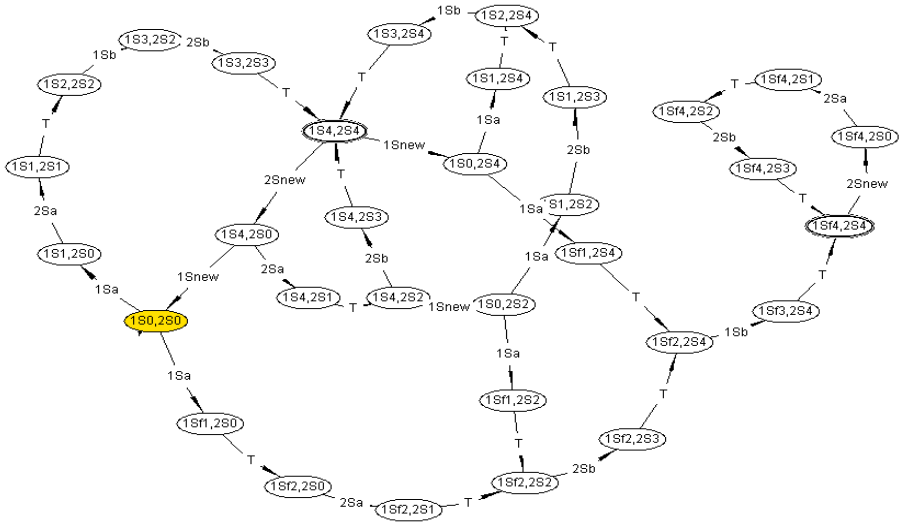

Figure 5: HPA Example: g_{21f}. As user specified in Figure 2, g_1 failed, g_2 had priority after failure, and no interleaving behavior after failure.

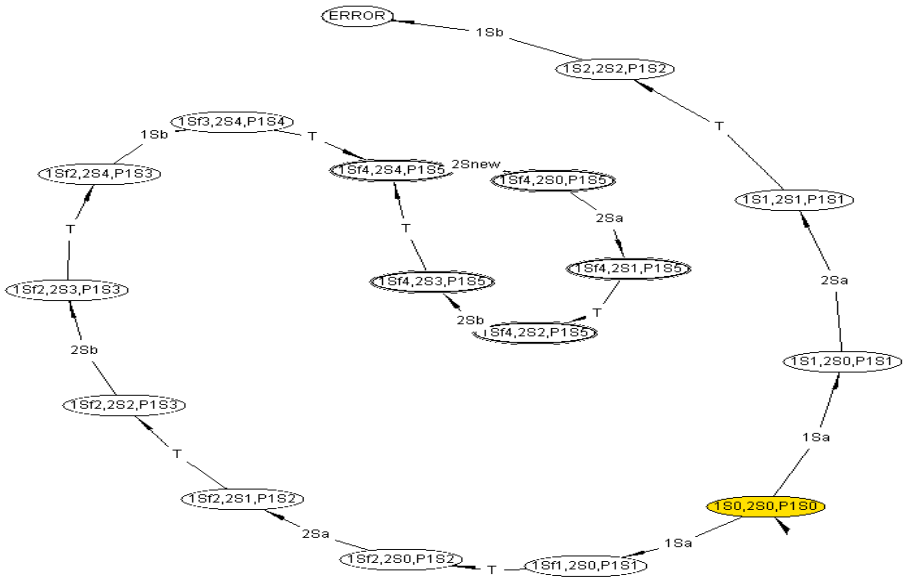

Figure 6: HPA Example: g_{21fp}, the composition of g_{21f} and $g_{p-determinized}$, synchronized on shared variable T.

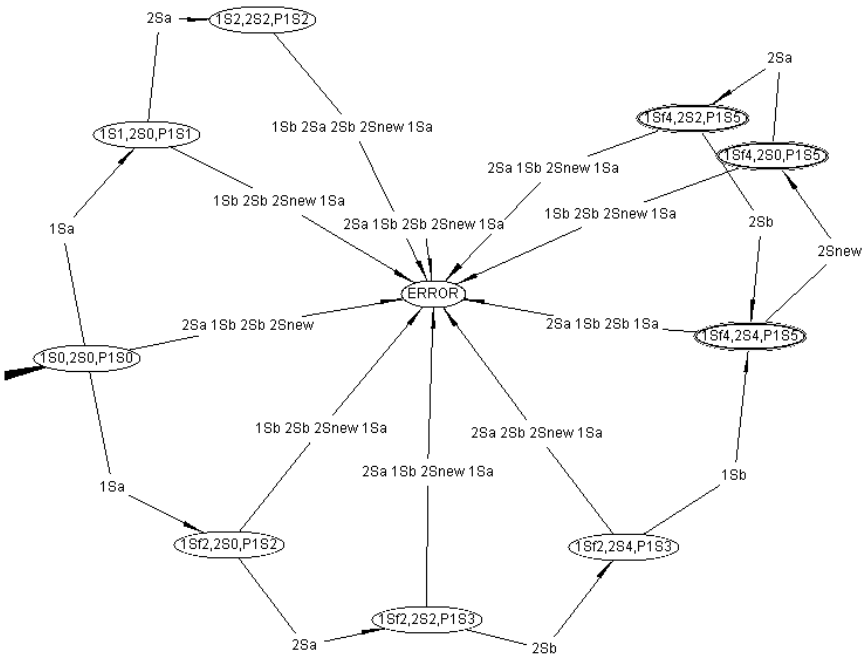

Figure 7: HPA Example: hpa

After generating the 1-HPA, user can run different verification algorithms. The log below is for the 1-HPA generated above. For non-empty results the tool will also give an example run.

```
----------ROBUSTNESS ALGORITHM----------
Note:L_UpperBound = 85899345920.
Minimum empty is 2 / 5 at L=3, thus robustness value is 3 / 5.
----------DECIDABILITY ALGORITHM: BACKWARD----------
L(A) is empty with threshold x=4001 / 10000 within L=3.
[Backward Convergence]
*Time spent on backward algorithm: CPU time=15600100
----------DECIDABILITY ALGORITHM: FORWARD----------
L(A) is empty with threshold x=4001 / 10000 at L=2.
[Forward Convergence]
*Time spent on forward algorithm: CPU time=15600100
----------DECIDABILITY ALGORITHM: FORWARD----------
L(A) is non-empty with threshold x=39999 / 100000 at L=1
Accepted sequence and runs: [2, 3]<-[1Sa]<-[0]
*Time spent on forward algorithm: CPU time=0
----------DECIDABILITY ALGORITHM: BACKWARD----------
L(A) is non-empty with threshold x=39999 / 100000 at L=2.
Accepted sequence and runs: [0]->[1Sa]->[2, 3]
*Time spent on backward algorithm: CPU time=0
```

4 Manual

4.1 Part 1. Abstracting 1-HPA Models

Part 1 of the HPA Demo Tool does the following: it generates 1-HPA from Single Process, Failure Specification, and Property.

To generate a 1-HPA, user loads one PA file as g_0, another PA file as g_p, and specify failure definitions. Then simply click "Generate HPA!", the program will generate a HPA. The way the program creates multi sessions is that it creates 2 different instances of g_0 with only one thing common - the T symbol.

In detail, user loads one PA file as g_0, specify failure definitions, and load another PA file as g_p. Then simply click "Generate HPA!", the program will generate a HPA.

The way the program creates multi sessions is that it creates 2 different instances of g_0 with only one thing common - the T symbol. Note that in each session, the set of states absorbing T symbols and the set of states absorbing external inputs have to be disjoint. Any violation will fail our abstraction process.

We then compose each two sessions in the traditional way with partial order reduction, and also introduce failure on one session. Note the single sessions are all PAs with only probability-1 transitions, but the composed session with failure is a 1-HPA, say g_{12f} , which has probabilistic distributions on the states where failure occurred.

4.1.1 PA File Formatting

Our tool allows users to specify PAs and HPAs in the form of adjacency lists with transitions labeled with input symbols and probabilities. Optionally, it also allows definition of propositions over states and transitions. All processes in the HPA Demo tool part 1 (Figure 2) share the same formatting for probabilistic automata (PA). Below is an sample PA file.

```
//Comments are after "//", will be omitted in parsing
3 //First specify total number of states
//each state line starts with the state name (String),
//followed by propositions separated by #
s0 #INITIAL //only one initial state
s1
s2 #FINAL //space is NOT required before #
//Transition format:
//source state name-input->
//end state name1,pr1 #prop1;end state name2,pr2 #prop2;...
s0-a->s1 //one end state, probability undefined, so pr=1
s0-b->s1,0.8 #SYNC;s2,0.2 //a probability distribution
s1-T->s2 #proposition-required-for-triggering
```

4.1.2 PA Attributes

INITIAL state and FINAL state(s) For now it's required there is one and only one INITIAL state, and the initial state has no incoming edge from other states than FINAL

state(s) on "new" symbol.The number of FINAL states are not forced.

Propositions Simple string propositions are defined to states and transitions in the process file. State propositions implies "required-for-sync" in property processes. Transition propositions are classified into two groups of "required-for-triggering" and "satisfied", while all propositions defined in PA files are required for triggering.

T symbol and T-transition The T symbol is a special input on transitions, different from external inputs. It's used temporarily for timing and synchronizing purpose, and will be removed before generating HPA, thus in the final HPA there will be no T symbols nor T-transitions. Each T symbol (i.e. one T-transition) denotes a time unit dealing with an external input, so T-transitions always come after external inputs. For now, only the T-transitions require and force synchronization in composing g_1 and g_2.

Cache is allowed, so it's legal to define multiple external inputs and then add T-transitions in process files, e.g. $1 - a \rightarrow 2 - b \rightarrow 3 - T \rightarrow 4 - T \rightarrow 5 - T \rightarrow 6$.

4.1.3 Composition with Partial Order Reduction - g_{12}

Partial order reduction is automatically applied while composing the two processes g_1 and g_2. Since g_1 and g_2 are identical, g_1 is always given priority in the reduction. We will use the notation "$g_1 > g_2$" to represent this priority. Thus whenever g_1 and g_2 are both active, g_1 will keep executing until reaching T-transitions or final states. Specially, we have "new" symbols defined for each process. Each FINAL state goes to the INITIAL state on a special input "new", and beginning a new session executes nondeterministically. Thus when both servers are at FINAL states, both of them can start new sessions on their own "new", ignoring priority. In the case priority of $g_1 > g_2$ is defined, when g_1 is at FINAL state while g_2 is not, g_1 can start new session on "$g_1$new" when g_2 is not dealing with external inputs; when g_2 is at FINAL state while g_1 is not, g_1 will keep running, and no synchronization required since g_2 is inactive.

4.1.4 Failure Specification

As stated before, we are considering probabilistic failure. For two concurrent processes, only one process may fail, thus the failure is defined over the input and state of one single process only. After failure occurs, no more failure will occur. After both processes complete current sessions at failure, only the remaining process which did not fail will start new sessions. We provide two options for the overall system to recover from failure and complete remaining tasks. After failure and before starting new sessions: either one process completes first then the other; or they each take one step interleavingly and execute one external input.For example, suppose g_2 may fail and the defined after-failure priority is $g_2 > g_1$, then either: 1) g_2 will complete session first, and then g_1 completes current session, and finally g_2, the server which didn't fail, will start new sessions, and this generates g_{2f1}; or: 2) g_2 then g_1 will execute one external input alternatively, then g_2 start new sessions, and this generates g_{2f1_i}. Therefore

before generating HPA user needs to specify which process may fail, the failure definitions, which process has priority after failure, and whether they execute interleaving after failure.

Next, We compose g_{12f} with an incorrectness property, which specifies a safety property and is denoted by a deterministic PA g_p with only probability-1 transitions.

4.1.5 Composition with Property Automaton g_p

Currently we study only safety property, and in the implementation we use incorrectness property instead of correctness property. A property automaton g_p is defined on one session, say g_1, but determinized for all symbols in g_1 and g_2. An ERROR state is defined for determinization, g_p will stay at current state on g_2-only inputs, and go to ERROR state on g_1's undefined inputs.
The new FINAL states after composition are those states where both g_p and g_1 (suppose g_p is defined on g_1) are at FINAL states.
Note property automaton can be defined on either session, but without failure the property automaton won't reach final states, i.e. concurrent processes will not violate the property, thus generated HPA will have no final states and robustness is always 1.

Finally, we remove all T symbols and do a normal determinization. Using the approach, we obtain a 1-level HPA and check its emptiness using both the forward and backward algorithms.

4.2 Part 2. Verification

Part 2 implements the HPA-based verification algorithms. It loads a HPA file, does simply validity checking and leveling (Section 6.2), and execute different verification algorithms (Section 6.1) according to user's command.

4.2.1 HPA File Formatting

The HPA loaded in the HPA Demo tool part 2 (Figure 2) is specified in the HPA formatting. HPA is different from PA in that: HPA requires determinization and reachability of all states from initial state. Their formatting only differ in specifying transition.

```
....

//Transition Format:
//"source state id" input "end state id" probability distribution
//(sid) input (eid1) pr1 (eid2) pr2 ...
2 2Sitem 1 1
2 1Sbid 3 3/5 5 2/5
....
```

HPA are similar to PA except that each state/node in a HPA is assigned a level.

4.2.2 Emptiness

To check the emptiness of the language $\mathsf{L}_{>x}(\mathcal{A})$, user needs to specify the probability threshold x, which is given a default value of 0.5 in the tool UI. Optionally, user can also specify the maximum length of the runs, which may not be used as the theoretical boundary of the maximum length of the runs is already applied in the verification algorithms.

To execute verification, click on the "forward" and/or "backward" buttons.

4.2.3 Robustness

Exact robustness can be obtained in case backward algorithm is applicable to the HPA instance; or else, the "precision" specified by user will be used to decide a robustness range.

5 Future Work

- Support composition of different single processes.
- Support composition of more than two processes.
- Support shared variables among single processes.
- Support complex propositions.
- Support infinite acceptance in verification.

6 Theoretical Background

6.1 Verification Algorithms

For a 1-HPA $\mathcal{A}$, the problem of determining if $\mathsf{L}_{>x}(\mathcal{A})$ is non-empty (i.e., non-emptiness problem) has been shown to be in **EXPTIME**.

We will use the following notations throughout this section: Let $\mathcal{A} = (Q, q_s, \delta, F)$ be 1-HPA over an input alphabet Σ having n states (i.e., $|Q| = n$) and m transitions. The number of transitions of $\mathcal{A}$ is defined to be the cardinality of the set $\{(q, a, q') \mid \delta(q, a, q') > 0\}$. Recall that $\mathcal{Y}$ is the collection of all good witness sets, while $\mathcal{X}$ is the set of all witness sets U such that $U \cap Q_1$ is a good semi witness set. As pointed out earlier, $\mathcal{Y} \subseteq \mathcal{X}$.

Let $x \in [0, 1]$ be a rational threshold of size at most r. It has been shown, in [**?**], that $\mathsf{L}_{>x}(\mathcal{A}) \neq \emptyset$ if and only if there is a finite word u and a good non-empty witness set or semi-witness H, such that $|u| \leq 4rn8^n$ and $\delta_u(q_s, H) > x$. Now, let $L = 4rn8^n$.

6.1.1 Backward Algorithm

For a witness set W, let q_W be the unique state in $W \cap Q_0$. Define *possible a-successors* of W as the set of witness sets $\{V \mid V \in \mathcal{X}, post(W \cap Q_1, a) \subseteq V\}$. A function

$Prob(.,.)$ which maps each pair (W,i), where $W \in \mathcal{X}$ and $i \geq 0$, to a probability value, is defined as follows.

$$Prob(W,i) = max\{\delta_u(q_W,V) \mid u \in \Sigma^*, V \in \mathcal{X}, post(W \cap Q_1, u) \subseteq V, |u| \leq i\}.$$

In the above definition, the maximum of an empty set is defined as 0. It has been shown $\mathsf{L}_{>x}(\mathcal{A}) \neq \emptyset$ iff $Prob(\{q_s\}, L) > x$. the following inductive algorithm is proposed to compute $Prob(.,.)$.

$$\begin{aligned} Prob(W,1) &= \max\{\delta_a(q_W,V) \mid a \in \Sigma, \text{V is a possible a-successor of W}\}; \\ Prob(W,i+1) &= max(\{\{Prob(W,i)\} \\ &\quad \cup \{\delta_a(q_W,q_V)Prob(V,i) + \delta_a(q_W, V \cap Q_1) \\ &\quad \mid a \in \Sigma, \text{V is a possible a-successor of W}\}\}). \end{aligned} \tag{1}$$

6.1.2 Forward Algorithm

In forward algorithm, we first define a quantity $\mathsf{val}(C,x,u)$ (see [**?**]) for a set C of states in HPA $\mathcal{A}$, a threshold $x \in (0,1)$, and a finite word u, as follows.

$$\mathsf{val}(C,x,u) = \begin{cases} \frac{x-\delta_u(q_s,C)}{\delta_u(q_s,Q_0)} & \text{if } \delta_u(q_s,Q_0) \neq 0 \\ +\infty & \text{if } \delta_u(q_s,C) < x, \delta_u(q_s,Q_0) = 0 \\ 0 & \text{if } \delta_u(q_s,C) = x, \delta_u(q_s,Q_0) = 0 \\ -\infty & \text{if } \delta_u(q_s,C) > x, \delta_u(q_s,Q_0) = 0 \end{cases}. \tag{2}$$

This quantity $\mathsf{val}(C,x,u)$ measures the fraction of $\delta_u(q_s,Q_0)$ that still needs to move to C such that the probability of reaching the accepting states at the end exceeds the threshold x. The val values for witness sets play an important role in deciding whether a word κ is accepted by a HPA. It has been shown in [**?**] that κ is accepted with probability $> x$ iff κ can be divided into strings u, κ' such that there is a witness set W such that one of the following conditions is satisfied:

- $\mathsf{val}(W,x,u) < 0$ and κ' is definitely accepted from $W \cap Q_1$
- $0 \leq \mathsf{val}(W \cap Q_1, x, u) < 1$ and κ' is definitely accepted from W.

Now, to check the existence of a string κ satisfying the above property, we define another quantity $Val(W,i)$ for each $W \in \mathcal{X}$ and $i \geq 0$ as follows.

$$Val(W,i) = min\{\mathsf{val}(W \cap Q_1, x, u) \mid |u| \leq i, post(q_s,u) \cap Q_0 \cap W \neq \emptyset\}.$$

In the above definition, $min\{\emptyset\}$ is taken to be ∞.

$\mathsf{L}_{>x}(\mathcal{A}) \neq \emptyset$ iff for some $i \geq 0$, either ($\exists W \in \mathcal{X} : Val(W,i) < 0$) or ($\exists W \in \mathcal{Y} : 0 \leq Val(W,i) < 1$).

Now, for any witness $W \in \mathcal{X}$, $a \in \Sigma$, $q \in Q_0$, let $W_{a,q} = \{r \in Q_1 | \delta_a(r,W) > 0\} \cup \{q\}$. It should be easy to see that $W_{a,q} \in \mathcal{X}$. To calculate the Val values, we

present an incremental algorithm as follows.

$$
\begin{aligned}
Val(W,0) &= \begin{cases} x & if(q_s \in W); \\ \infty & else. \end{cases} \\
\text{For } i>0,\ Val(W,i) &= min(Val(W,i-1), \\
&\quad min\{\frac{Val(W_{a,q},i-1)-\delta_a(q,W\cap Q_1)}{\delta_a(q,W\cap Q_0)} \mid \\
&\quad a\in\Sigma, q\in Q_0, \delta_a(q,W\cap Q_0)>0\}\ )
\end{aligned}
\tag{3}
$$

By induction on i, it can be shown the above algorithm computes $Val(.,.)$ correctly, i.e., the computed values agree with the definition of $Val(.,.)$ given earlier. Clearly, $Val(W,i)$ is monotonically non-increasing with increasing values of i.

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

6.1.3 Robustness

We model the incorrectness of a open system under failures as a 1-level HPA. In this case, we define the robustness of a HPA $\mathcal{A}$ as $1-y$ where y is the least upper bound of the set of values $\{x \mid \mathsf{L}_{>x}(\mathcal{A}) \neq \emptyset\}$. This value is the greatest lower bound of $\{z \mid z$ is the probability of rejection of some input string by $\mathcal{A}\}$. The value of y, and hence the robustness, can be found within some accuracy by using binary search repeatedly employing the 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 or backward 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 2^{-k}. Equivalently, an accuracy of ρ can be guaranteed by iterating the binary search for $\log_2(\frac{1}{\rho})$ number of times.

Sometimes we can compute the exact value of y using backward algorithm. Suppose the backward algorithm reaches a fix point after k iterations, then $y = Prob(\{q_s\}, k)$ where q_s is the initial state of $\mathcal{A}$. In this case, robustness has exact value $(1-y)$.

6.2 Implementation of the Verification Algorithms

In this section, we will present the implementation of our verification algorithms. As part of this, we have developed an algorithm that detects if a given PA $\mathcal{A}$ is a HPA. If so, it will compute the smallest integer k such that $\mathcal{A}$ is a k-HPA and it will also classify the states of $\mathcal{A}$ in to different levels. We will also describe how we automatically abstract

1-HPA models from web applications assuming at most one failure. We will present experimental results demonstrating the effectiveness of our verification algorithms on the abstracted simple web applications.

There are three key steps in our verification process. Firstly, we validate whether a given PA is a HPA and assign levels at the same time. Secondly, we obtain good witness sets for 1-HPA. Finally, we run the forward and backward verification algorithms on good witness sets.

6.2.1 Detect HPAs and Assign Levels

In this section, we describe the algorithm for checking if a given PA is a HPA, and for classifying its states into different levels.

Let $\mathcal{A} = (Q, q_s, \delta, \mathsf{Acc})$ be the given PA over some input alphabet Σ. Let n, m be the number of states and transitions of $\mathcal{A}$ and $s = |\Sigma|$. First, we construct a directed graph $G_{\mathcal{A}} = (Q, E)$ with Q as its set of nodes and $E = \{(q, q') \mid \exists a \in \Sigma \,;\delta(q, a, q') > 0 \}$. Without loss of generality we assume that all nodes in Q are reachable from the initial node q_s. We use the standard definition of *strongly connected components* (SCCs) of $G_{\mathcal{A}}$ and the component graph of $G_{\mathcal{A}}$ (see [**?**]). A *strongly connected component* of G is a maximal subset of Q such that there is a path in $G_{\mathcal{A}}$ between every pair of nodes in it. A component graph of $G_{\mathcal{A}}$ is the directed graph $F_{\mathcal{A}} = (\mathcal{C}, \mathbb{E})$ such that $\mathcal{C}$ is the set of SCCs of $G_{\mathcal{A}}$ and $(C, D) \in \mathbb{E}$ iff $C \neq D$ and $\exists q \in C, q' \in D$ such that $(q, q') \in E$. It is well known that $F_{\mathcal{A}}$ is acyclic, i.e., has no cycles. Note that using standard graph algorithms, $G_{\mathcal{A}}$ and $F_{\mathcal{A}}$ can be constructed in time $O(n + m)$. We call $C \in \mathcal{C}$ as the *initial* SCC if $q_s \in C$, and let C_0 denote the initial SCC. From our assumption, it is easy to see that every node in $F_{\mathcal{A}}$ is reachable from C_0.

A node $C \in \mathcal{C}$ 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 $\delta(q, a, q_2) > 0$. In order for $\mathcal{A}$ to be a HPA, for every $C \in \mathcal{C}$, all nodes in C should be at the same level. From this it is not difficult to see that $\mathcal{A}$ is a HPA iff there are no conflicting nodes in $\mathcal{C}$, i.e., there are no conflicting SCCs in $G_{\mathcal{A}}$. Now assume $\mathcal{A}$ is a HPA, and we will show how to determine the minimum k such that $\mathcal{A}$ is k-HPA. Define the length of a path in $F_{\mathcal{A}}$ to be the number of edges in it, and let ℓ be the length of the longest path in $F_{\mathcal{A}}$. It is easy to see $\mathcal{A}$ is a ℓ-HPA. However ℓ may not be the minimum k such that $\mathcal{A}$ is a k-HPA.

An edge $(C, D) \in \mathbb{E}$ is said to be *conflicting* if $\exists a \in \Sigma$ and $\exists q \in C,\ q_1, q_2 \in D$ such that $q_1 \neq q_2$ and $\delta(q, a, q_1) > 0$ and $\delta(q, a, q_2) > 0$. If (C, D) is a conflicting edge in $\mathbb{E}$, then level of the nodes in D should be strictly higher than those of the nodes in C. For any path in the component graph $F_{\mathcal{A}}$, let the weight of the path be the number of conflicting edges in it. Now, we have the following theorem which is easily proved.

$\mathcal{A}$ is a HPA iff $G_{\mathcal{A}}$ has no conflicting SCCs. Further more, if $G_{\mathcal{A}}$ has no conflicting SCCs and k is the maximum weight of any path in $F_{\mathcal{A}}$, then $\mathcal{A}$ is a k-HPA and k is the minimum such integer.

Now assume that $G_{\mathcal{A}}$ has no conflicting SCCs and k is the maximum weight of any path in $F_{\mathcal{A}}$. Now, we give an algorithm that computes k and partitions the states of $\mathcal{A}$ in to the $k+1$ levels $0, 1, ..., k$. For any node C in $F_{\mathcal{A}}$, let $weight(C)$ be the maximum weight of any path from the initial node C_0 to C. Note that $weight(C_0) = 0$. The

values $weight(C)$, for each $C \in \mathcal{C}$, can be computed using a standard algorithm that runs in linear time of the size of $F_{\mathcal{A}}$ and is of complexity $O(nm)$. Such an algorithm is obtained by considering $F_{\mathcal{A}}$ as an edge weighted graph, where the weight of conflicting edges is taken as 1 and that of non-conflicting edges is taken as 0. For $i = 0, ..., k$, the set of states Q_i at level i is taken to be the set of all states whose weight is i.

6.2.2 Obtain Good Witness Sets

Now assume that $\mathcal{A} = (Q, q_s, \delta, F)$ is a 1-HPA and Q_0, Q_1 are the sets of states at level 0 and level 1, respectively. In section **??**, we defined the set $\mathcal{Y}$ which is the set of good witness sets, and set $\mathcal{X}$ which is the set of all witness sets C such that $C \cap Q_1$ is a good semi-witness set. To compute these sets, we basically start from the set F of final states of $\mathcal{A}$, and traverse backwards using each input symbol and its predecessors, and compute the sets $\mathcal{X}$ and $\mathcal{Y}$ incrementally. Equivalently, we construct a standard non-deterministic automaton $\mathcal{A}^{-1}$ which, intuitively, is a reversal of $\mathcal{A}$, i.e., $\mathcal{A}^{-1}$ has the same set of states as $\mathcal{A}$; there is a transition from q to q' on input a iff $\delta(q', a, q) > 0$; the set of its initial states is F. Our algorithm for computing $\mathcal{X}, \mathcal{Y}$, is similar (with some critical modifications) to that of determinizing $\mathcal{A}^{-1}$ using standard subset-construction. The complexity of this algorithm is $O(wsmn)$ where n is the number of states, m is the number of transitions, $w = |\mathcal{X}|$ and $s = |\Sigma|$.

6.2.3 Run the Verification Algorithms using the Witness Sets

After classifying the states of $\mathcal{A}$ into the two levels, and after computing the sets $\mathcal{X}, \mathcal{Y}$ as described above, we run the forward and backward algorithms described in earlier sections. These algorithms use the threshold value x which is input with $\mathcal{A}$.

In our implementation of the two algorithms, we pre-compute and store some of the values that are repeatedly used in the algorithm; for example, in the forward algorithm, for each $W \in \mathcal{X}$, $q \in Q$ and $a \in \Sigma$, we precompute and store the sets $W_{a,q}$, also the set of of possible a-successors of W, etc. (see Section 6.1).