Revisiting Underapproximate Reachability for Multipushdown Systems

S. Akshay[1], Paul Gastin[2], Krishna S[1], and Sparsa Roychowdhury[1]

[1] IIT Bombay, Mumbai, India
{akshayss,krishnas,sparsa}@cse.iitb.ac.in
[2] ENS Paris-Saclay, France
paul.gastin@lsv.fr

A Details for Experimental Section

A.1 More details regarding the Bluetooth driver example

In this section, we give a few more modeling details about the Bluetooth driver example as well as the error witness found by BHIM. The driver contains four functions except main, namely,

- **Add** function is used to add a new request to perform IO operations on the device. It calls another function **IoIncrement** and stores the return value of this function to a local variable **status**. If the **status** contains the value 0 (which means a successful execution of **IoIncrement**) the thread can execute the IO operations on the device. After the IO completes, it calls the function **IoDecrement**.
- **IoIncrement** increases the variable **PIO**. But, before increasing the value it checks if **SFlg** is true or not. If it is true then it returns -1 else it increases the value of **PIO** using atomic instruction and returns 0.
- **IoDecrement** decreases the variable **PIO**. If the value **PIO** equals to 0 then it sets **SEvnt** to true, as there are no pending IO operations.
- **Stop** sets **SFlag** to true and calls **IoDecrement**. Then, it waits for **SEvnt** to become true, then releases all the resources and marks **stop** flag to true.

An interesting sub-part of the model is depicted in Fig 1 (the actual number of states and transitions is 57 and 96 respectively). This driver is modeled using a MPDA, where the initial state is marked with ($\rightarrow$) and the bad state is marked with double rectangles. The transitions are marked with name as (t_i) and the operations are mentioned as $push^i$, pop^i for push($\downarrow_i$) and pop($\uparrow_i$) respectively, and nothing is mentioned in case of nop. The states represent the valuations of global variables, i.e., **SFlg**, **SEvnt**, **Stop**, **status**.

The interesting scenario is when the thread starts adding an IO operation by calling the function **Add**. From the function **Add** it calls **IoIncrement** function. In **IoIncrement** just after checking the flag **SFlg** the scheduler decides to change context and choose the other thread to execute. This thread will call the **Stop** function and without any change of context it calls **IoDecrement**, which marks

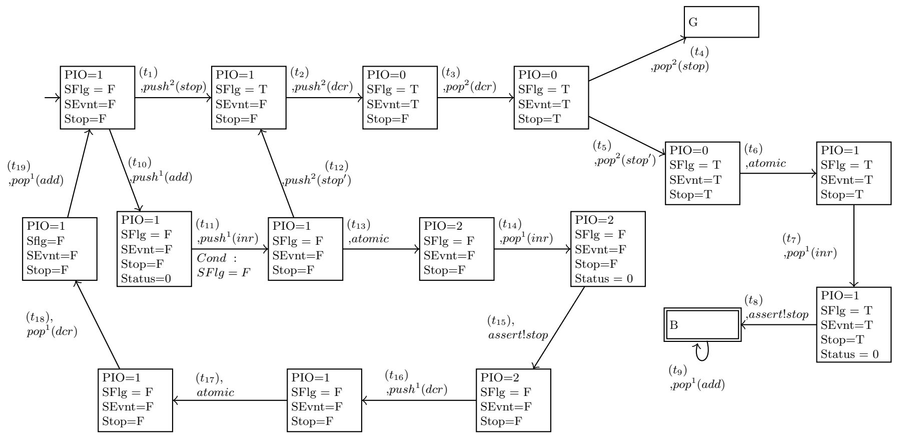
G
(t_4) ,$pop^2(stop)$
PIO=1 SFlg = F SEvnt=F Stop=F
(t_1) ,$push^2(stop)$
PIO=1 SFlg = T SEvnt=F Stop=F
(t_2) ,$push^2(dcr)$
PIO=0 SFlg = T SEvnt=T Stop=F
(t_3) ,$pop^2(dcr)$
PIO=0 SFlg = T SEvnt=T Stop=T
(t_5) ,$pop^2(stop')$
PIO=0 SFlg = T SEvnt=T Stop=T
(t_6) ,$atomic$
PIO=1 SFlg = T SEvnt=T Stop=T
(t_7) ,$pop^1(inr)$
PIO=1 SFlg = T SEvnt=T Stop=T Status = 0
(t_8) ,$assert!stop$
B
(t_9) ,$pop^1(add)$
(t_{19}) ,$pop^1(add)$
(t_{10}) ,$push^1(add)$
(t_{12}) ,$push^2(stop')$
PIO=1 Sflg=F SEvnt=F Stop=F
PIO=1 SFlg = F SEvnt=F Stop=F Status=0
(t_{11}) ,$push^1(inr)$
$Cond$: $SFlg = F$
PIO=1 SFlg = F SEvnt=F Stop=F
(t_{13}) ,$atomic$
PIO=2 SFlg = F SEvnt=F Stop=F
(t_{14}) ,$pop^1(inr)$
PIO=2 SFlg = F SEvnt=F Stop=F Status = 0
(t_{15}), $assert!stop$
(t_{18}), $pop^1(dcr)$
PIO=1 SFlg = F SEvnt=F Stop=F
(t_{17}), $atomic$
PIO=1 SFlg = F SEvnt=F Stop=F
(t_{16}) ,$push^1(dcr)$
PIO=2 SFlg = F SEvnt=F Stop=F

Fig. 1. Part of 2-stack PDA modeling the Bluetooth driver example[1]

the **SFlag** to true and **stop** flag to true as well, which is not a desirable situation in the driver. This is marked as a bad state in Figure. 1.

Upon finding the existence of a run from the start state to the bad state in this automata, BHIM generated the following run as witness. Where, (t_i) represents

(t_{10})	(t_{11})	(t_{12})	(t_2)	(t_3)	(t_5)	(t_6)	(t_7)	(t_8)	(t_9)
$(\downarrow_1)$	$(\downarrow_1)$	$(\downarrow_2)$	$(\downarrow_2)$	$(\uparrow_2)$	$(\uparrow_2)$	(nop)	$(\uparrow_1)$	(nop)	$(\uparrow_1)$

the transitions in Figure 1 and $\downarrow_i, \uparrow_i$ represents push in stack i and pop in stack i respectively in the run.

A.2 Bluetooth 2

Bluetooth 2 is a modified version of the previously mentioned example, where the authors tried to resolve the bug. But as mentioned in [2],[3] this modified version of the driver is not bug free. BHIM was able to find two bugs in this program:

- **Counter under flow error:** where the counter which indicates the number of active threads currently using the driver, can go negative due to a possible schedule of threads. This can lead to some major problem in kernel memory space. BHIM found this bug with 0-holes. It is to be noted that this bug can be found using 3 context changes in the MPDA.
- **Interrupted IO:** There exists a schedule of the threads where, the monitoring thread can stop the driver while other threads are using it. BHIM found this bug using only 2-hole bound on the MPDA. But, this bug is also detectable using 4 context changes.

A.3 Concurrent Insertion in Binary Search Tree (BST)

As explained earlier, we consider the algorithm proposed in [4] which solves this problem for concurrent implementations. But, if the locks are not implemented properly then it is possible for a thread to overwrite others. We have modified this algorithm so that the algorithm becomes buggy and then we tried to model it using MPDA, BHIM was able to detect the bug. In short, we need two stacks to simulate two recursive threads of binary search tree insertion. The problem (reachability query) arises when both threads tries to update the same sub tree, simultaneously which is a bad run of the system. Then we note that to find such bad runs at least 4 context changes are required in the underlying MPDA, BHIM can find such bad runs with bound 2 on holes.

In more detail, assume the following implementation for the `insert` [4] operation in a BST:

```
BSTNode* insert(BSTNode* root,int value)
{
1     if(root == NULL){
2        return createNewNode(value);
3     }
4
5     if(root->data == value){
6         return root;
7     }
8     else if(root->data > value){
9         while(root->leftLock);
10         if(!root->left){
11              root->leftLock = true;
12              root->left = insert(root->left,value);
13              root->leftLock = false;
14         }
15         else{
16              root->left = insert(root->left,value);
17         }
18      }
19      else{
20          while(root->rightLock);
21          if(!root->right){
22              root->rightLock = true;
23              root->right = insert(root->right,value);
24              root->rightLock = false;
25          }
26          else{
27              root->right = insert(root->right,value);
28          }
29      }
30
31      return root;
32
}
```

Consider the BST in Figure 2.

Suppose 2 threads t1 and t2 are invoked simultaneously trying to insert values 25 and 26 respectively and currently are at BSTNode with value 20 (The rightmost node). To reach that node both of the threads require to do recursive calls, which depend on the size of the tree. In between a lot of context switching can happen between the threads, here we present just the context changes upon reaching the destination sub-tree i.e., 20.

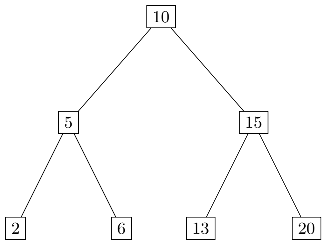

Fig. 2. An example of BST

```
a. t1:
            1. if(root == NULL)  //not true, will go to line 5.
            //switch

b. t2:
            1. if(root == NULL)  //not true, will go to line 5.
            //switch

c. t1:
            5. if(root->data == value){  //not true, will go to line 8.
            8. else if(root->data > value) //not true, will go to line 19.
            //switch

d. t2:
            5. if(root->data == value){  //not true, will go to line 8.
            //switch

e. t1:
            19    else{
            20        while(root->rightLock);  // lock is not held by anyone, so continue.
            21        if(!root->right){
            //switch
f. t2:
            8. else if(root->data > value) //not true, will go to line 19.
            19    else{
            20        while(root->rightLock);  // lock is not helpd by anyone, so continue.
            21        if(!root->right){
            22            root->rightLock = true;
            //switch

g. t1:
            22            root->rightLock = true;
            23            root->right = insert(root->right, value);
            //switch

h. t2:
            23            root->right = insert(root->right, value);
            24            root->rightLock = false;
            //switch
```

As we can see in the above code section f,g and h that both t1 and t2 are entering into critical section without knowing the presence of each other.

Now we wish to check if the following set of instructions executed in one go:

```
20          while(root->rightLock);
21          if(!root->right){
22              root->rightLock = true;
```

Observations:

- We need two stacks to simulate the recursive procedure `insert` on two threads.
- Note, that we can simulate the sequence of recursion call of each thread to destination sub-tree as a hole.
- Here the reachability query is whether there exists a run, where both the threads enter in the same sub tree. Here, they might be in same line of code, but depending on the stack contents it will be determined whether they are indeed accessing the same sub tree or not.

A.4 Maze

We designed the maze example inspired by the examples in [5], where a robot has to visit certain locations meeting some time constraints. There are 9 junction locations in the maze, as shown in Figure 3. The robot enters at point 1, and exits via point 9 when it has spent time T in the maze. This T is a variable which is fed in the algorithm along with the automata. Under differing constraints, we have to find if it is possible for the robot to reach point 9. The robot performs the following tasks. (1) The robot must spend K=5 units of time in the maze; (2) it collects item from location 3 and deposits it at location 5; (3) the time elapse for collection of an object at location 3 to depositing it at location 5 is in the time interval [2,4]; (4) from location 5, it collects another item and deposits it at location 7; (5) the time elapse from collection at location 5 to deposit at location 7 is in the time interval [1,4]; (6) It must move from location 1 to location 4 and from location 4 to location 6 within the interval [2,3]; (7) it can elapse time in the interval [0,1] at locations 3 and 8, and can elapse any time at locations 6 and 4; and (8) it can not elapse time in any other location. At each visit to location 3, the robot collects one item, and on each visit to location 5, it deposits one item collected at location 3 and collects a new item to drop at location 7.

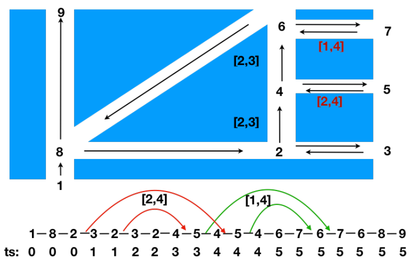

Fig. 3. A Maze, and a witness run.

The first thing that allows us to explore the power of modeling using TMPDA is visiting three locations the same number of time. Depending on a timing constraint the robot may have to visit the locations multiple times. Which can not be handled by the tool described in [5]. This clearly shows the modeling power of TMPDA. As we discussed in Section 6 of the main paper, we run this example with five units of time as a parameter, which the robot has to spend inside the maze. Our tool not only tells us it is possible to reach from the entry point (1) of the maze to the exit point (9) by spending five units of time while meeting all the constraints, it also shows a possible path and where the robot can spend time. This is given as a witness run.

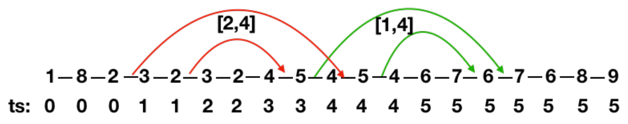

Fig. 4. Witness of Maze

Maze Witness In the run generated by our tool as a witness, the robot starts at location 1, then moves to location 2 via location 8 without elapsing any time. From location 2 it moves to location 3 and collects the item from location 3, where it spends 1 unit of time which is the maximum time it can spend at location 3, then it moves to location 2, where it can not spend time. Now, as the total unit of time elapse from location 1 till location 2 is 1, it can not move to location 4 via location 2. Hence, it visits location 3 again via location 2, elapses 1 unit of time and comes back to location 2. Then it moves to location 4 from the location 2 meeting the timing restrictions. Then from location 4 it has to visit location 5 twice to deposit the items collected from the location 3 and comes back to 4 same number of time collecting a new item. From location 4 it goes to 6 meeting timing restrictions and deposits the items collected from location 5 and by spending one unit time it moves to location 7 and completes its journey to location 9 via location 8.

B Varying Parameters of BHIM

We experimented by varying different parameters like number of *holes*, $c_{\max}$ (for timed setting), number of locations, and number of transitions. Which gives us an excellent idea about the performance of BHIM. Here we are presenting the graphs generated by varying the parameters for the example L_{crit} by modifying the MPDA $\mathcal{A}_{crit}$ to $\mathcal{A}^{\emptyset}_{crit}$ by making the set of accepting states to empty. For the timed case we call the MPDA $\mathcal{A}^{\emptyset}_{Tcrit}$. Recall that we chose $\mathcal{A}^{\emptyset}_{crit}$ because, according to the algorithm 1 if the language accepted by the given MPDA(TMPDA) M is empty, the algorithm will generate all possible lists μ_i, recall by μ_i we represent the nodes of the BFS exploration tree. The number of such μ_i will depend on the number of open *holes* K. As we already discussed that the number of such μ_i is finite; hence, the algorithm will terminate. So, when we run the algorithm with the input $\mathcal{A}^{\emptyset}_{crit}$ the algorithm will check all possible μ_i for an accepting run. In other words, this captures the worst case running time of the tool BHIM. First we will show the performance of BHIM in the un-timed settings and then we will redo the experiments on $\mathcal{A}^{\emptyset}_{Tcrit}$.

Un-timed: The graphs for un-timed system showing the running time of BHIM is in Figures 5, 7. And the memory requirement for the same is shown in Figures 6, 8.

It is clearly seen in Figure 5 that the time is exponential with the holes, and linear with the number of locations as seen in Figure 7.

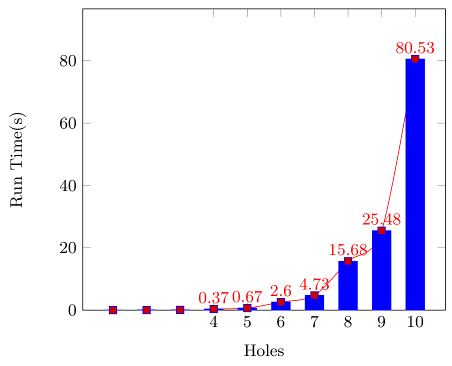

Fig. 5. Performance of BHIM with the number of *holes* varying on input $\mathcal{A}^{\emptyset}_{crit}$

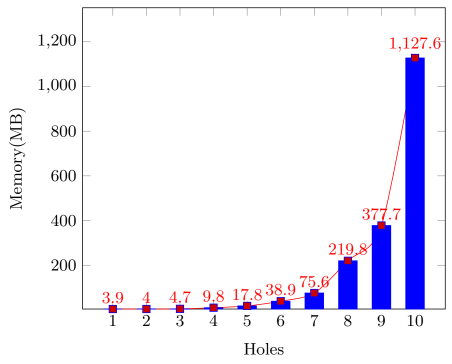

Fig. 6. Memory consumption of BHIM with the number of *holes* varying on input $\mathcal{A}^{\emptyset}_{crit}$

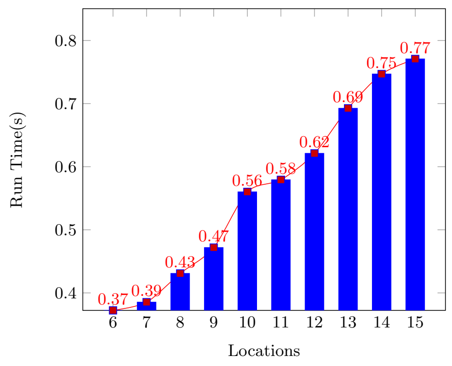

Fig. 7. Performance of BHIM with the number of locations varying on input $\mathcal{A}^{\emptyset}_{crit}$.

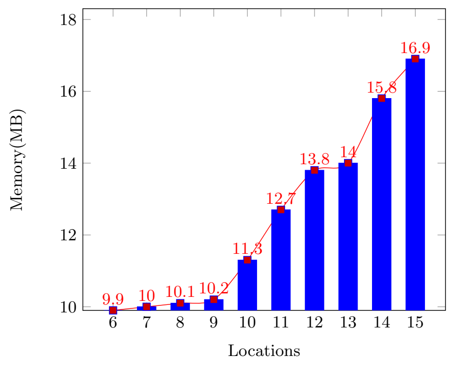

Fig. 8. Memory consumption of BHIM with the number of locations varying on input $\mathcal{A}^{\emptyset}_{crit}$.

Timed: Figure 9 shows the result when we vary the number of *holes* allowed in a given run. When we run the experiments by varying the maximum constant in the TMPDA, we get the results in Fig. 11. Here we can see the time increases polynomially with c_{max} and exponentially with the number of *holes* as expected.

Finally, when we run the experiments by varying the maximum constant ($\mathrm{c_{max}}$) in the TMPDA, we get the results as showed in Fig. 11. We can see increasing the $\mathrm{c_{max}}$ increases the time as a polynomial function. However, when we run experiments by varying the number of clocks, we get the result showed in Fig. 13 and the Graph in indicates that time grows exponentially when the number of clocks increased. For both cases memory consumption is shown in Fig. 12 and Fig. 14 respectively.

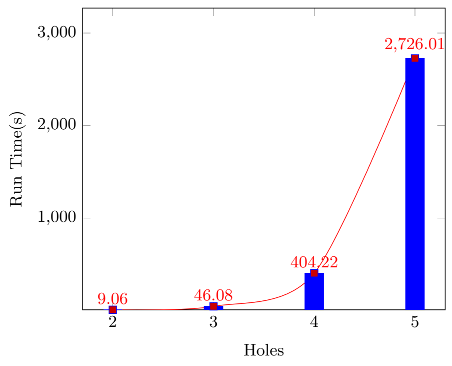

Fig. 9. Performance of BHIM with the number of *holes* varying on $\mathcal{A}^{\emptyset}_{Tcrit}$ as input

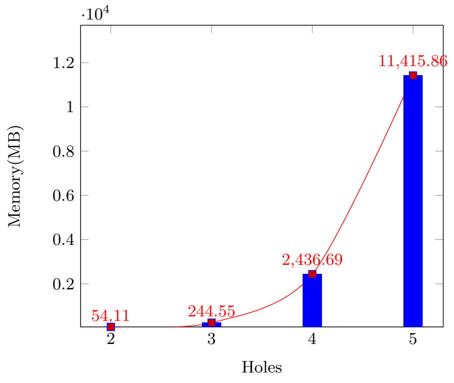

Fig. 10. Memory consumption of BHIM with the number of *holes* varying on $\mathcal{A}^{\emptyset}_{Tcrit}$ as input

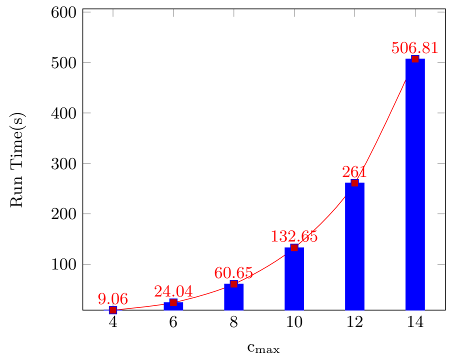

Fig. 11. Performance of BHIM with $c_{\max}$ varying on $\mathcal{A}^{\emptyset}_{Tcrit}$ as input

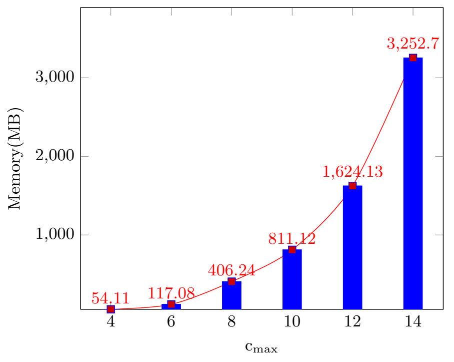

Fig. 12. Memory consumption of BHIM with $c_{\max}$ varying on $\mathcal{A}^{\emptyset}_{Tcrit}$ as input

We also checked the performance by varying the number of locations of TMPDA. As expected, the time increased polynomially with the increase of locations in the TMPDA c.f. Graph in Fig. 15 and for memory in Fig. 16

B.1 Varying parameters of L_{bh}

We also did the same for L_{bh} example, we constructed the MPDA $M_{L_{bh}}$ which accepts the language L_{bh} and then modified $M_{L_{bh}}$ to make its set of final states empty, so that BHIM generates all possible list (μ) and returns emptiness. As we discussed earlier this pushes BHIM to its worst case complexity. Details can be found in the Tables 8, 9 and in the Graphs 17, 18.

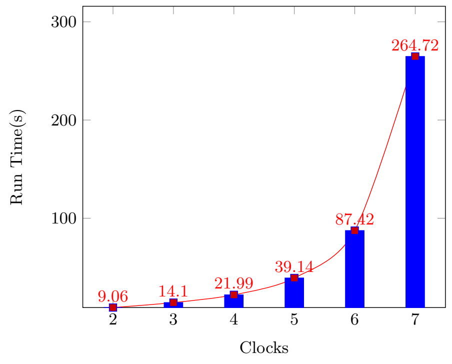

Fig. 13. Performance of BHIM with the number of clocks varying on $\mathcal{A}^{\emptyset}_{Tcrit}$ as input.

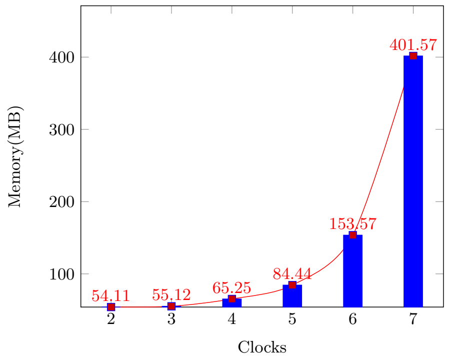

Fig. 14. Memory consumption of BHIM with the number of clocks varying on $\mathcal{A}^{\emptyset}_{Tcrit}$ as input.

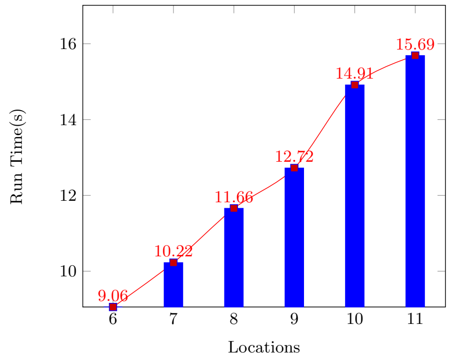

Fig. 15. Performance of BHIM with the number of locations varying on $\mathcal{A}^{\emptyset}_{Tcrit}$ as input.

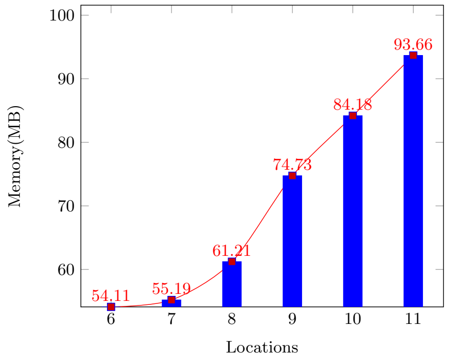

Fig. 16. Memory consumption of BHIM with the number of locations varying on $\mathcal{A}^{\emptyset}_{Tcrit}$ as input.

B.2 Varying parameters on non-empty MPDA

Bluetooth parameterized non-empty

Changing Holes As we discussed above, there exists a run in the Bluetooth example from the start state to the bad state, which is well-nested. Which means that our algorithm will get the possible run from `WR` as described in the Algorithm 1 and increasing the holes does not change the time required to compute `WR`. Note that the model has no clocks but timed stacks; hence the column c_{max} will contain maximum constant for stacks. c.f. Table 1

L^{crit} This example is one of the most simple examples that we worked on, and the parameters of this example, like c_{max}, Clocks can easily be changed to see

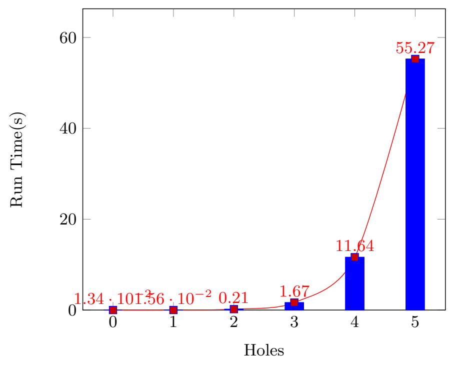

Fig. 17. Performance of BHIM with the number of *holes* varying as in Table 8

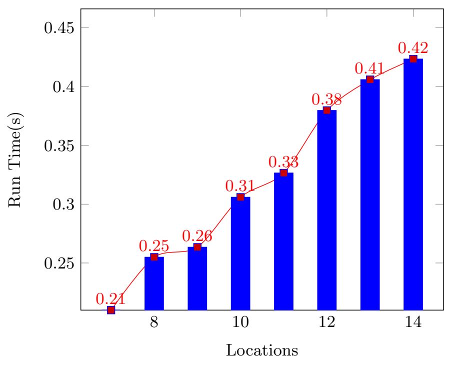

Fig. 18. Performance of BHIM with the number of locations varying as in Table 9

Clocks	Stacks	Locations	Transitions	c_{max}	Holes	Time(sec)	Memory(KB)	Empty(Y/N)
0	2	57	96	2	0-5	∼0.1543	∼5248KB	N

Table 1. Bluetooth With Changing Holes

the impact on running time. Below we present some tables generated by varying the parameters.

Varying Maximum Constant($\mathbf{c_{max}}$) Table 5 shows the run time and memory consumption by the algorithm when the maximum constant(c_{max}) is varied. Note, that the automaton is non-empty so it has a run, and as soon as it finds a path from the start state to the final state it stops and produces an accepting run using witness algorithm as described in the main paper.

Varying Holes Changing *holes* should increase time exponentially but, in this example as the automaton has a run with only 2 *holes* so it terminates when ever it finds the final state. Even if we allow more *holes* the program terminates before introducing more *holes*, which is why time saturates after 3 *holes*. Look at Table 6 for details.

Increasing Clocks Though the increasing number of clocks in this current model makes no sense, due to the lack of proper benchmarks, we had to find some way to understand the scalability of our algorithm so, we randomly increased the number of clocks with a fixed value of clock constraint, so that we get some idea about how the algorithm scales when the clocks are increased for a given example. Increasing clocks increases the number of states exponentially. So, every time we add a clock, the number of state valuation pairs multiplies according to the maximum constant of that clock, which in turn increases the time to compute transitive closure. On the other hand, the number of lists depends on the state valuation pairs, which are reachable from the previous state valuation pairs by some transitions. The addition of an extra clock does not change the number of reachable pairs in this example. Hence, even if the number of state valuation pairs increased, the number of lists μ generated remain same s.f. Table 7

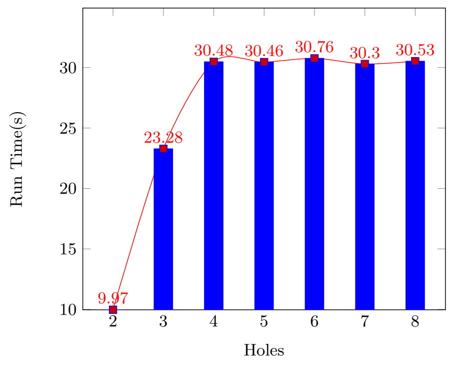

Fig. 19. Performance of BHIM with varying number of holes as in Table 6

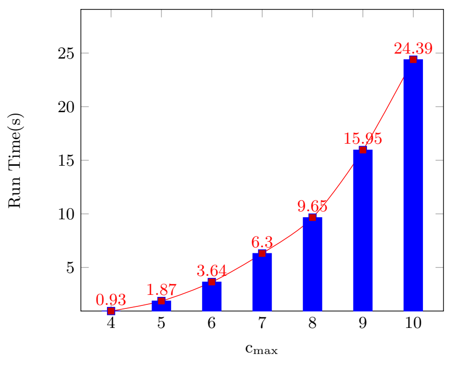

Fig. 20. Performance of BHIM with the c_{max} varying as in Table 5

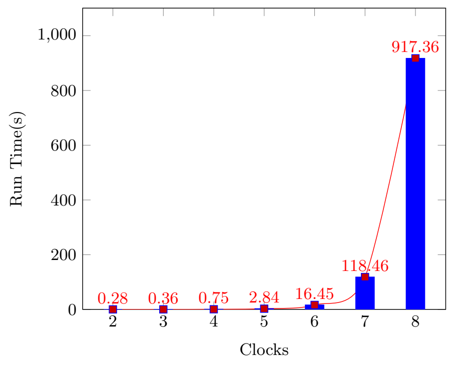

Fig. 21. Performance of BHIM with varying number of clocks as in Table 7

C Scalability

We also tried to run scalability test of BHIM by checking maximum number of holes BHIM can handle within a given memory limit. Here we used the MPDA $\mathcal{A}^{\emptyset}_{crit}$, $\mathcal{A}^{\emptyset}_{Tcrit}$ and $\mathcal{A}^{\emptyset}_{prodcon}$. Just like $\mathcal{A}^{\emptyset}_{crit}$ and $\mathcal{A}^{\emptyset}_{Tcrit}$ we also modified the producer consumer MPDA to $\mathcal{A}^{\emptyset}_{prodcon}$ for scalability analysis. Table 2, Table 3 and Table 4 shows the scalability of BHIM on $\mathcal{A}^{\emptyset}_{crit}$, $\mathcal{A}^{\emptyset}_{Tcrit}$, and $\mathcal{A}^{\emptyset}_{prodcon}$ respectively with memory limit of 8GB.

Stacks	Location	Transitions	Holes	Time(sec)	Memory(MB)
2	6	9	0	0.015	3.889
2	6	9	1	0.0110	3.892
2	6	9	2	0.0371	4.096
2	6	9	3	0.0891	4.788
2	6	9	4	0.3723	9.960
2	6	9	5	0.6749	17.776
2	6	9	6	2.5988	38.964
2	6	9	7	4.7267	75.564
2	6	9	8	15.681	219.844
2	6	9	9	25.4822	377.624
2	6	9	10	80.5287	1127.9
2	6	9	11	151.612	1937.5
2	6	9	12	211.584	5468.2
2	6	9	13	662.98	8GB(Killed)

Table 2. $\mathcal{A}^{\emptyset}_{crit}$ scalability with respect to holes and memory limited to 8GB

Clocks	Stack	Locations	Transitions	$c_{\max}$	Holes	Time(sec)	Memory(MB)
2	2	6	9	4	0	0.0802	5.7
2	2	6	9	4	1	0.09	5.7
2	2	6	9	4	2	67.126	40.6
2	2	6	9	4	3	385.59	1980.220
2	2	6	9	4	4	858.9	8GB(Killed)

Table 3. $\mathcal{A}^{\emptyset}_{Tcrit}$ scalability with respect to holes and memory limited to 8GB

Scalability in terms of states: We also tried to check the scalability of BHIM by increasing the number of states of MPDA. We were able to handle 4000 state MPDA with 0 holes, in 9181 sec and 4.7GB of memory.

Stacks	Location	Transitions	Holes	Time(sec)	Memory(MB)
2	11	15	0	0.0120	5.7
2	11	15	1	0.0144	5.78
2	11	15	2	0.2042	6.096
2	11	15	3	0.33345	6.3
2	11	15	4	1.9466	17.60
2	11	15	5	3.99	34.776
2	11	15	6	15.5988	132.964
2	11	15	7	27.7267	295.05
2	11	15	8	88.5287	742.520
2	11	15	9	146.137	1509.844
2	11	15	10	433.4822	3554.624
2	11	15	11	640.612	7755.520
2	11	15	12	780.584	8GB(Killed)

Table 4. $\mathcal{A}^{\emptyset}_{prodcon}$ scalability with bounded memorys

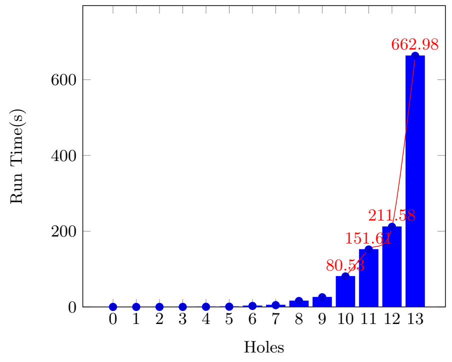

Fig. 22. Scalability of BHIM with the number of *holes* varying with memory limited to 8GB(8192MB) as in Table 2

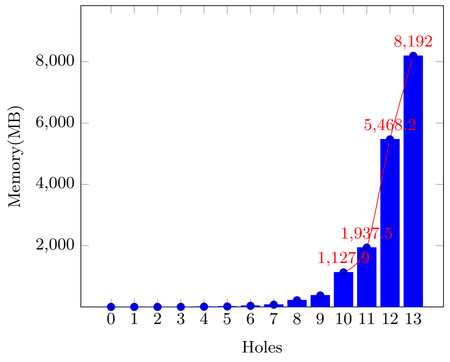

Fig. 23. Memory consumption of BHIM with number of *holes* varying and memory limited to 8GB(8196MB) as in Table 2

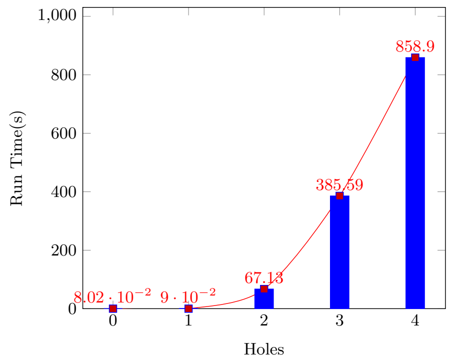

Fig. 24. Scalability of BHIM with the number of *holes* varying as with memory limited to 8GB Table 3

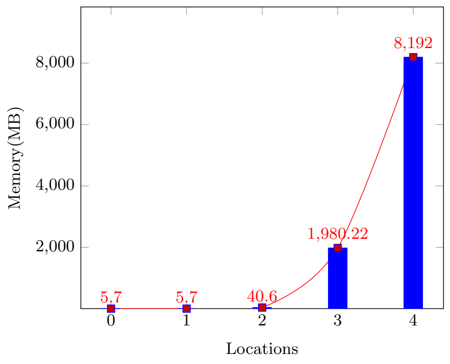

Fig. 25. Memory consumption of BHIM with the number of *holes* varying with memory limited to 8GB as in Table 3

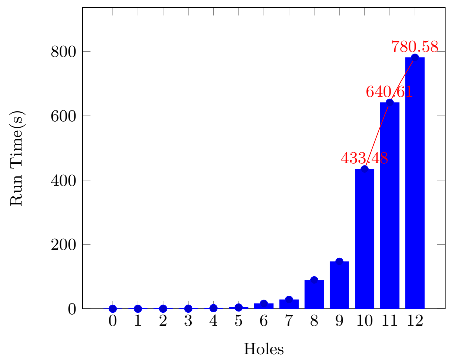

Fig. 26. Scalability of BHIM with the number of *holes* varying with memory limited to 8GB(8192MB) as in Table 4

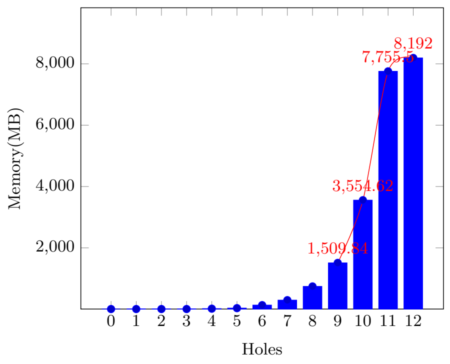

Fig. 27. Memory consumption of BHIM with varying holes and memory bounded to 8GB(8196MB) as in Table 4

300 ## D Tables

Clocks	Stack	Locations	Transitions	c_{max}	Holes	Time(sec)	Memory(KB)	Empty(Y/N)
2	2	6	9	4	2	0.9326	28804	N
2	2	6	9	5	2	1.8694	54056	N
2	2	6	9	6	2	3.6386	103904	N
2	2	6	9	7	2	6.3043	204560	N
2	2	6	9	8	2	9.6519	303012	N
2	2	6	9	9	2	15.9538	404908	N
2	2	6	9	10	2	24.3861	811040	N

Table 5. L_{crit} With Changing Maximum Constant

Clocks	Stack	Locations	Transitions	c_{max}	Holes	Time(sec)	Memory(KB)	Empty(Y/N)
2	2	6	9	8	2	9.9652	203396	N
2	2	6	9	8	3	23.2813	414100	N
2	2	6	9	8	4	30.4787	443064	N
2	2	6	9	8	5	30.4565	443192	N
2	2	6	9	8	6	30.7647	443236	N
2	2	6	9	8	7	30.2986	443120	N
2	2	6	9	8	8	30.5274	443188	N

Table 6. L_{crit} With Changing Holes

Clocks	Stack	Locations	Transitions	c_{max}	Holes	Time(sec)	Memory(KB)	Empty(Y/N)
2	2	6	9	4	2	0.276977	7336	N
3	2	6	9	4	2	0.362076	8268	N
4	2	6	9	4	2	0.753226	12880	N
5	2	6	9	4	2	2.84098	29020	N
6	2	6	9	4	2	16.4543	104832	N
7	2	6	9	4	2	118.46	397944	N
8	2	6	9	4	2	917.355	1580424	N

Table 7. L_{crit} With Changing Clocks

301 b

Stacks	Location	Transitions	Holes	Time(sec)	Memory(KB)	Empty(Y/N)
2	7	13	0	0.0134	4128	Y
2	7	13	1	0.0156	4200	Y
2	7	13	2	0.2099	7300	Y
2	7	13	3	1.6664	33408	Y
2	7	13	4	11.6436	297656	Y
2	7	13	5	55.2659	1408160	Y

Table 8. $M_{L_{bh}}$ Varying Holes

Stacks	Location	Transitions	Holes	Time(sec)	Memory(KB)	Empty(Y/N)
2	7	13	2	0.2099	7300	Y
2	8	14	2	0.2549	9804	Y
2	9	15	2	0.2634	9904	Y
2	10	16	2	0.3059	9996	Y
2	11	17	2	0.3255	10600	Y
2	12	18	2	0.3797	10164	Y
2	13	19	2	0.4059	15736	Y
2	14	20	2	0.4234	15876	Y

Table 9. $M_{L_{bh}}$ Varying Locations

Stacks	Location	Transitions	Holes	Time(sec)	Memory(KB)	Empty(Y/N)
2	6	9	4	0.3723	9960	Y
2	7	10	4	0.3853	9996	Y
2	8	11	4	0.4308	10092	Y
2	9	12	4	0.4718	10160	Y
2	10	13	4	0.5600	11336	Y
2	11	14	4	0.57922	12763	Y
2	12	15	4	0.62017	13808	Y
2	13	16	4	0.692599	14028	Y
2	14	17	4	0.746832	15716	Y
2	15	18	4	0.770715	16918	Y

Table 10. $\mathcal{A}^{\emptyset}_{crit}$ Varying Locations-Transitions

Clocks	Stacks	Location	Transitions	$c_{\max}$	Holes	Time(sec)	Memory(KB)	Empty(Y/N)
2	2	6	9	4	2	9.05609	54108	Y
2	2	6	9	6	2	24.0387	117076	Y
2	2	6	9	8	2	60.6463	406236	Y
2	2	6	9	10	2	132.648	811124	Y
2	2	6	9	12	2	261.001	1624132	Y
2	2	6	9	14	2	506.806	3252700	Y

Table 11. $\mathcal{A}^{\emptyset}_{Tcrit}$ Varying $c_{\max}$

Clocks	Stacks	Location	Transitions	$c_{\max}$	Holes	Time(sec)	Memory(KB)	Empty(Y/N)
2	2	6	9	4	2	9.05609	54108	Y
3	2	6	9	4	2	14.1044	55124	Y
4	2	6	9	4	2	21.9928	65248	Y
5	2	6	9	4	2	39.1368	84440	Y
6	2	6	9	4	2	87.4203	153572	Y
7	2	6	9	4	2	264.71	401572	Y

Table 12. $\mathcal{A}^{\emptyset}_{Tcrit}$ Varying Clocks

Clocks	Stacks	Location	Transitions	$c_{\max}$	Holes	Time(sec)	Memory(KB)	Empty(Y/N)
2	2	6	9	4	2	9.05609	54108	Y
2	2	7	10	4	2	10.2249	55192	Y
2	2	8	11	4	2	11.6579	61212	Y
2	2	9	12	4	2	12.7217	74732	Y
2	2	10	13	4	2	14.913	84176	Y
2	2	11	14	4	2	15.6878	93660	Y

Table 13. $\mathcal{A}^{\emptyset}_{Tcrit}$ Varying Locations

References

1. Shaz Qadeer and Dinghao Wu. Kiss: keep it simple and sequential. *Acm sigplan notices*, 39(6):14–24, 2004.
2. Sagar Chaki, Edmund Clarke, Nicholas Kidd, Thomas Reps, and Tayssir Touili. Verifying concurrent message-passing C programs with recursive calls. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, page 334–349. Springer, 2006.
3. Gaël Patin, Mihaela Sighireanu, and Tayssir Touili. Spade: Verification of multithreaded dynamic and recursive programs. In *International Conference on Computer Aided Verification*, pages 254–257. Springer, 2007.
4. HT Kung and Philip L Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems (TODS)*, 5(3):354–382, 1980.
5. S. Akshay, Paul Gastin, Shankara Narayanan Krishna, and Ilias Sarkar. Towards an efficient tree automata based technique for timed systems. In *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*, pages 39:1–39:15, 2017.