Improvements for a Simple PDR Implementation

Yu-Wei, Fan
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan
b07901047@ntu.edu.tw

***Abstract*—This report presents the a more efficient version of PDR [?] (Property Directed Reachability) implementation based on the given one. The given simple PDR implementation, called *SIMP*, suffers from severe run-time and memory leak issues. Also, SIMP can't generate a counter-example if the property is violated. From the experiments, our improvements improve both run-time and memory a lot comparing to SIMP and is comparable with the *V3* built in PDR model checker in most of the cases. The improvements is organized as the following. Firstly, we implement the counter-example trace to the *Cube* data structure to generate a counter example when the property is violated. Secondly, *ternary simulation* technique and related optimization details will be shown in the next part. Then, we will describe the reason for memory leak in SIMP and how to fix it. Finally, experiments result for vending machine and HWMCC benchmarks will be demonstrated in the last section.**

***Index Terms*—formal verification, reachability, model checking**

I. Counter Example Trace

THE Cube data structure in SIMP only contain a data member latchValues, which stores the state of this cube, making it unable to record the counter-example trace. To do so, we add two extra data members. The first is *nxt*, a pointer to a Cube. The second is *inputV*, a vector of bool representing an input pattern. With these two data members, we properly maintain Cubes such that, given a cube and its inputV, the next state is the Cube nxt pointing to. We also maintain a pointer to a Cube, head, in PDRMgr. Therefore, whenever the counter example is found, we simply traverse through a sequence of Cube from the head and print out the corresponding input patterns until meeting a NULL nxt pointer.

To maintain such property, we only set the input pattern and nxt pointer when either function getBadCube(Cube c) or solveRelative(Cube c) is called. When getBadCube() is called, since the cube is in the last frame, we set the nxt pointer to NULL and record the input pattern that will intersect !P. When solveRelative(Cube c) for Q2 (EXTRACTMODEL) is called, it will return a pre-image for Cube c. By definition, we'll set the nxt pointer in the pre-image to c, and record the input pattern that lead the pre-image to c.

II. Ternary Simulation

One of the main technique that makes PDR so efficient is *ternary simulation*, which greatly improves the quality of learnt cubes. In this section, we will describe the implementation and some optimization.

A. Implementation

Ternary simulation is called by solveRelative (Cube c) and getCube (). When solveRelative (Cube c) is SAT, let c be the image and the model be the pre-image. We will flip latch value to X in c one-by-one. As long as the X-value doesn't propagate to the image, we can remove that flipped latch from the pre-image, making it smaller. Please refer to algorithm.1.

Algorithm 1 ternarySim (preImg, img, isMonitor)

```
1: for $i = 1, 2, \dots, latchSize$ do
2:     $preImg.latch[i] \leftarrow X$
3:     if $!isMonitor$ and $propagate(img)$ then
4:         restore $preImg.latch[i]$
5:     end if
6:     if $isMonitor$ and $propagate(monitor)$ then
7:         restore $preImg.latch[i]$
8:     end if
9: end for
10: return (preImg)
```

B. Simulation Optimization

In this subsection, we describe several optimization technique to speed up simulation procedure.

1) On-the-way X-Value Check: Observe that as long as any latch value of img changes from 0/1 to X, the flipped latch value in pre-img has to be restored. At the same time, no further simulation is needed. Therefore, we first construct an orderedNets in topological order, and record the position for the latches of img. Whenever a X is propagated to img, we break the simulation.

2) Care-Latch Fanin Cone Construction: From the previous observation, we also notice that if the original latch value of img is X, it is a don't-care in a sense that we don't need to perform simulation on it. Therefore, we skip those X-value latches when constructing the orderedNets.

3) Monitor Net Reusing: The monitor net will be used when called by getBadCube(). The target is monitor, which is always *true*. We thus reuse the monitor net by making it a data member, since the net will not change during the solving process.

III. Memory Issue

The reason that SIMP suffers from memory leak is the improper coding style and data structure design. In the original version, a Cube will be new in a local function, and return the pointer. In the outer function, the LHS pointer will be iteratively overwritten by the return pointer without deletion, causing severe memory leak. To resolve this problem, we modified some part of the code in a better coding style to prevent memory leak.

- Using $vector\langle type\rangle$: Instead of using pointer to store fixed-length data, we use dynamic-vector provided in C++ STL to simplify memory management.
- Using object rather than pointer: When the data is local to the function, there is no need to new a object. Simply declaring a local object to represent it is enough.
- Remember to delete a pointer: Whenever a pointer is going to be overwritten by another one, and can't no longer be accessed, always remember to delete it in advance.

IV. Verification Result for Vending Machine

The verification result is listed below (table I). Vending.v is the original buggy vending machine RTL design, while Vending-fixed.v is the fixed one with extra monitors. We compare our implementation called *PDR_mine* with the V3 built in PDR engine called *PDR_ref*. Both engines can prove the monitor p violated in Vending.v in a reasonable time. Only PDR_ref can prove p safe in Vending-fixed.v. Only PDR_mine can prove p_item safe in Vending-fixed.v. Other properties are still undecided in for both engines.

Ver./design	Vending.v	Vending-fixed.v
ITP_mine	0.08	4.38
ITP_ref	0.03	8.8

TABLE I

V. Experiments

The experimental results were carried out on Linux machine with Intel i7-8550U 4GHz processors. The time limit was set to 900 s and the memory limit to 16GB.

For the evaluation, we used the HWMCC benchmarks containing 45 cases. We compared our implementation with PDR_ref. Table.II shows that our implementation is slightly slower than PDR_ref in most of the cases while memory usage is comparable. PDR_mine is significantly outperformed by PDR_ref in the last cases.

To evaluate the effectiveness of the above-mentioned optimization technique, we compare SIMP with only ternary simulation (T), T + memory opt.(M) and T + S + optimization(M), which is PDR_mine. From table.III, it shows that with ternary simulation, SIMP can solve almost all the cases. SIMP+T with sim. opt. is much more faster than the one without it, which reveals that simulation effectiveness greatly affects the overall performance. We notice that among the three sim. opt. technique, Care-latch Fanin cone Construction is the most powerful one. It is clear that after we re-implemented some part of the code prone to create severe memory leak, memory usage becomes reasonable.

Cases/Version	PDR_ref		PDR_mine	
	time(s)	memory(MB)	time(s)	memory(MB)
6s209b0	1.09	30.2	2.72	31.5
6s210b037	0.05	11.4	0.22	8.22
6s210b105	0.04	11.6	0.2	8.26
6s215rb0	2.92	13.4	9.12	15
6s221rb18	0.94	157.3	0.65	158
6s275rb253	0.76	17.2	2.58	20
6s275rb318	0.86	17.2	4.55	21
6s277rb292	0.98	17.2	3.93	15.8
6s277rb342	0.67	17	3.27	16
6s282b01	0.03	13.9	0.05	12.5
6s289rb00529	0.31	52.2	3.4	62.7
6s289rb05233	0.3	52.6	1.8	62.4
6s291rb18	1.39	11	6.6	5.35
6s291rb77	1.38	11	2.44	5.4
6s317b14	13.79	16.2	17.13	9
6s317b18	14.93	16.1	3.53	7
6s318r	0	8.5	0.01	5.45
6s322rb265	2.17	255	1.68	203
6s325rb072	0.06	14.5	0.24	13.2
6s327rb10	0.5	19.7	1.25	18.32
6s327rb19	0.05	19.1	0.07	15.8
6s330rb06	0.22	52.7	2.13	58.9
6s330rb11	0.23	54.3	0.62	58.3
6s335rb09	0.09	0.74	0.19	9.67
6s335rb60	0.05	11.8	0.67	10.47
6s344rb054	0.12	39.2	0.17	28.8
6s362rb1	0.11	11.1	0.03	8.43
6s372rb26	0.65	11.5	17.44	20.8
6s380b129	0.88	30.1	12.45	38
6s381rb630	0.45	48	3.83	57
6s384rb024	0.2	41.6	0.61	47
6s388b07	0.05	20.8	0.06	21.6
6s388b09	0.05	20.8	0.05	21.7
6s389b02	0.05	20.7	0.03	21.6
6s389b11	0.16	21.1	1.57	22.7
6s391rb379	0.03	13.4	0.04	11.8
6s400rb07819	0.19	53.8	0.21	58.8
6s406rb067	0.36	62.5	2.94	64.58
6s421rb050	0.22	10.6	1.5	9
6s421rb083	0.27	10.8	1.49	8.9
6s515rb1	0	8.6	7.95	8.12
oski1rub05	0.21	80.6	0.3	85.8
oski1rub05i	0.19	57.9	0.24	79.8
oski1rub06	0.21	81.5	0.31	85.8
oski3ub2i	0.86	42.7	90.65	26.2

TABLE II: runtime & memory

VI. Conclusion

In this report, we improve SIMP a lot through ternary simulation, simulation optimization and modified data structure. Experiments have shown how these improvements work. The final version PDR_mine is almost comparable with PDR_ref. For further improvements, fast containment check for Cube is an option, since PDR also spends a lot of timing checking Cube containment while blocking cube.

VII. Final Submission Link

https://drive.google.com/drive/folders/ 1oHbunCv0dvssc3Yj2KrZjEyknWyDQLjO?usp= sharing

Cases/Version	SIMP+T		SIMP+TM	
	time(s)	memory(MB)	time(s)	memory(MB)
6s209b0	543.7	3426	65.33	31.5
6s210b037	2.178	36.59	0.57	8.22
6s210b105	4.98	36.55	0.56	8.26
6s215rb0	309.2	1423	95.4	15
6s221rb18	1.21	209	0.7	158
6s275rb253	743.8	7020	63.8	20
6s275rb318	697.4	6522	74.4	21
6s277rb292	731	6410	8.62	15.8
6s277rb342	458	4711	8.12	16
6s282b01	2.91	59.4	0.28	12.5
6s289rb00529	TO	-	132.5	62.
6s289rb05233	199.8	4395	88.13	62.4
6s291rb18	57.95	1115	0.07	5.35
6s291rb77	175.1	1625	0.06	5.4
6s317b14	37.1	42	20.93	9
6s317b18	34	43	5.78	7
6s318r	0.61	12	0.08	5.45
6s322rb265	2.17	383	1.42	203
6s325rb072	47.5	448	6.35	13.2
6s327rb10	223	1454	34.5	18.32
6s327rb19	1.43	42.5	0.05	15.8
6s330rb06	151.8	3160	92.34	58.9
6s330rb11	61.8	403	7.62	58.3
6s335rb09	20.56	224.8	5.4	9.67
6s335rb60	86.4	860	14.83	10.47
6s344rb054	0.38	52.6	0.16	28.8
6s362rb1	4.58	85.9	0.38	8.43
6s372rb26	TO	-	83.34	20.8
6s380b129	TO	-	281.7	38
6s381rb630	512	5877	112.3	57
6s384rb024	309	5616	48.7	47
6s388b07	1.58	24	0.06	21.6
6s388b09	1.53	24	0.05	21.7
6s389b02	1.23	24	0.04	21.6
6s389b11	29.4	262	51.87	22.7
6s391rb379	5.86	85	0.02	11.8
6s400rb07819	0.1	74.3	0.23	58.8
6s406rb067	540	2293	29.47	64.5
6s421rb050	21.6	178	6.78	9
6s421rb083	21.6	171.4	7.48	8.9
6s515rb1	10.3	68	13.7	8.12
oski1rub05	0.54	86.8	0.28	85.8
oski1rub05i	0.3	80.8	0.23	79.8
oski1rub06	0.39	86.8	0.3	85.8
oski3ub2i	690.6	3020	81.8	26.2

TABLE III: Optmization Effectiveness.