Using Cubes of Non-state Variables With Property Directed Reachability

John D. Backes and Marc D. Riedel

Department of Electrical and Computer Engineering
University of Minnesota
200 Union St. S.E., Minneapolis, MN 55455
{back0145, mriedel}@umn.edu

***Abstract*—A new SAT-Based algorithm for symbolic model checking has been gaining popularity. This algorithm, referred to as "Incremental Construction of Inductive Clauses for Indubitable Correctness" (IC3) or "Property Directed Reachability" (PDR), uses information learned from SAT instances of isolated time frames to either prove that an invariant exists, or provide a counter example. The information learned between each time frame is recorded in the form of cubes of the state variables. In this work, we study the effect of extending PDR to use cubes of intermediate variables representing the logic gates in the transition relation. We demonstrate that we can improve the runtime for satisfiable benchmarks by up to 3.2X, with an average speedup of 1.23X. Our approach also provides a speedup of up to 3.84X for unsatisfiable benchmarks.**

I. Introduction

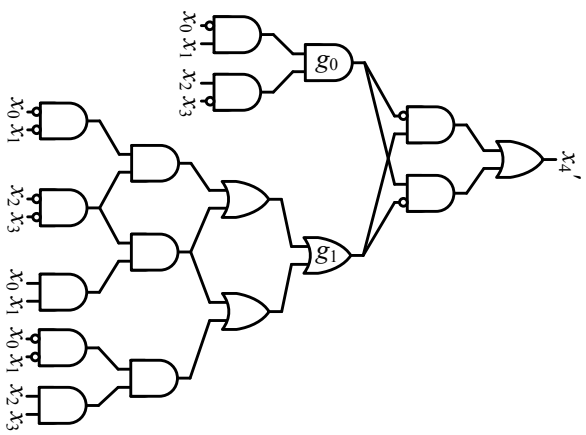
Methods for symbolic model checking originally focused on the use of Binary Decision Diagrams (BDDs) [3], [9], [5]. These algorithms iteratively compute the image of a current set of states until a fixed point is reached, or the property is violated. While BDDs can very quickly verify certain properties, often times the size of the BDD can explode during image computation.

SAT-Based techniques based on induction have also been proposed [12]. These techniques unroll a circuit across multiple time frames and prove that a property holds over the longest loop-free path. Problems with long loop-free paths become intractable because they require solving a very large unrolling of the circuit. In 2003, McMillan proposed a SAT-Based method for model checking based on Craig's Interpolation Theorem [10], [6]. In this algorithm, interpolants are used to compute an over approximation of the set of reachable states. If the over approximation converges to a fixed point, the property is proved to hold. Interpolation can solve many instances faster than induction because the maximum number of transitions needed to prove a property is bounded by the diameter of the state space. However, if a counter example is discovered, it may be spurious. To block these spurious counter examples, interpolants may need to be generated from large unrollings of the circuit.

In 2011, Bradley proposed "Incremental Construction of Inductive Clauses for Indubitable Correctness" (IC3), a new SAT-Based method for symbolic model checking that does not require solving large unrollings, or generating messy abstractions of the reachable state space [2]. The technique works by iteratively solving a SAT-instance representing a single time frame of the underlying circuit. States that violate the property are recorded in the form of cubes of state variables. These cubes then must be blocked recursively by previous time frames. The process halts under two conditions; either a set of cubes extending from the initial state are found to reach a cube that violates the property, or the set of cubes blocked in one frame are shown to be blocked inductively in every future frame (proving the property to be invariant).

The IC3 algorithm works well when it is able to produce very small cubes (covering many states). Others have proposed an improved implementation of the algorithm referred to as "Property Directed Reachability" (PDR) [8]. One of the main improvements in PDR is the use of ternary valued simulation to reduce the size of state cubes. Using ternary valued simulation allows cubes to be shortened quickly without putting an unnecessary burden on the solver.

In this work, we extend the framework of PDR to allow for cubes containing *functions* of state variables. This idea is illustrated with an example in Figure 1. Figure 1 shows a truth table for an incompletely specified Boolean function for next state variable x'_4 and a circuit level implementation of x'_4. Assume that x_4 must be blocked in the next frame. There are four cubes of state variables in the current frame that need to be blocked to make this so: $\bar{x_0} \wedge \bar{x_1} \wedge \bar{x_2} \wedge \bar{x_3}$, $\bar{x_0} \wedge \bar{x_1} \wedge x_2 \wedge x_3$, $\bar{x_0} \wedge x_1 \wedge x_2 \wedge \bar{x_3}$, and $x_0 \wedge x_1 \wedge \bar{x_2} \wedge \bar{x_3}$. In this example, none of these cubes can be simplified to smaller cubes. However, there are only two cubes in terms of variables g_0 and g_1 that must be blocked to prevent the justification of x_4 in the next frame: $\bar{g_0} \wedge g_1$ and $g_0 \wedge \bar{g_1}$. In this example, only half as many cubes in terms of intermediate logic variables need to be added to the current frame in order to block x'_4.



x_0, x_1, x_2, x_3	g_0, g_1	x'_4
0 0 0 0	0 1	1
0 0 1 1	0 1	1
0 1 1 0	1 0	1
1 1 0 0	0 1	1

Fig. 1. An example netlist where fewer cubes in terms of intermediate variables need to be blocked than cubes in terms of state variables

978-3-9815370-0-0/DATE13/©2013 EDAA

We study the affect of extending PDR to allow cubes of intermediate logic variables. We then present the results of our implementation on the HWMCC '11 benchmarks [1].

II. Background and Definitions

A. Definitions and Notation

This paper adopts the same definitions and notations originally adopted by Bradley in [2]. However, we use the following convention for defining a finite state machine (FSM). A FSM $M = \{Z, X, I, T\}$ consists of a set of primary input variables Z, a set of state variables X, a set of initial states $I \subseteq \{0,1\}^X$, and a transition relation $T \subseteq \{0,1\}^X \times \{0,1\}^X$. We use an apostrophe ($x'$) to indicate state variables in the next state of a transition relation. When we apply an apostrophe to a set (X') we are indicating that we are referring to the next state variables of the set.

B. Review of the PDR Algorithm

In this section we briefly review the Property Directed Reachability algorithm. The algorithm operates on sets of clauses, denoted by F_i, called *frames*. The clauses in frame F_i symbolically encode an over approximation of the states that are reachable up to the ith time frame of an FSM. In other words, if state s can be reached within i steps of the initial states, then $F_i \wedge s$ is satisfiable. The collection of these frames is referred to as the *trace*. The trace maintains the following properties.

1) The 0th frame only contains the initial states ($F_0 = I$)
2) Every assignment that satisfies the current frame also satisfies the next frame ($F_i \rightarrow F_{i+1}$)
3) Every state that can be reached in one transition from a state in the current frame, satisfies the next frame ($F_i \wedge T \rightarrow F_{i+1}$)
4) The property is satisfied in every frame except the last one ($F_i \rightarrow P$ for every F_i except F_n)

At the start of the algorithm, there exists just one frame $F_0 = I$. Each major iteration of the algorithm starts by checking to see if the property can be violated in one transition from the states in the highest frame. This is done by solving the SAT instance: $F_n \wedge T \wedge \bar{P}$. If this query is satisfiable, a satisfying state cube $s \models F_n \wedge T \wedge \bar{P}$ is extracted. The algorithm then proceeds to see if this cube can be blocked by the previous frame. This is done by solving the SAT instance: $F_{n-1} \wedge T \wedge s'$. If this query is satisfiable, a satisfying state cube is extracted from this SAT instance. The algorithm continues to try to block cubes in each previous frame. If, eventually, a cube cannot be blocked by the initial frame, F_0, a counter example is provided.

Whenever a cube is successfully blocked in some frame F_i, a clause blocking this cube is added to every frame F_k where $k \leq i$. This maintains property 2 of the trace. A clever improvement to this algorithm, given in [2], is to a priori add $\bar{s}$ to the query: $F_{n-1} \wedge \bar{s} \wedge T \wedge s'$. This improves the chances of blocking s in F_i and is sound because $F_0 \rightarrow \bar{s}$ and $F_i \rightarrow F_{i+1}$.

Once the query: $F_n \wedge T \wedge \bar{P}$ is unsatisfiable, a new frame F_{n+1} is created and the propagation phase begins. During this part of the algorithm, cubes learned in previous frames are attempted to be blocked in later frames. This is accomplished by repeatedly solving $F_i \wedge T \wedge \bar{c}'$ for each clause $c \in F_i$. If this formula is unsatisfiable, then c can be added to frame F_{i+1}. If at any point two frames become identical (contain the same clauses), then an invariant is proved. Because $F_i \equiv F_{i+1}$ and $F_i \rightarrow F_{i+1}$, any clause present in F_i can be added to all future frames, therefore the property will hold in all future frames because $F_i \rightarrow P$.

III. Extending Cubes To Gate Variables

A. General Concept

In the previous implementations of the algorithm, states are symbolically encoded as cubes of state variables [8], [2], [4]. However, as Figure 1 demonstrates, there may be a significant advantage to allowing cubes in terms of intermediate variables. This extension does not change the conceptual flow of the algorithm greatly, but some care does need to be taken when choosing which intermediate variables to allow.

Figure 2 shows the transition relation that is used in the standard implementation of PDR. The logic in the transition relation can be partitioned into two types of logic: logic that only contains state variables in its transitive fanin (shown in grey) and logic that contains both state variables and primary input variables in its transitive fanin (shown in white). Because the grey logic only contains state variables in its transitive fanin, it is unaffected by valuations of the primary input variables. Consider the gates that are on the boundary between the grey and white logic. We use the set $G = \{g_0, g_1, \ldots, g_{k-1}\}$ to denote these gate variables. Note that some state variables may fanout directly into "white logic" shown in Figure 2. In this case, we consider these state variables to be among the set of variables G.

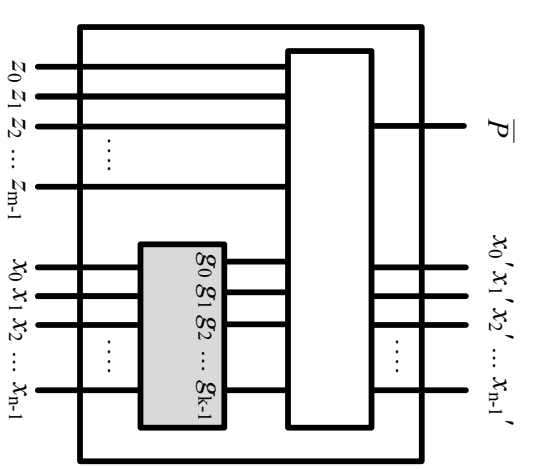


Fig. 2. A transition relation that is used in a frame for the standard PDR implementation

Proposition 1
Let S be the set of all cubes of state variables that can reach a cube m in one transition from frame i. Formally: $S = \{s \in \{0,1\}^X : s \models F_i \wedge T \wedge m'\}$. Let W be the set of all cubes of the variables in G that can reach m in one transition from frame i. Formally: $W = \{w \in \{0,1\}^G : w \models F_i \wedge T \wedge m'\}$. Then cube m can be blocked in Frame $i+1$ if and only if all cubes of S are blocked in frame i or all cubes of W are blocked in frame i. Formally: $(m' \not\models (\bigwedge_{s \in S} \bar{s}) \wedge F_i \wedge T) \leftrightarrow (m' \not\models (\bigwedge_{w \in W} \bar{w}) \wedge F_i \wedge T)$.

Proof:

The proof has been omitted due to page number requirements.

■

Proposition 1 allows us to pick a restricted set of gate variables to be used as state cubes. However, in order to block cubes of state variables, we must modify the transition relation shown in Figure 2 to be of the form shown in Figure 3. The transition relation in Figure 3 is a sort of *half unrolling* of the standard transition relation. The current state variables $(x_0, x_1, \ldots, x_{n-1})$ are replaced by outputs of the gate variables G in the current frame $(g_0, g_1, \ldots, g_{k-1})$. Also, the next state variables $(x'_0, x'_1, \ldots, x'_{n-1})$ extend to the outputs of the gate variables G in the next frame $(g'_0, g'_1, \ldots, g'_{k-1})$.

In the standard transition relation, cubes of variables of X are always extracted from the satisfying assignment. However, in our implementation we allow for cubes of variables of G. in order to block these cubes in the next time frame, the transition relation must be able to justify cubes of variables G'. As Figure 3 suggests, this does not increase the amount of logic in the transition relation, it only changes the placement of the logic.

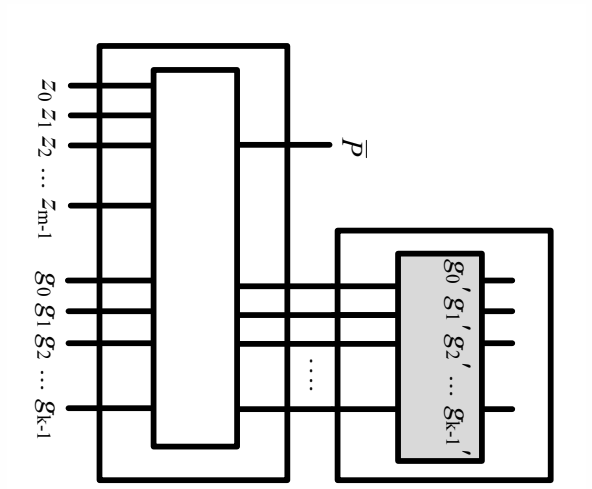


Fig. 3. A transition relation that is used for blocking cubes of gate variables

B. Details for Ternary Valued Simulation

The use of ternary valued simulation to reduce state cubes increases the performance of the algorithm tremendously [8]. The reduction of these state cubes is accomplished by first finding a satisfying assignment $a \in \{0,1\}^{X \cup Z}$ to the query $F_i \wedge T \wedge m'$. The transition relation shown in Figure 2 is then simulated with the values of a. A cube s of state variables in the transitive fanin of the variables of m' is chosen from this assignment. Then, one by one, the value of each state variable $x \in s$ is replaced by the unknown value $\perp$. Each time the value is replaced, the transition relation is re-simulated. If the values of m' remain the same through this simulation of the transition relation, then x is removed from the cube s, and the value of x remains at $\perp$ in the transition relation. If some value in m' changes, then x is restored to its original value, and x remains in s.

Ternary valued simulation can also be applied to cubes containing gate variables, but some care must be taken. The order in which the cube variables are set to $\perp$ can greatly affect the number of variables removed from the cube. To solve this problem we propose the following ternary simulation algorithm.

1) Sort the variables in s in ascending order by the variable's logic level (the maximum distance of the variable's corresponding gate from a primary input or latch output) in the transition relation and set $i = 0$.
2) If $i = |s|$, then return s. Otherwise, for the ith variable v_i in s do the following: If v_i is a state variable, proceed to step 4. Otherwise, proceed to step 3.
3) If the value of v_i can be determined to be 1 or 0 (not $\perp$) by its fanins, then remove v_i from s and go back to step 2. Otherwise proceed to step 4.
4) Set v_i to $\perp$ and simulate the transition relation. If no variable in m' evaluates to $\perp$, then remove v_i from s and proceed to step 2. Otherwise, set v_i back to its original value, re-simulate the transition relation, increment i, and proceed to step 2.

When only cubes of state variables are used, the value of each cube variable is independent of each other. In other words, the value of one state variable is never a function of another state variable. However, with cubes of gate variables this is not necessarily the case. By enumerating the cube variables in order by logic level, we can eliminate variables whose value is determined by other cube variables.

IV. Experiment and Results

A. Experiment Setup

To test our approach, we modified the version of PDR implemented in Berkeley ABC [8], [11]. We changed the ternary valued simulation portion of the implementation in the following way:

- The ternary valued simulation algorithm was run twice, once allowing cubes of gate variables in the cone-of-influence (COI) of the next state variables and once only allowing cubes of state variables in the COI of the next state variables.
- In the first pass, the cube containing gate variables was reduced. The order in which cube variables were set to $\perp$ was determined by the logic level of the gate variable (as discussed in Section III-B) and by a priority assigned to each variable. A variable's priority increased if it was unable to be removed from the cube; otherwise, it decreased. Variables with higher priority were given a greater chance of being removed in later rounds of ternary valued simulation.
- In the second pass, a cube containing only state variables was reduced. In this case, the order in which cube variables were set to $\perp$ was determined by a fixed ordering.
- At the end of both passes, whichever cube was smaller (containing fewer literals) was returned.

To compare the performance of allowing gate cubes, rather than just state cubes, we ran this version of the algorithm (which we refer to as the *gate cube* version) on the HWMCC '11 benchmarks [1]. We compared the results against a second implementation, which was the same, except only state cubes were allowed in both passes of the ternary valued simulation. We refer to this version of the implementation as the *state cube* version. This way both implementations were given two chances to reduce cubes during ternary valued simulation, but only one was allowed to return cubes of gate

variables. The runtime differences between the two versions should then have been more heavily influenced by types of cubes, rather than by the priority scheme that we introduced. The transition relation was converted into a CNF formula using the standard Tseitin transformation [13] (as opposed to a transformation using variable elimination like in [8], [7]). This was done for the sake of an easier implementation. There are no conceptual problems with implementing our solution with other CNF transformations.

All of the benchmarks were run on a 4-core Intel® Core™ i7-2600 CPU @ 3.40GHz with 8GB of RAM. Only one core was utilized for each benchmark.

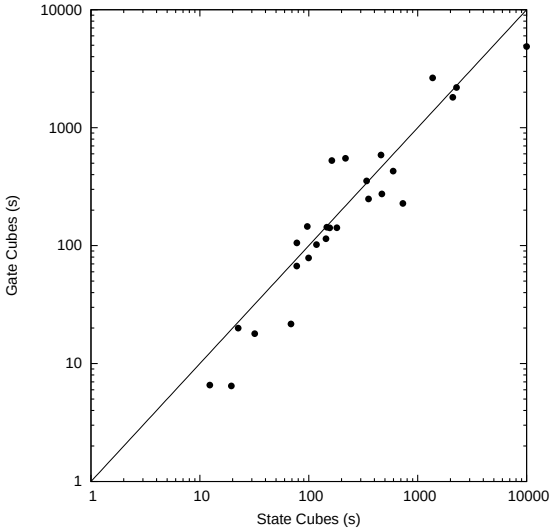


Fig. 4. Runtime comparison for satisfiable benchmarks.

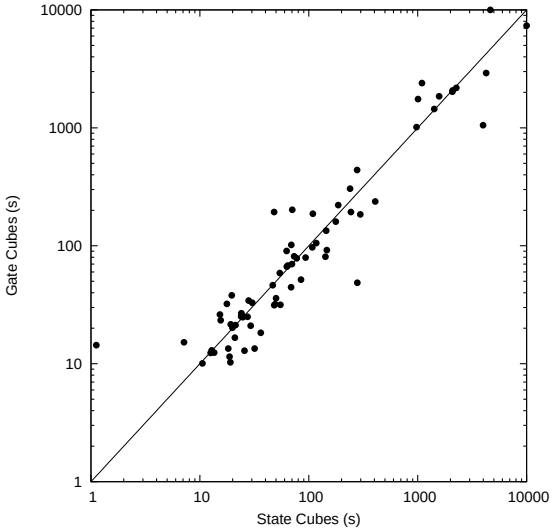


Fig. 5. Runtime comparison for unsatisfiable benchmarks.

B. Results

The results of our experiment are displayed in Figures 4 and 5. Figure 4 contains benchmarks where the property was proved not to hold (satisfiable benchmarks), and Figure 5 lists benchmarks where the property does hold (unsatisfiable benchmarks). We omitted benchmarks that took less than 10 seconds for both of the methods to solve. We set a timeout of 10000 seconds for all of the benchmarks, and we did not include results where both methods timed out.

The results demonstrate that for satisfiable benchmarks, allowing cubes of gate variables seems to have a generally positive effect on the performance of the algorithm. The best time ratio for the gate cube approach is .31 (which corresponds to a 3.2X runtime improvement for the benchmark `bc57sensorsp3`). The average time ratio for satisfiable benchmarks is .82 (which corresponds to an average speedup of 1.21X). For the unsatisfiable benchmarks, the performance is not as reliable. The best performance increase was seen by the benchmark `6s34`; which has a time ratio of .26 (a 3.85X speedup). For some of the benchmarks, the performance was close to the same. This likely indicates that both versions of the algorithm frequently chose cubes from the second pass of ternary simulation.

V. Discussion

The results demonstrate that allowing cubes of gate variables can cause very drastic performance changes between the benchmarks. In general we found that the gate cube version of PDR works better for satisfiable benchmarks, but the trend is not as clear for unsatisfiable benchmarks. The variation in the results indicates that perhaps there exists a better heuristic for choosing what logic to include in a certain cube. Regardless, the results show that the gate cube version of the algorithm does perform much better in many of the benchmarks.

Future work will focus on different heuristics for choosing what logic to include in each cube and different heuristics for cube minimization.

References

[1] A. Biere and K. Heljanko. Hardware Model Checking Competition (HWMCC) 2011 Benchmarks. Available at: http://fmv.jku.at/hwmcc11/, 2011.

[2] A. R. Bradley. SAT-based model checking without unrolling. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[4] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo. Incremental formal verification of hardware. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 135 –143, 2011.

[5] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV : a new symbolic model checker. *Software Tools for Technology Transfer*, 1998.

[6] W. Craig. Linear Reasoning: A New Form of the Herbrand-Gentzen Theorem. *Symbolic Logic*, 22(3):250–268, 1957.

[7] N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In *In Proceedings of SAT05*, pages 61–75. Springer, 2005.

[8] N. Eén, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 125 –134, 2011.

[9] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[10] K. L. McMillan. Interpolation and SAT-based model checking. In *International Conference on Computer-Aided Verification*, pages 1–13, 2003.

[11] A. Mishchenko et al. ABC: A system for sequential synthesis and verification, 2007.

[12] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pages 108–125, London, UK, UK, 2000. Springer-Verlag.

[13] G. S. Tseitin. *On the Complexity of Derivations in Propositional Calculus*. 1968.