Enhancing SAT-based Sequential Depth Computation by Pruning Search Space*

Yung-Chih Chen and Chun-Yao Wang
Department of Computer Science, National Tsing Hua University, HsinChu, Taiwan
{ycchen, wcyao}@cs.nthu.edu.tw

ABSTRACT

The sequential depth determines the completeness of bounded model checking in design verification. Recently, a SAT-based method is proposed to compute the sequential depth of a design by searching the state space. Unfortunately, it suffers from the search space explosion due to the exponential growth of design complexity. To alleviate the impact of state space explosion, we propose a search space reduction method. We collect the learned states and consider them constraints for further path searching. Furthermore, we propose a heuristic to guide the SAT-solver to efficiently find a shortest path. The experimental results show that as compared to another method which also enhances the previous SAT-based method using a branch-and-bound strategy, our approach obtains more improvements.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*verification*

General Terms

Algorithms, Verification

Keywords

Satisfiability(SAT), sequential depth

1. INTRODUCTION

Bounded Model Checking (BMC) [1] [2] [3] has become an important technique in modern design verification. The basic idea of BMC is to search an error violating a property within a bounded depth k in a finite state system. If no error is found, BMC repeats the search by incrementing k until either an error is found or a pre-computed bound is reached. This bound can be considered a threshold of the design which determines the completeness of BMC. This is because when BMC has explored the complete state space and no error is found, it ensures that certain property holds in the design. With the threshold, BMC can be used not only to find errors, but also to prove correctness.

Previously, many work have addressed the problem of computing the threshold for BMC. In [6], Sheeran et al. observe that no more iterations should be performed if no new states exist in future iterations. They present a SAT-based method to identify a path of length i starting from the initial state. If a path exists, a new state exists, and then they repeat to identify a longer path by incrementing i until no path exists. However, the computed value, i, is always an over-approximated solution and may be much larger than the threshold. On the other hand, in [7], Yen et al. propose a simulation-based method to estimate the threshold, but the method may result in over- or under-approximations.

Recently, Mneimneh et al. [4] propose a SAT-based method to identify the sequential depth. The proposed algorithm iteratively uses a SAT-solver to search the largest path of all shortest paths starting from an initial state to any other reachable states. The sequential depth of a design is the length of the identified largest path and it is the exact threshold of the design. Additionally, Mneimneh et al. formulate a shortest path identification as a logical inference problem for Quantified Boolean Formulas (QBFs). They simplify the QBFs by applying simple transformations to the circuit netlist and use a SAT-solver to check their satisfiability. However, since the method exhaustively searches a path and then checks whether it is the shortest, the method tends to spend much CPU time while staying at the same iteration especially for large circuits having enormous search space.

To prevent the problem, Yen and Jou [8] present an algorithm to improve [4] by applying an effective branch-and-bound strategy. The algorithm starts by performing an exhaustive search for a short period of time to calculate the initial solution. Then, based on the initial solution, it uses the branching method to divide the entire problem into many sub-problems, and the bounding method to avoid solving useless sub-problems. With this strategy, they may eliminate many unnecessary states and thus compute a more accurate sequential depth in large circuits.

To improve the exhaustive-search approach [4], we propose a method to prune search space. We collect the learned states once finding a path, and consider them constraints to prune certain non-shortest paths for further path searching. Since having a smaller search space, we can more efficiently find the sequential depth. Additionally, we propose a heuris-

*This work was supported in part by the National Science Council of R.O.C. under Grant NSC 97-2220-E-007-042 and NSC 97-2220-E-007-034.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GLSVLSI'09, May 10–12, 2009, Boston, Massachusetts, USA.
Copyright 2009 ACM 978-1-60558-522-2/09/05 ...$5.00.

tic to guide the SAT-solver to find a shortest path. The experimental results show that our method reports a more accurate sequential depth in most large circuits as compared to [4] and [8] within a run time limit of 3,600 seconds. The more accurate sequential depths we provide in the large circuits can be the valuable information for further research on sequential depth computation.

This paper is organized as follows. Section 2 reviews some preliminaries. Section 3 briefly describes the previous method in [4]. Section 4 presents the proposed method of pruning search space. Section 5 introduces the proposed heuristic of guiding the SAT-solver. Finally, results and conclusion are presented.

2. PRELIMINARIES

A synchronous sequential circuit can be modeled using a finite state machine (FSM) M. The state transition graph of an FSM M, STG(M), is a directed graph (V, E), where each vertex $v \in V$ corresponds to a state s in M, and each edge $e \in E$ between two vertices v^i and v^j corresponds to a transition from state s^i to s^j in M.

Given a directed graph (V, E). A *walk* of length k is a succession of k directed edges v^0v^1, v^1v^2, ..., $v^{k-1}v^k$. Let $Walk(k, v^0, v^k)$ denote a walk of length k starting at vertex v^0 and ending at vertex v^k. If the vertices in the directed edges of a walk are all different, the walk is called a *path*. Let $Path(k, v^0, v^k)$ denote a path of length k starting at vertex v^0 and ending at vertex v^k. A path of length k is *shortest* if no path of length j that has the same start-vertex and end-vertex exists for each $j < k$. In addition, the *distance* of two vertices in a path is the number of edges between them. Let $Dist(v^i, v^j, P)$ denote the distance between v^i and v^j in a path P.

Consider two vertices v^0 and v^k in a directed graph (V, E). v^k is *reachable* from v^0 if a path exists from v^0 to v^k. Given an FSM M with a single initial state s^0. The *sequential depth* of M is the length of the longest of all the shortest paths starting at s^0 and ending at any states reachable from s^0.

3. PREVIOUS WORK

In [4], Mneimneh et al. propose a SAT-based approach to sequential depth computation. Given an FSM M with an initial state s^0. The approach starts from $k = 0$. Then it iteratively searches a shortest path of length $k + 1$ starting at s^0 and increments k. The approach stops when it cannot find any shortest path of length $k + 1$. Finally, the length of the longest of all the shortest paths is k, and k is the sequential depth of M.

The approach uses a two-stage algorithm to identify a shortest path of length $k + 1$. At the first stage, it tries to identify a path of length $k + 1$. It constructs a *Conjunctive Normal Form* (CNF) formula to present a path of length $k + 1$ starting at s^0 and uses a SAT-solver to find a satisfiable assignment. If the CNF formula is satisfiable, a path, $Path(k + 1, s^0, s^{k+1})$, is found. Next, at the second stage, the approach checks whether the path is the shortest. It first modifies the given FSM M such that each state in M has a directed transition to s^0. That is each vertex in STG(M) has a directed edge to the vertex of s^0. Then it constructs a CNF formula to present a walk, $Walk(k, s^0, s = s^{k+1})$, and uses the SAT-solver to solve it. If the CNF formula is unsatisfiable, the path is the shortest. Otherwise, the path is non-shortest.

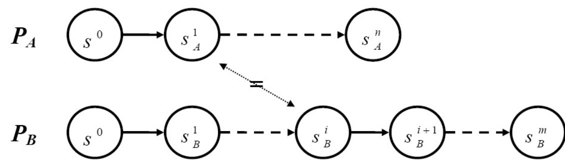

Figure 1: Example of pruning search space.

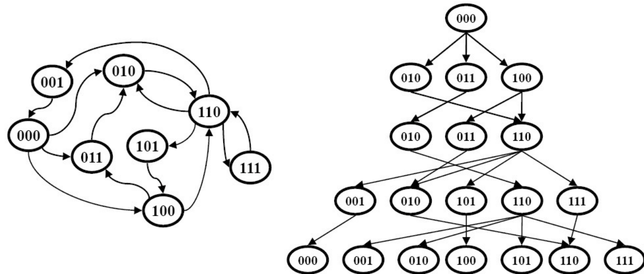

Figure 2: An example of sequential depth computation.

The procedure is as follows. The algorithm starts by reading the sequential circuit and setting the variable $k = 0$. Consider the iteration k of the algorithm, the algorithm first searches a path which satisfies $Path(k + 1, s^0, s)$. If no such path exists, the algorithm terminates and returns k as the sequential depth. If the path exists, the algorithm then checks if there exists a walk which satisfies $Walk(k, s^0, s)$. If no such walk exists, the algorithm has found a shortest path of length $k+1$. Then it increments k and continues the next iteration. If a walk exists, the path is not shortest and the algorithm adds the learned clause to the path formula to force the SAT-solver to find another path having a different end-state.

The drawback of this algorithm is that it may spend much run time to search a large amount of paths while they are not the shortest. As a result, the program execution stays at the same iteration and the k value remains unchanged for a long time. To deal with this drawback, we present a search space reduction method described in the following section.

4. SEARCH SPACE REDUCTION METHOD

Firstly, we use an example in Fig. 1 to demonstrate our basic idea of pruning search space. Fig. 1 shows two paths, P_A and P_B. P_A starts at an initial state s^0 with a length n. P_B also starts at s^0 but ends at a different state with a longer length m. In addition, the state s^1_A in P_A is identical to the state s^i_B in P_B. Let's consider the sequential depth computation using the algorithm mentioned in Section 3. The algorithm exhaustively searches a path and then check whether it is the shortest. When $k = (n - 1)$, P_A can be a solution while searching a path. Also, P_B can be a solution when $k = (m - 1)$. After identifying P_B, the algorithm will find that P_B is not a shortest path since there exists a shorter path also starting at s^0 and ending at s^m_B. Because s^1_A is identical to s^i_B, the shorter path can be $s^0 \rightarrow s^1_A \rightarrow s^{i+1}_B \cdots s^m_B$.

Our objective of search space reduction is to prune paths

that cannot be the shortest. In this example, P_B is obviously not a shortest path because there is a transition from s^0 to s_B^i and s_B^i is identical to s_A^1. As a result, we can exploit s_A^1 to prune P_B at the iteration $k = (m-1)$. More precisely, once we find a path, we can consider the states of the path constraints to prune certain non-shortest paths during further path searching. The following observation states the principle of the search space reduction method.

OBSERVATION 1. *Suppose s is a state in a path P and $Dist(s^0, s, P) = i$. According to s, each path P' having $Dist(s^0, s, P') > i$ cannot be a shortest path.*

Since the distance between s^0 and s in P is shorter than that in P', P' is not a shortest path. Let's review the algorithm mentioned in Section 3. The algorithm always identifies a path and then checks whether the path is the shortest. It may waste much run time while finding non-shortest paths. To improve it, once it finds a path, we can record the learned states and their distances from the initial state in the path. Then, we add them as constraints to the CNF formula to avoid the algorithm finding certain non-shortest paths during further path searching. These non-shortest paths are the paths that also have the learned states with longer distances.

The details of the search space reduction method are as follows: First, let's consider collecting the learned states. We construct a table to record the learned states and their distances from the initial state. When a path is found, we check if a state in the path has been recorded in the table. If not, we add it and its distance to the table. If it has been recorded and its new distance is shorter than the original one, we update its distance. Otherwise, the data of the table is intact. In addition, if the found path is not the shortest, we also decrement and update the distance of its end-state. Next, let's consider adding constraints to the CNF formula. Suppose s is a learned state and its distance from the initial state is i. When we want to find a path P' of length $k+1$, the added constraints based on s can be formulated as $\prod_{j=i+1}^{k+1}(s^j \neq s)$, where s^j is the jth state in P'. Based on this formulation, when we want to find a path, we will add all learned states in the table as constraints to the CNF formula.

Let's take the example in Fig. 2 to demonstrate the search space reduction method. Fig. 2 shows the STG of a sequential circuit, and its corresponding computation tree that results from unrolling the STG with an initial state 000. Consider integrating the algorithm mentioned in Section 3 and the search space reduction method. At the first iteration $k = 0$, we construct a CNF formula corresponding to $Path(1, 000, s^1)$ and use a SAT-solver to find a satisfiable assignment. One solution is $000 \rightarrow 010$. We then record the learned state 010 and its distance 1. The table includes (State, Distance) $= \{(010, 1)\}$. Since $000 \rightarrow 010$ is the shortest, we increment k and continue the next iteration.

At the iteration $k = 1$, we add the constraint $(s^2 \neq 010)$ and construct a CNF formula corresponding to $Path(2, 000, s^2) \cdot (s^2 \neq 010)$. One satisfying solution is $000 \rightarrow 100 \rightarrow 011$. Again, we record the learned states and their distances. The table becomes (State, Distance) $= \{(010, 1), (100, 1), (011, 2)\}$. Then, to check if $000 \rightarrow 100 \rightarrow 011$ is the shortest, we construct a CNF formula corresponding to $Walk(1, 000, 011)$. Since the formula is satisfiable, the path is not the shortest and then we update the distance of 011 by decrementing it to 1. The table becomes (State, Distance) $= \{(010, 1), (100, 1), (011, 1)\}$. Next, we add the constraints and construct a CNF formula corresponding to $Path(2, 000, s^2) \cdot (s^2 \neq 010) \cdot (s^2 \neq 100) \cdot (s^2 \neq 011)$. A satisfying solution is $000 \rightarrow 100 \rightarrow 110$. Since 100 has been recorded and its distance is the shortest, we only record the new learned state 110 and its distance 2. The table becomes (State, Distance) $= \{(010, 1), (100, 1), (011, 1), (110, 2)\}$. Furthermore, the formula corresponding to $Walk(1, 000, 110)$ is unsatisfiable. Thus, $000 \rightarrow 100 \rightarrow 110$ is the shortest. We then increment k and continue the next iteration.

Again, at the iteration $k = 2$, we add the constraints and construct a CNF formula corresponding to $Path(3, 000, s^3) \cdot (s^2 \neq 010) \cdot (s^3 \neq 010) \cdot (s^2 \neq 100) \cdot (s^3 \neq 100) \cdot (s^2 \neq 011) \cdot (s^3 \neq 011) \cdot (s^3 \neq 110)$. A satisfying solution is $000 \rightarrow 010 \rightarrow 110 \rightarrow 001$. Similarly, we record the new learned state 001 and its distance 3. Then we check whether there is a walk of length 2 ending at 001. The checking returns none. Thus, the path is the shortest. We then increment k and continue the next iteration.

Finally, at the iteration $k = 3$, we also add the constraints to the corresponding path formula. We find that the formula is unsatisfiable. Thus, the sequential depth is 3.

In conclusion, since we perform search space reduction, we can more efficiently identify the sequential depth than the original algorithm. For example, consider at the iteration $k = 3$. The original algorithm needs to iteratively find 5 paths (they are the paths ending at 001, 100, 101, 110 and 111, respectively) and check all of them are not the shortest. Then, it identifies the sequential depth of 3. However, by pruning the search space, we can directly determine that no shortest path exists and identify that the sequential depth is 3 as well.

5. HEURISTIC

Although pruning the search space can speed up the SAT-based sequential depth computation, its improvement is limited for large circuits having enormous search space. In this section, we present a heuristic for sequential depth computation of large circuits. With this heuristic, we can obtain more accurate sequential depths in large circuits, and they can be valuable information for further research on sequential depth computation.

Let's consider the process of SAT-based sequential depth computation. Suppose a circuit's sequential depth is n. We can divide the process into two parts: The first part is to search a shortest path during each iteration $k = 0 \sim (n-1)$. The second part is to identify that no shortest path exists at the iteration $k = n$. The spent run time at the first part depends on how fast we can find a shortest path. Thus, if we can make the SAT-solver find a shortest path faster, we can speed up the computation process as well. Based on this idea, we propose a heuristic to guide the SAT-solver. The following observation shows a necessary condition of a shortest path, and it is the basic of the heuristic.

OBSERVATION 2. *Suppose P is a path $s^0 \rightarrow s^1 \rightarrow \cdots \rightarrow s^{n-1} \rightarrow s^n$. The partial path of P, $s^0 \rightarrow s^1 \rightarrow \cdots \rightarrow s^{n-1}$, being a shortest path is a necessary condition for P to be a shortest path.*

According to Observation 2, when we use a SAT-solver to find a path of length n, if we can confine $s^0 \rightarrow s^1 \rightarrow$

Table 1: The improvements by pruning the search space and guiding the SAT-solver. The run time limit is 3,600 seconds.

Circuit	#FFs	Seq. Depth			
		[4]	Ours_P	Ours_PG	[8]
s1269	37	9	9	9	*
s1423	**74**	**20**	**22**	**81**	**37**
s3271	116	14	14	14	*
s3330	132	7	7	7	*
s3384	183	12	12	13	*
s4863	104	4	4	4	*
s5378	164	16	16	17	38
s6669	239	5	5	5	*
s9234	211	22	23	34	25
s13207	669	61	65	80	40
s15850	597	83	86	86	60
s35932	1728	19	19	66	17
s38417	1636	15	15	70	22
s38584	1452	44	44	46	62

$\cdots \rightarrow s^{n-1}$ to a shortest path, the SAT-solver has a higher possibility to return a shortest path. However, during the SAT-based sequential depth computation process, the shortest paths we know are only that found at each iteration. Thus, at each iteration k, when we want to find a path, $Path(k+1, s^0, s^{k+1})$, we first add constraints to confine $s^0 \rightarrow s^1 \rightarrow \cdots \rightarrow s^k$ to the shortest path found at the last iteration. Then, if the path is the shortest, we continue the next iteration. Otherwise, we remove the constraints and continue to find another path.

In summary, the heuristic exploits the previously found shortest path to find a path at the beginning of each iteration. Additionally, it is only used one time at each iteration. Once the found path is not the shortest, we remove the constraints and continue to find another path. The benefits of the heuristic are demonstrated in the experimental results.

6. EXPERIMENTAL RESULTS

The experiments are conducted over a set of ISCAS'89 benchmarks within SIS [5] environment on a Sun Blade 2500 workstation with 4GB main memory. We implement the proposed method in C language and use the MiniSat [9] as our SAT-solver. For comparison, we also reimplement [4] according to the algorithm mentioned in Section 3. The experimental results are summarized in Table 1.

Table 1 shows the improvements on large circuits by pruning the search space and guiding the SAT-solver. In this experiment, we set the run time limit to 3,600 seconds for each approach. Column 1 lists the benchmarks. All these benchmarks have enormous search space. Column 2 lists the number of FFs in each benchmark. Column 3 shows the sequential depth of each benchmark reported by [4]. Additionally, Columns 4 and 5 show the sequential depth reported by Ours_P and Ours_PG, respectively. Ours_P denotes our method of just pruning the search space. Ours_PG denotes our method of both pruning the search space and guiding the SAT-solver. For each benchmark, these three approaches all reach the run time limit. Column 6 shows the results reported in [8]. The work in [8] also set the same run time limit. "*" indicates that the result with respect to the benchmark is not reported in [8]. For example, $s1423$ benchmark has 74 FFs. Its sequential depth reported by [4], Ours_P, Ours_PG, and [8] within a run time limit of 3,600 seconds are 20, 22, 81 and 37, respectively.

According to Table 1, 4 out of 14 benchmarks' more accurate sequential depths are obtained by pruning the search space. They are $s1423$, $s9234$, $s13207$ and $s15850$ benchmarks. However, the improvements are limited. On the other hand, with simultaneously pruning the search space and guiding the SAT-solver, we can obtain a more accurate sequential depth for 9 benchmarks. Especially, the improvements are up to 61 steps for $s1423$ benchmark. Additionally, as compared to [8], we can obtain a more accurate sequential depth for most benchmarks, except for $s5378$ and $s38584$ benchmarks.

7. CONCLUSION

We propose an approach to speed up a SAT-based sequential depth computation algorithm [4] by pruning the search space. We collect the learned states, and then consider them constraints to prune certain non-shortest paths. Additionally, for large circuits having enormous search space, we also propose a heuristic to make the SAT-solver have a higher possibility to find a shortest path. The experimental results show that as compared with [4] and [8], we report a more accurate sequential depth for most benchmarks within a certain time limit. The reported more accurate sequential depths in the large circuits can be the valuable information for further research on sequential depth computation.

8. REFERENCES

[1] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking Using SAT Procedures instead of BDDs," in *Proc. Design Automation Conf.*, pp. 317-320, 1999.

[2] A. Biere, A. Cimatti, E. M. Clarke, O. Strichmanm and Y. Zhu, "Bounded Model Checking," *Advances in Computers*, vol. 58, 2003.

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Proc. Tools and Algorithms for the Construction and Analysis of Systems.*, number 1579 in LNCS, 1999.

[4] M. Mneimneh, and K. Sakallah, "SAT-based Sequential Depth Computation," in *Proc. Asia South Pacific Design Automation Conf.*, pp. 87-92, 2003.

[5] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Technical Report UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.

[6] M. Sheeran, S. Singh, and G. Stalmarck, "Checking Safety Properties Using Induction and a SAT-solver," in *Proc. Formal Methods in Computer-Aided Design*, pp. 108-125, 2000.

[7] C.-C. Yen, K.-C. Chen and J.-Y. Jou, "A Practical Approach to Cycle Bound Estimation for Property Checking," in *Proc. International Workshop on Logic Synthesis*, pp. 149-154, 2002.

[8] C.-C. Yen and J.-Y. Jou, "Enhancing Sequential Depth Computation with a Branch-and-Bound Algorithm," in *Proc. High-Level Design Validation and Test Workshop*, pp. 3-8, 2004.

[9] http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/