Static Property Checking Using ATPG v.s. BDD Techniques

Chung-Yang (Ric) Huang[1] Bwolen Yang[2] Huan-Chih Tsai[2] Kwang-Ting (Tim) Cheng[1]

[1]University of California, Santa Barbara
Santa Barbara, CA

[2]Verplex Systems, Inc.
San Jose, CA

Abstract

Static property checking is a technique for verifying some pre-defined design rules such as "bus contention", "racing condition", and "don't-care case". It contains formal verification engines so that a property can be completely verified and if the property is proven false, a counterexample will be generated for debugging the design. Among the different static property checking approaches, ATPG-based and BDD-based are the most powerful and successful ones. We implement both engines with several optimization techniques on the same framework such that their performance can be compared. The experimental results on some industrial designs show that both approaches have different strength and weakness in proving the static properties. Furthermore, our experience shows that they often complement each other and therefore a hybrid approach will be the best strategy. We combined these two approaches by alternating them with successively increased proof effort. The experimental results show that this combined approach do produce the best performance.

1. Introduction

Verifying functional correctness is becoming the single largest bottleneck in the design process for today's complex ICs. The objective of functional verification is to ensure that the design of an IC is consistent with its functional specification. To date, simulation remains the most popular method in debugging the functional design errors. The simulation-based functional verification process typically proceeds by creating a simulation driver which produces simulation stimuli for the RTL model and a simulation monitor which monitors the correctness of the outputs. Recent progress in simulation has primarily focused on either raising the level of abstraction of

simulation (e.g., from gate-level to RTL to behavior) or improving the speed of simulation (e.g., from event-driven to compiled-code or cycle-based to emulation-based). However, simulation-based verification has some fundamental drawbacks: (1) generation of good simulation stimuli remains a very time-consuming and tedious work, and (2) only exhaustive simulation can verify a property. As the design complexity continues to grow, it becomes increasingly expensive to have a high simulation coverage. Therefore, it is likely that simulation-based verification will miss some corner-case bugs even under long simulation cycles.

Recently significant research effort has gone into exploring ways to apply formal techniques to functional verification. Equivalence checking [1] and model checking [2] are among the more successful techniques of formal verification. Model checking, taking a model of the design and properties as inputs, determines whether or not the design satisfies the properties. If a property is proven to be not always satisfied, a counterexample is generated. There are two main tasks in model checking: (1) specification of desired property for verification, and (2) determination of whether or not the given design satisfies the specified properties.

The typical properties for verification can be classified into two categories: (1) safety properties and (2) liveness properties. Safety properties ensure that bad things never happen. For example, the design of a traffic-light controller must show a red light in at least one direction at all times. On the other hand, liveness properties guarantee good things happen eventually. For example, the traffic light controller must assure that a particular direction will eventually see the green lights. Temporal logics are typically used for expressing properties of a temporal nature. Temporal logics such as Linear Temporal Logic (LTL) [3] and Computation Tree Logic (CTL) [4] are commonly used for property specification. Property specification languages differ in their expressiveness. There are properties expressible in some languages, but not in other, less expressive, ones. Also, the complexity of the verification task depends on the choice of logic. In general, it is more difficult or computationally expensive to prove a property expressed in a more expressive language.

After expressing the properties for verification as temporal formulas, some search techniques are used to verify whether the design satisfy the formulas. Unlike simulation, the model checking approach searches exhaustively through all input combinations and the state space. Therefore, the correctness of the properties can be completely verified. The current model checking search methods can be classified into three categories: (1) explicit methods, (2) symbolic methods, and (3)

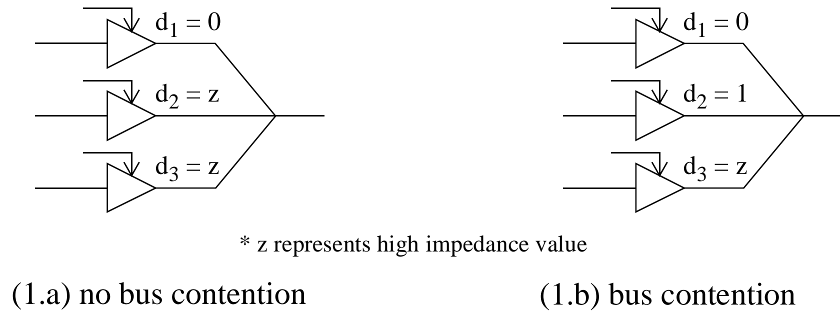
methods combining ATPG and analytical techniques. Explicit methods take the finite state model of the design and enumerate the states individually and explicitly. They often store the enumerated states in a hash table. The examples of this category are Cospan [by Kurshan] and Murϕ [5]. In general, this approach is relatively easier to use than the symbolic approach but the limitation is its poor scalability to larger design (it is limited by the number of states that it can enumerate). The symbolic methods represent a set of states using a data structure whose size is not proportional to the number of states. Therefore, they can handle more states than explicit methods. An example of this class of methods is SMV [6] which uses BDDs to represent sets of states. The third class of methods (e.g. [7][8]) use ATPG techniques integrated with analytical constraint-solving techniques with the goal of increasing the engine's capacity and performance.

Today's model checking solution encounters the following two major limitations: (1) The property specification language is hard to use and sometimes ambiguous. It is difficult to specify certain properties and also often difficult to interpret a complex temporal formula correctly. As a result, a property may be proven false due to the incorrect formulation of the property itself, not a design error. Therefore, it leads to the need of further validation of the correctness of temporal formulas with respect to the actual intention the specifier has in mind. (2) The search engine (either explicit or symbolic) often has limited capacity. The general modeling checking approach can usually handle sub-blocks in a circuit with less than 50K gates, which is 1 to 2 orders smaller than today's ASIC designs.

Static property checking aims at verifying a set of commonly encountered properties (called *static properties*) which can be extracted from the design automatically and thus need not be explicitly specified by property specification languages. This eliminates the ambiguity problem (caused by property language) mentioned above. Static properties are automatically extracted from the design based on some pre-defined design rules. These design rules are usually described in a simple natural language such as English, and are often described by a set of unambiguous templates in the CAD tools. For example, it is a common design rule that a design should not have multiple signals driving the same net with different values (Fig. 1). This bus-contention static property can be automatically generated and expressed in the following:

mutual-exclusivity (d_1, d_2,... d_n) is always true, where d_i i = 1,..., n is the ith driving source of the net.

Other examples of static properties that can be automatically extracted are "the clock signals of two different latches cannot be on at the same time if these two latches can form a combinational



(1.a) no bus contention (1.b) bus contention

Fig. 1 Bus contention property

path (racing condition)"[9], and "the default or unspecified cases in the Verilog *switch* statement (i.e. the don't-care condition) should not be on at any time",... etc.

Although the static property checking approach does not have the freedom to specify some general properties, it can cover a large class of properties which include most of the important design rules. In addition, because the static properties are usually the assertion type properties with less temporal recursion than the general ones, they can usually be verified for larger industrial designs. In short, the static property checking approach is easier to use, more scalable for larger designs, and can complement simulation to support the RTL signoff.

Without loss of generality, static properties can be checked by the same engines as those of the general model checking. In this paper, we discuss and compare two different approaches for checking the static properties: (1) Binary Decision Diagram (BDD)-based [2][10], and (2) Automatic Test Pattern Generation (ATPG)-based [9][11][12]. We implement both engines on the same framework containing a commercial RTL parser and logic synthesizer. Through experiments on real industrial designs, we found that both techniques have different strength and weakness in proving the static properties and most important of all, they can often complement with each other. We further combine these two approaches by alternating them with successively increased proof effort. The experimental result shows that this combined engine can have the best performance.

We organize the rest of the paper as follows: The basic concepts, strength, and weakness of the ATPG and BDD techniques are briefly addressed in Section 2. In Section 3, we compare the bit-level ATPG and BDD approaches by using the bus-contention property as the experimental driver. The main purpose of the comparison is not to draw conclusion on which technique is superior, but to investigate their strength and weakness for the application of static property checking.

In Section 4, we discuss a recently proposed method which combines word-level ATPG and modular arithmetic techniques [8] and demonstrate how it can help improve the capacity and performance of static property checking. Section 5 concludes this paper.

2. Background

Most of the static properties are assertion (safety) type properties. In other words, if satisfied, they assure that something bad never happens. To verify such properties, the proof engine needs to search through all input combinations and the entire state space in order to conclude that no witness sequence can trigger undesirable behaviors. In this section we discuss the basic concepts of two different techniques for this search process — BDD-based and ATPG-based techniques.

BDD is a compact and canonical data representation for Boolean functions. When BDDs are used to represent sets of states and the design's transition relation in the symbolic methods, the state space can be traversed implicitly through a sequence of operations on sets of states and state transitions, instead of one state and one transition at a time. However, for applications whose search goal is just to find one solution (such as test pattern generation for a stuck-at fault), the BDD-based symbolic technique may be an overkill. On the contrary, for applications which are likely to have no or very few counterexamples (such as assertion-type property checking), BDD-based approach seems to be a natural choice as it requires the traversal of the whole or most part of the search space to find a counterexample or to conclude that there is no counterexample.

The ATPG-based technique, on the other hand, uses branch-and-bound algorithms to search for the solutions. With good cost functions to guide the search and good decision-making ordering, it can be highly efficient for applications whose solutions are likely to exist. However, for proving an assertion property, which requires to examine all possible decision combinations, intuitively, ATPG may not be as efficient as the BDD-based technique.

However, such conclusion is over-simplified and may not even be correct. In many situations, ATPG-based approach works better than BDD-based approach for static property checking. In this paper, we carefully examine the strength and weakness of these two approaches and study various speed-up optimizations used by them for the applications of static property checking.

For example, suppose the target property to check is if the bus **out$BUS,** shown in Fig. 2.a, has bus contention. By constructing a *mutual-exclusivity* gate ($\mathbf{m_x}$) for the data sources $\mathbf{d_1}$, $\mathbf{d_2}$, and $\mathbf{d_3}$, checking the bus contention property can be transformed into the problem of checking

whether "$\mathbf{m_x} = 1$" (Fig. 2.b) is always true or not. The BDD-based approach basically construct the BDD for node $\overline{\mathbf{m}}_{\mathbf{x}}$ (Fig. 2.c). If there is no bus contention, then the BDD of $\overline{\mathbf{m}}_{\mathbf{x}}$ should be reducible to a constant "0" node. Otherwise each path from the BDD node of $\overline{\mathbf{m}}_{\mathbf{x}}$ to the terminal "1" node represents a vector causing bus contention.

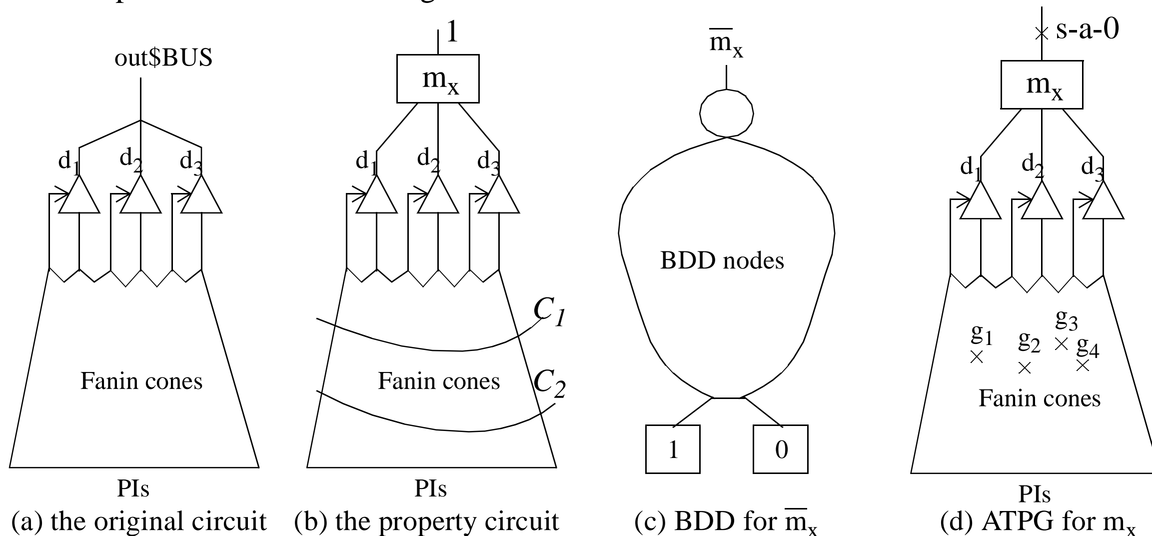


Fig. 2 Bus contention checking by BDD and ATPG

Even if the final BDD of $\overline{\mathbf{m}}_{\mathbf{x}}$ can be reduced to a constant node (when the static property is true) or to a small BDD (when only a small number of counterexamples exist), the explosion of the BDD size in the intermediate steps is still very likely to occur and thus causing memory explosion. Several techniques can be used to minimize the BDD size. For example, if we can construct the local BDD for $\overline{\mathbf{m}}_{\mathbf{x}}$ using a cut set C_1 as illustrated in Fig. 2.b (i.e. the BDD is built assuming the signals of the cut set, instead of the primary inputs, as input signals, or called supports, of the BDD). If the local BDD using this cut set can be reduced to a constant "0" node, it guarantees that the global BDD using primary inputs (PIs) as supports can also be reduced to a constant "0" node. On the other hand, if the local BDD is not a constant "0" node, we need to further verify whether the counterexample expressed in terms of C_1 can be produced by an vector at PIs. This can be determined by either reconstructing the BDDs or substituting the counterexamples into BDDs based on a new cut closer PIs (e.g. C_2 shown in Fig. 2(b)).

The "short-circuit" technique [13], another speed-up technique for the BDD-based approaches which incorporates the "best-first" search strategy, attempts to find a counterexample as early as possible. For example, to witness a "0" on an AND gate, we can heuristically choose a good input

to recursively construct its witness-0 BDD. If found, then we can ignore the remaining inputs. The cost of this approach is that we need to maintain the *care space*, i.e., the set of all valid witnesses. To illustrate this point, let us look at a multiplexer: **mux**($\mathbf{d_0}$, $\mathbf{d_1}$, **s**), where the output is equal to $\mathbf{d_0}$ when **s** is 0, and to $\mathbf{d_1}$ when s is 1. To witness a "0" on this gate under care space $\boldsymbol{C}$, we can recursively compute the witness-0 BDD for $\mathbf{d_0}$ with the new care space $\boldsymbol{C} \wedge (\mathbf{s} = 0)$. Similarly, we can compute the witness-0 BDD for $\mathbf{d_1}$ with the care space $\boldsymbol{C} \wedge (\mathbf{s} = 1)$. For this case, the extra cost incurred is in further restricting the care space $\boldsymbol{C}$.

Using ATPG for checking bus contention is equivalent to proving that the output stuck-at-1 (s-a-1) fault is undetectable (Fig 2.d). Most of the optimization techniques developed for test pattern generation of stuck-at faults can also be applied to static property checking. For example, indirect implication (also called "learning"), which tries to derive more logic implications between signals, can effectively reduce the search space and the possibility of making the wrong decision in the search process. On the other hand, when a conflicting assignment is encountered in the ATPG process, the "conflict analysis" technique [14] tries to identify the necessary condition that causes the conflicting assignment. Once identified, the necessary condition is transformed into logic implications which can then be used to guide the later decision-making process to prevent similar invalid decisions.

The most important factor which makes the ATPG-based approach comparable to the BDD-based approach for the applications of static property checking is that the static properties can often be proved by making decisions at the internal nodes. For example, in generating a test for the s-a-1 fault at $\mathbf{m_x}$, if all the decisions made at the internal gates $\mathbf{g_1}$, $\mathbf{g_2}$, $\mathbf{g_3}$, and $\mathbf{g_4}$ lead to conflicting value assignments (Fig. 2.d), we then can conclude that the s-a-1 fault is undetectable and thus the property is true.

This optimization technique is somewhat similar to the local BDD technique, which, as mentioned earlier, uses the internal cut set as new supports to construct the BDD for the property gate. However, in the local BDD technique since the gates on the cut set are treated as independent pseudo primary inputs, therefore, their correlations are completely ignored. On the contrary, because ATPG uses backward implications in addition to forward implications, the chosen decision points are not treated as independent signals. Therefore, the implications in the fanin of a decision point may lead to conflicting assignments at fanins of other decision gates. This can help

identify the invalid decision earlier and prevent the need to backward expanding the decision nodes (the cut set) as in the BDD approach.

In addition to the bit-level optimization techniques, ATPG-based approach can be easily extended to adopt the higher-level design abstraction. This is especially useful since the design methodology has moved from gate-level design entry to higher-level (e.g. RTL or behavior-level) hardware description language (HDL). The high-level (word-level) circuit information can then be used to derive more implications with less computational effort and to guide the decision-making process. Although BDD can also be combined with other word-level decision diagrams such as BMD[15], and HDD[16], ...etc., these decision diagrams are not compatible thus cannot be easily integrated together in a mixed word/bit level framework. Furthermore, ATPG can be easily integrated with other analytical techniques such as modular arithmetic or linear constraint solvers [8][17] in which the ATPG is used to solve the constraints on the control logic and the arithmetic, analytical techniques is for the datapath functions. In Section 4, we will discuss a recently proposed method which combines word-level ATPG and modular arithmetic techniques [8] and demonstrate how it can help improve the capacity and performance of static property checking.

In short, the BDD-based and ATPG-based approaches differ in the data representation, the searching strategies, the applicable optimization techniques and the characteristics of the run time and memory usage. As a consequence, they have different strengths and weakness in proving the static properties. Furthermore, we found that they can often complement each other and a hybrid approach combining them could be the best strategy. In our experiments, this hybrid strategy solves all our testcases without any abort.

3. ATPG v.s. BDD Techniques for Bus Contention Checking

We have implemented a bus contention checker using two different engines — one using ATPG and the other using BDD. In this section we discuss some performance factors on the runtime and memory usage.

In the ATPG engine, we try two different decision-making techniques: PODEM[18], which makes decisions on primary inputs only, and FAN[19], which allows decisions to be made at internal multiple-fanout gates. Other than this, we utilize the same optimization algorithms for both techniques as discussed in Section 2. The experimental results of using both decision-making techniques will be presented in Section 3.2.

In the BDD engine, we include the local BDD and short-circuit techniques. The effectiveness of using these optimizations will be presented in Section 3.3.

As discussed in Section 2, ATPG and BDD complement with each other in the proving capability. Therefore, we combine these two engines by alternating them and successively increasing the limit of proof effort after each iteration. The experiments show that this combined approach can produce the best results.

We conducted our experiments on six industrial circuits. Some statistics of these circuits are shown in Table 1. Among these circuits, two of them have a revised version (with extension ".r"), and another two of them have submodules declared as black boxes, where the black boxes come from undefined modules or irrelevant modules specified by the designers. We treat the black box inputs and outputs as pseudo primary outputs and inputs, respectively. Note that all of them have combinational loops. However, along the combinational loops there are several tri-state switches and therefore, these combinational loops may not be true loops (need to be checked by the racing condition checker). The number of bus contention properties is equal to the number of buses we checked for each design.

Table 1: Circuit Statistics

	#PI	#PO	#PIO	#Bus	#loop	#BBOX	#gate	#line	RTL/ Gate
ckt_1	31	2	18	93	121	0	1016	2632	Gate
ckt_2	37	9	23	308	528	0	4033	5829	RTL
ckt_3	168	171	134	515	284	1309	8463	31510	RTL
ckt_3.r	168	171	134	513	284	1309	8428	38722	Gate
ckt_4	170	205	0	284	270	0	9819	11408	RTL
ckt_4.r	170	205	0	284	273	0	14430	4028	Gate
ckt_5	61	25	0	152	96	0	25434	5720	Gate
ckt_6	163	149	0	7517	6469	61	193119	137873	RTL

- #PI, #PO, #PIO: number of primary inputs, outputs, and inouts.
- #Bus: number of buses.
- #loop: number of combinational simple loops.
- #BBOX: number of black boxes.
- #gate: total number of gates.
- #line: number of lines for the Verilog codes.
- #RTL/Gate: RTL or Gate level design

The experiments are performed on a Sun Untra-5 machine with 1 GB memory. The discussion and results will be shown in the following subsections.

3.1 Bus Contention Checking Using ATPG

PODEM and FAN are two most popular algorithms used in the traditional ATPG engines for the manufacturing fault (e.g. stuck-at fault) testing. Among the differences between these two algorithms, the most important one is that in PODEM, decisions are made at the primary inputs only, while in FAN, selected internal gates can also be used as decision points. Both algorithms have been successfully implemented in the commercial and academic tools and it still remains debatable about which algorithm is superior in the test pattern generation for the manufacturing faults.

For the application of static property checking, the results of the comparison of these two algorithms could be very different from those for the manufacturing fault test generation. Based on our implementation and the experiments on the real industrial circuits we tested, we found that for designs containing several submodules, it is usually the case that the static properties can be verified by examining the local modules only. As a result, FAN turns out to be the better algorithm for the static property checking. The rationale behind this can be illustrated by Fig. 3.

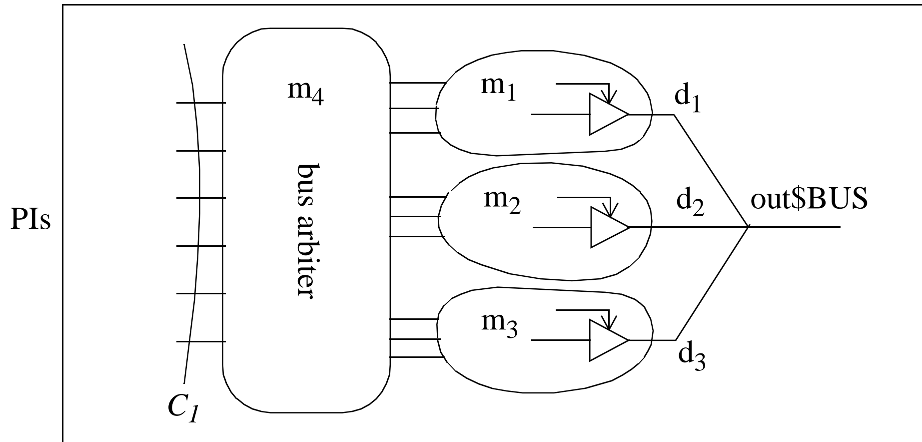


Fig. 3 A bus contention checking example

In this circuit, the bus signal **out$BUS** has 3 data inputs coming from different design modules ($\mathbf{m_1}$, $\mathbf{m_2}$, and $\mathbf{m_3}$), and these submodules have a common fanin module ($\mathbf{m_4}$) acting as the bus arbiter for these data inputs. The bus arbiter is responsible for generating the correct tri-state control signals and data input values such that it will not cause bus contention at any time. Therefore, if the bus-contention property is true (i.e. no bus contention), then making a decision at a signal on the cut C_1 will conflict with the implications of the other logic assignments on the cut. As mentioned in Section 2, the conflicting assignment can happen both between the cut and the property

gate, and between the cut and the PIs. The first occasion is due to the function of the bus arbiter, and the second one, which requires backward implications, shows that some input patterns of the bus arbiter are illegal due to their fanin cone logic.

We modified the FAN algorithm slightly for static property checking. Fig. 4 shows the modified FAN algorithm. Some key steps in this algorithm are explained in the following.

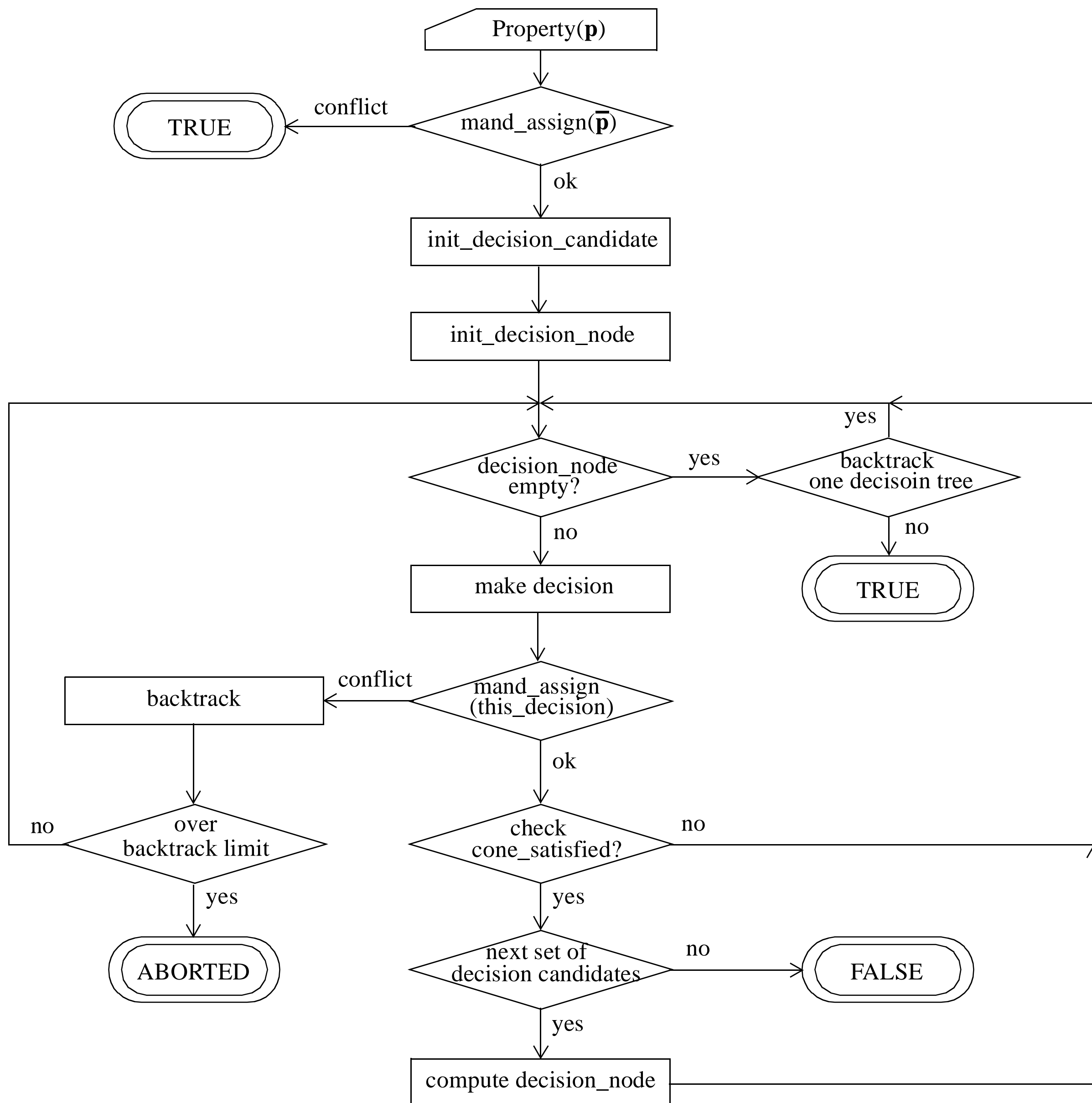


Fig. 4 The modified FAN algorithm for static property checking

(1) Selecting the decision candidates. Instead of finding the decision nodes one by one, we

select a set of *decision candidates* by backward traversing the fanins of the unjustified gates in a bread-first-search order and stopping at PIs, state elements, and multiple-fanout gates. These stopping gates are called the decision candidates.

(2) Minimizing the number of decision nodes. If the number of decision candidates are too large, using all of them as the decision points may make the decision-making process less efficient. Therefore, if the number of decision candidates exceeds a limit, based on the number of fanouts of each candidate, a subset of them is selected as the decision nodes.

(3) Determining the ordering of the decision nodes. The decision nodes are then ordered based on some estimates of signal probabilities of the nodes.

(4) Finding a next set of decision candidates and decision nodes. The ATPG justification process makes the decision one by one based on the ordered decision list. Once a new decision is made (i.e. assigning a new value at a decision node), logic implication is followed. If the logic implication results in conflicts, backtracking takes place. Otherwise ATPG continues until all the gates in the fanout cone of the decision nodes are fully justified. If some of the current decision nodes are not primary inputs, Steps (1) to (3) are repeated and a new set of decision candidates and decision nodes closer to the primary inputs are then selected. On the other hand, if all the current decision nodes are primary inputs, a counterexample to disprove the static property is found.

3.2 Experimental Results for ATPG-based Approaches - PODEM vs. FAN

In the experiments, we first compare the performance of using PODEM and FAN algorithms for bus-contention checking. The cpu time shown here is for checking all of the bus-contention properties in each design, and the memory usage includes the parsing and flattening of the circuit. The maximum number of backtracks is set to 1024. For properties that cannot be proved within this backtrack limit, we report them as aborted. Table 2 shows the experimental results.

In general, the FAN algorithm has much better performance than the PODEM algorithm. Among the 9 testcases, the PODEM-based approach only slightly outperforms the FAN-based approach in one case (ckt_5). The reason why PODEM is better for this particular testcase is that 120 out of its 152 bus contention properties are false. That is, there exist some counterexamples,

so making decisions directly at PIs can find the counterexample earlier.

Table 2: Bus contention checking using PODEM v.s. FAN

	PODEM						FAN					
	#Bus	**#T**	**#F**	**#A**	**time**	**mem**	**#Bus**	**#T**	**#F**	**#A**	**time**	**mem**
ckt_1	93	18	0	75	122	6.96	93	93	0	0	0.35	6.96
ckt_2	308	300	0	8	17.8	7.42	308	308	0	0	1.46	7.42
ckt_3	515	350	96	64	238	23.6	515	387	128	0	7.93	16.5
ckt_3.r	513	353	114	46	157	22.5	513	385	128	0	8.84	18.4
ckt_4	284	61	0	223	638	19.8	284	284	0	0	1.97	8.39
ckt_4.r	284	0	0	284	379	39.8	284	284	0	0	19.7	15.8
ckt_5	152	32	120	0	24.0	28.0	152	32	120	0	28.9	28.0
ckt_6	7517	3713	64	3740	17998	545	7517	6891	586	40	848	214

- #Bus: number of bus contention properties
- #T: number of true properties
- #F: number of false properties
- #A: number of aborted properties
- time: cpu time in seconds
- mem: memory usage in mega bytes

3.3 Bus Contention Checking Using BDDs

We implement the BDD engine with both local BDD (i.e. using the cuts) and short-circuit optimization techniques. In the first set of experiments we study the effectiveness of the short-circuit optimization. The proof effort for the BDD engine is limited to 3M BDD nodes and 5M BDD recursive operations.

The results in Table 3 show that our implementation of the short-circuit optimization is not always beneficial. This is partially because that in order to perform witness computation accurately, we also need to compute and maintain the care space (Section 2). The extra effort of computing the care space sometimes outweights the gain of short-circuiting computation. These results indicate that we may be able to further improve the overall performance by controlling effort spent in computing the care space.

We then study the effectiveness of the local BDD technique (with the short-circuit optimization turned on). The results (Table 4) show that the local BDD technique can significantly improve the perfomance of some testcases by reducing the number of aborted properties and the run time, particularly for circuits 4 and 6. The memory usage, on the other hand, remains about

the same.

Table 3: Short circuit optimization for BDD-based approach

		Without short circuit optimization					With short circuit optimization				
	#Bus	**#T**	**#F**	**#A**	**time**	**mem**	**#T**	**#F**	**#A**	**time**	**mem**
ckt_1	93	93	0	0	0.28	26.0	93	0	0	0.18	26.0
ckt_2	308	308	0	0	0.83	30.6	308	0	0	0.79	30.6
ckt_3	515	387	96	32	250	305	387	96	32	163	304
ckt_3.r	513	385	96	32	199	329	385	96	32	144	328
ckt_4	284	278	0	6	190	362	275	0	9	263	364
ckt_4.r	284	99	0	185	2081	443	90	0	189	1700	469
ckt_5	152	32	120	0	8.96	49.6	32	120	0	8.54	49.6
ckt_6	7517	6769	336	385	4370	671	6710	586	211	10652	573

- #Bus: number of bus contention properties
- #T: number of true properties
- #F: number of false properties
- #A: number of aborted properties
- time: cpu time in seconds
- mem: memory usage in mega bytes

Table 4: Bus contention checking using global v.s. local BDDs

		Global BDDs					Local BDDs				
	#Bus	**#T**	**#F**	**#A**	**time**	**mem**	**#T**	**#F**	**#A**	**time**	**mem**
ckt_1	93	93	0	0	0.18	26.0	93	0	0	0.74	26.0
ckt_2	308	308	0	0	0.79	30.6	308	0	0	2.82	30.7
ckt_3	515	387	96	32	163	304	387	96	32	298	310
ckt_3.r	513	385	96	32	144	328	385	96	32	336	333
ckt_4	284	275	0	9	263	364	284	0	0	2.53	31.0
ckt_4.r	284	90	0	189	1700	469	284	0	0	135	91.5
ckt_5	152	32	120	0	8.54	49.6	32	120	0	26.7	57.9
ckt_6	7517	6710	586	211	10652	573	6911	586	20	3899	586

- #Bus: number of bus contention properties
- #T: number of true properties
- #F: number of false properties
- #A: number of aborted properties
- time: cpu time in seconds
- mem: memory usage in mega bytes

3.4 Combined ATPG and BDD Approach

Based on previous experiments, we have found that ATPG and BDD can complement each other for bus-contention checking. For example, ATPG can outperform BDD for **ckt_3** both in run-time and memory usage, while BDD has less aborted properties for **ckt_6**. Therefore we com-

bine these two approaches by alternating these two engines and increasing the proof effort successively. We first try the ATPG engine (FAN algorithm) with very low effort (backtrack limit = 100), and if aborted, we continue for BDD (with local BDD and short-circuit optimizations) and ATPG again both with the same proof effort as in Section 3.3.

Table 5 shows the experimental results after the first iteration (i.e. fast ATPG then BDD then ATPG). For the purpose of comparison, we repeat the results for ATPG and BDD here. The combined approach solves all the properties except 14 for **ckt_6**. Its run-time, on the average, is better than the individual approach.

Table 5: Bus contention using ATPG, BDD, and combined approach

		Combined approach					**ATPG**			**BDD**		
	#Bus	**#T**	**#F**	**#A**	**time**	**mem**	**#A**	**time**	**mem**	**#A**	**time**	**mem**
ckt_1	93	93	0	0	0.29	26	0	0.35	6.96	0	0.74	26.0
ckt_2	308	308	0	0	1.28	30.6	0	1.46	7.42	0	2.82	30.7
ckt_3	515	387	128	0	8.21	38.8	0	7.93	16.5	32	298	310
ckt_3.r	513	385	128	0	7.37	40.6	0	8.84	18.4	32	336	333
ckt_4	284	284	0	0	2.06	30.9	0	1.97	8.39	0	2.53	31.0
ckt_4.r	284	284	0	0	19.5	52.1	0	19.7	15.8	0	135	91.5
ckt_5	152	32	120	0	30.3	50.2	0	28.9	28.0	0	26.7	57.9
ckt_6	7517	6917	586	14	928	671	40	848	214	20	3899	586

- #Bus: number of bus contention properties
- #F: number of false properties
- time: cpu time in seconds
- #T: number of true properties
- #A: number of aborted properties
- mem: memory usage in mega bytes

For the remaining 14 properties, we increase the proof effort for both ATPG (backtrack limit = 16386) and BDD (4M BDD nodes and 50M BDD recursive operations) engines. Using 64.6 seconds, we successfully prove these properties with the same peak memory usage.

In the next Section, we will present a recently proposed word-level ATPG technique and discuss its applicability to the static property checking.

4. Combined Word-level ATPG and Modular Arithmetic Techniques

Structural ATPG techniques can be easily extended to adopt the high-level abstraction information. Our recent work in [8] proposes a combined word-level ATPG and modular arithmetic constraint-solving techniques for model checking. In this Section, we briefly summarize this

method and then discuss its applicability to static property checking.

Fig. 5 shows the flow of this hybrid approach using word-level ATPG and modular arithmetic constraint-solving techniques. In the beginning of the flow a RTL netlist is read into the frame-

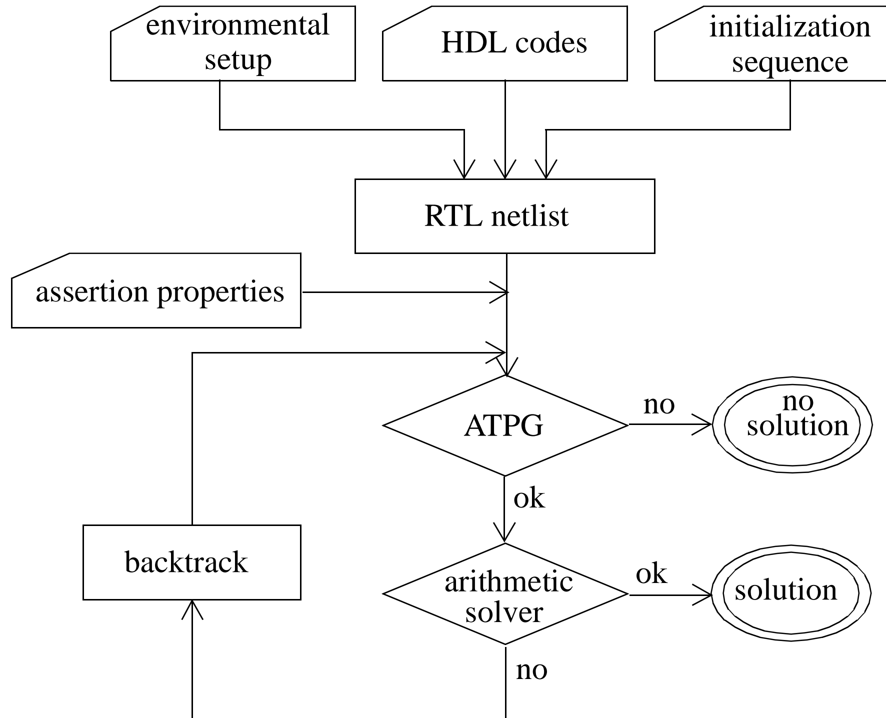


Fig. 5: The word-level assertion checking framework.

work and some heuristics are applied to partition the circuit into two parts: (1) control logics which contains bit-level logic gates, and (2) datapath which includes arithmetic and word-level logic units. The inputs to the control logics are bit-level PIs, state elements, and comparators with fanins coming from the datapath. On the other hand, the inputs of the datapath are word-level PIs (i.e. each PI is a word with multiple bits), state elements and multiplexors with control inputs coming from the control logics. Then we first apply the word-level ATPG to solve the constraints in the control logics. In the word-level ATPG approach, logic implications are not only applied to the Boolean gates, but also to the datapath units. In addition, an implication translation technique is proposed such that the implications learned on the datapath (control logics) can be translated as implications to the control units (datapath).

For example, suppose a 4-bit "greater" (>) gate has output value 1 (TRUE) and input values "in_a = 4'bx01x" and "in_b = 4'b1x0x" (Fig. 6). By setting all the x's to 0's and then to 1's, we

learn that "in_a" has the minimum and maximum values [min_a, max_a] equal to [2, 11], and "in_b" has [min_b, max_b] = [8, 13]. However, for the "greater" gate to be evaluated "TRUE", it implies "min_a" must be greater than "min_b", and "max_b" must be smaller than "max_a". Adjusting the values of "min_a" and "max_b", we have [min_a, max_a] = [9, 11] and [min_b, max_b] = [8, 10]. To map the new ranges back to 3-valued logic, we use the following rules:

(Rule 1) Only bits with value "x" can have new boolean implications.

(Rule 2) More significant bits must have implication prior to less significant ones.

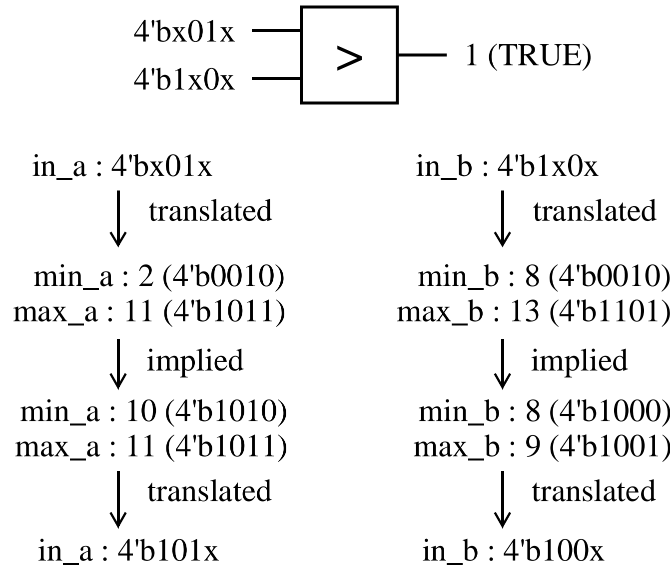


Fig. 6: Implication for comparators

While Rule 1 is trivial in logic implication, Rule 2 is based on the fact that only the most significant "x" bit can divide the original range into two disjoint sub-ranges. For this example, implication on the second highest bit of input **b** (with original value 4'b1x0x) can split the original range [8, 13] into two distinct sub-ranges [8, 9] (implied "0") and [12, 13] (implied "1"). On the other hand, implication on the least significant bit will produce two overlapped ranges [8, 12] and [9, 13]. Therefore, to have the new implied range [8, 10], it is mandatory that the second highest bit be implied "0" because this new range has null intersection with the other implied range [12, 13]. Implication on the least significant bit cannot draw any conclusion on this.

Likewise, we can learn that the most significant bit of "in_a" must have implied value "1". Therefore, we will have the new ranges for "in_a" and "in_b" equal to [10, 11] and [8, 9], respectively. Mapping these new ranges back to 3-valued logic, we will have implications "in_a =

4'b101x" and "in_b = 4'b100x".

Once the word-level ATPG finds a solution in terms of the inputs of the control logics, the constraints on the control logic are all solved and the only remaining constraints are in the datapath. We then apply the modular arithmetic constraint solver to solve the datapath constraints. Note that although the datapath constraints may across many timeframes, we can create a combinational model by treating the state elements (D flip-flops) as buffers and adding necessary new variables for the inputs of each timeframe. As a result, the datapath constraints can be represented by a set of arithmetic equations.

Arithmetic constraints can be divided into two types: linear and nonlinear. Nonlinear arithmetic constraints are those derived from multipliers and shifters. Since completely solving them could be very difficult, if not impossible, we apply some analytical approaches like prime number factoring to heuristically enumerate the possible solutions and substitute them into the arithmetic equations so that the constraints become linear and can be solved by our linear constraint solver.

Linear constraints, on the other hand, arise from adders, subtractors, and multipliers with one constant input. They make up most of the arithmetic units for industrial RTL circuits in various applications. Given a linear subcircuit with m outputs and n inputs, we can transform it to a problem of m linear equations with n variables and further formulate it into the matrix form as $\mathbf{A}*\mathbf{x} = \mathbf{b}$, where $\mathbf{A}$ is a $m*n$ matrix representing the coefficients in the m equations, $\mathbf{x}$ is a $n*1$ column matrix containing the n variables, and $\mathbf{b}$ is a $m*1$ column matrix for the output constraints. Solving the input vectors that can satisfy the output constraints is equal to finding the solution to the matrix equation.

The detailed constraint-solving algorithms for the linear equations can be found in [8]. In short, the solutions can be represented in a closed form as:

$$\mathbf{x} = \mathbf{N} * \mathbf{f} + \mathbf{x_0}$$

where $\mathbf{N}$ is called the *null matrix* (because multiplying it with the constraint matrix $\mathbf{A}$ will result in a zero matrix), $\mathbf{f}$ is a column matrix containing some free variables, and $\mathbf{x_0}$ is a solution and can be derived from $\mathbf{A}$, $\mathbf{N}$ and $\mathbf{b}$ in linear time. Applying different values of the free variables in $\mathbf{f}$, we can obtain different values of $\mathbf{x}$ and each of which is a solution of $\mathbf{A} * \mathbf{x} = \mathbf{b}$. The computational complexity of solving the linear constraints (finding all solutions) under the modular number system is $O(n^3)$, where n is the number of input variables.

Based on the solution found by ATPG for the control logic, if the arithmetic constraint solver finds a solution for the datapath, then a complete counterexample for the property under verification is found. Otherwise, backtracking will take place in the word-level ATPG to find the next solution for the control logic. Starting from the new solution for control logic, the arithmetic constraint solver is called again. This process repeats until all the solutions in the control units are exhausted or a counterexample is found.

We have conducted experiments using this combined word-level ATPG and arithmetic constraint solver for model checking on several industrial designs [8]. Currently, we are extending and optimizing it for static property checking. The experimental results of this hybrid approach will be presented in the final paper.

5. Conclusion

In this paper, we use the bus-contention checker as the experimental driver to examine two popular approaches, ATPG-based and BDD-based, in the static property checking. The experimental results show that they often complement each other for different testcases. Therefore, we propose a hybrid approach by alternating these two engines with successively proof effort. The experimental results show that this hybrid approach can have the best performance.

We also present our recently-proposed combined word-level ATPG and modular arithmetic techniques. This approach can improve the traditional bit-level ATPG by utilizing the high-level information and preventing the expansion of the arithmetic units. It also has the low memory usage advantage as the bit-level ATPG. We are now extending it to work with the BDD technique and optimizing it for the static property checking. The experimental results of this hybrid approach will be presented in the final paper.

We currently combine ATPG and BDD by alternating these two techniques. However, during the tool implementation we found that we can have even better results if we can have a tighter link for the engines. This is because in the search process, some parts of the search space are tough for the ATPG engine while other parts of the search space are difficult for the BDD engine. With a proper interface, a problem can be divided such that each engine can solve the correspondingly easier sub-problems. This will be our future work.

Reference

[1] S-Y Huang and K-T Cheng, "Formal Equivalence Checking and Design Debugging", Kluwer Academic Publishers, June 1998.

[2] K.L. McMillan, "Symbolic Model Checking: An Approach to the State Explosion Problem", Kluwer Academic Publishers, 1993.

[3] A. Pnueli, "A Temporal Logic of Concurrent Programs", Theoretical Computer Science, 13:45-60, 1981.

[4] E. M|. Clarke and E. A. Emerson, "Synthesis of Synchronization Skeletons for Branching Time Temporal Logic", Lecture Notes in Computer Science, Vol. 131, pp. 244-263, May 1981.

[5] "Murphi description language and verifier", http://verify.stanford.edu/cgi-bin/wrap/dill/Murphi.

[6] "The SMV System", http://www.cs.cmu.edu/~modelcheck/smv.html.

[7] R. C-Y Huang, and K-T. Cheng, "Solving Constraint Satisfiability Problem for Automatic Generation of Design Verification Vectors", Proc. IEEE International HIgh Level Design Validation and Test Workshop, Nov. 1999.

[8] R. C-Y Huang, and K-T. Cheng, "Assertion Checking by Combined Word-level ATPG and Modular Arithmetic Constraint-Solving Techniques", to appear in Proc. Design Automation Conference, June 2000.

[9] B. Keller, K. McCauley, J. Swenton, and J. Youngs, "ATPG in Practical and non-Traditional Applications", Proc. International Test Conference, pp. 632-640, Oct. 1998.

[10] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", IEEE Transactions on Computers, Vol. C-35, No. 8, pp. 677-691, August, 1986.

[11] P. Wohl and J. Waicukauski, "Using ATPG for Clock Rules Checking in Complex Scan Design", Proc. 15th IEEE VLSI Test Symposium, pp. 130-136, April 1997.

[12] K-T. Cheng and A. Kristic, "Current Directions in Automatic Test-Pattern Generation", Computer, vol. 32, no. 11, IEEE Computer Soc., Nov, 1999.

[13] Y-A. Chen and R. E. Bryant, "Verification of Floating-Point Adders", Proc. Computer Aided Verification. 10th International Conference, pp. 488-499, June 1998.

[14] J. P. M. Silva and K. A. Sakallah, "Dynamic Search-Space Pruning Techniques in Path Sen-

sitization", Proc. Design Automation Conference, pp. 705-711, June 1994.

[15] R. E. Bryant and Y-A. Chen, "Verification of Arithmetic Circuits with Binary Moment Diagrams", Proc. Design Automation Conference, pp. 535-541, June 1995.

[16] E. M. Clarke, M. Fujita and X. Zhao, "Hybrid Decision Diagrams Overcoming the limitations of MTBDDs and BMDs", Proc. International Conference on Computer-Aided Design, pp. 159-163, Nov. 1995.

[17] F. Fallah, S. Devadas, and K. Keutzer, "Functional Vector Generation for HDL Models Using Linear Programming and 3-Satisfiability", Proc. Design Automation Conference, pp. 528-533, June 1998.

[18] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuit", IEEE Trans. Computer, vol. C-31, pp. 215-222, 1981.

[19] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms", IEEE Trans. Computer, vol. C-31, pp. 1137-1144, 1983.