Property Learning Techniques for Efficient Generation of Directed Tests

Mingsong Chen, *Member*, *IEEE*, and Prabhat Mishra, *Senior Member*, *IEEE*

Abstract—Property falsification in model checking is widely used for automated generation of directed tests. Due to state space explosion problem, traditional model checking techniques cannot handle large scale designs. SAT-based bounded model checking is promising to address the prohibitively large time and resource requirements during the property falsification. This article presents several efficient learning techniques that can improve the overall test generation time for a single property as well as a cluster of similar properties. The goal is to exploit both variable assignments and common conflict clauses of the prechecked partial or similar SAT instances for property falsification. Our method makes three novel contributions: 1) investigates the decision ordering-based learnings for a single SAT instance; 2) applies the decision ordering learnings between similar SAT instances; and 3) exploits the relation between the decision ordering-based learning and conflict clauses-based learning. Our experimental results using both software and hardware benchmarks demonstrate that our approach can drastically reduce the overall test generation time.

Index Terms—Bounded model checking, directed test generation, conflict clause forwarding, decision ordering.

1 Introduction

Increasing complexity combined with decreasing time-to-market requirement makes the functional validation a major bottleneck in the hardware/software design flow. Simulation is the most widely used form of validation using random, constrained-random tests, and directed tests. Random and constrained-random testing [1] are easy to implement, nevertheless it is hard to guarantee the convergence to the testing target (i.e., functional coverage). In contrast, directed testing [2] uses fewer tests to obtain the required functional coverage, and therefore the overall validation effort can be significantly reduced. However, most directed test generation methods assume the expert knowledge of the *Design Under Validation* (DUV). These approaches can be laborious and error-prone due to the inevitable human intervention. Therefore, it is necessary to develop efficient techniques to automate the generation of directed tests.

Due to the ability of property falsification, model checking [6] is promising for automated generation of directed tests [5]. Fig. 1 outlines the traditional test generation method using model checking. The design specification is transformed to a formal model (e.g., SMV [6]), and the negated coverage requirements are transformed as safety temporal logic properties. During the verification of each property, model checkers exhaustively enumerate all the possible states. If one state contradicts the specified property, the model checker will report a counterexample. Such a counterexample is a sequence of variable assignments which can be used as a test to exercise the property.

To relieve the state space explosion problem when checking a large design, Boolean Satisfiability (SAT)-based Bounded Model Checking (BMC) [3], [4] is proposed. By unrolling the design and a property k times (k is the bound of the property), BMC converts the *k-step* state search problem into a Boolean SAT instance. If the property fails within k steps, a SAT solver will report a satisfiable assignment (i.e., a counterexample to falsify the property). This counterexample can be refined to a sequence of variable assignments, which can be used as a test to check the functional scenario specified by the property. Otherwise, if the property is true, the SAT instance is always false. Since this paper focuses on test generation, the generated SAT instances are assumed to be satisfiable.

In this paper, the primary objective of test generation is how to quickly get satisfiable assignments for a single SAT instance or a set of correlated SAT instances. Since Boolean satisfiability problem is a classical NP-Complete problem [7], it can be practically infeasible to achieve an optimal method. Therefore, various heuristic methods and tools [11], [12], [13], [14] are proposed to improve the SAT searching time. *Decision ordering* plays an important role during the SAT search because different decision orderings imply different decision trees as well as different search paths in the decision tree which strongly affect the search time. Existing decision ordering methods focus on exploiting the useful statistics from the checked SAT problems involving only a single SAT instance [31], [15], [16] or a series of incremental SAT instances of a single property [33]. However, none of them investigates the learning for decision ordering during the SAT search.

During test generation involving a complex design, it may have a large set of diverse properties to be checked. Usually, BMC-based methods will check each of them individually, which obviously is not time efficient. In fact,

- M. Chen is with the Software Engineering Institute, Room 211 (East), Math Building, Zhongshan North Campus, East China Normal University, Shanghai 200062, China. E-mail: mschen@sei.ecnu.edu.cn.
- P. Mishra is with the Department of Computer and Information Science and Engineering, University of Florida, PO Box 116120, Gainesville, FL 32611-6120. E-mail: prabhat@cise.ufl.edu.

Manuscript received 7 July 2010; revised 18 Nov. 2010; accepted 31 Jan. 2011; published online 18 Feb. 2011.
Recommended for acceptance by S. Shukla.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2010-07-0379.
Digital Object Identifier no. 10.1109/TC.2011.49.

0018-9340/11/$26.00 © 2011 IEEE Published by the IEEE Computer Society

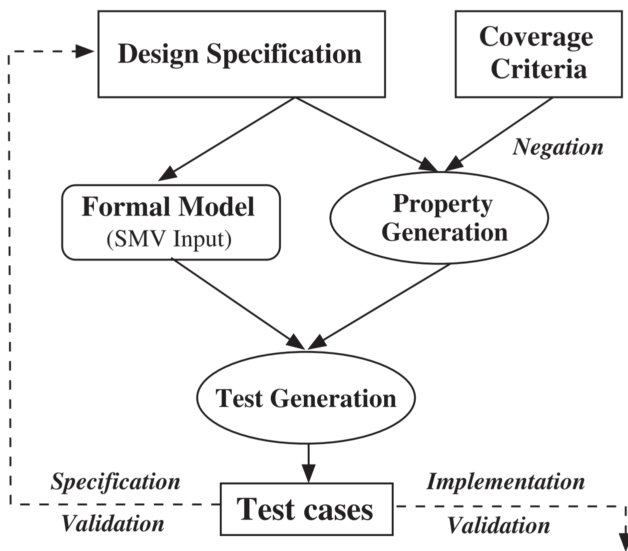

Fig. 1. Test generation using model checking.

for the same design, similar properties describe correlated functional scenarios. Therefore, the respective counterexamples are expected to have a significant overlap which can be exploited for sharing learnings. Furthermore, even for a single SAT instance, the result of the local search can also benefit the global search. Our method exploits the learnings of both conflict clauses [17], [12] and decision ordering in the context of test generation involving one or more properties of a design. Based on this observation, this paper makes three important contributions: 1) investigates the decision ordering-based learnings inside a single SAT instance; 2) applies the decision ordering-based learnings between similar SAT instances; and 3) exploits the relation between the decision ordering and conflict clause forwarding-based learning methods.

The rest of the article is organized as follows: Section 2 presents related work on SAT-based BMC, learning techniques and decision ordering heuristics. Section 3 introduces the background of SAT-based BMC as well as the implementation details of SAT solvers. Section 4 describes our learning techniques based on conflict clauses and decision ordering. Section 5 proposes our test generation methodology using efficient learning techniques. Section 6 presents the experimental results. Finally, Section 7 concludes the article.

2 Related Work

Functional validation is a major bottleneck in overall design methodology. In order to derive efficient directed tests to find design faults, various modeling and test generation methods are proposed [30]. Symbolic model checking techniques have been widely accepted as a promising method for automated generation of tests, and many powerful tools have been proposed for debugging complex designs [28], [29]. Due to the scalability issues of conventional Binary Decision Diagram (BDD)-based methods [18], SAT-based BMC is proposed as a complementary solution for large designs. Many studies [8], [9], [10] in both software and hardware domains show that BMC has better capacity and productivity over unbounded model checking for real designs. Currently, there are several techniques to improve test generation performance. Abstraction [19] is widely used to scale down the design state space. Based on counterexample guided abstraction refinement, Bjesse and Kukula [20] presented a method to generate *stepping stones* from abstract systems and divide the search into a number of short searches. Unlike abstraction, property decomposition proposed by Koo and Mishra [21] is another way to scale down the complexity. They proposed a method that can decompose a complex property into several equivalent subproperties. By combining the tests obtained from subproperties, this method generates the counterexample for the complex property. Both abstraction and decomposition techniques seems promising, however, it is hard to implement them automatically.

Sharing learning across properties can improve overall test generation performance since the repeated validation efforts can be avoided. Due to the incremental nature of BMC, SAT techniques [22], [23] try to exploit the commonality between SAT instances and reuse previously learned conflict clauses to prune current search tree. Strichman [23] found that when solving the SAT instance series of a property, some conflict clauses can be replicated and forwarded because of the symmetry of the transition part of the property. In [24], Chen and Mishra observed that during directed test generation using SAT-based BMC, similar properties can be clustered and solved together. For each cluster, they reused the learned knowledge of base (first) property to solve other properties. For a large design, there may be a large number of similar properties for test generation. They proposed several useful clustering criteria to cluster the properties and then share the learning (i.e., conflict clauses) among the properties in a cluster. Such clustering method can reduce the overall test generation time. However, the learning techniques based on the conflict clause forwarding need to calculate intersection between different SAT instances which may be time consuming. In addition, checking the first property in a cluster will remain a major bottleneck since there is no prior knowledge (learning). In this paper, we investigate both variable ordering and conflict clauses as learning knowledge to reduce the overall test generation time.

Different variable ordering will lead to different search trees, therefore a good branching heuristics can improve the SAT searching performance significantly [31]. In [15], Durairaj and Kalla proposed a promising method based on constraint partitioning on the hypergraph of a CNF-SAT instance. By analyzing the process of partitioning, the variable ordering can be derived to guide the CNF-SAT search. They also investigated the initial static variable ordering for efficient SAT search in [16]. They proposed a new metric to indicate the degree of correlation among pairs of variables. The variable activity and correlation information can be modeled as a weighted graph, and the variable ordering information can be derived by analyzing the topology of the graph. As a popular SAT solver, zChaff uses the Variable State Independent Decaying Sum (VSIDS) heuristic [12]. This heuristic contains two parts: 1) the static part collects the statistics of the Conjunctive Normal Form (CNF) literals prior to SAT solving and sets the initial decision ordering, and 2) during the SAT solving, the dynamic part periodically updates the priority based on conflict clauses. Although the above general-purpose heuristics are promising for propositional formulas, they

neglect some unique information of BMC. In [32], Strichman exploited the characteristics of the BMC formulas for a variety of optimizations including decision ordering. When the bound is unknown, SAT-based BMC needs to increase the unrolling depth one-by-one until finding a counter-example. Wang et al. [33] analyzed the correlation among different SAT instances of a property. They used the *unsatisfiable core* of previously checked SAT instances to guide the variable ordering for the current SAT instance. In [27], Zhang et al. studied the BMC-specific ordering strategies for SAT solvers. They proposed a method using clever orchestration of variable ordering and learned information in an incremental framework for BMC.

Our approach uses both decision ordering and conflict clauses to reduce the test generation time for a single SAT instance as well as for a cluster of similar SAT instances. The comparison between various learning techniques is provided in Section 6.

3 Preliminaries

This section first introduces SAT-based BMC. Next, we present two important issues during the SAT search: conflict clause forwarding and decision ordering.

3.1 SAT-Based BMC

SAT-based BMC is very promising to locate the errors and report the counterexample for a faulty property when bound is known a priori. Given a model M, a safety property p, and a bound k, BMC will unroll the model k times and transform the problem into a Boolean formula as follows:

$$BMC(M, p, k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i). \quad (1)$$

It consists of three parts: 1) $I(s_0)$ presents the system initial state, 2) $T(s_i, s_{i+1})$ describes the state transition from state s_i to state s_{i+1}, and 3) $p(s_i)$ tests whether property p is true on state s_i. Then this formula will be transformed to CNF and solved by SAT solvers. Semantically, if there is a satisfiable assignment for this property, then the property is false, written $M \not\models_k p$. Otherwise, it means that the property holds for the design within bound k, written $M \models_k p$.

Davis-Putnam-Logemann-Loveland (DPLL) algorithm [35] is widely used for SAT search. Algorithm 1 shows its implementation in zChaff. It contains three parts:

- Periodic function updates the SAT configuration triggered by some specified events, such as updating the scores of literals after a certain number of backtracks.
- Boolean Constraint Propagation (BCP) is implemented in *deduce*. It figures out all possible implications by previous decision assignment.
- Conflict analysis does a proper backtrack when encountering a conflict. It analyzes the reason for the conflict and make it as a conflict clause to avoid the same conflict in future processing.

Studies [12] show that modern SAT solvers spend approximately 80 percent of time to carry out BCP. In addition, during the conflict analysis, long distance back-tracks will increase the burden of SAT solvers. Our method tries to optimize both parts by using the learning based on

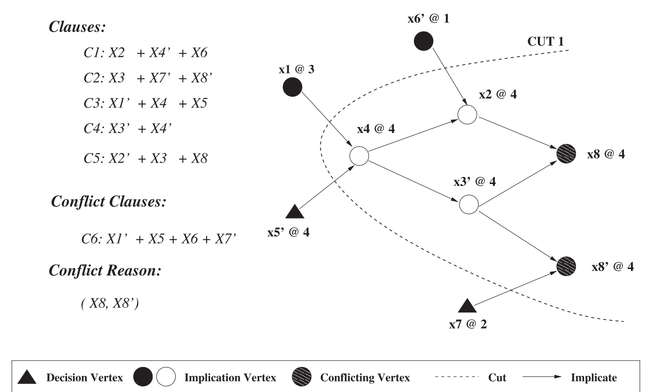

Fig. 2. Conflict analysis using an implication graph.

decision ordering. The learning can guide the SAT search so that it can drastically reduce the search time.

Algorithm 1: DPLL search procedure in zChaff

```
while TRUE do
    run_periodic_functions();
    if decide_next_branch() then
        while deduce() == CONFLICT do
            blevel = analyze_conflicts();
            if blevel < 0 then
                return UNSAT;
            end
        end
    else
        return SAT;
    end
end
```

3.2 Conflict Clause Forwarding

As shown in Algorithm 1, SAT solvers use the conflict analysis technique to trace the reason for a conflict. The conflict analysis contains two parts: *conflict-driven back-tracking* and *conflict-driven learning*. Conflict-driven back-tracking enables the nonchronological backtracking up to the closest decision which caused the conflict. Conflict-driven learning learns some knowledge and save them in *conflict clauses* and adds them to the original clauses, in order to avoid the same conflict in the future. Both techniques can drastically boost the performance of the SAT solvers.

The kernel of the conflict analysis technique is the implication graph [13], [17]. The graph keeps the current state and the implication history of the searching during the SAT solving by recording the dependence of the variable assignments. The implication graph is a directed acyclic graph where each vertex represents an assignment to a variable and each edge implies that all the in-edges implicate the assignment of the vertex.

Fig. 2 shows a small example of conflict analysis using an implication graph. As shown at the left of the figure, there are five original clauses $C1$-$C5$. The right part is a scenario of implication graph for $C1$-$C5$. In this example, $x4@4$ means variable $x4$ is assigned value 1 at decision level 4. The node has a corresponding clause $(x1' + x4 + x5)$, we call it the *antecedent* clause of $x4$, i.e., the assignments $x1 = 1$ and

$x5 = 0$ imply $x4 = 1$. Only the implication vertex (nondecision vertex) has an antecedent clause. A *conflict* happens when there are two nodes in the implication graph that have the different value assignments to the same variable. For example, the implications in the graph lead to the conflicting assignment to variable $X8$ ($X8 = 0$ and $X8 = 1$). When encountering a conflict, conflict analysis will trace back along the implication relations to find the reason for the conflict and encode the reason using a conflict clause. A conflict clause can be found by a bipartition of the implication graph. The side containing the conflicting vertex called *conflict side*, and the other side is called *reason side* which can be used to form the conflict clause. In Fig. 2, $CUT1$ is a cut that divides the implication graph into two parts. The conflict analysis stops at $CUT1$. The left part of $CUT1$ in the implication graph is the reason side, and the right part is the conflict side. From the reason side, we can get the *conflict cluase* $C6 = (X1 + X5' + X6' + X7)$. That means, the assignment of variables $X1 = 1$, $X5 = 0$, $X6 = 0$, and $X7 = 1$ will always lead to a conflict because of the clauses $C1$-$C5$. Lemma 3.1 indicates that the generated conflict clauses during the SAT search can be added to original clause set as an assignment constraint. Therefore, we can add the clause $C6$ to the original clause set to avoid the same conflict in the future.

Lemma 3.1. *Given a set of CNF clauses $S1$ and a conflict clause ψ (derived during the conflict analysis), $S1$ is satisfiable iff $S1 \bigwedge \psi$ is satisfiable.*

Proof. Because $S1 \bigwedge \psi$ is a superset of $S1$, so if $S1 \bigwedge \psi$ is satisfiable then $S1$ is satisfiable. According to the definition of the conflict clause, the assignments that make the clause ψ false will make the clause set $S1$ false. If $S1$ is satisfiable, then there exists a variable assignment that makes $S1$ true. This assignment should make ψ true. So the assignments will make $S1 \bigwedge \psi$ true. □

For two SAT instances, if one instance is a subset of the other SAT instance, according to Theorem 3.2, the conflict clauses generated from the smaller SAT instance can be forwarded to the larger SAT instance. In other words, the local learning can be forwarded as a knowledge for global searching. Usually the average cost of locally learned conflict clauses is much cheaper than the globally learned conflict clauses.

Theorem 3.2. *Given two CNF clause sets $S1$ and $S2$, where $S1 \subseteq S2$, and a conflict clause ψ derived from the clauses in $S1$, written $S1 \vdash \psi$, $S2$ is satisfiable iff $S2 \bigwedge \psi$ is satisfiable.*

Proof. Because $S2 \bigwedge \psi$ is a superset of $S2$, so if $S2 \bigwedge \psi$ is satisfiable then $S2$ is satisfiable. Because $S1 \vdash \psi$ and $S1 \subseteq S2$, then ψ is also a conflict clause of $S2$. According to Lemma 3.1, $S2$ is satisfiable iff $S2 \bigwedge \psi$ is satisfiable. □

According to the (1), similar properties will share a large part of the CNF clauses. Regardless of the cone of influence, the equation shares the system part (transition relation $T(s_i, s_{i+1})$) and the part of property testing (i.e., $p(s_i)$). Sharing a large part of CNF clauses indicates that when checking of the first property, the learned knowledge (conflict clauses) can be forwarded to the second property without affecting the truth assignments of the CNF clauses of the second property.

Theorem 3.3. *Given two set of CNF clauses $S1$ and $S2$, and let $\omega = S1 \bigcap S2$ be the common clauses shared by both $S1$ and $S2$. ψ is a conflict clause derived only by the clauses in ω, written $\omega \vdash \psi$. Then $S2$ is satisfiable iff $S2 \bigwedge \psi$ is satisfiable.*

Proof. Because $S2 \bigwedge \psi$ is a superset of $S2$, so $S2 \bigwedge \psi$ is satisfiable then $S2$ is satisfiable. Because $\omega \vdash \psi$ and $\omega \subseteq S2$, then $S2 \vdash \psi$. According to Lemma 3.1, $S2$ is satisfiable iff $S2 \bigwedge \psi$ is satisfiable. □

3.3 Decision Ordering

Decision ordering plays an important role during the SAT search. It indicates which variable will be selected first and which value (true or false) will be first assigned to this variable. Similar to BDD-based methods [18], variable ordering determines the performance of the SAT solving time. In the VSDIS heuristics implementation of zChaff, each literal l is associated with a *zchaff_score(l)* which is used for decision ordering at *decide_next_branch()*. Initially the score is equal to the literal count in corresponding CNF file. During the SAT solving, the score will be updated in the periodic function after a certain numbers of backtracks. The calculation of the new literal score is as follows:

$$chaff_score(l) = chaff_score(l)/2 + lits_in_new_confs(l), \quad (2)$$

where *lits_in_new_confs(l)* is the number of newly added conflict clauses which contain literal l since last update.

4 SAT-Based Learning Techniques

Similar properties usually have similar counterexamples which indicates that they may have similar Boolean constraints during the test generation. Consequently the generated SAT instances should have a large overlap in CNF clauses and can be clustered to share the learning. This section presents our learning heuristics which can be incorporated in the test generation approaches proposed in Section 5.

4.1 Overview

As discussed in Section 3.1, the most time-consuming parts are BCP and long distance backtracking. They are indicated by implication number and conflict clause number which represent the successful decision ratio and backtrack number, respectively. Ideally, a search method can get a satisfiable assignment by making the assignment for each variable only once. However, generally it is impossible to achieve such scenario. For a cluster of similar properties and predetermined bounds, the objective of our method is to reduce the number of implications and conflict clauses of unchecked properties by incorporating the learned decision ordering knowledge from previously checked properties.

Assuming that we have two similar properties, both properties will have a large overlap on CNF clauses and counterexample assignments. Fig. 3 shows the partial views of search trees and search paths of the two properties. The search paths are formed according to the decision ordering (shown on top of the search trees). For each variable v in the ordering, there are two literals (v means $v = 1$ and

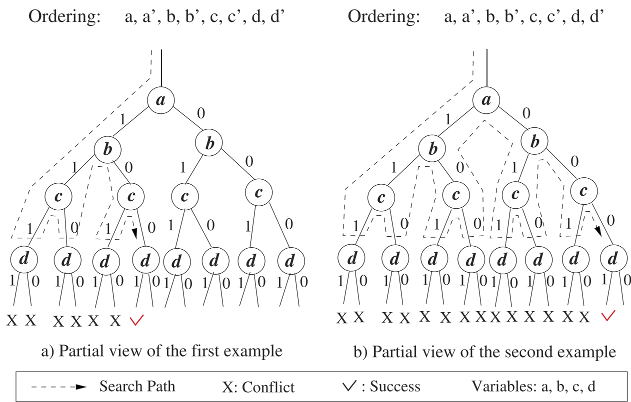

Fig. 3. Two examples of SAT search.

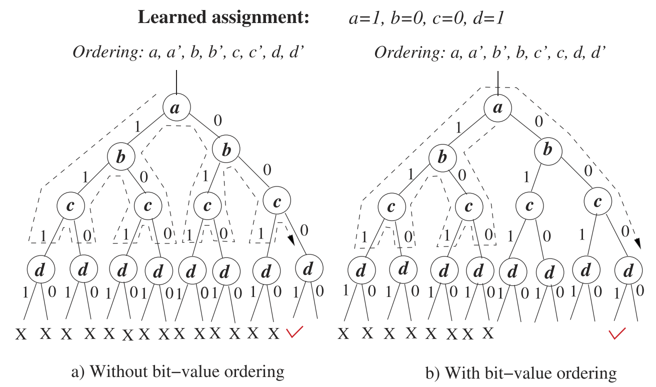

Fig. 4. A scenario where bit-value ordering works.

v' means $v = 0$). As shown in Fig. 3a, there are six conflicts encountered. The search stops after finding a satisfiable assignment $a = 1$, $b = 0$, $c = 0$, and $d = 1$ in this scenario. In Fig. 3b, the search will be successful only when $a = 0$, $b = 0$, $c = 0$, and $d = 1$ after encountering 14 conflicts. Therefore, the search of the second example will be more time consuming because of more backtracks.

Because of the large overlap in the assignment of counterexamples, the result of previously checked properties can be used as a learning for unchecked properties. For example, in Fig. 3, the result of first example strongly indicates the assignment of the second example because of the satisfiable assignment intersection $b = 0$, $c = 0$, and $d = 1$. If the second example uses the decision ordering based on the variable assignments in the first example, the searching time of the second example can be drastically reduced as shown in Fig. 6.

4.2 Decision Ordering-Based Learning (DOL)

Decision ordering involves two questions: 1) Given a Boolean variable, which value (true or false) will be chosen first? and 2) For a set of Boolean variables, which one will be determined first? This section presents how to utilize the decision ordering-based learnings to guide the SAT search.

4.2.1 Bit-Value Ordering

Similar properties generally have a large intersection of both corresponding CNF clauses and counterexample assignments. This indicates that the satisfiable assignment of checked SAT instances contain rich decision ordering knowledge for unchecked SAT instance. In SAT search, incorrect value selection for each variable will cause conflicts which will result in backtracks to remove the reason of the conflicts. A good decision ordering can mostly avoid such faulty assignments. Unlike pruning the search tree using conflict clause forwarding [24], bit-value ordering changes the *search path*. By setting the *bit priority* (choose 0 or 1 first) for each variable using the knowledge of previous property checking, the length of the search path can be reduced.

Fig. 4 shows an example where bit-value ordering works. As shown in Fig. 3a, we can get a satisfiable assignment $a = 1$, $b = 0$, $c = 0$, and $d = 1$. This assignment can be used to change the bit-value ordering of the second example. That means, when node b is encountered, the search chooses $b = 0$ first in its search path. The same rule also applies on other nodes. Applying such heuristics in Fig. 4b, there are only eight conflicts encountered compared to 14 conflicts in Fig. 4a. In addition, the search path is also shortened. Therefore, the searching time is reduced.

It is important to note that the bit-value ordering itself is not always helpful for the SAT searching. For example in Fig. 5, $a = 1$, $b = 1$, $c = 0$, and $d = 1$ is the only satisfiable assignment in the given scenario. The searching in Fig. 5a without bit-value ordering is faster than the searching in Fig. 5b because of less conflicts. If the learning assignment in Fig. 5 was $\mathrm{a} = 0$, $\mathrm{b} = 1$, $\mathrm{c} = 0$, and $\mathrm{d} = 1$, the searching performance will be much worse than the search in Fig. 5b. Clearly, in the search tree, the high-level variables (e.g., node a) strongly affect the performance of the searching if they are not consistent with learned bit-value ordering.

4.2.2 Variable Ordering

Although bit-value ordering is promising in general, there are still a lot of conflicts encountered during the search. According to the example shown in Fig. 5, if high-level nodes (e.g., node a) make the wrong decision, the search path will be lengthened due to the long distance backtrack. To reduce the searching time, it is necessary to restrict the conflict detection and reasoning in a small area.

Efficient combination of variable ordering and bit-value ordering is very promising. As shown in Fig. 6b, the search time is better than that in Fig. 6a due to a shorter search path and less conflicts. The reason of this improvement is that we enhance the priority of variables b and c. Since a is the

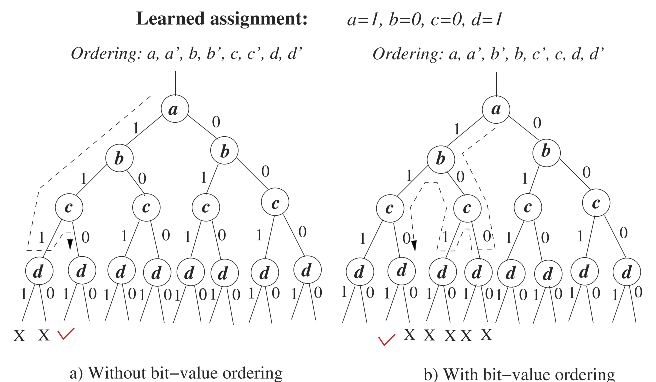

Fig. 5. A scenario where bit-value ordering fails.

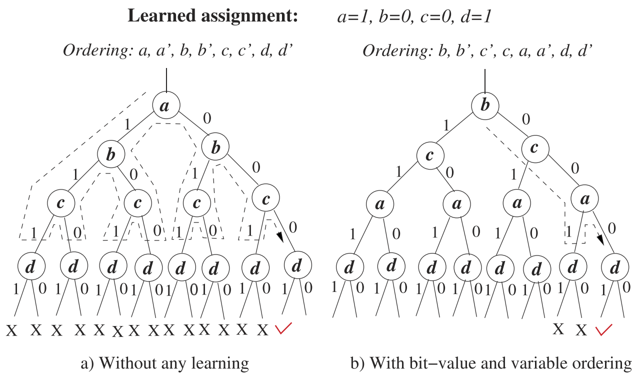

Fig. 6. An example of bit-value and variable ordering.

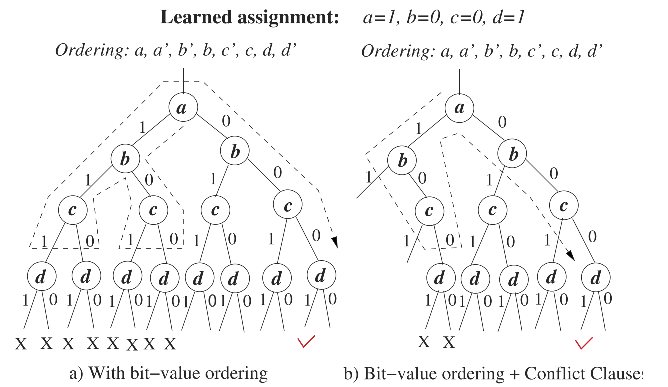

Fig. 7. An example using hybrid learning.

variable with different values between the two satisfiable assignments shown in Fig. 3, lowering down the priority of such variables (ones with different values between two CNFs) can efficiently avoid the long distance backtrack. Generally, before SAT solving, it is hard to figure out the difference between two satisfiable CNF variable assignments. However, based on the value assignment statistics of the checked properties, the variable ordering can be constructed. For a variable with the lower assignment value variation, which indicates high chance of same value, we will enhance its priority by increasing the score of its two literals.

4.3 Hybrid Learning from Conflict Clauses and Decision Ordering

Conflict clause is promising to avoid repeated conflicts during the SAT searching. Therefore, it is widely used as a learning during the test generation [24]. In essence, conflict clause forwarding can be used to prune the decision tree and can be utilized as a complementary approach for the decision ordering techniques proposed in Sections 4.2.1 and 4.2.2. For two similar SAT instances, if the conflict clauses of the checked SAT instance can be forwarded to the unchecked one, it will reduce the conflicts, thus further shorten the search path.

Fig. 7a shows application of bit-value ordering on the example shown in Fig. 3b. There are eight conflicts during the SAT search in this case. Let's assume the conflict clauses generated from Fig. 3a can be forwarded to the CNF clauses of Fig. 3b. The generated six conflict clauses are as follows:

$$\left.\begin{array}{c}(a' \vee b' \vee c' \vee d')\\ (a' \vee b' \vee c' \vee d)\\ (a' \vee b' \vee c \vee d')\\ (a' \vee b' \vee c \vee d)\end{array}\right\} \Rightarrow (a' \vee b'), \tag{3}$$

$$\left.\begin{array}{c}(a' \vee b \vee c' \vee d')\\ (a' \vee b \vee c' \vee d)\end{array}\right\} \Rightarrow (a' \vee b \vee c').$$

Equation 3 shows the resolution of the forwarded conflict clauses. Based on the result, we can prune the search tree as shown in Fig. 7b. It indicates that there are only two conflicts by applying the bit-value ordering on the pruned search tree. Therefore, the test generation time can be significantly reduced. For the example shown in Fig. 6b, the conflict clause forwarding is not beneficial since the search does not traverse the pruned part of the decision tree. Generally, the conflict clause forwarding can further improve the performance of the decision ordering based methods.

5 Test Generation Using Learning Techniques

For model checking-based test generation, each property is a negation of a desired system behavior. Consequently each property can produce a counterexample (test). Since our method adopts SAT-based BMC, in this paper, we assume that the bound can be predetermined and the generated SAT instances are satisfiable. Determination of bound is hard in general. However, for directed test generation, the bound can be estimated by exploiting the structure of the design. That means we can make sure the generated SAT instance for the specified property is satisfiable. The process of test generation for the property with a known bound is to figure out a satisfiable assignment for this SAT instance.

To reduce the overall test generation effort, this section utilizes the heuristics proposed in Section 4 as learnings. Section 5.1 applies learnings for test generation of a single property. In Section 5.2, we present an algorithm which shares the learnings among a cluster of similar properties.

5.1 Test Generation for a Single Property

When checking the first (base) property using property clustering techniques, or when checking only a single property, current methods solve the SAT instance alone since there is no source of learning. Therefore, it is time consuming and it can be a major bottleneck of the clustering-based test generation approach [24].

During test generation, if the bound of a property is increased by one, the test generation time will be drastically increased. Based on the observation of [32], the reason of time-consuming search is due to the long distance backtracking. Since large set of clauses that belong to different distant cycles are being satisfied independently (locally), [32] found that there are three typical scenarios which can cause the conflicts:

- Distant cycles are being satisfied independently until they collide each other with assignment conflict.
- Some cycle assignment collides with the constraints imposed by the initial state.
- Some cycle assignment collides with the constraints imposed by the negation of the specified property.

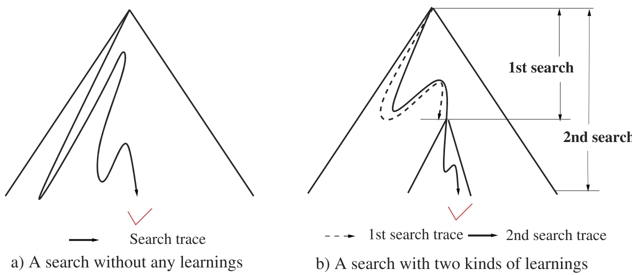

Fig. 8. Learning techniques for a single property.

The resolution of such conflicts needs to cancel large number of variable assignments between the conflicting cycles. Especially for the SAT instance with large bound, the cost of nonchronological backtracking is still huge since large bound indicates huge number of clauses and variables.

To alleviate long distance backtrackings during test generation, learning is required to guide the SAT search. Conflict clause is a promising learning that can prune the decision tree. However, in a SAT instance with large bound, the cost of deriving conflict clauses is costly due to large interleaving of irrelevant variables. Furthermore, a large set of CNF clauses is likely to generate a large number of conflict clauses which can affect the search performance. Therefore, if we can get conflict clauses from a smaller SAT instance, the average cost of conflict clause generation will be reduced. As an alternative, decision ordering can be used as learning. Since the SAT instance is assumed to be satisfiable, each segment[1] of the CNF clauses should be satisfiable. The searching time for a segment is much shorter than the original SAT instance. Although a segment cannot reflect the global view of the system, if the satisfiable assignment of the segment is consistent to the partial variable assignment of the original SAT instance, it will reduce the overall test generation time of the original SAT instance.

5.1.1 Heuristic Implementation

The basic idea of our heuristic for test generation involving a single property is to use the learning from a small part of the SAT instance to guide the search of the whole SAT instance. By dividing the SAT instance into two segments, we can get the first segment which contains the initial state constraints and the second segment which contains property constraints. After checking any one of them, we can get the partial variable assignments which can be used as decision ordering learning, and we can get the conflict clauses which can be forwarded to the original property according to Theorem 3.3. Fig. 8 demonstrates an example of using such learnings. In Fig. 8b, we first check one part of the SAT instance and get the corresponding learnings. Then during the checking of whole SAT instance, under the guidance of the learned knowledge, the overall search path is shortened compared to Fig. 8a.

Our decision ordering heuristics implementation uses an array $var[sz]$ (sz is the largest variable number for CNFs) to indicate the satisfiable assignment result of the first search. Each element of the array $var[i]$ $(0 < i \leq sz)$ has three values: 1 means that the ith variable is assigned with 1; 0 means that the ith variable is assigned with 0; and -1 implies that the variable is not assigned during the first search. So during the second search, the literal score is calculated using the following equation where $max(v_i) = MAX(chaff_score(v_i), chaff_score(v_i')) + 1$:

$$score(l_i) = \begin{cases} max(v_i), & (var[i] == 1 \quad \& \quad l_i = v_i), \\ & or(var[i] == 0 \quad \& \quad l_i = v_i'), \\ chaff_score(v_i), & otherwise. \end{cases} \tag{4}$$

1. A CNF-SAT instance can be viewed as a union of a set of segments where each segment consists of a set of CNF clauses.

5.1.2 Test Generation Using Intraproperty Learning

Algorithm 2 describes our test generation procedure for a single property using learnings from some part of the SAT instance corresponding to the original property. Step 1 initializes all the elements of var with -1. Step 2 generates the CNF clauses for the property p. After dividing the CNF in step 3, step 4 solves the clauses in any one part and derives the learning in the form of decision ordering and conflict clauses. Step 5 updates the var. Finally, step 6 uses the learning to guide the test generation of the original property. In this paper, for intraproperty learning, we divide a SAT instance into two segments with the same number of clauses (when the total clause number is odd, the difference of clause number between segments is 1).

Algorithm 2. Test Generation for a Single Property
Input: i) Formal model of the design, D
ii) Property p with bound b
Output: A test t for p with generated conflict clauses
1. Initialize var;
2. $CNF = BMC(D,\ p,\ b)$;
3. Divide CNF into CNF_1 and CNF_2;
4. $(assign, conf_cls1) = \mathrm{SAT}(CNF_1$ or $CNF_2, var, \mathrm{NULL})$;
5. Update var using $assign$;
6. $(t, conf_cls2) = \mathrm{SAT}(CNF, var, conf_cls1)$;
return $(t, conf_cls1 + conf_cls2)$;

It is important to note that our heuristic for a single property is based on the assumption that the decision ordering knowledge learned from the first search has a large overlap with a satisfiable assignment of the second search. Although the forwarded conflict clauses can prune the decision space, it is still possible that the first search may mislead the second search which will aggravate the overall searching time. Since we divide the SAT instance into two parts and each part can be checked individually, for test generation, we use the following three strategies in parallel:

1. Directly solve the original SAT instance.
2. Solve the first part and then use the learnings to solve the original instance.
3. Solve the second part and then use the learnings to solve the original instance.

Once one of above methods finds a satisfiable assignment for the original SAT instance, the remaining two processes will be terminated. Therefore, we can guarantee the worst case of the test generation time is the same as directly solving the original SAT instance.

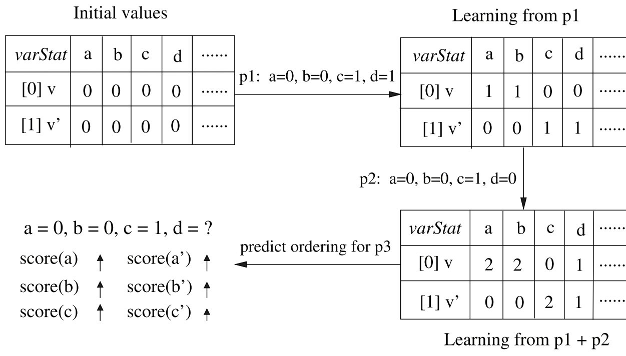

Fig. 9. Statistics for two properties.

5.2 Test Generation for Similar Properties

For similar properties, there exists a large overlap between corresponding counterexamples. Therefore, the satisfiable assignments of checked properties can be used as a learning for solving other properties in the cluster. Some of the derived conflict clauses can also be forwarded as learning. This section will discuss how to extract the bit-value ordering and variable ordering-based learnings from the checked properties in details. Also, we will describe an algorithm to utilize the learning based on decision ordering for test generation of a cluster of similar properties.

5.2.1 Heuristic Implementation

In our heuristic implementation, we predict the decision ordering based on the statistics collected from the checked properties. Let $varStat[sz][2]$ (sz is the largest variable number for CNFs) be a 2-dimensional array to keep the count of variable assignments. Initially, $varStat[i][0] = varStat[i][1] = 0$ $(0 < i \leq sz)$. $varStat$ will be updated after checking each property. Assuming we are now checking property p_j, if the value of variable v_i in the assignment of the p_j is 0, then $varStat[i][0]$ will be increased by one; otherwise, $varStat[i][1]$ will be increased by one. This updated information of $varStat$ will be utilized when checking property p_{j+1}.

For example, if we have three properties p_1, p_2, and p_3, the statistics after checking p_1 and p_2 are shown in Fig. 9. When checking p_3, we can predict its decision ordering based on the collected information saved in $varStat$. The content of $varStat$ indicates that variables a and b are more likely to be 0, c is more likely to be 1 and d can be assigned any value. Furthermore, $varStat$ implies that the assignments for variable a, b, and c are more consistent than the assignment for variable d. Thus, the score of variable a, b, and c will be increased. In other words, they will be searched first as described in Section 4.2.2.

Assuming l_i is a literal of v_i, we use the following equation to predict the bit-value assignment of v_i when checking p_{j+1}:

$$potential(l_i) = \begin{cases} 1, & (varStat[i][1] < varStat[i][0] \& l_i = v_i), \\ & or(varStat[i][1] < varStat[i][0] \& l_i = v_i'), \\ 0, & otherwise. \end{cases} \quad (5)$$

Here, $potential(l_i) = 0$ means that value of l_i is more likely to be 0 in the satisfiable assignment of p_{j+1}. For example, in Fig. 9, $potential(a) = 0$ which means that a is more likely to be assigned with 0. Let

$$ratio(i) = \frac{max(varStat[i][0], varStat[i][1]) + 1}{min(varStat[i][0], varStat[i][1]) + 1}, \quad (6)$$

indicates the assignment variance of variable v_i. The larger $ratio_i$ means that the value assignments for variable v_i are more consistent. So it can be used for variable ordering.

Our decision ordering heuristic is based on VSIDS. The only difference is that our method incorporates the statistics of previously checked properties. For each literal l_i, we use $score(l_i)$ to describe its priority. Initially, $score(l_i)$ is equal to the literal count of l_i. At the beginning of search as well as periodically decaying time, the literal score will be recalculated using the following equation where $max(v_i) = MAX(score(v_i), score(v_i')) + 1$:

$$score(l_i) = \begin{cases} max(v_i) * ratio(i), & pontential(l_i) = 1, \\ score(l_i) * ratio(i), & otherwise. \end{cases} \quad (7)$$

5.2.2 Test Generation Using Interproperty Learning

Algorithm 3 describes our test generation methodology. The inputs of the algorithm are a formal model of the design and a cluster of similar properties. The first step initializes $varStat$ which is used to keep statistics of the variable assignments. Step 2 generates the CNF clauses for the base property p_1. Step 3 generates the CNF clauses for other properties. Step 4 derives the test for a base property using the method proposed in Section 5.1.2. Steps 5-6 generate tests for the remaining properties in the cluster. When solving each property, we need to update the $varStat$ accordingly in step 5. Step 6 solves the current property using the learnings based on decision ordering. Finally, the algorithm reports all the generated counterexamples (tests). It is worthy noting that sharing conflict clauses among properties needs to calculate the intersection between the base property and other properties in a cluster. The overhead of intersection calculation is not negligible and can be larger than the test generation time for the nonbase properties. Therefore, in this algorithm we use the hybrid learning for the base property, and for the remaining properties we adopt the decision ordering-based learning. We refer this approach as $Hybrid \rightarrow DOL$.

Algorithm 3: Test Generation for a Property Cluster

```
Input: i) Formal model of the design, D
       ii) Property cluster, P, with satisfiable bounds
Output: Test-suite
1. Initialize varStat;
2. Select the base property p1 and generate CNF, CNF1;
for i is from 2 to the size of cluster P do
  | 3. Generate CNF, CNFi = BMC(D, pi, boundi);
end
4. test1 = Algorithm1(D, p1, bound1);
Test-suite = test1 ;
for i is from 2 to the size of cluster P do
  | 5. Update varStat using test(i-1);
  | 6. testi = SAT(CNFi, vatStat, NULL);
  | Test-suite = Test-suite ∪ testi;
end
return Test-suite;
```

TABLE 1
Test Generation Results Using Intraproperty Learnings

SAT Instance	CNF Size		zChaff [39]	Our Intra-Property Learning			Max
	#Variable	#Clause	Time (s)	CCF (s)	DOL (s)	Hybrid (s)	Speedup
BMC-galileo-8	58074	294821	1.60	0.67	1.18	**0.63**	2.54
BMC-galileo-9	63624	326999	2.84	1.59	1.47	**0.86**	3.30
BMC-ibm-10	59056	323700	13.24	**6.65**	13.60	12.94	1.99
BMC-ibm-11	32109	150027	12.29	8.00	**3.17**	12.44	3.88
VLIW-1	521188	13378461	1332.46	1047.5	2002.25	**474.95**	2.81
VLIW-2	521158	13378532	196.58	**65.84**	215.38	290.93	2.99
VLIW-3	521046	13376161	145.35	150.81	54.70	**51.73**	2.81
VLIW-4	520721	13348117	1104.79	288.39	605.09	**93.50**	11.82
VLIW-5	520770	13380350	858.61	742.23	686.27	**165.59**	5.19
VLIW-6	521192	13378781	209.85	**52.66**	526.58	308.22	3.98
VLIW-7	521147	13378010	**87.42**	189.79	345.48	391.73	1.00
VLIW-8	521179	13378617	1200.68	931.99	441.59	**369.33**	3.25
VLIW-9	521187	13378624	941.25	**180.99**	1476.84	1539.59	5.20
VLIW-10	521182	13378625	1725.82	**896.52**	902.19	1540.70	1.93
PIPE-1	138917	4678756	1327.92	777.63	284.67	**284.60**	4.84
PIPE-2	138918	4678718	1710.66	1767.09	412.45	**412.32**	4.29
PIPE-3	138917	4678757	825.78	374.07	**376.47**	995.45	2.28
PIPE-4	138563	4675040	1080.10	33.00	418.74	**14.49**	77.14
PIPE-5	138918	4678760	626.9	583.25	614.63	**117.92**	5.48
PIPE-6	138795	4671352	**0.43**	0.61	118.72	118.72	1.00
PIPE-7	138918	4678760	1734.26	1013.67	1391.29	**549.90**	3.26
PIPE-8	138711	4688614	113.07	2.50	0.62	**0.62**	190.9
PIPE-9	138916	4676007	6062.27	2664.06	364.36	**359.18**	17.48
PIPE-10	138918	4678760	1430.29	1086.73	**285.67**	986.53	5.09
Statistics	-	-	23238.77	12866.24	11534.41	**9092.87**	2.56

6 EXPERIMENTS

This section presents case studies for efficient test generation using our decision ordering as well as conflict clause-based heuristics. Section 6.1 presents the case studies using intraproperty learnings for checking individual SAT instances. The benchmarks collected are all pregenerated satisfiable SAT instances. Section 6.2 presents two case studies: a VLIW implementation of the MIPS architecture [38] and a stock exchange system [25]. For each case study, we generate a number of properties and group them into several clusters according to their similarity. By using intraproperty learning for the base property of the cluster and interproperty learning for the other properties in the same cluster, the overall test generation time can be reduced. We used NuSMV [40] to generate the CNF clauses (in DIMACS format). We modified the SAT solver zChaff [39] to incorporate our proposed decision ordering heuristic on top of VSDIS. The experimental results are obtained on a Linux PC using 2.0 GHz Intel Core i7 CPU with 3 GB RAM.

6.1 Intraproperty Learning

The benchmarks are collected from [36] and [37]. In [36], there are 13 SAT instances given in the benchmark set which are all taken from real industrial hardware designs (contribution of IBM research and Galileo). We only chose four complex instances from them, because for most other SAT instances provided in [36] the falsification time is so short that the improvement of test generation is not obvious. Apart from these four benchmarks, we also chose the benchmarks of two complex designs from [37] as follows: Since we are focusing on test generation, the collected SAT instances are all satisfiable.

1. **VLIW-SAT-4.0**, buggy VLIW processors with instruction queues and 9-stage pipelines; the processors support advanced loads, predicated execution, branch prediction, and exceptions.
2. **PIPE-SAT-1.1**, buggy variants of the pipe benchmarks and generated as presented in [34].

Table 1 shows the test generation details using various intraproperty learning techniques. The first column shows the names of the SAT instances. The second and third columns indicate the CNF size information including the variable number and clause number. The fourth column indicates the checking time by directly using zChaff [39] without any other learning information. The fifth column shows the checking time using intraproperty learning based on conflict clause forwarding, and the sixth column shows the test generation time using our decision ordering-based technique. The seventh column presents the result of our hybrid learning which incorporates both conflict clause forwarding and decision ordering techniques as described in Section 4.3 and implemented in Algorithm 2. The execution time in columns 5-7 includes the learning time from divided/segmented CNFs. It is important to note that all the learning methods are not always helpful for the test generation. This is because the learning methods may lead the search in a wrong way with more conflicts. However since we run different methods on different computers with the same settings, when one machine gets the satisfiable assignment, all the remaining SAT searches on the other machines will be terminated. Therefore, the SAT searching time is the minimum searching time among these techniques.

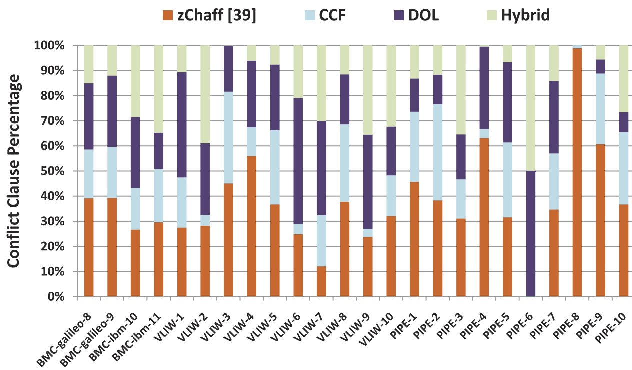

Fig. 10. Conflict statistics using various intraproperty learnings.

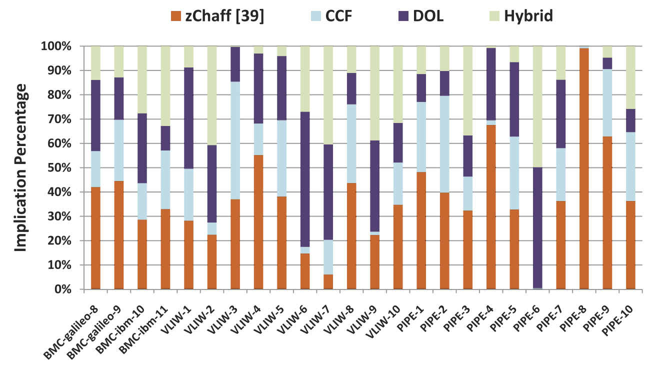

Fig. 11. Implication statistics using various intraproperty learnings.

Based on such minimum time, the last column indicates the maximum speedup using the following equation:

$$max_speedup = \frac{zChaff}{MIN(zChaff,\ CCF,\ DOL,\ Hybrid)}, \quad (8)$$

where $zChaff$, CCF, DOL, and $Hybrid$ indicate the test generation time (in seconds) of column 4-7 in Table 1, respectively. In the last row, we provide the maximum speedup for the 24 SAT instances using different methods. To explicitly compare the performance of the four methods, in this table, we make the best results in bold fonts.

Table 1 shows that our methods can drastically reduce the test generation time (up to 191 times) in most cases (22 out of 24 SAT instances). We can observe that the conflict clause forwarding-based method achieves better performance than zChaff for 20 out of 24 examples. For decision ordering-based method, it has a better overall test generation time (11,534.41 s) than conflict clause forwarding-based method (12,866.24 s). However, this is largely due to the test generation improvement in the SAT instance "PIPE-9." Therefore, compared to the conflict clause forwarding-based method, decision ordering is not quite a promising intraproperty learning for the test generation of a single SAT instance. As shown in the last row of the table, the hybrid method achieves the best overall performance (with a speedup of 2.56 times in total test generation time). The hybrid method outperforms conflict clause forwarding-based method in 15 SAT instances and outperforms decision ordering-based method in 14 out of 24 SAT instances. Moreover, the hybrid method can achieve the best performance in 14 out of 24 SAT instances. Therefore, the hybrid method is the first choice of intraproperty learning when there is only one computer available.

Figs. 10 and 11 show the statistics of conflicts and implications for the collected benchmarks using various intraproperty learning methods. We normalized the generated conflict clauses for each learning method using the total conflict clauses and implications generated by the four different methods shown in Table 1. The vertical axis of the stacked graphs shows the normalized percentage of conflict clauses and implications, respectively. We can find that the result of the percentages of conflict clauses and implications are consistent. In other words, less conflicts will result in less implications. Furthermore, these figures also are consistent with the test generation performance results shown in Table 1. For example, in case of "PIPE-1" in Table 1, the order of the test generation time is $zChaff > CCF > DOL > Hybrid$. In Figs. 10 and 11, we can find the percentage of the evaluated metrics is also in the same order. Therefore, if parallel invocation of methods is not possible, hybrid learning should be employed.

6.2 Interproperty Learning

6.2.1 A MIPS Processor

This case study investigates a simplified version of an MIPS processor [38]. It consists of five pipeline stages: fetch, decode, execute, memory, and writeback. In this case study, we focus on the validation of pipeline paths (ALU, DIV, FADD, and MUL) in the execute stage. Targeting to check whether each pipeline path can give the correct outputs, we derived a set of 16 properties to generate the required directed tests by applying our methodology.

Table 2 shows an example of an LTL property derived for test generation. The property is in the form of "$!F(p)$" which asserts that the functional scenario p will never happen. Since we assume that the specified scenario p is correct, the corresponding SAT instance will be satisfiable and its true assignment can be used to derive the test to activate p. For example, the property p refers to a functional scenario of the multiplication operation from the decode stage to the writeback stage. When decoding an instruction, we can get the value of its operands ($mulOP_src1 = 4$ and $mulOP_src2 = 5$) from registers, operation ($mulOP_opcode = mul$) and target register address (mulOP_dest=1). Since multiplication pipeline needs seven clock cycles to finish the calculation, there is a delay of eight clock cycles to get the final result in the writeback stage. The property p asserts that after eight clock cycles, the target $Reg1$ address does not have the correct value 20. By using our framework, we can get a directed test (i.e., a sequence of instructions) to falsify this property.

According to the structure similarity proposed in [24], we cluster the properties of each pipeline path together to share the learning. There are four clusters and each cluster

TABLE 2
A Negated Property for MIPS Processor

p	$!F((mulOp_src1 = 4\ \&\ mulOp_src2 = 5\ \&\ mulOp_opcode = mul$ $\&\ mulOp_dest = 1)\ \&\ X(X(X(X(X(X(X(X(Reg1 = 20)))))))))$

TABLE 3
Test Generation Result for MIPS Processor

MIPS Unit	Prop. (Tests)	CNF Size #Variable	CNF Size #Clause	zChaff [39] Time (s)	CCF [24] Inter. (s)	CCF [24] Time (s)	CCF [24] Speedup	DOL Time (s)	DOL Speedup	Hybrid → DOL Time (s)	Hybrid → DOL Speedup
ALU Unit	p_1 *	40881	467869	19.55	0	19.61	1	19.75	0.99	10.60	1.84
	p_2	41327	467861	19.24	1.90	1.44	5.76	0.10	192.40	0.23	83.65
	p_3	41774	467869	16.10	1.21	0.36	10.25	0.11	146.36	0.10	161.00
	p_4	42228	467861	17.38	1.47	0.21	10.35	0.26	66.8	0.33	52.64
Summary	all	-	-	72.27	26.20		2.76	20.22	3.57	11.26	**6.42**
DIV Unit	p_5 *	40903	467880	13.34	0	13.28	1	13.49	0.99	8.13	1.64
	p_6	41341	467799	17.84	1.42	1.26	6.66	0.18	99.11	0.10	178.40
	p_7	41808	467880	15.10	1.13	1.08	6.83	0.16	94.38	0.35	43.14
	p_8	42274	467880	14.84	1.79	0.31	7.07	0.17	87.29	0.46	32.26
Summary	all	-	-	61.12	20.27		3.02	14.00	4.37	9.04	**6.76**
FADD Unit	p_9 *	40879	467892	16.50	0	16.37	1	16.55	1	9.61	1.72
	p_{10}	41368	467892	19.53	1.48	0.26	11.22	0.25	78.12	0.17	114.88
	p_{11}	41838	467892	21.29	1.25	0.91	9.86	0.15	141.93	0.10	212.90
	p_{12}	42309	467892	19.44	1.78	0.52	8.45	0.15	176.73	0.10	176.73
Summary	all	-	-	76.76	22.57		3.40	17.06	4.50	9.99	**7.68**
MUL Unit	p_{13} *	52099	600928	49.94	0	49.94	1	49.90	1	22.53	2.22
	p_{14}	52824	600928	49.28	2.21	5.20	9.48	0.21	234.67	0.18	273.78
	p_{15}	53551	600928	45.58	1.80	2.35	19.40	0.26	175.31	0.18	253.22
	p_{16}	54276	600928	54.55	1.52	3.83	14.24	0.14	389.64	0.17	320.88
Summary	all	-	-	199.35	61.32		3.25	50.51	3.94	23.06	**8.64**

* Base property

has four property. Table 3 shows the test generation results for each cluster. The first column indicates the component under test. The second column shows the properties used for test generation. The third and fourth columns show the CNF size information including the variable number and clause number. The fifth column gives the test generation time using zChaff [39]. The sixth, seventh, and eighth columns present the results using the method proposed in [24]. Since the conflict clause forwarding-based method needs to explore the common clauses, we need to figure out the intersection between SAT instances. Therefore, the sixth column gives the intersection time. The seventh column gives the checking time under the learning of conflict clauses. The eighth column gives the speedup over zchaff (speedup $= \frac{zChaff\ Time}{Intersection\ Time + Checking\ Time}$). The ninth and tenth columns give the test generation result only using our decision ordering-based learnings. They indicate both the result of test generation time and speedup over zChaff. It is important to note that DOL does not consider how to reduce the test generation time for the base property. To further reduce the overall test generation time, we adopts the $Hybrid \rightarrow DOL$ method described in Section 5.2.2 and implemented in Algorithm 3 which is a combination of intra and interproperty learnings. The last two columns show the result using this method.

In Table 3, we found that the decision ordering is a better interproperty learning than conflict clauses. The decision ordering learning-based method can achieve 3.5-4.5 times improvement compared to the method using zChaff. Furthermore, $Hybrid \rightarrow DOL$ method outperforms other three methods. Since the base property is a major bottleneck of the clustering methods [24], the test generation time reduction of the base property using hybrid learning can drastically increase the overall performance. Therefore, $Hybrid \rightarrow DOL$ method can achieve the best performance with 6-9 times improvement compared to the method using zChaff.

During the SAT searching, the number of conflict clauses and the number of implications strongly indicate the searching time. Fig. 12 illustrates the conflict clause generation for each property during the search using different methods. Fig. 13 shows the corresponding implication numbers. It can be seen that, by using our methods, the number of conflict clauses and implications can be reduced drastically by several orders-of-magnitude, which results in

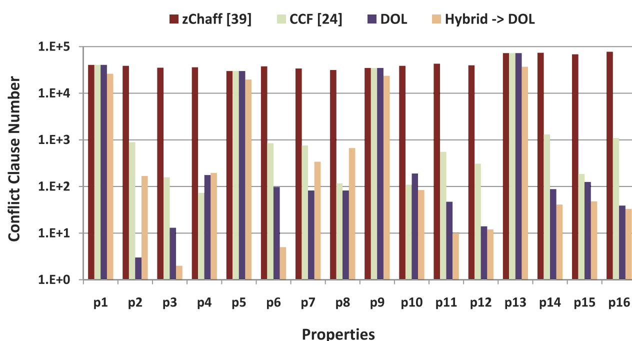

Fig. 12. Conflict statistics for MIPS processor.

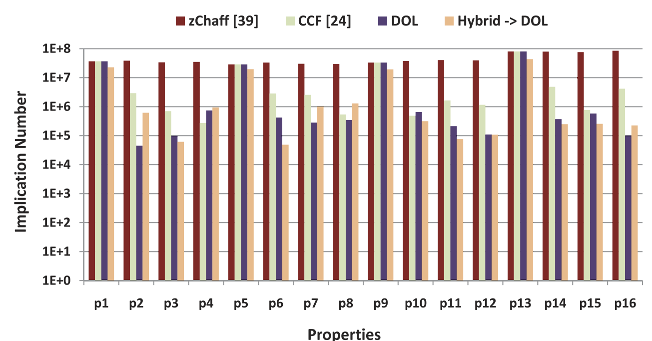

Fig. 13. Implication statistics for MIPS processor.

TABLE 4
Negated Properties of Cluster 4 (C4) Used in Table 5

Index	Property
p_1	$!F(verify_order_form = done$ & $add_order_form_list = done$ & $get_new_order = done$ & $trade_market_order_sale = done$ & $trade_failure = done)$
p_2	$!F(verify_order_form = done$ & $add_order_form_list = done$ & $get_new_order = done$ & $trade_market_order_sale = done$ & $update_orderDB_failure = done)$
p_3	$!F(verify_order_form = done$ & $add_order_form_list = done$ & $get_new_order = done$ & $trade_market_order_buy = done$ & $trade_failure = done)$
p_4	$!F(verify_order_form = done$ & $add_order_form_list = done$ & $get_new_order = done$ & $trade_market_order_buy = done$ & $update_orderDB_failure = done)$

significant improvement in test generation time. We can find that the decision ordering based method performs better than conflict clause forwarding-based method because of less conflicts and implications encountered. Furthermore, the decision ordering method does not need to calculate the CNF intersections which is time consuming. Among the four methods, $Hybrid \rightarrow DOL$ method can achieve least number of conflicts and implications for base properties (i.e., p_1, p_5, p_9, and p_{13}), which justifies our discussion in Section 4.3. It can achieve the best performance in 8 out of 12 nonbase properties (i.e., $p_3, p_6, p_{10}, p_{11}, p_{12}, p_{14}, p_{15}$, and p_{16}). Therefore, $hybrid \rightarrow DOL$ method gives the best performance in the overall test generation time.

6.2.2 A Stock Exchange System

The online stock exchange system (OSES) is a small-size software system which is implemented in JAVA and consists of seven packages, 39 classes, 372 methods, and 2,510 lines. The formal NuSMV description of OSES is derived from its UML activity diagram specification, which contains 27 activities, 29 transitions. It mainly deals with three scenarios: accept, check, and execute the customers' orders (market orders and limit orders). A path in the UML activity diagram indicates a stock transaction flow (e.g., limit buy, market sale, etc.).

There are a total of 49 properties generated based on the path coverage criteria. We group them into nine clusters. Because of the limitation of space, in Table 4 we only present the second cluster C_2 with four properties. Each property in C_2 refers to a sequence of actions. For example, p_1 asserts a scenario where a market sale transaction happens but the transaction fails at last, and p_3 asserts a scenario where a market buy transaction happens but the transaction fails at last. Because of the textual similarity [24], we group these four properties together to share the learnings.

Table 5 shows the test generation results involving all the nine clusters. The first column indicates the clusters. The second column indicates the size of each cluster (i.e., number of properties). The third column presents the test generation time (including base property) using zChaff. The fourth column gives the result using conflict clause-based property learnings [24]. The fifth column presents the result using our decision ordering-based property learnings. In this method, we do not consider the intraproperty learning for the base property. The last column indicates the test generation time using the method proposed in Algorithm 3. In this case study, we can find that the $hybrid \rightarrow DOL$ method can produce an average of 13.32 times overall

TABLE 5
Test Generation Result for Stock Exchange System

Cluster	Size	zChaff [39] (s)	CCF [24] (s)	DOL (s)	Hybrid $\rightarrow$ DOL (s)
C_1	3	26.95	25.32	14.86	9.04
C_2	4	74.05	38.91	3.71	4.02
C_3	8	350.99	313.77	17.99	28.47
C_4	4	4.12	5.75	2.67	1.27
C_5	4	62.33	71.55	8.97	5.44
C_6	8	535.06	269.22	23.06	40.07
C_7	2	10.13	6.94	4.66	4.59
C_8	8	768.32	332.68	73.15	51.26
C_9	8	241.99	145.86	40.95	11.59
Total	**49**	**2073.94**	**1210.00**	**190.02**	**155.75**
Speedup	**-**	**1**	**1.71**	**10.91**	**13.32**

improvement in test generation time compared to zChaff. It is important to note that the $hybrid \rightarrow DOL$ method can achieve the best performance, which is consistent with the results obtained in Section 6.2.1.

7 Conclusions

Simulation using directed tests is promising for functional validation, since the overall simulation effort can be reduced with fewer tests while the coverage requirement can still be achieved. However, the applicability of directed test generation using model checking is limited due to the capacity restriction of corresponding tools. To address the complexity of test generation using SAT-based BMC, this paper presented a novel methodology which explores the intraproperty learning within a SAT instance and interproperty learning between similar SAT instances. All these learnings are based on decision ordering heuristics as well as conflict clause forwarding techniques. To the best of our knowledge, our work is the first attempt to share the decision ordering learnings on different parts of a SAT instance as well as across multiple properties. By exploiting the commonalities during the search of satisfiable assignments, the test generation time of a single property as well as a set of similar properties can be drastically reduced. The experimental results using both hardware and software designs demonstrated the effectiveness of our method.

Acknowledgments

This work was partially supported by grants from US National Science Foundation (NSF) CAREER award 0746261, National Natural Science Foundation of China (No. 61021004 and No. 90818024), and National Grand Fundamental Research 973 Program of China (No. 2009CB320702). A preliminary version [26] of this paper has appeared in Design, Automation, and Test in Europe (DATE) 2010.

References

[1] J.W. Duran and S.C. Ntafos, "An Evaluation of Random Testing," *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 438-444, Apr. 1999.
[2] S. Fine and A. Ziv, "Coverage Directed Test Generation for Functional Verification Using Bayesian Networks," *Proc. Design Automation Conf.*, pp. 286-291, 2003.

[3] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking," *Advances in Computers*, vol. 58, no. 3, pp. 118-149, 2003.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic Model Checking Without BDDs," *Proc. Fifth Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems*, pp. 193-207, 1999.

[5] P.E. Ammann, P.E. Black, and W. Majurski, "Using Model Checking to Generate Tests from Specifications," *Proc. Int'l Conf. Formal Eng. Methods*, pp. 46-55, 1998.

[6] E. Clarke, O. Grumberg, and D. Peled, *Model Checking.* MIT Press, 1999.

[7] J.E. Hopcroft, R. Motwani, and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, third ed. Addison-Wesley, 2006.

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

[9] N. Amla, X. Du, A. Kuehlmann, R. Kurshan, and K. McMillan, "An Analysis of SAT-Based Model Checking Techniques in an Industrial Environment," *Proc. Correct Hardware Design and Verification Methods*, vol. 3725, pp. 254-268, 2005.

[10] H. Koo and P. Mishra, "Test Generation Suing SAT-Based Bounded Model Checking for Validation of Pipelined Processors," *Proc. Great Lake Symp. VLSI*, pp. 362-365, 2006.

[11] H. Zhang, "SATO: An Efficient Propositional Prover," *Proc. 14th Int'l Conf. Automated Deduction*, pp. 272-275, 1997.

[12] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *Proc. 38th Design Automation Conf.*, pp. 530-535, 2001.

[13] J. Marques-Silva and K. Sakallah, "Grasp: A Search Algorithm for Propositional Satisfiability," *IEEE Trans. Computers*, vol. 48, no. 5, pp. 506-521, May 1999.

[14] I. Lynce and J. Marques-Silva, "Efficient Data Structures for Backtrack Search SAT Solvers," *Annals of Math. and Artificial Intelligence*, vol. 43, no. 1, pp. 137-152, 2005.

[15] V. Durairaj and P. Kalla, "Guiding CNF-SAT Search via Efficient Constraint Partitioning," *Proc. Int'l Conf. Computer-Aided Design*, pp. 498-501, 2004.

[16] V. Durairaj and P. Kalla, "Variable Ordering for Efficient SAT Search by Analyzing Constraint-Variable Dependencies," *Proc. Int'l Conf. Theory and Applications of Satisfiability Testing*, pp. 415-422, 2005.

[17] L. Zhang, C.F. Madigan, M.H. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," *Proc. Int'l Conf. Computer-Aided Design*, pp. 279-285, 2001.

[18] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulations," *IEEE Trans. Computers*, vol. 35, no. 8, pp. 677-691, Aug. 1986.

[19] E.M. Clarke, O. Grumberg, and D.E. Long, "Model Checking and Abstraction," *ACM Trans. Programming Languages and Systems*, vol. 16, no. 5, pp. 1512-1542, 1994.

[20] P. Bjesse and J. Kukula, "Using Counter Example Guided Abstraction Refinement to Find Complex Bugs," *Proc. Conf. Design, Automation, and Test in Europe*, pp. 156-161, 2004.

[21] H. Koo and P. Mishra, "Functional Test Generation Using Design and Property Decomposition Techniques," *ACM Trans. Embedded Computing Systems*, vol. 8, no. 4, Article 32, July 2009.

[22] H. Jin and F. Somenzi, "An Incremental Algorithm to Check Satisfiability for Bounded Model Checking," *Proc. Int'l Workshop Bounded Model Checking (BMC '04)*, vol. 119, no. 2, pp. 51-65, 2005.

[23] O. Strichman, "Pruning Techniques for the SAT-Based Bounded Model Checking Problem," *Proc. 11th IFIP WG 10.5 Advanced Research Working Conf. Correct Hardware Design and Verification Methods*, vol. 2144, pp. 58-70, 2001.

[24] M. Chen and P. Mishra, "Functional Test Generation Using Efficient Property Clustering and Learning Techniques," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 3, pp. 396-404, Mar. 2010.

[25] M. Chen, X. Qiu, W. Xu, L. Wang, J. Zhao, and X. Li, "UML Activity Diagram Based Automatic Test Case Generation for Java Programs," *The Computer J.*, vol. 52, no. 5, pp. 545-556, 2009.

[26] M. Chen and P. Mishra, "Efficient Decision Ordering Techniques for SAT-Based Test Generation," *Proc. Conf. Design, Automation and Test in Europe*, pp. 490-495, 2010.

[27] L. Zhang, M.R. Prasad, and M.S. Hsiao, "Incremental Deductive and Inductive Reasoning for SAT-Based Bounded Model Checking," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design*, pp. 502-509, 2004.

[28] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," *Proc. Conf. Programming Language Design and Implementation*, pp. 213-223, 2005.

[29] K. Sen, D. Marinov, and G. Agha, "Cute: A Concolic Unit Testing Engine for C," *Proc. Fifth Joint Meeting of the European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng. (ESEC/FSE '05)*, pp. 263-272, 2005.

[30] D.A. Mathaikutty, S.K. Shukla, S.V. Kodakara, D. Lilja, and A. Dingankar, "Design Fault Directed Test Generation for Microprocessor Validation," *Proc. Conf. Design, Automation and Test in Europe*, pp. 761-766, 2007.

[31] J.P. Marques-Silva and K.A. Sakallah, "The Impact of Branching Heuristics in Propositional Satisfiability," *Proc. Ninth Portuguese Conf. Artificial Intelligence*, pp. 62-74, 1999.

[32] O. Shtrichman, "Tuning SAT Checkers for Bounded Model Checking," *Proc. 12th Int'l Conf. Computer Aided Verification*, pp. 480-494, 2000.

[33] C. Wang, H. Jin, G.D. Hachtel, and F. Somenzi, "Refining the SAT Decision Ordering for Bounded Model Checking," *Proc. Design Automation Conf.*, pp. 535-538, 2004.

[34] M.N. Velev, "Automatic Abstraction of Equations in a Logic of Equality," *Proc. Analytic Tableaux and Related Methods (TABLEAUX)*, pp. 196-213, 2003.

[35] M. Davis, G. Logemann, and D.W. Loveland, "A Machine Program for Theorem-Proving," *Comm. the ACM*, vol. 5, no. 7, pp. 394-397, 1962.

[36] *SAT Benchmark Problems*, http://www.satlib.org/Benchmarks/SAT/BMC/description.html, 2011.

[37] *Miroslav Velev's SAT Benchmarks*, http://www.miroslav-velev.com/sat_benchmarks.html, 2011.

[38] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 2003.

[39] *zChaff*, http://www.princeton.edu/~chaff/zchaff.html, 2011.

[40] *NuSMV*, http://nusmv.irst.itc.it/, 2011.

Mingsong Chen (S'08-M'11) received the BS and ME degrees from the Department of Computer Science and Technology, Nanjing University, China, in 2003 and 2006, respectively, and the PhD degree in computer engineering from the University of Florida, in 2010. He is currently an associate professor with the Software Engineering Institute of East China Normal University. His research interests are in the area of design automation of embedded systems, formal verification techniques and software engineering. He is a member of the IEEE.

Prabhat Mishra (S'00-M'04-SM'08) received the BE degree from Jadavpur University, India, the MTech degree from the Indian Institute of Technology, Kharagpur, and the PhD degree from the University of California, Irvine—all in computer science. He is currently an associate professor with the Department of Computer and Information Science and Engineering, University of Florida. He has published two books, nine book chapters, and more than 70 research articles in premier journals and conferences. His research has been recognized by several awards including an NSF CAREER Award in 2008, two best paper awards (VLSI Design 2011 and CODES+ISSS 2003), several best paper award nominations (including DAC 2009 and VLSI Design 2009), and 2004 EDAA Outstanding Dissertation Award from the European Design Automation Association. He currently serves as information director of *ACM Transactions on Design Automation of Electronic Systems*, guest editor of *IEEE Design and Test of Computers*, and as a program/organizing committee member of several ACM and IEEE conferences. His research interests include design automation of embedded systems, reconfigurable architectures, and functional verification. He is a senior member of the IEEE.