Functional Test Generation Using Efficient Property Clustering and Learning Techniques

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

***Abstract*—Functional verification is one of the major bottlenecks in system-on-chip design due to the combined effects of increasing complexity and lack of automated techniques for generating efficient tests. Several promising ideas using bounded model checking are proposed over the years to efficiently generate counterexamples (tests). The existing researchers have used incremental satisfiability to improve the counterexample generation, involving only one property by sharing knowledge across instances of the same property with incremental bounds. In this paper, we present a framework that can efficiently reduce the overall test generation time by exploiting the similarity among different properties. This paper makes two primary contributions: 1) it proposes novel methods to cluster similar properties; and 2) it develops efficient learning techniques that can significantly reduce the overall test generation time for the properties in a cluster by sharing knowledge across similar test generation instances. Our experimental results using both software and hardware benchmarks demonstrate that our approach can drastically reduce (on average three to five times) the overall test generation time compared to existing methods.**

***Index Terms*—Bounded model checking, functional verification, property clustering, SAT, test generation.**

I. Introduction

FUNCTIONAL verification is one of the most time-consuming and costly phases in the system-on-chip (SOC) design flow due to the combined effects of increasing design complexity and decreasing time-to-market. Existing validation approaches use a combination of simulation-based techniques and formal methods. Simulation is the most widely used form of SOC validation using random, constrained-random, and directed test vectors. As expected, the directed tests exploit the structural and functional information of a system. As a result, the directed tests can achieve the same coverage goal using orders-of-magnitude less tests compared to random or constrained-random tests, and therefore can drastically reduce the overall simulation time and validation effort. However, directed test generation is mostly performed by human intervention. Hand-written tests entail laborious and time-consuming effort of verification engineers who have deep knowledge of the design under verification. For a complex design, it is infeasible to manually generate all directed tests to achieve a comprehensive coverage goal. Therefore, it is necessary to develop tools and techniques for automated directed test generation.

Boolean satisfiability (SAT) based bounded model checking (BMC) [1], [2] is very promising for automated generation of directed tests. Given a model M, a safety property p, and a bound k, BMC will unfold the model k times and encode it using the following logic formula. Here, $I(s_0)$ means the initial state of the system, $T(s_i, s_{i+1})$ describes the state transition from state s_i to state s_{i+1}, and $p(s_i)$ tests whether property p holds on state s_i. This formula is transformed to conjunctive normal form (CNF) and checked by a SAT solver [3], [4]. If there is a satisfiable assignment, the property is false and the assignment will be reported as a counterexample. Recent SAT solvers use conflict analysis techniques to trace the reason for a conflict. It saves the knowledge using *conflict clauses* and adds them to the original clauses, in order to avoid the same conflicts in the future

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

Incremental SAT-based BMC [5] can reduce test generation complexity by exploiting similarity between incremental SAT instances. However, existing approaches are restricted for a test generation scenario consisting of one design and only one property (with varying bounds). According to (1), a set of similar properties will have a large overlap between their CNF clauses, since the equation shares the system part [transition relation $T(s_i, s_{i+1})$] and partial property checking part [$p(s_i)$]. Such a large overlap of CNF clauses indicates that when searching a counterexample for a property, some learned knowledge (conflict clauses) can be reused by other similar properties. If the shared information can be exploited and reutilized across the similar properties, many repeated verification efforts can be avoided. Therefore, knowledge sharing can reduce complexity and improve the overall verification effort. A major challenge in implementing this idea is to identify the property similarities and perform efficient clustering to share learning and thereby reduce the overall test generation time. This paper makes two primary contributions to address this problem: 1) we propose several methods to efficiently cluster the similar properties; and 2) we develop a number of conceptually simple, but very effective,

Manuscript received April 14, 2009; revised August 10, 2009. Current version published February 24, 2010. This work was supported in part by the National Science Foundation Faculty Early Career Development Award 0746261. This paper was recommended by Associate Editor T. Lynch.

The authors are with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611-6120 USA (e-mail: mchen@cise.ufl.edu; prabhat@cise.ufl.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2010.2041846

0278-0070/$26.00 © 2010 IEEE

learning techniques to generate tests for a cluster of similar properties.

The rest of this paper is organized as follows. Section II presents related work addressing test generation approaches using model checking. Sections III–VI present our methodology for functional test generation using property clustering and learning techniques. Section VII presents our experimental results using both software and hardware benchmarks. Finally, Section VIII concludes this paper.

II. Related Work

Formal verification techniques such as model checking [6] have been successfully used in software and hardware verification domain as a test generation engine [7], [8]. Traditional model checking techniques do not scale well due to the state explosion problem. Biere *et al.* [1] introduced the framework of SAT-based BMC which can complement the existing model checking techniques. SAT-based BMC is an incomplete method that cannot guarantee a true or false determination when a counterexample does not exist within a given bound. However, once the bound of a counterexample is known, large designs can be falsified very fast, and searching a counterexample in an arbitrary order consumes much less memory than traditional techniques. Several recent developments in related techniques have been presented in [5]. The performance of bounded and unbounded algorithms was analyzed on a set of industrial benchmarks in [9]. An Intel study [10] shows that BMC has better capacity and productivity over unbounded model checking for real designs taken from the Pentium-4 processor. The performance of SAT based test generation and validation can be further improved by employing various approaches. Clarke *et al.* [11] used a combination of integer linear programming and machine learning for abstraction refinement to enable faster verification of safety properties. Cheng and Hsiao [12] proposed a novel framework for software verification that combines data mining with BMC. After mining a set of high-level potential property invariants from the dynamic execution data of software, this method uses it as the constraint to prune the search space during assertion checking. However, these approaches use learning to improve verification time involving only one property, whereas our approach tries to improve overall verification time by exploiting similarities across properties.

Incremental SAT approaches [13]–[17] try to leverage the similarity between the elements of a sequence of SAT instances—most do so by re-utilizing learned knowledge (conflict clauses). Majority of the existing approaches exploit incremental satisfiability to improve the test generation time involving only one property with different bounds. There are very few approaches such as [18] where both static and dynamic learning is used across test generation instances for path-delay fault model by dynamically excluding the untestable path during the test generation. Since the learning is employed across all test scenarios without efficient clustering methods, the improvement in test generation time is small (6% on average) and have a wide variation (−7% to 27%) on different ISCAS circuits. To the best of our knowledge, our approach is the first attempt to cluster similar test generation instances involving multiple properties and utilize shared knowledge across similar instances in the context of directed test generation.

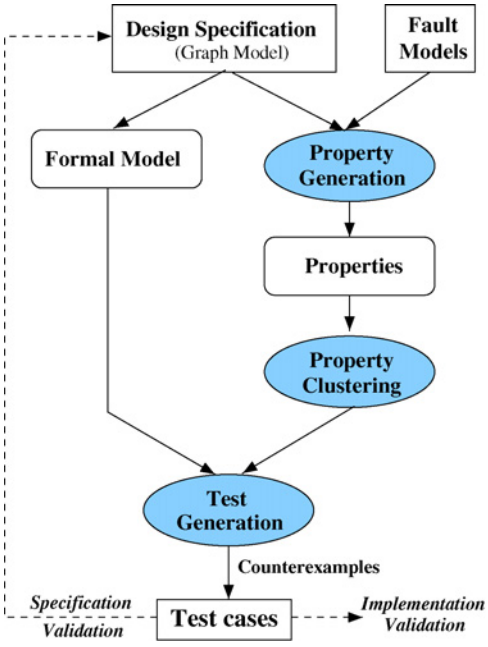

Fig. 1. Our test generation methodology.

III. Test Generation Using Property Clustering and Learning Techniques

Fig. 1 shows the overall methodology of our test generation framework. The proposed methodology has three important steps: coverage-driven property generation, clustering of similar properties, and test generation using learning techniques. It is important to note that each of these three steps can be independent. For example, this paper uses a graph model of the design and a fault model based on functional coverage to enable coverage-driven property generation. The other two steps (primary contributions of this paper) will produce beneficial results even if other design or fault models are used to generate properties. Designers can even add various properties manually to the set of generated properties without affecting the usefulness of our approach.

While each of these steps can work independently, there is a strong correlation between them. Automated property generation methods can be helpful for property clustering. For example, property clustering based on textual overlap (discussed in Section V-B) will benefit if the property generation is performed using a uniform method compared to if the same set of properties is written by different designers. Similarly, property clustering can significantly influence the learning efficiency during test generation. Our experiments show that certain clustering techniques perform better than others in many scenarios. Moreover, clustering techniques that can generate a small number of large clusters are beneficial in terms of overall test generation time compared to the ones that create many small clusters.

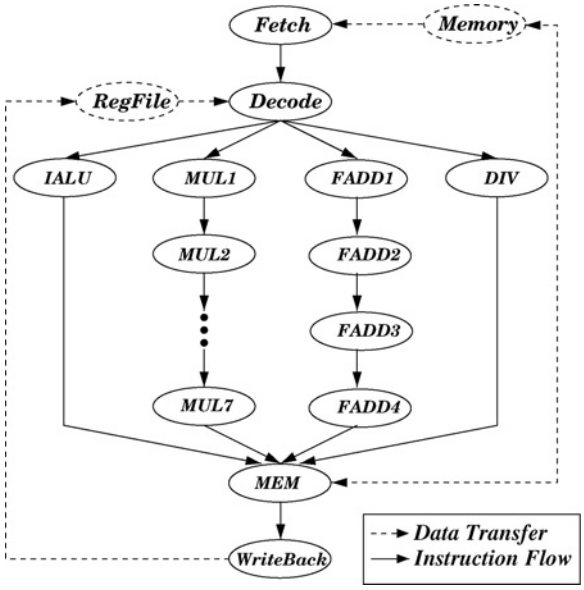

Fig. 2. Graph model of a VLIW MIPS processor.

The following three sections describe three important steps in this methodology. Section IV describes how to generate properties based on specific fault models. Next, Section V describes how to efficiently cluster the generated properties using design and property related information such as structure or behavior similarity. Finally, Section VI presents a set of learning techniques that can share knowledge for solving a set of similar properties to reduce the overall test generation time.

IV. Coverage-Driven Property Generation

Property generation is a crucial step in automated generation of directed tests. To automatically generate properties, we need to have a model of the design and specific fault models. Clearly, the model of the design is dependent on the fault model and vice versa. For example, it is infeasible to use a high-level design model for a low-level fault model. This paper uses a graph-based model of the design and a high-level fault model for property generation.

Graph-based modeling of hardware and software designs is widely used as a suitable high-level specification. For software designs, it can be viewed as a task graph where each node is an activity (task), and an edge between two nodes (activities) indicates data communication. Similarly, for hardware designs such as SOC designs, the nodes in the graph correspond to SOC components and edges correspond to the connectivity between components. For example, consider the multi-issue microprocessor without interlocked pipeline stages (MIPS) processor [19] shown in Fig. 2. The figure shows the graph model of the processor that can issue up to four operations (an integer arithmetic logic unit (ALU) operation, a floating-point addition operation, a multiply operation, and a divide operation). In the figure, rectangular boxes denote units, dashed rectangles are storages, bold edges are instruction-transfer (pipeline) edges, and dashed edges are data-transfer edges. Such a graph model for the MIPS processor can be automatically generated from an architecture description language specification [20].

Fault model plays an important role in directed test generation. The efficiency of directed tests is directly related to the generated properties which in turn are related to the associated fault model. Our proposed fault model considers the following four types of faults (errors) in the graph-based model of the design. It is important to note that these fault models are by no means the "golden" model rather it is a representative model which can be refined or modified for improved verification methodology. For example, the fault model presented below considers only valid execution for each node. This model can be extended by considering other possible states for each node such as performing a specific computation, stalled, interrupted, encountered exception, and so on.

1) *Node Fault:* Each node is faulty. For example, a node cannot be activated.
2) *Edge Fault:* Each edge is faulty. For example, the respective nodes cannot be activated in that order.
3) *Path Fault:* Each execution path is faulty. For example, the associated nodes and edges are either faulty or their behavior cannot be composed correctly to activate the path.
4) *Interaction Fault:* Each interaction is faulty. For example, an interaction involving a set of nodes cannot be activated simultaneously.

We generate one property for each fault in a fault model. So the transformation from the fault model to the properties (in the form of temporal logic [6]) is a one-to-one mapping. Because a fault is already a negation of the system required behavior, it can be directly used to derive a property for test generation. Since in this paper we focus only on safety property generation for the above fault models, the majority of the properties will be in the form of ∼F (p) or G(∼p). However, other forms of safety properties are also possible and allowed in our framework. The following example shows four properties (one for each fault type) for the graph model shown in Fig. 2.

```
Prop 1: The node Fetch cannot be activated.
   ~F (fetch_active = 1)
Prop 2: The edge between node MUL4 and MUL5
            cannot be activated.
   ~F(mul4_active=1 -> X(mul5_active = 1))
Prop 3: The path of FADD cannot be activated.
   ~F (fetch_active = 1 & decode_active = 1
      & fadd1_active = 1 & fadd2_active = 1
      & fadd3_active = 1 & fadd4_active = 1
      & mem_active = 1 & writeback_active = 1)
Prop 4: DIV, FADD4 and MUL7 cannot be
        activated at the same time.
   ~F(div_active = 1 & fadd4_active = 1
      & mul7_active = 1)
```

The *functional coverage* can be defined based on the fault coverage in fault models applied on graph models. As indicated earlier, our framework generates one property for each fault in the fault model. For example, consider a graph model with n nodes, e edges, and p paths, then we need to generate n properties for node fault, e properties for

Algorithm 1: *Property Clustering*

Inputs: i) A set of properties, P
ii) Similarity strategy CS, and threshold W_{th}
Outputs: Clusters consisting of similar properties
Begin
$PropClusters = \phi$;
1. Construct a graph, G where each node is a property.
for each pair of nodes n_i and n_j in G
Weight w_i^j = ComputeSimilarity(n_i, n_j);
if ($w_i^j \geq W_{th}$) then
Create an edge (n_i, n_j) with weight w_i^j.
endif
2. $k = 1$; /* first cluster */
while G is not empty
$Base_k$ = Node with highest edge weight.
$Cluster_k$ = all the nodes connected to $Base_k$.
$G = G$ - $Cluster_k$
$PropClusters = PropClusters \cup Cluster_k$
$k = k + 1$;
endwhile
return *PropertyClusters* ;
End

edge fault, and p properties for path fault. The number of properties required for interaction fault depends on the number of simultaneous interactions. For example, if we allow up to two node interactions, the total number of properties will be $n(n-1)/2$. Due to the overlap between different fault models, some of the generated properties may lead to redundant tests. Therefore, property compaction can be employed to reduce the number of properties without affecting coverage goals [21].

V. Clustering of Similar Properties

Given a set of properties, a clustering method determines how to divide the properties into several groups such that each group contains similar properties that can benefit from each other during test generation. The similarity can be structural or behavioral but the assumption is that there is a significant overlap between the counterexample generation traces involving a set of similar properties.

Algorithm 1 outlines the major steps in property clustering. The first step constructs a property graph[1] where the properties are nodes and edges represent similarity. An edge is added between two properties (nodes) when they are similar. Each edge e_j includes weight information (w_j, $0 \leq w_j \leq 1$) to quantify the similarity. An edge with weight 0 or 1 is not possible since an weight of 0 means no similarity, and an weight of 1 implies same (identical) property. To compute the weight information for each edge we propose four methods: structural, textual, influence, and CNF intersection based similarity. Each method will use a similarity threshold for clustering. In other words, there will be no edge between two properties when the weight value is below certain threshold. The second step determines the clusters based on the base property. The base property is the property (node) with highest weight (summation of weights of all edges connected to that node). The cluster is formed by adding all the adjacent nodes with the base property. All the nodes selected for a cluster are deleted from the property graph for the next iteration. The remainder of this section describes four different ways of computing similarity between two properties.

A. Similarity Based on Structural Overlap

A simple and natural way to cluster properties is to exploit the structural information of the design model and its properties. The intuition is that two structure similar properties will share similar variable assignments (global variables and local variables[2]) in the counterexamples. In fact, a conflict clause is a constraint on the assignment of the variables. Therefore, properties with similar structural information will share a lot of conflict clauses.

As mentioned earlier, in the context of directed test generation, properties are generated based on functional coverage of the design. These properties try to cover different parts of the design (e.g., all computation nodes, various interactions, and so on). Therefore, we can cluster the properties that try to cover a specific functionality or interactions. For example, in an SOC environment, the properties can be clustered based on whether they are related to verifying the processor, coprocessor, field-programmable gate array, memory, bus synchronization, or controllers. Each cluster can be further refined based on structural details of each component. For example, the processor related properties can be further divided based on which execution path they activate such as ALU pipeline, load-store pipeline, and so on.

In the pipelined processor example in Fig. 2, there are four execution pipelines: $IALU$, MUL, $FADD$, and DIV. The corresponding paths are as follows:

1) $\rho_1 = FET \rightarrow DEC \rightarrow IALU \rightarrow MEM \rightarrow WB$;
2) $\rho_2 = FET \rightarrow DEC \rightarrow MUL1 \cdots \rightarrow MUL7 \rightarrow MEM \rightarrow WB$;
3) $\rho_3 = FET \rightarrow DEC \rightarrow FADD1 \cdots \rightarrow FADD4 \rightarrow MEM \rightarrow WB$;
4) $\rho_4 = FET \rightarrow DEC \rightarrow DIV \rightarrow MEM \rightarrow WB$.

Consider two properties $p1 = \sim F(fadd3\_active = 1)$ and property $p2 = \sim F(fadd4\_active = 1)$. They share the same path ρ_3, and the bound of $p1$ is just one smaller than $p2$. So we can cluster them together. Also for the interaction property $p3 = \sim F(fadd4\_active = 1 \ \& \ mul3\_active = 1)$ and $p4 = \sim F(fadd3\_active = 1 \ \& \ mul4\_active = 1)$, the two interactions are related to the same set of paths ρ_2 and ρ_3 and have similar bounds. Therefore, clustering them together is a good choice.

[1]In this paper, we use three different types of graphs in three different purposes. The graph model of the design (or *design graph* in short) is used to model the design. The *implication graph* is used to store the dependence of variable assignments that is used for conflict analysis. The *property graph* models the similarity between properties and used for clustering.

[2]A local variable is defined locally in a node whereas the scope of a global variable is valid across nodes.

B. Similarity Based on Textual Overlap

Another simple way to quantify similarity is to measure the textual differences between two properties. For example, the similarity between $\sim F(a \,\&\, b \,\&\, c)$ and $\sim F(b \,\&\, c \,\&\, d)$ is 67% since they share a common sub-expression consisting of two variables b and c. In this paper, we focus on bounded model checking of invariants (safety properties) such as the property in the form $\sim F(p)$. Informally, $BMC(M, p, k)$ is true means from cycle 0 to cycle k, the property will be false. So the invariant cannot always be true and one counter example will be reported. Because the part $I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$ comes from the design, so for different properties this part is the same. The part $\bigvee_{i=0}^{k} \neg p(s_i)$ usually determines the difference among the properties.

The negative format of each literal in the conflict clause is a false assignment for the logic formula $BMC(M, p, k)$. In fact, the conflict clause can be regarded as a constraint for the variable assignment. Let P and Q be two properties of the model, the properties P, $P \wedge Q$ and $P \vee Q$ can be expanded as follows.

1) $BMC_1(M, P, k)$
$= I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg P(s_i)$.
2) $BMC_2(M, P \wedge Q, k)$
$= I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg (P \wedge Q)(s_i)$
$= I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} (\neg P(s_i) \vee \neg Q(s_i))$.
3) $BMC_3(M, P \vee Q, k)$
$= I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} (\neg P(s_i) \wedge \neg Q(s_i))$.

In the expanded CNFs above, we assume that the same variable in respective expansion has the same meaning. Let A be a partial assignment of the CNF variables that can make the whole CNF false, then $A \not\models BMC_1$ implies $A \not\models BMC_3$, $A \not\models BMC_2$ implies $A \not\models BMC_1$, and $A \not\models BMC_2$ implies $A \not\models BMC_3$. In other words, the conflict clauses of BMC_1 can be forwarded to BMC_3, and conflict clause of BMC_2 can be forwarded to both BMC_1 and BMC_3.

In most existing BMC tools, the variables in the generated CNF file are not well mapped. The conflict clauses of the stronger property cannot be directly forwarded to some weaker properties. For example, some conflict clauses of property $P \wedge Q$ cannot be forwarded to check property $P \vee Q$. However, when properties have the relation of implication, and their textual similarity is high, clustering them together will have a positive effect. If two properties are in the same format and have a significant (more than 50%) textual overlap, the two properties can benefit from each other.

Textual clustering is very fast but it may not be very accurate. For example, the properties $\sim F(a)$ and $\sim F(c)$ have no overlap, however, it is possible that both variables are very closely related in the design model (such as activates the same path), and therefore they are good candidates for clustering. Unfortunately, in the absence of such structural information, pure textual clustering may not generate significant savings in test generation time. Textual clustering is beneficial when information regarding the design or original fault model is not available and/or when there are too many properties.

C. Similarity Based on Influence

An assignment to a global variable determines the state transition of various components in the design (graph) model. For example, in the MIPS pipeline model, when the instruction buffer contains only division instruction, only the components in DIV path will be activated. However, it is time consuming to analyze all the global and local variables of the model since it needs to consider the state transition of each component. Based on the structure of the graph model, we can determine various cause-effect relations. For example, the state change of $MUL6$ will be one clock cycle later than $MUL5$. This means the execution of $MUL5$ has an influence on the execution of $MUL6$. The influence nodes indirectly reflect the assignment of the global variables, since the assignment of global variables is relevant to the variable assignment in the counterexample.

Prior to clustering, we need to figure out the influence node set for each node in the graph model. We can compute the influence node set for each node using depth first search (DFS) algorithm. If there is a path starting from the start node to the current node, then all the nodes on this path are influence nodes for the current node. DFS can explore all the paths (except the paths with loops) from the start node to the current node. For example, the influence node sets for $MUL2$, $FADD3$ and WB are as follows.

1) $Influence(MUL2) = \{FET,\ DEC,\ MUL1,\ MUL2\}$.
2) $Influence(FADD3) = \{FET,\ DEC,\ FADD1,\ FADD2,\ FADD3\}$.
3) $Influence(WB) = \{n|$ n is a node in the MIPS graph model$\}$.

A property corresponds to several nodes (modules) in the graph model. So the influence node set of a property is the union of the influence of all relevant nodes. When comparing the similarity of two properties, we need to compute the intersection of influence sets. For example, the influence set of property $\sim F(mul2\_active = 1 \,\&\, fadd3\_active = 1)$ is $S_1 = \{FET,\ DEC,\ MUL1,\ MUL2,\ FADD1,\ FADD2,\ FADD3\}$ and the influence set for $\sim F(mul3\_active = 1 \,\&\, fadd3\_active = 1)$ is $S_2 = \{FET,\ DEC,\ MUL1,\ MUL2,\ MUL3,\ FADD1,\ FADD2,\ FADD3\}$. The two sets share a large intersection. For set S_1, the similarity with S_2 is $7/7 = 100\%$. For set S_2 the similarity with S_1 is $7/8 = 87.5\%$. Based on our experience, when the overlap of influence sets is larger than 70%, forwarding conflict clauses is beneficial. In this example, S_1 and S_2 can be clustered together.

D. Similarity Based on CNF Intersection

One obvious, but costly, way to determine property similarity for clustering is to compute the intersections of CNF clauses between properties. We can cluster the properties that have a relatively large number of clauses in the intersection. Based on our experience, a threshold of 0.9 is beneficial. In other words, when two properties share at least 90% common clauses, it is beneficial to forward conflict clauses between two instances. This method is very time consuming because it requires $O(n^2)$ intersections for n properties. When n is large, this method is not feasible, because the calculation of intersection of irrelevant properties may waste more time

than actual SAT solution time. Moreover, in certain scenarios, forwarding conflict clauses may not improve the overall test generation time for a cluster, since it may change variable ordering and searching heuristics. CNF based clustering is a good choice when the number of properties is small or when other methods fail to find beneficial clusters.

E. Determination of Base Property

Determination of the base property in a cluster is crucial for success of test generation using learning techniques. The base property is solved first and its conflict clauses are shared between the remaining properties in the cluster. Although, any property in the cluster can be used as the base property for that cluster, our studies have shown that certain properties serve better as base property and thereby generate better overall savings for the cluster. We need to consider two important factors while choosing a base property for a cluster. First, the base property should be able to generate a large number of conflict clauses. In other words, a weak base property may find the satisfiable assignment quickly without making mistakes (generating conflict clauses). In this scenario, the remaining properties have nothing to learn from the base property. Moreover, the SAT checking time for the base property should be relatively small. This will ensure that the overall gain is maximized by reducing the solution time of the properties which takes longer time to solve. None of these requirements can be determined without actually solving them. Based on our experience, we have observed that the following heuristics works well most of the time.

1) Choose a property that has significant variable and/or sub-expression overlap with other properties in the cluster.
2) If bound for each property is known, choose the property whose bound is closest to the remaining properties.
3) Compute intersections for every pair of properties in the cluster, and choose the one that shares the most with the remaining properties.

VI. Test Generation Using Learning Techniques

The basic idea of our test generation approach is to learn from solving one property and share learning (through conflict clauses) for solving the similar properties in the cluster. While solving the first property (base property), the SAT solver may have taken many wrong decisions (lead to conflicts) and therefore needs long time to find a counterexample. Forwarding conflict clauses ensures that these wrong decisions are avoided while solving the similar properties. An important question is whether all the wrong decisions of the first property are relevant to all the other properties in the clusters? Since the properties are similar but not the same, some of the decisions are not relevant. In our approach, we determine the common CNF clauses by computing the intersection of clauses and use this intersection information to exactly identify the conflict clauses that are relevant to solving the respective properties. The remainder of this section describes how to identify relevant conflict clauses and how to use such learning knowledge for test generation.

A. Identification and Reuse of Common Conflict Clauses

Our implementation of relevant conflict clause determination is motivated by the work of [17] which proved that for two sets of CNF clauses C_1 and C_2, and their intersection φ, use of conflict clauses generated from φ when checking C_1 will not affect the satisfiability of the CNF clauses $C_2 \bigcup \varphi$. Therefore, the conflict clauses generated from the intersection when checking the base property can be shared by other properties in the cluster. Strichman [17] suggested an isolation procedure that can isolate the conflict clauses which are deduced solely from the intersection of two CNF clause sets. We have modified the isolation procedure to improve the efficiency of test generation for a cluster of properties. We have modified zChaff [22] SAT solver and used it in our framework. The zChaff provides utilities for implementing incremental satisfiability. For each clause, it uses 32 bits to store a group id to identify the group where this clause belongs. Use of group id allows us to generate the conflict clauses for different properties when checking the base property. If the ith bit of the clause's group id is 1, it implies that the clause is shared by the CNF clauses of property P_i. If the clause of the base property is not shared by any property, the field will be 0.

In our approach, each conflict side clause has a group id which is marked during the preprocessing step or marked during the conflict analysis if it is a conflict clause. The procedure of group id determination of a conflict clause is described in Algorithm 2. This algorithm traces back from the conflicting assignment to a cut such as first unique implication point (UIP) [23] in zChaff. The conflict side will contain all the implications of the variable assignments of the reason side. For UIP, they are implication variable assignments in the same decision level as the conflicting variable assignment which led to the conflict. The group id of the conflict clause is the logical "AND" value of all the group ids of the conflict side clauses. This algorithm can guarantee that if the ith bit of the group id of the conflict clause is 1, then this conflict clause can be forwarded to the ith CNF clause set.

Fig. 3 illustrates how this computation is done. The implication graph belongs to a base property of a cluster. Each clause in this graph is marked with the group id information. Here we use four bits to express the group id. For example, the group id of clause $(x3' + x4')$ is "1010." This means that this clause exists both in CNF clause set 2 and CNF clause set 4. The group id of the conflict clause is the logical "AND" of all conflict side clauses, and the result is 0010. That means, this conflict clause can be forwarded to clause set C_2. Therefore, the use of this conflict clause in solving P_2 will reduce the SAT solving (test generation) time.

B. Test Generation Using Learning Techniques

Algorithm 3 describes our test generation methodology. It accepts a list of clusters where each cluster consists of a set of similar properties. Since one property is used to generate a test, the number of input properties is exactly the same as the number of output tests. The first step generates the CNF clauses for all the properties in each cluster using the design and respective bounds. The second step performs

Algorithm 2: *Determination group ID of conflict clauses*

Inputs: i) Conflicting node N
Outputs: Conflict clause with its group id
Begin
  $Visited = \{N\}$;
  $ConflictAssign = \{\}$;
  $groupID$ = group id of $N's$ antecedent clause;
  while the set *Visited* is not empty
    1. v = RemoveOneElement(*Visited*) ;
    2. *clause* = AntecedentOf(v);
    $groupID = groupID$ *"AND"* group id of *clause*;
    if v is on the conflict side
      3. Put all the nodes of *clause* in implication graph except v to the set *Visited*;
    else
      4. $ConflictAssign = ConflictAssign \cup \{v\}$;
    endif
  endwhile
  5. *ConflictClause* = Logical disjunction of negated assignments of all elements in *ConflictAssign*;
  return *ConflictClause* and *groupID* ;
End

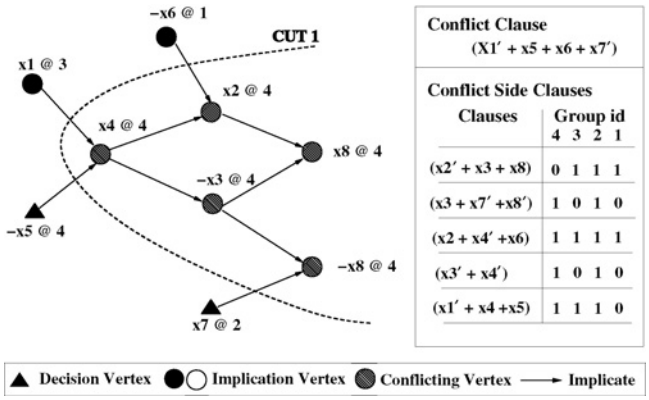

Fig. 3. Example of conflict clause reuse.

name substitution to maximize knowledge sharing. The third step computes the intersection of CNF clauses between the base property and all the remaining properties in the cluster. The first three steps can be omitted, if CNF intersection based clustering is employed. The fourth step marks the clauses in the base property to indicate whether a particular clause is also in the clause set of another property in the cluster. The next step uses a SAT solver to generate the conflict clauses and the counterexample for the base property. Based on the intersection information with the base property, the set of conflict clauses is filtered to identify the relevant ones for solving the remaining properties in step 6. The final step uses the relevant conflict clauses to solve the remaining properties using our approach. The algorithm reports all the generated counterexamples.

We use a simple example to illustrate how Algorithm 3 works. Let us assume that we are generating tests using n properties for a design. The input is a list of m ($m \leq n$)

Algorithm 3: *Test Generation using Learning Techniques*

Inputs: i) Design model, D
  ii) Clusters of similar properties
Output: Tests
Begin
  for each cluster, i, of properties
    Generate CNF for the base property P_1^i, CNF_1^i
    for j is from 2 to the $size_i$ of cluster i
      /* P_j^i is the j^{th} property in the i^{th} cluster */
      1. Generate CNF, $CNF_j^i = BMC(D, P_j^i, bound_j^i)$
      2. Perform name substitution on CNF_j^i
      3. INT_j^i = ComputeIntersection(CNF_1^i, CNF_j^i)
      4. Mark the clauses of CNF_1^i using INT_j^i
    endfor
    /* Generate a counterexample and conflict clauses */
    5. $(ConflictClauses_i, test_1^i)$ = SAT(CNF_1^i)
    $Tests = \{test_1^i\}$
    for j is from 2 to the $size_i$ of cluster i
      /* Find relevant ones for P_j^i from conflict clauses
      6. CC_j^i = Filter ($ConflictClauses_i$, j)
    endfor
    for j is from 2 to the $size_i$ of cluster i
      7. $test_j^i$ = SAT($CNF_j^i \bigcup CC_j^i$)
      $Tests = Tests \cup test_j^i$
    endfor
  endfor
  return *Tests*
End

clusters based on property similarities. Each cluster can have different numbers of properties. In the worst case, each cluster can have only one property which will be verified normally. However, this scenario is rare in practice since a typical design uses thousands of properties for directed test generation and majority of them share significant parts of the design functionality. For ease of illustration, let us assume that there is a cluster with three similar properties, $\{P_1, P_2, P_3\}$. Let us further assume that the second step selects P_1 as the base property using the method described in Section V. The fourth step computes intersection of CNF clauses of P_1 with P_2, and P_1 with P_3. This information is used to filter conflict clauses (generated while solving P_1) relevant for P_2 and P_3 in step 6. The last step adds the relevant conflict clauses while solving the respective properties to reduce the test generation time.

VII. Experiments

We have applied our test generation methodology for validation of various software and hardware designs. In this section, we present two case studies: a very long instruction word (VLIW) implementation of the MIPS architecture, and a stock exchange system. Both experiments were performed on a Linux PC using 2.0 GHz Core 2 Duo central processing unit with 1 GB RAM. In our experiments, we used the NuSMV [24] as our BMC tool to generate the CNF clauses (in the DIMACS format) for the design and properties. We developed

TABLE I
PROPERTY CLUSTERING AND VERIFICATION FOR MIPS PROCESSOR

Methods	Structure	Textual	Influence	Intersection
# of Clusters	16	32	27	17
Clustering (s)	0.24	0.06	0.22	187.90
Base (s)	169.09	436.60	322.44	324.18
Original (s)	3105.98	2830.13	2918.56	2999.16
Improved (s)	788.09	442.53	431.92	239.28
Speedup	**3.42**	**3.72**	**4.33**	**5.90 (4.42)**

TABLE II
PROPERTY CLUSTERING AND VERIFICATION FOR OSES

Methods	Structure	Textual	Influence	Intersection
# of Clusters	18	9	12	13
Clustering (s)	0.05	0.01	0.05	42.77
Base (s)	562.05	142.81	277.42	313.73
Original (s)	1557.11	2017.11	2034.05	1820.53
Improved (s)	377.15	784.16	668.72	437.98
Speedup	**2.26**	**2.33**	**2.44**	**2.84 (2.69)**

the tool *PropertyCluster* which accepts the graph model, the coverage criteria and the clustering strategies as inputs. This tool generates the required properties (using different coverage criteria presented in Section IV) and clusters them using the clustering strategies proposed in Section V. We also modified zChaff [22] to incorporate our techniques including name substitution, clause intersection, and constraint sharing. The modified zChaff can accept a cluster of properties and check them together. The result of our approach is compared with the original zChaff that does not use any clustering techniques.

A. A VLIW MIPS Processor

We applied our methodology on a single-issue MIPS [19] architecture. Fig. 2 shows the simplified version of the VLIW MIPS architecture. It has five pipeline stages: fetch, decode, execute, memory (MEM), and writeback. The *execute* stage has four parallel execution paths: integer ALU, 7 stage multiplier (MUL1–MUL7), four stage floating-point adder (FADD1–FADD4), and multicycle divider (DIV). The oval boxes represent units and dashed boxes represent storages. The solid lines represent instruction-transfer paths and dotted lines represent data-transfer paths. The *PropertyCluster* generated 171 properties using the node coverage, 2-interaction coverage, and the path coverage criteria.

Table I compares the four clustering techniques. The first row shows our proposed clustering methods. The second row indicates the number of clusters using the respective clustering methods, and the third row shows the corresponding clustering time (in seconds). The fourth row presents the test generation time for the base property. The original time refers to traditional (no clustering) verification time for all the properties, excluding the base property. The sixth row presents the verification time for all the properties except the base property using the respective clustering methods. The speedup is computed using the formula (base time + original time)/(clustering time + base time + improved time). For the first three clustering methods, the clustering is very fast and the associated cost (time) is negligible. However, for the intersection-based clustering, the intersection time is longer compared to other three methods and is not negligible. Therefore, for intersection-based clustering, we provide speedup values for both scenarios—without considering clustering time (the first number) as well as with clustering time (the number in parenthesis).

It is important to note that intersection-based clustering is most beneficial for reducing overall test generation time. However, the clustering overhead is much more than other strategies. When a large number of complex properties are involved, the intersection overhead may become prohibitively large. In such cases, influence-based clustering is most beneficial. Interestingly, textual clustering consumes least amount of clustering time but generates better results than structure-based clustering. When detailed information about the design is not available, textual clustering is most beneficial.

B. A Stock Exchange System

This section presents the test generation results of an online stock exchange system (OSES) software that can process three scenarios: accept, check, and execute the customer's orders (market order and limit order). The specification uses the unified modeling language (UML) activity diagram as its behavior specification. The extracted graph model has 27 nodes (activities) and 29 edges. To validate various exchange scenarios, we generate 51 properties based on the path fault model. We applied the clustering methods discussed in Section V on all the properties to generate the tests.

Table II summarizes the test generation results using four clustering methods where two to three times improvement is achieved. It is important to note that the results for OSES are consistent with the results for MIPS in Table I. As Table II shows, intersection-based clustering is most beneficial for reducing overall test generation time. However, when clustering overhead is prohibitively large, influence-based clustering is beneficial. Similarly, when detailed information about the design is not available, textual clustering is the best choice.

On both case studies (MIPS and OSES) our approach demonstrated three to five times improvement in overall test generation time using efficient integration of property clustering and learning techniques.

VIII. CONCLUSION

The feasibility of existing model checking based approaches for directed test generation is limited due to capacity restrictions of the automated tools. This paper addressed the test generation complexity by exploiting the commonalities between automatically generated properties. Our primary contribution is the development of an automated framework that can enable coverage-directed property generation, property clustering based on similarity, and efficient test generation by sharing knowledge across multiple properties. We developed four efficient property clustering techniques and compared them to evaluate their suitability on different test generation scenarios. We also developed a number of conceptually simple,

but extremely effective, techniques including name substitution and selective forwarding of learned conflict clauses to reduce the overall test generation time. Our experimental results using both hardware and software designs demonstrated a drastic reduction (three to five times) in overall test generation time.

This paper is focused on efficient directed test generation using property clustering and learning techniques. However, the proposed framework is not restricted for test generation (property falsification). Since the essence of our methodology is to avoid repetitive validation efforts, we believe that it will also be beneficial in other verification scenarios that involve multiple properties. We plan to extend our clustering and learning techniques for standard property verification, as well as assertion-based verification.

REFERENCES

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Proc. Tools Algorithms Construction Anal. Syst.*, 1999, pp. 193–207.

[2] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Adv. Comput.*, vol. 58, no. 3, pp. 118–149, 2003.

[3] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proc. 38th Design Autom. Conf.*, 2001, pp. 530–535.

[4] J. Marques-Silva and K. Sakallah, "Grasp: A search algorithm for propositional satisfiability," *IEEE Trans. Comput.*, vol. 48, no. 5, pp. 506–521, May 1999.

[5] M. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in SAT-based formal verification," *Int. J. Softw. Tools Technol. Transfer*, vol. 7, no. 2, pp. 156–173, 2005.

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

[7] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *ACM SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 146–162, 2003.

[8] P. Mishra and N. Dutt, "Specification-driven directed test generation for validation of pipelined processors," *ACM Trans. Design Autom. Electron. Syst.*, vol. 13, no. 3, article no. 42, Jul. 2008.

[9] N. Amla, X. Du, A. Kuehlmann, R. Kurshan, and K. McMillan, "An analysis of SAT-based model checking techniques in an industrial environment," in *Proc. Correct Hardw. Design Verification Methods*, 2005, pp. 254–268.

[10] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi, "Benefits of bounded model checking at an industrial setting," in *Proc. Comput. Aided Verification*, 2001, pp. 436–453.

[11] E. M. Clarke, A. Gupta, J. H. Kukula, and O. Strichman, "SAT based abstraction-refinement using ILP and machine learning techniques," in *Proc. Comput.-Aided Verification*, 2002, pp. 265–279.

[12] X. Cheng and M. S. Hsiao, "Simulation-directed invariant mining for software verification," in *Proc. Design Autom. Test Europe*, 2008, pp. 682–687.

[13] H. Jin and F. Somenzi, "An incremental algorithm to check satisfiability for bounded model checking," in *Proc. Int. Workshop Bounded Model Checking (BMC)*, vol. 119. 2005, pp. 51–65.

[14] J. Whittemore, J. Kim, and K. Sakallah, "SATIRE: A new incremental satisfiability engine," in *Proc. Design Autom. Conf.*, 2001, pp. 542–545.

[15] L. Zhang, M. Prasad, and M. Hsiao, "Incremental deductive and inductive reasoning for SAT-based bounded model checking," in *Proc. Int. Conf. Comput.-Aided Design*, 2004, pp. 502–509.

[16] N. Eén and N. Sörensson, "Temporal induction by incremental sat solving," in *Proc. Int. Workshop Bounded Model Checking*, vol. 89. 2003, pp. 541–638.

[17] O. Strichman, "Pruning techniques for the SAT-based bounded model checking problem," in *Proc. Correct Hardw. Design Verification Methods*, 2001, pp. 58–70.

[18] K. Chandrasekar and M. S. Hsiao, "Integration of learning techniques into incremental satisfiability for efficient path-delay fault test generation," in *Proc. Design Autom. Test Europe*, 2005, pp. 1002–1007.

[19] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Boston, MA: Morgan Kaufmann Publishers, 2006.

[20] P. Mishra and N. Dutt, *Processor Description Languages: Applications and Methodologies*. Boston, MA: Morgan Kaufmann Publishers, 2008.

[21] H. Koo and P. Mishra, "Specification-based compaction of directed tests for functional validation of pipelined processors," in *Proc. Int. Symp. Hardw./Softw. Codesign Syst. Synthesis*, 2008, pp. 137–142.

[22] *zChaff* [Online]. Available: http://www.princeton.edu/˜chaff/zchaff.html

[23] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," in *Proc. Int. Conf. Comput.-Aided Design*, 2001, pp. 279–285.

[24] *NuSMV* [Online]. Available: http://nusmv.irst.itc.it/

Mingsong Chen (S'08) received the B.S. and M.E. degrees in computer science from the Department of Computer Science and Technology, Nanjing University, Nanjing, China, in 2003 and 2006, respectively. He is currently pursuing the Ph.D. degree in computer engineering from the Department of Computer and Information Science and Engineering, University of Florida, Gainesville.

His research interests include the area of design automation of embedded systems, design verification, and software engineering.

Prabhat Mishra (S'00–M'04–SM'08) received the B.E. degree from Jadavpur University, Jadavpur, India, the M.Tech. degree from the Indian Institute of Technology Kharagpur, Kharagpur, India, and the Ph.D. degree from the University of California, Irvine, all in computer science in 1994, 1996, and 2004, respectively.

He was with various semiconductor and design automation companies including Texas Instruments, Bangalore, India; Synopsys, Bangalore, India; Intel, Santa Clara, CA; and Freescale, Austin, TX. He is currently an Assistant Professor with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville. His research interests include design automation of embedded systems, reconfigurable architectures, and functional verification.

Dr. Mishra currently serves as the General Chair of the IEEE High Level Design Validation and Test Workshop, the Information Director of the Association for Computing Machinery (ACM) Transactions on Design Automation of Electronic Systems, and as a Program/Organizing Committee Member of several ACM and IEEE conferences. His research has been recognized by various awards, including an National Science Foundation Faculty Early Career Development Award in 2008, the European Design and Automation Association Outstanding Dissertation Award in 2005, and the International Conference on Hardware-Software Codesign and System Synthesis Best Paper Award in 2003.