# Efficient Techniques for Directed Test Generation using Incremental Satisfiability*

Prabhat Mishra and Mingsong Chen

Department of Computer and Information Science and Engineering
University of Florida, Gainesville FL 32611-6120, USA
{*prabhat, mchen*}*@cise.ufl.edu*

## Abstract

*Functional validation is a major bottleneck in the current SOC design methodology. While specification-based validation techniques have proposed several promising ideas, the time and resources required for directed test generation can be prohibitively large. This paper presents an efficient test generation methodology using incremental satisfiability. The existing researches have used incremental SAT to improve counterexample (test) generation involving only one property with different bounds. This paper is the first attempt to utilize incremental satisfiability in directed test generation involving multiple properties. The contribution of this paper is a novel methodology to share learning across multiple properties by developing efficient techniques for property clustering, name substitution, and selective forwarding of conflict clauses. Our experimental results using both software and hardware benchmarks demonstrate that our approach can drastically (on average four times) reduce the overall test generation time.*

## 1 Introduction

Functional verification is a major bottleneck in System-on-Chip (SOC) design due to the combined effects of increasing design complexity and decreasing time-to-market. Simulation-based validation is the most widely used form of SOC verification using functional test programs. There are three types of test generation techniques: random, constrained-random, and directed. The directed tests can reduce overall validation effort since shorter tests can obtain the same coverage goal compared to the random tests. However, directed test generation is mostly performed by human intervention. Due to manual development, it is infeasible to generate all directed tests to achieve a comprehensive coverage goal. Automatic test generation is the alternative to address this problem.

Test generation using model checking is one of the most promising approaches due to its capability of automatic test generation. However, it is unsuitable for large designs due to the state space explosion problem. SAT-based bounded model checking (BMC) restricts search space that is reachable from initial states within a fixed number ($k$) of transitions, called *bound*. After unrolling the model of design $k$ times, the BMC problem is converted into a propositional satisfiability (SAT) problem. SAT solver is used to find a satisfiable assignment of variables that is converted into a counterexample. If the *bound* is known in advance, SAT-based BMC is typically more effective for counterexample (test) generation because search for counterexample is faster and SAT capacity reaches beyond BDD capacity [1]. The effectiveness of BMC can be further improved by observing that the search for counterexamples of increasing lengths is translated into a sequence of SAT checks. Therefore, it is possible to exploit the similarity of these SAT instances by forwarding clauses learned during conflict analysis from one instance to the next.

Incremental SAT-based BMC is very promising to reduce the test generation complexity but all the existing approaches are applicable for a test generation scenario consisting of one design and only one property (with varying bounds). This paper proposes a novel methodology to exploit incremental SAT in the context of test generation involving one design and multiple properties. The basic idea of our approach is to reuse the learning of one test generation instance for other related test generation scenarios. It is important to note that the design remains the same for all test generation scenarios. Although, each test generation instance requires a different property, several properties related to testing specific functionalities are similar or have a significant overlap. A major challenge in implementing this idea is to identify the property similarities and perform efficient clustering to share learning and thereby reduce the test generation time. To enable the knowledge sharing across multiple properties, we have developed a number of efficient techniques for property clustering, name substitution, and selective forwarding of conflict clauses. The contribution of this paper is the development of a number of conceptually simple, but very effective, techniques to generate drastic reduction in directed test generation time.

The rest of the paper is organized as follows. Section 2 presents related work addressing test generation approaches. Section 3 presents our test generation methodology using incremental satisfiability. Section 4 presents the experimental results. Finally, Section 5 concludes the paper.

## 2 Related Work

Model checking [4] has been successfully used in software and hardware verification as the test generation engine [3, 18].

*This work was partially supported by grants from Intel Corporation and NSF CAREER award 0746261.

Model of design is applied to a model checker along with negated temporal logic properties to exploit falsification capability of model checking. However, traditional model checking does not scale well due to the state explosion problem. Biere et al. [2] introduced bounded model checking (BMC) combined with satisfiability solving. The recent developments in SAT-based BMC techniques have been presented in [15]. 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 since SAT solvers [14] do not require exponential space, and searching counterexample in an arbitrary order consumes much less memory than breadth first search in model checking. The performance of bounded and unbounded algorithms was analyzed on a set of industrial benchmarks in [16]. The capacity increase of BMC techniques has become attractive for industrial use. Intel study [5] showed that BMC performs better over unbounded model checking for real designs. The efficiency of test generation can be further improved by employing incremental SAT solving.

Incremental SAT solvers [6, 12, 17] try to leverage the similarity between the elements of a sequence of SAT instances; most do so by re-utilizing conflict clauses, though when many closely related instances must be solved, caching solutions [11] and incremental translation [13] can also be effective. If a SAT instance is obtained from another by adding some clauses (as in [7]), then all conflict clauses of the first can be forwarded to the second. This is correct because the second instance implies the first, which in turn implies all its (conflict) clauses. Therefore, when clauses are only added through the sequence of instances, there is no need to screen conflict clauses to determine which ones can be forwarded. This, on the other hand, is necessary when arbitrary clauses are both added and subtracted to create a new instance. A common approach to such general case is to have the incremental SAT solver keep track of whether a conflict clause depends on some removed clauses. The approach of [12] is to record, for each conflict clause, the clauses that made up the corresponding implication graph. This approach does not require any prior knowledge of the subsequent SAT instances to be solved incrementally, and does not restrict the changes possible from one instance to the next; however, keeping track of dependencies may be expensive. Strichman [17] was the first to observe that in BMC some clauses are known to survive through all instances in the sequence. A formula passed by BMC to the SAT solver contains clauses that describe the transition relation of the model unrolled a number of times. These clauses are not discarded when the length of the counterexample is increased. Hence, a conflict clause that depends only on them can be forwarded.

To the best of our knowledge, all the existing approaches exploit incremental satisfiability to improve the test (counterexample) generation time involving only one property with different bounds. Our approach is the first attempt at utilizing incremental satisfiability across multiple properties in the context of directed test generation for validation of SOC designs.

## 3 Test Generation using Incremental SAT

The goal of our approach is to reduce the overall functional validation effort by reducing the test generation time for directed tests. We assume that the designer have developed a set of properties (one property to generate one directed test) based on a specific fault model. For example, a pipelined processor with $n$ functional units needs $n(n-1)/2$ properties to activate all 2-unit interactions. Based on the structure of the design and the fault model, the generated properties can be clustered using functional as well as structural similarity e.g., all properties related to a particular execution path can be placed in a cluster. The basic idea 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, the SAT solver may have taken many wrong decisions (lead to conflicts) and therefore took long time to find a counterexample. By forwarding conflict clauses, we are ensuring that these wrong decisions are avoided while solving the similar properties. An important question is whether all the wrong decisions are relevant to all the properties in the clusters? Since the properties are similar but not the same, all the decisions are not relevant. Therefore, adding all the conflict clauses while solving the similar properties may increase the solution time. In our approach, we determine the common CNF (conjunctive normal form) 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.

This paper focuses on test generation for safety properties. In this context, we are interested in finding a counterexample for each property. We assume that the bound is pre-determined and given as input to our method. Determination of bound is intractable in general. However, in the context of directed test generation, it is possible to determine the bound based on the structure of the design and the associated property.

Algorithm 1 describes our test generation methodology. It accepts a design and a set of properties as inputs and generates the testcases. Since one property is used to generate a testcase, the number of input properties is exactly same as the number of output testcases. The first step is to partition the set of properties into different clusters based on their similarity in terms of both structure and behavior. The second step is to select the base property who has the potential to generate maximum overall savings for the cluster by sharing learned conflict clauses. The third step computes the CNF clauses for all the properties in each cluster using the design and the respective bound. The fourth step performs name substitution to maximize knowledge sharing. The fifth step computes the intersection of CNF clauses between the base property and all the other properties in the cluster. The sixth 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 marking information includes the identifier of the property (or properties) with which the clause is identical. The next step uses an existing SAT solver to generate the conflict clauses and the counterexample (test) 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 other properties in step 8. The final step uses the relevant conflict clauses to solve the remaining properties using our approach. The algorithm reports all the generated counterexamples (tests).

**Algorithm 1**: *Test Generation using Incremental Satisfiability*
**Inputs**: i) Design, $D$
ii) Properties, $P$, and their respective satisfiable bounds
**Outputs**: Testcases
Begin
**1.** Cluster the properties based on similarity
**for** each cluster, $i$, of properties
**2.** Select the base property $P_1^i$ and generate CNF, $CNF_1^i$
**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 */
**3.** Generate CNF, $CNF_j^i = BMC(D, P_j^i, bound_j^i)$
**4.** Perform name substitution on $CNF_j^i$
**5.** $INT_j^i$ = ComputeIntersection($CNF_1^i$, $CNF_j^i$)
**6.** Mark the clauses of $CNF_1^i$ using $INT_j^i$
**endfor**
/* Generate the counterexample and record conflict clauses */
**7.** $(ConflictClauses_i, test_1^i)$ = SAT($CNF_1^i$)
$Testcases = \{test_1^i\}$
**for** $j$ is from 2 to the $size_i$ of cluster $i$
/* Find relevant ones for $P_j^i$ from conflict clauses
**8.** $CC_j^i$ = Filter ($ConflictClauses_i$, $j$)
**endfor**
**for** $j$ is from 2 to the $size_i$ of cluster $i$
**9.** $test_j^i$ = SAT($CNF_j^i \bigcup CC_j^i$)
$Testcases = Testcases \cup test_j^i$
**endfor**
**endfor**
**return** *Testcases*
End

We use a simple example to illustrate how Algorithm 1 works. Let us assume that we are interested in generating tests using $n$ properties for a design. The first step divides the properties into $m$ ($m \leq n$) clusters based on property similarities. Each cluster can have different number of properties. In the worst case, each cluster can have only one property which is not suitable for test generation using incremental satisfiability. 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 3.1. 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 8. The last step adds the relevant conflict clauses while solving the respective properties to reduce the test generation time. The remainder of this section describes three important steps in our approach: property clustering, computation of intersections, and identification of relevant conflict clauses.

### 3.1 Clustering of Similar Properties

One obvious, but costly, way to determine property similarity for clustering is to compute the intersection of CNF clauses between properties. We can cluster the properties that have a relatively large number of clauses in the intersection. This is a very time consuming step because it requires $n(n-1)/2$ intersections for $n$ properties. A simple and natural way to cluster properties is to exploit the structural and behavioral information of the properties. As mentioned earlier, in the context of directed test generation, properties are generated based on a set of fault models to obtain a functional coverage goal. Each fault model tries to cover different parts of the design (e.g., all computation nodes, execution paths, various interactions, etc.). Therefore, we can cluster the properties that try to cover a specific part of the design using the same fault model. For example, in an SOC environment, the properties can be clustered based on whether they are related to verifying the processor, coprocessor, FPGA, 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 what execution path they cover such as ALU pipeline, load-store pipeline etc.

Once clustering is completed, we need to determine the base property of the cluster. In our approach, the base property is solved first and its conflict clauses are shared between the remaining properties. 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 generates maximum 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. Secondly, 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. In fact, 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.

- Choose the property with the smallest bound that have significant variable and/or sub-expression overlap with the rest of the properties in the cluster.

- If the property bounds in the cluster are same, choose the property with the smallest number of variables that have significant variable and/or sub-expression overlap with the rest of the properties in the cluster.

### 3.2 Name Substitution for Computation of Intersections

Name substitution is an important preprocessing step in Algorithm 1. Currently there are a few BMC tools that can support the name mapping from the variables of the CNF clauses and the names in the model of the unrolled design. So the variables of the CNF clauses of two different properties may not have any name correspondence. In other words, the same variable in two properties may have different name in their respective CNF clauses. Therefore, without name substitution (mapping), it will miss the structural similarity. As a result, the computed intersection will be small and will adversely affect the sharing of learned conflict clauses. Our experimental studies have shown that the improvement in test generation time without using name substitution is negligibly small due to very small number of clauses being forwarded as a result of small number of clauses in the intersection. Since the properties are similar and the design is exactly the same, the size of the intersection is very large when our name substitution method is employed.

Our framework uses zChaff SAT solver which accepts the input in the DIMACS format. The input DIMACS file for each property provides the name mapping from the CNF variable to the unrolled design. The following example shows that the variable 8 is used in CNF to refer to the $7^{th}$ bit of variable *var* in the design specification at time step 1. This can also be written as, $8 => var[6]\_1$.

```
c 8 = V1_var[6]
```

Given two DIMACS files $f1$ and $f2$ for two properties $P_1$ and $P_2$ respectively, the name substitution is a procedure that changes the names of clause variables of $f2$ using the name mapping defined in $f1$. Figure 1 shows an example for name substitution. Before the name substitution, the intersection $(f1 \cap f2)$ is empty. However, after the substitution, there are two common clauses in the intersection $(f1 \cap f2')$. The complexity of both name substitution and computation of intersection is linear (using hash table) to the size of the DIMACS file of the properties. Therefore, the time required for name substitution and intersection computation is negligible compared to the SAT solving time for the complex properties.

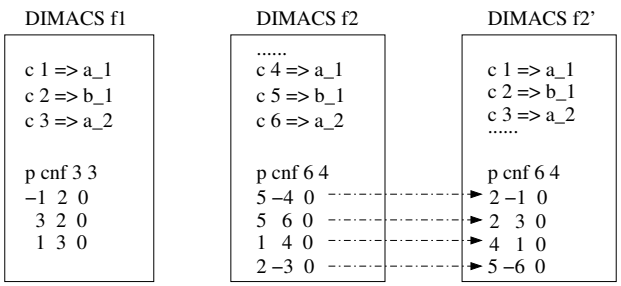


**Figure 1. An example of name substitution**

It is important to note that the same variable at different time steps can be assigned a different number. Therefore, the name mapping (substitution) method needs to consider the same variable at different time steps in the CNF clauses of the same property as well as in the CNF clauses for the different properties in the same cluster. Moreover, the name mapping routine needs to remap some of the variables in the CNF clauses. For example, when the variable 4 in file $f2$ (in Figure 1) is replaced with the variable 1 (in $f2'$), the name mapping routine needs to remap the original variable 1 in file $f2'$ to a different variable.

### 3.3 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 set of CNF clauses $C_1$ and $C_2$, and their intersection φ, using the conflict clauses generated from the 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 set. We have modified the isolation procedure to improve the efficiency of test generation for a cluster of properties. We have modified zChaff [9] and use it as the SAT solver 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 $i^{th}$ bit of the 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.

Assume that there are $k$ properties in a cluster with $P_1$ as the base property. Therefore, there are $k$ sets of clauses with $C_1$ as the base set (CNF clauses for $P_1$), and $C_2, C_3, ..., C_k$ are $k-1$ similar sets with $C_1$. We use the following steps to calculate the conflict clauses for $C_2, C_3, ..., C_k$ when solving $C_1$.

1. During preprocessing, for each clause in $C_1$, if this clause exists in $C_i (2 \leq i \leq k)$, then mark the $i^{th}$ bit of $C_1$'s group id 1.

2. When one conflict clause is encountered during the checking of the base property, collect all the group ids of the clauses which leads to the conflict. The group id of the conflict clause is the logical "OR" of these group ids.

3. For each conflict clause, if the $i^{th}$ bit of the group id is 1, then this conflict clause can be shared by $C_{i+1}$.

Figure 2 illustrates how this computation is done using an example implication graph. The implication graph is a directed acyclic graph where each vertex represents an assignment of the variable and each edge implies that all the in-edges implicate the assignment of the vertex. For example, $x4@4$ means variable $x4$ is assigned value 1 at decision level 4. The graph has a clause $(x1' + x4 + x5)$, we call it the *antecedent* clause of $x4$ i.e.,

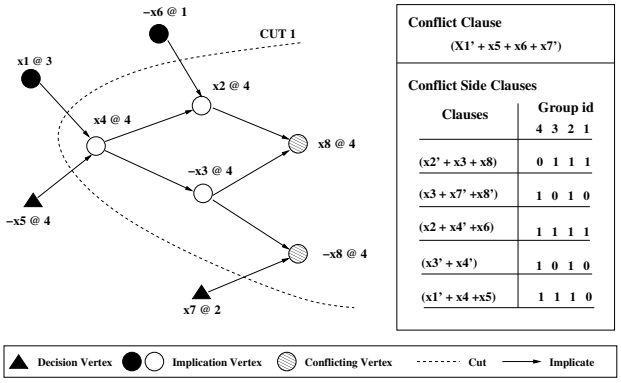


**Figure 2. An example of implication graph**

the assignments $x1 = 1$ and $x5 = 0$ imply $x4 = 1$. Only the implication vertex (non-decision vertex) has an antecedent clause. The conflict exists when in the implication graph, one variable is assigned both value 0 and 1. So the SAT solver will analyze the conflict and find the reason expressed by the 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 Figure 2, $CUT1$ is a cut that divide the implication graph into two parts. We list the conflict clause on the reason side and all other clauses on the conflict side. In our approach, each conflict side clause has a group id which is marked during the preprocessing step. So the group id of the conflict clause is the logical "OR" value of all the group ids of the conflict side clauses. For example, in Figure 2, we use four bits to express the group id, and the group id of the conflict clause is 0010. In other words, this conflict clause can only 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.

# 4 Experiments

We have applied our test generation methodology for validation of various software and hardware designs. In this section, we present two case studies from two different domains: a stock exchange system, and a VLIW implementation of the MIPS architecture. Both experiments were performed on a Linux PC using 2.0GHz Core 2 Duo CPU with 1 GB RAM. In our experiments, we used the NuSMV [8] as our BMC tool to generate the CNF clauses (in the DIMACS format) for the design and properties. We modified zChaff [9] to integrate our methods for name substitution, clause intersection and constraint sharing described in Section 3.

## 4.1 A Stock Exchange System

The purpose of the on-line stock exchange system (OSES) is to process three scenarios: accept, check and execute the customer's orders (market order and limit order). The system uses the UML activity diagram as its behavior specification and JAVA based implementation. The UML specification has 27 activities, 29 transitions and 18 key paths. We translate the specification into the NuSMV input and generate the corresponding test case based on various functional coverage criteria. We group the properties into five clusters. The first cluster consists of $\{Path_1, Path_2\}$ with $Path_1$ as the base property in the cluster. Similarly, each of the remaining clusters consists of four properties with the first one as the base property.

**Table 1. Test Generation for Stock Exchange System**

| Properties (Tests) | Preproc. Time (s) | zChaff [14] (sec) | Our Method (sec) | Improv. Factor |
|---|---|---|---|---|
| $Path_1$ | Base | 0.37 | 0.37 | 1.00 |
| $Path_2$ | 3.79 | 59.45 | 4.06 | 14.66 |
| $Path_3$ | Base | 2.82 | 2.82 | 1.00 |
| $Path_4$ | 4.16 | 2.89 | 0.54 | 5.35 |
| $Path_5$ | 4.09 | 29.67 | 4.04 | 7.34 |
| $Path_6$ | 3.73 | 42.75 | 6.28 | 6.81 |
| $Path_7$ | Base | 0.44 | 0.44 | 1.00 |
| $Path_8$ | 4.14 | 7.36 | 0.30 | 24.86 |
| $Path_9$ | 3.88 | 113.70 | 33.23 | 3.42 |
| $Path_{10}$ | 3.79 | 40.41 | 6.53 | 6.19 |
| $Path_{11}$ | Base | 4.58 | 4.58 | 1.00 |
| $Path_{12}$ | 4.24 | 13.04 | 1.20 | 10.91 |
| $Path_{13}$ | 4.44 | 23.74 | 14.60 | 1.63 |
| $Path_{14}$ | 4.02 | 102.76 | 31.42 | 3.27 |
| $Path_{15}$ | Base | 0.25 | 0.25 | 1.00 |
| $Path_{16}$ | 4.36 | 64.04 | 1.26 | 50.66 |
| $Path_{17}$ | 4.38 | 185.03 | 5.05 | 36.62 |
| $Path_{18}$ | 4.02 | 176.77 | 68.78 | 2.57 |
| **Average** | **4.08** | **48.33** | **10.32** | **4.68** |

Table 1 shows the results involving all the 18 properties of key paths. The first column indicates the properties used for test generation. The second column indicates the preprocessing time that includes the time for both name substitution and computation of clause intersection. This column is not applicable for the base property since the base property is solved using the existing approach. The third and fourth columns indicate the time (in seconds) required to generate the counterexample (test) by zChaff [14] and our approach, respectively. The last column indicates the *improvement factor*[1] in test generation time. The last row in the table presents the averages of all the entries. Our approach can produce almost five times improvement compared to zChaff, a popular CNF SAT solver. Our approach should be used in the test generation scenarios where SAT takes a long time to find a counterexample. As a result, the cost (preprocessing time) will be negligible compared to the savings in test generation time.

## 4.2 A VLIW MIPS Processor

We applied our methodology on a single-issue MIPS [10] architecture. Figure 3 shows the simplified version of the VLIW MIPS architecture. It has five pipeline stages: fetch, decode, execute, memory (MEM), and writeback. The *execute* stage

[1]The improvement factor is computed as the ratio between the third and fourth column entries.

has four parallel execution paths: integer ALU, 7 stage multiplier (MUL1 - MUL7), four stage floating-point adder (FADD1 - FADD4), and multi-cycle 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.

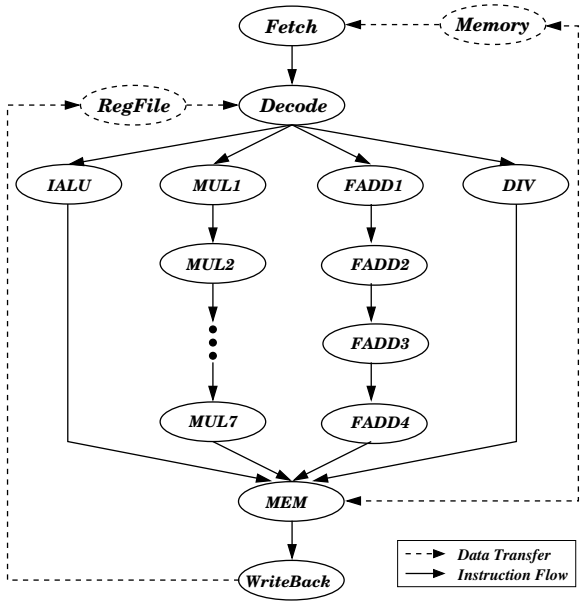


**Figure 3. The VLIW MIPS architecture**

We translated the specification into the NuSMV input and generated the required directed tests based on various path coverage criteria. Table 2 shows the results. Each row presents the averages of all the properties in that cluster. The first column indicates the four clusters corresponding to the four execution paths in Figure 3. The second column represents the average number of CNF clauses for each property in that cluster. The third column shows the average number of clauses in the intersection. The fourth and fifth columns indicate the time (in seconds) required to generate the counterexample (test) by zChaff and our approach, respectively. Our approach is able to improve the test generation time on average by four times.

**Table 2. Test Generation for MIPS Processor**

| Clusters | CNF Clauses | Intersec. Size | zChaff [9] | Our Method | Improv. Factor |
|---|---|---|---|---|---|
| $CL_{ALU}$ | 460994 | 457168 | 19.35 | 5.10 | 3.79 |
| $CL_{FADD}$ | 592119 | 67894 | 61.61 | 42.46 | 1.45 |
| $CL_{MUL}$ | 854386 | 522283 | 718.85 | 159.21 | 4.51 |
| $CL_{DIV}$ | 526517 | 457160 | 35.07 | 8.19 | 4.28 |
| **Average** | **608504** | **376126** | **208.72** | **53.74** | **3.88** |

## 5 Conclusions

Directed test vectors can reduce overall validation effort of both hardware and software designs since shorter tests can obtain the same coverage goal compared to the random tests. The applicability of the existing 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 a set of similar properties using incremental satisfiability. The existing incremental SAT approaches are applicable only on a single property and it utilizes the learning between various bounds of the same property. To the best of our knowledge, our work is the first attempt to share learning across multiple properties. To enable knowledge sharing across multiple properties, we have developed a number of conceptually simple, but extremely effective, techniques including property clustering, name substitution, and selective forwarding of learned conflict clauses. Our experimental results using both hardware and software designs demonstrated on average four times reduction in directed test generation time.

## References


[1] A. Biere, A. Cimatti, and E. M. Clarke. Bounded model checking. *Advances in Computers*, 58, 2003.

[2] A. Biere, A. Cimatti, E. Clarke and Y. Zhu. Symbolic model checking without BDDs. *TACAS*, 193–207, 1999.

[3] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *ACM SIGSOFT Software Engineering Notes*, volume 24, 146–162, 1999.

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

[5] F. Copty et al. Benefits of bounded model checking at an industrial setting. *CAV*, 436–453, 2001.

[6] H. Jin and F. Somenzi. An incremental algorithm to check satisfiability for bounded model checking. *BMC*, 51–65, 2004.

[7] J. Hooker. Solving the incremental satisfiability problem. *Journal of Logic Programming*, 15(12):177–186, 1993.

[8] http://nusmv.irst.itc.it/. *NuSMV*.

[9] http://www.princeton.edu/ chaff/zchaff.html. *zChaff*.

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

[11] J. Kim et al. On solving stack-based incremental satisfiability problems. *ICCD*, 379–382, 2000.

[12] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. *DAC*, 542–545, 2001.

[13] M. Benedetti and S. Bernardini. Incremental compilation-to-sat procedures. *SAT*, 2004.

[14] M. Moskewicz et al. Chaff: Engineering an efficient SAT solver. *DAC*, pages 530–535, 2001.

[15] M. Prasad, A. Biere and A. Gupta. A survey of recent advances in SAT-based formal verification. *STTT*, 7(2):156–173, 2005.

[16] N. Amla et al. An analysis of SAT-based model checking techniques in an industrial environment. CHARME, 254–268, 2005.

[17] O. Strichman. Pruning techniques for the sat-based bounded model checking problem. *CHARME*, 58–70, 2001.

[18] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. *DATE*, 182–187, 2004.