# Synchronized Generation of Directed Tests using Satisfiability Solving

Xiaoke Qin, Mingsong Chen and Prabhat Mishra

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

## Abstract

*Directed test generation is important for the functional verification of complex system-on-chip designs. SAT based bounded model checking is promising for counterexample generation which can be used in directed testing. Existing research has explored two directions to accelerate the SAT solving process: learning during solving of one property with different bounds, or solving multiple properties with known bounds. This paper combines the advantages of both approaches by introducing a novel SAT-solving technique which exploits the similarities among SAT instances for multiple properties and bounds on the same design. The proposed technique ensures that the knowledge obtained in previous solving iterations be shared across different bounds as well as between different properties. Our experimental results demonstrate that our approach can significantly reduce overall test generation time (on average 10 times) compared to existing methods.*

## I. Introduction

Functional verification is one of the most important stages in the development cycle of modern system-on-chip designs. Due to dramatic increase in complexity of embedded applications and decreased time-to-market, it is becoming more challenging to meet the required coverage before the deadline using conventional simulation methods with random tests. Directed tests are promising to address this problem since it requires significantly less number of directed tests to achieve the same coverage goal. Currently, directed tests are usually generated with the help of human intervention, which cannot match today's time sensitive development cycles. Therefore, it is desired to employ a fully automatic approach for directed test generation.

Model checking seems to be an attractive solution for automatic generation of directed tests, because the counterexample of the negated version of a property can be used as a test to activate the property. However, symbolic model checker sometimes cannot handle large designs due to the state space explosion problem. SAT-based bounded model checking (BMC), on the other hand, restricts the search space by only checking the property on the states reachable from the initial state within a fixed number ($k$) of transitions, called bound. This is achieved by unrolling the design $k$ times, then converting the unrolled design and the property description into a proposition satisfiability (SAT) problem. Next, SAT solver is employed to find a satisfiable assignment of variables, which can be used as a counterexample or test. The effectiveness of SAT-based BMC is based on the fact that SAT-BMC capacity is usually beyond BDD capacity for practical designs [1].

Since the SAT instances for BMC are generated by unrolling the same design for multiple times, their structural regularity can be exploited to accelerate the SAT solving process. Existing research in this area can be divided into two categories based on whether it addresses one property or multiple properties. The first category is applicable for test generation for one design and one property with varying bounds [2], [3]. However, the knowledge obtained are not shared when solving for other properties on the same design. In contrast, the methods in the second category tries to accelerate the test generation for multiple properties with known bounds [4]. They exploit the fact that although each test generation instance is created for a different property, these instances still have a large overlap, because the design remains unchanged. The major drawback of this solution is that it assumes that the bound is known. In general, it is very difficult to determine the bound upfront without actually solving the SAT instance, which limits the applicability of this solution.

In this paper, we combine the advantages of both approaches by developing a novel BMC based test generation technique for multiple properties of the same design, which enables the reuse of learned knowledge across different bounds and properties. The basic idea of our approach is to synchronize the solving process of multiple properties for different bounds, so that the utilization of learned knowledge can be maximized. One may think that solving many SAT instances together can be dramatically complex than solving one instance, and therefore many be impractical. On the contrary, since all these instances are generated by unrolling the same design for several times, we successfully developed a simple but effective approach to significantly reduce the overall SAT solving time by forwarding knowledge among different solving processes. Our experimental results demonstrate an order-of-magnitude reduction in overall test generation time.

The rest of the paper is organized as follows. Section 2 describes related work on BMC and directed test generation. Section 3 briefly discusses the background on SAT-based BMC. Section 4 describes our test generation methodology for multiple properties and bounds. Section 5 presents our

This work was partially supported by NSF CAREER award 0746261.

experimental results. Finally, Section 6 concludes the paper.

## II. Related Work

Model checking has been widely used for system verification and test generation [5], [6]. Since conventional model checking techniques are not suitable for large designs due to the state explosion problem, Biere et al. [7], [1] introduced bounded model checking with satisfiability solving. In general, BMC cannot validate a safety property when a counterexample is not found within a specific bound. However, SAT-based BMC is quite effective to falsify a design by providing a counterexample when the bound is not large, because in this case, SAT solvers usually require less space and time than conventional symbolic model checkers [8]. As a result, it is quite attractive in the field of directed test generation, where a counterexample typically exists within a relatively small bound. A study [9] illustrated that SAT-based BMC outperforms unbounded model checking for real designs.

A great deal of work has been done to accelerate the SAT solving process during BMC [10], [11], [3], [2], [4]. The basic idea is to exploit the regularity of the SAT instances and make use of the knowledge learned in previous SAT solving iterations. For example, incremental SAT solvers [12], [13] reduce the solving time by recording and utilizing the previously learned conflict clauses. A common approach is to keep generated conflict clauses as long as the clauses which led to the conflicts are not removed from the database. In [14], the clauses that are responsible for the deduction of a new conflict clause in the implication graph is recorded, so that such conflict clauses can be used in the future when applicable. The only problem is that keeping track of such dependencies might be expensive. Interestingly, such dependency in BMC are much easier to track than general cases. For single property checking, Strichman [3], [2] observed that if a conflict clause is deduced only from the transition part of a SAT instance, it can be safely forwarded to all instances with larger bounds, because the transition part of the design will still be in the SAT instance when we unroll the design for more times. Nevertheless, this approach was designed to check one property on one design at a time and cannot be directly applied to accelerate the SAT solving of multiple properties.

In directed test generation, Mishra et al. [4] forwarded clauses between properties to speed up the SAT solving process. They found that although the properties are different, since they are checked on the same design, the SAT instances have a large overlap. By solving a "base" property, many common conflict clauses can be generated to accelerate the solving processes of other properties. However, this method requires the bound as an input. Since the bound calculation itself is usually time-consuming, and may be impossible in many scenarios without solving the SAT instances, the applicability of their approach is restricted.

Although there are some attempts to attack the SAT-based BMC with multiple properties and unknown bound, none of them is suitable for directed test generation. For example, Fraser et al. [10] concatenated all properties into a large property with "AND" and checked them together. This approach is clearly not applicable in test generation because we have to find tests for each property. The simultaneous SAT solver [11] enabled the learned clauses to be reused by other solving objectives. However, it restricted the expression of the safety property or "proof objective" to have only one literal, which is not the case in realistic properties. Our approach, on the other hand, provides a general solution which can be applied in practice.

## III. Background : SAT-based BMC

This section briefly describes the basic concepts of BMC, directed test generation based on BMC, and modern SAT solving techniques. BMC checks whether there is a counterexample for the property within a given bound [1]. Given a design $D$, a safety property $p$, and a bound $k$, BMC will unroll the design $k$ times and encode it using the following formula:

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

where $I(s_0)$ is the initial state of the system, $R(s_i, s_{i+1})$ represents the state transition from state $s_i$ to state $s_{i+1}$, and $p(s_i)$ checks whether property $p$ holds on state $s_i$. The formula is then transformed to CNF and checked by a SAT solver. If the SAT solver finds some assignment which makes the CNF true, it implies that the property does not hold at bound $k$, i.e., $M \nvDash_k p$. Otherwise, if no such assignment is found, we conclude that the property holds up to $k$, or $M \vDash_k p$.

In directed test generation [5], [6], the negated version of the property is checked by BMC. The SAT solver will find a sequence of input assignments which drive the design from the initial state to the state where the negated version of the property fails. Therefore, we can use such a counterexample as a test to activate the intended functionality during simulation-based validation. Such tests are called directed tests, which is used for validation of both specification and implementation.

Many techniques and heuristics are employed in SAT solvers to accelerate the solving process. Modern SAT solvers such as zChaff [15] adopt the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and conflict-driven non-chronological backtracking. The basic idea behind these techniques is to save the knowledge learned during resolving current conflict to avoid the same conflict in the future [16]. A conflict occurs, when the current assignment of some variables, through a set of clauses, implies that one variable must be true and false at the same time. In this case, conflict analysis will trace back along the implication relations and find the closest assignment of variables that led to the conflict. We can forbid such assignment from occurring again by adding a carefully designed clause, i.e., conflict clause, to the original CNF. Generally, conflict clauses are only meaningful within the same SAT instance. However, when the set of clauses that led to the conflict clause are shared by multiple SAT instances, we can also forward conflict clauses across instances.

## IV. Synchronized Test Generation

Our work is motivated by previous works on incremental SAT for single property with unknown bound [2] as well as

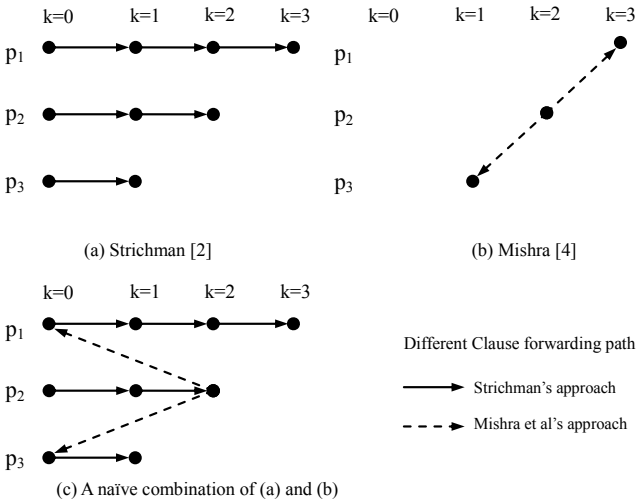


**Fig. 1. Different incremental SAT solving techniques**

test generation for multiple properties with known bounds [4]. These two different approaches are illustrated in Figure 1. In this example, there are three properties $p_1$, $p_2$, and $p_3$ with bounds 3, 2, and 1 respectively. We use solid dots to represent different SAT instances and lines to indicate the conflict clause forwarding paths. As discussed in Section II, Strichman et al. [2] solved each property separately, and passed the knowledge (deduced conflict clauses) "horizontally" within instances for the same property (Figure 1a). In contrast, Mishra et al. [4] solved one "base" property first, (e.g., $p_2$ in this case), then forward the learned clause "vertically" between other SAT instances for different properties, as shown in Figure 1b.

Clearly, it should be profitable if we can appropriately forward conflict clauses "vertically" between properties while solving for each property "horizontally". In this way, the knowledge learned during checking a property for a specific bound can benefit itself with larger bounds as well as across other properties. One intuitive way to combine the two approaches, as shown in Figure 1c, is to choose some property as based property ($p_2$ in Figure 1c), check this property for different bounds, and then forward the learned conflict clauses to other SAT instances for other properties. Unfortunately, this simple combination has three problems. First, it is very hard to choose the base property, that should yield a large number of conflict clauses which can be shared by other properties. Unlike [4], where each property has only one SAT instance, we do not know how many SAT instances we have to solve. As a result, it is impossible to apply the clustering technique proposed in [4], to determine the base property. Secondly, even if we correctly find the optimal base property, it is still difficult to choose the suitable bound of the receiving property to forward clauses, because SAT instances with inappropriate bounds may be solved trivially. Moreover, the learning during checking non-base properties is wasted. For example, $p_3$ in Figure 1c is not a base property. Thus, the conflict clauses we learned during checking $p_3$ cannot be shared by other property like $p_1$. When the number of properties is large, this may cause a great waste of computational power, because we have to explore the same search space for many times, if the space is not visited during the solving process of the base property.

Our approach to solve this problem is based on the effective identification of conflict clauses that can be shared by other SAT instances across properties and bounds. In fact, for any $k_0 \geq 0$, all SAT instances generated during BMC (Equation 1) with $k \geq k_0$ clearly share the transition clauses $I(s_0) \wedge \bigwedge_{i=0}^{k_0-1} R(s_i, s_{i+1})$, although their property terms $\bigvee_{i=0}^{k} \neg p(s_i)$ are different. This observation implies that all conflict causes deduced based on these common clauses during solving process of *any* SAT instance can be forwarded to *any* other SAT instances with $k \geq k_0$, because all of them have the same set of clauses that led to the conflict clause. Therefore, if we check all properties together for $k = 0, 1, 2, ...$, i.e., "synchronously", all conflict clauses can be safely shared by all subsequent SAT instances.

**Algorithm 1**: Synchronized Test Generation for Multiple Properties

**Input**: Design $D$, Property Set $P$, Maximum bound $K_{max}$
**Output**: Test Set $TS$
Bound $k \longleftarrow 0$
Common Conflict Clause Set $CCS \longleftarrow \emptyset$
$TS \longleftarrow \emptyset$
**while** $P \neq \emptyset$ *and* $k \leq K_{max}$ **do**
  Clause Set $CS_T^k \longleftarrow BMC(D, true, k)$
  **for** $p \in P$ **do**
    Clause Set $CS_p^k \longleftarrow BMC(D, p, k)$
  **Step1**: In $CS_p^k$, mark all clauses that also exist in $CS_T^k$
  **Step2**: $(ConflictC, test_p) \longleftarrow$ SAT($CCS \bigcup CS_p^k$)
  **Step3**: $CCS \longleftarrow CCS \bigcup$ CheckMark ($ConflictC$)
    **if** $test_p \neq null$ **then**
      remove $p$ from $P$
      $TS \longleftarrow TS \bigcup test_p$
    **end**
  **end**
  $k \longleftarrow k + 1$
**end**

Algorithm 1 outlines our synchronized test generation method for multiple properties. It accepts a design and a property set as inputs and produces corresponding tests. As indicated before, this algorithm will check all properties synchronously for each bound. In each iteration, we first generate the transition clause set $CS_T^k$ (corresponding to $I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$) using BMC(D,true,k), then randomly choose a property $p$ from the property set $P$, and create its own clause set $CS_p^k$ (corresponding to $I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$). Next, we perform following 3 steps.

1) Mark all clauses in $CS_p^k$ which are also in $CS_T^k$. Since $CS_T^k$ remains same for all properties at $k$, this step can be implemented efficiently by table lookup, as described in Section IV-B.
2) Use a SAT solver to solve the CNF formula $CCS \bigcup CS_p^k$, which contains not only $CS_p^k$, but also all previously learned conflict clauses in $CCS$.
3) For new conflict clauses $ConflictC$ learned by SAT

solver, merge the clauses deduced purely by marked clauses into $CCS$. This step is similar to the isolation technique proposed in [3] and [4].

If the satisfied assignment, or a counterexample $test_p$ is found in step 2, we record it in test set $TS$ and remove $p$ from P. This process repeats until tests for all properties are found or the maximum bound $K_{max}$ is reached. Finally, the algorithm returns all generated tests.

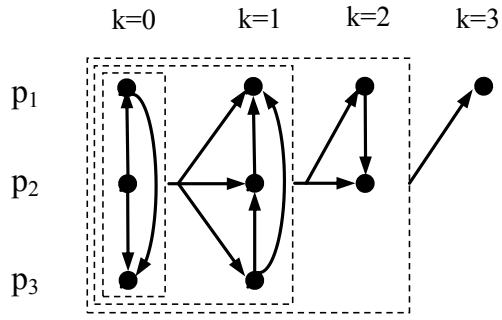


**Fig. 2. Synchronized test generation for multiple properties**

We use the same example in Figure 1 to illustrate the flow of Algorithm 1. The clause forwarding path are shown in Figure 2. In the first iteration for $k = 0$, suppose we randomly pick $p_2$ from the property set. At the beginning, the common conflict clause set $CCS$ is empty. Thus, $p_2$ is solved directly. Since the bound of $p_2$ is 2, the SAT instance is not satisfiable and no test is generated. However, all conflict clauses deduced based on clauses in $CS_T^0$ are now recorded in $CCS$, and will be used to accelerate the solving process of both $p_1$ and $p_3$ at bound 0. Similarly, the conflict clauses generated during solving $p_1$ at $k = 0$ will be used to speed up $p_3$ at $k = 0$ (assumes $p_3$ is solved last). In the next iteration, all instances will be solved with the help of conflict clauses learned by all three SAT instances at $k = 0$, because all conflict clauses are recorded in $CCS$. Eventually, three tests will be generated at bound 3, 2, and 1 for $p_1$, $p_2$ and $p_3$ respectively.

Note that our algorithm does not require the SAT instances to be preprocessed using Cone-Of-Influence (COI) optimization as in [2] and [4], because original SAT instances have more overlapped clauses, which are effectively exploited by our approach to accelerate the overall solving process. Our experimental results in Section V show that our approach (without COI) outperforms [2] and [4] that use COI optimization.

In the remainder of this section, we show the correctness of our approach and the implementation details of our synchronized test generation algorithm.

### A. Correctness of the proposed approach

To show the correctness of our test generation approach, we need to show that in Algorithm 1, solving $CCS \bigcup CS_p^k$ is equivalent to solving $CS_p^k$. Formally, let $\varphi_p^k$ and $\psi$ be the CNF formulae formed by clause set $CS_p^k$ and $CCS$ respectively, we need to prove that $\varphi_p^k$ is satisfiable iff $\varphi_p^k \wedge \psi$ is satisfiable using the following lemma.

*Lemma 4.1:* $\varphi_p^k \vdash \psi$ for all $p \in P$ and $k \geq 0$.

*Proof:* Let $\varphi_T^k$ be the CNF formula formed by $CS_T^k$. We first show that

$$\varphi_T^k \vdash \psi \tag{2}$$

for $k \geq 0$ by induction on the size of $\psi$. In the basis step, formula 2 obviously holds because $\psi$ is empty.

Considering the moment before a new conflict clause $\pi$ is added to $\psi$ in some iteration when the bound $k' \leq k$, $\pi$ must be deduced from $\varphi_T^{k'} \wedge \psi$, i.e., $\varphi_T^{k'} \wedge \psi \vdash \pi$. By induction hypothesis, $\varphi_T^k \vdash \psi$ before $\pi$ is added into $\psi$. We also know that $\varphi_T^k \vdash \varphi_T^{k'}$, because their original forms satisfy

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \;\vdash\; I(s_0) \wedge \bigwedge_{i=0}^{k'-1} R(s_i, s_{i+1})$$

Hence, $\varphi_T^k \vdash \varphi_T^{k'} \wedge \psi$. As a result, we have $\varphi_T^k \vdash \pi$ and $\varphi_T^k \vdash \psi \wedge \pi$, which means formula 2 still holds, after any new clause is added to $\psi$, as long as $k' \leq k$.

On the other hand, we notice that

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i) \;\vdash\; I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$$

or

$$\varphi_p^k \vdash \varphi_T^k$$

Therefore, we conclude that

$$\varphi_p^k \vdash \psi$$

for all $p \in P$ and $k \geq 0$. ■

Since $\varphi_p^k \vdash \psi$, we have $\varphi_p^k \leftrightarrow \varphi_p^k \wedge \psi$. In other words,

*Theorem 4.2:* $\forall p \in P$ $\varphi_p^k$ is satisfiable iff $\varphi_p^k \wedge \psi$ is satisfiable.

The correctness of our approach is therefore justified.

### B. Implementation Details

Our synchronized test generation algorithm is built around zChaff SAT solver [15], which provides clause management scheme to support incremental SAT solving. zChaff maintains all input clauses and generates conflict clauses within an internal clause database $DB$. When invoked, it will solve the CNF formed by all clauses currently in $DB$. The management of clauses within database $DB$ is based on "group". For each clause, zChaff assigns a 32-bit group ID. Each bit identifies whether that clause belongs to a certain group or not. When a conflict clause is deduced by clauses from multiple groups, its group ID is a "OR" product of the group ID of all its parent clauses, i.e., this clause belongs to multiple groups. zChaff also allows user to add or remove clauses by group ID between successive solving processes. If one clause belongs to multiple groups, it is removed when any of these groups are removed.

With these utilities, the step 1 and 3 in Algorithm 1 can be implemented efficiently as follows:

1) In the clause marking step, add all clauses in $CS_T^k \bigcap CS_p^k$ into $DB$ with group ID 1.
2) Add other clauses in $CS_p^k$ into $DB$ with group ID 2.

3) After solving all clauses in $DB$ with zChaff, remove clauses with group ID 2.

In this way, $CCS$ is implicitly maintained within $DB$, because only conflict clauses generated purely based on clauses in $CCS \bigcup CS_T^k$ are kept after each iteration.

There is another potential overhead in step 1. Before we mark it in $CS_p^k$, we have to identify whether it is in $CS_T^k$ or not. Since $CS_T^k$ remains same for all properties at $k$, we build a hash table to record all clauses in $CS_T^k$. It takes $O(1)$ time to determine whether a clause from $CS_p^k$ is in $CS_T^k$ or not. Therefore, the overall time consumption of step 1 and 3 in our algorithm is negligible compared to the SAT solving time.

## V. Experiments

We have evaluated our test generation approach using different software and hardware designs. In this section, we used two benchmarks, a stock exchange system and a VLIW implementation of the MIPS architecture, to enable comparison with [4]. The systems and properties are described in SMV language and converted to CNF clauses (DIMACS files) using NuSMV [17]. We use zChaff [15] as our SAT solver to implement our test generation algorithm. Both experiments were performed on a PC with 3.0GHz AMD64 CPU and 4GB RAM.

### A. A Stock Exchange System

The design in our first case study simulates the behavior of a common online stock exchange system (OSES). It can accept, check and execute the customers orders (market order and limit order). The system is specified using UML activity diagram and implemented in JAVA. Its UML behavior specification has 27 activities, 29 transitions and 18 key paths. As indicted before, the specification is translated into NuSMV input to generate corresponding SAT instances. Then we apply our synchronized SAT solving approach to find the satisfiable assignments, which can be used as tests. We compared our approach with Strichman's approach [2] and a simple combination of [2] and [4] on different properties with unknown bounds. For Strichman's approach [2], we use it to solve a sequence of SAT instances for the same property with varying bounds until a satisfiable instance is found. The simple combination of [2] and [4] is developed as described in Section IV. After SAT instance generation, we applied cone of influence (COI) to speed up Strichman's approach. When our approach was applied, we did not use COI as indicted in Section IV.

Table 1 shows the results of 20 most time consuming properties using Strichman's approach [2]. The first column shows the properties used for test generation. The second column indicates corresponding bounds of each property. The third column shows the test generation time (in seconds) for each property using our approach. The time consumed by step 1 and 3 in Algorithm 1 is also counted in this column. The fourth column indicates the time required by Strichman's approach [2] to generate the test for the same property. The time is calculated as the summation of the time to solve all the SAT instances from $k = 0$ to the bound of the property. The fifth column shows the speedup[1] of our approach over [2]. The last two columns present the test generation time using the simple combination of [2] and [4] and the speedup of our approach over it. It can be seen that our approach can produce more than 10 times improvement compared to [2], because many more conflict clauses are reused by subsequent iterations. This is especially important for "hard" SAT instances, which have to explore a potentially large assignment space. For example, the "hardest" property $p_1$ for [2] actually consumes less than 3 seconds in our approach. Clearly, the time consumption for solving multiple SAT instances using our approach is significantly smaller than the summation of time to solve each instances independently. The overall time consumption is reduced by knowledge sharing during solving all properties synchronously.

**TABLE I. Test generation time comparison for OSES**

| Prop. | Bound | Our Approach Time (s) | [2] vs ours | | [2] + [4]* vs ours | |
|---|---|---|---|---|---|---|
| | | | [2] Time (s) | Speed-up | [2] + [4] Time (s) | Speed-up |
| 1 | 15 | 2.94 | 180.31 | 61.24 | 67.58 | 22.96 |
| 2 | 14 | 2.55 | 150.49 | 59.06 | 57.70 | 22.64 |
| 3 | 14 | 3.12 | 149.89 | 48.04 | 61.11 | 19.59 |
| 4 | 15 | 10.54 | 139.56 | 13.25 | 42.53 | 4.04 |
| 5 | 14 | 19.38 | 130.58 | 6.74 | 55.74 | 2.88 |
| 6 | 14 | 2.97 | 107.13 | 36.09 | 61.66 | 20.77 |
| 7 | 16 | 6.61 | 101.67 | 15.39 | 35.86 | 5.43 |
| 8 | 16 | 3.54 | 89.31 | 25.20 | 3.76 | 1.06 |
| 9 | 15 | 1.73 | 84.19 | 48.72 | 38.97 | 22.55 |
| 10 | 12 | 1.96 | 84.07 | 42.80 | 5.51 | 2.80 |
| 11 | 13 | 1.21 | 83.94 | 69.48 | 22.54 | 18.66 |
| 12 | 15 | 2.83 | 83.80 | 29.59 | 39.77 | 14.04 |
| 13 | 15 | 5.60 | 83.01 | 14.81 | 23.49 | 4.19 |
| 14 | 14 | 1.34 | 80.25 | 59.88 | 22.60 | 16.86 |
| 15 | 14 | 11.16 | 79.79 | 7.15 | 22.53 | 2.02 |
| 16 | 15 | 0.85 | 78.72 | 92.39 | 10.94 | 12.85 |
| 17 | 15 | 0.88 | 78.28 | 88.95 | 14.51 | 16.49 |
| 18 | 15 | 0.86 | 78.19 | 90.49 | 12.60 | 14.58 |
| 19 | 12 | 79.40 | 74.96 | 0.94 | 75.10 | 0.95 |
| 20 | 12 | 1.38 | 73.46 | 53.23 | 5.43 | 3.93 |
| Total | - | 160.87 | 2011.62 | **12.50** | 679.93 | **4.23** |

* This is an intuitive combination of Strichman [2] and Mishra et al. [4] (Figure 1c). We have shown these results to demonstrate how our approach is superior than any simple combination of existing methods [2] and [4].

One interesting observation is that the most time consuming property $p_{19}$ in our approach has a bound of only 12. The reason for this is that the clauses learned during the solving process of easier properties like $p_{19}$ eliminated some useless searching attempts for the solution of harder properties like $p_1$. More importantly, these clauses are more effective than the conflict clauses learned during solving SAT instances of the same property with smaller bounds. Although $p_{19}$ itself, which was solved first, did not benefit from other properties, the overall time consumption was dramatically reduced. As a result, our approach outperforms [2], which only forwards clauses within SAT instances of the same property.

[1] calculated as (previous column / third column)

For the simple combination of [2] and [4], we chose $p_{19}$ as the base property and forwarded the clauses learned during solving it to other properties at bound 11. These parameters are selected to illustrate the best possible performance of the combination. It is remarkably faster compared to Strichman's approach [2], although it is still 4 times slower than our approach. It should be noted that in reality, it is impossible to choose the optimal parameter for this combination because the bounds are unknown for all properties. Thus, its performance will be much worse than what we illustrated here, while the performance of our approach will not change.

### B. A VLIW MIPS Processor

We also applied our test generation approach to a single-issue MIPS processor [6]. There are five pipeline stages: fetch, decode, execute, memory access, 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 multi-cycle divider (DIV).

We translated the design into the NuSMV input and used the three approaches to solve the generated SAT instances for different properties and bounds. For the combination of [2] and [4], we chose $p_{17}$ as the base property and forwarded learned clauses to bound 7. The results are given in Table 2. We only show the results on 20 most time consuming properties using Strichman's approach. It can be seen that our approach outperforms both Strichman's approach [2] and the simple combination of [2] and [4] by 15 and 3 times respectively.

**TABLE II. Test generation time comparison for MIPS**

| Prop. | Bound | Our Approach Time (s) | [2] vs ours | | [2] + [4]* vs ours | |
|---|---|---|---|---|---|---|
| | | | [2] Time (s) | Speed-up | [2] + [4] Time (s) | Speed-up |
| 1 | 8 | 0.78 | 139.29 | 179.48 | 18.66 | 24.04 |
| 2 | 8 | 0.74 | 132.07 | 178.46 | 19.45 | 26.29 |
| 3 | 8 | 0.76 | 125.18 | 164.70 | 18.18 | 23.93 |
| 4 | 8 | 0.76 | 120.02 | 158.74 | 18.45 | 24.40 |
| 5 | 8 | 0.76 | 115.84 | 151.61 | 27.14 | 35.53 |
| 6 | 9 | 0.86 | 111.13 | 129.81 | 58.26 | 68.06 |
| 7 | 8 | 0.81 | 108.09 | 133.76 | 26.63 | 32.95 |
| 8 | 9 | 0.95 | 104.56 | 110.29 | 53.59 | 56.52 |
| 9 | 8 | 0.75 | 96.25 | 128.67 | 16.77 | 22.41 |
| 10 | 8 | 0.77 | 87.24 | 113.00 | 16.47 | 21.33 |
| 11 | 8 | 0.76 | 87.23 | 114.77 | 17.37 | 22.85 |
| 12 | 8 | 0.77 | 84.98 | 110.64 | 16.45 | 21.42 |
| 13 | 7 | 0.65 | 81.08 | 125.11 | 13.35 | 20.60 |
| 14 | 9 | 32.31 | 80.25 | 2.48 | 31.61 | 0.98 |
| 15 | 8 | 0.76 | 75.47 | 99.30 | 7.25 | 9.54 |
| 16 | 8 | 0.76 | 72.05 | 94.30 | 20.63 | 26.99 |
| 17 | 7 | 76.54 | 71.72 | 0.94 | 72.30 | 0.94 |
| 18 | 8 | 1.00 | 70.05 | 70.33 | 19.46 | 19.53 |
| 19 | 8 | 0.76 | 69.85 | 91.90 | 6.98 | 9.19 |
| 20 | 8 | 0.76 | 65.80 | 87.03 | 11.08 | 14.65 |
| Total | - | 122.99 | 1898.13 | **15.43** | 490.06 | **3.98** |

## VI. Conclusions

Automatic generation of directed tests is promising for simulation based functional validation because it requires less number of test vectors to achieve the same coverage requirement. However, its applicability is limited due to the capacity restriction of current model checking tools. Existing incremental SAT approaches are suitable only for a single property with unknown bound or for multiple properties with known bounds. This paper presented an efficient technique for test generation by reusing learned knowledge across multiple properties and different bounds. To enable knowledge sharing among properties as well as bounds, we presented a synchronized test generation technique for multiple properties with different bounds. SAT instances for different properties are solved together, so that the discovery and utilization of the common conflict clauses can be maximized. The overall time consumption of checking multiple properties using our approach is remarkably smaller than the summation of time to check each property independently. Our experimental results on both hardware and software designs illustrated an order-of-magnitude reduction in overall test generation time.

## References


[1] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, 2001.

[2] O. Strichman, "Accelerating bounded model checking of safety properties," *Formal Methods in System Design*, vol. 24, no. 1, pp. 5–24, 2004.

[3] O. Shtrichman, "Pruning techniques for the SAT-based bounded model checking problem," in *Proc. of CHARME*, 2001, pp. 58–70.

[4] P. Mishra and M. Chen, "Efficient techniques for directed test generation using incremental satisfiability," in *Proc. of VLSI Design*, 2009, pp. 65–70.

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

[6] P. Mishra and N. Dutt, "Graph-based functional test program generation for pipelined processors," in *Proc. of DATE*, 2004, pp. 182–187.

[7] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. of TACAS*, 1999, pp. 193–207.

[8] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *Proc. of DAC*, 2001, pp. 530–535.

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

[10] R. Fraer, S. Ikram, G. Kamhi, T. Leonard, and A. Mokkedem, "Accelerated verification of RTL assertions based on satisfiability solvers," in *HLDVT*, 2002, pp. 107–110.

[11] Z. Khasidashvili, A. Nadel, A. Palti, and Z. Hanna., "Simultaneous SAT-based model checking of safety properties." in *Proc. of Haifa Verification Conference*, 2005, pp. 56–75.

[12] J. N. Hooker, "Solving the incremental satisfiability problem," *Journal of Logic Programming*, vol. 15, no. 1-2, pp. 177–186, 1993.

[13] H. Jin and F. Somenzi, "An incremental algorithm to check satisfiability for bounded model checking," in *Proc. of BMC*, vol. 119, no. 2, 2005, pp. 51 – 65.

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

[15] zChaff. [Online]. Available: http://www.princeton.edu/∼chaff/zchaff.html

[16] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," in *Proc. of ICCAD*, 2001, pp. 279–285.

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