Efficient Self-Learning Techniques for SAT-Based Test Generation

Ang Li and Mingsong Chen
Shanghai Key Lab of Trustworthy Computing
East China Normal University, Shanghai, China
{ali,mschen}@sei.ecnu.edu.cn

ABSTRACT

SAT-based approaches are promising for automated generation of directed tests. However, due to the state space explosion problem, these methods do not scale well for complex designs. Although various heuristics are proposed to address test generation complexity, most of them require expert knowledge regarding the detailed structure and behavior information of designs explicitly, which limits their usage. This paper proposes promising techniques to derive profitable learnings from the SAT instance itself. The obtained self-learnings can efficiently reduce the chance of long distance backtracks and improve satisfying assignment convergence rate during the SAT search. Experimental results demonstrate that our method can reduce the test generation time by several orders of magnitude.

Categories and Subject Descriptors

B.7.3 [**Integrated Circuits**]: Reliability and Testing—*Test generation*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*

General Terms

Verification, Algorithms, Experimentation

Keywords

Functional Validation, Test Generation, SAT

1. INTRODUCTION

Under the pressure of increasing complexity and decreasing time-to-market, functional validation is becoming a major bottleneck in System-on-Chip (SoC) design. To achieve a coverage goal, current complex designs need to run trillions of random or constrained-random tests. Therefore up to 70% of the design development efforts are wasted in the validation process. As an alternative, directed testing methods have been widely investigated, since they use fewer tests and less simulation time to achieve the required functional coverage. However, most directed test generation approaches [12] need human intervention which is laborious and error-prone. Therefore it is desired that the process of directed test generation can be fully automated.

As a promising method, model checking [7, 22] can be used for automated generation of directed tests. It translates the investigated design into a formal model and the testing target into a property in the form of temporal logic. Then a corresponding model checker searches the state space of the design for a counterexample of the specified property. Such a counterexample is an assignment of variables, which can be used as a directed test to trigger the testing target. However, large designs generally involve complex interactions among multiple components. Model checking of such designs often terminates abnormally due to the out of computation or memory resource. To address the checking complexity, Boolean Satisfiability (SAT) based Bounded Model Checking (BMC) [2] is proposed to restrict the search range to find a shortest path which can falsify the specified property. Assume that the design is M and the property is p. SAT-based BMC unrolls the design and the property k timesteps using the following Boolean formula:

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

It consists of three parts: i) $I(s_0)$ represents the initial state, ii) $T(s_i,s_{i+1})$ denotes the transition from state s_i to state s_{i+1}, and iii) $p(s_i)$ tests whether p holds on state s_i. Then this formula will be transformed to Conjunctive Norm Form (CNF) and checked by a SAT solver. If M has a false state against p within bound k, the Boolean formula will be evaluated to be *satisfiable*. The derived satisfying assignment for this formula can be used as a directed test against p.

Since the test generation is to find a satisfying assignment for the checking SAT instance, the performance of SAT searching plays an important role in determining the test generation time. To enable quick SAT search, various heuristics based on CNF structures (e.g., literal count [13] and variable dependencies [17]) have been investigated. However, most of them focused on general purpose SAT problems. When the test generation becomes the target, the potential improvement of SAT search can be further exploited. Learning-based approaches [3, 6] are promising candidates to quickly achieve satisfying assignments for SAT instances. It can be used to increase satisfying assignment convergence rate as well as to avoid long-distance backtracks during the SAT search. Therefore it is promising for SAT-based test generation. However, the majority of learning-based approaches are based on the assumption that designs should be well formed to facilitate the exploration of the structure and behavior information, which strongly limits the applicability of these methods. Unlike most approaches that learn from external sources, this paper presents our self-learning techniques which can exploit the profitable learnings from the SAT instance itself to accelerate test generation. It makes two major contributions: i) it investigates the structure of both BMC formulas and CNF clauses as learning objects to derive profitable self-learnings; and ii) it proposes three promising partitioning approaches to derive self-learnings from the SAT instances with low overhead.

The rest of the paper is organized as follows. Section 2 introduces related work on efficient approaches for SAT-based model checking and directed test generation. Section 3 presents the details of our self-learning techniques. Section 4 provides the experimental results. Finally, Section 5 concludes the paper.

2. RELATED WORK

Model checking is being gradually employed for automated generation of directed tests. Unlike traditional Binary Decision Diagram (BDD) based unbounded model checking [7] which easily suffers from the state space explosion problem, SAT-based BMC [2] restricts the search within short possible bounds. The limited search space can reduce BMC validation efforts drastically. Therefore SAT-based BMC succeeds in validating many real industry scale designs [1].

SAT solver acts like the engine of the SAT-based BMC. To improve its performance, various learning-based approaches are developed. Since conflict clauses [14] can avoid repetitive conflicts, they can be used as a kind of learning. For example, when verifying a safety property without knowing the minimum bound in advance, incremental SAT solvers [18] try to check a series of SAT instances with increasing bounds. During the checking, the conflict clauses derived from small-bound SAT instances can be forwarded to large-bound SAT instances as assignment constraints. Such constraints can prune the search space. Hence the overall validation effort can be saved. To exploit more such constraints, Strichman [18] investigated the transition symmetry indicated in Equation (1). He found that conflict clauses derived only from transition relations can be replicated to further reduce the checking time. As an alternative, decision ordering [13] (e.g., VSIDS [14]) plays an important role in determining SAT searching performance, since it can be used to avoid long-distance backtracks when encountering a conflict [17]. To enable quick checking, Zhang et al. [21] proposed an approach which combines clever orchestration of decision ordering and learned information in an incremental framework for BMC. By analyzing the correlation among different SAT instances of a property, Wang et al. [20] used the unsatisfiable core of previously checked SAT instances to derive the decision ordering for unchecked SAT instances. Although all the above approaches can reduce the checking complexity, most of them focus on model checking rather than test generation.

Learning techniques have also been investigated in SAT-based verification and test generation. In [3], Chandrasekar and Hsiao presented an approach to generate tests for path-delay fault models. By dynamically excluding untestable paths, both static and dynamic learning can improve the test generation time. In [10], Lu et al. utilized the circuit topological information and signal correlations to enforce a decision ordering, which enables efficient solving of circuit-based SAT problems. When checking a large set of false properties, Chen and Mishra [4] proposed an approach that can identify the similarity between properties and cluster them together. By sharing the learnings derived from checked properties, the test generation time for the unchecked properties can be drastically reduced. However, the first property checking becomes a major bottleneck due to the lack of learning. To address this issue, they proposed a property decomposition method [6]. By achieving learnings from the spatially and temporally decomposed sub-properties, the test generation time of the original complex property can be reduced. However, the requirement of clear understanding of the structure and behavior information of both properties and designs limits the applicability of these approaches.

This paper is inspired by the intra-property learning approach presented in [5]. The basic idea of [5] is to halve the checking SAT instance and solve one half first. By sharing the learning from the checked half, the test generation time of the original complex property can be reduced. However, this method does not fully investigate other potential learning objects as well as partition heuristics to further reduce the test generation time. To the best of our knowledge, most existing approaches exploit the learnings from a set of correlated properties or sub-properties to reduce test generation time. This paper presents a promising method which can obtain profitable self-learnings from various sources to improve the test generation time.

3. OUR APPROACH

Since our goal is to efficiently generate direct tests using BMC, our approach only focuses on safety properties. Although the bound determination is generally hard in BMC, for directed test generation, the bound can be estimated based on the structure of designs [4]. As shown in Equation (1), BMC encodes a property checking problem into a SAT instance. One satisfying assignment of this SAT instance can be used as a test to activate the property. Therefore the efficiency of the search for a satisfying assignment determines the performance of directed test generation.

Ideally, optimal SAT solvers can find a satisfying assignment without making any decision mistakes. However, such kinds of solvers do not exist. Currently, most SAT solvers adopt the Davis-Putnam-Logemann-Loveland (DPLL) procedure [8] which is widely recognized as an efficient approach in practical searching.

```
while (1){
        run_periodic_functions();
        if (decide_next_branch()) {
            while (deduce() == CONFLICT) {
                    blevel = analyze_conflicts();
                    if(blevel < 0)
                        return UNSAT;
            }
        }
        else return SAT;
}
```

Figure 1: DPLL search procedure

Figure 1 shows the skeleton of the DPLL implementation used in many SAT solvers, such as zChaff [14] and MiniSAT [15]. It contains three key parts which strongly affect the process of finding satisfying assignments.

- The function *decide_next_branch* decides Boolean variable values based on decision ordering, i.e., the priority of literals.
- The function *deduce* propagates the effect of the new variable assignment using the implication deduction.
- The function *analyze_conflict* removes the effect of variable assignment conflicts using proper backtracks. It marks the reason of a conflict and creates a constraint (i.e., conflict clauses) to avoid the same conflict in future processing.

In most DPLL-based approaches, the decision of next branch is mainly based on the CNF statistics, which cannot properly give a satisfying variable assignment prediction in many situations. Therefore, SAT search performance can be easily deteriorated because of bad decisions, especially in the following two scenarios: i) when a large number of implications are involved between an early incorrect decision and a late conflict, a costly *long distance backtrack* is needed to resolve the conflict, and ii) the *satisfying assignment convergence* will be disturbed due to frequent bad decisions. If all these two scenarios can be avoided during the DPLL search, the test generation time can be drastically improved.

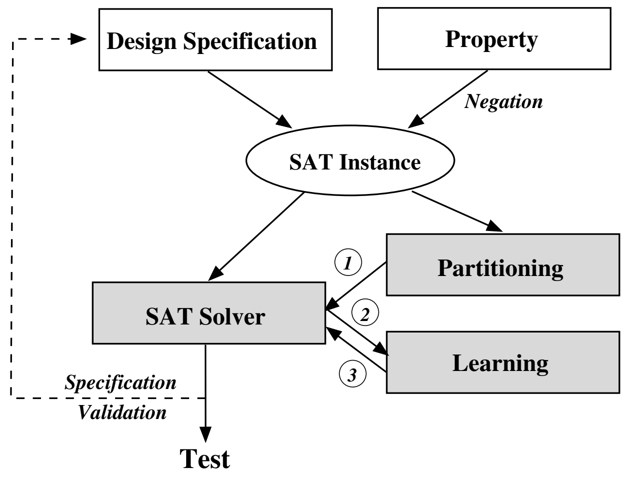


Figure 2: The workflow of our method

This paper presents an approach which adopts self-learning techniques to efficiently address the above two issues. Figure 2 shows the workflow of our method. Firstly, the design specification and a checking property are encoded into a SAT instance. Then the SAT instance is divided into small pieces (indicated by ①) using our partitioning heuristics. By the local search on these small pieces of CNF clauses, proper self-learnings (indicated by ②) can be achieved with much lower cost compared to the original SAT instance checking. However, such accumulated learnings (indicated by ③) are profitable, since it can effectively reduce the chance of both long-distance backtracks and bad decision ordering predictions when processing the unchecked parts. Therefore the overall test generation time can be drastically reduced.

3.1 Self-Learning Objects

Due to lack of external learning sources, our approach tries to exploit learnings from the checking SAT instances to enable quick test generation. It is required that self-learnings are of high quality that can efficiently reduce the chance of long-distance backtracks and bad variable assignment decision as much as possible. Since conflict clauses can be used as guards to avoid specific variable assignment, it is a promising candidate of self-learning. As well, decision ordering heuristics can be utilized as another self-learning to guide the prediction of satisfying variable assignment. However, due to the indistinct study objects, the quality and the effectiveness of self-learnings in traditional SAT search are not well studied. In the context of test generation, the capabilities of both self-learnings are limited. If the learning object is set to avoid long-distance backtracks and accelerate satisfying assignment convergence, the potential of both self-learnings can be further exploited. The following sub-sections present what are good objects for high quality self-learning derivation.

3.1.1 Learn from the Structure of BMC Formulas

The structure information of BMC formulas is a good source to achieve self-learnings. Assume that *prop* is a safety property for design M and it fails at the n_{th} clock cycle. According to the formula shown in Equation (1), the SAT instance for *prop* can be decomposed into three parts, $I = I(s_0)$, $T = \bigwedge_{i=0}^{n-1} T(s_i,s_{i+1})$ and $P = \bigvee_{i=0}^{n} \neg prop(s_i)$. It is important to note that T only constructs the links between states from s_0 to s_n without any explicit variable assignment constraints. When a bad decision is made for a variable, its effect can be propagated along one transition path easily by deduction. If a large number variables are involved in the deduction through the media T, the corresponding backtracking will be costly, since the rollback will cancel all assignments for these variables.

Figure 3 shows two examples which can easily cause conflicts during the SAT search for *prop*. As shown in Figure 3a), a transition variable V in clock cycle k ($0 < k \leq n$) is determined first. Through the deduction on other transition variables of T, its effect will be propagated to either the part of I or the part of P. Since both I and P indicate strong constraints for state variable assignments, when the deduction arrives, it often causes an assignment conflict, i.e., $\neg V$. If the clock cycle k is far away from the the part of I or P, a long-distance backtrack to resolve the conflict is inevitable. As an extreme case shown in Figure 3b), a variable V which belongs to the n_{th} clock cycle is decided in an early stage. Due to lack of disturbance from the constraints imposed by transitions, the effect of the assignment can be propagated easily to the other end, i.e., I. Before encountering a conflict $\neg V$, there may be thousands of implications and new decisions made during this deduction process. Consequently resolving such a conflict is very costly.

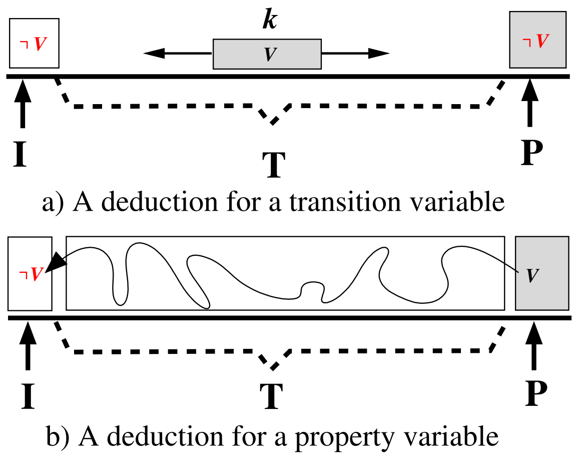

k
¬V
V
¬V
I
T
P
a) A deduction for a transition variable
¬V
V
I
T
P
b) A deduction for a property variable

Figure 3: Two cases of variable deduction

From the above examples, we can find that there are two major reasons that lead to long-distance backtracks: i) the constraint imposed by the transition part T is weak; and ii) a bad decision is made too early but detected too late. To enforce constraints on the transition part T, one applicable way is to instrument some local guards along the transition path, which can block the potential long-distance backtracks in an early stage. To improve the accuracy of Boolean variable assignment decisions in the mean time, proper heuristics should be applied. By our observation, conflict clause and decision ordering are two promising self-learning candidates to address the above two problems.

Conventional DPLL-based learnings cannot be directly applied to solve the long-distance backtracks. Since the learning object in conventional SAT searching is the whole SAT instance, the derived conflict clauses act like global improvisatory guards to avoid the second occurrence of the same conflicts. However, it cannot prevent long-distance backtracks in a proactive manner. In addition, traditional VSIDS-based decision ordering is derived according to the statistics of CNF clauses without considering BMC structure information, which results in inaccuracy in decision ordering prediction.

Unlike conventional methods, we set the learning object to a small part of the checking SAT instance. In our method, the SAT instance is divided into several smaller parts as described in Section 3.2. By checking each of them against the constraints of both I and P, some "local" self-learnings can be achieved and applied on the original SAT instance. The collected conflict clauses can be used as local guards of transitions at different clock cycle. It can effectively break down the route of long-distance backtracks. As well, since the small SAT piece indicates the partial behavior of the original SAT instance, its satisfying result can be useful to guide the decision ordering prediction on the original SAT instance. These ideas may have the effect of improving the convergence rate of the SAT search.

3.1.2 Learn from the Structure of CNF Clauses

SAT search heuristics based on CNF structure have been widely investigated. However, most of them focus on deriving decision ordering based on literal and clause statistics as well as variable dependence. Few of them utilize the CNF structure to guide the conflict clause generation to enable efficient search space pruning. Alternatively, this paper takes the clause size into account to derive high-quality conflict clauses. Since smaller conflict clauses can prune more search space, the size of a conflict clause determines its quality. If the derived conflict clauses are of small size, the test generation time can be improved.

For test generation, a SAT instance may consist of a large set of CNF clauses. If all the CNF clauses are of small size, generally the SAT search will be terminated in a reasonable time. This is because, during the SAT search of small-size clauses, the impact of a variable decision can be instantly propagated so that variable assignment conflicts can be detected quickly. That means, due to the strict constraints imposed by small clauses, only a few new decision levels are involved between the level of decisions and the level of the corresponding conflict. Therefore, the long-distance backtracks can be effectively avoided. In addition, since the number of backtracked levels determines the conflict clause size, the derived conflict clauses are more likely to be of small size. Consequently, the SAT search space can be efficiently pruned due to the high quality of the derive learnings.

However, most practical SAT searches encounter a mixture of CNF clauses with both large and small size. The existence and introduction of large clauses can strongly affect the SAT search performance. The following is an example using a SAT instance segment:

$$\ldots(y \vee x_1 \vee x_2 \ldots \vee x_k) \wedge (\neg y \vee x_1 \vee x_2 \ldots \vee x_k \vee z_1 \vee z_2 \ldots \vee z_j)\ldots$$

When adopting the decision sequence $\neg z_1$, $\neg z_2$, ..., $\neg z_j$, $\neg x_1$, $\neg x_2$, ..., $\neg x_k$, a conflict of y's assignment happens. To resolve such conflict, a cancel of $k+j$ decision levels may be needed, which involves a large set of determined Boolean variables in these levels. Moreover, to avoid such scenario again, a conflict clause $(x_1 \vee x_2 \ldots \vee x_k \vee z_1 \vee z_2 \ldots \vee z_j)$ will be generated. Since SAT solvers are not sensitive to large conflict clauses, the influence of such a conflict clause is negligible in most scenarios.

Few of existing SAT solvers try to reduce the chance of large conflict clause generation in a proactive manner. Most of them drop large conflict clauses on-the-fly during the SAT search, which is a waste of validation effort. However, learning directly from the original SAT instance cannot affect the size of conflict clauses. So we adopt the similar strategy as the heuristic proposed in Section 3.1.1. The basic assumption is that a set of small CNF clauses has a high chance to derive small-size conflict clauses. The small CNF clauses of a large SAT instance can be grouped and check together first. The achieved small-size conflict clauses put stricter restrictions during the SAT search, which can efficiently avoid long-distance backtracks. In addition, the satisfying assignment for the small clause group is also a promising source for deriving decision ordering for the original SAT instance.

3.2 Learning Oriented Partitioning Heuristics

Deriving self-learnings directly from a SAT instance is costly. In our approach, a SAT instance is partitioned into small parts and self-learnings can be achieved from them with low overhead. By combining the local learnings together, we can economically form a "global" learning to accelerate the original SAT searching.

3.2.1 Segmental Partitioning

When handling a large SAT instance as a whole, the effect of a bad decision may propagate to a large range. The interleaving of a large quantity of implications and decisions will postpone the conflict detection. Although conflict clauses can prune the search space, it cannot actively reduce the occurrences of long-distance backtracks. To reduce the chance of long-distance backtracks, we developed the segmental partitioning approach which utilizes self-learnings to prevent long-distance backtracks. By dividing the original SAT instance into several segments with equal size and checking each of them separately, we can achieve both local conflict clause and partial decision ordering based learnings in a short time due to the small size of the learning objects.

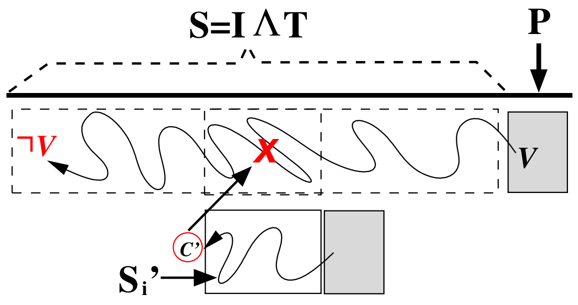


Figure 4: Segmental partitioning for self-learning

Figure 4 shows an example of our segmentation based partitioning method. Let S'_i $(1 \leq i \leq k)$ be a segment of $I \wedge T$ such that $\Sigma_{i=1}^{k} S'_i = (I \wedge T)$ and $sizeof(S'_i) = sizeof(S'_j)$ $(1 \leq i,\ j \leq k$ and $i \neq j)$. Our method derives self-learnings by checking each segment $S'_i \wedge P$, which indicates whether the segment S'_i satisfies the property requirement described by P. Because $sizeof(P)$ is much smaller than $sizeof(I \wedge T)$, the $sizeof(S'_i \wedge P)$ is also much smaller than $sizeof(I \wedge T \wedge P)$. Due to the small size of a segment, self-learnings in the form of conflict clauses and decision orderings can be derived in a short time. In addition, such learnings are very beneficial. By enhancing the variable assignment restrictions on segment S'_i, the derived conflict clauses can be used as local guards to block implications which traverse through this segment. Therefore many long-distance backtracks in the original SAT instance can be efficiently blocked. For example, in Figure 4, the conflict clause C' derived from $S'_i \wedge P$ can be used as a local guard to block a long-distance backtrack which happens in the original SAT instance checking. Furthermore, since $S'_i \wedge P$ is a part of $I \wedge T \wedge P$, the derived satisfying assignment for $S'_i \wedge P$ may have a large overlap with the partial satisfying assignment for $I \wedge T \wedge P$. Thus, the variable assignment for $S'_i \wedge P$ can be used as a learning to tune the decision ordering of the original SAT instance.

3.2.2 *Incremental Partitioning*

Segmental partition focuses on preventing long-distance backtracks by establishing local guards during the searching. However, the self-learnings derived using this method only have a local view, especially for the decision ordering based learnings.

When composing the satisfying assignments of all segments, conflicts arise due to the lack of global view. In addition, the conflict clauses derived from the local segments may neglect the long-distance backtracks which only happen in the global search. Therefore it is necessary to find a solution which takes both local and global views into account.

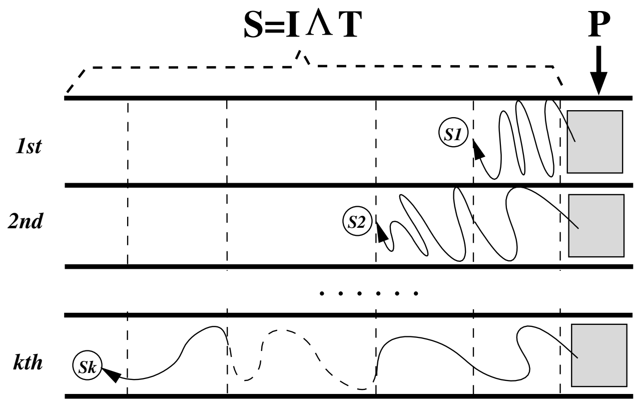


Figure 5: Incremental partitioning for self-learning

Figure 5 presents our incremental approach which can approximate the search range gradually to the original SAT instance. Assume that $I \wedge T$ is divided into k parts (i.e., s_1, s_2, ..., s_k) with same size. The incremental approach does k iterations to achieve a satisfying assignment for the SAT instance. The first iteration checks $s_1 \wedge P$, which is similar to the segmentation method. However, the second iteration checks $s_1 \wedge s_2 \wedge P$, which doubles the size of the non-property part. In our incremental method, the learnings derived from the first iteration are forwarded to help the second iteration checking. Thus the SAT checking time in the second iteration can be drastically reduced. The following iterations deal with the remaining parts with increasing size in the same way until the $(k-1)_{th}$ iteration. Finally, the k_{th} iteration checks the original SAT based on learnings from all previous $k-1$ iterations. Since the larger SAT instance checking is based on the smaller SAT instance checking, the overhead for incremental process is small. However, the accumulated self-learnings in all iterations can drastically improve the overall SAT checking time.

3.2.3 *Clause Size Aware Partitioning*

As described in Section 3.1.2, the clause size determines the quality of the conflict clauses based self-learning. To guarantee the high quality of the derived self-learning, we develop the clause size aware partitioning strategy which can make the size of derived conflict clauses as small as possible. The implementation of clause size aware partitioning is similar to the incremental partitioning. The only difference is that the SAT instance is partitioned based on the size of clauses rather than the temporal order of transitions.

Figure 6 shows our clause size aware partitioning approach. Let k be the step size chosen to partition $I+T$. $(I+T)_{1..k}$ indicates the set of clauses in $I+T$ which have a size between 1 and k. In an incremental order, $P+(I+T)_{1..k}$ is checked first. Based on the profitable learning from the checked part, checking $P+(I+T)_{1..2k}$ becomes easier. Assume that the iteration has n steps. In the $(n-1)_{th}$ step, $P+(I+T)_{1..(n-1)k}$ will be checked. And all the learnings collected during the $(n-1)$ iterations will be forwarded to help the checking of $I+T+P$. Since the derived small conflict clauses are with strict constraints on variable assignments, it can effectively prevent the long-distance backtracks. Therefore the overall SAT performance can be improved.

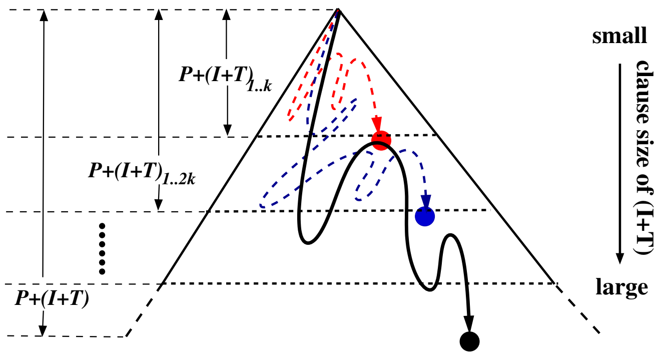


Figure 6: Clause size aware partitioning for self-learning

3.3 Self-Learning Based Test Generation

Test generation using SAT-based BMC focuses on how to efficiently achieve satisfying assignments quickly. Our test generation approach tries to exploit self-learnings to boost the SAT solving using both conflict clauses and decision orderings.

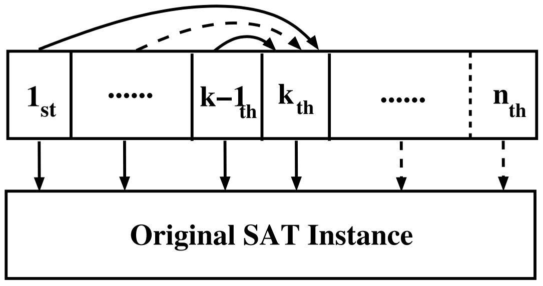


Figure 7: Conflict clause forwarding pattern

In our test generation framework, we collect all the conflict clauses during the checking iterations for partitions. Figure 7 shows our conflict clause forwarding pattern. When handling test generation with n partitions, our method uses an array $lBucket[n]$ of n learning buckets to store conflict clause base learnings. It is important to note that the k_{th} $(1 < k \leq n)$ partition checking is guided by the learning stored in $lBucket[1..k-1]$, i.e., the learning derived from previous $k-1$ partitions. Therefore the checking time of the k_{th} partition can be drastically reduced. Finally all the learnings collected from each partition checking are applied on the original SAT instance checking. Compared to the original SAT instance checking, the time of self-learning derivation process can be negligible. However, the achieved self-learnings are profitable, which can drastically reduce the complexity of the original SAT instance checking.

Decision ordering is another kind of self-learning to accelerate test generation in our approach. Since each partition is a subset of the original SAT instance, it indicates partial behavior of the original SAT instance. Therefore, the intermediate satisfying results for the partitions may have a large overlap with the original SAT instance on satisfying variable assignments. In our approach, under the help of intermediate satisfying results, the decision ordering of the unchecked partitions is affected and gradually tuned by the results of checked partitions. If all the collected information can be used to derive decision ordering for the original SAT instance, the test generation time can be drastically reduced.

Let $vStat[sz][2]$ (sz is the number of variables in the original SAT instance) be a 2-dimensional array which stores the accumulative results for the checked partitions. Initially the value of all the elements of $vStat[sz][2]$ are 1. After checking one partition, the value of $vStat$ will be updated. Assume that the k_{th} partition is the latest checked part. If the variable v_i is involved in the evaluation and its value is 1, then $vStat[i][1]$ will be increased by 1. If the variable v_i is involved in the evaluation and its value is 0, then $vStat[i][2]$ will be increased by 1. Otherwise, $vStat[i][\,]$ will remain unchanged.

In this paper, we investigate two different SAT solvers - zChaff [16] and MiniSAT [15], which adopt different decision ordering heuristics. zChaff supports the VSIDS heuristic, which calculates the score (i.e., priority) for each literal l dynamically. Initially, the *score(l)* equals to the literal count in the SAT instance. Then during the SAT search, after a certain number of backtracks, *score(l)* will be updated in the period function using the following formula

$$score(l) = score(l)/2 + new\_conflict\_count(l) \qquad (2)$$

where *new_conflict_count(l)* is the number of newly added conflict clauses which contain literal l since last update. Based on the VSIDS framework, our method incorporates the learning information from $vStat[sz][2]$. Instead of replacing the Equation (2), at the initialization of SAT searching and periodic score decaying, our method first calculates the literal score using the Equation (2) followed by the Equation (3):

$$score(l_i) = \begin{cases} max(v_i) + step & (vStat[i][1] > vStat[i][2] \& l_i = v_i) \\ & or (vStat[i][1] < vStat[i][2] \& l_i = v_i') \\ score(l_i) & otherwise \end{cases} \qquad (3)$$

where $max(v_i) = MAX(score(v_i), score(v_i'))$ and the value of $step$ is determined by the type of partitioning. For segmental partitioning, $step$ is set to be 1 since all the partitions have the similar size. When adopting incremental or clause size aware partitioning heuristics, the result of later checked partition will be more approximate to a satisfying assignment of the original SAT instance. In this case, $step$ is set to be k when handling the k_{th} partition, since it can reflect the importance of later checked partitions.

Similar to zChaff, MiniSAT employs a variant of the VSIDS heuristic. However, MiniSAT does not support explicit ordering for literals. It only keeps activity scores for variables and clauses, which cannot be used to predict the variable polarities (i.e., Boolean values of variables). Based on the statistics collected in $vStat$, if a variable has not been determined yet, its polarity can be predicted at the beginning and the restart of the search using the following formula:

$$polarity(v_i) = \begin{cases} true & (vStat[i][1] > vStat[i][2]) \\ false & (vStat[i][1] < vStat[i][2]) \\ skip & (vStat[i][1] = vStat[i][2]) \end{cases} \qquad (4)$$

Algorithm 1 describes our self-learnings based test generation approach. The algorithm has three inputs: i) a formal model of the design; ii) a specified property and its minimum satisfiable bound; and iii) the partition strategy and the number of partitions. The first step initializes the learning buckets $lBucket[n]$ with no learnings and resets the elements of $vStat[sz][2]$ with all 1s. Step 2 encodes the property falsification into a SAT instance in CNF format, and step 3 divides the CNF clauses into n partitions using the specified partitioning strategy. Then the following iterations try to collect both conflict clause and decision ordering based learnings. In the i_{th} iteration, step 4 checks the i_{th} partition based on the accumulative learnings derived from all previous $i-1$ iterations. Step 5 collects the conflict clause based self-learning derived in i_{th} iteration and save it in $lBucket[i]$. Step 6 updates the decision ordering based learning using $vStat[sz][2]$. Step 7 checks the original SAT instance using both the learnings from $lBucket[1..n]$ and $vStat[sz][2]$. Finally, the algorithm reports a test t_P to falsify the property P.

```
Algorithm 1: Test Generation using Self-Learnings
Inputs: i) Formal model of the design, $D$
        ii) Property $P$ with a satisfiable bound $bound_P$
        iii) Partitioning type $type$ and the number of partitions $n$
Outputs: A test $t_P$ to falsify $P$
Begin
  1. Initialize $lBucket[1..n]$ and $vStat[1..sz][\,]$;
  2. $CNF = BMC(D, P, bound_P)$;
  3. $\{p_1, p_2, \ldots, p_n\}$ = Partition($CNF$, $type$, $n$);
  for $i$ is from 1 to n do
      4. $(assign_i, conf_i)$ = SAT($p_i$, $lBucket[1..i-1]$, $vStat$);
      5. $lBucket[i] = conf_i$;
      6. for $j$ is from 1 to sz do
             if $(assign_i[j] == 0)$
                $vStat[j][2]++$;
             else if $(assign_i[j] == 1)$
                   $vStat[j][1]++$;
         endfor
  endfor
  7. $(t_P, )$ = SAT($CNF$, $lBucket[1..n]$, $vStat$);
  return $t_P$
End
```

4. EXPERIMENTS

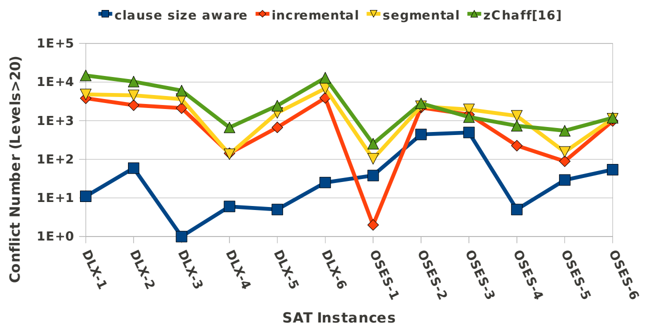
By applying our proposed self-learning techniques, this section presents the results of the benchmarks from two different sources: i) a set of SAT instances derived from a VLIW MIPS processor [6] and a stock exchange system OSES [23]; and ii) a set of SAT instances from 10 buggy variants of 12-pipelined processors named *PIPE-SAT-1.1* [19]. All such SAT instances are pre-determined to be satisfiable. The first benchmark set is generated by the BMC tool NuSMV [11]. Although the second benchmark is not generated by any BMC tools, our approach can also be applied to derive corresponding tests quickly. To evaluate the applicability of our approach on unsatisfiable SAT instances, we also conducted the experiment on a set of bigger variants of the pipe_ooo benchmarks named *PIPE-OOO-UNSAT-1.1* provided in [19]. We modified both the SAT solvers MiniSAT-2.2 [15] and zChaff [16] to incorporate our proposed partitioning and learning techniques. In these experiments, all the partitioning heuristics divide SAT instances into 4 partitions. Based on the experimental results of [9], for self-learnings, we only forward conflict clauses whose size is smaller than 9. The experimental results are obtained on a Linux PC with 3.3GHz AMD FX-6100 processors and 4 GB RAM.

Table 1: Test Generation Results for DLX Processor and OSES

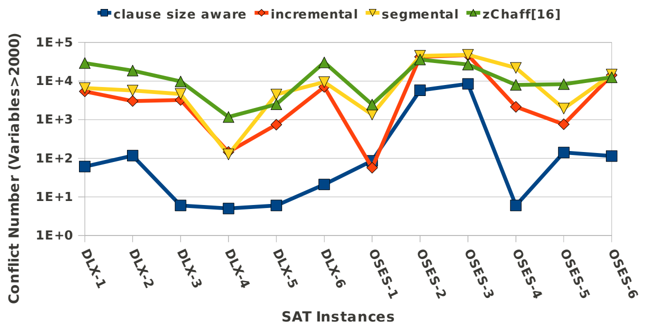
SAT Instance	MiniSAT[15] /zChaff[16]	Segmental Partitioning ([15]/[16])			Incremental Partitioning ([15]/[16])			C. S. A. Partitioning ([15]/[16])			Max Speedup
		cls	var	cls+var	cls	var	cls+var	cls	var	cls+var	
DLX-1	1.9/104.5	3.6/72.8	4.3/22.2	**3.3**/23.0	4.6/133.4	4.9/28.6	3.7/24.6	3.5/23.1	4.3/6.7	3.4/**6.2**	0.6/16.9
DLX-2	1.5/58.0	3.5/58.5	3.3/20.8	**3.1**/19.0	4.3/90.0	4.5/19.1	3.8/13.9	3.8/35.3	4.1/9.5	3.2/**5.7**	0.5/10.2
DLX-3	0.8/32.4	1.7/26.3	2.1/12.7	**1.6**/15.2	2.1/189.2	2.0/8.7	2.0/11.8	1.9/16.9	2.1/5.0	1.8/**3.2**	0.5/10.1
DLX-4	0.2/2.1	0.5/2.4	0.6/0.8	0.5/**0.5**	0.7/1.5	0.6/0.7	**0.5**/0.6	0.5/1.4	0.5/0.8	0.5/0.7	0.4/4.2
DLX-5	0.6/7.9	1.3/6.1	1.3/4.2	1.3/3.8	1.4/9.6	1.5/4.9	1.2/2.8	1.1/6.6	1.3/2.9	**1.0/2.7**	0.6/2.9
DLX-6	1.1/101.3	2.6/105.0	3.0/35.5	2.7/31.0	3.2/106.1	2.8/27.2	2.9/25.6	2.6/71.7	2.6/3.7	**2.3/2.9**	0.5/34.9
OSES-1	0.6/5.5	**0.3**/0.6	0.4/18.5	0.4/4.2	0.3/6.3	0.4/1.7	0.3/0.8	0.4/1.0	0.4/**0.5**	0.4/0.7	2.0/11.0
OSES-2	0.6/209.3	**0.5**/80.2	2.1/200.1	0.9/290.6	2.2/131.1	1.6/177.6	1.1/249.0	1.1/246.6	2.3/73.1	1.0/**24.8**	1.2/8.4
OSES-3	4.7/121.7	1.7/89.8	9.5/181.5	1.0/238.9	**0.9**/89.5	2.0/140.8	9.2/269.5	1.9/74.5	1.3/**6.7**	5.4/35.3	5.2/18.2
OSES-4	0.9/28.8	3.5/47.3	7.9/91.4	4.0/99.6	4.0/39.0	3.4/45.2	**1.0**/8.7	5.1/36.4	18.6/**0.8**	5.0/0.9	0.9/32.0
OSES-5	0.1/21.8	**0.2**/20.2	0.3/8.8	0.3/5.2	0.6/12.0	0.3/3.5	0.4/2.7	0.4/21.1	0.3/**0.6**	0.3/0.7	0.5/36.3
OSES-6	0.9/53.8	**0.3**/74.4	0.4/76.5	0.4/67.4	0.7/122.8	0.5/15.2	0.7/78.8	1.2/136.4	1.4/**1.7**	1.5/2.3	3.0/31.6

4.1 Checking BMC-Based SAT Instances

In this case study, we derive tests to validate two designs. The first one is a VLIW MIPS processor with five-stage pipelines. The main testing target is to investigate all the interactions between pipeline paths (i.e., ALU, DIV, FADD and MUL). The second design is an on-line stock exchange system (OSES), which mainly deals with various transactions of customers' orders (market orders and limit orders).



(a) The statistics of canceled levels



(b) The statistics of canceled variable assignments

Figure 8: Backtrack analysis for Table 1

Table 1 shows the test generation details of both designs using various self-learning heuristics. Due to the limit of the space, we only choose 6 typical tests for each design. All the other remaining tests show the similar results. In Table 1, the first column shows the index of SAT instances. For columns 2-11, we use the notation *MiniSAT/zChaff* to denote the test generation time (in seconds) using MiniSAT [15] and zChaff [16] respectively. The column 2 presents the test generation result without any improvement. Columns 3-5 present the result of segmental partitioning using different kinds of self-learnings. Column 3 shows the result only using conflict clauses based learning. Column 4 utilizes decision ordering as its single self-learning. Column 5 combines both learnings to further improve the checking performance. Similarly, columns 6-8 and columns 9-11 indicate the test generation time using incremental and clause size aware partitioning strategies individually with different self-learnings. For each SAT instance, we mark the best test generation time in bold font. It is important to note that all the results in columns 3-11 include the overhead of learning derivation. The final column presents the speedup of our methods over unmodified MiniSAT and zChaff solvers respectively using the following formula:

$$max\_speedup = \frac{column2}{MIN(column3,\ column4,\ \ldots,\ column11)} \qquad (5)$$

In this benchmark, it can be found that the self-learnings do not take the effect in MiniSAT searching in most cases, since the test generation time using MiniSAT without learnings is already small enough (i.e., smaller than 1 second). The introduction of extra learnings may degrade the search performance. However, the self-learnings are still effective for MiniSAT in the complex scenarios. For example, in the case of OSES-3, we can achieve 5.2 times improvement with self-learnings. Unlike MiniSAT, zChaff needs more time to derive tests in this benchmark. It is important to note that in this case our self-learning techniques can reduce the test generation time drastically. Especially the clause size aware partitioning method is best suited to this benchmark. When employing clause size aware partitioning heuristic, 5 out of 12 SAT instances can achieve the minimum test generation time by using decision ordering based self-learning only, and 6 out of 12 SAT instance can achieve the minimum test generation time because of the synergy of different kinds of learnings. By using our self-learning approaches in zChaff, a 3-37 times improvement can be achieved.

In the experiment, we assume that a long-distance backtrack involves a rollback with more than 20 decision levels or a cancel of more than 2000 variable assignments. Since zChaff needs more test generation time in this benchmark, the effect of our self-learnings can be observed more easily. Figure 8 analyzes the long-distance backtrack details for Table 1 using zChaff with different partitioning heuristics. For each partitioning method, we only investigated the composite learning using both conflict clauses and decision orderings. For each SAT instance, Figure 8a) counts the conflicts which require backtracks with more than 20 decision levels, and Figure 8b) indicates the number of conflicts which involve cancels of more than 2000 variable assignments. The result shows that,

Table 2: Test Generation Results for PIPE Processors

SAT Instance	MiniSAT[15] /zChaff[16]	Segmental Partitioning ([15]/[16])			Incremental Partitioning ([15]/[16])			C. S. A. Partitioning ([15]/[16])			Max Speedup
		cls	var	cls+var	cls	var	cls+var	cls	var	cls+var	
PIPE1	12/854	28/488	234/761	190/572	8/507	**1**/242	**1**/924	26/844	2/**7**	2/8	12.0/122.0
PIPE2	1/1090	**1**/2085	**1**/421	**1/1**	11/943	**1**/174	**1**/217	**1**/1124	**1**/11	**1**/11	1.0/1090.0
PIPE3	15/524	8/879	169/1185	144/1495	8/692	**1**/521	1/**142**	16/707	4/276	4/276	15.0/3.7
PIPE4	6/670	36/77	1055/162	**1**/12	5/311	NA/**4**	118/4	18/684	2/237	2/236	6.0/167.5
PIPE5	5/395	2/1345	3/2544	2/2613	9/90	2/595	10/1240	16/401	**2/55**	4/**55**	2.5/7.2
PIPE6	1/1	**1**/98	4/106	11/55	1/2	2/122	4/142	**1/1**	**1/1**	**1/1**	1.0/1.0
PIPE7	13/1117	284/505	322/2069	112/1184	450/681	6/1188	**4**/162	13/1059	4/**38**	4/39	3.3/29.4
PIPE8	2/73	**6/1**	44/69	91/61	7/110	NA/75	NA/76	9/73	**6**/30	11/30	0.3/73
PIPE9	30/3897	32/2522	2/404	2/**395**	43/421	3/1295	**1**/964	6/3874	1/1022	1/1020	30.0/9.9
PIPE10	174/907	112/647	672/973	4/497	111/446	**2**/139	**2/118**	17/661	4/148	4/147	87.0/7.7

Table 3: SAT Solving Results for UNSAT Instances of PIPE Processors

SAT Instance	MiniSAT[15] /zChaff[16]	Segmental Partitioning ([15]/[16])			Incremental Partitioning ([15]/[16])			C. S. A. Partitioning ([15]/[16])			Max Speedup
		cls	var	cls+var	cls	var	cls+var	cls	var	cls+var	
pipe2-ooo	0.3/0.06	0.1/**0.05**	**0.04**/0.08	0.05/0.09	0.1/0.06	0.1/0.10	0.1/0.10	0.3/0.08	0.1/0.19	0.1/0.13	7.5/1.2
pipe3-ooo	10.7/0.8	12.5/0.9	6.7/0.9	5.9/1.1	11.6/1.1	12.5/1.0	8.0/0.9	8.4/**0.8**	**0.8**/1.5	1.2/1.1	13.4/1.0
pipe4-ooo	94/6	142/4	118/3	111/4	114/**3**	77/4	197/4	50/6	13/5	**10**/5	9.4/2.0
pipe5-ooo	280/18	356/15	64/15	**47**/14	498/14	NA/12	1065/**11**	476/13	51/18	62/21	6.0/1.6
pipe6-ooo	731/69	NA/65	NA/64	NA/61	279/67	NA/50	NA/**48**	645/66	74/78	**70**/70	10.4/1.4
pipe7-ooo	1016/266	**702**/278	NA/285	NA/294	NA/303	NA/**199**	NA/208	NA/285	NA/304	NA/295	1.4/1.3
pipe8-ooo	NA/1107	NA/1330	NA/1083	NA/1104	NA/1387	NA/772	NA/**684**	NA/1326	NA/1354	NA/1334	1.4/1.6
pipe9-ooo	NA/1697	NA/1716	NA/1565	NA/1656	NA/1688	NA/**940**	NA/989	NA/1497	NA/2005	NA/1913	NA/1.8

compared to the method using zChaff, our self-learning techniques are effective to reduce the number of long-distance backtracks. Especially, the clause size aware partitioning approach achieves the best performance in most cases, which is consistent with the result presented in Table 1.

4.2 Checking Non-BMC-Based SAT Instances

Our approach can not only accelerate the test generation for SAT instances generated by BMC tools, but also it can be applied on SAT instances derived by other tools. Table 2 shows the test generation results for buggy variants of PIPE processors which are generated by a non-BMC tool [19]. The table structure is the same as the Table 1. The notation *NA* (i.e., INDETERMINATE) in the cells of the table indicates that the search run out of the specified memory when using the MiniSAT. In this case study, we can achieve an up to 87 times improvement using MiniSAT with our self-learning techniques. By using zChaff, we can achieve an up to 1090 times improvement, which is consistent with the results presented in Table 1. It can be observed that, although it cannot always achieve the best results, the clause size aware partition with composite learnings (i.e., column 11) outperforms other methods in most cases.

4.3 Discussion

From Section 4.1 and Section 4.2, we can find that MiniSAT needs less time than zChaff in test generation for both BMC-based and non-BMC-based methods. Since zChaff needs longer time for directed test generation, it implicitly needs to handle more long-distance backtracks during the SAT search. Therefore, we can clearly observe the effects (up to 1090 times improvement) of self-learning methods. As a state-of-art SAT solver, MiniSAT-2.2 can efficiently solve the simple SAT instances with less long-distance backtracks. But it does not mean that our self-learnings are not beneficial during the MiniSAT searching. In Table 2, we can find that the more complex the test generation problem is for MiniSAT, the more speedup can be achieved using our self-learning methods. For example, the test generation for PIPE10 costs 174 seconds without learnings, however by using the composite learnings as well as the incremental partitioning, the modified MiniSAT only needs 2 seconds to achieve the directed tests.

As shown in Section 4.1 and Section 4.2, the clause size aware partitioning coupled with the composite learning can achieve the best test generation time generally. However, no matter what SAT solvers are used for test generation, it is true that no single heuristic can guarantee the best performance for all cases. Resorting to the multi-processing capability is a feasible way to address this problem. Since most present computers have multiple cores, we can check a SAT instance with different partitioning approaches on different cores using the similar parallel strategy proposed in [9]. It means that if one of the cores finishes first, all the remaining SAT searches on the other cores can be terminated safely. In this way, we can always achieve the best test generation performance.

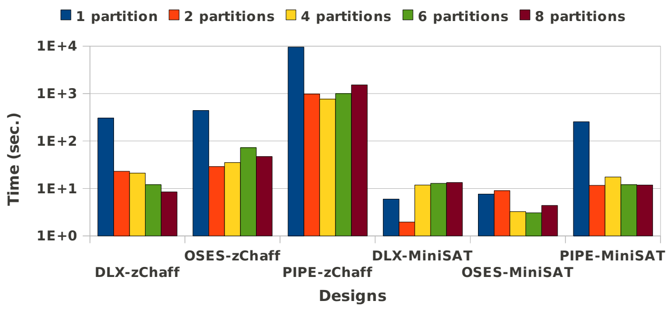


Figure 9: The analysis of the number of partitions

In our approach, the number of partitions plays an important role in determining the test generation time. We re-conducted all the benchmarks using both MiniSAT and zChaff with different number of partitions. Figure 9 shows the comparison results. The height of each bar in the figure indicates the overall minimum test generation time (i.e., the best one in all the 9 possible options as shown in columns 3-11 in Table 1 and Table 2) for all the SAT instances in the same benchmark. For example, the blue bar in the cluster *DLX-zChaff* indicates the overall minimum test generation time for all the 6 DLX SAT instances using zChaff. It can be found that MiniSAT generally needs less time than zChaff to derive tests. For MiniSAT, the approaches using 2 partitions or 4 partitions are good enough for test generation. And for zChaff, we can find that all the cases using 2, 4, 6, 8 partitions can achieve a drastic improvement for the three benchmarks. Especially, the methods using 4 partitions show a better performance for these three benchmarks.

Since our approach can effectively reduce long-distance backtracks, our self-learning can also be applied in the checking of UNSAT instances. We applied our self-learning and partition approaches on the *pipe_ooo* benchmarks generated by [19]. Here we only selected 8 out of 14 cases since the unselected ones cannot be solved by both MiniSAT and zChaff in one hour. Table 3 shows the checking result. Interestingly, we can find that zChaff outperforms MiniSAT in this benchmark, and our self-learning techniques can achieve an up to 13.4 times improvement.

5. CONCLUSIONS

Directed testing is promising for function validation, since coverage requirements can be achieved using fewer tests and less simulation effort. However, most automatic directed test generation methods, especially SAT-based approaches, are impeded by the state space explosion problem as well as the requirement of expert knowledge. This paper presented an efficient approach which utilizes our self-learning techniques to accelerate test generation. By exploiting high quality self-learnings to avoid long-distance backtracks and improve satisfying assignment convergence rate, the test generation time using SAT-based methods can be drastically reduced. Since our approach does not need the structure and behavior information of designs, it can be fully automated. The experimental results using various benchmarks demonstrated the effectiveness (by several orders of magnitude) of our approach.

6. ACKNOWLEDGEMENTS

This work was partially supported by the grants from Doctoral Fund of Ministry of Education of China 20110076120025, State High-Tech Development Plan of China 2011AA010101 and Natural Science Foundation of China 61021004, 90818024.

7. REFERENCES

[1] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME)*, pages 254–268, 2005.

[2] A. Biere, A. Cimatti, and E. M. Clarke. Bounded model checking. *Advances in Computers*, 58(3):118–149, 2003.

[3] K. Chandrasekar and M. S. Hsiao. Integration of learning techniques into incremental satisfiability for efficient path-delay fault test generation. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 1002–1007, 2005.

[4] M. Chen and P. Mishra. Functional test generation using efficient property clustering and learning techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 29(3):396–404, 2010.

[5] M. Chen and P. Mishra. Property learning techniques for efficient generation of directed tests. *IEEE Transactions on Computers*, 60(6):852–864, 2011.

[6] M. Chen and P. Mishra. Decision ordering based property decomposition for functional test generation. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 167–172, 2011.

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

[8] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem proving. *Communication of the ACM*, 5:394–397, 1962.

[9] Y. Hamadi, S. Jabbour and L. Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 6(4):245–262, 2009.

[10] F. Lu, Li-C. Wang, K. Cheng, and R. Huang. A circuit SAT solver with signal correlation guided learning. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 10892–10897, 2003.

[11] NuSMV. Available from: http://nusmv.irst.itc.it/.

[12] S. Fine and A. Ziv. Coverage directed test generation for functional verification using bayesian networks. In *Proceedings of Design Automation Conference (DAC)*, pages 286–291, 2003.

[13] J. P. Marques-Silva and K. A. Sakallah. The impact of branching heuristics in propositional satisfiability. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence*, pages 62–74, 1999.

[14] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceeding of Design Automation Conference (DAC)*, pages 530–535, 2001.

[15] MiniSAT-2.2.0. Available from: http://minisat.se/.

[16] zChaff. Available from: http://www.princeton.edu/˜chaff/zchaff.html.

[17] O. Strichman. Tuning SAT checkers for bounded model checking. In *Proceedings of Computer Aided Verification (CAV)*, pages 480–494, 2000.

[18] O. Strichman. Pruning techniques for the SAT-based bounded model checking problem. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME)*, pages 58–70, 2001.

[19] M. N. Velev. Automatic abstraction of equations in a logic of equality. In *Proceedings of Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, pages 196–213, 2003.

[20] C. Wang, H. Jin, G. D. Hachtel, and F. Somenzi. Refining the SAT decision ordering for bounded model checking. In *Proceedings of Design Automation Conference (DAC)*, pages 535–538, 2004.

[21] L. Zhang, M. R. Prasad, and M. S. Hsiao. Incremental deductive & inductive reasoning for SAT-based bounded model checking. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 502–509, 2004.

[22] M. Chen, X. Qin, H. Koo and P. Mishra. *System-Level Validation: High-Level Modeling and Directed Test Generation Techniques*. Springer, 2012.

[23] M. Chen, X. Qiu, W. Xu, L. Wang, J. Zhao and X. Li. UML activity diagram-based automatic test case generation for Java programs. *The Computer Journal* , 52(5):545–556, 2009.