Coverage-directed test generation through automatic constraint extraction

Onur Guzey, Li-C. Wang
Department of ECE, UC-Santa Barbara
oguzey, licwang@ece.ucsb.edu

***Abstract*— Generating tests to achieve high coverage in simulation-based functional verification can be very challenging. Constrained-random and coverage-directed test generation methods have been proposed and shown with various degrees of success. In this paper, we propose a new tool built on top of an existing constrained random test generation framework. The goal of this tool is to extract constraints from simulation data for improving controllability of internal signals. We present two automatic constraint extraction algorithms. Extracted constraints can be put back into constrained test-bench to generate tests for simultaneously controlling multiple signals. We demonstrate the effectiveness and scalability of the constraint extraction tool based on experiments on OpenSparc T1 microprocessor.**

I. Introduction

Functional verification continues to be one of the most expensive and time-consuming components in a typical design process. Practical functional verification relies on extensive simulation of directed and/or guided random tests due to its flexibility and scalability. Although simulation-based verification can be very effective, its success both in terms of total effort spent and final verification coverage achieved depends heavily on the quality of the tests in use. Effective tests can achieve higher verification coverage in shorter time, which saves engineering resources and improves confidence on the quality of the design under verification (DUV). However, generating effective tests for complex designs has always been a challenging problem.

In general, functional test generation can take one of three forms:

- Directed tests manually written by designers
- Constrained-random tests [1] instantiated randomly from biased and constrained test-benches developed by designers
- Automated coverage-directed tests [2], [3] dynamically generated by an algorithm to achieve higher verification coverage

Directed tests are written to cover corner cases and important features of a design. Writing directed tests has been a dominant test generation methodology even with the emergence of constrained-random test generation. Directed tests are crucial for verification as in many cases they are the only tests that can reach corner cases. This important advantage is partly offset by the amount of manual effort required for writing these tests. To write successful directed tests, engineers need to have a very good understanding of the design. This can be difficult for a verification engineer who often may not have all the required knowledge on the design. Constrained-random test generation alleviates part of the problem by producing a large number of controlled random tests where many verification targets can be fortuitously covered. This reduces the need for directed tests. However, for complex designs, achieving a required coverage goal even with both approaches can still be very challenging.

Limitations of the constrained-random approach led to the development of coverage-directed test generation (CDTG) techniques. These techniques dynamically analyze coverage results and automatically adapt the test generation process to improve the coverage. Some techniques utilize knowledge from high-level specifications or restricted symbolic simulation to help generate directed tests. CDTG process is guided by the coverage metric and different metrics such as state coverage [4], transition coverage [4] and coverage hole-analysis have been investigated in the literature.

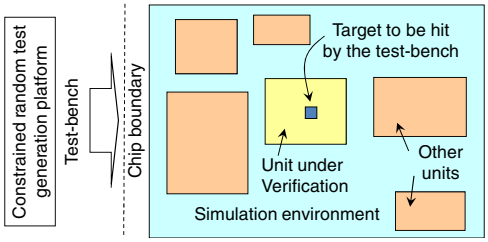

Fig. 1. **The issues of complexity and accessibility in a simulation environment can make test-bench development very challenging**

In this paper, we propose a novel approach that is built on top of an existing constrained-random test generation framework for coverage-directed test generation. At the core of this approach is an automatic tool that analyzes existing simulation data and extracts input constraints. These constraints are then used to control internal signals. By enhancing internal signal controllability, our approach essentially allows designers to specify constraints on internal signals in addition to primary inputs. This can greatly enhance the ability to modify a test-bench to cover specific verification targets. Because constraints are extracted from simulation data, our approach does not require a formal specification [3] or any structural search on the design [5]. The analysis is not based on a design so design's structural complexity has little impact on the effectiveness of our tool. In our approach, most parts of the design are essentially treated as blackboxes. Figure 1 illustrates this concept. The surrounding units essentially are ignored in the analysis. Only the simulation trace on the signals of the unit under verification is required.

Figure 2 summarizes our approach. Verification takes place in a constrained-random verification environment. A commercial constraint-solver generates constrained random tests based on the given constraints. These tests are applied to the design to produce simulation trace and coverage results. The trace and the results are analyzed by the coverage-directed test generation tool. Targets that have been missed so far are identified. Internal signals intended to be used to cover those targets are collected. Constraints to control those signals are generated by the automatic constraint extraction tool based on the simulation traces obtained so far. This process can iterate until desired coverage is achieved. Our approach can fit very well with existing functional verification flows such as the one reported in [6]. The constraint extraction tool can just be an add-on tool without major changes to an existing constrained verification methodology. For any proposed verification tool to be practical, scalability is a must. Hence, in this work we choose to conduct experiments based on OpenSparc T1 microprocessor. We will demonstrate the effectiveness of our approach based on this complex design and a commercial constrained verification framework.

The rest of the paper is organized as follows. Section II briefly reviews some of the coverage-directed test generation methods. Section III explains our approach to the signal controllability problem. Sections IV and V describe the development of two different automatic constraint extraction algorithms. Section VI discusses coverage experiments and Section VII concludes the paper.

1-4244-1480-6/07/$25.00 ©2007 IEEE

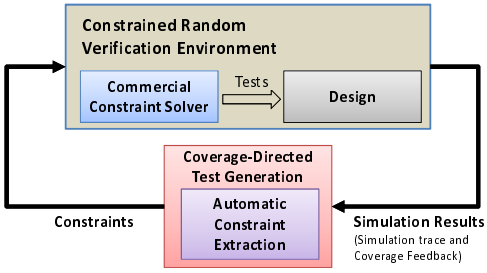

Fig. 2. **Summary of our proposed approach**

II. Coverage-directed test generation (CDTG)

Automated CDTG methods differ by their test generation algorithms and types of tests they produce. Geist et al. [5] propose using restricted symbolic simulation to avoid exponential blow-up problems faced by other formal methods. If high-level specification [3] [8] or a finite-state model [4] is readily available, they have been shown to be useful in test generation. Bayesian networks [9] and inductive logic programming [10] have been studied for learning based CDTG methods. These methods work with test program generation tools that generate high-level processor test programs. Test programs include variables like instructions, source and target registers. Machine learning methods are used to model the relationship between the coverage results and the variables in the test program generator so that the test generation process can be more effectively guided.

Our approach relies on automatic constraint extraction from simulation data and hence, differs from all the previously-proposed CDTG methods. However, it is more suitable to treat our work as an effort to complement existing approaches rather than to replace them. Next, we will explain the core of the constraint extraction problem.

III. Signal controllability problem

Through a test-bench only inputs of the DUV are directly controllable. In contrast, coverage targets are often defined in terms of internal signals or internal signals have to be controlled in order to cover them. Intuitively, if we can have control over internal signals, this controllability helps in generating coverage-directed tests. This brings up the fundamental question; how can we control internal signals from primary inputs? In a constrained-random verification environment, controllability of an internal signal means that we need to identify the input constraints to make the signal to have the value 1 (or 0). Such a constraint can be added into a random test-bench to cause the desired value on the signal when needed.

The obvious approach for getting the input constraints is through Automatic Test Pattern Generation (ATPG). For an ATPG tool to work, first we need to have a proper model (such as a gate-level model). Moreover, despite decades of effort to develop efficient functional ATPG tools, it remains difficult for such a tool to overcome the complexity of search on today's complex designs. Therefore, in this work we put aside the ATPG approach and focus on developing an alternative.

A. Search for input constraints

In this work, we assume that an input variable x represents a primary input signal y at a given clock cycle c. Without loss of generality, assume that we are given a set of n input variables $I = \{x_1, \ldots, x_n\}$ where each input can take on a value of 0, 1, or X (don't care). A *constraint* is a set of literals $\mathcal{L} = \{l_1, \ldots, l_m\}$ where each literal l_i is either x_j or $\overline{x_j}$ for some $j, 1 \le j \le n$. If l_i is x_j, then there is no literal $l_k \in L, l_k \neq l_i$ such that l_k is $\overline{x_j}$, and vice versa. We denote the size of the constraint as $|L| = m$. An *input pattern* P is an assignment of $\{0,1,X\}$ on all the input variables. If an input pattern does not contain X, we call it a *complete pattern*.

We assume that a test-bench $\mathcal{T}$ is a 4-tuple (H, I, L, B). H is a fixed test as the initialization sequence. I is the set of input variables. For N primary inputs over K cycles, we have in total $N \times K$ input variables. L is a constraint enforced on the input variables. B is a *random biasing scheme* defined on the primary inputs to enforce certain input distribution. For example, a primary input y may be biased toward 1 with a probability 0.9. However, on cycle 5, the constraint may enforce $y = 0$. A test t is 2-tuple (H, P) where P is an input pattern on I generated based on L and B.

Suppose we are given a simulation trace. A trace includes the values on the primary inputs and selected internal signals over a fixed number of cycles. This number can be large. Given an internal signal z, suppose this signal is included in the trace. Given an objective $z = a$ where $a = 0$ or 1, our goal is to identify a constraint $L(z)$ by analyzing the simulation trace such that if we produce q tests $\{t_1, \ldots, t_q\}$ from $\mathcal{T}_z = (H, I, L(z), B)$, at the last cycle of each test, we would have a very high probability to observe $z = a$. Let p out of the q achieve this objective. We note the success rate of $L(z)$ as $(p/q) * 100\%$. Note that for the objective, we may want to identify a set of constraints $\{L_1(z), \ldots, L_k(z)\}$ instead of just one.

B. Extracting constraints from patterns

The constraint extraction problem can be simplified by employing the following scheme. Suppose our objective is $z = a$. Given a simulation trace, we first identify all the cycles where z transitions from $1 - a$ to a. From each cycle inclusively, we look back in time w cycles and collect the input pattern over these w cycles. Suppose we collect k patterns $P_1, \ldots, P_k$. Then, the problem of extracting the constraints for $z = a$ can be simplified as the problem of extracting the constraints from $P_1, \ldots, P_k$. Figure 3 illustrates this simplification.

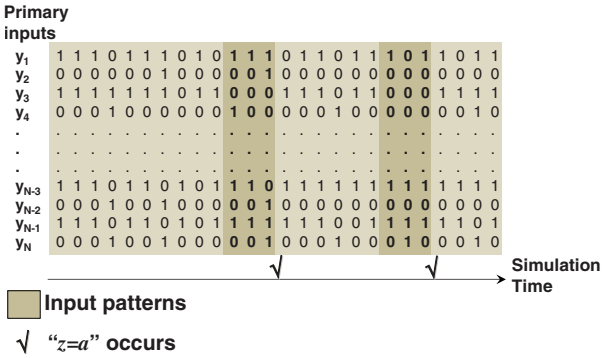

Fig. 3. **Target dataset to extract patterns** $P_1, \ldots, P_k$

Figure 4 uses a simple example to illustrate the simplified version of the constraint extraction problem. Suppose we are given four patterns p_1, p_2, p_3, p_4 on 8 variables $x_1, \ldots, x_8$. Suppose $x_1, \ldots, x_4$ correspond to primary inputs $y_1, \ldots, y_4$ in the first cycle and $x_5, \ldots, x_8$ correspond to them in the second cycle. By analyzing the patterns, we may extract various constraints.

For example, c_1 and c_2 are consistent with three patterns while c_3 and c_4 are consistent with only one pattern. All of the constraints are valid constraints that can be extracted from the collection of patterns. For this example one may consider c_1 as the most desirable choice because it is consistent with the most patterns and constrains on the fewest inputs (i.e. has the smallest size). When combining multiple constraints from multiple signals, a constraint with a smaller size

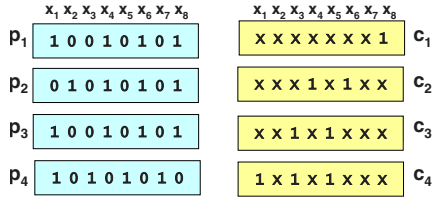

Fig. 4. **Extracting constraints from patterns**

would have less chance to be in conflict with others. However, c_2 may result in a higher success rate by being more restrictive.

There are many other constraints that can be extracted and are not shown in the figure. This simple example illustrates two points in the problem: (1) There are many different constraints that can be extracted from a set of patterns and we may not be able to get all of them. (2) We need an effective way to assess the merit of a constraint.

C. The Lattice view of the search space

Given a pattern P, we say that a constraint $L = \{l_1, \ldots, l_m\}$ is consistent with P if the following is true. For each variable x, if there is a literal $l_i = x$ in L then we have $x = 1$ in P. If $l_i = \bar{x}$ is in L, then we have $x = 0$ in P. In other words, the constraint is a subset of assignment of values $0, 1$ to some input variables. Given a set of k patterns $P_1, \ldots, P_k$, suppose the constraint L is consistent with j out of k patterns. Then, notice that any subset $L_s \subset L$ is also consistent with at least j patterns. We call j the *support* of L. We call j/k the *frequency* of L.

For a fixed k, the support of a constraint and its frequency means the same thing. Suppose now we want to find the constraints with support $j+1$. Notice that such a constraint may be a subset of L. Moreover, there can be more than one such constraints. Consider the example in Figure 4 again. The constraint $\{x_1, \overline{x_2}, \overline{x_3}\}$ is consistent with P_1 and P_3 and hence, has a support 2. Suppose we want to find the constraint with support 3. We see that the subset $\{x_1\} \subset \{x_1, \overline{x_2}, \overline{x_3}\}$ is consistent with P_1, P_3 and P_4.

Based on a given constraint L with a support j, suppose our goal is to identify all constraints with support $j' > j$. This problem is illustrated with a Lattice view as finding a *frequency cut* in the Lattice. Figure 5 depicts this concept. For any subset $L_s \subset L$, if L_s has the support j', then all subsets of L_s also have at least the support j'. This is called the *downward closure property*. Because of this property, the search space can be represented as a Lattice. Given j', to find all the constraints (or subsets of L) with the support at least j' is to find a *frequency cut* in the Lattice. Then, all subsets above this cut have a support value at least j'.

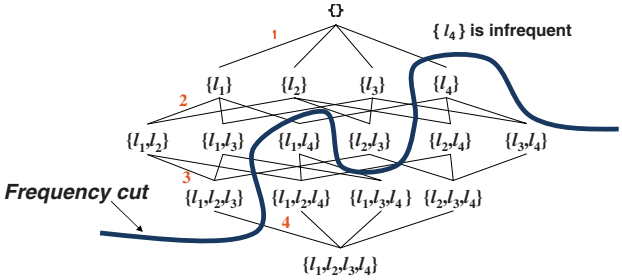

Fig. 5. **Finding the frequency cut in a Lattice**

D. Finding maximal size constraints

Given a set of patterns $\{P_1, \ldots, P_k\}$, initially we can let $L_1 = P_1$, …, $L_k = P_k$. For a pattern P, setting $L = P$ means that for each variable x, if $x = 1$ is in P, we include the literal x in L and if $x = 0$ is in P, we include the literal $\bar{x}$ in L. Without loss of generality, we assume that each L_i is consistent with only the pattern P_i and hence has support value 1. If not, we would have less than k constraints to start with and some constraints have support greater than 1. Recall that these patterns are extracted based on the method described in Figure 3. It is very possible that a constraint from a pattern P_i is also consistent with many other patterns. To simplify the discussion, we assume each L_i has support 1 to begin with.

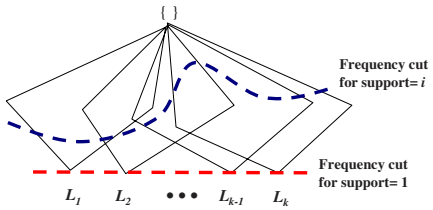

Fig. 6. **Finding a frequency cut in the combined Lattice**

Figure 6 illustrates the essence of the constraint extraction problem. The search space is the collection of Lattices where each is based on a constraint L_i as described in Figure 5. In this combined Lattice view, the nodes at the bottom have the least support value(s). By giving a support value requirement j, essentially we are moving the frequency cut upward in the Lattice. Suppose we have an algorithm to find such a cut based on a given support value requirement. Notice that all the constraints on the cut line satisfy three properties: (1) Their support value is equal to j. (2) They are all distinct. (3) Their sizes are maximal. For each such constraint L_{max}, all its subsets form a sub-Lattice and every subset has a support value at least j. All the constraints contain L_{max} as a subset has a support value less than j. Hence, L_{max} is a maximal-size constraint with a support value at least j. Hence, solving the constraint extraction problem can begin by finding the maximal constraints that satisfy a given support value j. In the next section, we will discuss the algorithm for finding those constraints. We will also show that maximal constraints are not the final answers to our constraint extraction problem. Further pruning is required on those constraints.

IV. Extracting Frequent Constraints

In the field of data mining, the problem of finding frequent *itemsets* is similar to the problem of finding frequent constraints described in the previous section. However, the search space for a typical itemset mining problem is quite different from the search space of our constraint extraction problem. In itemset mining [11], the search space can also be represented as a Lattice. Given a set of n items $\{e_1, \ldots, e_n\}$, an itemset is a subset of these items. If an itemset is frequent, then all its subsets are also at least as frequent as the itemset. This downward closure property allows the search space to be represented as a Lattice as described before.

In a practical application, an item can be a real item to be purchased in a market place. An itemset therefore represents a transaction. In such an application, although n can be very large, the size of each transaction is usually small and much smaller than n. In constraint extraction, each starting constraint is usually with the size of n where n is the number of input variables. This makes constraint extraction a much harder problem. For example, in a typical market data mining application, each transaction may contain up to tens of items. In constraint extraction, each starting constraint may consist of thousands of literals.

Because of the reason stated above, most of the itemset mining algorithm such as the popular Apriori algorithm [12] is not efficient for searching the constraint extraction space. Given a frequency

threshold, the Apriori algorithm performs a breath-first search on the Lattice starting from the empty set {}. The search stops at the frequency cut. This can be inefficient because the number of constraints that satisfy a frequency requirement can be very large.

Consider the example in Figure 4 again. Let L_1, L_2, L_3, L_4 be the constraints representing P_1, P_2, P_3, P_4, respectively. Suppose the required support is 3 (frequency = 75%). We see that $L_1 \cap L_2 \cap L_3 = \{\overline{x_3}, x_4, \overline{x_5}, x_6, \overline{x_7}, x_8\} = L_1$. In addition, we have $L_1 \cap L_3 \cap L_4 = \{x_1, \overline{x_2}\} = L_2$. We see that there are $2^6 - 2 = 62$ subsets of L_1, excluding itself and the empty set. All these subsets have a support at least 3. This number is exponential in the size of the maximal constraint. On the other hand, we see that there are only two maximal constraints. Notice that $L_1 \cap L_2 \cap L_4 = \phi$ and $L_2 \cap L_3 \cap L_4 = \phi$.

The example explains that the number of the maximal constraints can be substantially less than the number of constraints that satisfy a given frequency threshold. Moreover, if the required support is set at 2, then we would have 5 maximal constraints. This is because $L_2 \cap L_4 = \phi$ and all other five pairs share common literals. This indicates that with a low support value, the number of the maximal constraints can also explode. Recall in Figure 6 that we start with the k constraints with minimal support value(s). This k can be on the order of thousands. Hence, it is likely that with a low support, the number of maximal constraints is an exponentially large number.

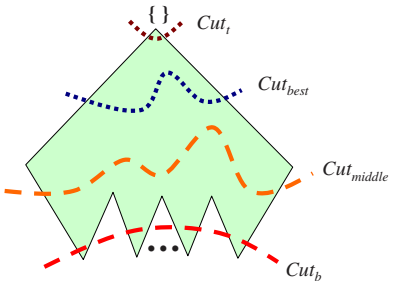

Fig. 7. **Finding the best frequency cut**

The discussion above can be summarized with Figure 7. In this figure, 4 frequency cuts are shown. With a very low support, we obtain a cut Cut_b. The number of maximal constraints may be limited. However, the sizes of these constraints are large. Large-size constraints are not favored because it is hard to combine them together for multi-signal controllability.

If we set the support very large, say close to k, then essentially we may find that no constraint is above the cut. This cut is shown as Cut_t in the figure. Between Cut_b and Cut_t, there can be many choices. Cut_{middle} can be the result of setting the support not large enough. In this case, we may end up with an exponential number of maximal constraints. Then, no known algorithm would be able to finish finding the cut. Therefore, our objective should be setting the support high enough (but not too high) such that a cut Cut_{best} can be found with a reasonable number of maximal constraints.

Even though the number of maximal constraints along the Cut_{best} is not large, the number of total constraints above the cut can still be an exponentially large number. Therefore, we would not want to do a breath-first search like the Apriori algorithm to find those maximal constraints. Instead, we can do a depth-first search.

Mafia [13] is a depth-first maximal frequent itemset algorithm. Essentially, the search path of this algorithm travels along the desired cut and avoids processing most of the constraints above the cut. Hence, if the number of the maximal constraints is not too large, the algorithm can be efficient. To control this number, we need to find the best support value threshold. This can be easily done by starting with a large value and stepping down. For example, we can start with finding a 95% frequency cut. If this returns an empty set or exceeds a certain run time limit (say 1 minute) without finishing, we abort the run and lower the frequency to 90%. This process continues until we find a set of maximal constraints.

A. The overall algorithm

Figure 8 summarizes the overall constraint extraction algorithm. Simulation traces generated from constrained random test-benches are parsed into internal data structures that our algorithm can handle. Then, for each signal controllability objective $z = a$, we produce the *target dataset* that is the set of patterns based on a given number of cycles. This is described in Figure 3. Then, the depth-first maximal frequency itemset algorithm is run to extract maximal constraints from the target dataset.

```
parse simulation trace
generate internal data structures
foreach objective z = a do
    identify the set of patterns {P_1, ..., P_k} and this set becomes
    the target dataset
    Find the set of maximal constraints with high support
    foreach maximal constraint do
        perform the significance calculation based on the
        non-target dataset
    end
    foreach significant constraint do
        perform minimization of constraint sizes
    end
    output the final constraints
end
```

Fig. 8. **Non-Query based constraint extraction algorithm**

Note that the maximal constraints are extracted based on only the target dataset. The non-target portion of the data is not used. We use this non-target dataset in the significance calculation described below.

Suppose there is a primary input y that is set at 1 (or always toggle). Then, y will become a constrained variable on all cycles in all maximal constraints. By looking at only the target dataset, we would not know that this is because y is always 1. This can only be figured out by checking the non-target dataset as well. We call such a check the *significance calculation*.

Given a maximal constraint L, being frequent (high support value) in the target dataset is not sufficient for it to be meaningful. This constraint can be frequent in the non-target dataset as well. If this is the case, the constraint becomes insignificant. To eliminate insignificant constraints, a significance value is calculated for each maximal constraint. The significance of a constraint L is defined as:

sig(L) = support$_t$(L) / support$_w$(L)

support$_t$: support of the constraint in the target dataset

support$_w$: support of the constraint in the entire dataset

To calculate support$_w$(L), we need to extract the pattern set from the non-target dataset. Figure 9 illustrates this extraction process. The non-target pattern set $\{Q_1, \ldots, Q_m\}$ are constructed and is union with the target pattern set $\{P_1, \ldots, P_k\}$. Then, support$_w$(L) is calculated based on the entire pattern set.

We use the significance calculation to remove a large number of the maximal constraints. In general, we only keep the top i most *significant* constraints for a small i (such as 10 or 20). Then, we minimize the sizes of these constraints through another procedure.

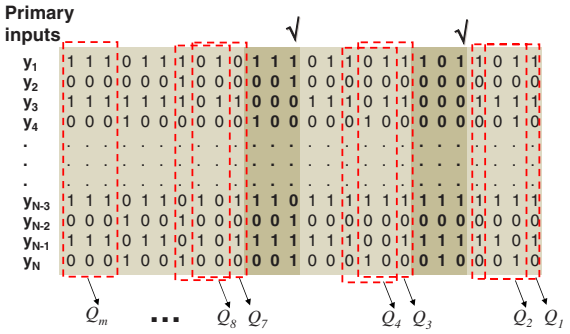

Fig. 9. **The patterns from the non-target dataset**

We note that the significance calculation is done on every maximal constraint. Usually, the number of the maximal constraints for an objective can be large (on the order of thousands). Hence, the calculation can be expensive. In the data mining field, several methods are proposed to speed up the support calculation for a large number of itemsets. They include using finite state machines and sliding windows [14]. In our implementation, we use the bitset based representation and method proposed in [13].

B. Size minimization

As discussed before, our ultimate goal is to use constraints to control multiple internal signals simultaneously. Because of this, the size of a constraint for each individual objective is an important consideration. Even though a constraint is significant for a given objective, a large size constraint, when combining with the constraints from other objectives, is likely to cause a conflict among them. This degrades our ability to use the constraint for controlling multiple signals. Hence, small size constraints are favored.

Given a constraint, our goal is to reduce its size without impacting its significance value. In this size minimization step, literals are removed one by one from a constraint. After removing a literal, significance is re-calculated. If removal does not decrease the significance value, the literal can be safely removed. Otherwise it remains in the constraint. This procedure continues until we cannot remove any literal without causing a drop in the significance value. As a result of this procedure, the number of variables in each constraint drops considerably and conflicts become less likely.

C. Experimental Results

1) OpenSparc T1: One of the important steps for developing a successful tool in functional test-bench automation is identification of a proper design driver. Often, the core of the problem in the tool development is handling complexity. Without a properly complex design, we might not have the opportunity to observe the limitation of a proposed approach, which may lead us to misunderstand the true essence of the problem. The publish of OpenSPARC [7] has greatly alleviated the burden of searching for a suitable design driver. OpenSPARC T1 is a 64-bit open-source microprocessor designed by SUN Microsystems.

OpenSPARC T1 has eight cores and each core can support up to four threads for a total of thirty-two threads. Within OpenSPARC T1 we chose Execution Unit(EXU) as our first target because of its complexity and diversity. It features arithmetic sub-units, control blocks and a large number registers which makes it a suitable target for this study. EXU is made of more than 10000 lines of rtl code and 32 submodules. There are four sub-units: an integer arithmetic logic unit, an integer multiplier unit, an integer divider and a shifter. Execution control logic and bypass logic blocks generate the control signals. EXU's internal integer register file features 128 registers.

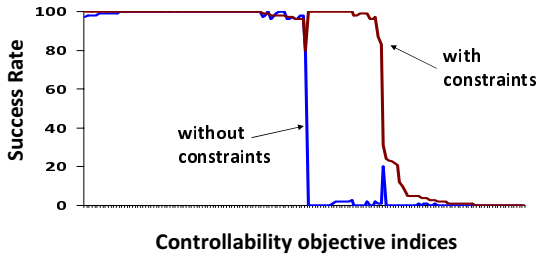

Fig. 10. **Comparison between non-query based and random results**

2) Results: We utilize one of the supplied tests that makes use of the execution unit as our initial test-bench. We simulate more than 15000 clock cycles to obtain the initial dataset for mining. From this dataset a constrained-random test-bench is generated. In this constrained test-bench all input signals, other than the system clock, are left random with bias values similar to the value distributions in the initial dataset.

For each signal we set the controllability objective as 1 or 0, and find constraints that make it 1 or 0 at the end of a specific clock cycle (after the initialization sequence). For each objective, the top 20 constraints based on their significance values are selected. Each constraint is applied back to constrain the initial test-bench. To evaluate the success rate, the modified test-bench is used to generate 100 distinct tests, each with the same initialization sequence. These 100 tests are simulated and the number of times the signal controllability objective is achieved is used as the success rate. Then, for each objective we report the highest success rate from the 20 trials. We compare this success rate to the rate using the original test-bench. Figure 10 shows the comparison result for 164 controllability objectives. These 164 objectives are based on the top level signals in the execution unit including top-level unit connections and outputs. Average run-time of the algorithms, although very dependent on the parameters in use, was approximately 5 minutes per objective.

We performed additional experiments to illustrate the effect of parameters used in the algorithm. Figure 11 shows the effect of support value on the number of maximal constraints and the average number of literals per constraint. These results concur with the explanation based on the Lattice view before. Moreover, the effect of the minimization process is given in Figure 12 based on constraints for one particular objective. Size minimization decreases the number of literals in each constraint and may result in a drop in success rate. However, we see that a number of constraints remain effective after size minimization. Finding those constraints which are effective and have much fewer literals can be seen as the goal of minimization. For each objective, we only need to keep a few effective constraints, not necessary all of them.

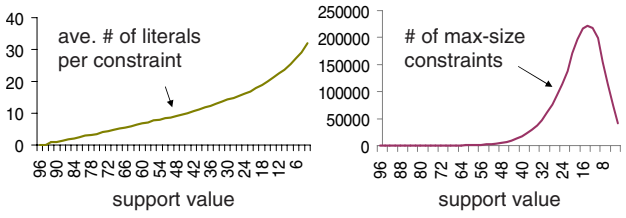

Fig. 11. **Effect of support value**

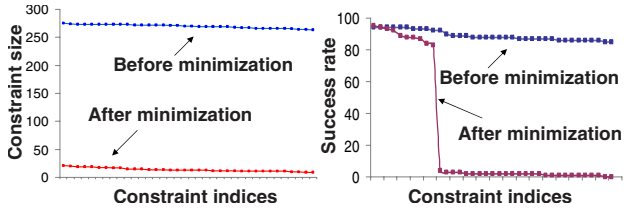

Fig. 12. **Effect of size minimization**

V. Query-based constraint size minimization

In this section, we describe a different constraint extraction algorithm that is *query-based*. Figure 13 illustrates the main difference between this algorithm and the *non-query-based* algorithm described in the previous section. In a query-based algorithm, we are allowed to query an *oracle* for the quality of a given constraint. Suppose that we begin with an initial constraint $L = \{l_1, \ldots, l_m\}$ for an objective $z = a$. For any subset $L_s \subset L$, we are allowed to ask the oracle the following question: Does L_s contain all the necessary literals to cause $z = a$? If L_s indeed contains all the necessary literals, the oracle would answer yes. If not, the oracle would answer yes with an error probability p_{yes}. This probability is not fixed and is inversely proportional to the number of necessary literals missing in L_s. With the help of such an oracle, a query-based algorithm can be designed for minimizing the size of a given constraint:

1) Input: L
2) Select a subset $L_s \subset L$ and ask the oracle.
3) If the answer is yes, then remove all literals in $L - L_s$ by setting $L = L_s$. Otherwise, go back to step 2.
4) Repeat steps 2 and 3

To make the algorithm efficient, we need to consider the following objectives: (1) Minimize the number of queries to the oracle. (2) Minimize the number of necessary literals wrongfully removed due to the errors made by the oracle. Suppose we have developed such an efficient query-based algorithm for constraint size minimization. Then, this algorithm can be used to develop a query-based constraint extraction algorithm as explained below.

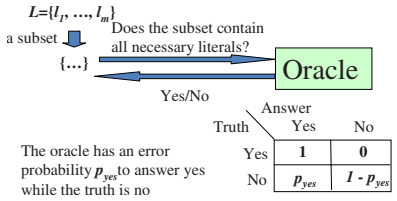

Fig. 13. **Illustration of queries to an oracle**

In our application, the oracle is implemented as the following. L_s is taken and applied to the test-bench $\mathcal{T}$ to produce h tests $t_1, \ldots, t_h$. These tests are simulated on the design and the objective $z = a$ is checked. The success rate (i.e. among these h tests, how many actually achieve the objective $z = a$) is calculated and compared to the success rate based on the original constraint L. If the success rate of L_s is not smaller than the success rate of L, then the answer is yes. Otherwise, the answer is no.

In the formulation of Figure 13, we assume that there are a subset of literals in L that are necessary. When these literals are included, the success rate would never drop. In reality, this may not be true because the success rate for L is also calculated based on a set of random tests sampled from the test-bench. However, this is not a serious issue because we can always assume the success rate of L is 100% to begin with (without actually calculating it). Then, if the success rate of L_s is less than 100%, the oracle outputs no. This only makes the oracle to be more conservative to answer yes.

We note that the more necessary literals the L_s misses, the lower the error probability p_{yes}. This is because missing more necessary literals means harder to produce tests that achieve the objective. With a lower p_{yes}, the oracle tends to answer no more.

A. Query-based constraint extraction

The query-based constraint extraction is based on a constraint size minimization algorithm. The overall algorithm is depicted in Figure 14. For each objective, the algorithm first identifies the target dataset of patterns. Then, an initial constraint L_i is nothing but one of these patterns P_i. The remaining steps try to minimize the size of this constraint by querying the oracle. To limit the number of queries, the literals in the constraint are ordered. This ordering follows the significance values of those literals. The significance calculation is based on the non-target dataset as before. Given the non-target patterns $\{Q_1, \ldots, Q_m\}$, we check how many patterns have the same assignment given by a literal (i.e. x_j means $x_j = 1$ and $\overline{x_j}$ means $x_j = 0$). Let this number be m'. Then, a literal is more significant if m'/m is smaller.

The literals are ordered from the least significant to the most significant. Then, removal of literals follows this ordered list. A number *block_size* is used to control the number of literals to be tried at any given iteration. This allows the removal process to be more aggressive than just removing one literal at a time.

When removing a block of *block_size* literals, the oracle may answer no because some necessary literals are removed. In this case, all removed literals are put back and then a smaller *block_size* is tried. This process repeat until *block_size* $= 1$. For *block_size* $= 1$, the literal is either removed from the constraint or removed from the *removal_set*, so that it will never be considered again. Hence, the inner while loop iterates no more than $\log(block\_size) \times |L_i|$ where $|L_i|$ is the size of the initial constraint. However, with a large *block_size*, a successful removal, it can dramatically reduce the number of queries to the oracle.

At the end of the inner while loop, a minimal-size constraint is found. Then, this constraint is compared with all the patterns in the pattern set. All patterns that are consistent with this constraint are removed from the set and never considered again. Because of this step, the query-based algorithm may find only a few constraints.

B. The chance of wrongfully removing a literal

A major concern in the algorithm is that, because the oracle may answer yes even though the truth is no, necessary literals can be wrongfully removed. However, this number should be small because removing a necessary literal reduces the probability that the oracle answers yes in the future.

For example, consider an initial constraint L with four necessary literals $\{x_1, x_2, x_3, x_4\}$. Suppose x_4 is wrongfully removed. Then, any constraint L_s that contains $\{x_1, x_2, x_3\}$ only has a 50% chance to generate the required input assignment $\{x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 1\}$ because x_4 is left random now. Suppose the oracle tries to simulate 10 tests based on L_s. The chance that these 10 tests all contain the required input assignment is $(0.5)^{10}$. Hence, the success rate for L_s is almost surely not 100%. Then, almost surely the oracle will answer no for L_s, which results in no additional removal.

The simple example above shows that when a necessary literal is wrongfully removed, it causes the probability of answering yes by the oracle to drop exponentially. Therefore, if a few necessary literals are removed, the algorithm almost surely would stop removing additional

```
parse simulation trace and generate internal data structures
foreach target objective z = a do
    identify the target dataset and find all the target patterns
    P = {P_1, ..., P_k}
    set block_size to an initial value (like 256)
    while P != ∅ do
        Select a pattern P_i from P and make the constraint
        L_i = P_i; order the literals in L_i by their significance
        add all literals to removal_set based on the ordering
        while removal_set != ∅ do
            remove the most insignificant block_size literals
            from the set removal_set
            remove these literals from L_i to produce a subset L_s,
            and query the oracle with L_s
            if answer is yes then
                make L_i = L_s
            else
                if block_size == 1 then
                    reset the block_size to its initial value
                else
                    put all literals back to removal_set and
                    divide the block_size by 2
                end
            end
        end
        remove all patterns from P that are consistent with L_i
    end
    output the set of constraints
end
```

Fig. 14. **Query based constraint extraction algorithm**

literals. This feature may result in a constraint whose size is not close to the minimal. However, it also ensures that a constraint found by the algorithm contains most of the necessary literals.

C. Experimental Results

For the query based algorithm we used the same experimental setup as in Section IV-C. Table I gives detailed results for the ouputs of the control block in the execution unit. Comparison between original test-bench simulation and constrained test-bench simulation for 164 signal controllability objectives is shown in Figure 15. Comparing this figure to Figure 10, we see a clear improvement.

Similar experiments were carried out on all top level signals within the Stream Processing Unit and the Floating Point Frontend Unit. Figures 16 and 17 show the comparison results. It should be noted that for a few targets the algorithm aborted because no transitions were found in the simulation data. These targets are not included in the results. We see that the query-based algorithm is powerful enough to find effective constraints for almost all signal objectives.

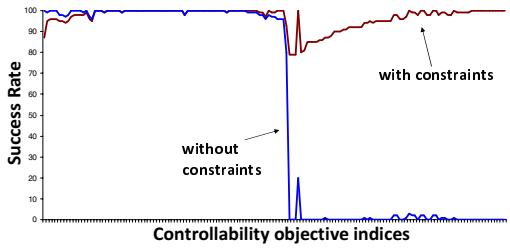

Fig. 15. **Comparison between results - Execution Unit**

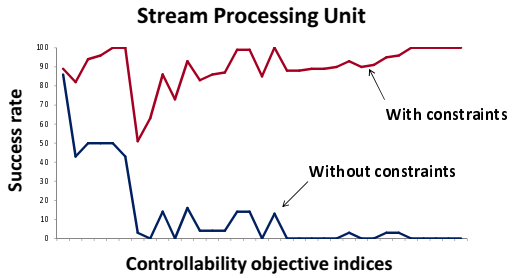

Fig. 16. **Comparison between results - Stream Processing Unit**

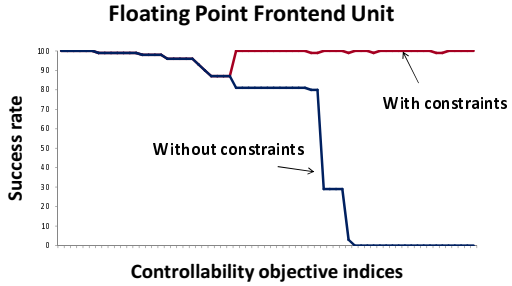

Fig. 17. **Comparison between - Floating Point Frontend Unit**

Although controlling individual signals is essential, ability to combine multiple constraints to control several signals simultaneously is more important for covering verification targets usually defined based on multiple signals. These targets are called cross-coverage targets. To investigate the potential of using the extracted constraints for multi-signal controllability we performed experiments by randomly choosing sets of signals and trying to find constraints that force all these signals to randomly selected values. Results for 250 randomly selected sets of 3 and 5 target events from the execution unit compared to random simulation are given in Figures 18 and 19. It should be noted that for experiments in this section and the following section we have not looked at the satisfiability of the targets, some of these targets may not be satisfiable because of architectural constraints. Hence, 100% success should not be expected. According to the results extracted constraints provide enhanced signal controllability for multiple signals.

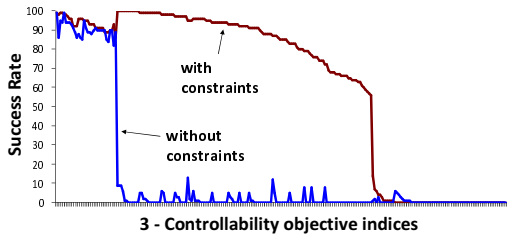

Fig. 18. **Comparison between 3 target results - Execution Unit**

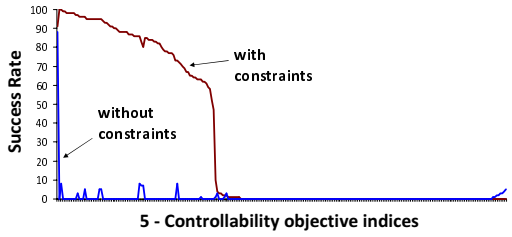

Fig. 19. **Comparison between 5 target results - Execution Unit**

TABLE I

Results for selected signals in the Execution Unit

Target Name	Forced to 0			Forced to 1			Target Name	Forced to 0			Forced to 1		
	t	c	Success %	t	c	Success %		t	c	Success %	t	c	Success %
shft-enshift-e-l	4	1	100	4	1	100	shft-op32-e	7	2	100	4	1	100
rml-instd-vld-w	3	1	98	3	2	100	irf-wen-w	5	1	100	30	17	98
ecc-rs2-use-rf-e	4	1	100	10	3	100	ecc-rs1-use-rf-e	4	1	100	9	2	100
byp-sel-restore-m	4	1	98	6	1	95	byp-sel-pipe-m	7	1	95	5	1	98
byp-sel-load-m	4	1	100	95	233	100	byp-rs3h-mux2-sel-um1	83	96	92	106	90	100
byp-rs3h-mux2-sel-rf	96	114	85	-	-	-	byp-rs3h-mux2-sel-e	4	1	85	95	114	100
byp-rs3h-mux1-sel-o.	74	90	92	82	96	100	byp-rs3h-mux1-sel-m	4	1	100	72	90	92
byp-rs3-mux2-sel-um1	6	1	100	18	24	100	byp-rs3-mux2-sel-rf	56	75	85	7	1	100
byp-rs3-mux1-sel-o.	19	24	91	7	1	100	byp-rs2-mux2-sel-um1	5	1	100	4	1	100
byp-rs2-mux2-sel-rf	4	1	99	6	1	100	byp-rs1-mux2-sel-um1	5	1	100	4	1	98
byp-rs1-mux2-sel-rf	5	1	98	7	1	100	byp-rs1-mux2-sel-e	3	1	100	56	89	98
byp-rcc-mux2-sel-rf	58	89	79	5	1	100	byp-rcc-mux2-sel-e	5	1	100	54	99	79
alu-sethi-inst-e	5	1	94	6	25	100	alu-out-sel-sum-e-l	4	1	99	8	38	95
alu-out-sel-shift-e-l	35	232	98	4	1	100	alu-out-sel-logic-e-l	8	38	93	3	1	100
alu-log-sel-or-e	4	1	100	35	96	100	alu-log-sel-move-e	8	30	94	7	2	98

c: Number of literals in the constraint; **t:** Constraint extraction time in minutes

VI. Coverage Experiments

The essense of coverage-directed test generation is increasing verification coverage for the design under verification. In this section we provide coverage results collected using the constraints extracted by the query-based constraint extraction algorithm. State coverage is selected as our coverage metric. For the experiments we randomly selected 2 different sets of 8 signals from the execution unit. Each of these 2 sets correspond to 2^8 states that are designated as verification targets that should be covered by the verification process for a total of 512 distinct verification targets. Tests that target each of these targets are generated by the automatic constraint extraction algorithms. Coverage results are given in Figure 20. Coverage results are compared with the original test-bench (labeled as random simulation) to show the improvement achieved by using coverage-directed tests. From the results it is clear that the extracted constraints help reach higher coverage in shorter amount of time.

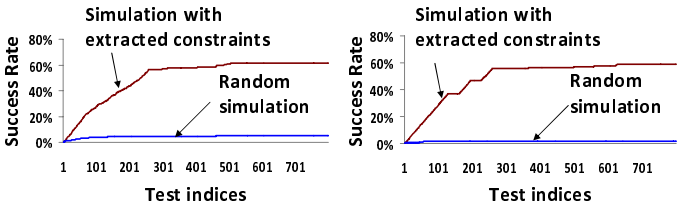

Fig. 20. **Coverage results compared with random simulation**

VII. Conclusion

In this paper we proposed novel automatic constraint extraction methods for increasing the efficiency and eventual success of simulation-based verification. Our methodology can be integrated into an existing verification flow as an add-on tool without major changes to the flow. Input constraints used to increase controllability are automatically extracted from simulation data without requiring knowledge about the structure of the design. We only assume that the designs are simulatable. Extracted constraints can be utilized to increase verification coverage by specifically targeting parts of the design according to selected coverage metrics. We implemented an experimental framework to demonstrate the practicality of our proposed approach. Experiments performed on OpenSparc T1 microprocessor show that our methodology can be used to greatly enhance signal controllability compared to the random simulation even for complex industrial designs.

In the near future, we plan to extend our experiments to other units of the OpenSparc T1 microprocessor. Different coverage metrics and their success with our overall methodology will also be investigated.

References

[1] J. Yuan, C. Pixley, A. Aziz and K. Albin. "A Framework for Constrained Functional Verification," Proc. 2003 *International conference on Computer-aided design* , pp. 142, 2003

[2] Gilly Nativ, Steven Mittermaier, Shmuel Ur, Avi Ziv. "Cost evaluation of coverage directed test generation for the IBM mainframe," Proc. 2001 *International Test Conference*, pp. 793-802, 2001

[3] Nina Saxena, Jacob A. Abraham, Avijit Saha. "Causality based generation of directed test cases," Proc. 2000 *Asia and South Pacific Design Automation Conference*, pp. 503-508, 2000

[4] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, R. Smeets. "A study in coverage-driven test generation," Proc. 1999 *Design Automation Conference*, pp. 970 - 975, 1999

[5] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, Y Wolfsthal. "Coverage-Directed Test Generation Using Symbolic Techniques," *International Conference on Formal Methods in Computer-Aided Design*, pp. 143-158, 1996

[6] A. Gluska, "Coverage-Oriented Verification of Banias," Proc. 2003 *Design Automation Conference*, pp. 280-285, 2003

[7] OpenSPARC at http://www.sun.com/processors/opensparc/

[8] P. Mishra, N. D. Dutt, "Functional Coverage Driven Test Generation for Validation of Pipelined Processors,"In Proc. 2005 *Design Automation and Testing in Europe*, pp. 678-683, 2005

[9] S. Fine, A. Ziv, "Coverage directed test generation for functional verification using Bayesian networks," In Proc. 2003 *Design Automation Conference*, pp. 289-261, 2003

[10] H.-W. Hsueh, K. Eder, "Test Directive Generation for Functional Coverage Closure Using Inductive Logic Programming," In Proc. 2006 *High-Level Design Validation and Test Workshop*,pp. 11-18,2006

[11] J. Hipp, U. Guntzer, and G. Nakaeizadeh. "Algorithms for Association Rule Mining - A General Survey and Comparison." Proc. ACM SIGKDD *International Conference on Knowledge Discovery and Data Mining*, 2000

[12] R. Agrawal, R. Srikant. "Fast algorithms for mining association rules," Proc. 20th VLDB Conference, pp. 487-499, 1994

[13] D. Burdick, M.Calimlim, and J. E. Gehrke. "MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases," 17th *Int. Conf. Data Engineering*, pp. 443-452, 2001

[14] H. Mannila, H. Toivonen , A.I. Verkamo. "Discovery of Frequent Episodes in Event Sequences," Data Mining and Knowledge Discovery, Volume 1, Issue 3, 1997