Assertion-Based Automated Functional Vectors Generation Using Constraint Logic Programming

Tun Li, Yang Guo, Si-Kun Li
National University of Defense Technology, P. R. China
tunlee@sina.com yguohn@yahoo.com.cn lisikun@263.net.cn

ABSTRACT
We present a novel approach to generate functional vectors based on assertions for RTL design verification. Our approach combines program-slicing based design extraction, word-level SAT and dynamic searching techniques. Through design extraction, vectors generation need only concern about the design parts related to the given assertion, thus large practical designs can be handled. Constraints Logic Programming (CLP) naturally models mixed bit-level and word-level constraints, and word-level SAT techniques solve the mixed constraints in a unified framework, which gain perfect performance. Initial states derived from dynamic simulation can dramatically accelerate the searching process of functional vectors generation. A prototype system has been built, and the experimental results on some public benchmarks and industrial circuits demonstrate the efficiency of our approach and its applicability to large practical designs.

Categories and Subject Descriptors
B.7.2 [Integrated Circuits]: Design Aids – simulation, verification

General Terms
Algorithms, Performance, Design, Verification.

Keywords
Assertion, Functional Verification, Constraint Logic Programming, Decision Diagrams, Test Generation

1. INTRODUCTION
With increasing complexity of modern design, functional verification has become the bottleneck of the design process. Both formal verification and simulation-based verification methods have advantages and disadvantages for design verification. Therefore, more and more researchers focus on semi-formal verification techniques in recent years, which integrate the advantages of the two verification methods.

Functional vectors generation techniques can enhance the coverage of hard-to-detect design errors. Generally, coverage metrics are code metrics, such as statement coverage, branch coverage, path coverage, toggle coverage, and some metrics related to the structure of the design, such as FSM state coverage, transition coverage and OCCOM coverage [1], etc. There are various functional vectors generation method for these coverage metrics [2 - 5].


However, the exiting coverage metrics did not take functional properties into consideration. In many practical designs, although 100% code coverage is gained, some corner case errors still escaped from being covered. Furthermore, the existing constraint solving methods are mostly based on bit-level techniques, which cannot be applied to large designs. Although the hybrid methods [9, 10] can deal with large practical designs, those techniques rely on heuristics to propagate the constraints between different domains. Hence the performances of the hybrid methods are limited by the heuristics that select the set of assignment to propagate constraints.

Assertions are recognized as a powerful tool for automatic runtime detection of design errors. There emerged some assertion libraries and assertion languages in recent years, such as OVL [11], Sugar/PSL [12], and 0-in Check [13], and some researches and commercial tools for static property checking based on assertions [14 -16]. However, in design practices, after adding assertions for runtime design errors monitor, it always wastes lots of simulation resource with large amount of randomly generated stimulus without triggering the inserted assertions.

In this paper we propose a new functional vectors generation approach based on assertions and constraint logic programming (CLP) techniques. It utilizes the strength of the assertion functional coverage metric and the word-level SAT solving techniques. The assertion properties can be transformed into a counter-example-generation problem which can therefore be solved by a word-level SAT solving engine.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GLSVLSI’04, April 26-28, 2004, Boston, Massachusetts, USA
Copyright 2004 ACM 1-58113-853-9/04/0004...$5.00.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GLSVLSI’04, April 26-28, 2004, Boston, Massachusetts, USA
Copyright 2004 ACM 1-58113-853-9/04/0004...$5.00.

This work is supported by the National Science Foundation of China (NSFC) under grant No. 60303011 and 90207019; the National High-Tech Research and Development Plan of China (863) under Grant No. 2002AA1Z1480
Given a Verilog design and an assertion, in order to generate test vectors to violate the assertion, we first extract the design codes which only related to the variables used in the given assertion [17], then build decision diagram (DD) models [21] for all the variables and signals used in the extracted design. Based on the DD models, we can translate the data propagation relations between primary inputs and the assertion variables to CLP constraints. At last, by solving CLP constraints using GNU Prolog [20], we can get functional vectors for violating the given assertion, which will detect design errors hidden in the Verilog codes and save precious verification resources.

There are several advantages of our method: (1) program slicing based design extraction can dramatically reduce the design complexity and make the followed process concentrated on the assertion related design, thus large practical design can be handled. (2) CLP constraints can effectively model and solve the bit-level and word-level constraints in a unified framework, which will get higher performance than hybrid methods. (3) using assertion property as functional coverage metric will accelerate error detection process.

The rest of the paper is organized as follows: in Section 2 we first outline our assertion based functional vectors generation framework. The creation of test generation goal will be described in Section 3, respectively. The detailed assertion based functional vectors generation method will be shown in Section 4. In Section 5 we show the experimental results of our method, and Section 6 concludes the paper.

2. TEST GENERATION FRAMEWORK

The input of our framework is RTL Verilog codes with embedded assertions.

Our framework compiles the inputs to extract the embedded assertions and uses them as test generation goals. For each assertion, the framework performs a goal creation operation by negating the assertion. Then, a circuit extractor [17] extracts the Verilog codes related to the assertion from the whole design description, and only the extracted codes are used for test generation. After translating the extracted Verilog codes to CLP constraints, the GNU Prolog [20] is used to solve the constraints to generate test vectors to violate the given assertion. GNU Prolog is an open source CLP solver, and has reported a good performance even comparable to some of the commercial tools. It is a native Prolog compiler with constraints solving over the finite domains, which makes it especially suitable for solving our problem. If no solution is found, we expand the time frame and continue the above process. This process continues until the test vectors for violating the assertion is generated or the time frame expanded or the memory and runtime exceeds the pre-set limit.

The assertions are similar to the ones defined in SystemVerilog [18], and only support a subset of SystemVerilog assertions. The assertions are used as Verilog comment with “/ *A:*” as the prefix, and include sequential assertion and concurrent assertion. Readers can refer to [18] for detailed assertion definition.

3. GOAL GENERATION

Inverting the conditional expression of the sequential assertions is very simple, which is the same as inverting a Verilog conditional expression. To invert the concurrent assertions, in general, for a sequence $f_1 \# \# f_2 \# \# f_3 \# \# \ldots \# \# f_{n+1}$, it can be converted to the corresponding LTL formula as $f_1 \rightarrow (X_{[n]} f_2 \land X_{[n+2]} f_3 \land \ldots \land X_{[\sum_{i=1}^{n}]} f_{n+1})$. The process to invert the formula can be expressed as formula (1). We can conclude from formula (1) that if the $f_i$ is true, then in sequel, any violation of the $f_i$ will violate the whole sequence expression.

$$
\neg (f_i \rightarrow (X_{[n]} f_i \land X_{[n+2]} f_i \land \ldots \land X_{[\sum_{i=1}^{n}]} f_{n+1}))
$$

$$
= (f_i \land X_{[n]} \neg f_i) \lor (f_i \land X_{[n+2]} \neg f_i) \lor (f_i \land X_{[\sum_{i=1}^{n}]} \neg f_{n+1})
$$

To violate the concurrent assertions in implication form, we first generate test to produce sequence matching the first sequence of the implication operation, and then generate test to violate the second sequence of the implication.

4. AUTOMATED TEST GENERATION

4.1 DD Models

The DD model is first introduced by R. Ubar [21], which can model the control and data dependence on variables and signals assignment in design descriptions. We extended the original DD model to deal with more general coding style in RTL descriptions, and proposed a method to build DD model from HDL descriptions. Figure 1 shows the DD model example, where the ellipse nodes correspond to the assignment statements in the HDL description, the rectangle nodes correspond to the conditional statements, and the root node represents the $cnt$ signal.

4.2 CLP Modeling

The GNU Prolog solver supports many built-in predicates in finite domains. These built-in predicates made our modeling of RTL designs quite straightforward.

Herein, we only discuss modeling for the conditional statements, such as $if$-$else$ and $case$ statements. If the conditional statements have complete selection entries, for example, a $case$ statement with all the selection entries is said complete, then modeling the conditional statement is straightforward with the GNU Prolog implication predicate. When the conditional statements are incomplete with default entries, then we must complete the conditional statements’ selection entries during CLP constraints generation.

For incomplete $if$-$else$ statements, the solution is simple. For example, for the node “$cnt<8$” in Figure 1, the constraints generated is shown as following. The suffix of $cnt$ is the temporal and spatial incarnation, and readers can refer to [3] for detailed discussion.

$$
(Cnt_0 \_0 \ 0 < 8) \implies Cnt_0 \_1 \implies Cnt_0 \_0 + 2,
$$

$$
(Cnt_0 \_0 \ 0 >= 8) \implies Cnt_0 \_1 \implies Cnt_0 \_0,
$$

For incomplete $case$ statements, the expressiveness of GNU Prolog provides a solution. That is, the selected inputs are
combined as a new integer variable; and another new Boolean variable is introduced to encode the fact that the default entry is taken. A case statement and the generated CLP constraints are shown in Figure 2.

\[
\begin{align*}
\text{case (\{put, get\})} & := \text{fiffo.v:39} \\
2'\text{#0:} & \\
\text{if (cnt=9)} & \begin{align*}
\text{cnt} & := \text{cnt}+2; \quad \text{fiffo.v:39} \\
\text{end}
\end{align*}
\end{align*}
\]

Figure 2 Incomplete case statement and its constraints

The CLP constraints generation is based on the variables’ DD model. The CLP generation process is started from the root node of the DD model. In our implementation, each node has a method GenCLP, which recursively call the child nodes’ GenCLP method to generate CLP constraints for its child nodes, and then combined with the CLP constraints generated for itself to construct the final constraints for the variable.

To propagate the value of the intermediate signals to the primary inputs, a signal set \( V \) is defined in the constraints generation process. When traversal the nodes of the signal’s DD model, each time the GenCLP method encounters an intermediate signal used in the nodes, it adds the intermediate signal to the set \( V \). After generating CLP constraints for all the signals used in the assertion, the second step of the CLP generation process will generate constraints for the signals in the set \( V \) based on their DD models. In this step, the intermediate signals encountered by the GenCLP will be added to the set \( V \). The process will continue until set \( V \) becomes empty. This will chain the signals used in the assertion with the primary inputs.

4.3 Constraints Solving and Test Extraction

The last phase of the functional vectors generation process consists in solving the constraint equations obtained for each time frame. There are two possible results:

- The set of constraints cannot be satisfied. In this case, if the preset limits are not exceeded, the constraint generation and solving process continue with one more time frame expanded.
- The set of constraints can be satisfied. In this case, the GNU Prolog solver can compute one possible solution or all the solutions for that set of constraints. Finally, by mapping the values of the solutions over the inputs ports along the time frames to obtain a complete testbench.

5. EXPERIMENTAL RESULTS

We have implemented the assertion-based test vectors generation prototype framework. The framework is applied to several public benchmark and practical circuits. Table 1 shows the characteristic of these designs.

In Table 1, the 1st column is the name of the design, where TLC is the traffic light control circuit and the decoder circuit is a practical design which implements the decoder of the pipeline of a 32-bit microprocessor core. The 2nd and 3rd column is the line number of the Verilog codes and the input number of the designs, respectively. The 4th column is the number of the versions we derived from the correct design descriptions, which are got by modifying the correct description to introduce errors corresponding to the given properties.

Table 2 shows the experimental results for functional vectors generation based on assertions conducted on the designs on a Windows 2000 PC with AMD 1.8GHz CPU and 256MB memory. We experiment both the find one and find all option of the CLP solver. The cycle columns are the maximum time frames expanded to detect all the errors introduced in the designs. The two time columns are the constraint solving time for the find one and find all options, respectively, where the time is in seconds. We can conclude from Table 2 that word-level SAT solver can obtain good performance. For the second property of the alarm clock circuit, we apply the acceleration techniques by pre-set the clock to “23:59”, then we only expand the time frames for 60 cycles for vectors generation. For the find one and find all options, we get solutions with the default 8MB memory allocation of the GNU Prolog for all the experimental circuits, which showed that our method is efficient in memory usage.

We also have implemented the method introduced in [3]. Table 3 shows the experimental results when comparing with the path orient test generation method proposed in [3]. We can conclude
from Table 3 that with the same time frames expanded, our method is efficient in error detecting and constraints solving by using assertion as the functional coverage metric, while the code coverage metrics cannot find some error deeply hidden in the designs. In Table 3, the Time is in second.

### Table 3 Comparing with Path Coverage Metric

<table>
<thead>
<tr>
<th>Name</th>
<th>Cycle</th>
<th>Assertion Based</th>
<th>Path Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time Detected</td>
<td>Time Detected</td>
</tr>
<tr>
<td>addr decoder</td>
<td>10</td>
<td>0.01</td>
<td>2</td>
</tr>
<tr>
<td>TLC</td>
<td>30</td>
<td>0.01</td>
<td>3</td>
</tr>
<tr>
<td>arbiter</td>
<td>11</td>
<td>0.01</td>
<td>2</td>
</tr>
<tr>
<td>alarm clock</td>
<td>60</td>
<td>0.03</td>
<td>3</td>
</tr>
<tr>
<td>decoder</td>
<td>4</td>
<td>0.01</td>
<td>1</td>
</tr>
</tbody>
</table>

We also compared our method with the random test vectors generation method for triggering assertion. The experimental results are shown in Table 4. We use the test generated in Table 2 and randomly generated test with length of 100 to trigger the assertions inserted to each design. We can conclude from Table 4 that our method is efficient in assertion triggering.

### Table 4 Comparing with Random Generation Method

<table>
<thead>
<tr>
<th>Name</th>
<th>Assertion Based</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cycle</td>
<td>Triggered</td>
</tr>
<tr>
<td>addr decoder</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>TLC</td>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>arbiter</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>alarm clock</td>
<td>60</td>
<td>3</td>
</tr>
<tr>
<td>decoder</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

6. CONCLUSION AND FUTURE WORK

We proposed a new functional vectors generation method combined assertion-based coverage metric, word-level constraint modeling and SAT solving techniques. The experimental results show that our method is very time and memory efficient, and efficient in detecting errors for large practical designs. The future research directions include providing support for more assertion syntax; determining the optimistic time frames to be expanded; and applying our method for model checking more general properties.

7. REFERENCES


