Chao HN, Li HW, Song X *et al.* Evaluating and constraining hardware assertions with absent scenarios. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(5): 1198–1216 Sept. 2020. DOI 10.1007/s11390-020-9708-x

Evaluating and Constraining Hardware Assertions with Absent Scenarios

Hui-Na Chao[1,2], Hua-Wei Li[1,2,3,*], *Distinguished Member, CCF, Senior Member, IEEE*
Xiaoyu Song[4], *Senior Member, IEEE*, Tian-Cheng Wang[1,2], *Member, CCF*, and
Xiao-Wei Li[1,2], *Fellow, CCF, Senior Member, IEEE*

[1] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China*

[2] *University of Chinese Academy of Sciences, Beijing 100049, China*

[3] *Peng Cheng Laboratory, Shenzhen 518052, China*

[4] *Department of Electrical and Computing Engineering, Portland State University, Portland, OR 97207, U.S.A.*

E-mail: {chaohuina, lihuawei}@ict.ac.cn; songx@pdx.edu; {wangtiancheng, lxw}@ict.ac.cn

Received May 10, 2019; revised September 12, 2019.

Abstract Mining from simulation data of the golden model in hardware design verification is an effective solution to assertion generation. While the simulation data is inherently incomplete, it is necessary to evaluate the truth values of the mined assertions. This paper presents an approach to evaluating and constraining hardware assertions with absent scenarios. A $Belief\text{-}failRate$ metric is proposed to predict the truth/falseness of generated assertions. By considering both the occurrences of free variable assignments and the conflicts of absent scenarios, we use the metric to sort true assertions in higher ranking and false assertions in lower ranking. Our *Belief-failRate* guided assertion constraining method leverages the quality of generated assertions. The experimental results show that the Belief-failRate framework performs better than the existing methods. In addition, the assertion evaluating and constraining procedure can find more assertions that cover new design functionality in comparison with the previous methods.

Keywords hardware formal verification, assertion generation, data mining, assertion evaluation, assertion coverage

1 Introduction

In the domain of verifying integrated circuit, functional correctness is the essential requirement. Traditionally, simulation and formal verification are the main ways of functional verification. Simulation is a relatively mature and widely-used method in industry applications, but cannot perform complete verification. Formal verification, which can perform complete verification, requires a complete assertion set to cover the design's functionality. If the assertion set is incomplete, more assertions should be generated. Consider the design that counts the occurrence of logic 1 within four continuous inputs of a primary input. The output is 1 if there are at least two logic 1s; otherwise, the output is zero, i.e., $O = (a \vee Xa) \wedge (X^2a \vee X^3a) \vee (a \wedge Xa) \vee (X^2a \wedge X^3a)$, where a is the one-bit input and O is the output. An assertion set

$$\left\{\begin{array}{l} A_1 : (a = 0) \wedge X\,(a = 0) \wedge X^2\,(a = 0) \rightarrow \\ \qquad X^3\,(O = 0)\,, \\ A_2 : (a = 1) \wedge X^2\,(a = 1) \rightarrow X^3\,(O = 1)\,, \end{array}\right\}$$

asserts that 1) starting from any instant, the output O is definitely logic 0 in the fourth cycle if there are three continuous 0s of input variable a, and 2) logic 1 in the first cycle and the third cycle will lead to an output of logic 1 in the fourth cycle. It is obvious that the assertion set misses many possible input sequences. That is, it is an incomplete set and more assertions are essential.

Regular Paper

The work was supported in part by the National Natural Science Foundation of China under Grant Nos. 61876173, 61432017, and 61532017.

*Corresponding Author

Assertion generation can be achieved by manual written or automatic generation. Writing a complete assertion set manually requires a large amount of time and extensive expert knowledge. Recently, researchers have explored to generate assertions automatically[1–12]. Automatic generation of assertions can be divided into two categories: static analysis and dynamic generation from simulation traces. The first category performs static analysis on design codes, and then the results of the static analysis are transformed into assertions. Static analysis needs much design information, and the format of assertions which can be generated is limited. The second category adopts the mining algorithms to generate assertions from simulation traces of the golden model of the hardware design under verification (DUV). The assertion generation process can be totally automatic and can provide a great number of assertions with a wide diversity.

However, as it is hard to get complete simulation data, the generated assertions may be false for the design. For example, a simulation trace of the above example can be expanded by four cycles (equal to the depth of the design, which is the largest number of cycles needed to traverse each reachable state) and the resulted simulation trace is shown in Table 1 (a, $[1]a$, $[2]a$ and $[3]a$ represent the variable at cycle delay 0, 1, 2 and 3 respectively).

Table 1. Example Simulation Trace

	$t1$	$t2$	$t3$	$t4$	$t5$	$t6$	$t7$	$t8$	$t9$	$t10$	$t11$
a	1	1	1	1	0	1	1	0	0	0	0
$[1]a$	1	1	1	0	1	1	0	0	0	0	1
$[2]a$	1	1	0	1	1	0	0	0	0	1	0
$[3]a$	1	0	1	1	0	0	0	0	1	0	0
O	1	1	1	1	1	1	0	0	0	0	0

From the incomplete simulation trace, an assertion set can be generated for the design, as shown in Table 2. Although all assertions are assured to have no conflict with the simulation trace, assertions A_2, A_3, A_4, A_5 andA_7 are true assertions and A_1 andA_6 are false assertions. Thus, the generated assertions should be further verified to assure truthfulness. Accordingly, the true/false judgment or quality evaluation of automatically generated assertions is essential for dynamic assertion generation.

Table 2. Generated Assertions from the Example Trace

ID	Assertion
A_1	$(a = 0) \wedge ([3]a = 1) \rightarrow (O = 0)$
A_2	$(a = 0) \wedge ([3]a = 0) \wedge ([1]a = 0) \rightarrow (O = 0)$
A_3	$(a = 0) \wedge ([3]a = 0) \wedge ([1]a = 1) \wedge ([2]a = 0) \rightarrow (O = 0)$
A_4	$(a = 0) \wedge ([3]a = 0) \wedge ([1]a = 1) \wedge ([2]a = 1) \rightarrow (O = 1)$
A_5	$(a = 1) \wedge ([2]a = 1) \rightarrow (O = 1)$
A_6	$(a = 1) \wedge ([2]a = 0) \wedge ([1]a = 0) \rightarrow (O = 0)$
A_7	$(a = 1) \wedge ([2]a = 0) \wedge ([1]a = 1) \rightarrow (O = 1)$

Formal verification has been the main approach to verifying the truth/falseness of assertions for a long time. There are many mature formal verification tools, including Verification Interacting with Synthesis (VIS)①, ABC②, Cadence Incisive Formal Verifier (IFV)③, etc. These tools traverse the finite state machine (FSM) of the design to check the true/false of the assertion on all reachable states of FSM. If the assertion is true in all reachable states, it is a true assertion for the design. Otherwise, it is false. While formal verification can give exact results, the traverse of FSM may lead to state exploration problem and need a large amount of time, which limits the design size it can be applied to. As for the verification of the automatically generated assertion set, which contains a great number of assertions, the time consumption and the design size limitation of formal verification are even obvious. Recently, some coarse methods which aim at the evaluation of automatically generated assertions have been presented[5, 16, 17]. Most of these metrics count the frequency of assertions in simulation data, and evaluate assertions by some mathematical formulas of the counted frequency. Compared with the formal verification, this kind of methods can give faster evaluation results, but with a lower accuracy.

From the further observation of the example above, it can be learned that the false assertion A_1 covers the simulation instant $t9$ and is generated because of the absent scenarios {01, 10, 11} of free variables $[1]a$ and $[2]a$. For instance, if there is an instant $t12$, with variable combination $\{a = 0,\ [1]a = 1,\ [2]a = 1,\ [3]a = 1\}$, assertion A_1 will not be generated. However, generating such simulation instants for all assertions is equal to the generation of a complete simulation trace, which is impossible. A replacement strategy is to find many approximate instants from the existing simulation data

①https://ptolemy.berkeley.edu/projects/embedded/research/vis/index.html, Sept. 2020.

②https://people.eecs.berkeley.edu/∼alanmi/abc/, Sept. 2020.

③https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/incisive-formal-verification-platform.html, Sept. 2020.

to approximate such missing simulation instants. That is, we collect absent scenarios $\{01, 10, 11\}$ of free variables $[1]a$ and $[2]a$ in the whole trace to approximate the missing combinations $\{a = 0, [1]a = 0, [2]a = 1, [3]a = 1\}$, $\{a = 0, [1]a = 1, [2]a = 0, [3]a = 1\}$, and $\{a = 0, [1]a = 1, [2]a = 1, [3]a = 1\}$. Instants $t1 - t6$, $t10$ and $t11$ are collected and six eighths of them have a conflict output with assertion A_1. Thus, we refer that the assertion A_1 has a high possibility to be falsified by absent scenarios and it needs to be constrained to develop a new true assertion.

Driven by the intuition above, we extend our previous work[15] in this paper and propose an improved hardware assertion evaluation and constraining framework, which is guided by finding the absent scenarios in the whole simulation trace. There are two main improvements. First, we optimize the workflow by adding a threshold TB of $Belief$ value, which enables the new flow to combine the advantage of the $Belief$ metric and the $failRate$ metric to make better evaluation. Second, we expand the application of the proposed $Belief$-$failRate$ metric from coarse-grain assertion evaluation (true or false) to assertions ranking.

The improved Belief-failRate framework uses the frequency of conflict absent scenarios, i.e., absent free variable assignment combinations of free variables, to perform the true/false evaluation of mined assertions. Firstly, we compute the *Belief* value[14] for each assertion, which quantifies the occurrence of free variable assignments under the assertion. Secondly, for the absent scenarios of an assertion, we count their occurrences in other simulation instants which are conflicted with the assertion. The ratio of conflict occurrence count to their total occurrence count in other simulation instants is computed as *failRate*[15]. A high *failRate* means that absent scenarios may falsify the assertion while a low *failRate* indicates that the absence of some assignments has no influence on the assertion. Under this Belief-failRate framework, all assertions can be evaluated. For the cases where most assertions are true, the ranking results of assertions by their $Belief$-$failRate$ values are similar to those by existing ranking methods, which indicates that the $Belief$-$failRate$ metric can work as a ranking method as well. Moreover, a *Belief-failRate* guided method for constraining assertions is given to improve the quality of assertion generation. Experimental results show that the proposed Belief-failRate framework can predict the truth/falseness of assertions more reasonably, and the *Belief-failRate* guided assertion constraining procedure generates assertions that have a high possibility to be true.

Return to the example in Table 1. Although no inconsistency exists between the assertions and the simulation trace, potential inconsistency can be found for assertions A_1–A_7 by computing its $Belief$-$failRate$ value. As a result, assertions A_1 and A_7 are given lower rankings and evaluated as false. To cover the missed subspace indicated by the false assertions A_1 and A_7, constraining is performed by adding the proposition of free variables $[1]a$, $[2]a$ and $[3]a$, respectively. Finally, a new complete assertion set can be generated.

The rest of the paper is organized as follows. Section 2 presents related work of assertions generation and evaluation. Section 3 gives the workflow of the proposed Belief-failRate framework and the guided assertion constraining procedure. Section 4 shows the experimental results and conclusions are given in Section 5.

2 Related Work

The generation and the quality evaluation of assertions are an important part of the functional verification of integrated circuits. In this section, some assertion generation and evaluation methods are introduced.

2.1 Automatic Assertion Generation

Vasudevan *et al.*[1, 2] proposed GoldMine, an assertion generation methodology using a binary decision tree based supervised learning algorithm. GoldMine contains four steps: simulation data generation, static analysis, assertion generation, and assertion verification and evaluation. Simulation data can be achieved by the directed or random simulation of the golden design. Static analysis is responsible for finding the cone influence of each target variable, to reduce the search space of assertion generation. Assertion generation can be achieved by searching engines using different category algorithms in machine learning. The generation methodology using the binary decision tree algorithm was implemented as the tree engine of GoldMine. Generated assertions will be verified on the golden design by the formal tool IFV and the true part should be further evaluated to estimate the effectiveness of the assertion generation algorithm.

Following this work, the group proposed a coverage guided assertion generation method[3], implemented as the coverage engine of GoldMine. This method generates assertions that can cover 50% input space firstly and they are verified to retain true assertions only. Then, assertions that can cover 25% input space, 12.5%

input space, ..., will be generated in each loop, until the covered input space of assertions generated in current loop is lower than the input space threshold $g_{\text{threshold}}$. The method prefers to generating short assertions first and increasing the length of assertions step by step, until the specified threshold is achieved. As the formal verification tool IFV is integrated to check assertions generated in each step, verification overhead is unavoidably time-consuming. To generate assertions in word level rather than in bit level, the group proposed another improved method [4] which considers only primary inputs and primary outputs. From the design code at the register transfer level (RTL), the dynamic weakest preconditions of the target can be computed. Then, concreate simulation paths can be generated and assertions can be achieved by the reverse replacement of variables in these paths. As the generated assertions may be over-constrained or invalid, further checks for conflict propositions need to be performed to remove redundant propositions.

Chang and Wang [5] proposed a time window and support-confidence based assertion mining algorithm. The support of an assertion is defined as the frequency of the assertion in the simulation trace, and the confidence of an assertion is the percentage of the support of the assertion to the support of its antecedent. By setting the thresholds of support and confidence, the simulation trace can be split into slices of time windows. Frequent patterns in each slice can be found and used to generate implication relations, and relations that satisfy the thresholds of support and confidence are the final assertion set.

Bertasi *et al.* [6] proposed an assertion extraction method that extends the format of assertions to arithmetic expression or logical expression. An assertion is the implication formula of its antecedent and consequent. The antecedent is one of the frequent propositions learned by the tool Daikon [16], while the consequent is generated from the corresponding *ST_trace* (a slice of simulation trace which can contain discontinuous simulation instant). This extraction is independent of the design model, and it works on designs at gate level, RTL, system level or even for embedded software.

Danese *et al.* [7] proposed to generate temporal assertions of specified formats, i.e., *Next, N-next, Until, Alternating, Next_or, N-next_or* and *Until_or*. First, cone of influence (COI) information from the static analysis is used to split simulation trace into time windows. Then, atomic propositions in all time windows are collected and combined to generate possible propositions. Finally, propositions can be further combined to construct temporal assertions. Hereafter, the authors extended the concept of the time window to input time window and output time window [8], and assertions can be generated as the implication relation between propositions in the input time window and the output time window. The method has been applied to the generation of power state machine (PSM) [9]. By mapping assertions switching to transitions of the PSM and assertions to states of the PSM, the PSM can be established along with the generation of assertions.

For gate-level arithmetic circuits, Ciesielski *et al.* [10] proposed a method to extract the polynomial function of the design. The method divides all internal variables into different layers according to their distances to input (output), expressed as $l_1, l_2, \ldots, l_n$ respectively. As a result, each variable in layer l_i can be expressed as a polynomial function of variables in l_{i-1}. Firstly, primary outputs (POs) are expressed by the function of variables in l_n. Then, all variables in l_n can be replaced by their polynomial expressions of variables in l_{n-1}. The replacement continues until the primary outputs are expressed by polynomial functions of variables in l_1. Finally, all variables in l_1 can be replaced by their polynomial functions of primary inputs (PIs). At last, POs are expressed as polynomial functions of PIs, i.e., the polynomial function of the circuit is achieved.

Hanafy *et al.* [11, 12] proposed a Breadth-First Decision Tree (BF-DT) based algorithm for assertion generation. Instead of splitting each node by one variable, as used in the tree engine of GoldMine, this method tries all possible variables to split the search space until an assertion is generated along a current branch or the corresponding space has been searched. This method prefers to short assertions and all possible assertions will be generated. As an exhausting searching method, the runtime is long even for small designs.

For all the above assertion generation methods, the true/false of the assertions are influenced by the completeness of the simulation data. When the simulation is far from complete, many false assertions may be generated. Therefore, the generated assertions should be further verified on the golden design and evaluated to filter out the false part.

2.2 Assertion Evaluation

There has been much work on the evaluation of true assertions. Hoskote *et al.* [17] defined an evaluation metric based on covered states. Given FSM M, a true assertion P, and a set of reachable states of FSM, S, the covered states of P are the subset of S that is essential for the true/false of P. That is, the values of observed variables on all states in the subset must be checked to decide the true/false of P. In other words, changing the value of each observe variable on any states in the subset will falsify P while changing values of the observed variables on any states outside the subset has no influence on P. Following this definition, Jayakumar *et al.*[18] and Chao *et al.*[19] presented two algorithms to compute the covered set.

Haedicke *et al.*[20] evaluated the coverage of passed assertions by dividing assertions into the safe part and the unsafe part according to whether the assertions have a dependence on internal signals. DUV is described by a diagram and assertions using Computation Tree Logic (CTL). Given the assertion set and the observed variable, uncovered scenarios are expressed by traces of the diagram, and then transformed to traces of signals. The coverage of the assertion set will be the weighted sum of the coverage of safe assertions and unsafe assertions.

For most generation methods, assertions are in the form of $A \rightarrow C$, where A is the antecedent and C is the consequent. The meaning of such assertions is that in any case, once A holds, C is true. The evaluation of the generated assertions involves true/false judgment and quality estimation.

Vasudevan *et al.*[1] proposed to use the value of support-confidence to rank assertions, the same way as Chang and Wang[5] did. The difference between them is that the former uses the support-confidence metric to rank assertions after assertion generation, while the latter uses it to guide the generation process.

Ghasempouri and Pravadeli[13] proposed to evaluate the quality of generated assertions by establishing a table based on the frequency of A, C, $!A$ and $!C$. The correlation coefficient of A and C can be computed according to the table and used to rank assertions. Assertions with higher correlation coefficients are in higher rankings.

Mitra *et al.*[14] proposed a *Belief* value to evaluation assertions. The *Belief* value of $A \rightarrow C$ is defined as the percentage of free variable combinations covered by the simulation trace that supports assertion $A \rightarrow C$. Similarly, assertions with a higher *Belief* value are considered as having a higher possibility to be true.

For all the evaluation methods of generated assertions above, most of them can give a coarse result quickly. However, it is still a challenge to improve the accuracy of assertion evaluation. In this paper, we propose a framework named Belief-failRate to assess the true/false of assertions. Addition to the *Belief* evaluation, this metric takes the influence of absent scenarios into consideration as well. First, the *Belief* value as defined in [14] is computed. For assertions with a high *Belief* value, it is considered as true. Otherwise, a *failRate* value is computed to make further decision. The *failRate* value of assertion $A \rightarrow C$ is defined as the percentage of free variable combinations that occur with $!C$. High *failRate* assertions are considered as false while others as true. Moreover, a *Belief-failRate* guided method for constraining assertions is given to improve the quality of assertion generation.

3 Proposed Method

The workflow of the proposed *Belief-failRate* based assertion evaluation and constraining framework is shown in Fig.1. First, assertions are generated from simulation traces of correct design using GoldMine[1]. Second, the *Belief-failRate* based evaluation is applied to predict the truth/falseness of generated assertions. Third, false assertions are further constrained to generate more true assertions.

3.1 Assertion Generation

The basic steps of assertion generation contain the generation of simulation traces, feature collection using category algorithms, and assertion generation. Traces can be achieved by random or directed simulation of the golden design. Category algorithms are used to collect different features of different values of the output. After the application of the category algorithm, assertions can be learned from the information provided by the category algorithm.

The proposed method is devoted to the evaluation and constraining of temporal assertions mined from simulation traces, while temporal assertions are mined by expanding the simulation traces by the certain number of cycles. In this paper, we use GoldMine[1] as a miner to generate temporal assertions. GoldMine has four modes to generate assertions under different searching engines: tree[1, 2], forest, prism and coverage[3]. The antecedent A is represented as $A = a_0 \wedge Xa_1 \wedge XXa_2 \wedge \cdots \wedge X^m a_m$ and the consequent C as $C = X^n c_n$, where a_i and c_n $(i = 0, 1, ..., m \leqslant n)$ are propositions, and X^j represents a delay by j cycles. Usually, n determines how many cycles the simulation data should be expanded and is set by the user. The value of m is determined in the process of assertion

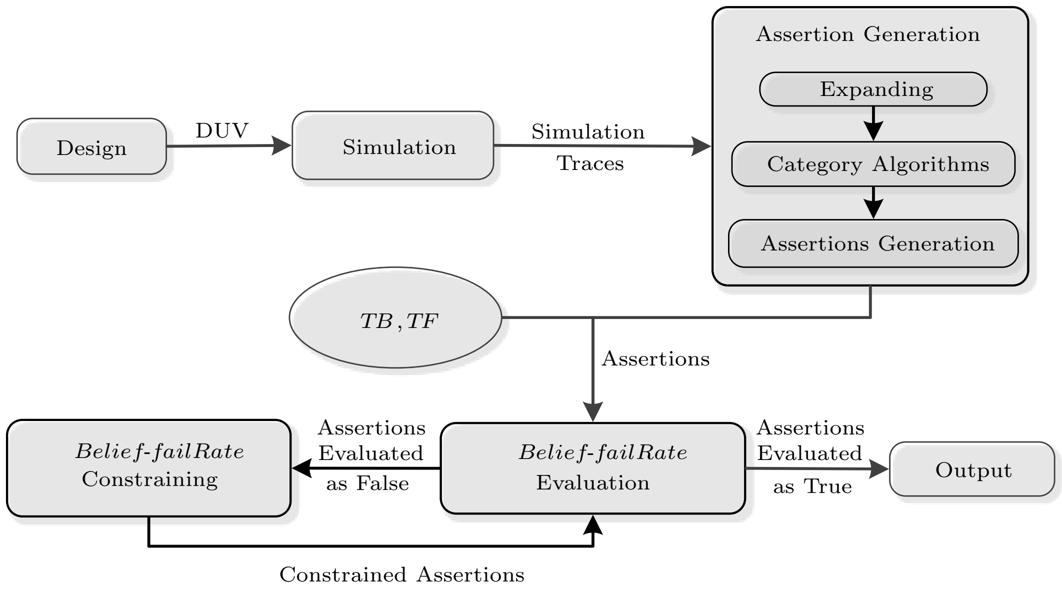


Fig.1. Workflow of the proposed framework.

generation by the cycles the propositions are used and may be different from assertion to assertion. In our implementation, we set n as the depth of the design to allow the generation algorithms to generate more assertions with inputs and the concerned variables. In other cases when the depth of the design is too large, a medium value of n is set and more assertions involving middle variables may be generated, depending on the generating strategy of GoldMine.

Although the assertions are in a simple form $A \to C$, they cover the minimum connection phrase $\{\neg, \wedge\}$, which has the same expression ability as the most complex semantics. With such assertions, verifiers are rescued from writing assertions from scratch and are allowed to pay more attention on the vulnerable part of the design.

Take the engine tree[1, 2] of GoldMine as an example. The simulation trace is represented by a decision tree, in which each node is a subset of the simulation trace. In each loop, a splitting variable is used to split one node into two child nodes until all outputs are the same in the node. To select a suitable splitting variable, the *mean* value and the *error* value are computed as the mean value of the output of the node and the absolute deviation of each output from the mean of the node. The *gain* of a splitting variable at a particular node is the reduction in *error* between that node and the child nodes produced when this feature variable is used to split the data space. In each split, the variable that has the highest gain value is selected as the splitting variable.

We use the example in Table 1 to explain the process of assertion generation with the engine tree of GoldMine. Using the tree engine of GoldMine, the process of assertion generation is shown in Fig.2. Initially, the total simulation data $t1$–$t11$ are represented as a root. As the output has six logic 1s and five logic 0s, the *mean* value is $(6 \times 1 + 5 \times 0)/11 = 6/11$. The *error* value is $(|1 - 6/11| \times 6 + |0 - 6/11| \times 5)/11 = 0.496$. To choose a best splitting variable, all free variables are tried to make a split. When a is tried, the simulation data is split into subspace $\{a = 0\text{: } t5,\ t8\text{–}t11\}$, and subspace $\{a = 1\text{: } t1\text{–}t4,\ t6,\ t7\}$. The output of the former subspace has one logic 1 and four logic 0s; thus the *mean* value is $(1 \times 1 + 4 \times 0)/5 = 1/5$ and the *error* value is $(|1 - 1/5| \times 1 + |0 - 1/5| \times 4)/5 = 0.32$. The output of the latter subspace has five logic 1s and one logic 0; thus the *mean* value is $(5 \times 1 + 1 \times 0)/6 = 5/6$ and the *error* value is $(|1 - 5/6| \times 5 + |0 - 5/6| \times 1)/6 = 0.278$. The *gain* of variable a at the root node is the error reduction $0.496 - (0.32 + 0.278) = -0.102$. If split by variable $[1]a$, the *mean* and the *error* of subspace $\{[1]a = 0\text{: } t4,\ t7\text{–}t10\}$ are $1/5$ and 0.32 respectively, the *mean* and the *error* of subspace $\{[1]a = 1\text{: } t1\text{–}t3,\ t5\text{–}t6,\ t11\}$ are $5/6$ and 0.278 respectively, thus the *gain* is $0.496 - (0.32 + 0.278) = -0.102$. Similarly, the *gains* of variable $[2]a$ and $[3]a$ at the root node are -0.264 and -0.369, respectively. As variables a and $[1]a$ have the highest *gain* value, either of them can be chosen as the splitting variable of the root node. Here, we choose variable a. After the splitting of the root node, the subspace which has an *error* value higher than zero is further split using other free variables, until all leaf nodes have an *error* value equal to zero. Finally, the splitting variables along each branch from the root node to a leaf node can be combined to construct the antecedent of an assertion, and the *mean* value of the corresponding leaf node is the consequent of the assertion. The generated assertion set is as shown in Table 2.

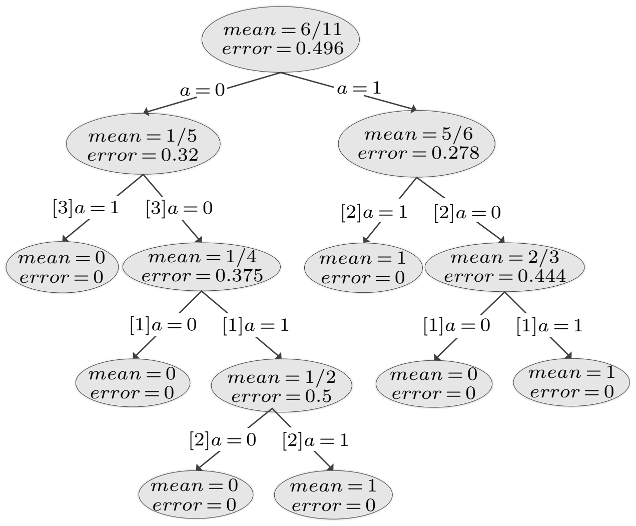


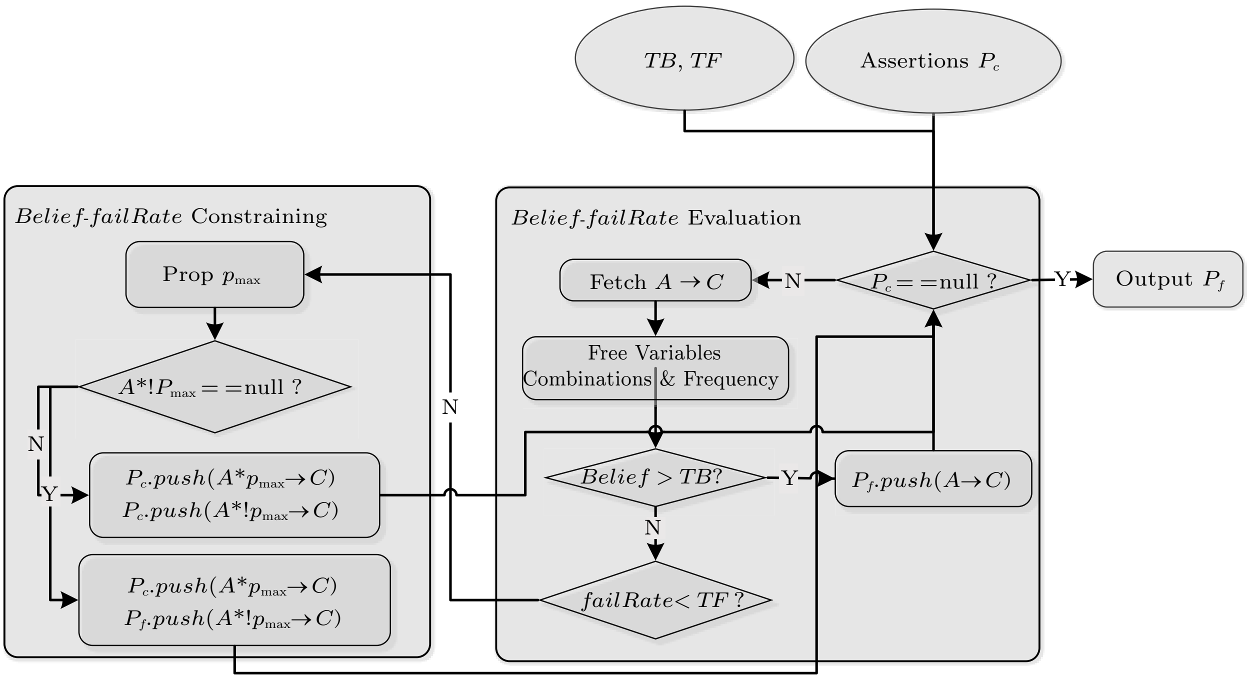
Fig.2. Process of assertion generation.

As the simulation trace is incomplete, the generated assertions are not assured to be true. Thus, evaluation is needed to get the true set of assertions.

3.2 Belief-failRate Framework

To reduce the time of formal verification for mined assertions on the golden design, we introduce the Belief-failRate method to evaluate the true/false of assertions and constrain assertions that are evaluated as false. Fig.3 shows the *Belief-failRate* evaluation and constraining procedure. In the procedure, P_c is initialized to the set of mined assertions to be measured. In the evaluation phase, thresholds *TB* and *TF* are set for the *Belief* value and the *failRate* value respectively. For each assertion in P_c, if the *Belief* value is higher than *TB*, we declare the assertion as true and add it to the final set P_f. Otherwise, we compute $failRate$ to make a further TF estimation. A maximum $failRate$ threshold *TF* is set to help decide whether to constrain an assertion or not. We consider only assertions with $failRate$ equal to or higher than *TF* as false and turn to the constraining procedure.

Note the *Belief* value gives visible evidence of an assertion to be true, while the *failRate* value indicates possible evidence of an assertion to be false. The setting of thresholds *TB* and *TF* enables the flexibility on the weight of the two values in evaluating assertions. A high *TB* indicates a strict visible evidence requirement, under which assertions covering more free variable combinations are considered as true. A high *TF* indicates a high tolerance of possible evidences of conflicts, i.e., an



TB, TF
Assertions P_c
Belief-failRate Constraining
Belief-failRate Evaluation
Prop $p_{\max}$
$A*!P_{\max}==$null ?
N
Y
$P_c.push(A*p_{\max}\to C)$
$P_c.push(A*!p_{\max}\to C)$
$P_c.push(A*p_{\max}\to C)$
$P_f.push(A*!p_{\max}\to C)$
N
Fetch $A\to C$
N
$P_c==$null ?
Y
Output P_f
Free Variables Combinations & Frequency
Belief > TB?
Y
$P_f.push(A\to C)$
N
failRate< TF ?

Fig.3. *Belief-failRate* based evaluation and constraining procedure.

assertion may be considered as true even though there are many conflict absent scenarios.

The evaluation phase contains three steps. First, the free variables of an assertion, on which *Belief* and *failRate* are both based, are figured out. Then, the frequencies of all possible free variable combinations in the simulation trace, from which the assertions are mined, are counted. Finally, the *Belief* and *failRate* values of the assertion are computed according to their definitions. Followings are the details of the computation of the *Belief* and *failRate* values and their application to predict the truth/falseness of assertions.

3.2.1 Computation of Belief

To quantify the confidence of mined assertions, Mitra *et al.*[14] proposed the metric of *Belief*. The assertions under evaluation are in the form of $A \to C$. Assume that $F_{A\to C} = \{v_1, v_2, \ldots, v_n\}$ is the set of free variables of $A \to C$ (that is, variables that are not occurred in $A \to C$), the set of possible assignments for v_i is expressed as $values\,(v_i)$. We define $Assign\,(F_{A\to C})$ as the set of all possible assignment combinations of free variables in $A \to C$.

$Assign\,(F_{A\to C}) \triangleq values\,(v_1) \times values\,(v_2) \times \cdots \times values\,(v_n)$.

Let T represent the simulation trace, $T_{A\to C}$ represent the set of different episodes (while an episode represents the trace data in one cycle) that $A \to C$ holds, and $T_{A\to C}$ represents the set of sub-episodes (sub-episode is a subset of the trace data in one cycle) obtained by retaining only assignments on free variables of each episode in $T_{A\to C}$. We define $posAssign\,(F_{A\to C})$ as the set of assignment combinations of free variables covered in $T_{A\to C}$.

$$posAssign\,(F_{A\to C}) = \{s_i | s_i \in T_v \cap Assign\,(F_{A\to C})\}\,.$$

Belief is defined as the percentage of covered free variable assignments.

$$Belief\,(A \to C) = \frac{|posAssign\,(F_{A\to C})|}{|Assign\,(F_{A\to C})|}.$$

Belief reflects the possibility of an assertion to be true and it is closely related to the completeness of the simulation trace. When *Belief* of an assertion equals 1, i.e., all free variable assignment combinations are occurred, the assertion is definitely true. However, if *Belief* is lower than 1, absent scenarios may falsify the assertion even when the *Belief* value is relatively high.

3.2.2 Computation of Belief-FailRate

In practical applications, the complete simulation trace is almost impossible to achieve. Or even, in most cases, the simulation trace is far from complete. Under such a case, the effectiveness of evaluating assertions using *Belief* is low. We tackle this problem using a metric called *Belief-failRate*. By taking the occurrence of absent scenarios in episodes outside the assertions into consideration, more information of the trace can be used to make a decision.

Still, the *Belief* value of an assertion is computed. If *Belief* equals 1, the assertion is definitely true. Otherwise, for the case that *Belief* is lower than 1, *failRate* is computed to measure the influence of missing assignments on the truth of the assertion.

We define $negAssign\,(F_{A\to C})$ as the set of assignments combinations of free variables not covered in $T_{A\to C}$.

$$\begin{aligned} & negAssign\,(F_{A\to C}) \\ = \; & Assign\,(F_{A\to C}) - posAssign\,(F_{A\to C})\,. \end{aligned}$$

Based on the definition of $negAssign\,(F_{A\to C})$, $failRate\,(A \to C)$ can be defined as the fraction of $negAssign\,(F_{A\to C})$ that has an output conflict with C.

$$\begin{aligned} & failRate\,(A \to C) \\ = \; & \frac{\sum_{s_i \in negAssign(F_{A\to C})} support\,(s_i)\,|T_{!C}}{\sum_{s_i \in negAssign(F_{A\to C})} support\,(s_i)\,|T}. \end{aligned}$$

Actually, $failRate$ quantifies the influence of missing free variable assignments on the output. High $failRate$ means that the missing assignments have a high possibility of occurrence along with a conflict output, which will falsify the assertion. Low $failRate$ indicates that the missing assignments will not change the output and the truth of the assertion will not be influenced by missing assignments.

Based on the definition of the $Belief$-$failRate$ metric, the evaluation phase computes the *Belief* value of the assertion to be verified first. If *Belief* is higher than threshold *TB*, the assertion is predicted as true. Otherwise, the *failRate* value of the assertion is computed. If the *failRate* value is lower than threshold *TF*, the assertion is predicted as true. Otherwise, the assertion is predicted as false. This evaluation method also gives rankings to assertions as follows: the assertions whose *Belief* values are higher than *TB* stay at the top of the ranking list and are ranked using their *Belief* values, while those that have *Belief* values lower than *TB* stay at the bottom of the ranking list and are further ranked using their *failRate* values.

Going back to the example in Table 1. Assume that the threshold of *Belief* and *failRate* are both 1/2. For assertion A_1, the free variable set is $\{[1]a, [2]a\}$ and the number of all possible assignment combinations of free variables $[1]a$ and $[2]a$ is $2 \times 2 = 4$. The only simulation episode covered by A_1 is $t9$ and only one free variable assignment combination is covered by these episodes. Thus, $Belief(A_1) = 1/4$. For assertion A_2, the free variable set is $\{[2]a\}$ and the number of all possible combinations of free variables $[2]a$ is $2^1 = 2$. The covered episodes of A_2 are $t8$ and $t10$ and two free variable assignment combinations are occurred. Thus, *Belief*$(A_2) = 2/2 = 1$. For assertions A_3 and A_4, the free variable set is $\{\ \}$ and the number of all possible combinations of free variables is $2^0 = 1$. The covered episode of A_3 and A_4 is $t11$ and $t5$, respectively, and one free variable assignment combination is occurred. Thus, *Belief*$(A_3) = 1/1 = 1$, *Belief*$(A_4) = 1/1 = 1$. For assertion A_5, the free variable set is $\{[1]a, [3]a\}$ and the number of all possible combinations of free variables $[1]a$ and $[3]a$ is $2^2 = 4$. The covered episodes of A_5 include $t1$–$t2$ and $t4$, i.e., three free variable assignment combinations are occurred. Thus, *Belief*$(A_5) = 3/4$. The free variable set of assertion A_6 is $\{[3]a\}$ and the covered episode is $t7$. As one free variable assignment combination is occurred, *Belief*$(A_6) = 1/2$. Similarly, the free variable set of assertion A_7 is $\{[3]a\}$ and the covered episodes are $t3$ and $t6$. As two free variable assignment combinations are occurred, *Belief*$(A_7) = 2/2 = 1$.

As the *Belief* values of A_2, A_3, A_4, A_5 and A_7 are bigger than the threshold, they are predicted as true. For other assertions which have *Belief* values lower than the threshold, the *failRate* values are computed for further decision. For assertion A_1, the absent scenarios are $\{[1]a \wedge [2]a\} = \{01, 10, 11\}$ and they are covered by $t1$–$t6$ and $t10$–$t11$. Among eight output of $t1$–$t6$ and $t10$–$t11$, six of them are in conflict with the output of A_1. Thus, *failRate*$(A_1) = 6/8$. For assertion A_6, the absent scenario is $\{[3]a\} = \{1\}$ and it is covered by $t1$, $t3$–$t4$ and $t9$. Among four output of $t1$, $t3$–$t4$ and $t9$, three of them are in conflict with the output of A_6. Thus, *failRate*$(A_6) = 3/4$. As the threshold of the *failRate* value is 1/2, assertions A_1 and A_6 are predicted as false.

In practical applications, the free variable set considered in assertion evaluation can include only a subset of all free variables to save computation overhead. Variables can be divided into control variables and data variables, which can be tackled in different ways. As control variables play essential roles in driving the state transitions of the design, all of them are added to the free variable set immediately. On the other hand, as data variables are tender to have weak influence on transition, we retain only data variables that occur in some antecedents to the free variable set. This strategy of selecting free variables allows a more convincing computation result of *Belief* and *failRate*.

3.2.3 Constraining Assertions

After the evaluation of assertions, a constraining procedure is proposed for high *failRate* assertions, in order to increase the possibility of their subsets to pass the design. Algorithm 1 shows the workflow of the *Belief-failRate* evaluation and constraining procedure. In the algorithm, P_c refers to the mined assertions to be measured, T refers to the simulation trace, and V refers to the variables of the design.

Algorithm 1. Evaluation and Constraining

```
procedure $AssEvaAndConst\ (V,\ T,\ P_c,\ TB,\ TF)$
1:  $P_f \leftarrow \emptyset$
2:  while $P_c! = \emptyset$:
3:      $A \rightarrow C = P_c.pop()$
4:      $belief = \boldsymbol{Belief}(A \rightarrow C)$
5:      if $belief > TB$ :
6:          $P_f.push(A \rightarrow C)$
7:          continue
8:      $failrate \leftarrow \boldsymbol{failRate}(A \rightarrow C)$
9:      if $failrate \geqslant TF$ :
10:         $p_{\max} = \{p|\max\{T_p \bigcap T_{A\rightarrow C}\},\ p \in$ atomic props of $V\}$
11:         $T_{\max} = T_{p_{\max} * A}$
12:         $P_c.push(A * p_{\max} \rightarrow C)$
13:         if $T_{\max} < T_{A\rightarrow C}$ :
14:             $P_c.push(A*!p_{\max} \rightarrow C)$
15:         else:
16:             $P_f.push(A*!p_{\max} \rightarrow !C)$
17:     else:
18:         $P_f.push(A \rightarrow C)$
19: return $P_f$
```

The constraining procedure involves two steps. First, an atomic proposition $p_{\max}$ is identified, which has the maximum intersection with the assertion $A \rightarrow C$ in the simulation trace, i.e., $\{p|\max\{T_p \cap T_{A\rightarrow C}\}\}$. Adding constraint $p_{\max}$ to $A \rightarrow C$ will lead to a new assertion $A * p_{\max} \rightarrow C$. As A is split by $p_{\max}$ and $!p_{\max}$, the consequent of $A*!p_{\max}$ is to be determined. If $!p_{\max}$ has no intersection with $A \rightarrow C$, a new assertion which predicts the absent scenario $A*!p_{\max}$ will lead to a proposition when a negative C is generated. Otherwise, assertion $A{*}!p_{\max} \rightarrow C$ is added to P_c for further evaluation.

Return to the example in Table 1 again. For assertion A_1, proposition $\{p_{\max}$: $[3]a = 0\}$ is found, two new assertions A_8: $(a = 0) \rightarrow ([3]a = 1) \wedge ([1]a = 0) \rightarrow (O = 0)$ and A_9: $(a = 0) \wedge ([3]a = 1) \wedge ([1]a = 1) \rightarrow (O = 0)$ are generated. For assertion A_8, the free variable set is $\{[2]a\}$ and the covered episode is $t9$, thus *Belief*$(A_8) = 1/2$. As the absent scenario is $\{[2]a\} = \{1\}$ and it is covered by $t1$, $t2$, $t4$, $t5$ and $t10$, among the five outputs $t1$, $t2$, $t4$, $t5$ and $t10$, four of them are in conflict with the output of A_8. Thus, *failRate*$(A_8) = 4/5$. As the threshold of *failRate* is $1/2$, assertion A_8 is predicted as false and the constraining procedure continues for assertion A_8. Further, proposition $\{p_{\max}$:$[2]a = 0\}$ is found, two new assertions A_{10}: $(a = 0) \wedge ([3]a = 1) \wedge ([1]a = 0) \wedge ([2]a = 0) \rightarrow (O = 0)$ and A_{11}: $(a = 0) \wedge ([3]a = 1) \wedge ([1]a = 0) \wedge ([2]a = 1) \rightarrow (O = 1)$ are generated. As no free variables exist, *Belief*(A_{10}) $=$ *Belief*$(A_{11}) = 1$. For assertion A_9, there is no free variable combination; therefore constraining stops.

For assertion A_6, proposition $\{p_{\max}$: $[3]a = 0\}$ is found, two new assertions A_{12}: $(a = 1) \wedge ([2]a = 0) \wedge ([1]a = 0) \wedge ([3]a = 0) \rightarrow (O = 0)$ and A_{13}: $(a = 1) \wedge ([2]a = 0) \wedge ([1]a = 0) \wedge ([3]a = 1) \rightarrow (O = 1)$ are generated. Since the *Belief* value of A_{12} and A_{13} equals 1, the loop terminates.

At the end of the procedure, we get the final assertion set: $\{A_2, A_3, A_4, A_5, A_7, A_{10}, A_{11}, A_{12}, A_{13}\}$. Note all the assertions in this set are true, it shows the prediction by the Belief-failRate framework is likely to be accurate. The new generated true assertions in the constraining procedure are A_{10}, A_{11}, A_{12} and A_{13}, and they cover all the four free variable assignments of $(a =1) \wedge ([1]a = 0) \wedge ([3]a = 1)$ and $(a =1) \wedge ([1]a = 0) \wedge ([2]a = 0)$. As the design has four Boolean variables which lead to a total input space as $2{\times}2{\times}2{\times}2 = 16$, the percentage of input space newly covered by assertions A_{10}, A_{11}, A_{12} and A_{13} is $4/16 \times 100\% = 25\%$.

4 Experimental Results

In this section, we discuss the experimental results of the proposed *Belief-failRate* based assertion evaluation and constraining framework. The benchmarks are taken from verification interacting with synthesis (VIS). All experiments were performed on a Linux Server with eight 2.13 GHz Intel Xeon CPUs and 1 T memory. VIS is used as the formal verification tool, and GoldMine [1–3] as the assertion miner.

For format compatibility to the formal verifier, all benchmarks are from the VIS benchmark set. The features of designs are shown in Table 3, which includes the number of primary inputs (PI), the number of primary outputs (PO), the number of flip-flops (#FF), the FSM depth (Depth), and the number of injected errors (#Errors). The assertion mining process of GoldMine was executed in all four modes (i.e., tree [1, 2], forest, prism, and coverage [3]), and using 100 cycles, 10 000 (10k) cycles and 1 000 000 (1M) cycles of simulation data respectively. Each design was expanded with several sequential depths and the one on which GoldMine generates most assertions was selected, and considered as the best depth for further evaluation and constraining. The best depth is 5 for b02 and b06, and 2 for b10 and b13.

Table 3. Features of Benchmarks

Benchmark	PI	PO	#FF	Depth	#Errors
b02	1	1	4	5	33
b06	2	6	9	4	105
b10	11	6	20	20	125
b13	10	10	53	2 151	160

4.1 Number of Assertions

Table 4 shows the number of assertions generated by GoldMine (GM) and the number of true assertions (GM-T) among the generated ones. The success rate (T-rate) of assertion generation is the ratio of GM-T to GM. Cells labeled with "–" means that time-out occurred in the assertion generation phase, and we will escape these cases in our further analysis. Assume that there are l variables considered, and the length of the simulation trace is k, the complexity of generating assertions under four engines differs from each other. The fast one is the tree engine, whose worst complexity is $O\left(l \times k \times \log_2 k\right)$. The slowest engine is coverage, whose worst complexity is $O\left(3^l\right)$ [3]. The implementation of the four engines tries to reduce the generation time by constraining the number of propositions in the antecedent of an assertion. In our experiments, we set a time-out of one hour, and the generation phase of each engine for most designs can be done in a reasonable time.

Information can be learned from Table 4. For design b02 whose size is small, the simulation data is relatively complete even under 100 cycles simulation. Thus, all engines can generate assertions with a high success rate. For middle-sized b06 and b10, the completeness of the

Table 4. Number of Assertions Generated

Design	Engine	100 Cycles			10k Cycles			1M Cycles		
		GM	GM-T	T-Rate	GM	GM-T	T-Rate	GM	GM-T	T-Rate
b02	tree	7	7	1.00	5	5	1.00	5	5	1.00
	forest	7	7	1.00	6	6	1.00	6	6	1.00
	prism	7	7	1.00	7	7	1.00	7	7	1.00
	coverage	8	7	0.88	8	8	1.00	8	8	1.00
b06	tree	122	68	0.56	221	207	0.94	223	208	0.93
	forest	150	79	0.53	221	205	0.93	227	189	0.83
	prism	131	43	0.33	205	165	0.80	212	187	0.88
	coverage	109	30	0.28	189	145	0.77	190	145	0.76
b10	tree	94	21	0.22	53	42	0.79	52	42	0.81
	forest	96	21	0.22	53	42	0.79	52	42	0.81
	prism	122	18	0.15	58	39	0.67	44	36	0.82
	coverage	94	15	0.16	66	50	0.76	–	–	–
b13	tree	50	47	0.94	59	55	0.93	59	55	0.93
	forest	65	52	0.80	69	64	0.93	65	60	0.92
	prism	56	50	0.89	65	60	0.92	65	60	0.92
	coverage	76	56	0.74	–	–	–	–	–	–

simulation data is closely related to the number of simulations cycles. Thus, the success rate increases with the number of simulation cycles. For larger-sized b13, as the increase of simulation cycles has little influence on the completeness of simulation data, the generation under all cycles can get assertion sets with a similar success rate but a small number.

4.2 Assertion Evaluation

The mission of the *Belief-failRate* evaluation phase is to distinguish false assertions from the assertions set.

In our experiment, to make comparison with both ranking methods and true/false evaluation methods, all assertions are ranked using the *Belief* values first and then the assertions whose *Belief* is lower than *TB* are ranked using their *failRate* values. Under this ranking circumstance, the threshold *TB* should be set carefully and *TF* can be set to an arbitrary value. On the one hand, *Belief* and *failRate* should play equal importance in ranking assertions. On the other hand, the simulation trace is far from complete in practical designs, while absent scenarios in the whole trace are easier to find. Thus, we set threshold *TB* to a rather lower value 0.3, a value that is lower than 0.5 but not much, and *TF* is set to 1. The *Belief-failRate* evaluation metric is compared with support-confidence[1], correlation coefficients based assertion ranking[13], and *Belief*[14]. Ranked assertions by each method are equally divided into four groups, recorded as Q4, Q3, Q2 and Q1. Q4 is the highest quarter of the ranked assertion set, while Q1 is the lowest quarter of the ranked assertion set. Ideally, good metrics can rank true assertions in higher positions while false assertions in lower positions. That is, more true assertions in Q4 and Q3 or more false assertions in Q2 and Q1 are preferable, while more false assertions in Q4 and Q3 or more true assertions in Q1 and Q2 indicate a worse evaluation.

4.2.1 *Comparison with Correlation Coefficient*

Table 5 gives the number of true/false assertions after the ranking of *Belief-failRate* (BF column) in comparison with that by correlation coefficient[13] (Corrs column). The numbers of false assertions and true assertions in each group are shown in the #F columns and the #P columns, respectively. Bold items label cases that one method outperforms another.

On the whole, there are 24 cases in which the $Belief$-$failRate$ metric performs better than correlation coefficient, and in eight cases the correlation coefficient ranking is better. For the remaining 13 cases, the two methods get the same results. It can be concluded that, as the correlation coefficient metric is proposed to rank true assertions, it does not perform so well as the $Belief$-$failRate$ metric, which aims at the evaluation of truth/falseness of mined assertions.

4.2.2 *Comparison with Support-Confidence*[1]

For assertion generation from simulation traces of golden design, as done by GoldMine, the Confidence values of generated assertions are definitely 1. That is, there are no cases in which the antecedent of an assertion is true but the consequent is false. Under this cir-

Table 5. Comparison Between Correlation Coefficient and *Belief-failRate*

Design	Number of Cycles	Engine	Q1				Q2				Q3				Q4			
			Corrs		BF		Corrs		BF		Corrs		BF		Corrs		BF	
			#F	#P	#F	#P	#F	#P	#F	#P	#F	#P	#F	#P	#F	#P	#F	#P
b02	100	tree	0	1	0	1	0	2	0	2	0	2	0	2	0	2	0	2
		forest	0	1	0	1	0	2	0	2	0	2	0	2	0	2	0	2
		prism	0	1	0	1	0	2	0	2	0	2	0	2	0	2	0	2
		coverage	0	2	**1**	**1**	0	2	**0**	**2**	0	2	**0**	**2**	1	1	**0**	**2**
	10k	tree	0	1	0	1	0	1	0	1	0	1	0	1	0	2	0	2
		forest	0	1	0	1	0	2	0	2	0	1	0	1	0	2	0	2
		prism	0	1	0	1	0	2	0	2	0	2	0	2	0	2	0	2
		coverage	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2
	1M	tree	0	1	0	1	0	1	0	1	0	1	0	1	0	2	0	2
		forest	0	1	0	1	0	2	0	2	0	1	0	1	0	2	0	2
		prism	0	1	0	1	0	2	0	2	0	2	0	2	0	2	0	2
		coverage	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2
b06	100	tree	7	23	**23**	**7**	14	17	**18**	**13**	18	12	**8**	**22**	15	16	**5**	**26**
		forest	11	26	**30**	**7**	19	19	**19**	**19**	19	18	**14**	**23**	22	16	**8**	**30**
		prism	21	11	**29**	**3**	21	12	**26**	**7**	25	8	**15**	**18**	21	12	**18**	**15**
		coverage	22	5	**26**	**1**	18	9	**22**	**5**	22	5	**16**	**11**	17	11	**15**	**13**
	10k	tree	4	51	**10**	**45**	5	50	**4**	**51**	3	52	**0**	**55**	2	54	**0**	**56**
		forest	5	50	**11**	**44**	5	50	**5**	**50**	4	51	**0**	**55**	2	54	**0**	**56**
		prism	14	37	**16**	**35**	11	40	**18**	**33**	10	41	**6**	**45**	5	47	**0**	**52**
		coverage	15	32	**21**	**26**	12	35	**16**	**31**	12	35	**7**	**40**	5	43	**0**	**48**
	1M	tree	3	52	**11**	**44**	4	52	**4**	**52**	5	51	**0**	**56**	3	53	**0**	**56**
		forest	15	41	**17**	**39**	11	46	**15**	**42**	10	47	**6**	**51**	2	55	**0**	**57**
		prism	8	45	**17**	**36**	6	47	**8**	**45**	7	46	**0**	**53**	4	49	**0**	**53**
		coverage	15	32	**23**	**24**	13	35	**14**	**34**	12	35	**8**	**39**	5	43	**0**	**48**
b10	100	tree	**23**	**0**	16	7	**19**	**5**	17	7	**16**	**7**	20	3	**15**	**9**	20	4
		forest	**24**	**0**	17	7	**19**	**5**	17	7	**16**	**8**	21	3	**16**	**8**	20	4
		prism	**30**	**0**	23	7	**30**	**1**	25	6	**26**	**4**	30	0	**18**	**13**	26	5
		coverage	**23**	**0**	17	6	**22**	**2**	21	3	**21**	**2**	22	1	**13**	**11**	19	5
	10k	tree	3	10	4	9	3	10	3	10	5	8	0	13	0	14	4	10
		forest	3	10	4	9	3	10	3	10	5	8	0	13	0	14	4	10
		prism	6	8	**7**	**7**	5	10	**7**	**8**	5	9	**5**	**9**	3	12	**0**	**15**
		coverage	4	12	**6**	**10**	5	12	**10**	**7**	6	10	**0**	**16**	1	16	**0**	**17**
	1M	tree	1	12	**6**	**7**	3	10	**4**	**9**	2	11	**0**	**13**	4	9	**0**	**13**
		forest	1	12	**6**	**7**	3	10	**4**	**9**	2	11	**0**	**13**	4	9	**0**	**13**
		prism	2	9	**5**	**6**	3	8	**3**	**8**	1	10	**0**	**11**	2	9	**0**	**11**
b13	100	tree	1	11	1	11	2	11	0	13	0	12	0	12	**0**	**13**	2	11
		forest	**8**	**8**	5	11	5	11	6	10	0	16	0	16	**0**	**17**	2	15
		prism	**6**	**8**	0	14	0	14	0	14	0	14	1	13	**0**	**14**	5	9
		coverage	**10**	**9**	0	19	**9**	**10**	2	17	**1**	**18**	5	14	**0**	**19**	13	6
	10k	tree	1	13	**3**	**11**	0	15	0	15	1	14	**0**	**15**	2	13	**1**	**14**
		forest	2	15	**3**	**14**	0	17	0	17	1	16	**0**	**17**	2	16	**2**	**16**
		prism	1	15	**4**	**12**	0	16	0	16	2	14	**0**	**16**	2	15	**1**	**16**
	1M	tree	1	13	**3**	**11**	0	15	0	15	1	14	**0**	**15**	2	13	**1**	**14**
		forest	2	14	**3**	**13**	0	16	0	16	1	15	**0**	**16**	2	15	**2**	**15**
		prism	2	14	**3**	**13**	0	16	0	16	1	15	**0**	**16**	2	15	**2**	**15**

cumstance, the ranking using the Support-Confidence metric is equal to the ranking using Support only. Thus, ranked assertions using their Support values are implemented as the result of Support-Confidence metric in our experiments.

Table 6 gives the number of true/false assertions after the ranking of *Belief-failRate* (BF column) in comparison with that by Support-Confidence (Support column). Again bold items label cases that one method outperforms another. As all the ranking results of both

Table 6. Comparison Between Support-Confidence and *Belief-failRate*

Design	Number of Cycles	Engine	Q1				Q2				Q3				Q4			
			Support		BF		Support		BF		Support		BF		Support		BF	
			#F	#P	#F	#P	#F	#P	#F	#P	#F	#P	#F	#P	#F	#P	#F	#P
b06	100	tree	14	16	**23**	**7**	7	24	**18**	**13**	19	11	**8**	**22**	14	17	**5**	**26**
		forest	18	19	**30**	**7**	16	22	**19**	**19**	22	15	**14**	**23**	15	23	**8**	**30**
		prism	25	7	29	3	23	10	26	7	25	8	15	18	15	18	18	15
		coverage	23	4	26	1	22	5	22	5	22	5	16	11	12	16	15	13
	10k	tree	7	48	**10**	**45**	1	54	**4**	**51**	5	50	**0**	**55**	1	55	**0**	**56**
		forest	7	48	**11**	**44**	2	53	**5**	**50**	6	49	**0**	**55**	1	55	**0**	**56**
		prism	12	39	**16**	**35**	14	37	**18**	**33**	11	40	**6**	**45**	3	49	**0**	**52**
		coverage	17	30	**21**	**26**	15	32	**16**	**31**	9	38	**7**	**40**	3	45	**0**	**48**
	1M	tree	7	48	**11**	**44**	0	56	**4**	**52**	4	52	**0**	**56**	4	52	**0**	**56**
		forest	15	41	**17**	**39**	13	44	**15**	**42**	7	50	**6**	**51**	3	54	**0**	**57**
		prism	4	49	**17**	**36**	8	45	**8**	**45**	11	42	**0**	**53**	2	51	**0**	**53**
		coverage	17	30	**23**	**24**	16	32	**14**	**34**	9	38	**8**	**39**	3	45	**0**	**48**
b10	100	tree	**20**	**3**	16	7	**21**	**3**	17	7	**17**	**6**	20	3	**15**	**9**	20	4
		forest	**21**	**3**	17	7	**21**	**3**	17	7	**18**	**6**	21	3	**15**	**9**	20	4
		prism	**30**	**0**	23	7	**27**	**4**	25	6	**23**	**7**	30	0	**24**	**7**	26	5
		coverage	**23**	**0**	17	6	**20**	**4**	21	3	**18**	**5**	22	1	**18**	**6**	19	5
	10k	tree	4	9	4	9	**4**	**9**	3	10	3	10	0	13	**0**	**14**	4	10
		forest	4	9	4	9	**4**	**9**	3	10	3	10	0	13	**0**	**14**	4	10
		prism	7	7	**7**	**7**	5	10	**7**	**8**	3	11	**5**	**9**	4	11	**0**	**15**
		coverage	5	11	**6**	**10**	6	11	**10**	**7**	4	12	**0**	**16**	1	16	**0**	**17**
	1M	tree	3	10	**6**	**7**	2	11	**4**	**9**	4	9	**0**	**13**	1	12	**0**	**13**
		forest	3	10	**6**	**7**	2	11	**4**	**9**	4	9	**0**	**13**	1	12	**0**	**13**
		prism	2	9	**5**	**6**	3	8	**3**	**8**	1	10	**0**	**11**	2	9	**0**	**11**
b13	100	tree	1	11	1	11	**1**	**12**	0	13	0	12	0	12	**1**	**12**	2	11
		forest	**9**	**7**	5	11	**2**	**14**	6	10	**1**	**15**	0	16	**1**	**16**	2	15
		prism	**1**	**13**	0	14	0	14	0	14	1	13	1	13	**4**	**10**	5	9
		coverage	**4**	**15**	0	19	**0**	**19**	2	17	**9**	**10**	5	14	**7**	**12**	13	6
	10k	tree	**4**	**10**	3	11	0	15	0	15	0	15	0	15	**0**	**15**	1	14
		forest	**5**	**12**	3	14	0	17	0	17	0	17	0	17	**0**	**18**	2	16
		prism	**5**	**11**	4	12	0	16	0	16	0	16	0	16	**0**	**17**	1	16
	1M	tree	**4**	**10**	3	11	0	15	0	15	0	15	0	15	**0**	**15**	1	14
		forest	**5**	**11**	3	13	0	16	0	16	0	16	0	16	**0**	**17**	2	15
		prism	**5**	**11**	3	13	0	16	0	16	0	16	0	16	**0**	**17**	2	15

methods for design b02 are the same, we omit the data of b02 in Table 6.

It can be concluded that, there are 15 cases when the Belief-failRate method performs better than Support-Confidence, and in 16 cases, the Support-Confidence ranking is better. For the remaining 14 cases, the two methods get the same results. Also, it can be learned from the distribution of all these three kinds of cases that, the *Belief-failRate* metric performs well for ranking assertions of design b06, whose success rate is low. On the other hands, the Support-Confidence ranking is better for design b10 and b13, whose assertions have a higher success rate. This result indicates that the Support-Confidence metric is more suitable for true assertions ranking while the proposed *Belief-failRate* metric performs better in distinguishing false assertions from an assertion set with a large number of false assertions.

4.2.3 *Comparison with Belief*[14]

Before comparing the results of *Belief* and the *Belief-failRate* metric, we take a look at the range of *Belief* between all generated assertions for each design, which reflects the completeness of the corresponding simulation trace. Table 7 gives the maximum and the minimum *Belief* value of the assertion set generated under each configuration. It is clear that the range of *Belief* differs greatly for different assertion sets. The bold items show the cases that the *Belief* values of generated assertions are widely distributed. In other cases, the *Belief* values of all generated assertions are either with similar low values or with similar high values. As the threshold *TB* is set to 0.3, it can be expected that the proposed *Belief-failRate* metric performs well in the former cases, where both *Belief* and *failRate* play a role. For the latter cases, where the result of Belief-failRate is the effect of only *Belief* or only *failRate*, the results may be not so good.

Table 7. Range of *Belief* Values

		100 Cycles				10k Cycles				1M Cycles			
		tree	forest	prism	coverage	tree	forest	prism	coverage	tree	forest	prism	coverage
b06	Min	**0.000 0**	**0.000 0**	**0.125 0**	**0.000 0**	0.500 00	0.500 00	0.500 00	0.500	0.500 0	0.500 00	0.500 00	0.5
	Max	**1.000 0**	**1.000 0**	**0.750 0**	**0.500 0**	1.000 00	1.000 00	1.000 00	1.000	1.000 0	1.000 00	1.000 00	1.0
b10	Min	0.015 0	0.015 0	0.015 0	0.007 0	**0.031 25**	**0.031 25**	**0.007 00**	**0.007**	**0.007 0**	**0.007 00**	**0.007 00**	–
	Max	0.437 5	0.437 5	0.312 5	0.437 5	**0.500 00**	**0.500 00**	**0.500 00**	**1.000**	**1.000 0**	**1.000 00**	**0.500 00**	–
b13	Min	**0.187 5**	**0.125 0**	0.003 0	0.000 9	0.001 90	0.000 24	0.000 24	––	0.001 9	0.000 24	0.000 24	–
	Max	**0.580 0**	**0.580 0**	0.010 0	0.005 0	0.045 00	0.005 80	0.005 80	––	0.045 0	0.005 80	0.005 80	–

Table 8. Comparison Between *Belief* and *Belief-FailRate*

Design	Number of Cycles	Engine	Q1				Q2				Q3				Q4			
			Belief		BF		Belief		BF		Belief		BF		Belief		BF	
			#F	#P	#F	#P	#F	#P	#F	#P	#F	#P	#F	#P	#F	#P	#F	#P
b06	100	tree	21	9	**23**	**7**	20	11	**18**	**13**	8	22	8	22	5	26	5	26
		forest	27	10	**30**	**7**	22	16	**19**	**19**	14	23	14	23	8	30	8	30
		prism	25	7	**29**	**3**	30	3	**26**	**7**	15	18	15	18	18	15	18	15
		coverage	24	3	**26**	**1**	24	3	**22**	**5**	16	11	16	11	15	13	15	13
	10k	tree	10	45	10	45	4	51	4	51	0	55	0	55	0	56	0	56
		forest	11	44	11	44	5	50	5	50	0	55	0	55	0	56	0	56
		prism	16	35	16	35	18	33	18	33	6	45	6	45	0	52	0	52
		coverage	21	26	21	26	16	31	16	31	7	40	7	40	0	48	0	48
	1M	tree	11	44	11	44	4	52	4	52	0	56	0	56	0	56	0	56
		forest	17	39	17	39	15	42	15	42	6	51	6	51	0	57	0	57
		prism	17	36	17	36	8	45	8	45	0	53	0	53	0	53	0	53
		coverage	23	24	23	24	14	34	14	34	8	39	8	39	0	48	0	48
b10	100	tree	**20**	**3**	16	7	**20**	**4**	17	7	**21**	**2**	20	3	**12**	**12**	20	4
		forest	**21**	**3**	17	7	**20**	**4**	17	7	**22**	**2**	21	3	**12**	**12**	20	4
		prism	**27**	**3**	23	7	**28**	**3**	25	6	27	3	30	0	**22**	**9**	26	5
		coverage	**19**	**4**	17	6	**24**	**0**	21	3	20	3	22	1	**16**	**8**	19	5
	10k	tree	4	9	4	9	3	10	3	10	0	13	0	13	4	10	4	10
		forest	4	9	4	9	3	10	3	10	0	13	0	13	4	10	4	10
		prism	5	9	**7**	**7**	8	7	7	8	6	8	**5**	**9**	0	15	0	15
		coverage	5	11	**6**	**10**	9	8	**10**	**7**	2	14	**0**	**16**	0	17	0	17
	1M	tree	3	10	**6**	**7**	6	7	4	9	1	12	**0**	**13**	0	13	0	13
		forest	3	10	**6**	**7**	6	7	4	9	1	12	**0**	**13**	0	13	0	13
		prism	3	8	**5**	**6**	4	7	3	8	1	10	**0**	**11**	0	11	0	11
b13	100	tree	0	12	**1**	**11**	1	12	**0**	**13**	0	12	0	12	2	11	2	11
		forest	4	12	**5**	**11**	7	9	**6**	**10**	0	16	0	16	2	15	2	15
		prism	**4**	**10**	0	14	**1**	**13**	0	14	**0**	**14**	1	13	**1**	**13**	5	9
		coverage	**11**	**8**	0	19	1	18	2	17	**1**	**18**	5	14	**7**	**12**	13	6
	10k	tree	**4**	**10**	3	11	0	15	0	15	0	15	0	15	**0**	**15**	1	14
		forest	**4**	**13**	3	14	0	17	0	17	0	17	0	17	**1**	**17**	2	16
		prism	**5**	**11**	4	12	0	16	0	16	0	16	0	16	**0**	**17**	1	16
	1M	tree	**4**	**10**	3	11	0	15	0	15	0	15	0	15	**0**	**15**	1	14
		forest	**4**	**12**	3	13	0	16	0	16	0	16	0	16	**1**	**16**	2	15
		prism	**4**	**12**	3	13	0	16	0	16	0	16	0	16	**1**	**16**	2	15

Table 8 shows the number of true/false assertions after the ranking of *Belief* (Belief column) in comparison with that by *Belief-failRate* (BF column). Similarly, all the same results of design b02 are omitted in Table 8.

To sum up, there are 11 cases when the *Belief-failRate* metric performs better than *Belief*, and in 12 cases, the ranking using *Belief* is better. For the remaining 22 cases, the two methods get the same results. Further observation shows that, the result is influenced by the threshold *TB* which is uniformly set without considering the distribution of the *Belief* values, and is consistent with our expectation. That is, for the cases where the *Belief* values are widely distributed, the *Belief-failRate* metric performs well. For other cases, the proposed metric loses its advantage.

Some conclusions can be learned from the experimental data above. Firstly, Support-Confidence and correlation coefficients are more suitable for ranking of true assertions, while *Belief-failRate* and *Belief* perform well in evaluation of truth/falseness of mined assertions. Secondly, the *Belief* metric is effective when the *Belief* values of mined assertions are narrowly distributed. Otherwise, the $Belief\text{-}failRate$ metric provides an effective supplement when the *Belief* metric is no longer effective.

4.2.4 Runtime of Assertion Ranking

Here, we compare the runtime of each assertion ranking method. It can be learned from the definition of Support-Confidence and correlation coefficient that their runtime is mainly consumed by the traverse of the simulation trace. After such a traverse, the computation of Support-Confidence and correlation coefficient consumes constant time. Similarly, the computation complexity of *Belief* and that of *Belief-failRate* are the same, which are mainly determined by the searching time of free variable combinations. Thus, instead of comparing the runtime of all four metrics, we compare the two classes of methods, i.e., the traverse-based and the free variable combinations based. Table 9 gives the runtime of the traverse-based metrics (S&C) and the free variable combinations based metrics (B&BF). Bold items label cases where the runtime of one class of metric is lower than another.

Results show that the runtime of *Belief* and Belief-failRate class is lower than that of Support-Confidence and correlation coefficient in most cases. For b13, who has a large number of variables, the runtime of traverse-based metrics is lower.

Table 9. Runtime of Assertion Ranking (s)

Design	Engine	100 Cycles		10k Cycles		1M Cycles	
		B&BF	S&CC	B&BF	S&CC	B&BF	S&CC
b02	tree	0.01	0.01	0.01	**0.00**	0.01	**0.00**
	forest	0.01	0.01	**0.00**	0.01	**0.00**	0.01
	prism	0.01	0.01	0.01	0.01	0.02	0.00
	coverage	0.01	0.01	0.01	0.01	0.01	0.01
b06	tree	**0.11**	0.44	**0.22**	2.58	**0.22**	2.76
	forest	**0.13**	0.60	**0.22**	2.65	**0.29**	2.51
	prism	**0.19**	0.41	**0.31**	1.85	**0.31**	2.04
	coverage	**0.16**	0.30	**0.31**	1.54	**0.31**	1.55
b10	tree	**0.13**	0.24	**0.31**	0.83	**0.36**	1.04
	forest	**0.13**	0.25	**0.32**	0.86	**0.35**	1.06
	prism	**0.22**	0.32	**0.41**	0.77	**0.38**	0.61
	coverage	0.26	**0.21**	**0.45**	1.19	–	–
b13	tree	0.12	**0.10**	3.83	**0.59**	3.82	**0.58**
	forest	0.16	**0.13**	8.51	**0.76**	7.48	**0.77**
	prism	0.92	**0.10**	19.57	**0.61**	7.30	**0.75**
	coverage	4.43	**0.26**	–	–	–	–

4.3 Assertion Constraining Results

In the constraining phase, the thresholds *TB* and *TF* for the *Belief* value and the *failRate* value are set as 1 and 0, respectively. That is, only assertions with a *Belief* value equal to 1 or a *failRate* value equal to 0 are predicted as true. Otherwise, constraining is performed to generate more assertions.

4.3.1 Number of Generated Assertions

Table 10 gives the number of assertions generated after constraining (#Num) and the number of true assertions (#Num-T) among the generated ones. The success rate (T-rate) of assertion generation is the ratio of #Num-T to #Num. Bold items label cases where

Table 10. Assertions After Constraining

Design	Engine	100 Cycles			10k Cycles			1M Cycles		
		#Num	#Num-T	T-Rate	#Num	#Num-T	T-Rate	#Num	#Num-T	T-Rate
b02	tree	26	17	0.65	5	5	1.00	5	5	1.00
	forest	26	17	0.65	6	6	1.00	6	6	1.00
	prism	49	31	0.63	10	10	1.00	10	10	1.00
	coverage	59	37	0.63	11	11	1.00	11	11	1.00
b06	tree	480	283	**0.59**	350	342	**0.98**	338	329	**0.97**
	forest	610	357	**0.59**	352	344	**0.98**	433	411	**0.95**
	prism	862	480	**0.56**	426	388	**0.91**	394	364	**0.92**
	coverage	750	410	**0.55**	423	383	**0.91**	422	381	**0.90**
b10	tree	510	143	**0.28**	10 773	8 732	**0.81**	338	314	**0.93**
	forest	523	143	**0.27**	3 384	3 257	**0.96**	338	314	**0.93**
	prism	830	248	**0.30**	12 349	8 548	**0.69**	539	500	**0.93**
	coverage	699	214	**0.31**	70	52	0.74	–	–	–
b13	tree	214	56	0.26	621	305	0.49	621	307	0.49
	forest	283	67	0.24	680	354	0.52	680	359	0.53
	prism	211	58	0.27	671	349	0.52	680	359	0.53
	coverage	261	60	0.23	–	–	–	–	–	–

there exist increases of success rate.

From Table 4 and Table 10, we can conclude that the number of true assertions after constraining increases dramatically compared with that without constraining. As for the success rate, the increase is obvious for cases in which the original success rate is low, b06 and b10 for example. For other cases in which the original success rates are relatively high, b02 and b13 for example, the success rate after constraining becomes low. The reason lies in the low threshold $f_{\text{threshold}}$ we set, which will lead to the constraining of many true assertions. For these cases, changing the threshold $f_{\text{threshold}}$ can improve the results.

4.3.2 Covered Errors

Although the constraining is effective in the aspect of the number of assertions being generated, it should be further evaluated in the aspect of functionality coverage. The effective way of function coverage estimation is by the evaluation of bit coverage, which can reflect the functionality well[21]. As all variables and signals of the design can be considered as a sequence of bits, the bit coverage model can be defined as permanent single-bit upset on all used variables of assignment statements and all control signals of branch statements. The number of faults totally injected for each design is shown in the last column of Table 3.

Table 11 compares the number of errors detected by assertions generated in each mode of GoldMine (GM) and that by the assertions after constraining (B-F). Bold items label cases where differences exist.

Table 11. Covered Errors

Design	Engine	100 Cycles		10k Cycles		1M Cycles	
		GM	B-F	GM	B-F	GM	B-F
b02	tree	29	29	29	29	30	30
	forest	30	30	30	30	30	30
	prism	30	30	30	30	30	30
	coverage	30	30	30	30	30	30
b06	tree	**95**	**101**	102	102	102	102
	forest	**98**	**101**	102	102	102	102
	prism	**97**	**101**	102	102	102	102
	coverage	**91**	**101**	102	102	102	102
b10	tree	**57**	**67**	**29**	**71**	**59**	**75**
	forest	**57**	**67**	**13**	**74**	**59**	**75**
	prism	**46**	**64**	**20**	**81**	**60**	**88**
	coverage	**46**	**65**	**52**	**56**	–	–
b13	tree	**30**	**42**	**37**	**65**	**37**	**65**
	forest	**30**	**42**	**37**	**65**	**37**	**65**
	prism	**31**	**57**	**34**	**65**	**37**	**65**
	coverage	**31**	**54**	–	–	–	–

For b02 and b06, whose error coverage is relatively high by assertions generated by GoldMine, there are few improvements in error coverage for assertions after constraining. For b10, the number of covered errors of constrained assertions is 1.07 to 5.69 times of that of assertions generated by GoldMine. For b13, the number of covered errors by constrained assertions is 1.40–1.91 times of that of assertions generated by GoldMine. On the whole, the constraining is effective in that it can generate a large number of true assertions, which can detect many new errors.

4.3.3 Runtime of Assertion Generation

In this subsection, we discuss the runtime of the proposed *Belief-failRate* based constraining procedure. Table 12 compares the number of generated assertions (#Assertions) as well as the runtime (Time) of GoldMine (GM) and the proposed constraining procedure (B-F) under each configuration. Actually, the worst complexity of the proposed algorithm is equal to that of GoldMine, when the assertion constraining procedure needs a search of the complete binary tree. In practice, as shown in Table 12, the runtime is closely correlated with the number of generated assertions, the completeness of simulation trace, as well as the number of free variables. For b02 and b06, in which cases the number of generated assertions by the constraining phase is in the same level with that by GoldMine and the simulation traces are rather complete for the small number of free variables, the runtime of constraining is shorter than that of GoldMine in most cases. For b10 and b13, in which cases the number of generated assertions differs greatly from those of the two methods, the completeness of simulation traces is not good, the number of free variables is large, and the runtime of the constraining phase increases greatly along with the increase of the number of assertions generated. One important reason lies in the fact that, the search depth of the constraining procedure is deeper than that of GoldMine. Taken the worst case for example, i.e., the generation of the forest engine for b10 by 10k cycles, as the search depth increases largely in the constraining phase, the runtime of constraining is 900 times of that by GoldMine. For other cases, for example, the generation of the forest engine for b10 by 1M cycles data the runtime of constraining is about 20 times of that by GoldMine. In some other cases, for example, the generation of the prism engine for b10 by 1M cycles data, as the number of propositions in each generated assertions is large，which indicates a very small number of

Table 12. Assertion Count and Runtime of Assertion Generation

Design	Engine	100 Cycles				10k Cycles				1M Cycles			
		#Assertions		Time (s)		#Assertions		Time (s)		#Assertions		Time (s)	
		GM	B-F	GM	B-F	GM	B-F	GM	B-F	GM	B-F	GM	B-F
b02	tree	7	26	1.13	**0.08**	5	5	1.17	**0.03**	5	5	8.30	**0.04**
	forest	7	26	1.13	**0.09**	6	6	1.17	**0.04**	6	6	7.02	**0.04**
	prism	7	49	1.10	**0.18**	7	10	1.20	**0.04**	7	10	7.14	**0.04**
	coverage	8	59	1.12	**0.19**	8	11	1.18	**0.05**	8	11	7.24	**0.05**
b06	tree	122	480	**1.17**	1.25	221	350	**1.67**	2.42	223	338	9.86	**2.38**
	forest	150	610	**1.17**	1.51	221	352	**1.67**	2.39	227	433	9.94	**2.23**
	prism	131	862	**1.20**	2.14	205	426	3.78	**1.86**	212	394	14.77	**1.98**
	coverage	109	750	**1.26**	1.84	189	423	9.31	**1.73**	190	422	25.13	**1.73**
b10	tree	94	510	**1.19**	4.11	53	10 773	**22.32**	8 347.00	52	338	**116.71**	2 347.00
	forest	96	523	**1.16**	4.22	53	3 384	**2.04**	1 840.00	52	338	**117.30**	2 342.00
	prism	122	830	**1.21**	6.40	58	12 349	**3 091.00**	4 437.00	44	539	6 075.00	**1 850.00**
	coverage	94	699	**1.42**	5.30	66	70	43.96	**26.34**	–	–	–	–
b13	tree	50	214	**1.19**	21.96	59	621	**2.95**	1 094.00	59	621	**202.40**	377.80
	forest	65	283	**1.20**	27.55	69	680	**2.90**	1 011.00	65	680	**205.60**	398.50
	prism	56	211	**1.21**	37.19	65	671	**4.31**	1 185.00	65	680	**205.60**	394.00
	coverage	76	261	942.60	**52.28**	–	–	–	–	–	–	–	–

free variables are available in the constraining phase, the runtime of the constraining procedure is only 30% of that by GoldMine.

5 Conclusions

A hardware assertion evaluation and constraining framework was presented in this paper, which introduces a new metric, *Belief-failRate* for evaluating and constraining assertions mined from simulation traces. The target of the method is the assertions generated from incomplete simulation traces. Generally, simulation traces used for assertion mining are incomplete, and the mined assertions contain a large number of false assertions, which will block the generation of other true assertions. By using the *Belief-failRate* based evaluation, which takes both the percentage of absent scenarios and the occurrence of absent scenarios in the whole trace into consideration, the assertion evaluation performs well in classifying assertions reasonably. Also, the assertion constraining procedure is given to generate more true assertions. Experimental results showed that the proposed method can evaluate assertions with high correctness and generate new assertions with a high success rate for cases in which the success rate of originally mined assertions is low. For other cases in which the success rate of mined assertions is relatively high, the constraining phase can still generate more assertions to detect many new errors of the design.

References

[1] Vasudevan S, Sheridan D, Tcheng D, Tuohy B, Johnson D. GoldMine: Automatic assertion generation using data mining and static analysis. In *Proc. the 13th Int. Conf. Design, Automation & Test in Europe*, March 2010, pp.626-629.

[2] Hertz S, Sheridan D, Vasudevan S. Mining hardware assertions with guidance from static analysis. *IEEE Trans. Computer Aided Design*, 2013, 32(6): 952-965.

[3] Sheridan D, Liu L, Kim H, Vasudevan S. A coverage guided mining approach for automatic generation of succinct assertions. In *Proc. the 27th Int. Conf. VLSI Design*, January 2014, pp.68-73.

[4] Liu L, Lin C H, Vasudevan S. Word level feature discovery to enhance quality of assertion mining. In *Proc. the 15th Int. Conf. Computer-Aided Design*, March 2012, pp.210-217.

[5] Chang P, Wang L C. Automatic assertion extraction via sequential data mining of simulation traces. In *Proc. the 15th Int. Conf. Asia and South Pacific Design Automation*, January 2010, pp.607-612.

[6] Bertasi M, Guglielmo G, Pravadelli G. Automatic generation of compact formal properties for effective error detection. In *Proc. the 11th Int. Conf. Hardware/Software Codesign and System Synthesis*, September 2013, pp.1-10.

[7] Danese A, Ghasempouri T, Pravadelli G. Automatic extraction of assertions from execution traces of behavioural models. In *Proc. the 18th Int. Conf. Design, Automation & Test in Europe*, March 2015, pp.67–72.

[8] Danese A, Filini F, Pravadelli G. A time-window based approach for dynamic assertions mining on control signals. In *Proc. the 23rd Int. Conf. Very Large Scale Integration*, October 2015, pp. 246-251.

[9] Danese A, Pravadelli G, Zandonà I. Automatic generation of power state machines through dynamic mining of temporal assertions. In *Proc. the 19th Int. Conf. Design, Automation & Test in Europe*, March 2016, pp.606–611.

[10] Ciesielski M, Yu C, Brown W, Liu D. Verification of gate-level arithmetic circuits by function extraction. In *Proc. the 52nd Int. Conf. Design Automation*, June 2015.

[11] Hanafy M, Said H, Wahba A M. Complete properties extraction from simulation traces for assertions auto-generation. In *Proc. the 24th Int. Conf. North Atlantic Test Workshop*, May 2015.

[12] Hanafy M, Said H, Wahba A M. New methodology for digital design properties extraction from simulation traces. In *Proc. the 10th Int. Conf. Computer Engineering & Systems*, December 2015.

[13] Ghasempouri T, Pravadelli G. On the estimation of assertion interestingness. In *Proc. the 23rd Int. Conf. Very Large Scale Integration*, October 2015, pp.325-330.

[14] Mitra S, Banerjee A, Dasgupta P, Kumar H. Counterexample ranking using mined invariants. *IEEE Trans. Computer Aided Design*, 2013, 32(12): 1978–1991.

[15] Chao H, Li H, Song X, Wang T, Li X. On evaluating and constraining assertions using conflicts in absent scenarios. In *Proc. the 26th Int. Conf. Asian Test Symposium*, November 2017, pp.195-200.

[16] Ernst M, Cockrell J, Griswold W, Notkin D. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Engineering*, 2001, 27(2): 99–123.

[17] Hoskote Y, Kam T, Ho P H, Zhao X. Coverage estimation for symbolic model checking. In *Proc. the 36th Int. Conf. Design Automation*, June 1999, pp.300–305.

[18] Jayakumar N, Purandare M, Somenzi F. Dos and Don'ts of CTL state coverage estimation. In *Proc. the 40th Int. Conf. Design Automation*, June 2003, pp.292–295.

[19] Chao H, Li H, Wang T, Li X. An accurate algorithm for computing mutation coverage in model checking. In *Proc. the 2016 IEEE International Test Conference*, November 2016.

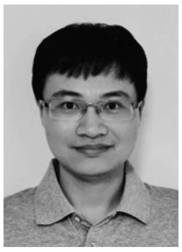
[20] Haedicke F, Große D, Drechsler R. A guiding coverage metric for formal verification. In *Proc. the 15th Int. Conf. Design, Automation & Test in Europe*, March 2012, pp.617-622.

[21] Fedeli A, Fummi F, Pravadelli G. Properties incompleteness evaluation by functional verification. *IEEE Trans. Computers*, 2007, 56(4): 528-544.

Hui-Na Chao received her B.S. degree in computer science and technology from Shandong University of Science and Technology, Qingdao, in 2012. She has been a Ph. D. candidate of the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, since 2012. Her research focuses on functional verification of VLSI, including formal verification and assertion generation.

Hua-Wei Li received her B.S. degree in computer science from Xiangtan University, Xiangtan, in 1996, and her M.S. and Ph.D. degrees in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 1999 and 2001, respectively. She has been a professor at ICT, CAS since 2008. She visited the University of California, Santa Barbara, from 2009 to 2010. She is a senior member of IEEE. Her current research interests include testing of VLSI/SOC circuits, design for reliability, fault tolerance, and approximate computing. She has published over 200 technical papers, and holds 31 Chinese patents. She was a recipient of the 2012 National Technology Invention Award of China. Prof. Li has served as an associate editor of IEEE Transactions on Very Large Scale Integration, and served on the editorial boards of two Chinese Journals: the Journal of Computer-Aided Design and Computer Graphics and the Journal of Computer Research and Development. She was the technical program co-chair of IEEE Asian Test Symposium (ATS) in 2007 and 2018, and the general co-chair of ATS in 2014. She was the technical program co-chair of IEEE International Test Conference in Asia in 2018. She has served as the steering committee chair of IEEE ATS (2020–2022), the steering committee chair of IEEE Workshop on RTL and High Level Testing (2014–2016), the Secretary General of the China Computer Federation (CCF) Technical Committee on Integrated Circuit Design since 2019, the chair of the CCF Technical Committee on Fault-Tolerant Computing (2016–2019), and served on the technical program committees for several IEEE conferences.

Xiaoyu Song received his Ph.D. degree in computer engineering from the University of Pisa, Pisa, in 1991. From 1992 to 1998, he was on the faculty at the University of Montreal, Montreal. He joined the Department of Electrical and Computer Engineering at Portland State University, Portland, in 1998, where he is now a professor. He was an editor of IEEE Transactions on VLSI Systems and IEEE Transactions on Circuits and Systems. He was awarded an Intel Faculty Fellowship from 2000 to 2005. His research interests include formal methods, design automation, embedded systems and emerging technologies.



Tian-Cheng Wang is an engineer of the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing. He received his B.S. degree in computer science and technology from University of Science and Technology of China (USTC), Hefei, in 2006, and his M.S. degree in computer architecture from ICT, CAS, in 2009. His research focuses on functional verification and hardware security. He is a member of CCF.

Xiao-Wei Li received his B.Eng. and M.Eng. degrees in computer science from Hefei University of Technology, Hefei, in 1985 and 1988, respectively, and his Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 1991. From 1991 to 2000, he was an assistant professor and an associate professor (since 1993) in the Department of Computer Science, Peking University, Beijing. He joined ICT, CAS as a professor in 2000. He is now the deputy (executive) director of the State Key Laboratory of Computer Architecture (ICT, CAS). He is a senior member of IEEE. Dr. Li's research interests include VLSI testing, design verification, dependable computing. He has co-published over 200 papers in academic journals and international conference, and holds 50 patents and 45 software copyrights. Dr. Li served as the chair of CCF Technical Committee on Fault Tolerant Computing from 2008 to 2015. He has been serving as the Test Technology Technical Council Vice Chair since 2018. He served as the steering committee chair of the IEEE Asian Test Symposium from 2011 to 2013. He also serves as an editorial board member of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Journal of Computer Science and Technology, the Journal of Electronic Testing: Theory and Applications, and Journal of Low Power Electronics.