Hindawi
Mathematical Problems in Engineering
Volume 2018, Article ID 6398616, 10 pages
https://doi.org/10.1155/2018/6398616

Research Article

Simulation-Based Hardware Verification with a Graph-Based Specification

Zhao Lv, Shuming Chen, and Yaohua Wang

College of Computer, National University of Defense Technology, Changsha 410072, China

Correspondence should be addressed to Yaohua Wang; nudtyh@foxmail.com

Received 18 September 2017; Revised 24 December 2017; Accepted 8 January 2018; Published 8 February 2018

Academic Editor: Xinkai Chen

Simulation-based verification continues to be the primary technique for hardware verification due to its scalability and ease of use; however, it lacks exhaustiveness. Although formal verification techniques can exhaustively prove functional correctness, they are limited in terms of the scale of their design due to the state-explosion problem. Alternatively, semiformal approaches can involve a compromise between scalability, exhaustiveness, and resource costs. Therefore, we propose an event-driven flow graph-based specification, which can describe the cycle-accurate functional behaviors without the exploration of whole state space. To efficiently generate input sequences according to the proposed specification, we introduce a functional automatic test pattern generation (ATPG) approach, which involves the proposed intelligent redundancy-reduction strategy to solve problems of random test vectors. We also proposed functional coverage criterion based on the formal specification to support a more reliable measure of verification. We implement a verification platform based on the proposed semiformal approach and compare the proposed semiformal approach with the constrained randomized test (CRT) approach. The experiment results show that the proposed semiformal verification method ensures a more exhaustive and effective exploration of the functional correctness of designs under verification (DUVs).

1. Introduction

Functional verification is the process that ensures conformance of a design under verification (DUV) to its specification. Due to the rapidly growing complexity of designs and time-to-market requirements, functional verification has become a major blockage in the hardware design cycle in terms of development costs and time.

Formal and simulation-based verification are two major techniques of functional verification. In formal verification, an engine exhaustively exploits the state space of the design to prove the functional properties. However, the state space grows exponentially with the increase of flip-flops in design. With a design with N flip-flops, the corresponding number of states is 2^N; considering a relatively complex design, the state space would be too large to explore. Although formal verification has been improved significantly in recent years, it is still unable to handle complex designs with reasonable computation and storage resources. On the other hand, due to its scalability and ease of use, simulation-based verification is still widely used in industrial manufacturing. However, simulation is time-consuming and labor-intensive; it requires abundant test vectors. Furthermore, due to the lack of exhaustiveness in simulation, the functional coverage is hard to guarantee.

A practical choice is semiformal verification, in which the specification of design functionality is formally completed and checking is undertaken through simulation. And the test vectors are automatically derived from the knowledge of the formal functional specification. Semiformal verification avoids the state-explosion problem due to the absence of exploiting the whole state space. Meanwhile, it improves the degree of automation in generating test vectors and checking functional correctness. Therefore, semiformal verification is a promising method for functional verification.

In this paper, a fast, semiformal verification approach with high scalability and completeness is presented. In this context, three main contributions are provided.

We propose an event-driven flow graph-based specification. The key idea is to order functional behaviors in the event-driven flow graph in a timing-accurate manner, so that the cycle-accurate functional behaviors can be described

without the exploration of whole state space. This can greatly increase the scalability and reduce modeling complexity.

We introduce a functional ATPG approach to efficiently generate input sequences according to the proposed specification. The equality of test-space distribution and the coverage of corner cases are two problems of random test vectors. To solve these two problems, we categorize the input ports into data-path ports and control-path ports. Data-path ports relate to data communication and control-path ports relate to the functional mode change. For the equality of test-space distribution, we propose test-space decomposition strategy to decompose the large test space into small parts and traverse each part evenly. This method ensures the same verification strength among different parts of test space. For the coverage of corner cases, we have two critical findings: (a) the corner cases are always the combinations of all corner values of data-path ports and all legal values of control-path ports; (b) with the guidance of expert knowledge base, the corner values of a data-path port can be inferred from the data type of the data-path port. To this end, we can generate corner cases based on the properties and data types of input ports automatically.

Functional coverage criterion is proposed based on the proposed EDFG and formal specification. In traditional simulation process, the functional coverage provided by these vectors is hardly measured, and it leads to uncertainty of exhaustiveness in the simulation. We define functional coverage based on the proposed specification, and it ensures that the coverage of design functionality provided by vectors is easily measured. Compared with traditional code coverage, the proposed functional coverage supports a more reliable measure in the simulation process.

2. Related Work

Simulation-based verification is still the main method of the functional verification process [1]. To promote verification efficiency, some technical approaches have been developed, such as coverage-driven verification [2], constraint-randomized verification [3], and assertion-based verification [4]. Although these methods accelerate procedure convergence, automatically generate test vectors, and locate defects, there are still some inevitable weaknesses in the simulation. Firstly, compared with the automatic capture of code coverage, the functional coverage of design by input vectors is difficult to measure due to the artificial analysis of the informal specification. Secondly, constrained randomized test (CRT) vectors are still ineffective due to the presence of many redundant vectors and targeting interesting corner test cases requires abundant human labor. Thirdly, assertion-based checking can only check the correctness of the functional properties and not arithmetical operations. Furthermore, the lack of exhaustiveness has historically been a difficult problem to solve.

Formal verification, specifically checking models and proving theorems, provides high-quality and exhaustive verification; however, it cannot be treated as a general solution in industrial verification in the near future [5]. The well-known state-explosion problem is the main limitation of checking models and the costs of skilled manual labor limit proving theorems. Moreover, checking models and proving theorems require detailed formal system specifications that are not always defined or up to date in the design flows. Therefore, design practices limit formal verification.

Semiformal verification is a practical alternative to simulation-based and formal verification. There is some academic and industrial research on semiformal verification, which consists of a standard language specifying correctness criteria and design and vector generation based on constraints [6]. The constraints are derived according to the given language and design. These works adopt different languages, which are categorized as either formal or semiformal.

Existing formal functional behavior descriptions of DUT include the transaction-based behavioral level description, such as SystemC [7, 8], UML [9], and other transactional description methods [10, 11]. These transaction-based descriptions can model functional behaviors into transactions, but they are unable to describe cycle timing sequence of function behaviors. On the other hand, the FSM-based behavior specification, including FSM [12], EFSM [13–15], assignment decision diagram [16, 17], and BDD [18], enables the description of cycle-accurate functional behaviors at the cost of exploring the whole state spaces and transitions, which limits the scale of design it can support.

Semiformal languages currently adopt simple natural languages with a specific format so that they can be processed by machines. These languages are parsed and converted into UML [19, 20] or temporal logic languages [21, 22] to guide vector sequence generation. These semiformal specifications are easy to use, but they are ambiguous and incapable of specifying all types of functional operations. The verification framework of semiformal languages also requires a great deal of human interaction. This limits the degree of automation of semiformal verification.

The proposed semiformal approach offsets these limitations and is well balanced in its scalability and exhaustiveness. In terms of the disadvantage of existing formal functional behavior descriptions, we propose the event-driven flow graph-based specification, where functional behaviors are ordered in a timing-accurate manner, so that the cycle-accurate functional behaviors can be described without the exploration of whole state space. In terms of the disadvantages of simulation, the proposed ATPG approach with an intelligent redundancy-reduction strategy can enhance the quality of random vectors and target interesting corner test cases with limited human intervention. Moreover, functional coverage based on the proposed graph-based specification is proposed to realize a reliable measure for simulation verification.

3. Graph-Based Specification

The functionality of the design is composed of separate function points. Each function point is defined as a DUV showing the corresponding actions in the specific scenarios. The specific scenarios are the triggers of the function point and are referred to as "condition." The corresponding actions are referred to as "reaction."

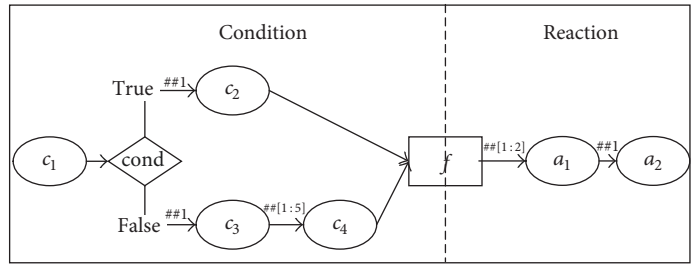

Figure 1: An example of our proposed EDFG.

Each function point of the design can be modeled with the help of our proposed event-driven flow graph (EDFG). The proposed graph-based specification (GBS) is the formal specification of EDFGs. In Section 3.1, we will detail the proposed EDFG and how a function point is described by EDFG. The proposed GBS is introduced in Section 3.2.

3.1. Event-Driven Flow Graph. The functionality of the design is presented by a set of variables. Among the variables, there are input and output parameters of the design and some local variables indicating inner register of the design. All predicates and functions mentioned below are defined over that set of variables.

The structure of EDFG is described by a directed graph where three kinds of nodes are admitted: *event nodes*, *control nodes*, and *function-point node*. Each event node is used for specifying the operation by the set of variables. Control nodes specify control flow branching, and the function-point node is used for specifying the function point and separating the condition and reaction of the function point in EDFG. Nodes in EDFG are connected and ordered by directed edges. Each directed edge has a weight which is used to mark the accurate cycle interval between joint nodes.

Next, we will further discuss the definition of event nodes and control nodes. Each event node describes an operation of the design which is described by $\{E\}U$, where E is an enabling predicate function and U is an update function. For an event node, if the enabling predicate function E is satisfied, the update function U is executed. If the enabling predicate function E is false, it indicates that the event node is interlocked and it will be unlocked when E is true. Control nodes specify control flow branching, which is described by branching predicate function C. When moving to a control node, the branching predicate function C of the control node is estimated and choice among two alternative outgoing edges is made.

An example of our proposed EDFG describing a function point f is shown in Figure 1. The ellipse nodes, the diamond-shaped node, and the square node represent the event nodes, the control node, and the function-point node, respectively. The left part of the graph is the condition of function point f, which is used to restrict situations in which f is activated. c_1, c_2, c_3, and c_4 are event nodes of the condition, and cond is a control node having two outgoing edges (marked by true and false). The right part of the graph describes the reaction of function point f; the resultant action after f is activated. a_1 and a_2 are the event nodes of reaction. The weight of the directed edge marks the accurate cycle interval between the two joint nodes. The cycle interval is described as "## num" with num being a specific number or a range between two numbers (described as "[number_1 : number_2]").

If the in-degree of an event node is equal to zero, the node is called an *initial node*. The path from the initial node to the function-point node is called a *condition path* which presents a specific situation of the function point. If the out-degree of an event node is equal to zero, the node is called a *final node*. The path from the function-point node to the final node is called a *reaction path* and presents the specific reactions of the function point. The path from the initial node to the final node is called a *function-point path* which presents the reactions corresponding to the specific conditions of the function point. Next, we will discuss the proposed graph-based specification where the base definitions of the EDFG are given.

3.2. Base Definitions of GBS. Hereinafter, we suppose that the port parameters or local variables can be unified by the general term *variables*. In the following definitions, we suppose that each variable v is associated with a set of possible values D_v, which is called the domain of variable v. If V is a set of variables, then D_V denotes a set of possible valuations of variables from set V.

Definition 1. An event-driven flow graph is a 4-tuple $\langle P, V, N, E\rangle$, consisting of the following:

(i) P is the set of port parameters and consists of the set of input port parameters X and the set of output port parameters Y, but the clock port is not contained by P. For a port $p \in P$, p is a tuple $\langle$id, width, direction, type$\rangle$, consisting of

  (a) id: the name of p in the DUV;
  (b) width: the width of p in the DUV;
  (c) direction: the direction of p, for each port p, direction $\in$ {input, output};
  (d) *type*: the data type of p. For each port p, type $\in$ {bit, int, long int, short int, float, double float}.

(ii) V is the set of local variables. For a variable $v \in V$, the v is a tuple $\langle$id, type$\rangle$, consisting of

  (a) id: the name of v;
  (b) type: the data type of v, for each variable, type $\in$ {bit, int, long int, short int, float, double float}.

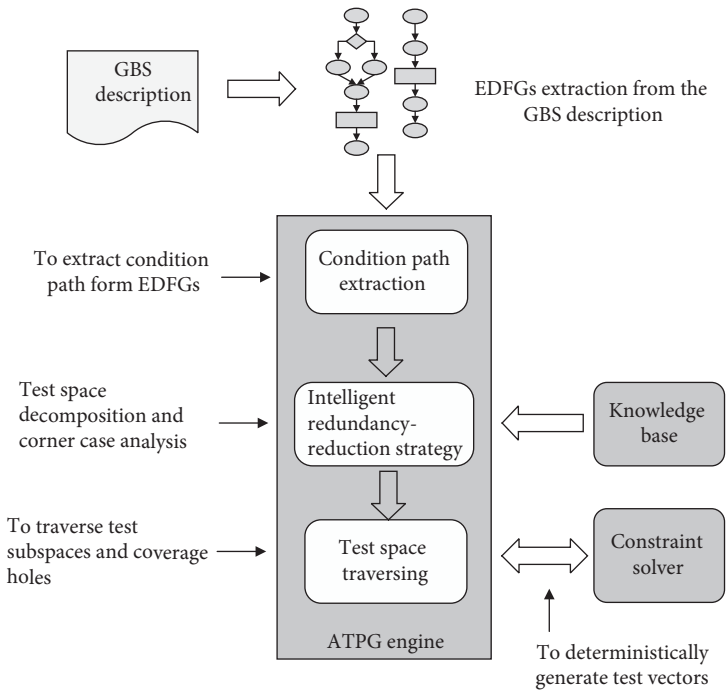

Figure 2: The proposed functional ATPG flow.

(iii) N is the set of nodes and consists of the set of event nodes EN, the set of control nodes CN, and a function-point node fn.

For an event node $en \in EN$, en is a tuple $\langle \alpha_{en}, \nu_{en} \rangle$, consisting of

(a) α_{en}: $D_P \times D_V \rightarrow \{\text{true}, \text{false}\}$: the enabling predicate function of the event node;
(b) ν_{en}: $D_P \times D_V \rightarrow D_V$: the update function of the event node.

For a control node cn, cn is specified by β_{cn}, which is

(a) β_{cn}: $D_P \times D_V \rightarrow \{\text{true}, \text{false}\}$: the branching predicate function of the control node.

A function-point node fn is used for specifying the identification of a function point and it can separate the condition path and reaction path from the graph.

(iv) E is the set of edges. For an edge $e \in E$, e is a tuple $\langle \text{Cycle}_e, n_e, n'_e \rangle$, consisting of

(a) Cycle_e: the time intervals from the initial node of the edge to the final node of the edge;
(b) n_e: the head node of the edge;
(c) n'_e: the tail node of the edge.

Definition 2. Assuming an EDFG $G(P, V, N, E)$, a pair $\langle n, d_V \rangle \in N \times D_V$ is called a *configuration* of G. The *reset configuration* for G is denoted as $\langle n_0, d_{V0} \rangle$, where n_0 is the reset node and d_{V0} is the vector of the reset values of the local variables V.

A function point that operates on the corresponding EDFG $G(P, V, N, E)$ is defined in this way. In the beginning, G is in the reset configuration $\langle n_0, d_{V0} \rangle$. G receives an input $x \in X$; it moves to the next configuration $\langle t, y \rangle$, iff

$$\begin{aligned} &\alpha_t = \text{true}, \\ &\exists e \in E, \\ &n_e = n_0, \\ &n'_e = t. \end{aligned} \tag{1}$$

Then the update function ν_t is executed. G continues to receive the input vectors and finally moves to the configuration $\langle fn, z \rangle$, where fn is the function node. This means that a *condition path* of G has been traversed. Then, G sends an output $y \in Y$; it moves to the next configuration $\langle h, w \rangle$ iff

$$\begin{aligned} &\alpha_h = \text{true}, \\ &\exists e \in E, \\ &n_e = f, \\ &n'_e = h. \end{aligned} \tag{2}$$

Finally, G moves to the configuration $\langle l, k \rangle$, where l is the *final node*. This means that the reaction path of G is traversed and the operation of the function point is finished.

Based on the above definitions, the functional behaviors of DUV can be formally described by the graph-based specification. Next, we will discuss the method of the test vector generation based on the specification.

4. Functional ATPG Based on GBS

We propose a functional ATPG approach to generate high-quality test vectors based on the GBS descriptions. Figure 2

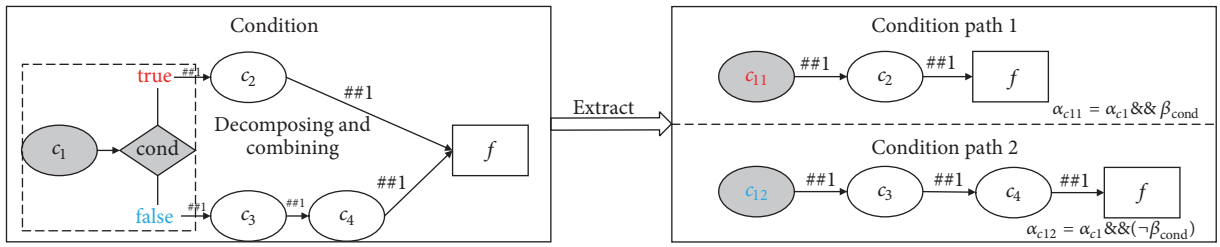

Figure 3: An example of condition path extraction. (a) The condition of the EDFG in Figure 1. (b) The condition paths extracted from the condition shown in (a).

shows the proposed ATPG process. Firstly, the EDFGs are extracted from the specification and loaded into the ATPG engine. The ATPG engine analyzes the EDFGs and extracts all condition paths from each of the EDFGs. An intelligent redundancy-reduction strategy is then applied to decompose the test space and to target corner cases for each condition path. This helps limit the disadvantages of random vectors and ensures an exhaustive and effective exploration of the test space for each condition path. Finally, the ATPG engine traverses all the test subspaces and corner cases to generate the test vectors with the help of the constraint solver. The knowledge base stores expert experiments and rules to guide the intelligent redundancy-reduction strategy. Next, we will discuss the key issues involved in the proposed ATPG engine.

4.1. Condition Path Extraction. The first phase of the ATPG flow is condition path extraction, which is used to extract all independent condition paths from the condition of each EDFGs. Generally, an EDFG has more than one condition path due to the presence of the control nodes. Due to the two outgoing edges of the control node, each control node will double the condition paths. Each condition path in the EDFG presents a specific test situation, so all the condition paths should be extracted.

The process of condition path extraction is based on the control nodes in the EDFG. For a control node, there are two condition paths along with the two outgoing edges of the control node. Two condition paths are extracted with modified event nodes. The two modified event nodes are generated in two steps: (1) duplicating the enabling predicate function of the previous event node and (2) *and-ing* the duplicated predicate function with the branching predicate and *not-ing* branching predicate of the control node, respectively.

An example of condition path extraction is shown in Figure 3. Figure 3(a) presents the condition of the function point f shown in Figure 1. Due to the existence of control node *cond*, there are two condition paths extracted from the condition, which is shown in Figure 3(b). For condition path 1, the enabling function of event node c_{11} is the *and-ing* of the previous event nodes c_1's enabling predicate function α_{c1} and the branching predicate (β_{cond}) of the control node cond. In the same sense, the enabling function of event node c_{12} is the *and-ing* of the c_1's enabling function α_{c1} and the branching predicate ($\neg\beta_{cond}$) in condition path 2.

4.2. Intelligent Redundancy-Reduction Strategy. Next, we will discuss how to generate test vectors for an exhaustive exploration of the test space for each condition path. The test space of a condition path is composed of test space of each event node in the condition path. The test space of an event node is comprised of the value ranges of all input parameters, which cause the enabling predicate function of the event node to be *true*. The constraint solver is used for solving the test space of each event node in a condition path and generates test vectors to traverse the path.

In general terms, the test space of a condition path is very large. It is necessary to traverse the same condition path many times to ensure an exhaustive exploration of the test space. However, the test vectors generated by the constraint solver are random. Due to the redundancy and uncertainty in random stimulus, it does not ensure complete coverage of the test space and activation of the corner cases with a considerable quantity of stimuli. So a strategy to decrease the redundancy and uncertainty in random stimulus is necessary.

The value range of each input parameter, which makes an event node's enabling predicate function *true,* is called the *legal value range*. Because the event node's test space is comprised of the legal value ranges of all input parameters, the property and data type of each of the input ports are concerned with random vector generation strategy. Except for the clock port, the properties of the input ports can be categorized as *data-path* or *control-path*. In general terms, the data-path port is concerned with communication and calculation and its long bit-width leads to a large legal value range. A certain number of random stimuli cannot ensure that the test space and corner cases are covered. The control-path port relates to functional mode change. Although the legal value range of the control-path port may be small, random stimuli cannot ensure that each legal value is covered.

Due to the above analysis, we can see that the uncertainty and redundancy of random stimuli cause incomplete coverage of large legal value range for data-path ports and corner cases. So we adopt two approaches to reduce the redundancy: (i) test-space decomposition and (ii) corner-case analysis.

Test-space decomposition aims at the incomplete coverage of large legal value range for data-path ports, which involves decomposing the test space into multiple subspaces and traversing them separately. The mathematical derivation can prove the efficiency of the approach. If a test space's size

is n, the expected number of the random vectors needed to cover the test space T is

$$T = O\left(n \bullet \log\left(n\right)\right). \tag{3}$$

We assume that one cut is made to decompose a test space into two average subspaces. After k cuts, the expected number of random vectors needed to cover the test space $T(k)$ is

$$T\left(k\right) = O\left(n \bullet \log\left(n - k\right)\right). \tag{4}$$

In general terms, especially for the data-path port, the size of a test space is extremely large. So decomposition of the test space can decrease some redundant random vectors.

Another approach is corner-case analysis, which is used for inferring corner cases based on the knowledge of each port. Generally, corner cases are combinations of all legal values of different control-path ports and all corner values of different data-path ports. For example, the test space of an event node is comprised of the legal value ranges of data-path port A and control-path port B. The data type of A is a double float and the legal value range of B is from *3'b000* to *3'b100*. Usually, the corner cases of the test space are mainly concentrated on combinations of each corner value of A and each legal value of B. Actually, the corner value of A can be inferred by its data type based on expert knowledge. The corner-case analysis can be guided by a knowledge base and generate corner test case automatically.

We propose an intelligent redundancy-reduction strategy that adopts the abovementioned approaches. It involves automatically decomposing the test space and targeting the corner cases. The intelligent strategy integrates an expert knowledge base to reduce the human labor involved. This expert knowledge can infer the corner values and guide the test-space decomposition based on the property and data type of each port. The intelligent redundancy-reduction strategy involves the following steps:

(1) For a condition path, setting each port involved in the condition path as a data-path port or a control-path port. This step is guided by a verification engineer or the expert knowledge base.

(2) Decomposing the test space of the condition path into test subspaces.

  (a) Decomposing the test space of each event node into multiple test subspaces.

    (i) Decomposing the legal value range of each data-path port into legal value subranges evenly. The number of subranges can be set by a verification engineer or the expert knowledge base.

    (ii) Forming the test subspaces of each event node, which are combinations of the legal value subranges of all data-path ports and the legal value ranges of all control-path ports.

  (b) Forming the test subspaces of the condition path, which are combinations of the test subspaces of all event nodes.

(3) Targeting the corner cases for the condition path.

  (a) Targeting specific cases for each event node in the condition path. Specific cases are combinations of all data-path ports' corner values and all control-path ports' legal values. The corner values of the data-path ports can be inferred from the data types according to the knowledge base.

  (b) Combining specific cases of all the event nodes and forming corner cases of the condition path.

The proposed intelligent redundancy-reduction strategy is illustrated in Figure 4. The condition path has two event nodes, $C1$ and $C2$, and a function node f. Ports $P1$, $P2$, $P3$, $P4$, and $P5$ are the ports in the condition path. $P1$, $P2$, and $P4$ are the data-path ports, and $P3$ and $P5$ are the control-path ports. The test space of $C1$ consists of the legal value space of $P1$, $P2$, and $P3$, and the test space of $C2$ consists of the legal value space of $P4$ and $P5$.

Section (a) in Figure 4 shows the test-space composition of the condition path. Firstly, the legal value range of the data-path ports is divided into subranges in the event node $C1$ and event node $C2$. For example, the legal value range of $P1$ is divided into $P1_a$, $P1_b$, $P1_c$, and $P1_d$. The subranges of each data-path port and the legal value ranges of each control-path ports are then combined to create the test subspace of the condition path. Each combination of subranges presents a test subspace of the condition path.

Section (b) in Figure 4 shows the corner-case analysis. Firstly, the corner value of each data-path port is inferred by the knowledge base. In Figure 4, $P1_1$ and $P1_2$ are the corner values of $P1$, and other data-path ports are similar to $P1$. Meanwhile, $P3_2$ and $P3_3$ are the legal values of $P3$. $P5$ is similar to $P3$. The corner values of each data-path port and the legal values of each control-path port are then combined to create the corner cases of the condition path. Each combination presents a corner case of the condition path.

4.3. Traversing the Test Space. This phase involves traversing and solving all the test subspaces and corner cases of all the condition paths generated in the second phase. The test subspaces should be traversed repeatedly to ensure an exhaustive exploration of the test space. The required traversal number of each test subspace can be set by a verification engineer or the knowledge base.

5. Implementation and Experiment Results

5.1. Implementation. We developed a specification-based verification system for the proposed semiformal approach. The verification system receives the graph-based specification and automatically verifies whether the DUV functionality conforms to its specification. For this purpose, the following tasks are implemented in the verification system:

(1) The generation of test sequences for exhaustive exploration of the test space based on the proposed intelligent redundancy-reduction strategy

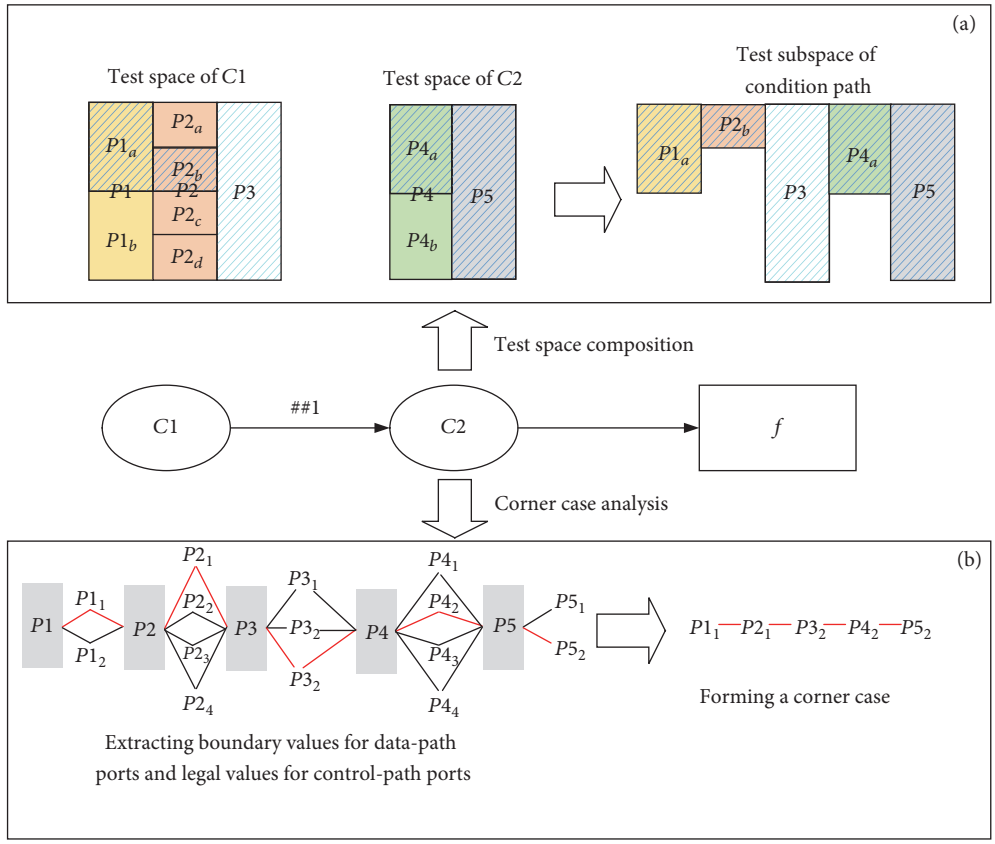

FIGURE 4: An example of the intelligent redundancy-reduction strategy. (a) The example of test-space decomposition strategy. (b) The example of corner-case analysis strategy.

(2) Making a connection between the verification system and the DUV

(3) Verification of the functional behaviors of the DUV in response to the given GBS descriptions

(4) Recording the simulation reports and coverage reports.

The whole semiformal verification system, involving the GBS test system and simulator, is implemented in C language and SystemVerilog language. The semiformal verification flow is shown in Figure 5.

Firstly, the function points of the DUV are extracted from the textual specification (SPEC), and modeling GUI helps the engineer to model the function points in GBS. The GBS descriptions are parsed into EDFGs, which are stored in the EDFG's data register.

The simulation is then launched. The testbench includes the DUV environment, the connector module, and parts of the GBS test system. The DUV environment contains a DUV instance. It initializes the instance, launches communication to the GBS test system through the connector module, receives test vectors from the GBS test system, and sends the DUV's behaviors to the GBS test system. The connector module employs the SystemVerilog Direct Programming Interface (DPI) to enable communication between the GBS test system and simulator. The parts of the GBS test system are the connector, the EDFG's data register, the constraint solver, and the test trace recorder. The EDFG's data register synchronizes the interface port parameters with the DUV's signal from the connector, checks compliance of the DUV's behaviors with the specified reaction in the EDFGs, and sends the corresponding constraints to the constraint solver according to timing. The constraint solver then solves the constraints and sends the test vectors to the DUV via the connector. The ATPG engine is integrated into the EDFG's data register and the constraint solver. The test trace recorder records the traversed event nodes and edges to guide test vector generation.

Finally, the simulation report, VCD File, and GBS report are generated after the test is completed. The simulation report records the code coverage and simulated test vectors. The VCD File records waveforms for the self-checking test. The GBS report records the GBS coverage. GBS coverage involves three kinds of coverage: *event-node coverage*, *conditional coverage*, and *edge coverage*. The event-node coverage is the percentage of the traversed event nodes in the EDFGs. The conditional coverage is the percentage of the traversed outgoing edges of the condition nodes in the EDFGs. The edge coverage is the percentage of the traversed edges in the EDFGs. GBS coverage supports a reliable measure for the functional coverage of DUV by input vectors.

5.2. Case Studies. For evidencing the scalability and efficiency of the GBS and the proposed semiformal verification

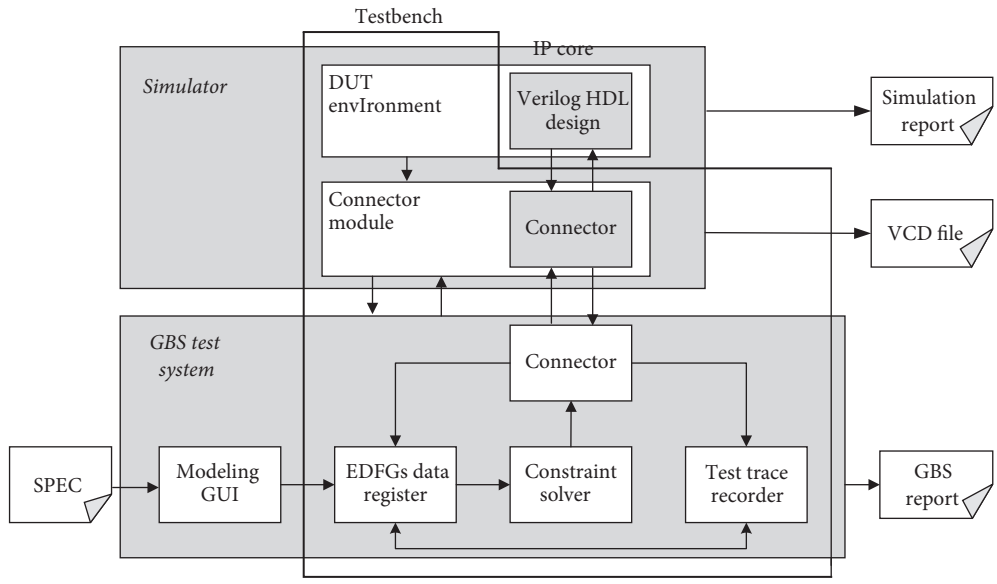

Figure 5: Diagram of the proposed semiformal verification platform.

Table 1: Characteristics of the DUVs.

Design	Characteristics of design			
	LoC	PI	Gates	Flip-flops
FPU	1496	134	105909	2870
MAC	1628	198	137125	3105
DMA	9523	3422	327931	4950
XMC	8187	3285	512987	6921

Table 2: Characteristics of GBS descriptions.

Design	Characteristics of GBS descriptions	
	Function point	Node
FPU	18	4.0
MAC	26	5.7
DMA	56	11.5
XMC	47	11.7

flow, we have selected two medium-scale arithmetic units and two large-scale control units for experimentation. The arithmetic unit is comprised of a floating-point unit (FPU) [13] and an integer multiply-add unit (MAC). The control logic unit is a data management access (DMA) and a controller between L2-cache and memory (XMC). The FPU is an open source IP core and others are in-house designs.

The four designs are all implemented by Verilog. The characteristics of the DUVs are shown in Table 1. Column *LoC* shows the number of lines of code of the DUVs, column *PI* reports the number of bits in the primary inputs, and *Gates* and *Flip-flops,* respectively, present the number of gates and flip-flops.

Firstly, the function points of the DUVs are described in the GBS by the modeling GUI. The characteristics of the GBS descriptions are shown in Table 2. Column *Function point* shows the number of the function points of the DUVs and column *Node* reports the average number of nodes in the function points of the DUVs. It is clear from the column *Node* that the complexity of the two control-sensitive DUVs is much more than the two arithmetic-sensitive DUVs.

We then adapted the proposed semiformal verification method to verify the correctness of the DUVs. We also implemented the verification platforms adopting the constrained randomized test (CRT) for the four DUVs. The test scenarios created in the CRT verification platform correspond to the function points in the GBS descriptions and the self-checking test is adopted in the CRT verification platform. For the purpose of presenting the quality of the test vectors generated in the GBS verification platform, the GBS verification platform and the CRT verification platform are simulated by NC Sim in Cadence Design Systems.

The code coverage comparisons for the four DUVs are shown in Table 3. The table shows the block coverage (BC), the expression coverage (EC), the toggle coverage (TC), and the FSM coverage (FC). Column *vector* shows the number of test vectors.

Based on the results of the code coverage, the procedural convergence of our proposal is faster than the CRT. This is especially so for large-scale designs such as DAM and XMC. This indicates that the test vectors generated by the proposed verification platform are more effective for large-scale control-sensitive designs. This can be explained as that the redundancy in the random vectors increases with the complexity of the function points and the increasing number of ports. So the effect of the intelligent redundancy-reduction strategy is outstanding. The experiment indicates that the proposed ATPG approach adopted in the verification platform ensures an effective exploration of the functionality of DUVs.

The results of the GBS coverage are shown in Table 4. The table shows the event-node coverage (NC), conditional

Table 3: Code coverage comparison between the semiformal method and CRT method.

Design	Vector	Code coverage					
		The semiformal method			CRT method		
		BC%	EC%	TC%/FC%	BC%	EC%	TC%/FC%
FPU	100	62.4	44.2	74.3/-	57.8	40.8	68.2/-
	400	79.6	67.9	91.2/-	68.5	58.2	84.0/-
	900	95.6	79.5	95.1/-	89.4	74.7	92.4/-
MAC	100	69.7	56.4	72.8/-	64.7	51.7	70.1/-
	400	82.1	77.2	89.6/-	78.2	69.3	83.2/-
	900	97.0	86.8	92.1/-	95.1	84.1	88.7/-
DMA	900	87.6	85.7	-/92.4	75.1	70.4	-/89.8
	2100	92.8	92.3	-/95.1	85.9	89.5	-/94.9
	4800	96.2	100	-/100	96.2	98.7	-/100
XMC	900	90.8	86.2	-/93.5	86.2	81.5	-/91.4
	2100	94.7	96.4	-/96.2	92.0	93.9	-/95.7
	4800	97.3	100	-/100	97.3	100	-/100

Table 4: GBS coverage results.

Design	Vector	GBS coverage		
		NC%	CC%	EDC%
FPU	100	76.8	100	69.3
	400	100	100	100
	900	100	100	100
MAC	100	83.2	100	74.6
	400	100	100	100
	900	100	100	100
DMA	900	67.2	74.6	63.2
	2100	88.0	91.4	89.1
	4800	95.9	96.3	96.3
XMC	900	71.3	81.6	64.8
	2100	91.4	98.5	89.4
	4800	97.1	98.5	98.5

coverage (CC), and edge coverage (EDC) in GBS coverage. Column *vector* shows the number of test vectors. The GBS coverage results for arithmetic-sensitive units and control-sensitive units are entirely different.

For the arithmetic units, the GBS coverage convergence is relatively fast. The complexity of the function points is higher for large-scale control-sensitive designs; therefore, GBS coverage convergence is slower. It is clear that the GBS coverage for DMA and XMC does not achieve 100% in the situation of code coverage convergence. Based on the GBS coverage holes, we find some individual special cases that cannot be activated in a certain number of test vectors, and we found one functional flaw for DMA and two functional flaws for XMC. It indicates that the GBS coverage supports a more reliable measure of the verification process than code coverage. It can better reflect the completeness and exhaustiveness of the simulation. Meanwhile, GBS coverage results indicate that the proposed semiformal verification method and functional ATPG method ensure an exhaustive exploration of the functionality of DUVs.

6. Conclusion

In this paper, we proposed a semiformal approach for hardware design functional verification, involving (i) an event-driven flow graph to present the functional behaviors of designs and a graph-based specification to formally describe the graph, (ii) a functional ATPG approach involving an intelligent redundancy-reduction strategy to generate high-quality test sequences based on given GBS descriptions, and (iii) a functional coverage based on the formal specification to support a more reliable measure of completeness and exhaustiveness in verification. We developed a verification platform to verify the functionality of the four DUVs. The results of the experiment show that the proposed semiformal approach can be applied to the large-scale complex designs, so it avoids state explosion in formal approach. Meanwhile, compared with CRT simulation method, it ensures a more exhaustive and effective exploration of the functional correctness of DUVs.

Future work should focus on improving the functional ATPG approach based on GBS, specifically (i) expansion of

knowledge base to further reduce human intervention in the process and (ii) studying and developing a backjumping algorithm to traverse the hard-to-traverse edges in the EDFGs.

Conflicts of Interest

The authors declare that they have no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research was supported by The National Natural Science Foundation of China (Grant no. 61402493) and The Major State Basic Research Development Program of China (Grant no. 1800067117008).

References

[1] H. D. Foster, "Trends in functional verification," in *Proceedings of the the 52nd Annual Design Automation Conference*, pp. 1–6, San Francisco, Calif, USA, June 2015.

[2] S. Saponara, L. Fanucci, and M. Coppola, "Design and coverage-driven verification of a novel network-interface IP macrocell for network-on-chip interconnects," *Microprocessors and Microsystems*, vol. 35, no. 6, pp. 579–592, 2011.

[3] Y. Naveh, M. Rimon, I. Jaeger et al., "Constraint-based random stimuli generation for hardware verification," *AI Magazine*, vol. 28, pp. 13–305, 2007.

[4] A. M. Gharehbaghi, B. H. Yaran, S. Hessabi, and M. Goudarzi, "An assertion-based verification methodology for system-level design," *Computers and Electrical Engineering*, vol. 33, no. 4, pp. 269–284, 2007.

[5] R. Jones, J. OLeary, and C.-J. H. Seger, "Practical Formal Verification in microprocessor design," *IEEE Design & Test of Computers*, vol. 18, no. 4, pp. 16–25, 2001.

[6] J. Bhadra, M. S. Abadir, L.-C. Wang, and S. Ray, "A survey of hybrid techniques for functional verification," *IEEE Design & Test of Computers*, vol. 24, no. 2, pp. 112–122, 2007.

[7] O. S. C. Initiative, "Functional specification for SystemC 2.0," *Outubro*, 2001.

[8] F. Ghenassia, *Transaction-level modeling with SystemC*, Springer, Dordrecht, The Netherlands, 2005.

[9] A. Habibi and S. Tahar, "Design and verification of SystemC transaction-level models," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 1, pp. 57–68, 2006.

[10] S. Suhaib, D. Mathaikutty, and S. Shukla, "System level design methodology for system on chips using multi-threaded graphs," in *Proceedings of the 2005 IEEE International SOC Conference*, pp. 133–136, Herndon, VA, USA, September 2005.

[11] M. Oliveira and A. Hu, "High-level specification and automatic generation of IP interface monitors," in *Proceedings of 39th Design Automation Conference*, pp. 129–134, New Orleans, LA, USA, June 2002.

[12] L. Zhang, I. Ghosh, and M. Hsiao, "Efficient Sequential ATPG for Functional RTL Circuits," in *Proceedings of the Proceedings International Test Conference 2003*, pp. 290–298, October 2003.

[13] G. Di Guglielmo, F. Fummi, G. Pravadelli, S. Soffia, and M. Roveri, "Semi-formal functional verification by EFSM traversing via NuSMV," in *Proceedings of the 2010 15th IEEE International High Level Design Validation and Test Workshop, HLDVT'10*, pp. 58–65, June 2010.

[14] G. D. Guglielmo, L. D. Guglielmo, F. Fummi, and G. Pravadelli, "Efficient generation of stimuli for functional verification by backjumping across extended FSMs," *Journal of Electronic Testing*, vol. 27, no. 2, pp. 137–162, 2011.

[15] K.-T. Cheng and A. S. Krishnakumar, "Automatic generation of functionial vectors using the extended finite state machine model," *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, no. 1, pp. 57–79, 1996.

[16] I. Ghosh and M. Fujita, "Automatic test pattern generation for functional register-transfer level circuits using assignment decision diagrams," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 3, pp. 402–415, 2001.

[17] L. Lingappan, S. Ravi, and N. K. Jha, "Test generation for non-separable RTL controller-datapath circuits using a satisfiability based approach," in *Proceedings of the Proceedings: 21st International Conference on Computer Design ICCD 2003*, pp. 187–193, October 2003.

[18] F. Ferrandi, F. Fummi, and D. Sciuto, "Implicit test generation for behavioral VHDL models," in *Proceedings of the 1998 IEEE International Test Conference*, pp. 587–596, October 1998.

[19] M. Riebisch and M. Hubner, "Traceability-Driven Model Refinement for Test Case Generation," in *Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pp. 113–120, Greenbelt, MD, USA.

[20] R. Drechsler, M. Soeken, and R. Wille, "Formal specification level: Towards verification-driven design based on natural language processing," in *Proceeding of the 2012 Forum on Specification and Design Languages (FDL)*, pp. 53–58, IEEE, 2012.

[21] J. Tretmans and E. Brinksma, *Automated Model-Based Testing*, 2003.

[22] C. M. Kirchsteiger, J. Grinschgl, C. Trummer et al., "Automatic test generation from semi-formal specifications for functional verification of system-on-chip designs," in *Proceedings of the 2008 2nd Annual IEEE Systems Conference*, pp. 1–8, IEEE, 2008.