

# Chapter 11

## Test Coverage



**Masahiro Fujita, Koichiro Takayama, Takeshi Matsumoto,  
Kosuke Oshima, Satoshi Jo, Michiko Inoue, Tomokazu Yoneda  
and Yuta Yamato**

**Abstract** Verification is a process to prove the correctness of the design of a system referring the design information to requirements specification, and test is a process to prove that a system in its actual embodiment in either prototype or real product performs up to the description of the specification. Whatever functions, performance, or dependability may have been conceived, designed, and built into a system, one can be certain that the actual product exhibits such properties only to the extent that the design has been verified and the product has been tested. In reality, comprehensive coverage of verification and test over ramified combination of functionalities, use cases, and operational conditions becomes increasingly more difficult as systems become more complex. The verification and test are thus very important for assuring system's quality. This chapter addresses some of the key issues of verification and test of electronic systems that use VLSIs as essential components with an emphasis on dependability. Section 11.1 is an overview of the issues and discusses the metrics of verification and test coverage. Section 11.2 addresses two topics: detection of errors in logic design and formal verification, the latter being a method to verify logic design by mathematical reasoning. Section 11.3 introduces the use of Built-in Self-Test (BIST) method to monitor circuit delays in a VLSI precisely enough to be able to predict failures due to device degradation in operation. Section 11.4 proposes a way to accurately measure the delays in the presence of temperature and voltage variation experienced in the field.

---

M. Fujita (✉) · S. Jo  
The University of Tokyo, Tokyo, Japan  
e-mail: fujita@ee.t.u-tokyo.ac.jp

K. Takayama  
Fujitsu Ltd., Kawasaki, Japan

T. Matsumoto  
Ishikawa National College of Technology, Tsubata, Ishikawa, Japan

K. Oshima  
Hitachi, Ltd., Tokyo, Japan

M. Inoue · T. Yoneda · Y. Yamato  
Nara Institute of Science and Technology, Ikoma, Japan

**Keywords** Test coverage • Formal verification • Debugging  
Circuit delay • BIST • Field test

## 11.1 Verification and Test Coverage

Koichiro Takayama, Fujitsu Ltd.

Coverage metrics are widely used in the verification and validation of both hardware and software to show the progress as well as the goal of the verification and validation process.

Some of the dependable design techniques described in this book are applied to logic circuits in a system. In order to achieve the intended dependability of the system, it is necessary to make sure that the circuits operate correctly.

The objective of the verification is to make a design bug free, but as far as we know, there are no coverage metrics to satisfy this objective such that by achieving 100% coverage it guarantees the design works perfectly.

In this article, topics of coverage metrics in verification and test are described.

### 11.1.1 Verification Coverage Metrics

In this section, coverage metrics for logic verification are described.

Coverage metrics provide aspects to express if the logic in the design under verification (DUV) is activated in logic simulation or not.

Coverage metrics are categorized into three as follows;

#### (1) Code coverage

This is one of the most basic metrics to show which parts of the source code of the DUV written in a hardware description language are covered. Most of the logic simulators can measure the coverage automatically during simulation of the DUV. Major code coverage metrics are as follows.

- Line coverage: the fraction of lines of source code which are executed.
- Condition coverage: the fraction of subexpressions of Boolean expressions which are evaluated to true and false.
- FSM coverage: the fraction of the states of a finite state machine (FSM) which are visited.

Code coverage does not guarantee that the DUV is totally correct since code coverage neither uncovers unimplemented features, nor measures concurrent or temporal behaviors of the DUV.

## (2) Functional coverage based on the design implementation

This metric is manually derived by a designer by focusing on the implementation-dependent features of the DUV, for example, if the read or write operation took place when a FIFO is empty or full, it is checked whether a pipeline is stalled, or a request is acknowledged within five cycles. It depends very much on the designer's experience and attention how high the resulting verification coverage will be.

## (3) Functional coverage based on the design specification

This metric is manually derived from the specification of the DUV by a designer. The derived behavior is concurrent or temporal but independent of the implementation. In many cases, the metric is written in SystemVerilog Assertions (SVA) [1] or PSL [2]. When the derived behavior is too difficult to write in SVA or PSL, a coverage model is written in a hardware description language and the coverage metrics [1] and/or [2] are applied to the model. The coverage will again depend pretty much on the designer's skills.

### 11.1.1.1 Coverage Metrics with High Correlation to Design Bugs

As described above, coverage metrics are used to define the verification goal. Importantly, it should be noted that more design bugs can be found by achieving 100% of a coverage metric and using better metric. From this point of view, it may be useful to make a new metric by analyzing the design bugs experienced in the past.

In this section, we illustrate an example of metric relating to a bug of incorrect priority of the branch condition. In Fig. 11.1, code A shows a code fragment with an incorrect order of the branch conditions while code B shows the correct one where the condition C should have a higher priority than condition B.

When a set of four vectors (condition A, condition B, condition C) = (true, false, false), (false, true, false), (false, false, true), (false, false, false) is applied to code A and code B, the line coverage for both code is 100%, but the bug is uncovered.

A vector (false, true, true) can uncover the bug. And, in order to verify the priority of all of the combinations of two conditions among three, a set of vector

<pre>if ( conditionA ) statementA ; else if ( <u>conditionB</u> ) statementB ; else if ( <u>conditionC</u> ) statementC ; else statementD ;</pre>	<pre>if ( conditionA ) statementA ; else if ( <u>conditionC</u> ) statementC ; else if ( <u>conditionB</u> ) statementB ; else statemetnD ;</pre>
(a) codeA : incorrect design	(b) codeB : correct design

**Fig. 11.1** Example of a design bug

(condition A, condition B, condition C) = (true, true, false), (true, false, true), (false, true, true), (false, false, true), (false, false, false) is required.

As described in this section, if a pattern in the structure of the source code (design implementation) can be identified by analyzing the bug, a coverage metric can be derived which contributes to the design quality effectively.

### **11.1.1.2 Cooperation of Logic Simulation and Formal Verification**

Formal verification is a technique to prove mathematically whether a design satisfies a property (specification). If a design does not satisfy a property, a counterexample, i.e., an input vector sequence which violates the property, is generated.

If the negation of a coverage metric can be written in a set of properties, by formally verifying the properties against the DUV, part of the coverage space can be eliminated when corresponding property is proved to be redundant, or an input vector sequence for a corner case can be generated as a counterexample. Such cooperation between logic simulation and formal verification has been widely used.

## ***11.1.2 Test Coverage***

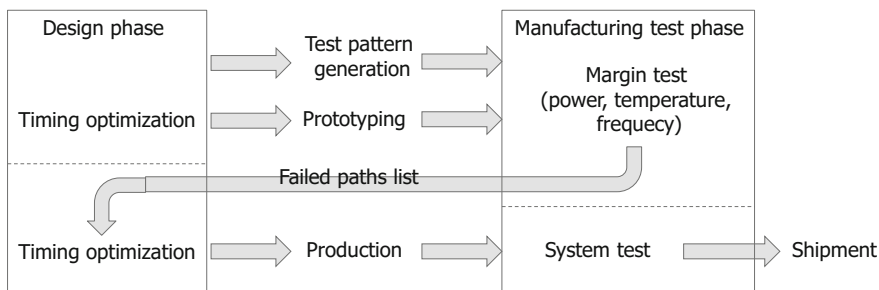
In this section, issues in manufacturing test to validate dependability of the computer system are discussed. The major causes of malfunction of the system in the field are timing errors by aging and occasional soft error by radiation events such as collision of cosmic rays or alpha particles.

### **11.1.2.1 Test Coverage for Soft-Error Resilience**

Soft-error resilience is one of the main topics of dependability described in Chap. 3. Many existing designs have adopted various techniques in order to recover from the intermittent 1-bit error, for example, error control coding (ECC) for data path, triple modular redundancy for control logic, soft-error tolerant flip-flops, hardware instruction retry. In order to validate the soft-error resilience circuitry after fabrication, the hardware might have a feature to inject a pseudo 1-bit error. It is another coverage problem what additional logic is required to validate soft-error resiliency efficiently.

### **11.1.2.2 Test Coverage for Timing Error**

Recent high-performance computing systems consist of more than tens of thousands processors. In order to reduce timing errors caused by aging in field, they



**Fig. 11.2** Example flow of a chip test

require appropriate timing optimization for critical paths at design phase and manufacturing test to validate critical paths while keeping a high signal active ratio.

Figure 11.2 shows an example of chip test flow from the view of timing error.

- At design phase, timing is optimized by taking signal integrity and power noise into account. Circuit simulation considering parasitic elements is useful to assess signal integrity affected by cross-talk, ringing, etc.
- At prototyping phase, engineering sample has been tested to extract paths with low margins.
- Timing is further optimized for the extracted paths.
- Manufacturing test is applied to eliminate chips with low margins caused by manufacturing variability.

There are two problems in the flow as follows.

- (1) At the design phase, static timing analysis (STA) is applied. When the latest device technology is used, calibration of the model parameters has to be done carefully.
- (2) Test pattern for critical paths extracted by STA should be generated with high signal active ratio to validate signal integrity and power noise effect.

### 11.1.3 Summary

In this article, we described the following challenges of coverage metrics:

- How a verification coverage metric can be addressed by taking the characteristics of design bug into account.
- How a test coverage metric can be addressed by taking signal integrity and power noise effect into account.

## 11.2 Design Errors and Formal Verification

Masahiro Fujita, The University of Tokyo

Takeshi Matsumoto, Ishikawa National College of Technology

Kosuke Oshima, Hitachi, Ltd.

Satoshi Jo, The University of Tokyo

### 11.2.1 Logic Design Debugging

Logic design verification and debugging is one of the most time-consuming tasks in VLSI design processes. Once incorrect behaviors are detected through simulation and/or formal verification process, the design must be logically debugged. Incorrect behaviors are represented in the form of counterexamples which are generated from simulation/formal verification. Counterexamples are the input/output patterns, where output patterns are different from the correct expected patterns inferred from the specification. In this section, we deal with logic debugging processes mostly targeting gate-level designs by analyzing counterexamples. We show through experiments that even with small numbers of counterexamples, complete logic debugging is feasible, which shows practical effectiveness of the proposed approach. Also, in order to make the proposed method powerful enough for practical designs in terms of logic debugging, we introduce a “necessary condition” for the selection of signals in Sect. 11.2.4.3, when correcting the logical bugs. This necessary condition takes a very important role to filter out non-useful signals for the selection and significantly improves the performance of the logical debugging.

Once counterexamples in the verification processes are generated, logic debugging processes must start. Logic design debugging consists of two phases. The first phase is to locate suspicious portions of the design by analyzing the internal behaviors of the buggy design with its counterexamples. The second phase is to actually correct those suspicious portions by replacing them with appropriate new circuits.

Path tracing and its generalization with SAT (satisfiability checking)-based formulation are the common and widely used approaches for the first phase. They can locate suspicious portions assuming that those may be replaced with new circuits whose inputs are possibly all primary inputs of the target circuit. These techniques are very briefly reviewed in Sect. 11.2.2. For details, please see [3].

We present a formulation based on LUT (Lookup Table) for the second phase of the logic debugging process. As a LUT can represent any logic functions with the given set of input variables, correction is guaranteed to succeed as long as the input variables of the LUTs are appropriately selected. We present the correction method as well as heuristic methods for selecting input variables of LUTs in Sect. 11.2.3.

The last subsection gives future perspectives on logic design debugging.

## 11.2.2 *Identification of Buggy Portions of Designs*

In general, a counterexample is an example run of the target buggy design, where some of the output values are different from the expected values inferred from the specification, that is, they are incorrect. The first phase of the logical debugging is to locate the cause of the bugs by analyzing a given set of counterexamples. It is realized by locating the suspicious internal signals, whose values are the cause of the wrong output values.

This analysis can be realized by tracing who are in charge of the incorrect output values, which is called “path tracing” in general. By tracing the functional dependencies in the logic circuits, the set of internal gates which determine the values of such outputs can be generated. They should include the root cause of the bugs in the sense that by modifying the functionality of those gates, the incorrect output values can become correct.

Path tracing methods are generalized using SAT-based formulation, called SAT-based diagnosis, in [3]. Please note that the methods such as path tracing and the ones in [3] examine the designs only with counterexamples given and do not refer to specifications. Also they guarantee correctability if the set of gates identified can be replaced with some appropriate logic functions with possibly all primary inputs. That is, we may need to identify the appropriate sets of inputs of the gates, which could be very different from the current sets of inputs, for correction. This is a critical issue for the correction part of debugging as shown below.

## 11.2.3 *Correction of Buggy Portions*

### 11.2.3.1 **Basic Idea**

In this paper, we focus on debugging gate-level designs. We assume existence of a specification in terms of golden models in RTL or in gate level. Our method tries to let a given circuit under debugging behave equivalently to the specification through modifications inside the circuit. That is, we need to identify the appropriate different functions for some of the internal gates for corrections. To achieve this, we introduce some amount of programmability with LUTs and MUXs in the circuit under debugging and find a way to program the introduced programmable circuits for the purpose of formulating the debugging processes mathematically. Please note that after identifying such appropriate functions for internal gates, those gates are assumed to be completely replaced with new gates corresponding to those functions. That is, programmability is introduced only for mathematical modeling and is nothing to do with actual implementations.

The basic idea of our proposed debugging methods is to correct a circuit under debugging by finding another logic function with the same set of inputs for each gate that is identified as a bug location, in such a way that the entire circuit becomes

equivalent to its specification. In other words, our method tries to replace each of the original (possibly) buggy gates with a different logic gate having the same input variables. So the new gate to be used for replacement may have to realize complicated logic functions with the same inputs and cannot be implemented with a single gate, but with a set of simple gates, such as NAND, NOR, etc. As described in the following, we utilize an existing method proposed by Jo et al. [4, 5] to efficiently derive logic functions of programmable circuits. There are, however, bugs which cannot be corrected if the input variables of the gates remain the same. In such cases, we need to add an additional input variable to LUTs or MUXs. When the number of variables in a circuit is very large, it is not practical to check all the variables one by one. To quickly find variables which cannot improve the chance of getting a correct logic when they are connected to LUTs or MUXs, we introduce a necessary condition that should be satisfied by each variable in order to improve the chance of correction. We also propose an efficient selection method based on that condition.

### 11.2.3.2 Base Algorithm: Finding a Configuration of LUTs Using Boolean SAT Solvers

For easiness of explanation, in this paper we assume the number of outputs for the target buggy circuit is one. That is, one logic function in terms of primary inputs can represent the logic function for the entire circuit. This makes the notations much simpler, and also extension for multiple outputs is straightforward.

As there is only one output in the design, a specification can be written as one logic function with the set of primary inputs as inputs to the function. For a given specification,  $SPEC(x)$  and an implementation with programmable circuits,  $IMPL(x, v)$ , where  $x$  denotes the set of primary input variables and  $v$  denotes the set of variables to configure programmable circuits inside, the problem is to find a set of appropriate values for  $v$  satisfying that  $SPEC$  and  $IMPL$  are logically equivalent, which can be described as QBF (Quantified Boolean Formula) problem as follows:  $\exists v \cdot \forall x \cdot SPEC(x) = IMPL(x, v)$ . That is, with appropriate values for  $v$ , regardless of input values (values of  $x$ ), the circuits must be equivalent to the specification, i.e., the output values are the same which can be formulated as the equivalence of the two logic functions for the specification and the implementation. There are two nested quantifiers in the formula above, that is, existential quantifiers are followed by universal quantifiers, which are called two-level QBF in general. Normal SAT formulae have only existential quantifiers and no universal ones.

In [4], Jo et al. proposed to apply CEGAR (Counterexample-Guided Abstraction Refinement)-based QBF solving method to the circuit rectification problem. Here, we explain the method using 2-input LUT for simplicity although LUT having any numbers of inputs can be processed in a similar way. A 2-input LUT logic can be represented by introducing four variables,  $v_{00}, v_{01}, v_{10}, v_{11}$ , each of which corresponds to the value of one row of the truth table. Those four variables are multiplexed with the two inputs of the original gate as control variables, as shown in



**Fig. 11.3** LUT is represented with multiplexed four variables as truth table values

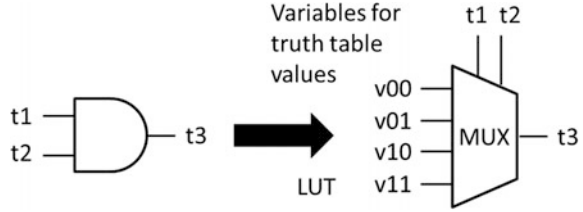


Fig. 11.3. In the figure, a two-input AND gate is replaced with a two-input LUT. The inputs,  $t_1$ ,  $t_2$ , of the AND gate becomes the control inputs to the multiplexer. With these control inputs, the output is selected from the four values,  $v_{00}, v_{01}, v_{10}, v_{11}$ . If we introduce  $M$  of 2-input LUTs, the circuit has  $4 \times M$  more variables than the variables existed in the original circuit. We represent those variables as  $v_{ij}$  or simply  $v$  which represents a vector of  $v_{ij}$ .  $v$  variables are treated as pseudo primary inputs as they are programmed (assigned appropriate values) before utilizing the circuit.  $t$  variables in the figure correspond to intermediate variables in the circuit. They appear in the CNF of the circuits for SAT/QBF solvers.

If the logic function at the output of the circuit is represented as  $f_t(v, x)$  where  $x$  is an input variable vector and  $v$  is a program variable vector, after replacements with LUTs, the QBF formula to be solved becomes  $\exists v \cdot \forall x \cdot f_t(v, x) = f_s(x)$ , where  $f_s$  is the logic function that represents the specification to be implemented. Under appropriate programming of LUTs (assigning appropriate values to  $v$ ), the circuit behaves exactly the same as specification for all input value combinations.

Although this can simply be solved by any QBF solvers theoretically, only small circuits or small numbers of LUTs can be successfully processed [4]. Instead of doing that way, we here like to solve given QBF problems by repeatedly applying normal SAT solvers using the ideas shown in [6, 7].

Basically, we solve the QBF problem only with normal SAT solvers in the following way. Instead of checking all value combinations on the universally quantified variables, we just pick up some small numbers of value combinations and assign them to the universally quantified variables. This would generate SAT formulae which are just necessary conditions for the original QBF formulae. Please note that here we are dealing with only two-level QBF, and so if universally quantified variables get assigned actual values (0 or 1), the resulting formulae simply become SAT formulae. The overall flow of the proposed method is shown in Fig. 11.4. For example, if we assign two combinations of values for  $x$  variables, say  $a_1$  and  $a_2$ , the resulting SAT formula to be solved becomes like:  $\exists v \cdot (f_t(v, a_1) = f_s(a_1)) \wedge (f_t(v, a_2) = f_s(a_2))$ . Then, we can just apply any SAT solvers to them. If there is no solution, we can conclude that the original QBF formulae do not have solution neither. If there is a solution found, we need to make sure that it is a real solution for the original QBF formula. Because, we have a solution candidate  $v_{assigns}$  (these are the solution found by SAT solvers) for  $v$ , we simply make sure the following:  $\forall x \cdot f_t(v_{assigns}, x) = f_s(x)$ . This can be solved by either usual SAT solvers or combinational equivalence checkers. In the latter case,

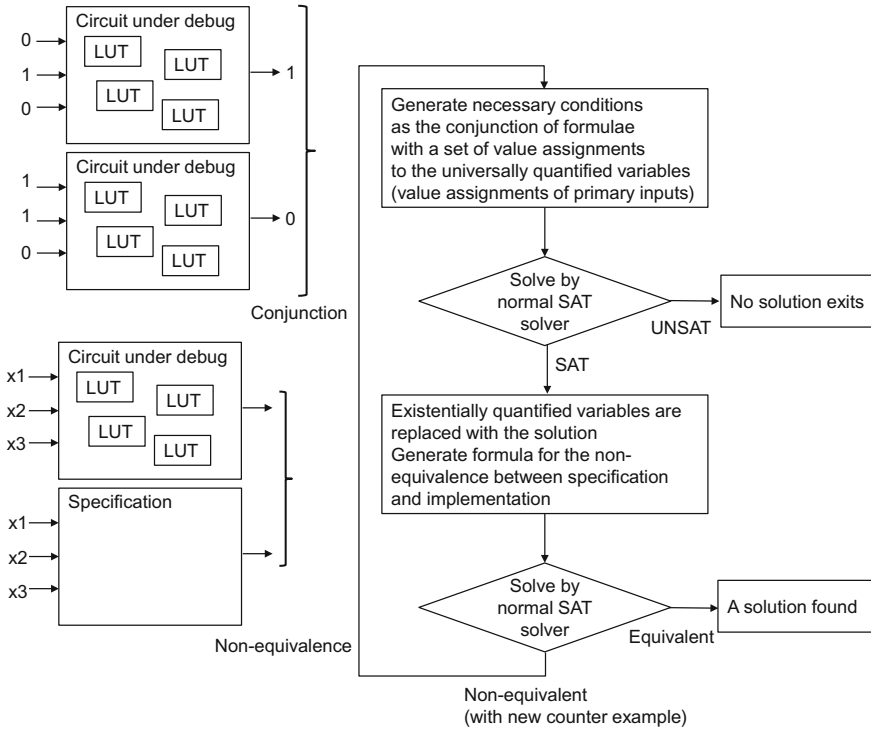


Fig. 11.4 Overall flow of the rectification method in [4]

circuits with tens of millions of gates may be processed, as there have been conducted significant amount of researches for combinational equivalence checkers which utilize not only state-of-the-art SAT techniques but also various analysis methods on circuit topology. If they are actually equivalent, then the current solution is a real solution of the original QBF formula. But if they are not equivalent, a counterexample, say  $x_{sol}$ , is generated and is added to the conditions for the next iteration:  $\exists v \cdot (f_I(v, a1) = f_S(a1)) \wedge (f_I(v, a2) = f_S(a2)) \wedge (f_I(v, x_{sol}) = f_S(x_{sol}))$ . This solving process is repeated until we have a real solution or we prove the nonexistence of solution. In the left side of Fig. 11.4, as an example, the conjunction of the two cases where inputs/output values are (0, 1, 0)/1 and (1, 1, 0)/0 is checked if satisfiable. If satisfiable, this gives possible solutions for LUTs. Then using those solutions for LUTs, the circuit is programmed and is checked to be equivalent with the specification. As we are using SAT solvers, usually nonequivalence can be made sure by checking if the formula for nonequivalence is unsatisfiable.

Satisfiability problem for QBF in general belongs to P-Space complete. In general, QBF satisfiability can be solved by repeatedly applying SAT solvers, which was first discussed under FPGA synthesis in [8] and in program synthesis in

[9]. The techniques shown in [6, 7] give a general framework on how to deal with QBF only with SAT solvers. These ideas have also been applied to so-called partial logic synthesis in [5].

### 11.2.3.3 Proposed Method to Correct Gate-Level Circuits

#### Overall Flow

Figure 11.5 shows an overall flow of our proposed correction method. Given

- a specification,
- an implementation circuit that has bugs, and
- a set of candidate locations of the bugs,

The method starts with replacing each logic gate corresponding to a candidate bug location with a LUT. Each inserted LUT has the same set of input variables as its original gate. Then, by applying the method in [4, 5], we try to find a configuration of the set of LUTs so that the specification and the implementation become logically equivalent. Once such a configuration is found, it immediately means we get a logic function for correction. Then, another implementation will be created based on the corrected logic function, which may require re-synthesis or synthesis for ECO (Engineering Change Order). Although the method to compute a configuration of LUTs for correction in [4, 5] is relatively more efficient than most of the other methods, it can solve up to hundreds of LUTs within a practical runtime. Therefore, it is not practical to replace all of the gates in the given circuit with

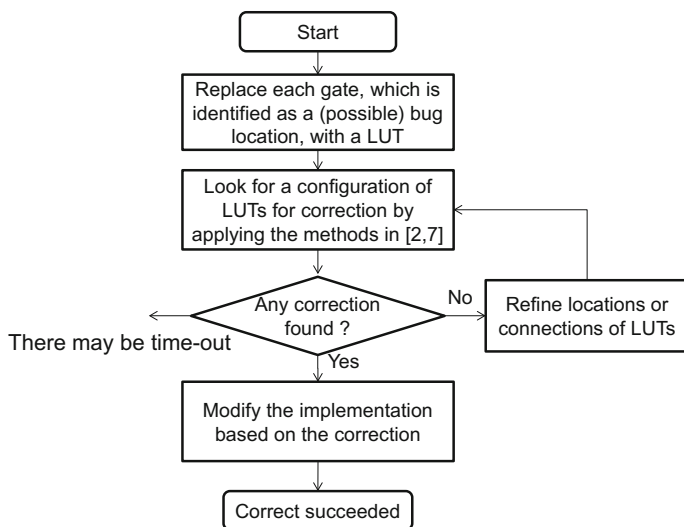
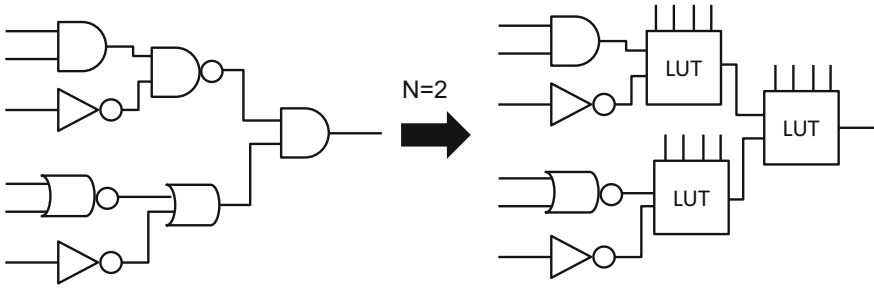


Fig. 11.5 An overall correction flow



**Fig. 11.6** An example of LUT insertions

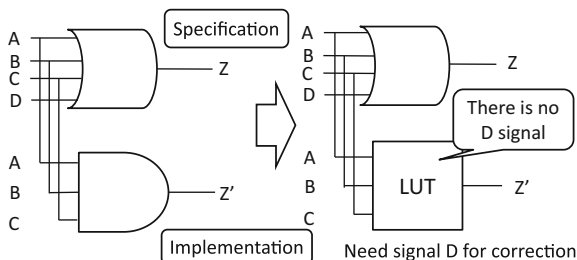
LUTs, and the number of LUTs inserted into the implementation influence a lot on the runtime for correction. In order to obtain candidate locations of bugs, existing methods such as [3] can be utilized. In this work, we employ a simple heuristic, which is similar to the path tracing method that all gates in logic cones of erroneous primary outputs are replaced with LUTs when they are within a depth of  $N$  level from the primary outputs. Figure 11.6 shows an example of such introduction of LUTs. In this figure  $N=2$ . In the experiments described in Sect. 11.4,  $N$  is set to 5. This number is determined through experiments. If the number is larger, there are more chances for the success of corrections. On the other hand, if the number is smaller, we can expect faster processing time.

There can be cases where any correction cannot be found for a given implementation with LUTs. There can be varieties of reasons on the failure. It may be due to the wrong selection of the target gates to be replaced, the inputs to LUTs are not sufficient, or other reasons. In this section, we assume that bugs (or portions that are implemented differently from designers' intention or specification) really exist within the given candidate locations. And, we may need to add more variables to the inputs of the LUTs to increase the chances of corrections, which are discussed in the next subsection.

### Adding Variables to LUT Inputs

As mentioned above, there are bugs that cannot be corrected with LUTs having the same set of input variables as their original gates, if so-called “missing wire” bugs in Abadir's model [10] are happening. Figure 11.7 shows a simple example. In this example, the logic function of an implementation generates  $A \wedge B \wedge C$ , while its specification is  $A \vee B \vee C \vee D$ . With a LUT whose inputs are  $A, B, C$  that replaces the original AND gate in the incorrect implementation, we cannot get any configuration of its truth table for correction, since  $D$  is essential to the correct logic function. In general, assuming that bugs really exist within the gates that are replaced with LUTs, the reason why we cannot obtain any correction is due to the lack of variables that should be connected to appropriate LUTs. Therefore, what we

**Fig. 11.7** An example bug that cannot be corrected with LUTs having the same inputs



need to do in the refinement phase in Fig. 11.5 is adding extra variables to LUT inputs and try to find a correction again. If we inappropriately add a set of variables to input of some LUTs, however, it simply results in no solution in the next iteration of the loop in Fig. 11.5. The number of ways to add extra variables to input of LUTs is large, which cannot be checked one by one in practice. In our method, we try to correct the implementation with adding as small numbers of variables as possible. First, all possible ways to add one variable to LUTs are tried. If no correction can be found, then the method looks for correction with two additional variables to one or two LUTs. Basically, we continue this process until we find corrections.

### Using MUXs to Examine Multiple Additional Variables

As discussed above, the method looks for any correction with adding variables to the inputs of LUTs. Even if only one variable is added to the input of some LUT, we need to iterate the loop in Fig. 11.5 many times until any correction is found or there is a proof of no solution. For a large circuit, the number of iterations may be too large even for the case of adding one variable to a LUT. To make this process more efficient, we introduce a multiplexer and connect multiple variables, which are candidates to be added to a LUT, to its inputs. The output of the MUX and the additional input of the LUT are connected as shown in Fig. 11.8. Then, we can select which variable to be added to the LUT by appropriately assigning values to the control variables of the MUX.

**Fig. 11.8** Additional input variables to a LUT

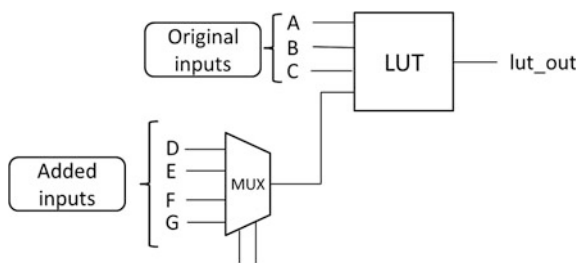


Figure 11.8 shows how multiplexers work for examining candidate variables that may need to be added to the inputs of the LUT in order to get a correction. The LUT in the example originally has three inputs,  $A$ ,  $B$ , and  $C$ , which means this LUT is supposed to be replaced with some 3-input logic gate for corrections. Assume that we want to examine which variable to be added as an input of the LUT, using the MUX in the example, we can examine four additional candidate variables,  $D$ ,  $E$ ,  $F$ , and  $G$  at one iteration. Here, we need to treat the control variables as program variables, that is, same as the ones in the LUTs. If any correction is found, the corresponding values of the control variables identify a variable for addition. That is, if it becomes an input of the LUT connected to the MUX, the implementation can be equivalent to its specification. Otherwise, all variables connected to inputs of the LUT cannot make the incorrect implementation equivalent to its specification. A straightforward way to realize something similar is to introduce LUTs having larger numbers of inputs rather than using MUX. This is definitely more powerful in terms of the numbers of function which can be realized at the output of the LUTs. In the example shown in Fig. 11.8, instead of using a MUX, a LUT having seven inputs may be used, and that LUT can provide much more different functions for possible corrections. The problem, however, is the number of required program variables. If we use a MUX in the example, we need  $2^4 + 2 = 18$  variables. If we use a seven-input LUT, however, we need  $2^7 = 128$  variables, which needs significantly more time to process.

Even when MUXs are used to examine multiple variables at the same time, we should be aware of the increase of the number of program variables. As can be seen in [4, 5], the number of program variables increases runtime for finding a correction, which corresponds to the runtime spent for each iteration of the loop in Fig. 11.5. Please note that one iteration in Fig. 11.5 may include many iterations in Fig. 11.4. In the experiments, we show a case study with varieties of numbers of inputs to MUX.

### Filtering Out Variables Based on Necessary Condition

When a variable is added to an input of a LUT, it may not be an appropriate variable to correct the target bug. Even with the more efficient method using MUXs described above, we should not try to examine a variable which cannot correct the bug. In this subsection, we propose a method for filtering out such non-useful variables from the set of candidate variables that can be connected to inputs of LUTs utilizing necessary conditions on the correctability.

#### *Necessary Condition for the Variables to Be Added*

For simplicity, the following discussion assumes that there is one LUT added to an implementation circuit. It can be easily extended to the cases of a set of multiple LUTs, where each LUT does not have any other LUTs in its fan-in cone (i.e., an

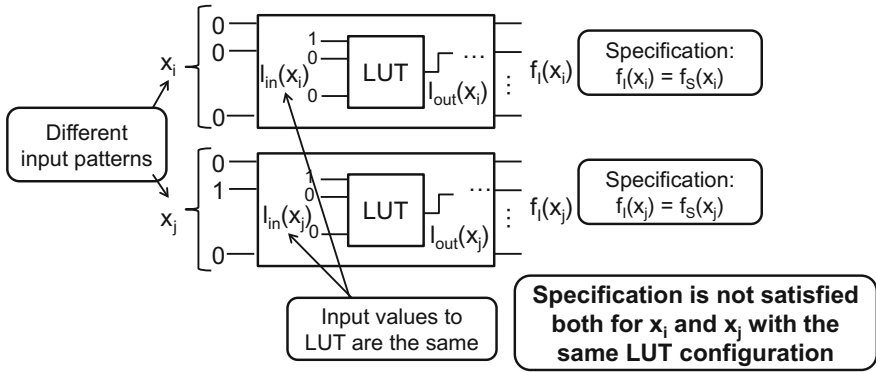


Fig. 11.9 Reason of no correction

LUT depends on other LUTs). However, we here omit such cases. When no correction is found, which corresponds to taking “NO” branch in Fig. 11.5, we cannot correct an implementation under debugging with the current LUT, which is the only LUT added, with its current input variables. The reason why there is no correction (i.e., no configuration of LUTs works correctly) is that the LUT outputs the same value for different two input values to the LUT. This happens when “No solution” is reached in Fig. 11.4. Figure 11.9 explains the situation. In the figure,  $x_i$  is an input pattern added in one of the previous iterations of the process shown in Fig. 11.4, and  $x_j$  is the pattern that is added as a result of the last iteration. Then, there can be situations where the following two conditions are satisfied.

1. For a pair of primary input patterns  $x_i$  and  $x_j$ , the input values to the LUT  $l_{in}(x_i)$  and  $l_{in}(x_j)$  are the same, where  $l_{in}$  represents a logic function that determines an input value to the LUT for a given primary input pattern. Therefore, the output values from the LUT are also the same, that is,  $l_{out}(x_i) = l_{out}(x_j)$ .
2. In order to make the implementation equivalent to the specification for both  $x_i$  and  $x_j$ , that is,  $f_I(x_i) = f_S(x_i) \wedge f_I(x_j) = f_S(x_j)$ ,  $l_{out}(x_i)$  and  $l_{out}(x_j)$  must be different, where  $f_S, f_I$  denote logic functions of primary outputs of the specification and the implementation, respectively.

Note that  $f_S(x_i)$  can be a different value from that of  $f_S(x_j)$ . With the conditions, there is no way to have an LUT configuration that satisfies the specification for both  $x_i$  and  $x_j$  at the same time. In this case, it cannot make an LUT configuration for both  $x_i$  and  $x_j$  if we add a variable  $v$  to the LUT that has the same value for  $x_i$  and  $x_j$  ( $v(x_i) = v(x_j)$ ), since  $l_{out}(x_i)$  and  $l_{out}(x_j)$  are still the same. This is because the output of the LUT can be represented as  $l_{out}(l_{in}(x), v(x))$  for a primary input pattern  $x$  and  $l_{in}(x_i) = l_{in}(x_j) \wedge v(x_i) = v(x_j)$  implies  $l_{out}$  are equivalent for  $x_i$  and  $x_j$  for any configuration of the LUT.

The observation above suggests that we must not add a variable to the LUT input if it has the same value for  $x_i$  and  $x_j$ . It gives us a necessary condition that the added

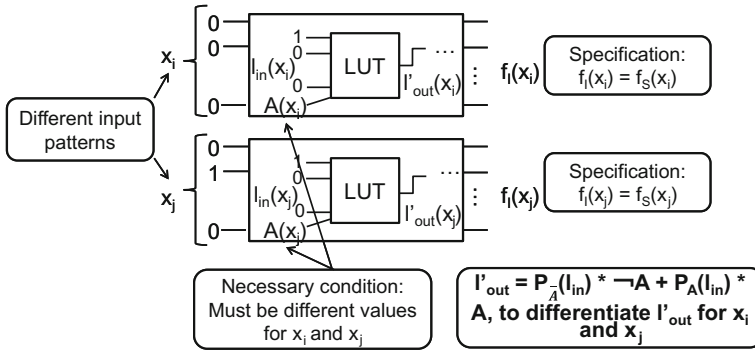


Fig. 11.10 Adding a variable satisfying a necessary condition

variable to a LUT must have different values for  $x_i$  and  $x_j$ . If this necessary condition is satisfied, there is a LUT configuration where  $l_{out}(x_i) \neq l_{out}(x_j)$  is satisfied, which is a requirement to make  $f_S = f_I$  for both  $x_i$  and  $x_j$ . Note that  $f_S = f_I$  may not be satisfied even  $l_{out}$  is different for the two patterns.

Figure 11.10 shows how to make the output of the LUT different by adding a variable that satisfies the necessary condition. Here, we denote the added variable to the LUT as  $A(x)$ , where  $x$  is the primary input variables. An LUT configuration with its input  $l_{in}(x)$  and  $A(x)$  is represented by  $l'_{out}(x)$ , which is rewritten as  $l'_{out}(x) = P_{\bar{A}}(l_{in}) * \bar{A} + P_A(l_{in}) * A$ , where  $P_{\bar{A}}(l_{in})$  and  $P_A(l_{in})$  represent truth table values for  $l_{in}$  when  $A=0$  and  $A=1$ , respectively. This is nothing but Shannon’s expansion of  $l'_{out}$ . If the added variable  $A(x)$  that satisfies the necessary condition takes 1 for  $x_i$  and 0 for  $x_j$ , we can make a LUT configuration satisfying  $l'_{out}(x_i) \neq l'_{out}(x_j)$  by setting the two truth tables  $P_{\bar{A}}$  and  $P_A$  appropriately. For the case of  $x_i=0$  and  $x_j=1$ , a LUT configuration can be obtained in a similar way.

Based on the discussion above, we can filter out variables from candidates when they have the same value for both  $x_i$  and  $x_j$ . Now, we show an example of such filtering. Figure 11.11a is the specification which is  $Z = A \vee B \vee C \vee D$ . Here, we assume that this is one of the specifications and there are other outputs in the target circuit. Now, assume that a wrong implementation is generated as shown in Fig. 11.11b. Here the output only depends only on  $C$  and  $D$ , which is clearly wrong. For the input values where all of inputs are 0, this implementation looks correct as it generate the same output value, 0, as the specification. Please note that the implementation has more gates in the circuit in order to realize the other outputs which are not shown in the figure.

Then, we find a counterexample, which is  $A=0, B=1, C=0, D=0$  as shown in Fig. 11.12a. For these values, the correct output value is 1, but the value of the output in the implementation is 0 as seen from Fig. 11.12b. Our debugging method first replaces the suspicious gate, the OR gate, with a LUT as shown in Fig. 11.12c. Unfortunately, there is no configuration for the LUT which makes the implementation correct, and so we need to add a variable to the LUT. Now, we have two



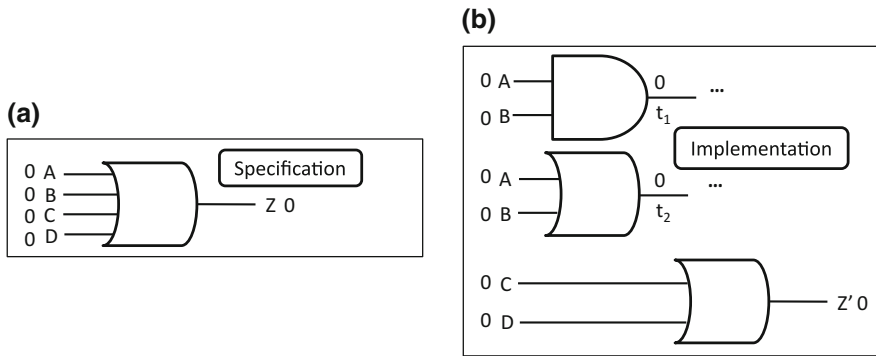


Fig. 11.11 An example specification and its buggy implementation

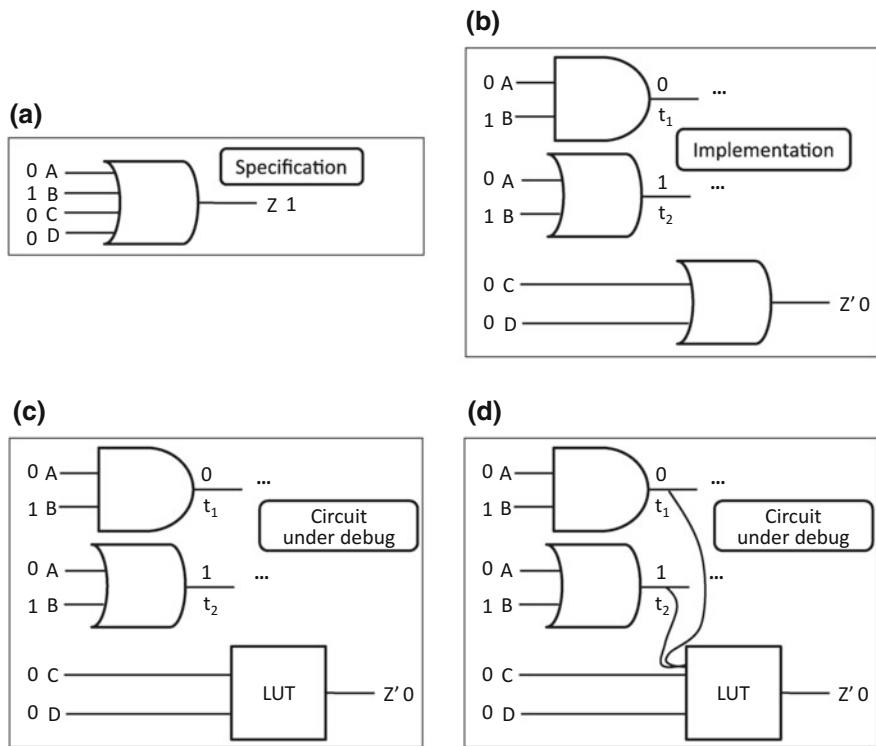


Fig. 11.12 A debugging process for the design in Fig. 11.9 with a necessary condition

candidates for the variable,  $t_1$  and  $t_2$  as shown in Fig. 11.12d. The necessary condition discussed above requires that the value of the variable must be different between the two cases,  $A=0, B=0, C=0, D=0$  and  $A=0, B=1, C=0, D=0$ . From this condition, the variable  $t_1$  is eliminated and the variable  $t_2$  is selected.

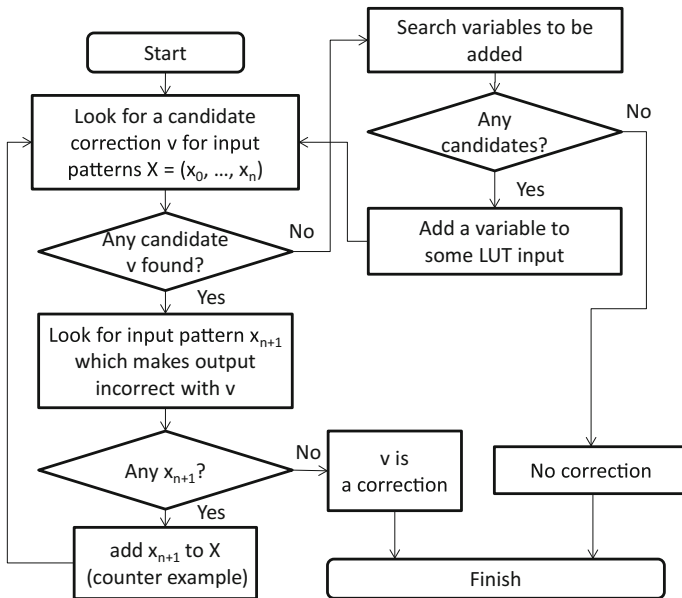
*An Improved Flow with Filtering Variables*

Figure 11.13 shows an improved flow with filtering variables based on the necessary condition discussed above. When no correction is found for all the input patterns so far, the method searches for a set of variables that can be added to an LUT input. During this search, variables which do not satisfy the necessary condition are filtered out. This consists of the following two steps:

1. Find an input pattern  $x_i$  that is added in one of the previous iterations and has the same input values of an LUT as those of the lastly added pattern  $x_j$ .
2. Find a variable having different values for  $x_i$  and  $x_j$ .

As a result, the method tries to add a variable satisfying the necessary condition to some LUT input. Then, with the added LUT input, the method looks for another correction  $v$  by applying CEGAR-based method in [4, 5].

If this filtering method is applied with the method using MUXs to examine multiple variables simultaneously that is described above, the filtering method needs to pick up  $N$  variables, where  $N$  is the total number of input variables to MUXs.



**Fig. 11.13** An bypass flow including filtering out variables

## 11.2.4 Experimental Results

### 11.2.4.1 Experimental Setup

Three sets of experiments are conducted in order to evaluate our debugging methods proposed in this paper. We use the following circuits for the experiments: ISCAS85 benchmark circuits, an industrial on-chip network circuit (“Industrial”), and an ARM Cortex microprocessor (“ARM processor”). While ISCAS85 circuits are combinational ones, the last two circuits are sequential ones. All are in gate-level designs. Table 11.1 shows the characteristics of these circuits. In order to apply our method, sequential circuits need to be time-frame expanded. The numbers of expanded time-frames (i.e., clock cycles for examinations) are shown in the second column for Industrial and ARM processor.

We use PicoSAT [11] as a SAT solver. In order to convert the netlists written in Verilog into SAT formulae, we use ABC [12] and AIGER [13]. All experiments reported in this section are run on a computer with Intel Core 2 Duo 3.33 GHz CPU and 4 GB Memory.

### 11.2.4.2 Simultaneous Examination on Multiple Variables Using Multiplexers

First, we perform an experiment with our method that introduces multiplexers (MUXs) into a circuit under debugging so that multiple extra variables are connected to LUTs through MUXs. In this experiment, we identify the erroneous primary outputs through simulation, and replace all gates in their logic cones within the depth of 5 levels from the erroneous primary outputs with LUTs. Then, we

**Table 11.1** Characteristics of circuits

	# of expansion	Inputs	Outputs	Gates
<i>ISCAS85 benchmarks</i>				
c499		202	41	32
c880		383	60	26
c1355		546	41	32
c1908		880	33	25
c2670		1193	233	140
c3540		1669	50	22
c5315		2307	178	123
c7552		3512	207	108
<i>Others</i>				
Industrial	3	1201	1216	8289
ARM processor	1	895	923	4666

**Table 11.2** Experimental results of simultaneous examination of candidate variables using MUXs

	Inputs of MUX	Change inputs	Time (s)
Industrial	1 (no MUXs)	–	Timeout (–)
	16	15	5281
	64	4	12794
	256	1	211
ARM processor	1 (no MUXs)	–	Timeout (–)
	16	8	11204
	64	2	8857
	256	1	5909

insert a  $N$ -input MUX to the circuit, and its output is connected to all LUTs. We randomly choose sets of variables out of all primary inputs of the circuit to be debugged, and they are connected to the inputs of MUXs.

If no solution for correction can be found, we replace all the input variables to MUXs with another set of variables that are not examined yet and execute the method again. In this experiment, the runtime is limited within 5 hours.

The results are shown in Table 11.2.  $N$ , the number of inputs to MUX, varies from 1 to 256.  $N = 1$  means no MUX, in other words, a variable is directly added to an input of all LUTs. “Change inputs” represents the number of variable sets that are examined for correction. If this number is  $M$ ,  $N \times M$  variables are examined in total. As can be seen in the table, we need to run the method in [4, 5] only a few times when the number of MUX inputs is 64 or 256. “Time” shows the total runtime. We can see that the runtime for 256-input MUX is the shortest in both circuits. Also, it is notable that we cannot find a correction within 5 hours without MUX, since a lot of iterations are performed in order to check many variables one by one.

### 11.2.4.3 Candidate Variable Filtering Using the Necessary Condition

Next, we experiment our method to search for candidate variables that can be added to LUT inputs for a correction of the circuits. The method is based on the necessary condition discussed in Sect. 11.3.3. In this experiment, only an incorrect gate is replaced with an LUT. The candidates of variables are all variables in the circuit under debugging. For this experiment, we need to record the values of internal variables for all input patterns. For this purpose, we use Icarus Verilog simulator [14].

The results are shown in Table 11.3. In this experiment, there is no MUX inserted for the examination of multiple variables at once. Instead each variable is examined one by one. From the table, we can see only small numbers of iterations are required until getting corrections. Comparing to the results in Table 11.2 with

**Table 11.3** Experimental results of filtering candidate variables based on the necessary condition

	Changed inputs	Time (s)
Industrial	29	524
ARM processor	24	293

$N=1$ , where any correction is not obtained within 5 h, the proposed filtering method based on the necessary condition makes the execution time much shorter. It implies that a large number of variables examined in the results shown in Table 11.2 do not satisfy the necessary condition. That is, the necessary condition works very well as filtering.

#### 11.2.4.4 Applying Both Multiple Variable Examination and Candidate Filtering

In the previous experiments, we evaluate our proposed methods for finding variables which can rectify circuits when added to LUT input. That is, simultaneous examination of multiple candidate variables using MUXs and filtering candidate variables based on necessary condition are applied. In this section, we see the effects of applying both of the methods at the same time. For this experiment, we use ISCAS85 circuits and Industrial circuit.

For the experiment, one gate in each ISCAS circuit is replaced with a LUT, and one of its inputs is removed from the LUT. As a result, we realize cases where a potentially buggy gate is replaced with a LUT, but it lacks one input for rectification because we intentionally remove it. The gate replaced with a LUT and a variable to be removed are randomly chosen, and we make five instances for each ISCAS circuit. For Industrial circuit, we replace one of the buggy gates with a LUT. This replaced LUT needs one more input for rectification (without intentionally removing one of its original input) as the original circuit is buggy.

We apply the following three methods for each instance.

- **(PI)** Examining all primary input variables one by one until one can rectify the circuit.
- **(Filtering)** Examining only primary input and internal variables one by one which satisfies the necessary condition discussed in Sect. 11.3.3.
- **(Filtering + MUX)** Examining multiple variables which satisfy the necessary condition using MUX.

The results are shown in Table 11.4. In the table, # of var, Rectified, and # of examined represent the total number of candidate variables, (the number of successfully rectified)/(the total number of instances), and the average number of examined variables in successfully rectified cases, respectively. When # of examined is N/A, it means that none of the experiment instances can be rectified by the corresponding method. Ratio means the ratio of the number of examined variables

**Table 11.4** Experimental results of applying both of our proposed method

Circuit	# of var	Method	Rectified	# of examined (ratio)	Runtime (s)
c499	243	PI	0/5	N/A	46.4
		Filtering	5/5	88.6 (36%)	48.3
		Filtering + MUX	5/5	88.6 (36%)	2.3
c880	443	PI	1/5	61.0 (14%)	80.8
		Filtering	5/5	54.2 (12%)	57.0
		Filtering + MUX	5/5	54.2 (12%)	2.6
c1355	587	PI	0/5	N/A	60.6
		Filtering	5/5	155.8 (27%)	227.7
		Filtering + MUX	5/5	155.8 (27%)	3.3
c1908	911	PI	2/5	34.0 (4.0%)	69.2
		Filtering	5/5	194.2 (21%)	284.5
		Filtering + MUX	5/5	194.2 (21%)	3.9
c2670	1194	PI	0/5	N/A	708.1
		Filtering	5/5	142.2 (12%)	83.2
		Filtering + MUX	5/5	142.2 (12%)	4.7
c3540	1670	PI	0/5	N/A	154.9
		Filtering	5/5	503.8 (30%)	915.9
		Filtering + MUX	5/5	503.8 (30%)	7.8
c5315	2476	PI	0/5	N/A	915.5
		Filtering	5/5	324.6 (13%)	268.1
		Filtering + MUX	5/5	324.6 (13%)	8.8
c7552	3604	PI	0/5	N/A	1484.3
		Filtering	5/5	1016.0 (28%)	3990.1
		Filtering + MUX	5/5	1016.0 (28%)	15.9
Industrial	3209	PI	0/1	N/A	Time out
		Filtering	1/1	100 (3.1%)	972.3
		Filtering + MUX	1/1	100 (3.1%)	172.5

with filtering to the total number of variables. Runtime in the table is the average runtime of the experimented instances.

From the table, we can see the following.

- When we want to rectify circuits utilizing programmability of LUT and one additional input to LUT, we need to add some internal variables (not primary input variables) to the LUT.
- When applying the filtering method to filter out variables not satisfying the necessary condition, we can reduce the numbers of examined candidates to 10–30% of the total variables.
- Examining multiple candidates simultaneously using MUXs reduces the runtime significantly.

### ***11.2.5 Summary and Future Works***

In this paper, we have proposed debugging methods for gate-level circuits applying partial synthesis techniques shown in [4, 5]. In the methods, possible bug locations, which may be given from bug locating methods, are replaced with LUTs, and a configuration of LUTs that makes an implementation under debugging and its specification equivalent is searched. To deal with the missing input variables to LUTs, we have also proposed methods to examine variables for LUT inputs in trial-and-error manner. Using MUXs, multiple variables are examined simultaneously, which largely reduces the number of iterations of the process. In addition, we have introduced a necessary condition that variables added to LUT inputs must be satisfied, so that variables not satisfying the condition can be removed quickly from the candidates. Through the experiments with ARM processor design, on-chip network controller taken from industry, and benchmark circuits, both of our proposals can significantly speed-up the process to get a correction (i.e., an appropriate configuration of LUTs to make an incorrect implementation correct).

We have also discussed about possible extensions of our proposed method in order to introduce sub-circuits having relatively larger numbers of inputs, such as 12 inputs to the buggy locations of the design under debugging. For such large numbers of inputs, it is not practical to represent the entire sub-circuit with a single 12-input LUT. Instead, we have discussed about the introduction of decomposition of such sub-circuits with sets of LUTs having much smaller numbers of inputs. Definitely, this is a very preliminary discussion and much of following works are expected.

As a future work, we plan to develop a method to reduce the candidate variables based on the necessary condition discussed in this paper for the cases where LUTs are dependent with each other. In such cases, the necessary condition may need to be refined to deal with dependency.

## **11.3 High-Quality Delay Testing for In-field Self-test**

Michiko Inoue, Nara Institute of Science and Technology

Tomokazu Yoneda, Nara Institute of Science and Technology

### ***11.3.1 Statistical Delay Quality Level SDQL***

Built-In Self-Test (BIST) is an embedded component in a circuit that can apply self-test to the circuit itself. Since it is embedded in a circuit, it can be used to apply in-field test and contribute to improve the reliability of the circuit. BIST for high-quality delay test that detects small delay defects is effective not only to

improve test quality of production test but also to efficiently and less costly realize failure prediction in field. This subsection introduces a method to improve test quality while reducing test data volume and test application time for BIST to be used in field. Failure prediction method by delay measurement using in-field BIST has been proposed [15]. To practically realize BIST in field, it is required to provide high-quality delay test under strict constraints on test data volume and test application time.

Statistical Delay Quality Level (SDQL) is proposed to evaluate delay test quality for small delay defects [16]. Intuitively, SDQL represents an amount of delay defects that can escape from detection by a given test set. Therefore, smaller SDQL means better test quality. For a given circuit, SDQL of a test set represents a total amount of delay defects that have to be detected but cannot be detected by the test set. Figure 11.14 shows a concept of SDQL. In Fig. 11.14a, there are two paths that pass through a delay fault  $f$ , where the lengths of the paths are 3 ns and 5 ns, respectively. If the path with length of 3 ns is sensitized by some test pattern (by propagating a transition through the path) with test clock of 6 ns,  $f$  is detected if the delay defect size exceeds 3 ns, while a delay defect exceeding 1 ns is detected if a different test pattern sensitizes the path with length of 5 ns. That is detectable delay defect size depends on test patterns. Suppose that the smallest detectable delay defect for  $f$  is 1 ns, that is, delay defect less than 1 ns is timing redundant and does not need to be detected. If the smallest detectable delay defect for  $f$  by a given test set is 3 ns, delay defect whose size is between 1 and 3 ns is escaped from the test. SDQL is a total amount of such test escapes over the all faults by considering a delay defect distribution (Fig. 11.14b).

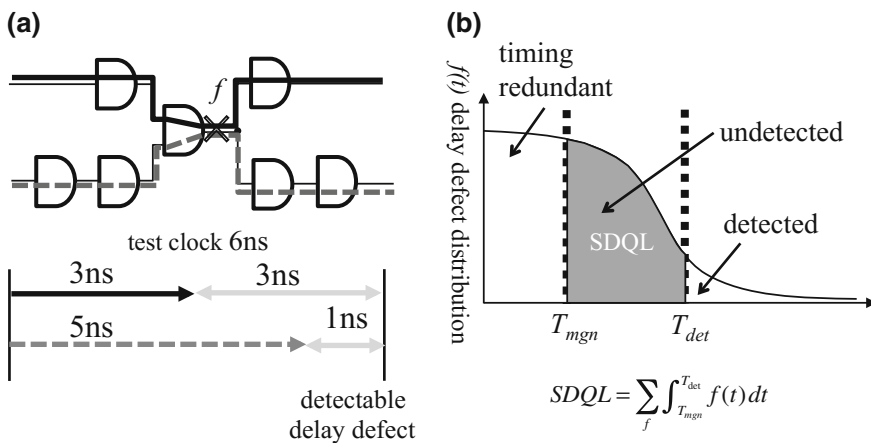
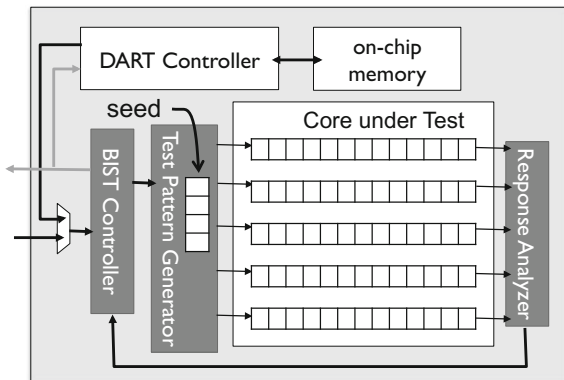


Fig. 11.14 Statistical delay quality level (SDQL)



**Fig. 11.15** In-field built-in self-test



### 11.3.2 In-field Test Using BIST

Failure prediction method using in-field BIST has been proposed [15], where in-field delay measurement is implemented by controlling BIST circuitry from on-chip DART controller (Fig. 11.15). Delay test is applied several times using different test clock frequencies, and actual delay is narrowed down between PASS and FAIL test clock frequencies.

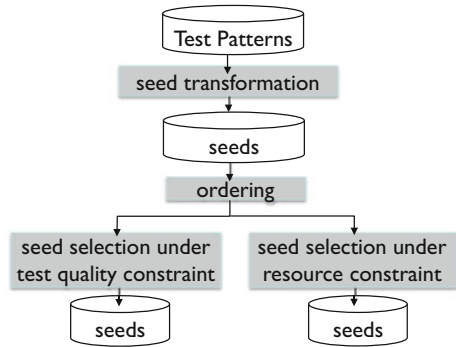
BIST generates one deterministic and multiple pseudo-random test patterns from one seed, and therefore, frequent re-seedings can achieve high test quality with short test application time. However, frequent re-seedings require more test data volume, and test data such as seeds and the corresponding expected signatures should be stored in on-chip memory. In-field test has several constraints on on-chip memory and test application time, and therefore, small number of seeds with high delay test quality is required for accurate in-field delay measurement.

### 11.3.3 Seed Selection for High-Quality Delay Test

Test pattern selection [17] and seed selection [18] for high delay test quality based on SDQL have been proposed. Figure 11.16 shows a flow of seed selection [18]. In this method, first, given test patterns are translated into seeds. Then seeds are sorted in the order so that SDQL is decreased (improved) as early as possible. The ordered seeds can be used to minimize the number of seeds under a constraint on test quality, or to maximize test quality (minimize SDQL) under a constraint on the number of seeds. Seeds are selected in the obtained order while a selected seed set satisfies a given constraint.

Seed ordering can be done by selecting test patterns one by one by repeatedly obtaining SDQL using timing-aware fault simulation. However, it is unpractical since timing-aware fault simulation is too time consuming. Seed ordering [18] is

**Fig. 11.16** Seed selection for high-quality delay test



accelerated by using lengths of sensitized paths ( $T_{det}$  in Fig. 11.14b) instead of actual SDQL values. In the proposed method, first lengths of sensitized paths for all the faults are obtained for each seed. Then seeds are selected one by one based on the obtained lengths of sensitized paths. Timing-aware fault simulation is required for once as a preprocessing in the first step, and it can order a given test pattern set in a reasonable time.

Experiments have been conducted for several ITC'99 benchmark circuits. Synopsys TetraMAX ATPG with Small Delay Defect Test mode [19] is used for timing-aware test generation and fault simulation. Table 11.5 shows the characteristics of the circuits and the results of test pattern generation and seed transformation and ordering. The columns “# test patterns”, “# seeds” show the number of test pattern generated by TetraMAX, and the number of seeds transformed from the test patterns. Test patterns generated by timing-aware ATPG and  $n$ -detect test patterns targeting transition faults are preliminarily compared, and timing-aware ATPG is adopted to obtain better SDQL with a smaller number of seeds. The columns “FC (%)”, “SDQL”, and “TGT (m)” show transition fault coverage, SDQL, and test generation time for the test patterns, respectively, and “ordering time (m)” show processing time for seed ordering. The results show that the proposed method efficiently orders the seeds.

Figure 11.17 shows the effectiveness of the proposed method. The figure compares SDQL transitions among the proposed seed ordering, random ordering, and an original ATPG order. The proposed seed ordering achieves less (better) SDQL with a smaller number of seeds. Tables 11.6 and 11.7 show some examples

**Table 11.5** Circuit characteristics, results of test generation, seed transformation, and ordering

Circuit	# gates	# faults	# FFs	# scan chains	# test patterns	# seeds	FC (%)	SDQL	TGT (m)	Ordering time (m)
b15	8,985	17,329	417	8	727	700	82.0	2498.0	1.89	0.6
b17	27,766	65,218	1,317	26	1,375	1,319	86.2	7841.8	9.18	2.3
b18	79,400	172,403	3,020	80	3,293	3,129	79.7	33986.1	43.82	19.7
b19	152,599	353,301	6,042	120	6,131	5,850	79.0	70768.2	115.39	74.9

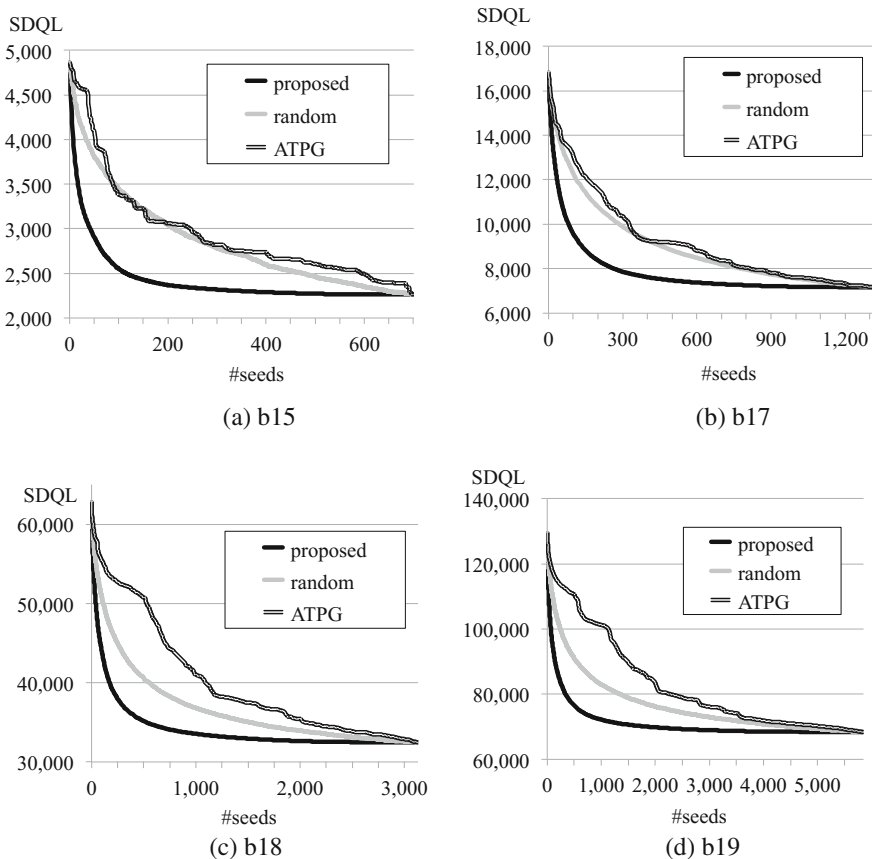


Fig. 11.17 SDQL transition

Table 11.6 The number of selected seeds under SDQL constraints

Circuit	b15		b17		b18		b19	
	2,500	3,500	8,000	10,000	35,000	45,000	70,000	90,000
Proposed	116	17	266	81	525	85	1,875	146
Random	473	83	780	288	1,516	249	4,541	542
ATPG	597	88	816	328	2,049	707	5,139	1,486

of seed selection under SDQL constraints and seed count constraints, respectively. The number of seeds and SDQL are reduced by ordering seeds using the proposed method, especially when SDQL constraints are relatively large or seed count constraints are relatively small. These cases correspond to requirements of small

**Table 11.7** SDQL of selected seeds under seed count constraints

Circuit	b15		b17		b18		b19	
# seeds constraint	100	200	500	1,000	500	1,500	500	1,500
Proposed	2,550	2,369	7,474	7,193	35,149	32,950	76,420	70,643
Random	3,443	3,044	8,826	7,566	40,555	35,047	90,964	78,917
ATPG	3,387	3,064	9,177	7,612	50,756	37,493	110,546	89,716

test data volume or short test application time. That is, the proposed seed ordering can be effectively used in in-field BIST environments.

There are some variations of the proposed method. A simplified version just targets transition fault coverage, where time-consuming fault simulation is not required and ordering can be done more efficiently. The proposed ordering method can be extended to handle mixed-mode BIST where one deterministic test pattern and multiple pseudo-random test patterns are generated from one seed. In an extended version of the proposed method, the longest sensitized paths are evaluated for each test pattern set generated from the same seed. These variations including the original version can give an adequate solution to meet several requirements such as test data volume (seed count), test quality (SDQL), and processing time.

## 11.4 Temperature-and-Voltage-Variation-Aware Test

Tomokazu Yoneda, Nara Institute of Science and Technology

Yuta Yamato, Nara Institute of Science and Technology

### 11.4.1 Thermal-Uniformity-Aware Test

In advanced technologies, temperature-induced delay variations during the test are as much as those caused by on-chip process variations [20]. Temperature difference within a chip can be as high as 50 °C and typical time intervals for temperature changes over time are very short time of milliseconds. Besides, the execution of online self-test may last for a long time [21]. For example, the extremely thorough test patterns that specifically target aging [22] may take several seconds to complete. This indicates that, for accurate aging prediction, we need to embed a lot of temperature sensors into several locations within a chip to collect spatially and temporally temperature profile during. However, this is not the cost-effective solution since it incurs a lot of overhead. Even if we could accept the overhead, it requires (1) a lot of data to be stored in the memory and (2) a complex procedure to eliminate the temperature-induced delay variations from the measured delay values

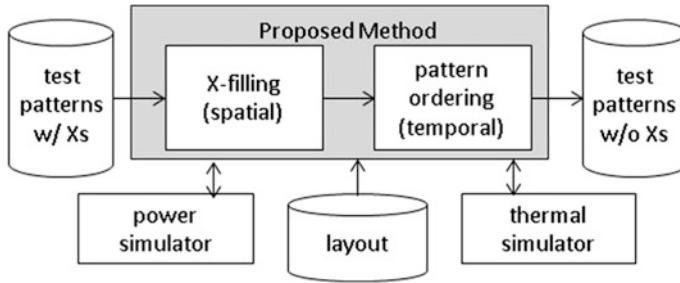


Fig. 11.18 Test pattern optimization flow

for aging analysis. Therefore, we need a cost-effective solution to eliminate the temperature-induced delay variations for accurate aging prediction.

Yoneda et al. proposed a test pattern optimization method to reduce the spatial and temporal temperature-induced delay variations. The proposed method consists of the following two steps: (1) X-filling [23] and (2) test pattern ordering [24] as shown in Fig. 11.18.

#### 11.4.1.1 X-filling for Spatial-Thermal-Uniformity

The proposed method starts with a test sequence including unspecified bits (X's) generated by a commercial ATPG. As the first step, the thermal-uniformity-aware X-filling technique [23] is performed to obtain a test sequence of test patterns with fully specified bits. For each test pattern  $i$  together with the test response of  $i - 1$ , the Xs are specified so as to minimize the spatial temperature variance during scan shift operation while preserving the power consumption at relatively low level. Figure 11.19 shows the temperature profile for ITC'99 benchmark b17 after Step 1. In this example, the circuit was divided into 16 blocks based on the layout and each line on the graph represents the temperature profile of a block. Compared to the temperature profiles of the test patterns with minimum transition fill (conventional low power patterns) in Fig. 11.20, the spatial temperature variation is reduced significantly. However, temporal temperature variations (around 20 °C in this example) are still remaining.

#### 11.4.1.2 Test Pattern Ordering for Temporal Thermal Uniformity

The proposed test pattern ordering technique [24] determines an order of the fully specified test patterns so that the temporal temperature variance is minimized while preserving the spatial temperature variance achieved in the first step. The main idea is to adopt a sub-sequence-based ordering strategy. The method divides the test pattern sequence into several sub-sequences based on thermal simulation, and

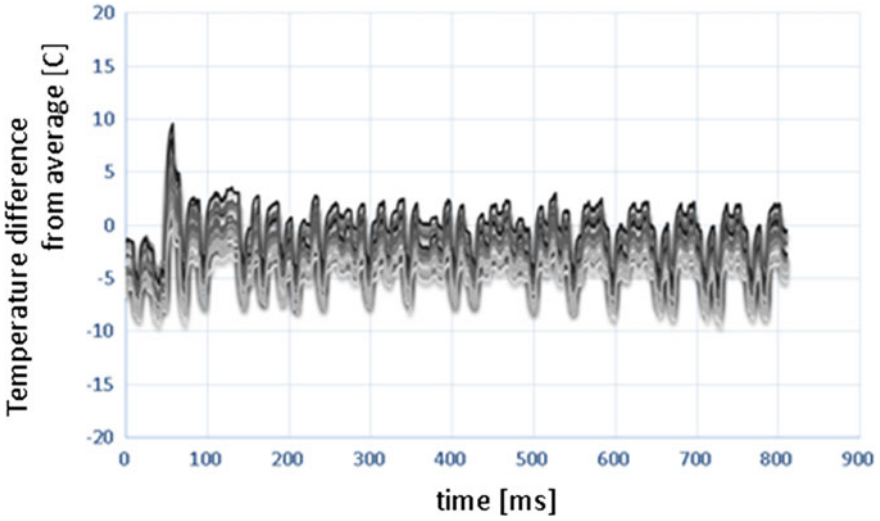


Fig. 11.19 Temperature variation in test patterns with thermal-uniformity-aware X-filling

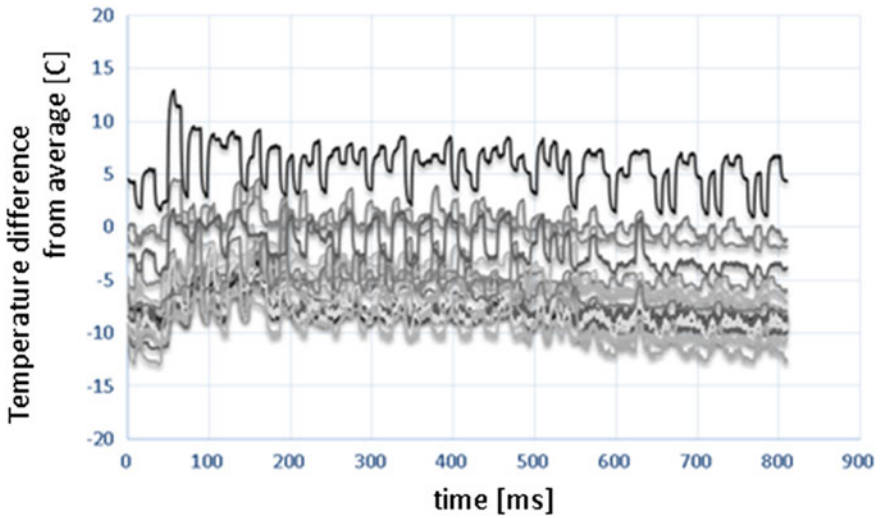


Fig. 11.20 Temperature variation in test patterns with minimum transition fill

orders the *heating* and *cooling* sub-sequences in an interleaving manner to reduce the temporal temperature variation. The spatial thermal-uniformity achieved in the first step is valid only for the current response-pattern pair which is simultaneously shifted during a test in the current order. Therefore, the sub-sequence-based ordering itself can preserve the spatial-thermal-uniformity without any consideration

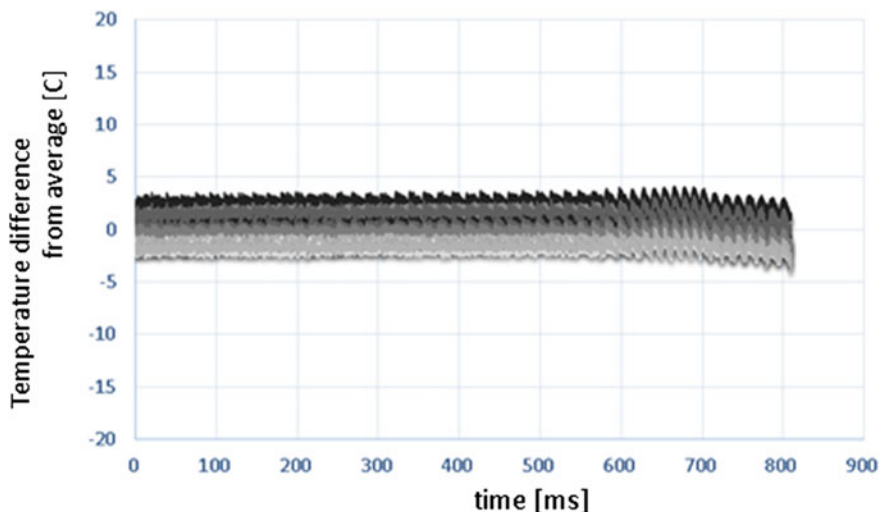


Fig. 11.21 Temperature variation in test patterns after thermal-uniformity-aware test pattern ordering

if the length of sub-sequences is long enough, and allow us to minimize the temporal temperature variance as shown in Fig. 11.21. Experimental results show that the proposed test pattern optimization method obtained 73% and 95% reductions in spatial and temporal temperature variation, respectively, on average for several ITC'99 benchmarks.

#### 11.4.2 Fast IR-Drop Estimation for Test Pattern Validation

In addition to temperature, voltage is another main contributor to delay variation. Excessive voltage drop causes severe yield loss problems during delay test. Generally, delay testing is performed using scan design, a typical design for testability (DFT) technique. During scan testing, circuits operate with high switching activity since it causes state transitions which cannot occur during functional operation. In case that instantaneous switching activity is high, large current flows along power distribution network in a short period of time. This results in voltage decrease at power supply port of cell instances due to resistance of metal wires (IR-drop). Since IR-drop increases signal transition delay at cell instances, if cumulative delay increase on a sensitized path exceeds the specified timing margin, timing failure occurs. In such case, even functionally good chips may also fail the test and will not be shipped, i.e., yield loss. Therefore, test patterns that can cause excessive IR-drop should be identified and removed or modified before test application [25].

**Fig. 11.22** Variation in path delay increase due to IR-drop

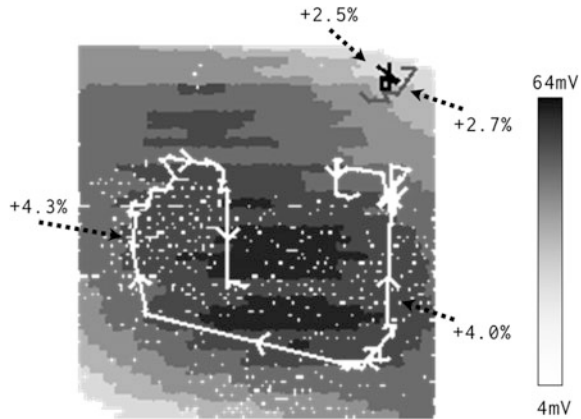


Figure 11.22 shows variation in delay increase of sensitized paths depending on the amount of IR-drop. It can be seen that paths running through the instances with high IR-drop suffers from higher delay increase. Based on this observation, it is necessary to compute the amount of IR-drop for every cell instance for each pattern to accurately evaluate test patterns. Though this is generally realized by precise circuit-level simulation, it usually takes long computation time and thus may be impractical for large industrial designs. On the other hand, evaluation using estimated power dissipation or signal switching count is much more scalable and widely used. However, since the length of sensitized paths differs depending on patterns even their power dissipations are similar, it can be difficult to directly evaluate the risk of IR-drop-induced timing failures.

To accurately evaluate test patterns in a realistic amount of time, the authors in [26] have proposed a fast pattern-dependent per-cell IR-drop estimation method. The basic idea is to reduce the number of time-consuming IR-drop analyses during entire analysis. General flow and the way to reduce the number of IR-drop analyses are described in the following subsections.

#### 11.4.2.1 General Flow

Figure 11.23 shows a comparison between typical IR-drop analysis flow that performs IR-drop analysis for all patterns and the proposed flow. Differently, from the typical flow, the method first selects a few representative patterns as targets of IR-drop analysis. IR-drop analyses are performed only for the selected patterns. Then, fast IR-drop estimation function is derived for each cell instance using analyzed IR-drop and corresponding power profile. After that, IR-drop for the remaining patterns is computed using the functions.



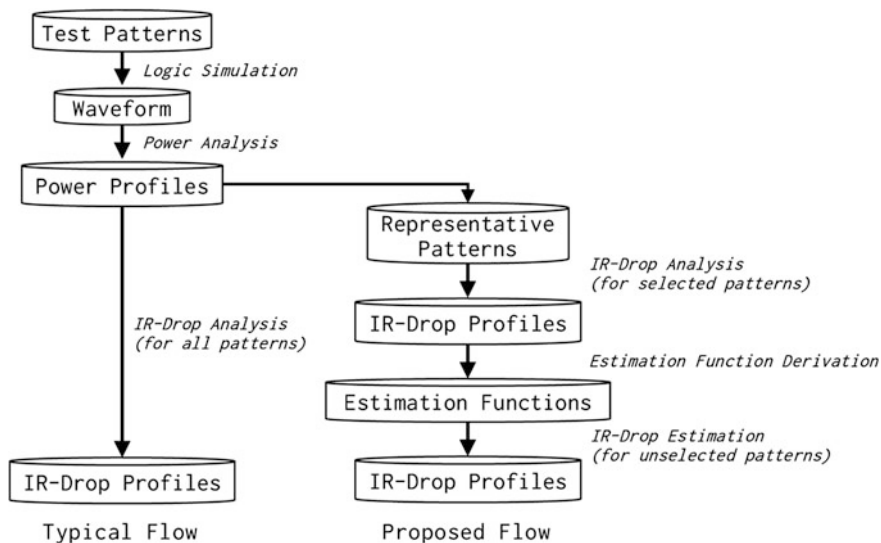


Fig. 11.23 Comparison between typical flow and proposed flow

### 11.4.2.2 Reducing Number of IR-Drop Analysis

The idea of reducing the number of IR-drop analyses is based on high correlation between circuit’s average power dissipation over a clock cycle and IR-drop for each cell. As can be seen in Fig. 11.24, by focusing on individual cell instance,

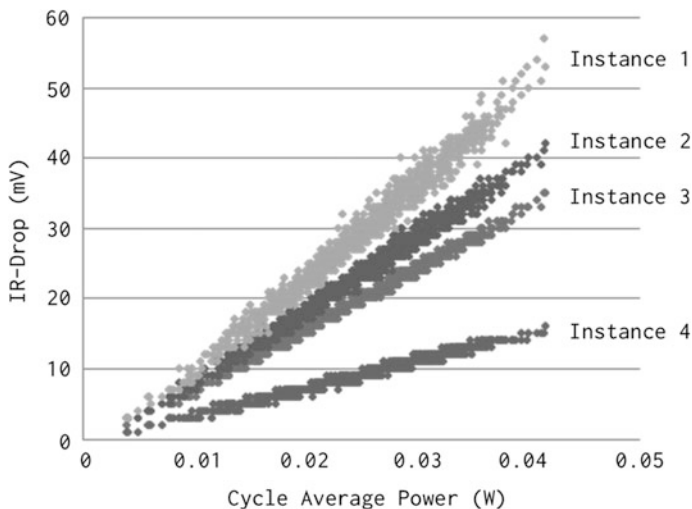


Fig. 11.24 Correlation between global cycle average power and individual cell instance

cycle average power and IR-drop has an almost linear correlation. The proposed method derives IR-drop estimation function for each cell by linear regression using a few IR-drop analysis results of representative patterns. For the remaining patterns, IR-drop for each cell instance can be computed using its cycle average power. Since the estimation functions are linear function, the computation effort is significantly reduced compared to IR-drop analysis.

Experimental results have shown that the proposed method achieves more than 20X speed-up to 6 mV error on average compared to typical flow.

## References

1. IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800™-2012, Chaps. 16, 19 (2012)
2. IEEE Standard for Property Specification Language (PSL), IEEE Std 1850™-2005 (2005), pp. 101–111
3. M. Fahim Ali, A. Veneris, A. Smith, S. Safarpour, R. Drechsler, M. Abadir, Debugging sequential circuits using Boolean satisfiability, in *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004* (2004), pp. 204–209
4. S. Jo, T. Matsumoto, M. Fujit, SAT-based automatic rectification and debugging of combinational circuits with LUT insertions. *Test Symposium (ATS), 2012 IEEE 21st Asian* (2012), pp. 19–24
5. M. Fujita, S. Jo, S. Ono, T. Matsumoto, Partial synthesis through sampling with and without specification. *ICCAD 2013*, 787–794 (2013)
6. M. Janota, J. Marques-Silva, Abstraction-based algorithm for 2QBF, in *Theory and Applications of Satisfiability Testing (SAT) 2011*. Lecture Notes in Computer Science, vol. 6695 (2011) pp. 230–244
7. M. Janota, W. Klieber, J. Marques-Silva, E. Clarke, Solving QBF with counterexample guided refinement, in *Theory and Applications of Satisfiability Testing (SAT) 2012*. Lecture Notes in Computer Science, vol. 7317 (2012), pp. 114–128
8. A. Ling, P. Singh, S.D. Brown, FPGA logic synthesis using quantified boolean satisfiability. *SAT 2005*, 444–450 (2005)
9. A.S.-Lezama, L. Tancou, R. Bodik, S.A. Seshia, V.A. Saraswat, Combinatorial sketching for finite programs. *ASPLOS* (2006), pp. 404–415
10. M.S. Abadir, J. Ferguson, T.E. Kirkland, Logic design verification via test generation. *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.* **7**(1), 138–148 (1988)
11. A. Biere, PicoSAT essentials. *J. Satisfiability, Boolean Model. Comput. (JSAT)* (2008), pp. 75–97
12. R. Brayton, A. Mishchenko, ABC: an academic industrial-strength verification tool. *Comput. Aided Verif.* **6174**, 24–40 (2010)
13. AIGER, <http://fmv.jku.at/aiger/>
14. Icarus Verilog, <http://iverilog.icarus.com/>
15. Y. Sato, S. Kajihara, T. Yoneda, K. Hatayama, M. Inoue, Y. Miura, S. Untake, T. Hasegawa, M. Sato, K. Shimamura, DART: dependable VLSI test architecture and its implementation, in *Proceedings IEEE International Test Conference* (2012), p. 15.2
16. Y. Sato, S. Hamada, T. Maeda, A. Takatori, Y. Nozuyama, S. Kajihara, Invisible delay quality-SDQM model lights up what could not be seen, in *Proceedings IEEE International Test Conference 2005* (IEEE, 2005) p. 47.1
17. M. Inoue, A. Taketani, T. Yoneda, H. Fujiwara, Test pattern ordering and selection for high quality test set under constraints. *IEICE Trans. Inf. Syst.* **95**(12), 3001–3009 (2012)

18. T. Yoneda, I. Inoue, A. Taketani, H. Fujiwara, Seed ordering and selection for high quality delay test, in *Proceedings IEEE Asia Test Symposium (ATS) 2010* (IEEE, 2010), pp. 313–318
19. TetraMAX ATPG User Guide, Version C-2009.06-SP2 (2009)
20. S.A. Bota, J.L. Rossello, C.D. Benito, A. Keshavarzi, J. Sequra, Impact of thermal gradients on clock skew and testing. *IEEE Des. Test Comput.* **23**(5), 414–424 (2006)
21. Y. Li, O. Mutlu, S. Mitra, Operating system scheduling for efficient online self-test in robust systems, in *Proceedings International Conference on Computer-Aided Design (ICCAD '09)* (Nov 2009), pp. 201–208
22. A.B. Baba, S. Mitra, Testing for transistor aging, in *Proceedings VLSI Test Symposium (VTS '09)* (May 2009), pp. 215–220
23. T. Yoneda, M. Inoue, Y. Sato, H. Fujiwara, Thermal-uniformity-aware X-filling to reduce temperature-induced delay variation for accurate at-speed testing, in *Proceedings VLSI Test Symposium (VTS '10)* (Apr 2010), pp. 188–193
24. T. Yoneda, M. Nakao, M. Inoue, Y. Sato, H. Fujiwara, Temperature-variation-aware test pattern optimization, in *European Test Symposium (ETS '11)* (May 2011), p. 214
25. P. Girard, Survey of low-power testing of VLSI circuits. *IEEE Des. Test Comput.* **19**, 80–90 (2002)
26. Y. Yamato, et al., A fast and accurate per-cell dynamic IR-drop estimation method for at-speed scan test pattern validation, in *Proceedings International Test Conference (ITC '12)*, paper 6.2 (2012)