Efficient Constraint Extraction for Template-Based Processor Self-Test Generation

Kazuko Kambe† Tsuyoshi Iwagaki‡ Michiko Inoue† Hideo Fujiwara†
†Graduate School of Information Science, Nara Institute of Science and Technology
Kansai Science City 630-0192, Japan
‡School of Information Science, Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
E-mail:†{kazuk-ka, kounoe, fujiwara}@is.naist.jp, ‡iwagaki@jaist.ac.jp

Abstract

This paper presents efficient method to extract constraints from a test program template and synthesize a test program using constraint circuits. A test program template is an instruction sequence with unspecified operands, and represents paths for justification of test patterns and observation of test responses for a module under test (MUT). The constraint circuit represents a relation between operand values and inputs/output of the MUT, therefore it enables to obtain operand values using a standard automatic test pattern generator. Experimental results show that the proposed method generates accurate and compact constraint circuits, and we obtain high fault efficiency.

1. Introduction

Recent processor designs with giga-hertz speed and rich functionality demand for accurate and at-speed testing. Software-based self-test attracts attention as a testing strategy enabling at-speed test [1, 2, 3, 4, 5, 6, 7]. In this approach, a processor enables self-test by running a sequence of instructions called a test program. Therefore, it does not induce any performance penalty, area overhead or excessive power. Recent works are towards hierarchical test generation approach targeting structural faults where gate level tests for modules in processors are linked with test programs at processor level. In this approach, values or sequences of values applicable to the modules inside the processor are restricted as a test program delivers test patterns to the modules. These works extract such restrictions for a module under test (MUT) as constraints, then apply an automatic test pattern generator (ATPG) to the MUT with the extracted constraints. Finally, the generated tests are transformed into test programs.

The method in [8] reduces a target circuit to a subset of the circuit surrounding an MUT by analyzing RTL descriptions and applies ATPG to the subset circuit. However, the generated test patterns might induce overtesting at instruction level, since the method does not consider constraints imposed instructions. Furthermore, it is possible that the extracted subset is the same size as the original design in the worst case.

The methods in [5], [6] and [7] are proposing test program generation based on test program templates (or just templates) targeting stuck-at faults. A template is an instruction sequence with unspecified operands that delivers tests to an MUT and observes the test responses. The methods extract constraints from each template since the template represents ways to propagate tests from primary inputs and the test response to primary outputs. Therefore, the generated tests for a module are guaranteed to be transformed into test programs. However, in [5], constraints are obtained by a regressive analysis for randomly generated values of operands, and it might be difficult to find accurate constraints if mapping functions between operands and MUT inputs are complicated.

In this paper, we present an efficient method to extract constraints from a template that generated by our previous method in [6]. Our technique analyzes RTL descriptions of a processor according to behavior of a template and synthesizes constraint circuits, which work as mapping functions between operands and MUT inputs. We obtain operand values from generated test patterns using standard ATPG. The method has the following advantages: (i) we can generate an accurate and compact constraint circuits even if a relation between operands and MUT inputs is complicated, (ii) a constraint circuit is synthesized as a combinational circuit by applying time expansion to a template, thereby we apply combinational ATPG to a combinational MUT.

This paper is organized as follows. In Sections 2 and 3, we give a brief explanation of our self testing system and template generation method in [6]. A method of constraint extraction is proposed in Section 4, and the experimental results are shown in Section 5. Finally, Section 6 concludes this paper.
2. Template based test program generation

In our methodology, constraints are extracted by templates. A template is a sequence of instructions with unspecified operands, and represents paths for justification of test patterns and observation of test responses for an MUT.

**Example 1** An example in Figure 2 is a template for testing Arithmetic Logic Unit (ALU) of Parwan processor [9], where \( loc1, loc2, loc3 \) and \( loc4 \) are operands of instructions. The values are unspecified. First four instructions, CLA, ASL, LDA and ADD, justify values of Status Register (SR) and Accumulator (AC). Next SUB applies test patterns to ALU. STA observes the test response stored in AC.

We explain an overview of our test program generation in Figure 1. We first analyze controllability and observability of registers to use for instruction selection. We identify redundant faults prior to test program generation. Here, we consider a fault is redundant if no test program detects the faults. The method in [7] proposed to identify redundant faults for sequential modules. Control signals applicable to combinational modules from a controller are restricted to constant values imposed by instruction code. We apply such restriction to the MUT as constraints. For each MUT, we repeat the following. We generate a template, and synthesize input/output constraints of the MUT from the template. After applying ATPG, a test program is transformed from the test patterns. In the next step, a fault simulation is applied to an entire processor. We generate multiple templates to cover input space for MUTs. These processes are repeated until all templates are generated or fault coverage reaches the required level. A methodology for template generation was described in [6]. In this paper, we focus the steps from constraint circuit synthesis to test program synthesis in shaded area of the figure.

3. Template generation

In our template generation method, we first select instructions to justify values of each input adjacent registers. Input/output adjacent registers of an MUT are defined as registers that are connected with inputs/outputs of the MUT directly or indirectly through combinational circuits. Once values of input adjacent registers are justified, the test patterns can be applied to inputs of the MUT. If some registers still require justification, more instructions should be selected until source registers of instructions reach the memory. We also select instructions to propagate the fault effect to the memory such that the instructions observe a value of an output adjacent register, and append to the end of the template. Since one template may not cover the whole input space of an MUT, we generate more templates. Our method uses an IE-Graph proposed in [4] to explore delivering routes for justification and propagation. An IE-Graph is a model of data flow between registers. Nodes are registers, IN and OUT, which model external world such as the memory. A directed edge between two nodes represents data transfer between registers controlled by an instruction. More details regarding template generation method can be found in [6].

4. Constraint extraction

Figure 2 illustrates an outline of test program generation for a template. The ALU has four inputs (\( a_{side}, b_{side}, alu\_code \) and \( in\_flags \)) and two outputs (\( alu\_out \) and \( alu\_flags \)). We synthesize input/output constraints from the template. The circuits describe relations between values of operands (\( mem[loc1], mem[loc2], mem[loc3] \) and \( mem[loc4] \)) and inputs/outputs of ALU in HDL coding style. We apply ATPG to a circuit that consists of input/output constraint circuits and the MUT, targeting faults in the MUT. Now, generated test patterns are directly linked to the operand values. In consequence, it is easy to synthesize a test program.

Time expansion of template

We are interested in relations between values of operands and inputs/output of an MUT. In our method, we perform time expansion of a template, and eliminate registers. We represent necessary data flow for justification/observation paths as a sequence of module instances where modules have different instances for different time frames. We can naturally obtain such data flow during traversing paths in an IE-graph to select instructions for template generation. We construct an instance data flow graph (IDFG) to represent data flow between module instances. Figure 3 shows an example of a partial IDFG. It corresponds to a part of
that control signals have constant values corresponding to analyzing behavioral descriptions of a controller, we can find nal. Such signals might be given from a controller. Ana-
statements to execute based on value of given selection sig-
branch statements are generally used to specify a set of
how to optimize RTL descriptions.

tion enables to reduce RTL descriptions. We will describe
meaningful functions of modules regarding a specified instruc-
tion of the module. A directed edge between represents
data flow between nodes. Each edge is labeled with a set of
source signal name, destination signal name] pairs. The
signal names are described in RTL descriptions. This in-
formation of labels is used to simplify RTL descriptions of
modules.

Optimization and synthesis of constraint circuits
When a module is instantiated to a specified state of con-
troller during the execution of an instruction, all of the paths
in modules are not activated. Therefore, extracting meaningful functions of modules regarding a specified instruction enables to reduce RTL descriptions. We will describe how to optimize RTL descriptions.

In a RTL description of module behavior, conditional branch statements are generally used to specify a set of statements to execute based on value of given selection signal. Such signals might be given from a controller. Analyzing behavioral descriptions of a controller, we can find that control signals have constant values corresponding to execution of an instruction sequence: ADD, SUB and STA. Nodes of an IDFG are instances of modules, IN or OUT. Each node is marked with a [state, instruction] pair, which means that data transfer occurs at the state of a controller during executing the instruction where an execution of the instruction consists of multiple cycles. Each module instance has a RTL description duplicated from the description of the module. A directed edge between represents the data flow between nodes. Each edge is labeled with a set of [source signal name, destination signal name] pairs. The signal names are described in RTL descriptions. This information of labels is used to simplify RTL descriptions of modules.

Figure 2. Test program generation.

Figure 3. Instance data flow graph.
a constraint for a cycle and the sequencer selects the constraint corresponding to the cycle.

5. Experimental results

Extracting meaningful functions of module instances yields dead codes and unnecessary ports. In this paper, we eliminated such dead codes and ports, and synthesized constraint circuits by a logic synthesis tool Design Compiler from Synopsys.

We evaluated the proposed method for Parwan processor. Table 1 shows the average of constraint circuit area for modules. The area of output constraint circuits for ALU, Shifter Unit(SHU) and Program Counter(PC) are 0.0, however, the circuits have some lines. This shows that the value of MUT outputs can be observed at primary outputs as it is. In this experiment, we assume a fault to be detectable if an error is propagated to the outside of a controller; accordingly, output constraint circuits for Controller(CTRL) were not synthesized. Notice that every constraint circuit is small. The input constraint circuits of CTRL and PC which are sequential modules are more than 1.5 times of the modules. However, the average of constraint circuit area per cycle is 126.2 for CTRL, and 98.2 for PC, respectively. The constraint circuits for sequential modules are also compact.

We combined constraint circuits with an MUT, and applied ATPG tool TestGen from Synopsys to the circuit. The results are shown in Table 2. Row 2 is the total number of generated templates, and row 3 is the number of templates that actually contributed to detecting faults. Other rows show total faults of each module, the number of faults detected by ATPG. Test programs detected faults in row 7. Consequently, we can achieve high fault efficiency for every module. The experimental results show that our method can synthesize accurate and compact constraint circuits efficiently.

6. Conclusion

This paper presented an efficient constraint extraction method based on templates. We proposed systematic method to extract constraints and synthesize test programs using constraint circuits. The constraint circuit represents relations between values of operands and inputs/output of the MUT. The experimental results showed that we could generate an accurate and compact constraint circuit. Hence, we can perform test generation and test program synthesis efficiently. We obtained high fault efficiency for each module.

Acknowledgments We would like to thank Prof. Satoshi Ohtake, Prof. Tomokazu Yoneda, Mr. Virendra Singh and everyone in Computer Design and Test Lab. of Nara Institute of Science and Technology for their valuable comments. This work was supported in part by Semiconductor Technology Academic Research Center (STARC) under the Research Project and in part by Japan Society for the Promotion of Science (JSPS) under Grants-in-Aid for Scientific Research B(2)(No. 15300018).

References