FACTOR: A Hierarchical Methodology for Functional Test Generation and Testability Analysis*

Vivekananda M. Vedula and Jacob A. Abraham
Computer Engineering Research Center
The University of Texas at Austin
Austin, TX 78712
{*vivek, jaa*}*@cerc.utexas.edu*

Abstract

*This paper develops an improved approach for hierarchical functional test generation for complex chips. In order to deal with the increasing complexity of functional test generation, hierarchical approaches have been suggested wherein functional constraints are extracted for each module under test (MUT) within a design. These constraints describe a simplified ATPG view for the MUT and thereby speed up the test generation process. This paper develops an improved approach which applies this technique at deeper levels of hierarchy, so that effective tests can be developed for large designs with complex submodules. A tool called FACTOR (***F****unction****A****l* ***C****ons****T****raint extract****OR****), which implements this methodology is described in this work. Results on the ARM design prove the effectiveness of FACTOR-ising large designs for test generation and testability analysis.*

1. Introduction

Chip complexity has been increasing exponentially with the advances made in manufacturing technology, allowing designers to incorporate more functionality onto a single chip. However, generating effective manufacturing tests for these chips is becoming more difficult. Several Design for Test (DFT) techniques are currently being developed to generate effective test patterns. Some of these include boundary scan insertion [4] and adding hardware to isolate/access individual modules [1]. These techniques may be simpler to apply and scale well with increasing complexity, but may result in high area and performance overhead.

State-of-art VLSI designs are implemented using deep sub-micron technologies, for which functional tests continue to be most widely accepted. Functional tests applied at speed have been shown to detect the most common manufacturing defects such as crosstalk, opens, shorts and delays. Maxwell and Aitken [5] have shown that at-speed functional test patterns with 75% stuck fault coverage can predict the defect level more accurately than scan test patterns with more than 90% coverage. However, manual generation of functional test vectors is too tedious, while a sequential ATPG tool would not be able to handle complete processor designs.

Commercial sequential ATPG tools work well on smaller circuits or modules in a design. Though the test generation time is greatly reduced, these patterns cannot be easily translated to the chip level due to the constraints imposed by the surrounding logic. If a technique to translate any test patterns generated for the MUT to the processor level is available, commercial tools can be used to generate high quality chip-level functional tests. Our goal is to develop functional test generation techniques for a module embedded in a large design with a fault coverage and test generation time comparable to that for the stand-alone module.

Several approaches have been suggested to improve the test generation process. Lee and Patel [3] proposed a solution based on two custom ATPG packages, one for full chip level justification and propagation and another for test generation at module level. Ramachandini and Thomas [7] suggested a synthesis based approach, in which functional constraints are extracted during the synthesis process and used to guide a custom ATPG tool. Vishakantaiah and others [11], developed a tool called ATKET to extract functional constraints in a VHDL RTL design, which are used with a custom test generation tool to obtain test patters. These approaches use a custom test generation tool and/or require the complete designs to be synthesized.

Tupuri *et al.* suggested a hierarchical abstraction technique [8, 9], wherein each module in a design hierarchy is targeted at a time. The logic surrounding this module is extracted and synthesized to gate level. These synthesized

*This work was supported by the Semiconductor Research Corporation under Contract 99-TJ-715.

constraints were provided to a commercial ATPG tool to generate test patterns for the MUT.

In large designs, the submodules may themselves prove to be too complex for the ATPG tool, and this technique may not be directly applicable. Modules at lower levels of hierarchy may be considered, but the surrounding logic may prove to be too complex. Therefore, a modified approach [10] was suggested in which the constraints were hierarchically extracted and composed to provide the top level functional constraints on the MUT. This systematic approach allows the constraints extracted to be reused. However, the constraints in [10] were extracted manually, and it was not clear how the process could be automated.

In this paper, an algorithm to apply this methodology is described. A tool called **F**unction**A**l **C**ons**T**raint extract**OR** (FACTOR) was implemented using this algorithm. This tool analyzes a Verilog RTL description of a design, identifies the constraints for the MUT at various levels of hierarchy and provides its ATPG view for test generation. The analysis also provides a valuable insight into the testability of the design, and identifies internal registers that can be used to further reduce the ATPG view. This tool was applied to a Verilog model of ARM, and the results show the effectiveness of this approach. Additionally, the tool provides testability data for any modules on which low test coverage was obtained.

A brief description of the test generation methodology is given in section 2. In section 3, the algorithm and implementation of FACTOR are described. Section 4 includes the results obtained on the ARM Verilog benchmark design, with conclusions in section 5.

2. Test Generation Methodology

2.1 Functional Constraint Extraction

The proposed test generation methodology exploits the natural division of designs into components or modules. The reachable state space of a module is defined by its interface with the rest of the design. If an ATPG tool is used to generate test patterns on a full chip while targeting faults in a module, it needs to explore the complete state space of the surrounding logic, which is too tedious. Figure 1 describes the basic idea that was suggested to deal with this problem.

Imagine a simple system consisting of module M, and the surrounding logic defined by the rest of the design S. Using this technique, a reduced version of S, which characterizes just the behavior of S visible to M is derived. This reduced environment, S', is synthesized to the gate level and combined with the module M, to give the *transformed module*. A commercial ATPG tool is used to generate test patterns targeting faults in M.

To further improve the test coverage, internal registers which can be accessed from the chip level using the load/store instructions are identified. These registers, referred to as *PIERs* (Primary Input/output accEssible Registers) help reduce the sequential depth during test generation. The patterns obtained are later translated back to the chip level.

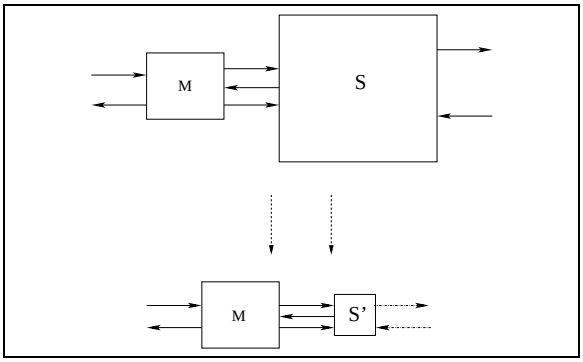

Figure 1. Basic Idea for Test Generation

2.2 Functional Constraint Composition

In large designs, each module may be composed of several submodules, making it too complex for this technique to be applied. Our results, presented later, will show that the fault coverage can be quite poor when dealing with large modules in complex designs. Therefore modules at lower levels of hierarchy need to be considered. This requires a modified approach [10] for constraint extraction, wherein the constraints need to be extracted in stages and *composed* to provide the final constraints on the module under test.

This improved strategy facilitates the reuse of constraints extracted at various levels of hierarchy, and thereby reduces the overall constraint extraction time. The approach is also able to exploit the synthesis tool, which removes the redundant constraints at a given level of hierarchy. We have automated the hierarchical constraint extraction and constraint composition processes. The tool methodology and implementation are described in the following section.

3. FACTOR: Tool Methodology

FACTOR is implemented in PERL, and is built using the *Rough Verilog Parser* [12]. The parser supports both Register-Transfer (RT) and gate level Verilog constructs. The data structure is modified to incorporate *def-use* chains and *use-def* chains, which are vital to traverse the code easily. These chains contain the statements where a signal definition is used and where a signal usage is defined, respectively. For each non-continuous assignment statement, the conditional statements, loops and concurrency constructs are derived. The final data structure is shown in Figure 2. Note that the leaf nodes of this connectivity tree are either Verilog statements or library primitives.

Additional subroutines are built for easy traversal of the data structure. The constraint extractor uses the parser mod-

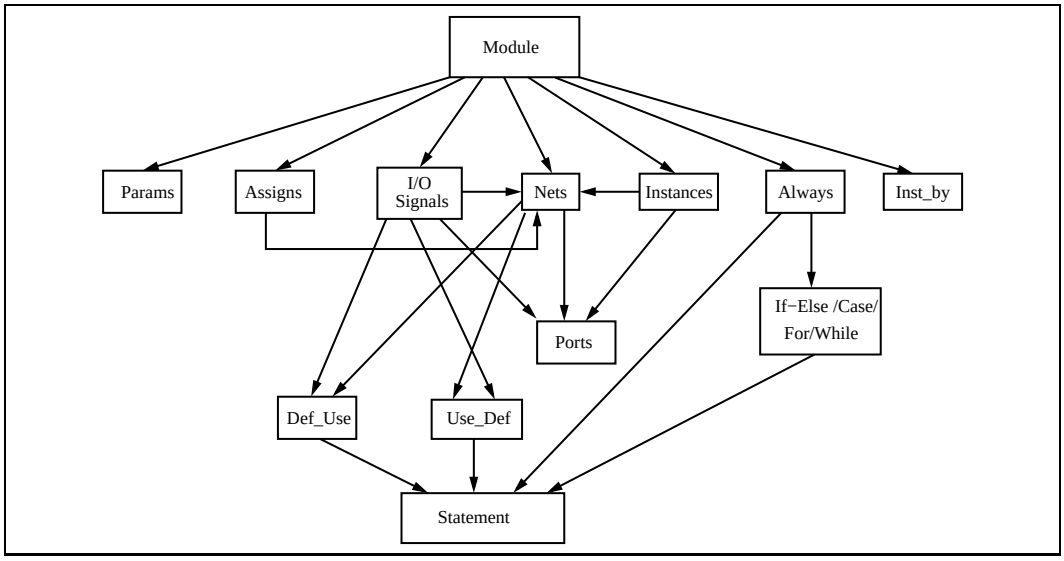

Figure 2. Internal Data Structure

ule to read in the Verilog files and build the internal data structure. Once the MUT and the top module are identified, FACTOR calls appropriate subroutines. For each input signal of the MUT the subroutine *find_source_logic* is called, and for each output signal the subroutine *find_prop_paths* is called. The pseudo-code for these recursive subroutines is given in Figure 3.

```
# Subroutine to extract source logic
sub find_source_logic (signal, module) {
  1. If (module eq top_module) then return;
  2. Traverse the data structure to identify the "driving_signal"
     and "driving_module";
  3. Find and save the ud−chain for the "driving signal"
     in the "driving_module";
  4. For each defn. in the ud−chain, find and save all enclosing
     conditional statements, loops and concurrency constructs;
  5. For each "enc_driving_signal" in the statements obtained in 4,
     find_source_logic (enc_driving_signal, driving_module)
  6. For each "rhs_driving_signal" in each defn. in the ud−chain,
     find_source_logic (rhs_driving_signal, driving_module)
}

# Subroutine to extract propagation logic
sub find_prop_paths(signal, module) {
  1. If (module eq top_module) return;
  2. Traverse the data structure to identify the "driven_signal"
     and "driven_module";
  3. Find and save the du−chain for the "driven_signal"
     in the "driven_module";
  4. For each  "rhs_driven_signal" (excluding driven_signal)
     in each use in the du−chain,
     find_source_logic (rhs_driven_signal, driven_module);
  5. For each "lhs_driven_signal" in each use in the du−chain,
     find_prop_paths (lhs_driven_signal, driven_module);
  6. For each use in the du−chain, find and save all the enclosing
     conditional statements, loops and concurrency constructs;
  7. For each "enc_driven_signal" in the statements obtained in 6,
     find_source_logic (enc_driven_signal, driven_module);
}
```

Figure 3. Main Subroutines in FACTOR

Once these subroutines build the constraint data structure, FACTOR writes out the constraints in the form of synthesizable Verilog netlists. It retains the original directory structure instead of creating unique instances or renaming nets, which makes the extraction process faster and also produces concise code. It also helps keep the constraints clean for further analysis, if necessary.

As described in the algorithms, the tool extracts all possible source and propagation paths at each level of hierarchy. This is necessary since the tool cannot predict which of these paths would eventually reach the chip level interface. The redundant logic or the dead code at each level of hierarchy is eliminated during synthesis by using the appropriate flags.

In addition to extracting the required constraints, the tool also provides a trace for any signals within a given module for which a *def-use* or *use-def* chain is empty. This implies that a path from the chip interface to the MUT's is not found, and will therefore result in poor coverage. This information provides a valuable and quick insight into the testability of the design prior to actual test generation. The designer may choose to make minor alterations to the design to remove the testability bottlenecks.

The potency of this tool lies in its ability to gather precious testability knowledge without having to build and analyze the state machine for the design, which would have been too complex in time and space.

4. Results

A Verilog model of the ARM-2 processor [2] was used as the benchmark for test generation and testability analysis. The proposed technique was applied to modules embedded two or more levels in the hierarchy. Table 1 gives the characteristics of the modules under test.

Table 1. Modules in ARM

Module Name	Hierarchy Level	Primary Inputs	Primary Outputs	Gates in Design	Gates in Surrounding Logic	Stuck-at Faults
arm_alu	2	77	36	3836	11463	11868
regfile_struct	3	101	99	7641	9658	28260
exc	2	8	3	15	17284	108
forward	2	29	4	84	17215	548

Table 2. Transformed Module Without Composition

Module Name	Extraction Time (s)	Synthesis Time (s)	Gates in Surrounding Logic	Surrounding Gate Reduction %	Primary Inputs	Primary Outputs
arm_alu	0.21	1.23	745	93.50	296	103
regfile_struct	1.60	0.98	678	92.98	225	164
exc	0.51	0.69	273	98.42	101	70
forward	0.82	0.73	373	97.83	159	71

Table 3. Transformed Module With Composition

Module Name	Extraction Time (s)	Synthesis Time (s)	Gates in Surrounding Logic	Surrounding Gate Reduction %	Primary Inputs	Primary Outputs
arm_alu	0.23	0.73	133	98.83	296	103
regfile_struct	0.53	0.72	179	98.14	108	98
exc	0.32	0.75	273	98.42	101	70
forward	0.40	0.71	347	97.98	160	71

The tool was used to extract constraints using the conventional methodology, *i.e.*, without any composition of the constraints. These constraints were synthesized to the gate level to give the virtual logic surrounding the MUT. A *transformed module* was built by combining the MUT with its surrounding virtual logic. The extraction and synthesis times are given in Table 2.

Constraint extraction followed by synthesis were performed at each level of hierarchy for the modules under test, as described in Section 2.2. The extraction and synthesis process details are given in Table 3.

The extraction process was performed using a 1 GHz Athlon processor with 256 MB RAM, while the synthesis was done using a 450 MHz UltraSPARC-II dual processor with 1 GB RAM. All the time units presented are in system CPU seconds. Note that the number of gates in the surrounding logic is drastically reduced in both cases. The extraction times are lower using composition technique, because constraints extracted at higher levels were reused.

4.1 Test Generation

A commercial ATPG tool was used to generate test patterns on an UltraSPARC-II processor. Test generation was done on the stand-alone module and at the processor level while targeting faults in the module. The results, presented in Table 4, are presented to demonstrate the difficulty of the ATPG tool for modules embedded in a fairly large design. The second and third columns show the fault coverage and test generation times when the entire processor is given to the commercial tool, with the faults in the module being targeted. The fourth and fifth columns are the results for the stand-alone module.

Table 4. Raw Test Generation

Module Name	Proc. Lvl. Cov. %	Proc. Lvl Time (s)	Std-Al Cov. %	Std-Al Time (s)
arm_alu	0.19	18.6	99.8	0.7
regfile_struct	0.16	66.9	81.67	1.2
exc	66.67	5.8	100	0.5
forward	0.23	5.8	100	0.5

The *transformed modules* obtained using both the traditional and compositional techniques are provided to the same ATPG tool, and the results are presented in Table 5 and Table 6, respectively.

A comparison with the raw test generation results shows that the overall methodology improved the fault coverage by several orders of magnitude, while also reducing the test generation time drastically. Between the two techniques used, the compositional technique resulted in better coverage, while reducing the test generation time by more than 50% in most cases. In the particular case of the *regfile_struct* module, this technique improved the coverage by a factor of

5, and reduced the overall test generation time by a factor of almost 10. This module is the biggest among the modules considered and the most deeply embedded in the design, which shows the necessity for the improved technique when dealing with large hierarchical designs.

Table 5. Test Gen. Without Composition

Module Name	Fault Cov. %	ATPG Eff. %	Test Gen. Time (s)	Total Time (s)
arm_alu	79.49	86.02	5.8	7.24
regfile_struct	**16.03**	**22.49**	**23.0**	**25.58**
exc	100	100	0.7	1.9
forward	87.12	87.66	0.6	2.15

Table 6. Test Gen. With Composition

Module Name	Fault Cov. %	ATPG Eff. %	Test Gen. Time (s)	Total Time (s)
arm_alu	80.87	88.58	3.0	3.96
regfile_struct	**81.49**	**84.56**	**1.4**	**2.65**
exc	100	100	0.5	1.57
forward	94.91	95.12	0.6	1.71

Note that the test generation time and the fault coverage for a *transformed module* are comparable to that of the corresponding stand-alone module, which is exactly the objective of this work. Our approach results in minimal constraints by eliminating the redundant gates in the virtual logic. The use of *PIERs* further helps reduce the sequential depth of the *transformed module*, and thereby helps obtain fault coverage close to that of a stand-alone MUT.

4.2 Testability Analysis

During the process of constraint extraction, the tool also helps estimate any loss in fault coverage. For example, the coverage obtained for the *arm_alu module* is less than its stand-alone coverage, because 10 out of its 13 input control signals are driven from a set of hard-coded values depending on a single input *alu_operation* signal. This type of constraints cannot be further simplified, and FACTOR flags a warning in such cases. In addition, it gives details about the MUT signal that is affected and a trace of all the signals in the aborted path. This information can be utilized by the designer to modify/add design elements for testability.

5. Conclusions

A powerful tool to implement hierarchical test generation is described in this paper. The tool reduces the overall test generation time while also providing valuable insight into the testability of a design. The improved approach removes the bottleneck of the earlier methodology [8, 9] by allowing the reuse of previously extracted constraints. The results generated using the tool show its capability to ease the ATPG complexity, thereby making this technique systematic and scalable for large designs.

6. Acknowledgements

The authors would like to thank the anonymous reviewers of this work for their helpful comments on an earlier version of this paper. Many thanks to Madhavi Valluri and Anand Ramachandran, Ph.D. students at the University of Texas at Austin, for their valuable suggestions to improve the presentation of this work.

References

[1] D. Bhattacharya, "Hierarchical Test Access Architecture for Embedded Cores in an Integrated Circuit,", *Proc. 16th IEEE VLSI Test Symposium*, April 1998, pp. 8-14.

[2] D. V. Campenhout, "A CMOS ARM Microprocessor," *Class Report*, 1995

[3] J. Lee, and J. H. Patel, "Hierarchical Test Generation under Intensive Global Functional Constraints," *Proc. 29th Design Automation Conference*, June 1992, pp. 261-266.

[4] C. M. Maunder and R. E. Tulloss, "The Test Access Port and Boundary-Scan Architecture," *IEEE Computer Society Press*, September 1990.

[5] P. C. Maxwell and R. C. Aitken, " Test Sets and Reject Rates: All Fault Coverages Are Not Created Equal," *IEEE Design & Test of Computers*, Vol. 10, Issue 1, March 1993, pp. 42-51.

[6] K. L. McMillan, "A Compositional Rule for Hardware Design Refinement," *Proc. 9th International Conference on Computer Aided Verification*, June 1997, pp. 24-35.

[7] R. S. Ramachandini and D. E. Thomas, "Behavioral Test Generation using Mixed Integer Non-linear programming," *Proc. International Test Conference*, October 1994, pp. 958-967.

[8] R. S. Tupuri and J. A. Abraham, "A Novel Functional Test Generation Method for Processors using Commercial ATPG," *Proc. International Test Conference*, November 1997, pp. 743-752.

[9] R. S. Tupuri, A. Krishnamachary and J. A. Abraham, "Test Generation for Gigahertz Processors using an Automatic Functional Constraint Extractor," *Proc. 36th Design Automation Conference*, June 1999, pp. 647-652.

[10] V. M. Vedula and J. A. Abraham, "A Novel Methodology for Hierarchical Test Generation using Functional Constraint Composition," *Proc. IEEE International High-Level Design Validation and Test Workshop*, November 2000, pp. 9-14.

[11] P. Vishakantaiah, J. A. Abraham and M. Abadir, " Automatic Test Knowledge Extraction from VHDL (ATKET)," *Proc. 29th Design Automation Conference*, June 1992, pp. 273-278.

[12] v2html, "Rough Verilog Parser," Version 6.0, www.burbleland.com/v2html/rvp.html.