Test Generation for Subtractive Specification Errors

Patricia S. Lee, Ian G. Harris
Technical Report CECS-10-02
April 26, 2010
Center for Embedded Computer Systems
6210 Donald Bren Hall
University of California, Irvine
Irvine, CA 92697-2625
{leep, iharris}@ uci.edu
http://www.cecs.uci.edu/

Abstract—**Scenario-Based Modeling (SBM) exposes Specification Translation Errors (STE), which are not captured using traditional coverage-based test generation techniques that look at the code, not the specification. We improve test generation by specifically exposing subtractive STE.**

Keywords-scenario; specification; verification; test generation; coverage metrics; HDL; behavioral; simulation-based; testbench

I. INTRODUCTION

While most metrics (such as line/statement coverage) focus on the structure of designs, **Specification Translation Errors (STE)**, which are misunderstandings of design specifications, are an important set of problems that targets aspects associated directly with the specification. About 54% of bugs analyzed by the Intel group for the Pentium® 4 processor design [15] were specification-level errors. However, STE are often overlooked. Adding specification based analysis to code-based methods is argued to provide better error coverage and additionally improve code-based coverage numbers. The modeling of designs at this level of abstraction is not what we are proposing, but rather we are proposing the automated process of creating testbenches at this level.

Specification-related errors are typical of most errors found in real-life examples, primarily arising when dealing with large projects. For instance, several designers that are part of a team or many teams of designers could be assigned to work on one specification but assigned to various parts or components of that specification. One designer or team is involved with implementing a part of the specification that affects the behavior of another part of the specification implemented by another team or designer. Because some intended or unintended dependencies defined by the specification exist in the design, it is possible to introduce errors due to miscommunication or incorrect implementation of the specification by one or both of the implementation designers or teams. A dependency defined by the specification could be incorrectly added or subtracted within the code due to this misunderstanding of the specification. Additionally, if the specification is long and detailed, designers typically will forget or miss certain aspects of the specification in the design.

STE are significant and are not directly captured using traditional coverage-based test generation techniques. Traditional methods analyze the code, not the specification. As a result, subtractive errors of omission (i.e. those which are not in the code) are overlooked and not detected. A testbench based on the Hardware Description Language (HDL) does not always catch bugs showing inconsistencies between the design and specification. With coverage-based metrics, the code corresponds directly with the metric, and the coverage is dependent on the code. To detect these STE, a concrete method of describing the specification is needed.

Scenarios [20, 21], like live sequence charts [19], describe "typical" behavior already present in the specification and can be derived from timing diagrams. They show a sequence of events to cause a typical behavior to occur. The only difference of information between scenarios and timing diagrams is that scenarios do not have real timing elements in it—it only contains sequences of events. Using **Scenario-Based Modeling (SBM)** techniques to create testbenches exposes errors missed by coverage-based techniques and so greatly enhances the error detection process by using both techniques together. In this paper, we are focusing on using our SBM method of test generation to improve defect detection coverage for subtractive STE over traditional techniques which rely primarily on coverage metrics.

After the previous and related works section, a system overview will be provided in III. Section IV is a description of scenarios, which are used for creating SBM testbenches, and Section V steps through examples and implementation details of the SBM method. Section VI explains STE in more detail, and Section VII describes the types of errors targeted. Section VIII details the system, experiments and algorithms. Section IX explains the testbench generation algorithm. Finally, Sections X and XI conclude with experimental results and a concluding analysis.

II. PREVIOUS AND RELATED WORKS

A. *Test Generation for Simulation-based Validation*

The focus of this paper is on the research of simulation-based verification using test generation techniques evaluated with the test criterion of coverage metrics. Input is generated to exercise the DUT (Device Under Test), and the output is evaluated in a response checker which compares observed and expected behavior [2].

Traditionally, test generation in simulation was performed using random and directed test programs that target design-based errors, those errors which are related to the design of the system/DUT and exercise the existing model of the circuit.

Some techniques involve a hybrid of these techniques with specialized algorithms such as [4] which uses a strategy to map high-level faults into logic-level faults, genetic [5] and b-algebra [6] which provide a more directed approach. Also, techniques in test generation have been proposed with formalize specifications [18]. This paper introduces Scenario-Based Modeling (SBM) to create testbenches from the original natural language specification.

One way of classifying test generation targeting a specific set of coverage metrics is by its abstraction levels of design. We focus on high abstraction levels of the design at the behavioral level. While [4, 5, 6] focus on structural errors, they are also targeting the HDL (Hardware Description Language) design at the behavioral level. Our proposed test generation algorithm utilizes high-level scenarios generated directly from the specification to target specification-based errors which are related to the specification utilizing functional validation rather than the relying on the design of the system/DUT to generate the testbench. This provides high test quality with respect to finding bugs, errors or faults within the device with functionality based on the specification to generate special cases that excite intended and missed design features. Also, our technique automates the testbench creation process from these scenarios.

B. *Coverage Metrics*

Coverage metrics are used to determine the adequacy of a test and to assess how thoroughly a program is exercised (whether the test promotes high system activation). It was first used in software testing to quantify the capacity of a given input stimulus to activate specific properties of the program code [7]. In hardware, the most common classifications of coverage metrics are as follows: code coverage metrics, metrics based on circuit activity, metrics based on finite-state machines, functional coverage metrics, error- (or fault-) based coverage metrics, and coverage metrics based on observability [2]. Our test generation technique is measured against code coverage metrics. We are targeting error detection (defect coverage) and traditional line (statement) coverage and comparing our approach to random test pattern generation for data.

C. *Specification-based Testing*

We assume that the interpretation by designers may be flawed. This translates to erroneous code. We are targeting this disconnect between the specification and design. [12, 13] are two software survey papers which highlight problems with requirements documents and specification. [12, 13] show that designers often implement their code incorrectly. [14] talks about the ambiguity inherent in specification and software problems associated with this.

III. System Overview

We provide an HDL generator based on scenarios, which describe "typical" behavior already present in the specification and can be derived from timing diagrams, for testbench creation. The output Verliog testbench is simulated to determine code coverage. If there is a change in the specification, modifications to the scenarios can easily be made. This allows customization of the output for direct testing using this proposed method.

Figure 1 shows a general overview of the test generation system. From the specification, a set of scenarios are created. From these scenarios, along with a set of random test vectors for data, a scenario testbench is generated with *n* copies of the original scenario generated testbench code. Our test generation technique that utilizes a modified set of these scenarios (*n* modified or perturbed iterations) also uses a set of random test vectors for data to generate a scenario testbench from the Verilog generator.

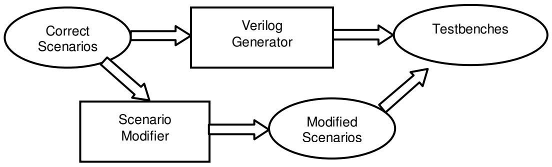

Figure 1. Scenario-Based Modeling (SBM) System Model.

IV. Scenarios

Scenarios describe the behavior of the system and can be modeled in the same way UML scenarios are modeled. Scenarios essentially contain the same information as UML scenarios. Scenarios describe the typical case or behavior that is present in the specification (i.e. not corner cases). These "typical" cases are used to generate variants which model the types of errors we are targeting in our approach. These original and variant cases are then used to generate tests.

To generate tests from scenarios, a verification engineer should describe a sequence of events on all inputs. In other words, this description of a class of input sequences becomes the set of tests for that specified design. A mapping relationship exists between scenarios and functionality specified in the design specification. A scenario describes the most important and common input sequences which map to a functionality within the specification. Input assignment statements (e.g. 'x = 1' and 'y = 0') are atomic units that are assigned to token names (e.g. 'x = 1' is be assigned to 'x_hi' and 'y = 0' is be assigned to 'y_lo'). These eventually build more complex statements using composition operators to connect the tokens.

Additionally, [1] defines symbols, at the lowest level of abstraction (tokens), to describe atomic sets of external signal transitions, while those at higher levels describe arbitrarily complex sequences of these tokens. Production-based specifications (PBS) [1] were described to be more concise and easier to debug and understand due to local nature of each production in the specification. Each production is simultaneously active for all input transitions so that the designer need not worry about the explicit construction of the global control flow, which is necessary in designing a procedural specification. In the same way, scenarios used in SBM are created from atomic tokens to create higher levels of complex sequences.

This work was supported by the National ARCS (Achievement Rewards for College Scientists) Foundation, Inc. under the named Hulings Scholar Fellowship.

V. Detailed Scenario Example

The IEEE 1500 specification [11] is used to expound further a scenario. The IEEE 1500 specification describes a test wrapper component which consists of a Wrapper Bypass (WBY), Wrapper Instruction Register (WIR), and Wrapper Boundary Register (WBR). Figure 6 shows an overview of the IEEE 1500 design. Most system on a chips (SoCs) consist of multiple cores that must interact with one another, and the IEEE 1500 specification describes this system of interaction as well as each individual core's wrapper behavior. This design was implemented in eight modules.

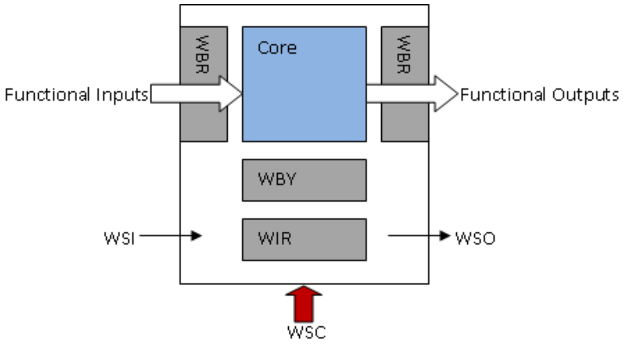

Figure 2. IEEE 1500 Design.

An example of a high-level scenario would be "placing an instruction into the Wrapper Instruction Register (WIR)." To do this, first activate the WIR (using the "SelectWIR" signal). Then, an instruction is shifted in. Finally, the WIR is updated with the instruction shifted in.

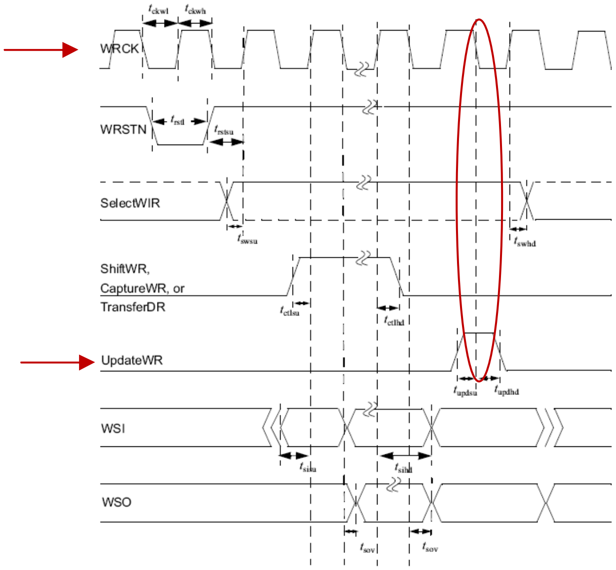

Figure 3. WSP Timing Diagram from IEEE 1500 Specification (with focus on Update).

From the timing diagram in Figure 3, the three steps mentioned for placing an instruction into the WIR can be extrapolated. The control signals are SelectWIR, ShiftWR, and UpdateWR. Each of the three steps is a building block for the main high-level scenario. To demonstrate what a part of the scenario may look like, the third step (update) will be further explained.

Section 14.1 of the IEEE 1500 Specification, Rule b reads as follows (see Figure 4):

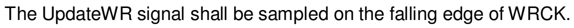
The UpdateWR signal shall be sampled on the falling edge of WRCK.

Figure 4. IEEE 1500 Specification Rule Example.

This rule can be translated into a scenario graph. Figure 5 shows an example of an excerpt of a scenario graph from a mid-level assignment that is built from low-level assignments.

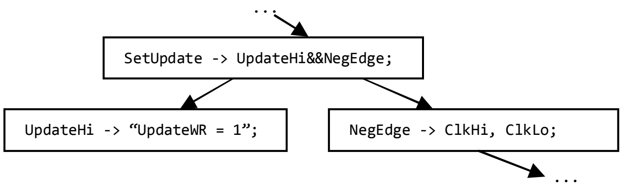

Figure 5. IEEE 1500 Specification Excerpt of Scenario Graph Example

These scenarios are modeled using the production algebra presented in [1]. These are typical expressions in turn derived from standard compiler theory. To derive a scenario from the timing diagram, we built a high-level scenario or event with lower level signal assignments which represent token events. From scenario graphs, such as that mentioned in Figure 5, a testbench is built. Scenarios are built hierarchically from lower-level signal assignments.

VI. Specification Translation Errors

Specification Translation Errors (STE) are misunderstandings of design specifications and are focused on aspects associated directly with the specification. The sequence of behavior with changed events creates a test space around the behavior. The motivation for changing the system with these changed events (i.e. the sequence of events are modified, deleted, or changed in order) models a possible misunderstanding within the neighboring behavior. To model the variances of added or subtracted code due to these possible misunderstandings, the scenarios are modified in order to capture additional potential bugs and catch corner statement coverage which are not detected by utilizing correct scenarios developed from the specification directly.

Three types of errors are modeled:

- Additive
- Modification
- Subtractive

An additive error is one in which the designer adds a dependency that is not specified in the design. In the code, added possibilities in the form of extra code are again not in the specification.

A modification error is one in which a set of signal modifications change the values of the signals (high to low, negedge to posedge and vice versa) as well as operations and control sequences. In the code, these modifications are the product of a misunderstanding of the specification.

A subtractive error is one in which the designer subtracts or leaves out a dependency that is in the specification of the design. Subtractive errors are not present in the code and only present in the specification.

In verifying that the behavior of a design matches the specification, SBM is needed to formalize the specification and help to provide a framework for modeling and generation of tests based on the specification rather than just focusing on the structure of a design. SBM used in conjunction with structure-based metrics will bolster error coverage because errors that do not fall within the rubric of those specific to the design's structure but based on dependencies or specified non-dependencies in the specification will additionally be identified.

These are not errors in the specification but in the translation of the specification. To explain the use of scenario-based modeling for test generation to discover specification-based errors, we propose simple application of the SBM technique to illustrate how scenarios assist in a deeper coverage of error detection for designs. For instance, a blackjack application has a certain set of rules.

One such rule is as follows (see Figure 6):

An ace doubles as a 1 and an 11 (depending on which is most advantageous to the player).

Figure 6. Example of a rule in blackjack.

Coded in its entirety, the rule is as follows (see Figure 7):

```
int handValue (hand) {
...
if (card == ace) {
        // total_value is the computed card value total
        if (total_value > 10) {
                ace_value = 1;
        } else {
                ace_value = 11;
        }
        total_value = total_value + ace_value;
}
}
```

Figure 7. Coded example of a rule in blackjack.

However, if the grayed out part of the specification happens to be left off or not caught by the designer (i.e. this is a missed part of the specification by the designer that may not have been implemented in the code, and it is *not* a problem with the specification itself), the rule and coding drastically change.

If the designer omitted the part of the rule in the specification which stated that an ace could be an 11 (in addition to being a 1), a traditional test generation technique would not catch this omission by the designer in the code because it is an error caused by an omission (not incorrect implementation) of a rule. Achieving 100% coverage will not necessarily reveal a bug such as this. Because of this, generating tests based on scenarios is a good idea because the scenarios which are based on the specification (not code) would not miss this property.

VII. Types of Targeted Errors

We are targeting subtractive errors that can easily happen in designs with large specifications and that cannot be detected without information from the specification itself. For instance, the IEEE 1500 specification is one of the smallest specifications available and is approximately 130 pages in length. There is a lot of room for misunderstandings and mistakes. Also, specifications often change (and so dependencies may be missed. If parts of the specification are implemented by separate groups, this adds a further level of complexity. It is easy for a designer to read a part of the specification out of context and add or leave out an important dependency. Perturbation errors match those errors that one would make due to a misunderstanding of the specification. These errors would model the erroneous from the original because it is an implementation and model of the misunderstanding of the specification. Statement coverage could not assist the verification engineer in identifying this type of error. This motivates why scenario-based modeling will detect these scenario-based errors.

Expanding on the scenario which was derived from the specification (Figure 4), a Verilog testbench (Figure 8) can be derived from the scenario (Figure 5).

```
// Verilog Header
// Additional signals
...
@(negedge WRCK)
UpdateWR = 1;
...
// Verilog Footer
```

Figure 8. IEEE 1500 Specification Scenario to Verilog Testbench Example.

Targeting the three errors which perturb the design in an undesirable way, the following are additive, modification and subtractive errors.

```
...
@(negedge WRCK)
UpdateWR = 1;
ShiftWR = 1;
...
```

Figure 9. Additive Error Injection Example.

Figure 9 is an example of a Verilog testbench excerpt created to detect a particular additive error. The "ShiftWR = 1;" statement is the added signal.

```
...
@(negedge WRCK)
UpdateWR = 0;
...
```

Figure 10. Modification Error Injection Example.

Figure 10 is an example of a Verilog testbench excerpt created to detect a particular modification error. The original "UpdateWR = 1;" statement is modified to be "UpdateWR = 0;".

```
...
@(negedge WRCK)
...
```

Figure 11. Subtractive Error Injection Example.

Figure 11 is an example of a Verilog testbench excerpt created to detect a particular subtractive error. The original "UpdateWR = 1;" statement is subtracted from the Verilog.

The original scenarios were perturbed to create the testbench. In these examples, we are concentrating on control sequences (not data sequences). The data is randomly generated.

VIII. SBM SYSTEM

Scenario-Based Modeling (SBM) techniques expose STE, errors based on a mistranslation of specifications. The SBM system consists of a scenario perturbation generator that generates scenario tests and a Verilog generator that generates the Verilog testbenches.

A. Scenario Perturbation Generator

In the example regarding the blackjack application, a scenario for the rule would be as follows (see Figure 12):

```
handValue -> total_value;
total_value -> ((current_value > 10: ~ace_value) ||
      (current value <= 10:ace_value)): current_value + ace_value;
card_ace = ace;
~card_ace = ~ace;
~ace_value = ACE_LO = 1;
ace_value = ACE_HI = 11;
```

Figure 12. Scenario Example.

Synonymous to the errors targeted are two types of perturbations to the scenarios are as follows. Additive perturbations will include adding variable assignments and branches in a control path. This includes operation perturbations will include changing comparator operations and arithmetic operations. Signal perturbations change values of the signals (high to low, negedge to posedge and visa versa) as well as operations and control sequences. Subtractive perturbations include taking away variable assignments and branches in a control path.

B. Verilog Generator

Just as the PBS compiler of [1], the compiler has been implemented in C code which compiles input specifications containing scenarios (instead of productions in [1]) and outputs testbenches in the form of Verilog code segments (instead of VHDL in [1]). The scenarios are composed of symbols and operators. The symbols are terminal and operators combine sub-machines to create more complex machines. K".

IX. TESTBENCH GENERATION ALGORITHM

Our goal is to create Verilog testbenches from scenarios directly created from the specification of a design. If the scenario is not complete, for example in the case when a signal dependency was not written into the scenario by the verification engineer, the designer should know what the results should be.

A. Algorithm

The algorithm for creating the testbenches using SBM is as follows (see Figure 13):

```
Input:  Timing diagram and/or specification scenario description

1. Generate scenarios.
      1.1. Start from high-level scenario of desired function/operation
      1.2. In breadth-first search (BFS) traversal, build from lower level
           hierarchy of nodes to higher level scenarios to create a scenario
           graph.
      1.3. Continue until leaf nodes (nodes which contain signal
           assignments) are reached.
2. Inject scenarios with additive, modification and subtractive errors.
3. Translate scenarios into Verilog control logic portion of testbench.
4. Add translated scenarios to complete testbench.
5. Add header and footer to testbench.

Output:  SBM testbench
```

Figure 13. Pseudo-code of Algorithm for Creating Testbench.

To create an SBM testbench, the designer starts with the specification. A timing diagram and/or specification scenario description at a high level is extracted from the complete specification to create each scenario until all scenarios are extracted from the complete specification.

For each scenario, the high-level scenario description is derived. To reach a lower level of the scenario hierarchy, the designer gets closer and closer to the leave nodes (which are tokens as described in [1]). Once the leaf nodes have been reached, the scenario graph is completely generated. (Figure 5 shows an excerpt example of this.)

Once the scenario graph is created, a textual representation using the production algebra presented in [1] can be generated for the scenario. Injected in this textual representation of the scenario are the additive, modification and subtractive errors. From this textual representation, an SBM testbench in Verilog code is generated. The additive, modification and subtractive errors should not be confused with those errors which may be erroneously injected into the design by the designer. These errors are translated into test sections for use in the creation of the testbench.

X. EXPERIMENTAL RESULTS

We compared our Scenario-Based Modeling (SBM) method with that of the random method of test generation of data by using the IEEE 1500 specification [11] as our benchmark design with a simple core. Figure 14 shows an overview of the simulation and results comparison. Based on the specification, we made scenario graphs which were translated into behavioral/register transfer level (RTL) Verilog. Our design (excluding the core that was tested within the design) had 779 lines of code (LOC) from the original specification that contained approximately 130 pages. We used our SBM model to generate the SBM testbench using 4 scenario graphs, each containing approximately 30 nodes and approximately 10 of which were leaf nodes. We compared our SBM method with that of a standard random algorithm.

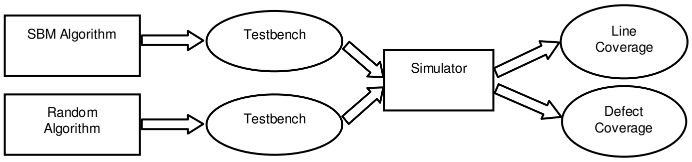

Figure 14. Simulation and results system.

A golden design was compared to 20 erroneous designs (each with one subtractive STE injected into the golden design). The erroneous designs modeled subtractive errors by line deletions and branch deletions. We then computed line (statement) coverage and defect coverage as shown in Table 1. The "**Erroneous Design"** column of Table 1 lists all 20 wrapper versions with errors injected in the design. The next columns compare the "**Line Coverage"** for both "**SBM"** and "**Random"** methods of testing. The "**Defect Detected?"** columns show a comparison of results found for "**SBM"** and "**Random"** methods of testing with respect to whether a defect or error was detected or not detected (a "Yes" in the column denotes that an error was *detected* and a "No" that an error was *not detected*).

The defect coverage results, displayed in Table I, were significantly higher for the SBM method of test generation over that of the random method. For defect coverage, the random method resulted in about 10% coverage while SBM resulted in about 70% coverage.

Contrary to the actual defect coverage results, the line coverage results, also shown in Table I, were showing at least 99.99% for both SBM and random generation methods.

TABLE I. TABLE OF RANDOM VS. SBM TEST GENERATION LINE COVERAGE AND DEFECTS DETECTED

Erroneous Design	Line Coverage		Defect Detected?	
	SBM	Random	SBM	Random
1	100	100	Yes	Yes
2	99.99	99.99	No	No
3	100	99.99	Yes	No
4	99.99	99.99	No	No
5	100	100	Yes	Yes
6	99.99	99.99	No	No
7	100	99.99	Yes	No
8	99.99	99.99	No	No
9	100	99.99	Yes	No
10	100	99.99	Yes	No
11	100	99.99	Yes	No
12	100	99.99	Yes	No
13	100	99.99	Yes	No
14	99.99	99.99	No	No
15	99.99	99.99	No	No
16	100	99.99	Yes	No
17	100	99.99	Yes	No
18	100	99.99	Yes	No
19	100	99.99	Yes	No
20	100	99.99	Yes	No

Comparison of Random vs. SBM Line Coverage and Determination of Defects Detected Results

Figure 15 shows the number of errors detected over the time taken to detect the errors (in simulation clock cycles). The defect coverage (rate of error detection) was derived from the fault coverage over time graphs generated for Figure 15. The complete simulation time for non-detected errors was 179465 cycles (i.e. the length of the entire testbench run to completion).

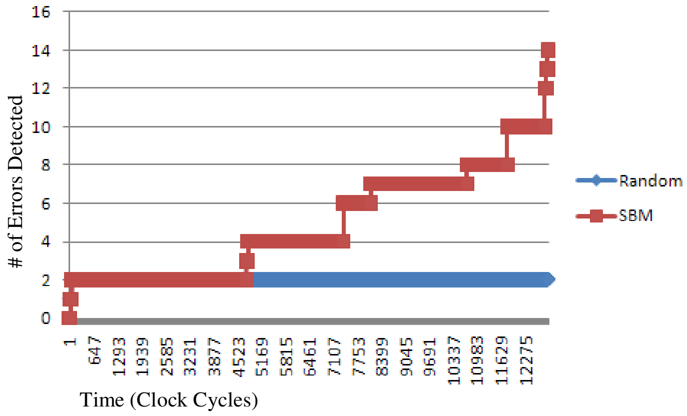

Figure 15. Defect coverage results.

Figure 15 shows the dramatic improvement of errors detected by SBM over the random method of test generation even with line coverage numbers showing 99.99% or above for both methods of test generation.

XI. CONCLUSION

A model and an implementation for hardware verification from scenarios are proposed in this paper. For many types of complex behavior, SBM has advantages over procedural methods which exclude specification analysis. This specification technique can be useful for the hardware verification engineering and joint co-validation techniques.

In this paper, we improved test generation by incorporating traditional coverage targeting techniques with that of our proposed SBM technique to specifically expose subtractive STE. The comparison of actual defect coverage improvements with those assumed using traditional coverage to gain confidence in validation and verification efforts have resulted in misleading information provided by relying solely on coverage details to determine verification and validation confidence in defect coverage. With SBM, subtractive STE were detected at a much higher rate than relying on traditional code based methods which focus on coverage metrics to determine verification and validation confidence.

Future work will evaluate similar techniques used for results checking.

ACKNOWLEDGMENT

We would like to thank the National ARCS Foundation, Inc. and the Hulings Family for support of this work.

REFERENCES

[1] A. Seawright and F. Brewer, “Clairvoyant: A synthesis system for production-based specification.” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 2, June 1994, pp. 172 – 185.

[2] D.K. Pradhan and I.G. Harris, eds., *Practical Design Verification*, Cambridge University Press, 2009.

[3] M.L. Bushnell, V.D. Agrawal, *Essentials of Electronic Testing for Digital, Memory, and Mixed-signal VLSI Circuits*, Kluwer Academic Publishers, 2000.

[4] F. Fummi , C. Marconcini , G. Pravadelli. “Logic-level mapping of high-level faults, Integration”, *the VLSI Journal*, v.38 n.3, p.467-490, January 2005.

[5] P. Prinetto, M. Rebaudengo, M. Sonza Reorda, “An automatic test pattern generator for large sequential circuits based on genetic algorithms,” *ITC94: IEEE International Test Conference*, Washington D. C. (USA), October 1994.

[6] C.H. Cho, J.R. Armstrong, “B-algorithm: A behavioral test generation algorithm,” *International Test Conference*, 1994, pp. 968 - 979.

[7] J.B. Goodenough, S.L. Gerhart, “Toward a theory of testing: data selection criteria,” In R.T. Yeh, ed., *Current Trends in Programming Methodology*, vol. 2, pp. 44-79. Prentice-Hall, Englewood Cliffs, 1977.

[8] P. Mishra, N.D. Dutt, *Functional Verification of Programmable Embedded Architectures: A Top-Down Approach*, Springer, 2005.

[9] M.P. Heimdahl, D. George, R. Weber, “Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria?” Proceedings of the Eight IEEE International Symposium on High Assurance Systems Engineering (HASE), 2004.

[10] “Standard Testability Method for Embedded Core-based Integrated Circuits,” *IEEE 1500 International Standard Specification*, IEC 62528, Ed. 1.0, 2007-11.

[11] IEEE Std 1500, *IEEE Standard for Embedded Core Test— IEEE Std. 1500-2004*. New York: IEEE, 2004.

[12] V. Berzins, C. Martell, Luqui, and P. Adams, “Innovations in Natural Language Document Processing for Requirements Engineering,” *Monterey Workshop 2007, LNCS 5320*, pp. 125—146, 2008. Springer-Verlag Berlin Heidelberg, 2008.

[13] E. Kamsties, D.M. Berry, B. Paech, “Detecting Ambiguities in Requirements Documents Using Inspections”, June 2001

[14] D.L. Parnas, G.J.K Asmis, J. Madey, “Assessment of Safety-Critical Software in Nuclear Power Plants,” *Nuclear Safety* 32(2), pp. 189-198, April—June 1991.

[15] B. Bently, "Validating the Intel Pentium 4 microprocessor." In *Design Automation Converence*, pp 244-248, 2001.

[16] A. Aho, R. Sethi, and J. Ullman. *Compilers - Principles, Techniques and Tools*. Addison Wesley, 1988.

[17] A. Bunker, G. Gopalakrishnan, S. McKee. “Formal hardware specification languages for protocol compliance verification” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, Vol. 9, Issue 1, pp. 1 – 32, January 2004.

[18] K. Shimizu, D. Dill. “Deriving a Simulation Input Generator and a Coverage Metric from a Formal Specification”, *IEEE Design Automation Conference (DAC*), New Orleans, 2002.

[19] C. Plock, B. Goldberg, L. Zuck. “From requirements to specifications*”, *Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems* (ECBS), 2005.

[20] R. Plosch. Contracts, Scenarios and Prototypes – An Integrated Approach to High Quality, Springer-Verlag Berlin Heidelberg New York, 2004, ISBN 3-540-43486-0.

[21] T. A. Alspaugh. “Relationships between scenarios”, *Technical Report UCI-ISR-06-7, Institute for Software Research*, University of California, Irvine, May 2006.