**Center for Embedded Computer Systems**
**University of California, Irvine**

# Message Sequence Charts for Assertion-based Verification

Patricia S. Lee, Ian G. Harris

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2620, USA

{leep, iharris}@uci.edu

CECS Technical Report 13-13
October 30, 2013

# Message Sequence Charts for Assertion-based Verification

Patricia S. Lee and Ian G. Harris
Computer Science
University of California Irvine
Donald Bren Hall, Room 3088
Irvine, USA
Phone: +1 949 824 8842
Fax: +1 949 824 4056
{leep, harris}@ics.uci.edu

*Abstract*— **This paper presents a technique to generate SystemVerilog assertions directly from high-level specification constructs of Message Sequence Charts (MSC) to bridge the productivity gap for current complex designs. Commercial solutions for automated assertion generation do not currently exist. We argue that our technique does span across the hardware/software continuum, allowing it to be applied to both hardware and software components of embedded designs.**

***Keywords-assertion; message sequence chart; verification; simulation; response-checking***

## I. Introduction

Design automation tools have matured to a level accepted and used by academics and industry alike. However, the same level of maturity does not hold for verification. As the complexity and density increases for embedded systems, the need for better, efficient, and automated ways to verify these systems becomes apparent. Although many of the same languages are used for both verification and design, the same structure and logic often does not. This paper focuses on verification which does not have tools that are as well-done and well-defined. Verification methodologies such as Open Verification Methodology (OVM) [11], EVM, and Universal Verification methodology (UVM) [12] are all good efforts to bring verification to the level of structure and logic of design; however, it is difficult to find work on whether this was proven to be implemented in a way which produced verifiable results.

In industry specifications, timing diagrams are commonly included in hardware, protocol, and system requirement specifications. A trend has been to also include higher-level constructs such as Message Sequence Charts (MSCs) in these types of specifications [2], [5], [8] and have been successfully applied in the area of telecommunication systems and the software domain for many years [10]. A way of verifying the correctness of designs through low-level assertions derived from high-level specification constructs such as MSC is one way of bridging the "productivity gap" [1] between the system's design and its complexity.

Our strategy is to apply design automation techniques for modeling to the system-design process for verification and automate the creation of assertions from MSCs and those created from timing diagrams in the specification. In converting MSCs to assertions for functional property and sequence verification, we detect specification-derived translation errors. Although commercial solutions for synthesis and verification at the system levels do not currently exist [1], we argue that our technique does span across the hardware/software continuum.

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

## II. Previous and Related Works

Assertion-based verification has been in practice for almost two decades in software [17] and just over a decade in hardware [15]. Several benefits exist for using assertions in the verification process. A few important benefits are that it is close to potential bug sources, automatically checks behavior, forces the documentation of design intent and encourages a better understanding of the design, and allows for focus on system-level issues.

Some disadvantages to using assertions are that we cannot know whether the property is 100% true, the property set may not be sufficient and complete, and a property might miss design behavior nuances (corner cases). For this reason, test stimulus is critical, and simulation can only provide checks that the test exercise and that coverage metrics reveal.

### A. *Automated Assertion-based Verification in Software*

Previous research [17] provides an approach of automated test data generation in which assertions are used to generate test cases. A test is found that uncovers specific test cases on which an assertion is violated. An empirical study on the "Inadequate-Assertion" problem in the context of automated test suites developed for open-source projects has been done on the test suites of three active open-source projects with a performance of mutation analysis and coverage analysis for evaluation [19].

### B. *Automated Assertion-based Verification in Hardware*

Goldmine [6] is a tool that automatically creates assertions based on simulation traces using machine learning techniques. The assertion is presented to the user (design/verification engineer) to determine whether the assertion is a good candidate to make into the design or whether the test should include additional cases to put the design in that state via a new scenario case. The main challenge is in the difficulty in learning all the sequences over time. Goldmine [6] addresses this challenge by focusing on one instance of time. We argue that our technique fills this need for covering sequence over time and is consequently complementary to the work of Goldmine. Another work that works with automatic assertion extraction at the input boundary of a given unit embedded in a system proposes a data mining approach that analyzes simulation traces to extract the assertions [20]. Our work is at a different level of abstraction and is also automated.

Reference [18] presents a methodology that uses the failing assertion, counterexample and mutation model to produce alternative properties that are verified against the design and serve to make possible corrections as they provide insight into the design behavior and the failing assertion. The results show that this process is effective in finding high quality alternative assertions for empirical instances. However, the process is not yet automated.

## III. System Overview

We present a system which generates System Verilog Assertions (SVAs) [14] that determine a sequence of events, recognize this sequence, and make an assertion based on the state of the system. Each assertion checks the state of the system via the registers to verify its correctness after detecting a sequence. We apply design automation techniques for modeling to the system-design process for verification to automatically generate SVAs from MSCs that are part of the specification description.

Fig. 1 shows a general overview of the system. The Specification-based Testbench Generation (SBTG) tool [7] provides a functional testbench. The testbench applies test vectors to the device under test (DUT) during simulation. The response checker uses assertions (SVAs), generated from the specification using MSCs, to verify the correctness of the output from the DUT.

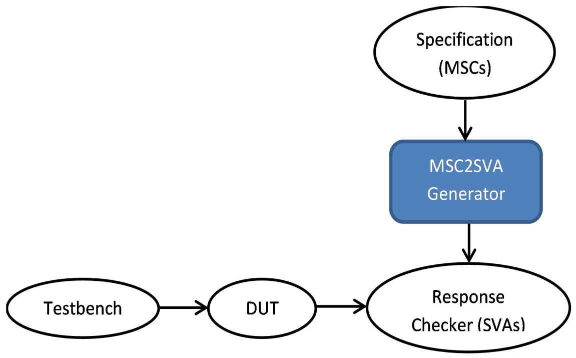

Figure 1. MSC2SVA system overview.

## IV. Message Sequence Charts

MSCs are in a graphical language that is standardized by the International Telecommunication Union (ITU). In recent times, MSCs have also found their way into hardware specifications due in part to the higher level of abstraction and description required to describe these increasingly complex systems [2], [8]. Because MSCs are viable and used in specifications to illustrate high-level transactions and communications at the package/transaction levels and because MSCs are a standard [3], [4], [5] which is supported by graphical and textual captures of this model, we believe it is an ideal candidate for automation. Since we are looking at hardware/software codesign and verification in embedded systems research, we further bolster our argument for the use of MSCs.

MSCs have long been used in the software domain and are trending to be used in the hardware domain. We bridge this high-level trend with lower-level assertions used in hardware response checking monitors.

Here we provide an example of an MSC that was derived from a timing diagram. Fig. 2 shows the Wrapper Serial Control (WSC) of the IEEE 1500 specification [13] which describes a test wrapper component consisting of a Wrapper Bypass (WBY), Wrapper Instruction Register (WIR), and Wrapper Boundary Register (WBR). The WSC are the main inputs that control the system. The Wrapper Serial Input (WSI) and Wrapper Serial Output (WSO) provide the serial interface to the design.

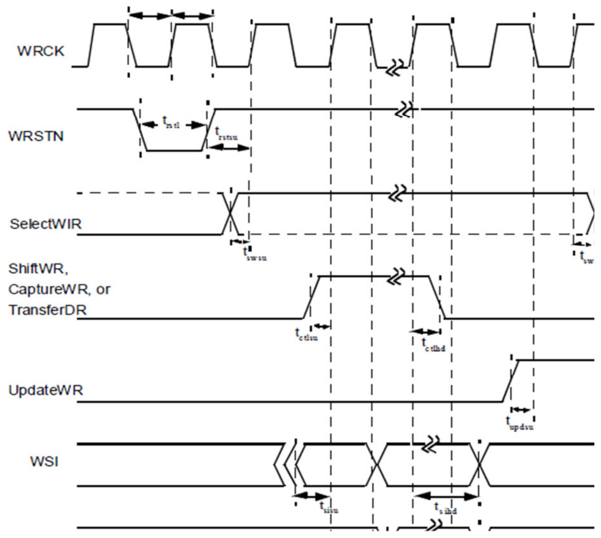

Figure 2. WSP Timing Digram from the IEEE 1500 Specification..

Fig. 3 is an example of an MSC representing the instruction BYPASS from the IEEE 1500 specification timing diagram shown in Fig. 2. In this example, the instruction register (WIR) contains “00” which represents the BYPASS instruction. From the timing diagram, we see that as SelectWIR goes high, two 0’s are shifted in from WSI to the WIR as ShiftWR high and CaptureWR, TransferDR, and UpdateWR are low.

For each component, we create instances of components in the MSC (blocks at top of Fig. 3) and draw out lifelines with termination blocks at the bottom of the MSC. The WRCK is the lowest level of timing granularity and is represented as time steps l = 0 to 17 in Fig. 3. It determines the assertion clock and is a positive edge from WRCK = 0 to 1 (denoted as "@(posedge WRCK_ip)" in Fig. 5) and a negative edge from WRCK = 1 to 0.

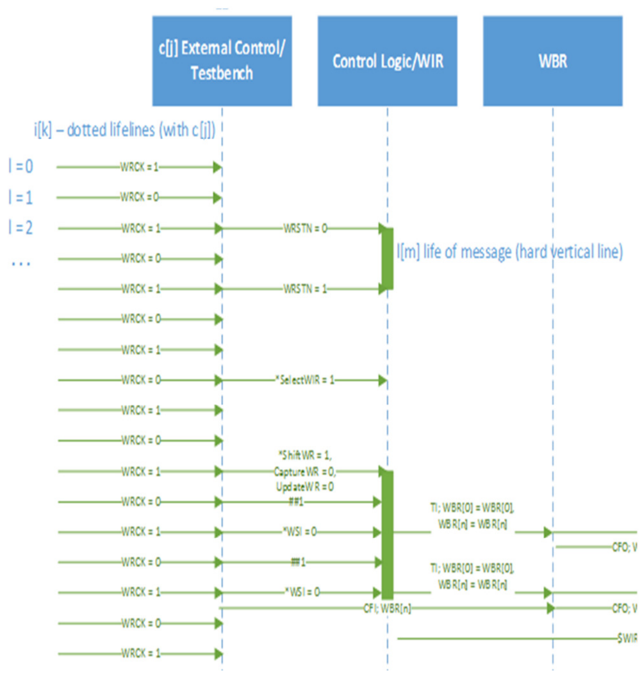
c[j] External Control/ Testbench
Control Logic/WIR
WBR
i[k] – dotted lifelines (with c[j])
l = 0
l = 1
l = 2
...
WRCK = 1
WRCK = 0
WRSTN = 0
l[m] life of message (hard vertical line)
WRSTN = 1
*SelectWIR = 1
*ShiftWR = 1, CaptureWR = 0, UpdateWR = 0
##1
*WSI = 0
TI; WBR[0] = WBR[0], WBR[n] = WBR[n]
CFO; V
CFI; WBR[n]
$WIR

Figure 3. Excerpt of MSC of IEEE 1500 Bypass operation.

The MSC is converted into text using a tool called Mscgen [16] which parses MSC descriptions and produces from the textual representation of the MSC shown in Fig. 4 from the graphical MSC shown in Fig. 3.

```
# bypass
msc {
 arcgradient = 8;
#instances
 a [label="ExternalCtrlTB"] , b [label="WIR"] , c [label="WBR"] , d
[label="CoreLogic"], e [label="ExternalCtrlMon"];
#messages
#wrck 1
#wrck 2
#wrck 3
a=>b
[label="*(SelectWIR_ip&ShiftWR_ip&~CaptureWR_ip&~UpdateWR_ip
)"];
#wrck 4
 a=>b [label="##1"];
#wrck 5
 a=>b [label="*(~WSI_ip)"];
#wrck 6
 a=>b [label="##1"];
#wrck 7
 a=>b [label="*(~WSI_ip)"];
#wrck 11
 |||;
#wrck 10
 a<=b [label="$WBR_OP_IN_ip==2'b0"];
#end
}
```

Figure 4. Textual representation of the MSC of the IEEE 1500 BYPASS operation.

## V. Assertions

Assertions act as constraints that determine and define legal and expected behavior when blocks interact with each other. We concentrate mainly on converting MSCs to concurrent assertions that are represented as the MSCs are as sequentially.

The layers of a concurrent assertion build from a Boolean expression to a sequence to a property to the top-most layer of the abstraction for concurrent assertions which is the assertion directive layer where a property is associated with a specific block of code with a clear intention of its instantiation [21]. Figure 5. shows an assertion created using the MSC2SVA tool we have developed. Comments have been added manually.

```
    //==================================================
    // bypass
    //==================================================
    module assertion_ip(input wire [7:0] a_ip, b_ip, sum_ip,
            input wire WSI_ip, WSO_ip, WRSTN_ip, ShiftWR_ip,
UpdateWR_ip, CaptureWR_ip, SelectWIR_ip, WRCK_ip,
            input wire [1:0] WBR_OP_IN_ip, WBR_OP_OUT_ip);
    //==================================================
    // Sequence Layer
    //==================================================
    sequence  bypass_seq;
    (SelectWIR_ip&ShiftWR_ip&~CaptureWR_ip&~UpdateWR_ip)##1(~
WSI_ip)##1(~WSI_ip);
    endsequence

    //==================================================
    // Property Specification Layer
    //==================================================
    property  bypass_prop;
    @(posedge WRCK_ip)
    disable iff(~WRSTN_ip)
     bypass_seq|=>WBR_OP_IN_ip==2'b0;
    endproperty

    //==================================================
    // Assertion Directive Layer
    //==================================================
    bypass_assert: assert property( bypass_prop)
    else
    $display("@%0dms bypass assertion failed", $time);
    endmodule
```

Figure 5. Excerpt of assertion generated from MSC of IEEE 1500 BYPASS operation.

The sequence which puts the WIR into BYPASS mode begins the sequence layer of the assertion (the antecedent of our assertion property). This is represented by the signals SelectWIR_ip and ShiftWR_ip going high while CaptureWR_ip and UpdateWR_ip go low. After one clock cycle, the WSI_ip is low, causing the value '0' to be shifted into the WIR. After another clock cycle, the WSI_ip is low again, causing another '0' value to be shifted into the WIR. This brings the wrapper to BYPASS mode with the WIR containing the value '00' which represents the BYPASS instruction in the design. The consequent of the assertion is the check that the appropriate instruction was entered into the WIR instruction register.

## VI. MSC2SVA System

The MSC2SVA program compiles input specifications in the form of MSCs and outputs assertions in the form of System Verilog. The SVA is completed with a header and footer added automatically by the program. The binding file to associate the assertion component block with the design block is created manually. The binding file and assertion file is also included manually in the testbench file. The program is implemented with the SVA generation algorithm presented in section VII as a Python scripted program.

After the SVA is run with the correct design, it is checked with the incorrect design to see whether the assertion is triggered. Several different errors could have triggered the assertions; however, we elected to stop after the first error triggered the assertion.

## VII. SVA Generation Algorithm

We address the aspect of managing ordering of events via the smallest measure of time in the logic design. Time advances via this clock and are represented as row numbers in the algorithm.

The input is a set of MSCs, and the output is derived from the algorithm mentioned in this section under Subsection B.

### A. Inputs

The input is a set $M$ of MSCs $m[i]$, where $I$ = number of elements in $M$ and $i$ = index of element in $M$, each with the following:

*1) Component instances:* A set $C$ of boxes representing physical/system component instances $c[j]$, where $J$ = number of elements in $C$ and $j$ = index of element in $C$. Component instances represent structural components within the design. They can send and receive signal messages which may trigger additional events to occur.

*2) Lifeline instances:* A set $I$ of dotted vertical lines representing lifeline instances $i[k]$ where $K$ = number of elements in $I$ and $k$ = index of element in $I$. (Note: $j = k$ and $J = K$ as each lifeline instance corresponds to exactly one component instance). Lifeline instances designate how long a component is active in the MSC. The virtical dotted line starts with a component instant and ends with a terminal character representing the end of the component's function life with respect to the MSC.

*3) Signal messages:* A set $S$ of horizontally-lined arrows representing signal messages $s[l]$ with text $t[l]$, where $L$ = total time progression represented as a total number of **rows** of an elements in all $S$ (i.e. the last time tick for all instances represented in $m[i]$) and $l$ = index of the **row** of an element in $S$ (Note: All rows span across all $S$). Signal messages are messages sent via wires or buses from one component to another. They determine what signals trigger certain events in time.

*4) Terminal characters:* A set E of boxes (at the end of each lifeline instance i[k] in I) representing terminal characters e[k]. Terminal characters denote the end of the life of the component's function with respect to the MSC. The last terminal character on the far right and associated with the component on the far right is the terminal character of the entire MSC life.

*5) Lifeline of messagess:* A set $L$ of vertical bars representing the life of the message $l[m]$, where $M$ = number of elements in $L$ and $m$ = index of element in $L$. Lifeline of the messages represent how long the signals in the message should be either set high or cleared low.

*6) Symbols:* A set of symbols within the message label denote whether the message causes an event (i.e. is an antecedent) or is affected by a previous message or event (i.e. is a consequent).

*a) *:* is an antecedent.

*b) \$:* is a consequent.

### B. Outputs

The output is a set $A$ of assertions written in System Verilog and created from each $m$ in $M$ MSC list:

```
1.  for each m[i] in M
2.    for each l until L
3.      for each i[k] in I
4.        if e[L] in i[K],  // last terminating block
    (e[l]) in last lifeline instance (i[k])
5.          then return
6.        if not t[l] of i[k],  // if there is no text t[l]
7.          then increment k
8.        if initial * in t[l] of i[k],
9.          then start antecedent of assertion 'if ('
    and
10.         then add to antecedent of assertion and
11.         then increment k
12.       if not initial * and * in t[l] of i[k],
13.         then add "AND" operation '&&' to
    antecedent of assertion and
14.         then add to antecedent of assertion and
15.         then increment k
16.       else if not * in t[l] of i[k], // i.e. end of the
    antecedent
17.         then end antecedent of assertion ')'
18.         // do not increment k
19.       if $ in t[l] of i[k],  // this is current k (for
    consequent/checker construction)
20.         then consequent checker and
21.         then increment k
22.
23.       if i[K],  // i.e. k = K
24.         then increment l and
25.         then initialize k
26.         then
```

```
27.         if * in t[l] of i[k],
28.             then add time increment '#1' to antecedent of assertion and
29.             then add to antecedent of assertion
30.     // end for each i[k] in I loop
31.   // end for each l until L loop
32. // end for each m[i] in M loop
```

This output algorithm was implemented in the Python script for the MSC2SVA program. For each MSC, the algorithm starts at the top left corner of the diagram and follows the horizontal signal messages from left to right until the last signal message is reached. Then, the next row of messages is evaluated (moving down one row). Again, the horizontal messages along the new row are evaluated from left to right until the last signal message is reached in that row.

These steps are repeated until the last termination character on the last block is evaluated (farthest right, bottom corner of the MSC). As each signal message is evaluated, the symbols "*" and "$" determine whether the signal is part of the antecedent or consequent of the concurrent assertion sequence and properties.

## VIII. Experimental Results

The results demonstrate that our MSC2SVA technique generates SVAs that are effective in detecting errors. The results demonstrate the possible automation of SVA creation from MSCs and from timing diagrams that can be converted into MSCs.

We implemented a simple version of the IEEE 1500 wrapper that does not include optional instructions or parallel instructions. Our IEEE 1500 design has 779 lines of Verilog code. The SVAs generated for the IEEE 1500 specification contained 3 assertions each approximately 30 lines. Each SVA was run first with the correct design and then with the design containing an error which would trigger the assertion. The types of errors included mainly bit flipping errors; however, the errors specified in [7] and [22] could also be utilized.

MSC2SVA is implemented as a Python script which was executed on a personal computer with Intel® Core™ i5-2450M CPU @ 2.50 GHz with 4.00 GB of RAM with a 64-bit Operating system running Windows 7 Home Premium.

The three instructions are as follows:

### A. *Instruction WS_BYPASS (BYPASS)*

The mandatory WS_BYPASS instruction enables the functional configuration of the wrapper. WS_BYPASS is selected when no test operation of that core is required and allows only the WBY to be selected. The WBY provides a minimum-length serial path between the wrapper's WSI and the WSO. This allows more rapid movement of test data to and from other core wrappers, provided the wrappers are connected serially [13].

### B. *Instruction WS_INTEST (INTEST)*

One core test instruction that allows the core to be tested according to a test procedure specified by the core provider or core user is required. IEEE Std 1500 does not describe how to test individual cores; this is the responsibility of the core provider. The core test invoked by the Wx_INTEST instruction (the x in Wx is a place holder for an S, P, or H to indicate whether the instruction is serial, parallel, or hybrid) is completely specified with the CTL provided for the core [13].

### C. *Instruction WS_EXTEST (EXTEST)*

The mandatory WS_EXTEST instruction allows testing of off-core circuitry and core-to-core interconnections. It allows circuitry external to the core wrapper, typically the interconnects and user-defined logic (UDL), to be tested. The wrapper boundary cells at WFOs are used to apply test stimuli, while the cells at wrapper input terminals capture test results. This instruction also allows testing of blocks of UDL between cores that do not themselves incorporate wrappers [13].

The three instruction assertions performance and memory measurements are shown in Table 1.

TABLE I. Assertion List

| No. | Assertions | | |
|---|---|---|---|
| | ***Assertion Name*** | ***Performance*** | ***Memory**** |
| 1 | Instruction WS_BYPASS | 0.073s | 8MB |
| 2 | Instruction WS_INTEST | 0.076s | 8MB |
| 3 | Instruction WS_EXTEST | 0.080s | 8MB |

*Memory includes overhead for Python infrastructure.

The assertions sequenced the selection of each instruction. A property was created from the sequence, which acted as the antecedent, and the resulting check of the instruction register, which acted as the consequent of the assertion property.

An error was injected into the design in order to trigger the assertion. For each of the instructions, an alternate value was provided in the WIR. Multiple errors might have triggered the assertions, but we tested only with one error to trigger each assertion.

Additional assertions could be made from these wrapper instructions that verified specific behavior with respect to the core (device under test for this wrapper). Also, assertions specific to the operations of the wrapper (namely, shift, update, transfer, and capture) could be made and evaluated using this method.

## IX. Conclusion

In this paper, we automate the response checking process by generating SVAs directly from MSCs representing the design specification. Our MSC2SVA technique can be used together with both an automated testbench generation technique and coverage-based test generation to detect errors. This research represents an effort to automate the traditionally manual process of response checking from high-level specification constructs. Our results have shown that errors can be detected utilizing this method of assertion generation.

## Acknowledgment

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

## References

[1] D. Gajski, S. Abdi, A. Gerstlauer, G. Schirner. Embedded System Design: Modeling, Synthesis, and Verification, Springer, 2009.

[2] Information technology—Serial Attached SCSI -3 (SAS-3) Specification by the Working Draft American National Standard Project T10/BSR INCITS 519, http://www.t10.org/drafts.htm#SCSI3_SAS, http://www.t10.org/cgi-bin/ac.pl?t=f&f=sas3r05b.pdf

[3] ITU-TS, Recommendation z.120, message sequence chart (msc), Technical Report Z. 120, International Telecommunication Union, Geneva.

[4] ITU-TS, Recommendaton z.120 annex b: algebraic semantics of message sequence charts, and recommendation z.120 annex c: syntax requirements of msc, Technical Report Z.120 B,C, International Telecommunication Union, Geneva, 1995.

[5] E. Rudolph, P. Graubmann, J. Grabowski, Tutorial on message sequence charts, Computer networks and ISDN systems 28 (12) (1996) 1629–1641.

[6] S. Hertz, D. Sheridan, S. Vasudevan. "Mining hardware assertions with guidance from static analysis." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2013.

[7] P. Lee, I. Harris, "Test generation for subtractive specification errors," IEEE VLSI Test Symposium, 2012.

[8] A. Ito, H. Saito, F. Nitta, Y. Kakuda, "Transformation technique between specification in SDL and specification in message sequence charts for designing protocol specifications" Communications, 1992. ICC '92, Conference record, SUPERCOMM/ICC '92, Discovering a New World of Communications., IEEE International Conference on 14-18 Jun 1992, pp. 442 - 447, vol.1.

[9] P. Murthy, S. Rajan, K. Takayama, "High level hardware validation using hierarchical message sequence charts," IEEE International High Level Design Validation and Test Workshop, Sonoma CA, Novemeber 2004.

[10] S. Mauw, M. Reniers, T. Willemse, "Message sequence charts in the software engineering process," Handbook of Software Engineering and Knowledge Engineering, S.K. Chang, editor. World Scientific, 2001.

[11] Cadence Designs Systems and Mentor Graphics Inc., "Open Verification Methodology User Guide" Version 2.0, Sept. 2008 available from http://www.ovmworld.org.

[12] Standard Universal Verification Methodology developed by the VIP Technical Committee available from http://www.accellera.org/downloads/standards/uvm

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

[14] System Verilog 3.1 – Accellera's Extensions to Verilog, Accellera, May 29, 2003.

[15] K. Chen, "Assertion-based verification for SoC designs." ASIC, 2003. Proceedings. 5th International Conference on, pp. 12 – 15, vol.1.

[16] Mscgen from http://www.mcternan.me.uk/mscgen/

[17] B. Korel, A. Al-Yami, "Assertion-oriented automated test data generation," Software Engineering, 1996., Proceedings of the 18th International Conference on, pp. 71-80, 1996.

[18] B. Keng, S. Safarpour, A. Veneris, "Automated debugging of SystemVerilog assertions." Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011.

[19] J Zhi, V. Garousi, "On Adequacy of Assertions in Automated Test Suites: An Empirical Investigation." Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on, pp. 382 - 391, 2013.

[20] P. Chang, L. Wang, "Automatic assertion extraction via sequential data mining of simulation traces." Design Automation Conference (ASP-DAC), 15th Asia and South Pacific, pp. 607-612, 2010.

[21] System Verilog Assertion Tutorial from Project-VeriPage.com: http://www.project-veripage.com/sva_3.php

[22] S. Verma, P.S. Lee, I.G. Harris, "Error Detection Using Model Checking vs. Simulation," IEEE International High-Level Design, Validation and Test Workshop, Nov. 2006.