Automated Formal Verification of X Propagation with Respect to Testability Issues

Mehdi Dehbashi*‡ Daniel Tille† Ulrike Pfannkuchen† Stephan Eggersglüß*‡
*Institute of Computer Science, University of Bremen, Bremen, Germany
‡ Cyber-Physical Systems, German Research Center for Artificial Intelligence (DFKI), Bremen, Germany
† Group of Design-For-Test (DFT), Infineon Technologies AG, Munich, Germany

***Abstract*—X values may be captured by scan flipflops during the scan test. An X value corrupts the signature generated by a *Multiple-Input Signature Register* (MISR). The MISR is used in the test structures such as *Logic Built-In Self-Test* (LBIST). In this paper, we propose an approach to automate formal verification of X propagation with respect to testability issues. The propagation of an X value from X sources to scan flipflops is comprehensively evaluated using formal verification considering all possible test patterns. The approach is utilized to find root causes of a corrupted signature generated by MISR and to rectify the erroneous behavior of a design because of dangerous X sources.**

***Keywords*—Formal Verification, Testability, Dangerous X Source, Observation Point**

I. Introduction

During the scan test, X values may propagate from X sources to observation points (scan flipflops). An X value may propagate from the output of a black box, a non-scan flipflop, a latch or other sources. Non-scan flipflops and latches are utilized in the design in order to reduce the area overhead and to decrease the power consumption of the IC. If the value of non-scan elements is not corretly initialized in the test setup, an X value may propagate to scan flipflops during the scan test. Also an X value may propagate from the output of a black box during the scan test, if the black box is not correctly isolated from the rest of the circuit. An X source which propagates an X value to scan flipflops during the scan test is called a *dangerous X source*.

An X value must be prevented to propagate to scan flipflops. Otherwise, the signature generated by MISR is corrupted and cannot be used to detect faults in LBIST structures. An X value also has negative effect on the compression rate and the test coverage in *Embedded Deterministic Test* (EDT) structures [1]. In this case, when there is an X value in the response data, an X masking method is used to prevent a captured X value to appear in the compacted response. However, X masking decreases the quality of test patterns to detect more faults, i.e., the number of test patterns increases.

Three-valued logic (01X) has been used to abstract a circuit for efficient model checking in [2]. The work in [3] uses a heuristic approach based on three-valued logic to find high quality counterexamples in order to debug logic bugs. X propagation is utilized in [4] to generate diagnostic traces in order to increase the diagnosis accuracy of design debugging. The work in [5] utilizes X propagation in order to determine cases of verification scenarios and, consequently, to improve the coverage in simulation-based verification.

In this paper, we propose an automated approach to formally verify X propagation to improve testability of digital circuits. At RTL, dangerous X sources are found and the design is improved preventing an X value to originate and to propagate to observation points. However, the scan structures are not available at RTL. Therefore, the exact verification cannot be performed at RTL to determine dangerous X sources with respect to testability issues. At gate-level, formal verification of X propagation is automated in order to determine dangerous X sources. We automate formal verification by automatically generation of suitable constraints and assertions for a given list of X sources and observation points (Section III-A and Section IV-A).

Using formal verification, the behavior of an X source is comprehensively evaluated with respect to all possible test patterns. An X value propagated from a dangerous X source corrupts the signature generated by MISR. Using our approach, root causes of a corrupted signature are found and are localized. This information is utilized by the designer to debug and to isolate the circuit from dangerous X sources. By this, the debugging process which typically has a large share in the design process can be significantly accelerated.

The remainder of this paper is organized as follows. In Section II, the overall design flow is explained. We describe X sources and a process to improve a design preventing X propagation at RTL in Section III. Section IV deals with dangerous X sources for scan test and isolation of dangerous X sources at gate-level. In this section, our approach to automate formal verification of X propagation is explained. Our approach to improve the efficiency of formal verification is demonstrated in Section V. Section VI presents experimental results on benchmark circuits. The last section concludes the work.

II. Design Flow

The VLSI system design methodology starts with a system design team writing the specification of the system as a text. Then, the system model is implemented and the concepts and the algorithms at the system level are verified [6]. The common

The work of S. Eggersglüß has been supported by the Institutional Strategy of the University of Bremen, funded by the German Excellence Initiative.

languages to describe the system model of a hardware are C, C++, and SystemC.

After validating the specification of an IC, the functions of the specification are implemented by *Hardware Description Languages* (HDL) such as Verilog and VHDL. The hardware functions are implemented at *Register Transfer Level* (RTL) using HDLs.

At RTL, the design is verified to find X sources. X values originate in a design because of a bug in the implementation, incompleteness of the implementation, uninitialized memory elements or black boxes. X sources at RTL are discussed in Section III in detail. X sources can be found by formal-based or simulation-based verification. Using the formal verification, the behavior of a design for all possible input stimuli can be evaluated and verified. However, the scalability of the formal verification is limited. For large SoCs, the simulation-based verification can be used to evaluate the behavior of the design under a specified set of input stimuli. In this work, we focus on the formal verification to determine dangerous X sources. After determining dangerous X sources, the RTL design has to be improved to prevent X propagation. In Section III-B, solutions to prevent X propagation at RTL are discussed.

After verifying the design at RTL, logic synthesis is performed. Logic synthesis converts a design into a gate-level circuit. At gate-level, also *Design-For-Test* (DFT) structures such as scan chains and LBIST are inserted in the design. New X sources may also appear at gate-level as the exact behavior of signals and delay information are available only at this level. Also, an inconsistency created by the synthesis tool may create an X source at gate-level. The exact formal verification of X propagation with respect to testability issues can be performed at gate level because scan flipflops are available at this level. In Section IV-B, we present an automated process to evaluate whether X sources are dangerous at gate level. The formal verification of X propagation is automated considering the effect of X propagation to scan flipflops. We also introduce an automated approach to improve the efficiency of the formal verification of X propagation in Section V.

The transistor level design is created by a place-and-route process for chip manufacturing. Then, the design is fabricated on silicon as a chip. The process to validate a fabricated chip is called *post-silicon validation*. The post-silicon validation process starts by applying test patterns to the IC or by running a test program, such as end-user applications or functional tests, on the IC [7] [8]. The test patterns are applied to the IC using DFT structures such as LBIST. A signature is generated for responses using a MISR in LBIST. When there is no X value in the response, the correct generated signature is utilized to detect physical faults in the IC.

III. X-Sources at RTL

We categorize X sources at RTL as follows:

1) Uninitialized memory elements and signals
2) Bug in the implementation
3) Incompleteness of the implementation
4) Black boxes

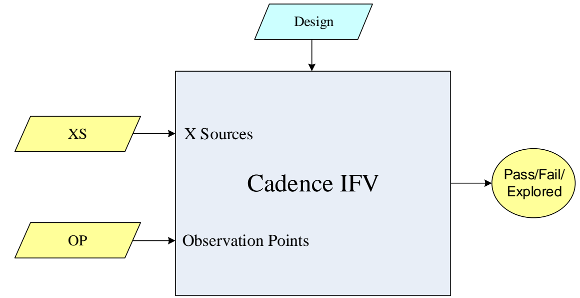

Fig. 1. Formal verification of X propagation at RTL

5) Registers with chip-specific values

The first category includes uninitialized memory elements such as flipflops, latches, registers and memory modules. Also, an X value may originate because of a bug in the implementation for example division by zero, index violation or function not returning a value [9]. Incompleteness of the implementation is another cause to originate an X value. One common example for this category of X sources is an incomplete case statement [9] [10]. As the value of output ports of a black box is unknown, these output ports originate X values. A black box can be an analog module which is not considered in the digital test process. Registers with chip-specific values contain information of each individual fabricated chip. After fabricating ICs, these registers contain different values in different chips. Therefore, registers with chip-specific values are another source of X values which may cause generating different signatures in different chips by MISR. These registers may also have different values in different runs of a same chip (e.g. adjust values).

Formal-based and simulation-based verification can be utilized to detect X sources. As an initial step, a design can be analyzed to detect possible X sources [9]. X sources of categories 1, 4 and 5 are easily detectable. However, X sources of category 2 and category 3 are more difficult to be detected because these categories are potential X sources. In this case, a reachability analysis is required to prove that these X states are reachable.

If the set of possible X sources is large, the designer's information is utilized to find out which X source is a real X source. For example, some uninitialized memory elements detected at module level are initialized when the module is used in a system and in a real testbench.

After finding X sources, another issue is to determine whether an X source is dangerous. An X source is dangerous when an X value originates at that source and propagates and arrives at observation points. At RTL, observation points can be for example state elements (memory elements) and primary outputs. Formal verification approaches are utilized to analyze X propagation to observation points [9] [11] [12]. For analysis of X propagation, three-valued logic is used in which each signal can have the value 0, 1, or X (unknown). This logic has been used in the field of formal hardware verification for creating strong counterexamples [13] [14] [3] and faster verification engines [15] [16] [12].

A. Automated Formal Verification of X Propagation at RTL

Figure 1 shows our approach to formal verification of X propagation at RTL. In our approach, we utilize Cadence *Incisive Formal Verifier* (IFV) [11] as an underlying formal engine. Our approach automatically verifies X propagation from X sources to observation points and, consequently, determines whether X sources are dangerous. As Figure 1 shows, the list of X sources (XS) and observation points (OP) are given to the tool. Then constraints are automatically generated to constrain the value of X sources to X by our approach. Also assertions are automatically generated to observe the behavior of observation points. An assertion specifies that an observation point has to capture no X value. Having constraints and assertions, formal verification evaluates whether an X value can propagate from X sources to an observation point. The result of formal verification is: *pass*, *fail* or *explored*. The result *pass* means the formal verification has proven there is no possibility to propagate an X value from X sources to the corresponding observation point. In the case of the result *fail*, a counterexample is generated. The designer uses the counterexample to correct the design in order to prevent an X value to originate and to propagate. The result *fail* indicates some X sources are dangerous for the corresponding observation point. When an assertion fails, the active X sources (dangerous X sources) which have contributed to X propagation to the corresponding observation point are reported by running the command "*debug assertion_name*" in the IFV console [11]. When the state of an assertion is *explored (n)*, it indicates the tool cannot prove that the corresponding assertion passes or fails within the specified effort (run-time and memory). However, the tool proves that there is no counterexample till the clock cycle n.

B. X-Prevention at RTL

After determining dangerous X sources, they have to be prevented to propagate an X value to observation points. Memory elements have to be initialized to prevent dangerous X sources of category 1 mentioned in Section III. An X source of category 2 indicates a bug in the implementation. In this case, a counterexample is generated by the verification tool. The counterexample shows the erroneous behavior of the implementation. This counterexample is given to the designer in order to debug the implementation and to fix the erroneous behavior.

When an X value propagates to observation points because of an incomplete section in the implementation (category 3), the verification tool generates a counterexample (a trace). The counterexample shows how the incomplete section of the implementation is activated and how an X value from the activated section propagates to observation points. This counterexample is given to the designer in order to complete the implementation.

In the case of a black box and registers with chip-specific values as dangerous X sources, they have to be isolated from

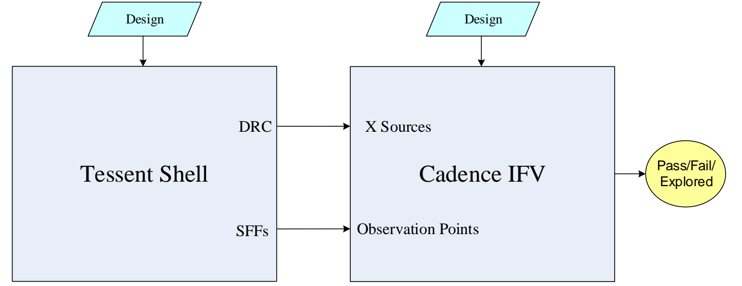

Fig. 2. Formal verification of X propagation at gate-level

the rest of the circuit in the test mode[1]. Scan test signals are required at RTL to isolate dangerous X sources. In summary, the solutions for X prevention at RTL are as follows:

1) Initializing memory elements
2) Fixing the X-related bug
3) Completing the implementation
4) Isolating dangerous X sources

IV. X-Sources at Gate-Level

In addition to remaining X sources from RTL, new X sources may appear at gate-level as delay information and exact behavior of signals are available at this level. At gate level we consider X sources in the scan test mode. An X value at gate-level may originate because of the following reasons [17]:

1) A violation on a wired gate
2) A violation on a bus gate
3) A violation on a tri-state gate or a switch gate
4) A violation on a transparent latch
5) A violation on a ROM or RAM gate
6) Tie-X non-scan flipflops and latches
7) Init-X non-scan flipflops and latches
8) Primary inputs which are not used as scan pattern input

If some wires are directly connected to each other, a wired gate is constituted. In this case, if the wires are derived by different values, an X value originates. In the case of a bus gate, if more than one of the bus-connected tri-state drivers or switches turn on simultaneously or all drivers turn off simultaneously (Z state), an X value originates [17]. If a tri-state driver gate or a switch gate does not connect to a bus gate, this Z state behaves as an X [17]. Also violations on input ports of a transparent latch and ROM/RAM may originate an X value. Tie-X non-scan flipflops and latches have always an X value. One example for this class of X sources is registers with chip-specific values which have different values in different ICs. Init-X non-scan flipflops and latches are state elements which are uninitialized at the start of scan test. External inputs and bidirectional pins which are not used as scan pattern input can also be X sources. For more details we refer the reader to [17].

[1]For a register with chip-specific value, not only the output port but also the input ports of the register have to be isolated. The input ports of a register with chip-specific value are isolated because the value changes of scan flipflops during the scan test must not affect the value of registers with chip-specific values.

EDA test insertion tools analyze a gate-level design to find violations originating X values [17]. The location of a violation is called an X source.

A. Automated Formal Verification of X Propagation at Gate-Level

Having X sources, the next step is to determine which X source is dangerous. For the test goal, the observation points are scan flipflops. Therefore, an X source is dangerous, if the X source propagates an X value to the scan flipflops.

Figure 2 illustrates our automated formal verification approach at gate-level to determine dangerous X sources. In Figure 2, scan flipflops (SFFs) are extracted using Tessent Shell [18] and are considered as observation points. *Design Rule Checks* (DRC) of Tessent tool help to extract the initial list of X sources. DRC D5 [18] reports the list of non-scan latches/flipflops and their values in the test mode. DRC E5 [18] reports the list of elements which may propagate an X value in the test mode.

If the list reported by DRCs contains many elements, the designer's information is utilized to find out which element of DRC is a real X source. The DRCs report also uninitialized elements for the module under test. However, these uninitialized elements of the module are initialized when the module is used in a system.

Given the list of X sources and observation points, our approach automatically creates suitable constraints and assertions for the corresponding X sources and observation points. Then the formal verification is invoked to investigate whether there is a possibility to propagate an X value from X sources to observation points. The X sources which cause an assertion fails are dangerous X sources.

B. X-Prevention at Gate-Level

After finding dangerous X sources at gate-level, they have to be prevented to propagate an X value to scan flipflops. X prevention can be performed by manipulating the gate-level design or the RTL design. If a gate-level X source is available at RTL, i.e., there is a mapping between the location of an X source at gate-level and its location at RTL, and the X source can be prevented at RTL by manipulating the RTL code, then the RTL design is improved and is resynthesized to create a new gate-level netlist. Manipulation of an RTL design is more desirable as the manipulation is kept for the future updates and extensions of the hardware core. In this case, the RTL and gate-level designs can also pass the equivalence checking. When signals related to a gate-level X source are not available at RTL, the manipulation has to be performed at gate-level.

Furthermore, some X sources originate because of a timing violation in signals' propagation which are available at gate-level. In this case, changing signals' timing at gate-level may prevent an X value to originate. Finding timing violations and debugging them are performed by *Static Timing Analysis* (STA) tools.

If a dangerous X source is detected at gate level and there is not enough time to manipulate the RTL design because of the tape-out deadline, the corresponding dangerous X source can be isolated at gate level. Figure 3 shows an example in which one X source is isolated during the scan test mode at gate level. The select line of the multiplexer is connected to the scan mode signal. The 1-input of the multiplexer is taken from a pre-determined logic or value. During the test mode, the output of the multiplexer has a pre-determined value.

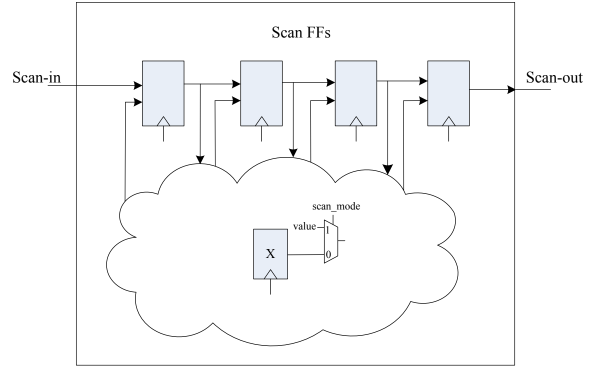

Fig. 3. An example for isolation of dangerous X sources

In summary, the solutions for the prevention of gate-level X sources are as follows:

1) RTL manipulation and resynthesis
2) Gate-level manipulation
3) Isolation of dangerous X sources

V. Increasing the Efficiency of the Proposed Formal Verification

The efficiency and the scalability of formal verification is limited with respect to the size and the complexity of the design under verification. For large designs which have a large number of scan-flipflops, the formal verification cannot be directly applied to evaluate X propagation to all scan-flipflops. In this case, not only the number of scan-flipflops but also the size of the design has to be decreased. To increase the efficiency of the formal verification for large designs, we use the following automated process at gate-level:

1) Creating a list of scan-flipflops by traversing the circuit from X sources until reaching a scan-flipflop
2) Creating connectivity pairs between X sources and the list of scan flipflops
3) Blackboxing irrelevant modules with respect to the connectivity pairs
4) Creating a new formal instance with respect to the list of blackboxes

In step 1, the circuit is traversed from each X source forwards. In this case, stop points are scan flipflops, i.e., if a scan flipflop is found, the traversing process does not continue through the corresponding scan-flipflop. Using this step, only the list of scan-flipflops which may be directly influenced by X sources are extracted. This step is performed using Tessent Shell.

In step 2, a file called *connectivity pairs* is created. Each connectivity pair contains a source and a destination [19]. In our case, a source is the output of an X source. A destination

TABLE I
VERIFICATION OF X PROPAGATION TO SCAN FLIPFLOPS

Benchmarks' Characteristics				Verification of X Propagation to SFFs							
Name	#Gates	#S	#NS	#Constraint_X	#Assertion_no_X	#Pass	#Fail	#Explored	Max n	Max Time (s) / Assertion	Max Mem. (MB)
Bench. 1	3462	248	40	1	248	16	0	232	9999	10 (low)	328
Bench. 2	6892	407	186	3	407	32	0	375	124	10 (low)	470
Bench. 3	12912	789	150	3	789	0	166	623	20	60 (mid)	451
Bench. 4	221671	6775	1632	43	189	2	0	187	21	300 (high)	3495

TABLE II
VERIFICATION OF X PROPAGATION TO PRIMARY OUTPUTS

Benchmarks' Characteristics				Verification of X Propagation to POs							
Name	#Gates	#S	#NS	#Constraint_X	#Assertion_no_X	#Pass	#Fail	#Explored	Max n	Max Time (s) / Assertion	Max Mem. (MB)
Bench. 1	3462	248	40	1	154	54	15	85	27	10 (low)	328
Bench. 2	6892	407	186	3	439	249	3	187	9999	10 (low)	470
Bench. 3	12912	789	150	3	66	5	4	57	9	60 (mid)	451

is the input of a scan flipflop. In this step, connectivity pairs are created from all X sources to scan flipflops extracted in step 1.

Having the file of connectivity pairs, the Cadence tool is utilized to automatically blackbox irrelavent modules according to the connectivity pairs [19]. In this step, a list of blackboxes is generated. Using this list, a new formal instance is created which has a smaller size.

VI. EXPERIMENTAL RESULTS

In this section, we demonstrate the experimental results for verification of X propagation at gate level. We use Cadence IFV [11] as an underlying formal engine.

Table I shows the experiments for verification of X propagation to scan flipflops. Columns 1 through 4 present the characteristics of the benchmarks. The benchmarks are Infineon hardware cores. Column $\#S$ indicates the number of scan flipflops in the benchmarks. Column $\#NS$ indicates the number of non-scan flipflops and latches in the benchmarks.

For X sources at gate level, constraints are generated to constrain their values to X for formal verification. Column $\#Constraint_X$ shows the number of generated constraints. For every scan flipflop, an assertion is generated to specify that the corresponding scan flipflop has to capture no X value. Column $\#Assertion_no_X$ shows the number of generated assertions. Then formal verification is called to prove the assertions. The columns $\#Pass$, $\#Fail$ and $\#Explored$ indicate the result of formal verification.

For $Bench.1$, 16 assertions pass. It implies that for 16 scan flipflops, there is no possibility to be disturbed by X sources (X constraints). For 232 scan flipflops of $Bench.1$, the result of formal verification is explored. In this case, formal verification cannot prove that the corresponding assertions pass or fail. However, the formal verification proves that there is no counterexample till clock cycle n. The maximum reported n for all explored assertions is written in column $Max\,n$. The verification time and the maximum memory consumption are reported in two last columns.

For $Bench.3$, 166 assertions fail. This initial result is pessimistic since formal verification investigates all possible values of primary inputs and registers. To improve the result, more accurate constraints for primary inputs and registers should be written to specify the realistic behavior of the module when the module is used in a system. For example for module m, the designer knows that primary input p has always value 0 when the module is used in a real testbench. Thus a constraint 0 has to be written for primary input p for formal verification. When the formal instance includes more accurate information about the behavior of primary inputs and registers, the verification yields more accurate results.

When an assertion fails, the active X sources (dangerous X sources) which have caused the corresponding assertion fails are reported by running the command "$debug\ \ assertion_name$" in the IFV console [11].

$Bench.1$, $Bench.2$ and $Bench.3$ are module-level benchmarks. $Bench.4$ is a system-level benchmark which contains multiple modules. $Bench.4$ has 6775 scan flipflops. In this case, formal verification cannot be run for all scan flipflops. As explained in Section V, only the list of scan flipflops which may be directly affected by X sources are selected for verification. Therefore, only 189 assertions are generated for the corresponding list of scan flipflops.

For module-level benchmarks, we also have to verify X propagation from X sources to primary outputs. Because an X value may propagate from an X source to primary outputs and then to a scan flipflop in another module. Table II shows the experimental results for X propagation from X sources to primary outputs. As shown in the table, X values can propagate to primary outputs.

To debug a failed assertion, the schematic view (Figure 4) and the waveforms (Figure 5) are utilized to observe the behavior of the counterexample. In the waveform window, the list of active X sources (dangerous X sources) is also displayed. In the schematic view, the path of X propagation is highlighted by red color (Figure 4). The designer uses the mentioned debug features to better understand how an X value

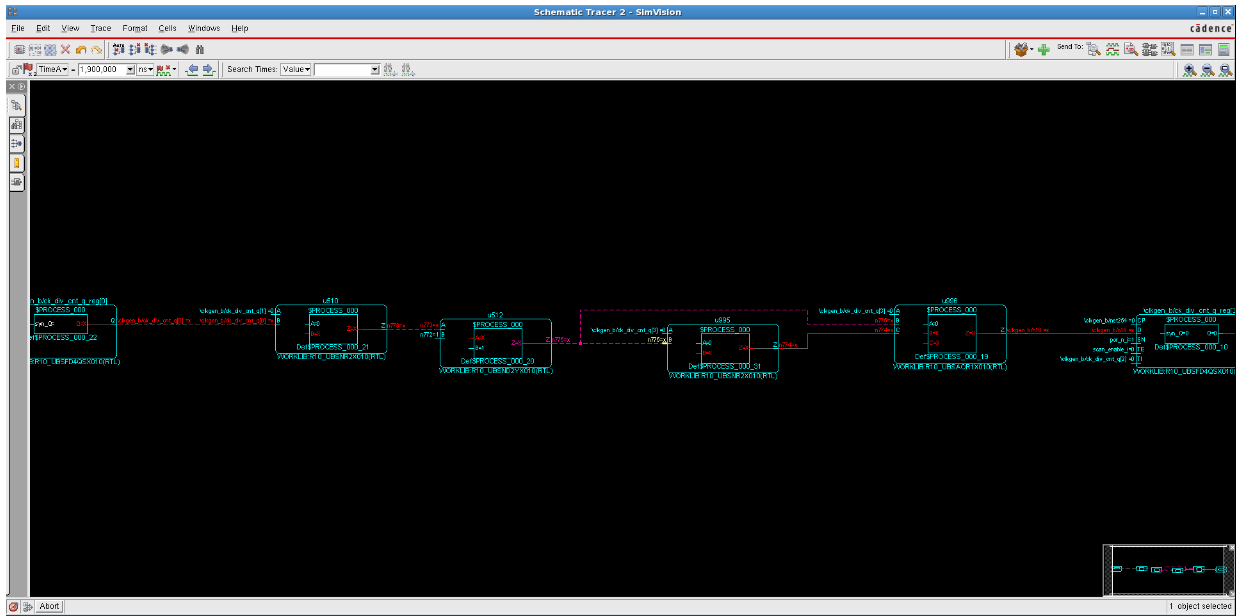

Fig. 4. Debug of X propagation using schematic view

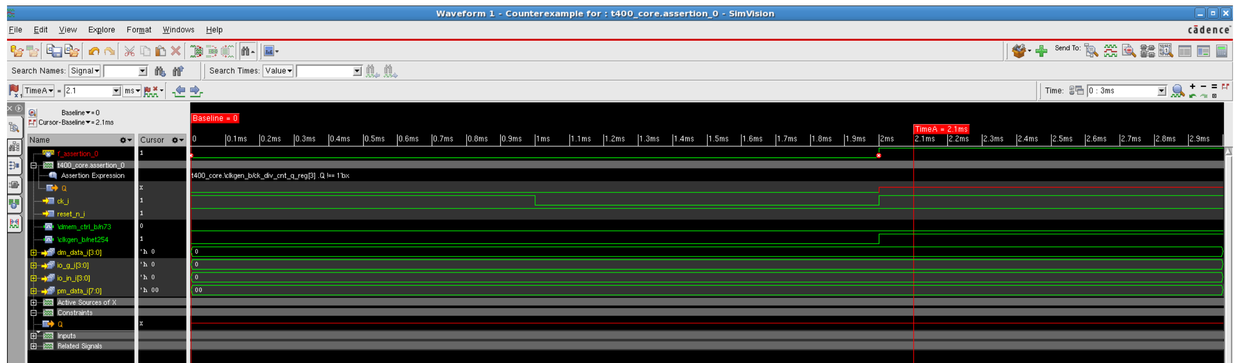

Fig. 5. Debug of X propagation using waveforms

has propagated and, consequently, to find a way to prevent X propagation to scan flipflops.

VII. Conclusion

In this paper, we presented an automated approach to verify X propagation at RTL and gate level. X sources have to be prevented to propagate an X value to scan flipflops in the scan test mode as they corrupt the signature generated by MISR in the test structures such as LBIST. Using our approach, a designer can find root causes of a corrupted signature generated by MISR because of an X value.

References

[1] J. Rajski, J. Tyszer, M. Kassab, N. Mukherjee, R. Thompson, K.-H. Tsai, A. Hertwig, N. Tamarapalli, G. Mrugalski, G. Eide *et al.*, "Embedded deterministic test for low cost manufacturing test," in *Int'l Test Conf.* IEEE, 2002, pp. 301–310.

[2] O. Grumberg, A. Schuster, and A. Yadgar, "3-valued circuit SAT for STE with automatic refinement," in *Automated Technology for Verification and Analysis (ATVA)*, 2007, pp. 457–473.

[3] A. Sülflow, G. Fey, C. Braunstein, U. Kühne, and R. Drechsler, "Increasing the accuracy of SAT-based debugging," in *Design, Automation and Test in Europe*, 2009, pp. 1326–1331.

[4] M. Dehbashi, A. Sülflow, and G. Fey, "Automated design debugging in a testbench-based verification environment," *Microprocessors and Microsystems*, vol. 37, no. 2, pp. 206–217, 2013.

[5] S. Yang, R. Wille, and R. Drechsler, "Determining cases of scenarios to improve coverage in simulation-based verication," in *Integrated Circuits and System Design (SBCCI)*, 2014.

[6] SystemC, "SystemC version 2.0 user's guide," *Open SystemC Initiative*, 2002.

[7] K.-H. Chang, I. L. Markov, and V. Bertacco, "Automating post-silicon debugging and repair," in *Int'l Conf. on CAD*, 2007, pp. 91–98.

[8] S.-B. Park, T. Hong, and S. Mitra, "Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA)," *IEEE Trans. on CAD*, vol. 28, no. 10, pp. 1545–1558, 2009.

[9] W. Fischer, "Formal analysis of X propagation," in *OneSpin Solutions GmbH, White Paper*, 2010, pp. 1–8.

[10] M. Turpin and P. V. Engineer, "The dangers of living with an X (bugs hidden in your verilog)," in *Synopsys Users Group Meeting*, 2003.

[11] Cadence Incorporation, *Formal Analysis, Incisive Formal Verifier User Manual*, 2014.

[12] A. Sülflow, U. Kühne, G. Fey, D. Große, and R. Drechsler, "WoLFram – a word level framework for formal verification," in *IEEE/IFIP Int'l Symposium on Rapid System Prototyping*, 2009, pp. 11–17.

[13] K. Ravi and F. Somenzi, "Minimal assignments for bounded model checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 2988, 2004, pp. 31–45.

[14] A. Groce and D. Kroening, "Making the most of BMC counterexamples," *Electronic Notes in Theoretical Computer Science*, vol. 119, no. 2, pp. 67–81, 2005.

[15] M. N. Velev, "Comparison of schemes for encoding unobservability in translation to SAT," in *ASP Design Automation Conf.*, 2005, pp. 1056–1059.

[16] S. Safarpour, A. Veneris, and R. Drechsler, "Improved SAT-based reachability analysis with observability don't cares," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 5, pp. 1–25, 2008.

[17] Mentor Graphics Corporation, *Tessent Common Resources Manual for ATPG Products*, 2012.

[18] ——, *Tessent Shell Reference Manual*, 2014.

[19] Cadence Incorporation, *Formal Analysis, Verification Apps*, 2014.