Improving Gate-level Simulation Accuracy when Unknowns Exist

Kai-Hui Chang and Chris Browy
Avery Design Systems, Inc., Andover, MA, USA
changkh@avery-design.com, cbrowy@avery-design.com

ABSTRACT

Unknown values (Xs) may exist in a design due to uninitialized registers or blocks that are powered down. Due to X-pessimism in gate-level logic simulation, such Xs cannot be handled correctly, producing false Xs that result in inaccurate simulation values. To improve gate-level simulation accuracy when Xs exist, we first trace the fan-in cone of Xs to check whether they are real. For the Xs that are not real, we extract small sub-circuits responsible for creating the false Xs. We then generate auxiliary code to repair gate-level simulation by replacing the Xs with the correct values. Our experimental results on commercial designs show that the proposed methods are both effective and efficient.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Simulation*

General Terms

Algorithms, Design, Verification

Keywords

X-pessimism, Gate-level logic simulation, Formal methods

1. INTRODUCTION

Gate-level logic simulation is one of the most commonly-used methods for verifying the correctness of design netlists. Even though equivalence checking between Register Transfer Level (RTL) code and gate-level netlists replaced some of the gate-level simulation tasks, the use of physical synthesis optimizations as well as Engineering Change Order (ECO) modifications created new needs to perform gate-level simulation. To utilize gate-level simulation for verification, stimuli are applied to the inputs of the netlist, and the simulation results are compared with a golden model or certain pre-defined checkers for correctness.

Gate-level logic simulation mimics the digital behavior of netlists and handles Boolean (0/1) values well. When unknown values (Xs) exist, however, it can no longer produce correct results due to X-pessimism. A simple example to illustrate the problem is shown in Figure 1. In the example, the output of gate $g6$ should be 0, but logic simulation generates an X. Such inaccuracy has a ripple effect and can produce numerous false Xs, rendering gate-level simulation useless. This problem is becoming severe due to physical optimizations and low-power requirements that allow more and more Xs to reside in the design after reset.

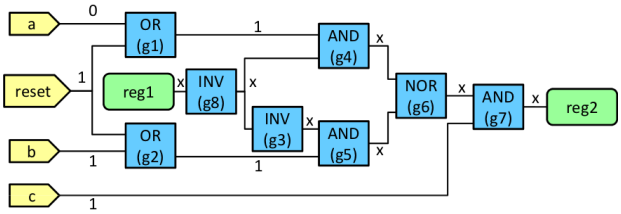

Figure 1: X-pessimism example. The output of $g6$ should be 0, but logic simulation produces X.

One simple solution to address such simulation problems is to replace Xs in registers with random values, such as the work by Hira *et al*. [5]. Such an approach eliminates X problems by converting the Xs into non-X values. Since Xs no longer exist in the design, logic simulation can produce correct simulation results. However, each deposited value only represents one of the two possible values that the X can have, and randomly choosing one of them for simulation can cause bugs to escape verification. For example, if a bug only manifests itself when twenty unresettable registers happen to be 1 after power up, then the probability for the random-deposit approach to detect the problem is less than one in a million. To properly fix gate-level simulation without masking any bugs, Chang *et al.* [1] proposed a formal-based methodology to find false Xs during simulation. Although effective in finding false Xs, their solution to fix the problem is not generic. More specifically, their solution replaces the false Xs in registers with the correct non-X values at the simulation time when the formal analysis was applied. This solution clears false Xs at the particular time, but it does not resolve subsequent false Xs even if the conditions that produced the false Xs are identical.

In this paper we propose a simple yet effective solution to improve gate-level simulation accuracy when Xs exist[1]. Our solution first identifies false Xs when simulating a given input trace. It then analyzes the combinational fan-in cones of such false Xs to find small sub-circuits responsible for the false Xs. Finally, it generates auxiliary-code to eliminate those false Xs. To achieve this goal, we propose a novel methodology and several innovative algorithms based on logic simulation and formal analysis. By utilizing logic simulation results in our analysis, we can considerably reduce the search space of formal engines, making our methods scalable and efficient. Our empirical results show that we can analyze a multi-million gate industrial design in two hours, and the generated fixes successfully eliminated the identified false Xs. The techniques proposed in this paper are currently in commercial production use, which further demonstrates their usefulness in solving real industrial problems.

[1]The proposed solution is currently patent pending.

DAC 2012, June 3-7, 2012, San Francisco, California, USA.

The rest of the paper is organized as follows. Section 2 provides a brief overview of related work. Section 3 presents our analysis of the gate-level X-pessimism problem. Our simulation repair solution is described in Section 4. Section 5 shows our experimental results, and Section 6 concludes this paper.

2. RELATED WORK

Most work that addresses the X problem in logic simulation focuses on the RTL because X-optimism in RTL logic simulation can easily mask bugs, whereas the X-pessimism problem at the gate level corrupts simulation but does not mask bugs. To this end, the paper by Piper *et al.* [8] provided comprehensive background on how Xs are generated in industrial designs. They also surveyed several engineers to find out what X problems they have and what solutions they need. However, even though the paper claimed that they provided a complete solution for X-optimism and X-pessimism problems, the proposed solution lacks technical details and is only tested on a small OpenCores design. Therefore, the effectiveness of their solution on industrial designs is questionable. Unlike Piper's work, Chou *et al.* [2] and Chang *et al.* [1] reported their results using real industrial case studies, suggesting that their solutions are more practical than Piper's work. More specifically, Chou *et al.* [2, 3] proposed to use formal methods to find Xs that are masked by X-optimism. They also proposed a methodology that is effective in reducing the amount of analysis that engineers need to perform. Chang *et al.* [1] extended Chou's work to find false Xs produced by X-pessimism at the gate level. Chang's technique successfully analyzed two industrial designs in a few hours. However, their analysis only provides correct X status at the checkpoint and cannot eliminate false Xs that occur later in the simulation trace. The solution provided in this work addresses this limitation.

Haufe *et al.* [4] proposed to change the RTL coding style to reduce X-optimism so that RTL simulation results are closer to the gate level when Xs exist. However, such work is typically not applicable to fixing gate-level simulation because it focuses on finding problems caused by X-optimism instead of X-pessimism.

Petlin [7] proposed a technique to find X sources as well as the locations where such Xs can be trapped. However, he did not provide solutions on how to use the analysis to improve gate-level logic simulation accuracy. In addition, his method analyzes all possible paths for X-propagation, while in this work we only analyze the paths that caused the false Xs. The latter has significant performance advantage because the search space is much smaller.

In industry, engineers also have ad hoc solutions for X-pessimism problems. For example, one company developed a sophisticated Perl script to recognize multiplexers and other gate-level structures that can create X-pessimism problems. However, the recognition was based on structure templates and could easily miss new constructs generated by physical synthesis tools. Maintaining the accuracy of the script eventually became a major challenge for the engineers, and they decided that a solution based on logic instead of structural analysis was required to properly address the X-pessimism problem in gate-level logic simulation.

3. ANALYSIS OF THE GATE-LEVEL X-PESSIMISM PROBLEM

Unlike RTL simulation where Xs may be injected to represent don't-cares or erroneous conditions, at the gate level Xs are typically from uninitialized registers or power-down blocks. Such Xs tend to disappear when simulation proceeds because known values will be written to those registers during the reset or power-up sequences. Given that specific sequences are required to reset a design or power up a block, typically only a small number of short traces need to be analyzed for X-pessimism problems. Once the false Xs are eliminated in those reset or power-up sequences, gate-level simulation should be clean afterward. In this work we assume that a trace is given, and the purpose of our flow is to generate fixes to eliminate the false Xs so that no false X can be latched into any register when simulating the trace. If there is more than one reset or power-up sequence, all the sequences need to be analyzed. However, since the fix generated by one trace can be applied to other traces, we expect the number of generated fixes to reduce when each additional trace is analyzed. This is not the case for other methods such as [1], where the fix for one trace cannot be applied to another one.

If a netlist is modeled using basic gate types such as AND, OR, XOR, INV, etc., then gate-level simulation has a special characteristic that simulating the combinational logic can only produce X-pessimism problems and cannot produce X-optimism problems. The reason is that for each gate, all its inputs will be evaluated during simulation, and the output of the gate is determined pessimistically — it produces a known value only if the Xs on the inputs are guaranteed not to propagate to the output. As a result, the non-X values in gate-level simulation are always correct, while the Xs may be false. In our algorithm we utilize this characteristic to reduce formal analysis, which can provide considerable performance gain compared with pure formal methods. Since combinational cells in cell libraries are typically composed of such basic gates, most designs possess this characteristic.

4. OUR SIMULATION REPAIR SOLUTION

In this section we present our methodology for correcting X-pessimism problems in gate-level simulation and the algorithms for implementing the methodology. In the following discussions we assume the Xs are eliminated by combinational logic. In Section 4.5 we will describe how our analysis can be extended to consider sequential elements.

4.1 Overall Methodology

The inputs to our methodology are a trace as input stimuli, a gate-level netlist, and a set of time points (called checkpoints) that the Xs should be checked to determine whether they are false or not. The output is auxiliary-code that when simulated with the trace, the false Xs at the checkpoints will be replaced with the correct values. If each cycle has a checkpoint, then it can be guaranteed that no false Xs will be latched into registers based on the current simulation values.

Our methodology works as follows: (1) at each checkpoint we check Xs in register data inputs (typically denoted as "d") to determine if they are false; (2) for each false X, we trace the fan-in cone of the register input to find a portion of the cone, called a sub-circuit, whose inputs have real Xs and whose output is a false X; (3) we generate auxiliary-code based on the sub-circuit to eliminate such Xs; and (4) the original trace is resimulated with the auxiliary-code and the code eliminates false Xs. This methodology allows gate-level simulation to produce correct results.

To support step (1), (2), and (3) of the methodology, we develop novel methods and algorithms, which will be discussed in the rest of the section.

4.2 Identifying False Xs

The algorithm for identifying whether an X is false is shown in Figure 2. The input to the algorithm is a register data input, d, that has X in logic simulation. The algorithm returns whether the X is false. The fan-in cone of d is also returned in *subckt*.

```
function checkX(input d, output subckt);
1   pi_frontier ← d;
2   while (pi_frontier not empty)
3     var ← pi_frontier.pop();
4     gate ← var.get_fanin_gate();
5     subckt ← subckt ∪ gate;
6     foreach input ∈ gate.get_inputs()
7       if (input.value = x &&
            input ∉ {design inputs, register outputs})
8         pi_frontier ← pi_frontier ∪ input;
9   return proveX(subckt);
```

Figure 2: Algorithm for checking whether an X is false. The sub-circuit responsible for producing the X is also returned.

In line 1 of the algorithm, d is inserted into $pi\_frontier$, which is a set of inputs to the fan-in cone logic collected in $subckt$ so far. We then expand the fan-in cone by popping a variable, var, from $pi\_frontier$ (line 3) and get the gate, $gate$, that fans out to the variable in line 4. The gate is then added to $subckt$ in line 5. In line 6 we check the inputs of $gate$ and add an input to $pi\_frontier$ if (1) the input has an X value in logic simulation, and (2) the input is not a primary input or a register output. The latter condition stops fan-in extraction at register boundaries, making our analysis combinational. Line 9 calls function $proveX$ to prove whether the X in $subckt$'s output, d, is real.

Function $proveX$ is implemented as follows. It first builds a Boolean function from $subckt$. For each input of $subckt$, if its corresponding variable has an X value in logic simulation, we make it an input of the function; otherwise, we assign the non-X value to the input and propagate the constant into the logic function. We then use a formal solver to check whether the output can have different values. In our implementation, we first use random simulation to calculate several values of the Boolean function. If the values can be different, the X is real, and $proveX$ returns true. If all the values are identical, we form a SAT instance from the Boolean function and constrain the output of the function to the opposite value from simulation. We then use a SAT solver to solve the instance. If SAT found a solution, the X is real and $proveX$ returns true. Otherwise, $proveX$ returns false.

Since there is no X-optimism in gate-level simulation, all non-X values are correct, allowing us to use them directly in $checkX$ without the risk of masking any real X problems. Therefore, only Xs need to become inputs to the Boolean function. This Boolean function is typically much smaller than the complete fan-in cone of d, which allows us to prove false Xs efficiently.

4.3 Minimizing X-eliminating Sub-circuits

In the previous algorithm, when $proveX$ returns false, the returned sub-circuit ($subckt$) produces a false X on its output. To repair logic simulation, we can monitor the simulated values on the inputs of $subckt$ and eliminate the X on its output when the condition matches. Nonetheless, $subckt$ may be unnecessarily large because its inputs are either primary inputs or register outputs, but the logic that produces the false X may only be a small portion of $subckt$. If we can identify this portion, the generated fix to repair logic simulation will be much more compact. In addition, the fix can potentially eliminate multiple false Xs that have overlapping fan-in cones, reducing the number of fixes that need to be generated. To achieve this goal, we propose two new algorithms. The first one reduces the sub-circuit from its output by tracing the X towards its inputs, while the second one proceeds from the inputs towards the output.

The first algorithm is called $ckt\_minimize1$ and is shown in Figure 3. The input to the algorithm is $subckt$ from $checkX$, and the output is a new sub-circuit, called $subckt_n$, that is a subset of the original sub-circuit and still produces false Xs.

```
function ckt_minimize1(input subckt, output subckt_n);
1   new_po ← subckt.get_output();
2   subckt_n ← subckt;
3   do
4     gate ← new_po.get_fanin_gate();
5     c_po ← new_po;
6     foreach input ∈ gate.get_inputs()
7       subckt_input ← fanin cone of input in subckt;
8       if (input.value = x &&
          proveX(subckt_input) = false)
9         new_po ← input;
10        subckt_n ← subckt_input;
11    if (c_po = new_po)
12      break;
13  return;
```

Figure 3: Algorithm for tracing the fan-in of false Xs to reduce the X-elimination sub-circuit from the output.

The algorithm starts from the original output of $subckt$ and traces the Xs in its fan-in cone until a real X is reached. The last false X then becomes the new output of the sub-circuit, $subckt_n$, that also eliminates false Xs. Note that during the tracing, if there is more than one input that has a false X for a given gate, our algorithm always picks the first one (lines 6-8). When this happens, more than one iteration may be necessary to eliminate all the false Xs because the X-eliminating sub-circuit now only eliminates the X on the chosen input. To handle this situation, after a fix is found, we replace the X in the fixed variable with its non-X value and perform logic simulation on the original fan-in cone of d. We then check if d is X. If it is, the same repair analysis is performed again. This process should repeat until d is no longer X. At this point, the false X at d is successfully repaired by the fixes generated for its fan-in cone. A simple example to illustrate the algorithm is shown in Figure 4.

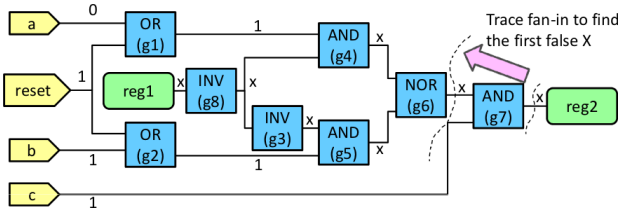

Figure 4: A simple example to illustrate the first step of reduction: tracing the fan-in cone of the false X to find the first appearance of false Xs. This analysis reduces the X-eliminating sub-circuit from its output.

Algorithm $ckt\_minimize1$ reduces the sub-circuit from its output. To further minimize the sub-circuit, we propose another algorithm named $ckt\_minimize2$. The algorithm is shown in Figure 5, and it moves the input frontier of the sub-circuit towards the output. The input to the algorithm is $subckt$, which is the sub-circuit ($subckt_n$) returned by $ckt\_minimize1$ shown above. The output is a new sub-circuit saved in $subckt_n$.

In line 1 of the algorithm, we copy $subckt$ to $subckt_n$. In lines 4-10 of the algorithm, we iteratively remove each gate that connects to the primary inputs of $subckt_n$ and then check whether the output

```
function ckt_minimize2(input subckt, output subckt_n);
 1   subckt_n ← subckt;
 2   do
 3     changed= false;
 4     foreach gate connected to subckt_n.get_inputs();
 5       subckt_n ← subckt_n \ gate;
 6       if (proveX(subckt_n) = false)
 7         changed = true;
 8         break;
 9       else
10         subckt_n ← subckt_n ∪ gate;
11   while (changed);
12   return;
```

Figure 5: Algorithm for minimizing an X-elimination sub-circuit by moving its primary inputs towards its output.

is still a false X. If the X is still false, we keep the change (lines 6-8). Otherwise, we add the gate back (line 10). This process repeats until no further gates can be removed (the do-while loop from line 2 to line 11). At this point, we have moved the inputs of the sub-circuit as close to its output as possible. An example to illustrate the procedure is shown in figure 6.

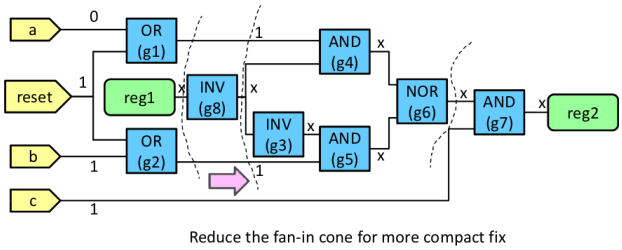

Figure 6: The second step of reduction: moving primary inputs of the sub-circuit towards its output.

4.4 Generating Simulation Repair Code

The sub-circuit ($subckt_n$) produced in the algorithms shown in Section 4.3 can be used to generate auxiliary-code to repair gate-level logic simulation. The algorithm to generate the repair code works as follows.

1. Traverse the inputs of $subckt_n$ to collect the condition for the false X to occur based on their logic simulation values. For example, if variable *var*1 is 1 and *var*2 is X in logic simulation, the condition will be "*var*1 === 1'b1 && *var*2 === 1'bx" when expressed in the Verilog language.

2. Generate code to replace the X on the output of the sub-circuit with the non-X value when the condition matches. The non-X value can be derived by assigning random values to $subckt_n$'s inputs and then check its output. Alternatively, the constant value proven by *proveX* can be retained and used here. When the condition does not match, such value-overwrite should be disabled. In the Verilog and System-Verilog hardware description languages, commands "force" and "release" can be used.

The fix for the example in Figure 1 is shown in Figure 7. The generated code can correct all false Xs that match the condition even if the Xs are not within the analyzed period of the trace or are in a different trace.

The reason why the generated code can correctly eliminate the false Xs is because the constructed sub-circuits follow the semantics of X and non-X values in logic simulation. In logic simulation, X can be either 0 or 1, which is consistent with our construction that the variable is an input to the sub-circuit. For non-X values, we propagate them into the sub-circuit by stopping at non-X boundaries when constructing the Boolean function for X-proving, and this is also consistent with the semantics of logic simulation. Therefore, the sub-circuit faithfully captures the behavior of the netlist, and this allows us to replace the output of the sub-circuit with the correct value during gate-level simulation without creating or masking any problem.

```
always @(g1.o or g2.o or g8.o)
if (g8.o === 1'bx && g1.o === 1'b1 && g2.o === 1'b1)
   force g6.o = 1'b0;
else
   release g6.o;
```

Figure 7: Code generated for correcting the X-pessimism problem in Figure 1 using the Verilog language. In the example, the output port of a combinational gate is named "o".

4.5 Analysis of Our Solution

If every cycle is a checkpoint, our methodology is guaranteed to repair all false Xs for a given trace because the number of false Xs produced when simulating the trace is limited, and our procedure repairs at least one false X in each iteration. Therefore, the repair process will eventually end. If not all cycles are checked, however, false Xs in the unchecked cycles can be missed. Nonetheless, checking every cycle can be time-consuming, making the selection of checkpoints important for the completeness and the efficiency of our analysis. Designing an effective heuristic for checkpoint selection is our on-going work.

Our analysis traces the fan-in cone of a register's data input and stops at primary inputs or registers. Therefore, the fixes we found are combinational. To repair false Xs that can be eliminated when sequential elements are involved, we can trace across register boundaries by backward time-frame expansion. In this way, false Xs that can be eliminated within N cycles can be found by expanding N time frames. To generate repair code, additional registers need to be used to keep track of the values of the involved registers up to N cycles before the X-elimination point. In practice, we have analyzed tens of netlists from several different companies and have only seen one case where sequential analysis was necessary. Given that sequential analysis is much more time-consuming than combinational analysis and is rarely used, we currently have not implemented this analysis mode.

Our algorithm is more efficient than other methods such as [1] because we simplify our analysis based on the fact that all non-X values in gate-level simulation are correct. As a result, the sub-circuit that needs to be proved only includes the logic along the paths that are Xs in logic simulation. By analyzing a smaller logic cone, the false Xs can be proved more efficiently.

When designing our solution for X-pessimism problems, we have considered three other approaches and found that they are impractical. The first approach is to abstract the gates to higher-level structures and then eliminate false Xs for those structures. For example, if multiplexers can be recognized, the false Xs on their outputs can be easily eliminated. However, we found that most netlists in industry have undergone various physical synthesis optimizations, making accurate abstraction extremely difficult. The second approach is to carry RTL information over to the gate level so that logic structures that cause X-pessimism problems can be known in advance. However, this is difficult without the support from synthesis tools. In addition, X-pessimism problems can be caused by physical syn-

thesis tools and ECO changes, limiting the X-pessimism problems that can be found by this approach. The third approach is to use pure-formal methods to enumerate all possible X-pessimism conditions and generate fixes before gate-level simulation is performed. However, we found that this approach can easily generate a huge number of fixes. For example, for a two-input multiplexer, at many as six fixes can be generated depending on how the multiplexer is implemented. However, most of the fixes are useless because the X-pessimism conditions never appear in real traces, and the large volume of fixes can significantly slow down gate-level simulation. Therefore, this approach is also impractical for industrial designs.

5. EXPERIMENTAL RESULTS

We implemented our methods on a proprietary simulator [9] and the ABC package from UC Berkeley [10]. We applied our methods to a multi-million gate commercial design to repair false Xs in its reset sequence. The reset sequence was 74 cycles long, and the checkpoint was set to the last cycle. Due to the non-disclosure agreement with the company, we could not describe the benchmark in more detail.

The work by Chang *et al.* [1] is one of the state-of-the-art methods to solve X-pessimism problems. In order to compare our results with Chang's, we used the same partitioned flow described in [1] even though such a step is not necessary due to the scalability of our methods. 202 partitions were produced by the partitioner, and 140760 register inputs were checked for false Xs. Our method used 1h36m to identify 125 false Xs, and it produced 145 fixes. Most of the fixes involved only three wires in their conditions, and a few of them involved more than 10 wires. Those involving more than 10 wires were mostly in arithmetic blocks. We also produced fixes whose conditions are all Xs. Further analysis showed that the wires being repaired were constant under all possible conditions. In other words, they were stuck-at-0 or stuck-at-1. To repair such false Xs, we generated force statements without any condition. We have rerun logic simulation with the fixes and confirmed that all the false Xs have been eliminated. We did not observe simulation speed degradation in this benchmark because the fixes only contained 145 forces for this multi-million gate design and the fixes were mostly inactive throughout simulation.

We also applied Chang's methods on the same design for comparison. Runtime of Chang's flow was 4h46m, and it identified 201 false Xs. The false Xs we found were a subset of those found by Chang's work. The reason is that we were performing combinational analysis, while Chang's method performed full-fledged sequential analysis. There were Xs eliminated in earlier cycles and then propagated to a register as a false X. For example, a register whose fan-in cone is a buffer may have latched a false X from its source register. In this case, Chang's work found the X to be false, while our analysis showed that the buffer was not able to eliminate the X. However, since the repair code we produced may eliminate the false Xs at earlier cycles if the conditions match, we actually eliminated 140 false Xs out of the 201 Xs proven to be false by Chang's flow, 15 more than the 125 false Xs identified in our flow. This result shows one advantage of this work compared with Chang's work: our fixes can repair unseen problems as long as the conditions match, while Chang's work can eliminate false Xs correctly at the checkpoint but cannot eliminate false Xs elsewhere during simulation.

To further illustrate the value of our methods, we provide two case studies from our industrial partners. In the first case, Xs kept reappearing during simulation due to power-related operations. Because the problematic block was from an IP vendor, no RTL was available for comparison, forcing the engineers to analyze the gate-level netlist directly. They spent a month tracing the problem and finally relied on a formal tool to prove that the Xs were harmless. Our methods analyzed the design in 1h30m and the generated fixes successfully repaired gate-level simulation. In the second case, our methods were used as a debugging tool to analyze gate-level X problems. In this use model, the X-proving step tells the engineer whether the X needs to be analyzed. If the X is false, the generated fix not only repairs simulation but also explains how the false X is generated. In one real case, a designer spent almost three hours to trace an X problem. Our solution took just a few minutes to prove that the X is false and produced a fix that involved only 7 gates. By inspecting the generated repair code, the engineer easily found that the inverters inserted by physical synthesis tools caused the false X. Without the rigorous formal analysis and repair minimization steps, it would be difficult to identify the root cause of the problem.

6. CONCLUSION

In this work we proposed a new technique for improving gate-level logic simulation accuracy when unknowns (Xs) exist. Our method uses simulation results to reduce the scope of analysis. We then apply formal techniques to rigorously prove whether the Xs are false. For false Xs, we identify a small portion of design logic responsible for creating the false Xs and then generate code for correcting the simulation results. Unlike existing solutions that repair X problems by depositing random values, our solution does mask real problems due to its rigorous formal analysis step and the robust repair method. The tool is in commercial production use, suggesting its effectiveness in solving industrial problems. We are currently integrating our methods with commercial simulators for easier adoption of our simulation-repair solution.

7. REFERENCES

[1] K.-H. Chang, H.-Z. Chou, H. Yu, D. Dobbyn and S.-Y. Kuo, "Handling Nondeterminism in Logic Simulation So That Your Waveform Can Be Trusted Again", *IEEE D&T*, DOI:10.1109/MDT.2011.75

[2] H.-Z. Chou, H. Yu, K.-H. Chang, D.Dobbyn and S.-Y. Kuo, "Finding Reset Nondeterminism in RTL Designs – Scalable X-Analysis Methodology and Case Study", *DATE, 2010*, pp. 1494-1499.

[3] H. Z. Chou, K. H. Chang, and S. Y. Kuo, "Handling Don't-Care Conditions in High-Level Synthesis and Application for Reducing Initialized Registers," *DAC, 2009*, pp. 412-415.

[4] C. Haufe and F. Rogin, "Ad-Hoc Translations to Close Verilog Semantics Gap," *workshop on DDECS, 2008*, pp. 1-6.

[5] K. Hira and N. A. Panchal, "Random Initialization of Latches in an Integrated Circuit Design for Simulation", *US Patent Application 2010/0017187 A1*

[6] A. Mishchenko, S. Chatterjee, R. Brayton, "DAG-Aware AIG Rewriting, A Fresh Look at Combinational Logic Synthesis", *DAC, 2006*, pp. 532-535.

[7] O. A. Petlin, "Verification Systems and Methods", *US Patent Application 2010/0313175 A1*

[8] L. Piper and V. Vimjam, "X-Propagation Woes: Masking Bugs at RTL and Unnecessary Debug at the Netlist", *DVCon, 2012*, session 5.3.

[9] Avery Design Systems Inc., http://www.avery-design.com

[10] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu/∼alanmi/abc/abc.htm