Are Logic Synthesis Tools Robust?

Alberto Puggelli, Tobias Welp, Andreas Kuehlmann and Alberto Sangiovanni-Vincentelli
Department of Electrical Engineering and Computer Science
University of California - Berkeley
Berkeley, CA
{puggelli,twelp,kuehl,alberto}@eecs.berkeley.edu

ABSTRACT
A systematic investigation is presented about the robustness of logic synthesis tools to equivalence-preserving transformations of the input Verilog file. We have developed a framework that: 1) parses Verilog behavioral models into an abstract syntax tree; 2) generates random equivalence-preserving transformations on the syntax tree, and; 3) writes the transformed design back in Verilog format. The original and the transformed Verilog descriptions are then checked for equivalence and synthesized. Results show that average (peak) improvements in area of 2.5% (11%) and length of the critical path of 4% (13%) are achievable. Indeed these figures are comparable to recent advancements in logic synthesis ([17] [8] achieve 4.9% (23%) 5% (24%) improvements area-wise, respectively), signaling a relevant lack of robustness in synthesis tools. This lack of robustness suggests that new synthesis algorithms should be evaluated by measuring the average improvement on several transformed files to assess their real contributions to the quality of the results.

Categories and Subject Descriptors
B.6.3 [Logic Design]: Design Aids—Automatic synthesis.

General Terms
Performance, Measurement

Keywords
Logic Synthesis, Robustness, Verilog

1. INTRODUCTION
In the last decades, multi-level logic synthesis has benefited of major advancement in the degree of automation. A synthesis flow is a suite of automated tasks that sequentially manipulate a circuit description at different levels of abstraction in order to achieve nearly optimal results in terms of area, delay and power. These tasks include steps for technology-independent optimization (e.g. constant propagation, redundancy removal, common sub-expression sharing, Boolean minimizations [5]), technology-dependent optimization and mapping [12], and timing correction [4]. Every stand-alone stage of the synthesis flow has been well understood and mathematically formalized, and several heuristic-based algorithms are capable of obtaining relevant improvements with respect to the previous stage in a short time. On the other hand, a full understanding of the relationships among different stages still remains elusive. The final output of a logic synthesizer depends both on the effectiveness of each stand-alone algorithm, and on the order in which these algorithms are run on the circuit representation. Since algorithms are not optimized to take into consideration interface conditions and composition with other algorithms, small perturbations at any stage of the overall flow might result in significant fluctuations of the final output of the synthesis, both in terms of area and timing.

We argue that even a minor change in the circuit description and a variation of the sequencing of optimization steps produce a relevant change in the synthesis results and that robustness of the present synthesis flows is in question. The first step towards the development of a robust synthesis flow is the creation of a framework where the effect of perturbations in the circuit description and in the sequencing of optimization steps can be studied and understood. This paper proposes an implementation of such a framework.

Since it is likely that perturbations injected earlier in the synthesis flow produce larger fluctuations at the output, we focused on perturbations at the highest level of abstraction provided to the designer, the behavioral description of the circuit. In particular, our framework is applied to designs written in Verilog. As far as the scope of the perturbations is concerned, we focused on equivalence-preserving transformations, since we can easily identify them as “small” perturbations of the input design description. We informally define a synthesis flow to be robust if equivalence-preserving transformations in the input behavioral descriptions of circuits produce negligible changes in the output optimized and technology-mapped circuit netlists. In Section 4, we experimentally show that state-of-the-art synthesizers are not robust according to the definition above, since equivalence-preserving perturbations cause a spread in area and delay at the output which is comparable to recent advancements in logic synthesis algorithms [17] [8]. We can thus consider this lack of robustness or inconsistency as relevant.

To the best of our knowledge, this is the first systematic study on the robustness of logic synthesizers with respect to equivalence-preserving transformations of the input behavioral description. We see much promise in this direction of inquiry for at least three reasons. First, as mentioned above, our framework can be used to gain insight on the lack of robustness of logic synthesizers, and foster research targeting more robust tools. Since most production cycles require many design iterations, a robust tool, one that preserves a consistent representation of the circuit along all design phases, would help designers in better understanding which are the consequences of their changes to the circuit, and would better guide the exploration of design trade-offs.
Moreover, as also argued in [3] in the context of physical synthesis, a more robust flow (called "stable" in [3]) might even justify a moderate loss in area and circuit delay, since it speeds up the convergence of synthesis loops. To make this point even more clear, let us consider the simple Quick Sort algorithm, when run on an array of unsorted and total ordered elements. While it is known that its runtime depends on the input initial ordering, a routine that implements the algorithm cannot be considered correct if the output order depends on the input order. We would like to have a similar scenario also in logic synthesis. Second, designers can immediately leverage the acquired insight into the operation of synthesis tools to improve the performance of their projects, by running several synthesis flows on perturbed input files. Third, new algorithms for logic synthesis can be validated within our framework. The problem of devising an effective experimental setup to assess the validity of a new heuristic has already been raised in the past, e.g. in [7]. In this paper, we propose to evaluate enhancements in performance on the basis of the change in average value produced on several perturbed input files, to identify the contributions due to the algorithms and their robustness.

The rest of the paper is organized as follows: in Section 2, we describe possible causes of inconsistency in synthesis tools and present related works on the topic; Section 3 describes in detail each step of the proposed framework; in Section 4, we show the obtained results and discuss possible implications of our research; we draw some final conclusions in Section 5.

2. BACKGROUND

We begin this section by briefly reviewing some of the possible causes of inconsistency in the outputs of logic synthesis tools. This review does not mean to be exhaustive, but it highlights some facts that are rarely commented in literature. We aim at pinpointing that some implementation choices that are absolutely reasonable from a space and time complexity perspective might be disadvantageous when considered robustness-wise.

We divide causes of inconsistency into two main categories, depending on whether inconsistency is introduced within a single stage of the synthesis process or across multiple stages.

Within a single stage, inconsistency can be caused by: Hashing. Algorithms for logic synthesis make use of hashing (e.g., structural hashing in And-Inverter Graphs [13]). If hash keys are constructed from non-canonical attributes, e.g., variable names, it is likely that logic synthesis would produce results that deviate significantly from one another if these attributes are changed, e.g., variable renaming.

Ordered greedy heuristics. Greedy heuristics that process data structures based on the order in which elements are stored (e.g., redundant cubes in Espresso [18]) reduce run-time, usually at a moderate decrease in the perceived stage-level performance. On the other hand, it is conceivable that a change in the order in which the data structure is built would produce a different output, and that the perturbation would be propagated along the synthesis flow.

Tuning of internal parameters. The behavior of a lot of algorithms is regulated by the tuning of some internal parameters. For example, in SAT solvers a time-out parameter is often used to exit difficult branches of the search space, and to restart in possibly more promising directions. Perturbations in the input of these algorithms might result in hitting a different set of thresholds, thus causing different output results.

Across multiple stages, inconsistency can be caused by: Partitioning complex tasks into a sequence of manageable subtasks. Following the divide and conquer approach, complex tasks are divided into simpler subtasks, that are mathematically and computationally tractable. The choice of the correct partitioning levels affects the overall output.

Ordering of synthesis algorithms and depth of iterations. While the above-mentioned synthesis flow partitioning introduces a partial ordering among algorithms, a high degree of freedom is left in selecting the total order of the transformations applied to the circuit representation. Iterations are introduced to retrieve some level of inter-stage dependency, at the expense of higher computational overhead. Thus, the ordering of the algorithms and the number and depth of iterations might even dynamically change between different runs, according to the state of the design with respect to the synthesis cost metrics.

Even though the robustness of synthesis tools has never been characterized systematically, designers and CAD researchers have long been aware that minor changes in the circuit behavioral description can produce relevant changes at the output of the synthesis flow. The author of [15] report that the synthesis output can be improved substantially by adopting a specific design style, based on a structured algorithmic description of circuit components. The authors only comment briefly on the causes of such discrepancies, and it is not clear whether the provided design guidelines would work for other circuits. Moreover, we believe that even more robust design techniques can be devised, if they are supported by a thorough understanding of the causes of inconsistency.

In [14], the authors propose a genetic algorithm to find the best order of execution of synthesis algorithms and the best tuning of algorithm parameters. The proposed approach thus aims at increasing the cross-stage robustness of the synthesis flow. Nevertheless, the problem of solving inconsistencies that are associated with each stage of the flow is not addressed, and it is not clear whether incremental changes in the input file would cause a different evolution of the algorithm ordering.

In [10], the authors address the problem of generating equivalence classes of circuit mutants for benchmarking CAD algorithms for physical design. Our work goes along a similar direction of inquiry, but we develop a framework that 1) generates perturbed input Verilog files that are equivalent to one another by construction, so they do not introduce spurious differences in the synthesis output results; 2) operates at the abstraction level that is used by circuit designers, so it gives them insights about which changes in the Verilog description can cause inconsistencies in the output results; 3) implements different classes of transformations and assesses the impact of each class on the robustness of the flow, thus giving insight to CAD developers about how to make synthesis algorithms more robust.

The framework proposed in this paper addresses exactly the problem of providing a suitable environment where the effectiveness of robustness-oriented algorithms and synthesis flows can actually be tested seamlessly. We do not aim at proposing transformations that trigger all possible causes of inconsistency: indeed, this task would be hopeless without a deep knowledge of the actual code of the synthesis tools. Our goal is to lay the bases of a sound and systematic study of the robustness of synthesis tools, as explained in details in the following of the paper.

3. THE FRAMEWORK

In order to inject equivalence-preserving transformations into Verilog design descriptions, we aimed at developing a framework that fulfills the following requirements: first, the framework must assure that the emitted Verilog files are structurally and syntactically correct and functionally equivalent to the corresponding original files (correctness); second, it is desirable that the framework allows convenient
implementation of a large number of transformations (extendibility); third, the framework must be able to cope with the huge scale of industrial designs, which requires an efficient implementation (scalability).

To meet these requirements, we implemented a framework with the following characteristics:

1. it parses the original designs using LEX[16]/YACC[11] with a modified version of the grammar provided in the VIS verification package [6] and it populates an Abstract Syntax Tree (AST);

2. it applies a random number of equivalence-preserving transformations to the AST;

3. it traverses the AST to emit a transformed design in Verilog format.

In the following two subsections, we justify why the AST is an efficient and effective data structure for our purposes, and we describe which transformations have been implemented.

3.1 The Abstract Syntax Tree

The AST of a Verilog model can be populated while parsing the input Verilog file: every time a rule in the parser is matched, a new node in the AST is created. Since rules corresponding to lower hierarchical levels are matched first, the tree is built bottom-up, and every node stores pointers to its children nodes. Figure 1 shows an example of how a Verilog description of a NAND-gate is represented in our framework. From the figure, it is apparent that the AST is capable of revealing explicitly the hierarchical structure of the design. Blocks of the original Verilog description are encapsulated at different levels of the tree, and their reciprocal relationship is explicitly captured. When implementing transformations, we need to operate only at the appropriate hierarchical level, while the information stored at different levels can be abstracted out. This allows a convenient implementation of the transformations which, in turn, eases the extension of the framework.

The AST is also a space-efficient representation of the design and both populating it from an original file and emitting it as a modified file take time linear with the size of the original file. As mentioned previously, the AST is constructed while parsing the input Verilog file. Since parsing a file is a linear time operation, and a linear number of rules are matched, we proved that the AST data structure can be built both in linear time and space. Furthermore, the transformed Verilog file can be retrieved with an in-order traversal of the complete AST. The traversal takes time linear with the size of the AST, hence also linear with the size of the original file. These results assure that our framework is capable of coping with very large input files.

3.2 Verilog Transformations

After the whole AST has been built, a large number of transformations can be implemented as simple graph transformations. As an example, consider the De Morgan transformation illustrated in Figure 2. Herein, on the left side, we have a structure representing the expression \( E = A + B \). By the De Morgan rule, this is equivalent to \( E = \overline{A \cdot B} \), as represented on the right side of the figure. Note that the transformation can be applied on an arbitrary level of the AST regardless of context, and that leaf expressions (A and B in the example) could actually be roots of complete subtrees. Also note that the transformation is symmetric, i.e. it can be applied in either direction.

Table 1 summarizes all the transformations that we implemented in our framework. Algebraic transformations exploit algebraic properties under which two expressions are equivalent. Re-ordering transformations consist of a random shuffling of the AST nodes where their order does not matter, for example with always statements. To achieve equally probable permutations of the AST nodes efficiently, we implemented Fisher-Yates shuffling [9]. Finally, statement transformations re-arrange sections of the AST as if different (but equivalent) rules had been matched while parsing the Verilog description, e.g. a switch statements can be transformed into a chain of if-then-else statements and vice versa. Since the AST preserves the structure of the design, it is possible to understand how transformations differently impact on the robustness of the synthesis flow, a further advantage of our framework.

During the in-order traversal of the AST, we enable transformations with probability \( p = 0.5 \) at each node where they can be applied. By doing so, a differently transformed Verilog file can be obtained at every run of the tool.

All transformations are by construction equivalence-preserving. Hence, a sequential application of them preserves equivalence as well. Despite this, we utilized a commercial equivalence checker to verify that the original and the modified Verilog descriptions of the designs are functionally equivalent. This assures the correctness of our approach and the reliability of our results.

4. EXPERIMENTAL RESULTS

In this section, we present how the proposed framework can be used to study the robustness of synthesis tools, and we show the obtained results. We conclude by discussing possible implications of our research.

In devising an appropriate set-up for our experiments, we aimed at avoiding any bias in the collected results. In par-

<table>
<thead>
<tr>
<th>Table 1: Analyzed transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algebraic</strong></td>
</tr>
<tr>
<td>De Morgan’s Theorem</td>
</tr>
<tr>
<td>Module item declarations</td>
</tr>
<tr>
<td>Expression</td>
</tr>
</tbody>
</table>

Figure 1: The Verilog description of the NAND module on the top-left corner is stored in the hierarchical AST data structure. The original file can be retrieved by traversing the graph from left to right and emitting all the tags of the terminal nodes.

Program
Source Text
Module
module NAND (Port List
Port List
Port List
Port
in1
, Port
in2
, Port ... 
endmodule
module NAND (in1, in2, out);
input in1, in2;
output out;
assign out = ~(in1&in2);
endmodule

particular, artifacts may be introduced both by weaknesses of a specific synthesis tool, and by the structure of the chosen class of benchmarks. To support the validity of our results, we thus opted for using two commercial state-of-the-art synthesis tools, that we will call Tool ‘A’ and Tool ‘B’, and a large set of heterogeneous benchmarks. We used the publicly available IWLS2005 [1] repository and some blocks of the OpenSPARC T1 processor [2].

4.1 Set-up

For each design, the testing process was developed in two steps. First, we generated a large number (400+) of Verilog files, whose functional equivalence was checked against the original file. These files are divided into 4 sets, according to the enabled transformations. The first 3 sets are respectively characterized by algebraic, re-ordering and statement transformations only, while we enabled all transformations for the 4th set. The robustness of the synthesis tools is investigated by means of a Monte Carlo analysis to show the spread of the output results, and by selecting the best achievable performance to measure the relevance of the introduced inconsistency.

Second, we synthesized the original file and all transformed Verilog files, and we collected the obtained results in terms of area and maximum negative slack. In order to have a common target function for both synthesis tools across all benchmarks, we modified the synthesis scripts by setting the target clock period to 0s. Other choices are possible, and this choice was made for its simplicity. The reported value of the maximum negative slack thus corresponds to the length of the critical path. In the following, we will use the term slack for brevity.

4.2 Results

The first goal of our testing experiment is to show that equivalence-preserving transformations of the input Verilog file are indeed cause of a relevant spread of the synthesis results.

Figure 3 shows the sample-wise synthesis results in terms of area and maximum negative slack for some of the designs. For both tools, 10 random samples per benchmark were selected and they were all normalized to their average value. The results show that fluctuations up to 10% can be achieved with respect to both metrics: we refer to these fluctuations with the term "inconsistency" in the outputs of the synthesis tool.

We studied the statistical distribution of the area and slack metrics across all the generated input files. The analysis of the behaviors of Tool A (B) shows that for 21 (19) designs transformations cause a Gaussian-shaped distribution of the synthesis results. For the other 12 (14) designs, transformations are able to drive the synthesizers to visit two or more local minima: this causes a Gaussian-spread of the results around every minimum. Figure 4 shows some of the obtained distributions, where the continuous behavior has been obtained using kernel density estimation. We summarize the results of the analysis in the left column (Std. Dev.) of Table 2. For each design and for both tools, we report the standard deviation of the samples normalized to the corresponding average value.

The results show that up to $\sigma = 9\%$ fluctuations can be obtained (area spread of des_perf and slack spread of wb_dma and s932j) by only performing equivalence-preserving transformations. The relative spread of the length of the critical path tends to remain constant across all designs, while the relative spread of area becomes smaller for larger designs. This result is due to normalization: designs vary largely with respect to area, while the length of their critical path is similar, since they are designed for similar clock frequencies. However, the absolute fluctuation in area be-
comes bigger for larger designs: indeed, the ratio between the biggest and the smallest design is $>100\times$, while the reduction in relative spread ($\sigma$) is around $10\times$. Figure 5 shows the trend of the relative and absolute spread as a function of area.

We compare our results with recently proposed algorithms for logic synthesis to assess the relevance of the reported lack of robustness. The Best Improvements column of Table 2 lists: 1) the highest improvements in area and slack with respect to the untransformed file that we obtained by applying equivalence-preserving transformations and synthesizing the perturbed behavioral description; 2) improvements reported in [17] [8] (only area values were reported in these papers). Results show that average savings around 2.5% area-wise and 4% slack-wise can be obtained, with peaks of more than 11% (area in usb and slack in asb,phy). These figures are comparable, and for some designs they exceed (e.g. de,perf, spi, s923f), the values reported in literature (4.2% average reduction in area on the benchmarks we used, 5% on the benchmarks reported in the papers). This shows that the measured performance gain of synthesis algorithms can be deeply affected by the lack of robustness of the overall flow.

Finally, we studied the effect of each kind of transformations on the spreading of the synthesis results, and how the transformations compose together. Table 3 shows the number (\#) of designs that are dominated by each kind of transformation and the corresponding average spread of performance. We note that Tool A is more robust to algebraic transformations, while Tool B is less sensitive to re-ordering transformations. The outcomes of both tools are severely affected by statement transformations, which is the dominant component for most designs. The average values of spread for both tools are around $\sigma = 2\%$ for the area and $\sigma = 3\%$ for the slack, when all transformations are enabled. We also note that transformations do not compose together in general, and that the biggest component dominates the distribution that is obtained when all transformations are enabled (All). Numerically, this is reflected by the fact that the average value of spread in the All column of Table 3 is equal to the weighted average of the previous columns.

### 4.3 Discussion

In the previous section, we have shown experimentally that equivalence-preserving transformations of the input Verilog files indeed cause synthesis tools to produce inconsistent output results. Our findings may be used to gain insight on the causes of lack of robustness of logic synthesizers, and foster research targeting more robust tools, as motivated in Section 1.

Designers might take advantage of a deeper understanding of the synthesizer operation, and they may directly use the proposed framework, since it operates at the behavioral description level. Although it is clear that it is not possible to obtain arbitrarily low values of area and negative slack (the Gaussian distributions are truncated), better results with respect to the original file can be obtained by running several synthesis flows on transformed input files. In particular, we note that the best performance area-wise and slack-wise shown in Table 2 were not obtained with the same design: the samples create a Pareto-curve from which designers can select the trade-off that best fits their current application. Figure 6 shows the obtained Pareto curves for two designs of the Faraday benchmarks.

Finally, the proposed framework may be used to validate new algorithms for logic synthesis. Since we have shown that synthesis results are spread across stochastic distributions, spurious improvements may be reported if the design lies at the extremes of the distributions. Using the same input file for the validation may not be enough, since different transformations at some stage of the synthesis flow might drive the rest of the flow to explore different areas of the mapping space. The improvement to be totally ascribed to the new algorithm is the average gain in performance across a large number of transformed input files, since
Table 4: Performance gain due to a higher mapping effort

<table>
<thead>
<tr>
<th>Tool</th>
<th>Ave. Area [%]</th>
<th>Min</th>
<th>Slack [%]</th>
<th>Ave. Slack [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>6.7</td>
<td>12</td>
<td>2</td>
<td>0.9</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>14</td>
<td>2</td>
<td>0.5</td>
</tr>
</tbody>
</table>

The averaging operation filters out inconsistencies injected along the synthesis flow. Even though a similar averaging behavior can be obtained also by testing an algorithm on a large number of different input circuits, we believe that our framework is easier to use, and that it produces a number of configurations that are more statistically relevant. Given that an amount of synthesis design is present benchmark suites. To illustrate this point, we synthesized the same input files with two different mapping efforts: for each design, we computed the relative gain in performance for the average case, and for the best and worst sample-wise case. Figure 7 shows the obtained results for the usb_funct benchmark: the improvement averaged over several perturbed files is $\Delta_{\text{ave}} = 3.8\%$, while the maximum and minimum sample-wise gain is $\Delta_{\text{max}} = 6.6\%$ and $\Delta_{\text{min}} = 1\%$, respectively. In Table 4, we show the overall gain in performance: for all designs, we considered both the average gain over 100 transformed inputs, and the extreme case where only maximum and minimum gains are selected. Although it is usually not true that an actual evaluation set-up just picks the extreme cases, results show that the average improvement can be considered as a reliable gain figure, while sample-wise improvements are affected by large discrepancies.

5. CONCLUSIONS
We proposed a framework for the study of the robustness of synthesis tools with respect to equivalence-preserving transformations of the input Verilog file. We showed that an abstract syntax tree is a flexible data structure that allows implementing a large number of transformations in an easily extensible way. The transformed description is then written back to the Verilog language, and checked for equivalence to the original file, to guarantee the correctness of the proposed approach. The obtained results show that average improvements of 2.5% area-wise and 4% slack-wise are achievable: these figures are comparable to recent advancement in logic synthesis, and they thus witness a relevant lack of robustness in synthesis tools.

We foresee at least three applications for the proposed framework. First, the framework can be used to support the development of more robust synthesis tools. While synthesis performance and run-time has usually been used as the most important metrics, new algorithms and the relative order in which they are run should also be evaluated with respect to the consistency of the produced outputs. We believe that robustness in the synthesis process is a key property to guarantee, since it reduces design time and errors. Second, designers may leverage the gained insight into the operation of synthesis tools to improve the performance of their circuits. Third, new algorithms for logic synthesis may be validated within our framework: enhancements in performance should be evaluated on the basis of the change in average value, and not sample-wise.

As future work, we plan to implement new transformations, e.g. sequential equivalence-preserving transformations, to further characterize the behavior of the synthesizers, and to adapt the framework to study the robustness of the physical design flow, e.g. place and route.

6. ACKNOWLEDGMENTS
The authors thank K. V. Aadithya for help in developing the code and for many enriching discussions. The authors acknowledge the support of the Gigascale Systems Research Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

7. REFERENCES