Using Formal Verification Methods for Optimization of Circuits under External Constraints

Daniel Große Lucas Klemmer Dominik Bonora
Institute for Complex Systems, Johannes Kepler University Linz, Austria
daniel.grosse@jku.at, lucas.klemmer@jku.at, dominik.bonora@jku.at

***Abstract*—This paper targets the optimization of circuit netlists by eliminating redundant gates under given external constraints. Typical examples for external constraints – which can be viewed as external don't cares – are restrictions on input operands, instruction subsets used by a processor for specific applications, or limited operation modes of an integrated IP block. Targeting external don't cares presents a challenge because the optimization problem changes from a completely specified Boolean function to a Boolean relation. We propose an optimization approach that utilizes formal verification methods. We demonstrate how to formulate *Property Checking* (PC) and *Equivalence Checking* (EC) problems to determine if a gate is redundant under given external constraints. Essentially, the validity of up to four rules must be checked per gate. We show that these checks can be solved concurrently, resulting in faster overall optimization. We have implemented our approach as the tool *Formal SYNthesis* (FSYN). FSYN utilizes open-source tools to scale the solving of formal instances with available hardware resources. We demonstrate that our approach can achieve substantial reductions in the number of gates for combinational circuits under given external constraints.**

I. Introduction

In the realm of *Very Large Scale Integration* (VLSI) design, synthesis is the process of converting a high-level hardware description, usually provided in the form of a behavioral *Register Transfer Level* (RTL) design, into a gate-level representation. In the early days, synthesis referred only to the direct transformation into circuitry. Therefore, the term *logic synthesis* was coined [1] to emphasize both, compilation and logic optimization of a design description during conversion into a netlist. The focus of this work is in particular on the second part, i.e. logic minimization of circuit netlists.

The 1980s saw a significant evolution in logic synthesis algorithms, where the treatment of *Don't Cares* (DCs) became more advanced and integral to the optimization process (see e.g. [2], [3]). The computation of *Internal Don't Cares*, which arise from the structure of the netlist itself, and their use for minimization has been integrated in all logic synthesis tools. However, this is not the case for *External Don't Cares* [4]. At its core external DCs are constraints on inputs that will not appear (more details in the next section). In contrast to internal DCs, external DCs are given by the environment or explicitly by the user. Typical examples for external DCs include restrictions on the values of input operands, instruction subsets used by a processor for specific applications, or limited operation modes of an integrated IP block.

Although there have been attempts to unify the characterization of different DC types [5], targeting external DCs presents a challenge because the optimization problem changes from considering a completely specified Boolean function to optimizing a Boolean relation. Additionally, it has been reported in [4] that there are currently no open-source synthesis tools available that accept external DCs.

From the practical side, [6] introduced the *Property-Driven Automatic Transformation* (PDAT) framework which targets the very specific application of eliminating the logic of unneeded instructions in a processor realizing an *Instruction Set Architecture* (ISA). PDAT employs *Property Checking* (PC) [7]–[9] to detect gates that are guaranteed not to toggle for the reduced ISA. These gates can be removed from the netlist. However, [6] has several shortcomings: relevant information such as formalization, used properties and verification directives are not or only partially given, run-time is not reported and only commercial tools are used. Recently, in [10] these problems have been partially solved. The paper introduces PSYN which also employs PC but uses open-source tools for each step, i.e. from synthesis to formal.

Contribution: In this paper, we propose an optimization approach that utilizes formal verification methods to reduce the number of gates in a netlist. Our approach extends the previous work by showing how to formulate PC and *Equivalence Checking* (EC) [11], [12] problems to determine if a gate is redundant under given external constraints. For this, we formalize the respective formal problems and introduce the rules which have to be checked/applied per gate. We show that these checks can be solved concurrently, resulting in faster overall optimization. Thereafter, we present our implementation called *Formal SYNthesis* (FSYN). FSYN uses only open-source tools, in particular Yosys [13], ABC [14], WAL [15], and Z3 [16]. Of course, FSYN is available on GitHub[1]. In the experiments, we demonstrate that our approach can achieve substantial reductions in the number of gates for different circuits under given external constraints.

The paper is structured as follows: Section II presents the formalization based on PC and EC of our approach as well as illustrative examples. In Section III, our tool FSYN is introduced. This includes the flow of FSYN, the description of the different phases as well as improvements for scalability.

[1]FSYN is available at: https://github.com/ics-jku/fsyn.

Section IV presents the experiments. Finally, the paper is concluded in Section V.

II. Circuit Optimization under External Constraints

In this section, we first give some background on logic synthesis and don't cares (Section II-A). Thereafter, we present a generic formalization of the proposed optimization approach in terms of PC and EC problems. This is supported through the use of simple examples (Section II-B).

A. Logic Synthesis, Don't Cares and External Constraints

Towards the end of the 1980s, techniques for optimizing multi-level logic became increasingly prominent. Examples include algebraic manipulation, factoring, common sub-expression elimination, and functional decomposition to minimize logic complexity [17]. As these methods advanced, the use of *Don't Cares* (DCs) became pivotal in the process of logic optimization. DCs offer considerable flexibility for optimization algorithms [2], [3]. Typically, *Internal Don't Cares* and *External Don't Cares* (XDCs) are distinguished. Internal DCs result from the netlist structure in the presence of reconvergent paths. They are divided into *Satisfiability Don't Cares* (SDCs) and *Observability Don't Cares* (ODCs). In essence, SDCs occur when certain input combinations are not generated for a node, while ODCs arise when the output value of a node is not important under specific conditions. The computation of both has been formulated using incompletely specified functions also called permissible functions [18], [19]. Significant improvements in terms of scalability became possible by using simulation and *Boolean Satisfiability* (SAT), see e.g., [20].

As already mentioned in the introduction, XDCs are given by the environment or explicitly by the user. [4] divides XDCs further in two classes by extending the definitions of SDCs and ODCs to the input and output boundaries, respectively. A value assignment to the primary inputs that will never appear (which extends the definition of an SDC to the inputs) is an XSDC. Similarly, external ODCs, termed XODCs, are conditions under which some primary output values are not of interest. In this paper, we focus on XSDCs. XSDCs can be easily expressed as formal constraints on input signals. From the practical perspective, users can leverage well known property languages to specify these *external constraints* in form of assumptions. In the simplest case these assumptions include or exclude values on the primary inputs, but our proposed approach is not limited to this. In the following, we present the formalization of the proposed optimization approach together with simple examples demonstrating the optimization potential.

B. Formalization

Given a netlist of a circuit, we are interested in optimizing the netlist under given external constraints. More precisely, our cost function is the number of gates, i.e. we aim to remove gates from the netlist and thus we are optimizing for area.

TABLE I: Gate Redundancy Rules

Gate	Rules
AND (A, B → Y)	R_AND_Y0: $Y = 0$ R_AND_Y1: $Y = 1$ R_AND_YA: $A \implies B$ R_AND_YB: $B \implies A$
OR (A, B → Y)	R_OR_Y0: $Y = 0$ R_OR_Y1: $Y = 1$ R_OR_YA: $B \implies A$ R_OR_YB: $A \implies B$
XOR (A, B → Y)	R_XOR_Y0: $Y = 0$ R_XOR_Y1: $Y = 1$ R_XOR_YA: $\overline{B}$ R_XOR_YB: $\overline{A}$
NOT (A → Y)	R_NOT_Y0: $Y = 0$ R_NOT_Y1: $Y = 1$

First, we introduce some general notations which we use in the remainder of this paper:

- N denotes a netlist,
- g_i denotes a gate from N, and
- E denotes the user-specified external constraints.

The proposed optimization approach aims to determine if a gate is redundant under specific external constraints. We follow the idea from [6], [10] and capture this gate-redundancy formally with ***rules***[2] to be checked and applied to the current gate g_i. Table I lists the rules for the 2-input gates AND, OR, XOR and the 1-input NOT gate. In general, a rule in Table I is defined as R_<gate>_Y<s>: *condition* where (i) the symbols before the colon constitute the name of the rule, and (ii) if the *condition* holds for the gate <gate> (we also say the rule holds), then the output Y is replaced by <s>. Let us consider the AND gate with the output Y and the inputs A and B as an example: for this gate there are four different rules. For instance, rule R_AND_Y0 describes if $Y = 0$ holds, then the AND gate can be removed and its output Y is replaced by the constant 0. Another example is the rule R_AND_YA which describes if $A \implies B$ holds for the AND gate, then Y is replaced by the input A, in other words the AND gate can be removed and the output Y is directly connected to input A.

Based on these rules, we formulate the core step of the optimization problem under external constraints, the **gate-redundancy check for gate g_i utilizing PC** as

$$G_{PC}(g_i) = \{E^k \wedge \bigwedge_{j=0}^{k-1} T(s_j, s_{j+1}) \ \wedge \ P^k(g_i) \\ \mid P^k(g_i) \in \mathit{Rules}(\mathit{type}(g_i)) \ \} \tag{1}$$

where k is the number of time frames ($k = 1$ in the combinational case), E^k are the external constraints, $T(s_j, s_{j+1})$ is the transition relation, and $P^k(g_i)$ is a concrete rule from all possible rules of the gate g_i of $\mathit{type}(g_i)$ in form of a property; please note that exponent k in E^k and $P^k(g_i)$ is used to make clear that the properties have to be considered from 0 to k. Let us look at this formalization: if g_i is for

[2]We use the term rules as we look at the problem in a more general way than [6], [10].

TABLE II: Truth Table of 2-bit Ripple-Carry Adder

a_1a_0	b_1b_0	$s_2\ s_1\ s_0$
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110

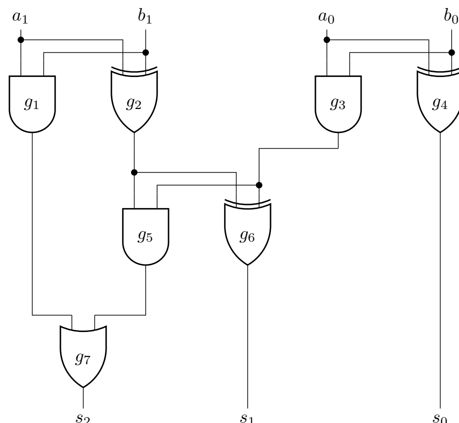

Fig. 1: 2-bit Ripple-Carry Adder

instance an AND gate, $G_{PC}(g_i)$ consists of four PC problems. To perform optimization for the complete netlist N, $G_{PC}(g_i)$ must be checked for all gates of N.

We now present a concrete example to demonstrate the concepts.

Example 1. *We consider a 2-bit* Ripple-Carry Adder *(RCA) which adds up the binary numbers a_1a_0 and b_1b_0 and produces the result $s_2s_1s_0$ and has no carry in. The truth table for a 2-bit RCA is shown in Table II. Fig. 1 depicts a classical netlist implementation using a full adder (consisting of the two half adders g_1, g_2 and g_5, g_6 and OR gate g_7) and another half adder (gates g_3, g_4) for the lower bits.*

If we now consider an external constraint that operand b is restricted to only even numbers, formally $b_0 = 0$, the circuit can be deeply optimized. The result is shown in Fig. 2. Note that in the truth table (Table II), only the light blue lines remain under this external constraint. Let us consider the AND gate g_3 in Fig. 1 as an example. For this gate the rule R_AND_Y0 holds, hence g_3 is redundant and can be removed; as a result the output is connected to the 0 constant.

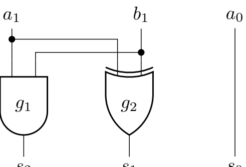

Fig. 2: Optimized 2-bit Ripple-Carry Adder under external constraint $b_0 = 0$

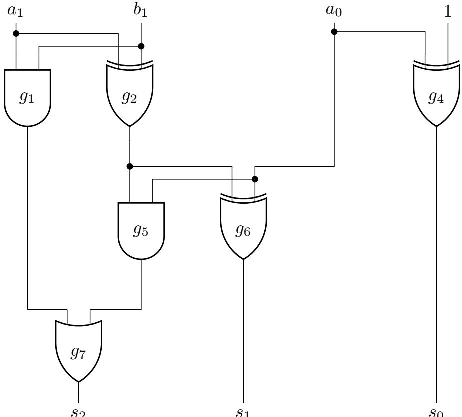

Fig. 3: Optimized 2-bit Ripple-Carry Adder under external constraint $b_0 = 1$

Another example is the XOR gate g_4 with inputs $B = a_0$ and $A = b_0$. Due to the external constraint $b_0 = 0$, which means $\overline{A} = \overline{b_0} = \overline{0} = 1$, rule R_XOR_YB holds and therefore a_0 is directly connected to s_0. The same argumentation holds for gate g_6 and thus the output of g_2 goes directly to s_1. Finally, for the OR gate g_7 the rule R_OR_YB holds and therefore the output of g_1 is connected to s_2. Please note as we use PC, so formal reasoning, the order in which we check the rules is not relevant for correctness.

Example 2. *Again, we consider a 2-bit RCA. However, we now have the external constraint that operand b is restricted to only odd numbers ($b_0 = 1$). In this case only a single gate (gate g_3) can be replaced by one of it's inputs (a_0) since rule R_AND_YB holds for g_3. The result is shown in Fig. 3.*

Both examples show that the RCA can be optimized under external constraints. As expected, the number of redundant gates depends on the circuit and the concrete external constraints.

We introduce an alternative formulation for the optimization problem at hand. The concept of the presented rules remains the same. However, instead of utilizing PC and describing the rules as properties, we now apply each rule for the gate g_i and create an EC problem. We can formulate the core step of the optimization problem under external constraints, the

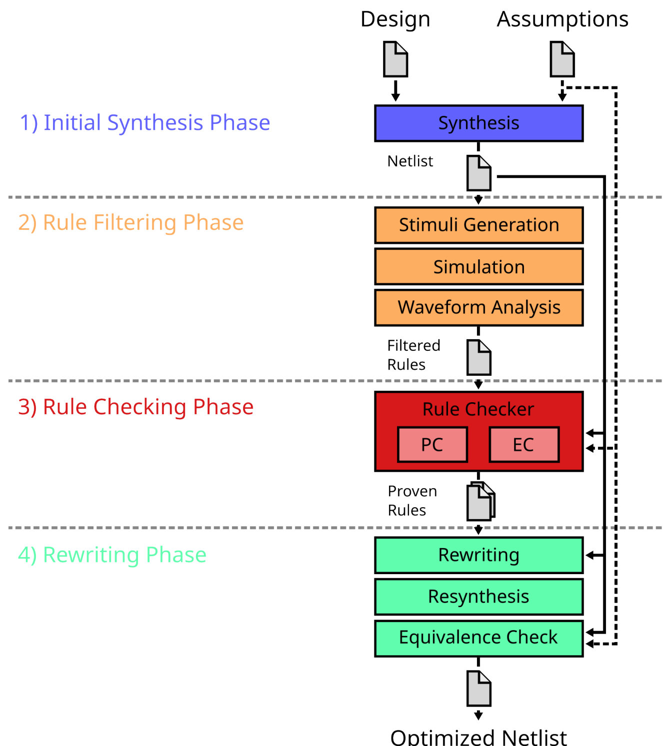

Fig. 4: The high-level flow of FSYN

gate-redundancy check for gate g_i utilizing EC as follows:

$$G_{EC}(g_i) = \{E \wedge Miter(N, \hat{N}(g_i)) \mid \hat{N}(g_i) \in Rules(type(g_i))\} \quad (2)$$

where E are the external constraints and $Miter(N, \hat{N}(g_i))$ forms a miter. This miter is built for each rule of gate g_i of $type(g_i)$ by applying the rule on the netlist (for instance by replacing the output of g_i by a constant or connect the output of g_i to the input A). Recall, if g_i is an AND gate, then four different miters are created. For the other gates, see Table I. Also in this formalization, for optimizing the complete netlist N all EC problems for each gate have to be solved.

The PC and EC formalizations presented so far allow to optimize a given netlist under external constraints. It is evident that the underlying formal instances can be solved concurrently. In the following section we introduce our implementation FSYN and discuss several improvements for scalability.

III. FSYN

In this section, we first present the overall flow of FSYN Thereafter, we describe the major phases of FSYN in the respective subsections.

A. FSYN Flow

A high-level flow of FSYN is depicted in Fig. 4. To perform optimizations on a design using FSYN two inputs are required, the *design* that should be optimized and a file containing the input *assumptions* (XSDCs) that form the basis for the optimization.

In general, FSYN is split into four distinct phases. In the first *Initial Synthesis Phase* (blue), the design is synthesized into a netlist. This phase serves two main purposes: First, it brings the design to the gate level at which FSYN operates and second, it unifies the netlist into a known and fixed format, which makes sure that all netlist components are supported by FSYN. The resulting netlist is used by all following steps of the FSYN flow.

In the second *Rule Filtering Phase* (orange), FSYN filters the list of all possible optimization rules. As a rule of thumb, there are 4 possible optimizations for every gate in the netlist. Since formal approaches can run into capacity problems with increasing design size, we employ a simulation-based approach to eliminate as many rules as possible by disproving their validity through random simulation. The rule filtering is performed in three steps: First, random input stimuli, which fulfill assumptions, are generated. Then, the stimuli are simulated on the netlist for a certain period of time. Finally, all gates in the resulting waveform are analyzed to find optimization rules that have been refuted. The result of the second phase is a list of filtered optimization rules which have to be formally checked by the formal *Rule Checker*.

The next phase is the *Rule Checking Phase* (red). The previous filtering phase could only disqualify rules based on counterexamples, not prove that a rule is a valid optimization. For this, the rule checking phase employs the formal checks which are the core of FSYN. The rule checker supports two alternative rule checking backends,. Broadly speaking, the PC flow attaches properties to all gates which are then checked and the EC flow directly performs the optimization and checks if the resulting netlist is still equivalent to the original netlist under the assumptions. The result of the third phase is a list of proven optimizations.

Finally, in the *Rewriting Phase* (teal), the netlist is rewritten by applying the proven rules to it. After the netlist is rewritten, it is synthesized again to perform some clean-up work (e.g., unnecessary assignment chains, which can be generated by FSYN, are shortened). As a last step, a final equivalence check is performed between the optimized netlist and the synthesized netlist.

B. Rule Filtering Phase

Applying an optimization rule to a netlist requires proving that the corresponding condition of the rule (cf. Table I) always holds under the external constraints. As explained in Section II-B, FSYN uses formal methods to perform this check. However, proving can be much more costly than refuting. For the latter, only a counter-example is needed, which in the best case can be easily found by simulation. In the context of this work, there is just one challenge, namely that the stimuli we use for simulation must fulfill the external constraints (given as assumptions). We formulate the problem of finding valid stimuli as an SMT problem over the inputs of the design.

This SMT problem consists of the external constraints and an always failing assertion. When solving the problem, we get a counter-example with a set of valid inputs that fulfill the external constraints. To generate multiple inputs we update the problem to exclude already seen input combinations using the technique described in [21].

We then use the generated stimuli to simulate the netlist to produce waveforms with the behavior of every gate in the design during the simulations. Now, for every gate and every rule we try to find a timestamp inside the waveform using the *Waveform Analysis Language* (WAL) [15], [22], [23] at which the rule is refuted.

An exemplary WAL expression that checks if the constant 0 rule is refuted is shown in the following listing:

```
(> 0 (count (! (= top.s[2] 0))))
```

WAL uses a Lisp style prefix notation. In the concrete WAL expression, we count[3] how often the signal `top.s[2]` is not 0 on the waveform, in other words if we see at least a single 1, the rule is refuted. For other rules, the subexpression `(= top.s[2] 0)` would be replaced accordingly, and the outer expression remains unchanged.

In FSYN, the rule filtering phase is optional and can be used to reduce the number of required formal proofs. The gate filtering also employs an SMT solver, however, compared to the PC and EC problems in the third phase, the stimuli generation scales only with the number of inputs, not with the number of gates. Analyzing the simulation waveform, the second step of the rule filtering phase, scales in theory quadratically with the number of gates in the design and the number of stimuli generated. However, even generating only a small number of stimuli is often sufficient to eliminate a large portion of the gates and to significantly reduce the checking time In Section IV-A, we will demonstrate the effectiveness of the rule filtering approach.

C. Rule Checking Phase

In this section, we describe the two rule checking implementations of FSYN. FSYN's architecture allows having multiple backends for the rule checking phase. Currently, two backends are implemented: PC and EC. The PC backend utilizes sby, a frontend for the formal verification capabilities of Yosys [13], for solving the properties. The checking of properties is split into many smaller tasks. Each task is a subset of rules for which properties will be checked in a single call to sby of Yosys.

The standard configuration is a task size of 80 gates (see analysis in [10]) with at most 4 properties/rules to be checked per gate, i.e. a single PC instance contains up to 320 properties. These tasks can be parallelized by leveraging the available threads of the local machine as well as to other computers. For the details on the PC backend we refer the reader to [10].

The EC backend is currently in an experimental stage. As presented in Section II-B, the EC backend rewrites the netlist eagerly and then checks if the rewritten netlist is equivalent to the original netlist using Yosys equivalence checking. The EC backend allows local parallelisation, however, it has yet to be integrated into the distributed system presented in [10].

[3] `(count expr)` counts and returns at how many timestamps the expression `expr` evaluates to true.

TABLE III: Rule Filtering Results

Benchmark	arbiter	mem_ctrl	32x32 Mul
Rules	47,356	161,844	27,216
Runtime PC 0 [s]	102	1,017	85
After filtering 40	1,339	49,859	12,038
Filtering time 40 [s]	21	97	22
Runtime PC 40 [s]	14	346	41
Runtime Sum 40 [s]	35	443	63

D. Rewriting Phase

The rewriting phase is the final phase of FSYN. In this phase, the proven rules are applied to the netlist. The rewritten netlist is then optimized again using Yosys to clean up remnants of the rewriting phase. For example, FSYN's rewriting can leave chains of assignments ($a = b = c$) which can be removed by Yosys. Finally, an equivalence check between the cleaned rewritten netlist and the original synthesized netlist is performed.

IV. Experiments

In this section, we present two experiments. First, we report results in combination with our novel rule filtering (Section IV-A). Second, we present first results with the EC backend (Section IV-B).

A. Rule Filtering

In this section, we evaluate the new rule filtering phase of FSYN in the PC flow. All experiments were conducted using an AMD EPYC 7713 64-Core Processor with 128 threads and 256 GB RAM.

Table III shows experimental results for runs of FSYN with and without the new rule filtering. The first row shows the benchmark on which we applied FSYN. The first two columns are the arbiter and mem_ctrl benchmarks from the EPFL benchmark suite [24]. We randomly generated external constraints for them. The last column is an unsigned 32×32 bit multiplier (stages: simple partial product generator, array, Carry look-ahead adder) generated with GenMul[4] [25]. As external constraint for the multiplier we set the lower 4 bit of the first input to 0.

The second row shows the number of all rules that have to be checked without the rule filtering. The third row shows the time it takes to check all rules without the rule filtering.

Next, in the fourth row, the number of rules which remained after the filtering (with 40 generated stimuli) is shown. The time the rule filtering took is shown in the fifth row and the runtime to check the remaining rules is shown in the sixth row. Finally, the combined runtime of the rule filtering and the checking of remaining rules is reported in the last row.

[4] http://www.sca-verification.org/genmul

TABLE IV: EC Analysis for 32×32 Multiplier

Gate	Result	PC Runtime [s]	EC Runtime [s]
N13330	abort	>10,800	8
N13063	holds	871	7
N13125	holds	1,805	7
N13228	abort	>10,800	7

Enabling rule filtering significantly reduces the number of rules that need to be formally proven, resulting in moderate (multiplier) or major (arbiter, mem_ctrl) runtime improvements.

B. FSYN with EC

In this section, we describe first experiments when utilizing the EC backend of FSYN. As a benchmark we consider the same 32×32 bit multiplier as in the previous section with the identical external constraints (set lower 4 bit of the first input to 0). We analyzed the runtimes of the different PC problems in a configuration where we considered single PC problems and configured the time-out for solving a PC instance to 10 minutes (i.e. after that time the respective rule will be considered as failing, so we assume no optimization of the gate in terms of the rule can be performed). This timeout occurred for 12 out of 27,216 PC instances. Table IV shows runtime data for some of these instances. The first column gives the line number of the gate inside the netlist. The second column reports the checking results for the rule. The third column shows the runtime of the PC problem while the last column lists the runtime when employing EC. As can be seen, there are PC instances which still can be solved by increasing the time-out. However, there is even an instance which did not finish in more than 3 hours. Please note that all of these problems can be solved using EC within $7 - 8$ seconds. Here, a deeper analysis is necessary.

V. Conclusions

In this paper, we proposed an optimization approach to reduce the number of gates under given external constraints (which can be viewed as external satisfiability don't cares) using formal verification methods. We have formalized the optimization problem in terms of PC and EC to check rules per gate. Depending on the validity of a gate's rule, it may either be removed as redundant, with its output connected directly to an input or a constant, or it may remain unchanged. Our implementation FSYN provides both PC and EC backends and leverages open-source tools for all steps, i.e. from synthesis to formal verification. Furthermore, we have demonstrated methods to accelerate the optimization process. This is achieved through parallelization of theoretically independent formal instances and a random-simulation based approach that analyzes waveforms to disprove rules.

Our experiments have shown that additional research is necessary for formal gate-level optimization. This involves conducting a more comprehensive analysis of the relationship between task size and rule filtering. Furthermore, different EC strategies as well as the interplay of EC and PC needs to be investigated. Finally, also other cost metrics and the potential in HW/SW optimization [26] is an interesting research direction.

Acknowledgments

This work has partially been supported by the German Research Foundation (DFG) within the project VerA (GR 3104/6-1) and by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References

[1] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis*. Kluwer Academic Publishers, 1984.

[2] D. Brand, "Redundancy and don't cares in logic synthesis," *TC*, vol. C-32, no. 10, pp. 947–952, 1983.

[3] H. Savoj and R. Brayton, "The use of observability and external don't cares for the simplification of multi-level networks," in *DAC*, 1990, pp. 297–301.

[4] S. Lee, H. Riener, and G. D. Micheli, "External don't cares in logic synthesis," in *Int'l Workshop on Boolean Problems*, 2022.

[5] A. Mishchenko and R. Brayton, "Simplification of non-deterministic multi-valued networks," in *ICCAD*, 2002, p. 557–562.

[6] N. Bleier, J. Sartori, and R. Kumar, "Property-driven automatic generation of reduced-isa hardware," in *DAC*, 2021, pp. 349–354.

[7] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, 1999, pp. 193–207.

[8] M. D. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz, "Unbounded protocol compliance verification using interval property checking with invariants," *TCAD*, vol. 27, no. 11, pp. 2068–2082, 2008.

[9] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *FMCAD*, 2000, pp. 108–125.

[10] L. Klemmer, D. Bonora, and D. Große, "Large-scale gatelevel optimization leveraging property checking," in *DVCon Europe*, 2023.

[11] J. P. Marques-Silva and T. Glass, "Combinational equivalence checking using boolean satisfiability and recursive learning," in *DATE*, 1999, pp. 145–149.

[12] P. Molitor and J. Mohnke, *Equivalence checking of digital circuits: fundamentals, principles, methods*. Springer Science & Business Media, 2007.

[13] C. Wolf, "Yosys open synthesis suite," https://yosyshq.net/yosys/.

[14] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *CAV*, 2010, pp. 24–40.

[15] L. Klemmer and D. Große, "WAL: a novel waveform analysis language for advanced design understanding and debugging," in *ASP-DAC*, 2022, pp. 358–364.

[16] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008, pp. 337–340, available at https://github.com/Z3Prover/z3.

[17] J.-H. R. Jiang and S. Devadas, "Chapter 6 - logic synthesis in a nutshell," in *Electronic Design Automation*, L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, Eds. Boston: Morgan Kaufmann, 2009, pp. 299–404.

[18] K. Bartlett, R. Brayton, G. Hachtel, R. Jacoby, C. Morrison, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Multi-level logic minimization using implicit don't cares," *TCAD*, vol. 7, no. 6, pp. 723–740, 1988.

[19] S. Muroga, Y. Kambayashi, H. Lai, and J. Culliney, "The transduction method-design of logic networks based on permissible functions," *TC*, vol. 38, no. 10, pp. 1404–1424, 1989.

[20] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization," in *DATE*, 2005, pp. 412–417.

[21] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable and nearly uniform generator of SAT witnesses," in *CAV*, 2013, pp. 608–623.

[22] L. Klemmer and D. Große, "Waveform-based performance analysis of RISC-V processors: late breaking results," in *DAC*, 2022, pp. 1404–1405.

[23] F. Skarman, L. Klemmer, O. Gustafsson, and D. Große, "Enhancing compiler-driven HDL design with automatic waveform analysis," in *FDL*, 2023, pp. 1–8.

[24] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *IWLS*, 2015.

[25] A. Mahzoon, D. Große, and R. Drechsler, "GenMul: Generating architecturally complex multipliers to challenge formal verification tools," in *Recent Findings in Boolean Techniques*, R. Drechsler and D. Große, Eds. Springer, 2021, pp. 177–191.

[26] S. G. Sørensen, C. Bartsch, D. Stoffel, and W. Kunz, "Generation of formal CPU profiles for embedded systems," in *VLSI-SoC*, 2022, pp. 1–6.