The Thirty-Seventh AAAI Conference on Artificial Intelligence (AAAI-23)

Circuit Minimization with QBF-Based Exact Synthesis

Franz-Xaver Reichl, Friedrich Slivovsky, Stefan Szeider

Algorithms and Complexity Group, TU Wien, Vienna, Austria
{freichl,fs,sz}@ac.tuwien.ac.at

Abstract

This paper presents a rewriting method for Boolean circuits that minimizes small subcircuits with exact synthesis. Individual synthesis tasks are encoded as Quantified Boolean Formulas (QBFs) that capture the full flexibility for implementing multi-output subcircuits. This is in contrast to SAT-based resynthesis, where "don't cares" are computed for an individual gate, and replacements are confined to the circuitry used exclusively by that gate. An implementation of our method achieved substantial size reductions compared to state-of-the-art methods across a wide range of benchmark circuits.

Introduction

Modern integrated circuits are so large and complex that their design would be impossible without a significant degree of automation. This includes the automated improvement of circuits (logic optimization), and the automated creation of circuits from specifications (logic synthesis), which jointly yield substantial reductions in the number of gates and circuit depth (De Micheli 1994; Brayton, Hachtel, and Sangiovanni-Vincentelli 1990).

Finding a small circuit implementing a given Boolean function is computationally intractable,[1] and *exact synthesis*, which yields provably optimal results, currently does not scale beyond circuits of about 10 gates (Kojevnikov, Kulikov, and Yaroslavtsev 2009; Haaswijk et al. 2020). Exact methods can nevertheless be applied to large circuits through so-called peephole optimization. This involves partitioning a circuit into small subcircuits amenable to exact minimization and replacing each subcircuit with a size-optimum implementation of its output function (Testa et al. 2020). Such an implementation can either be obtained from a pre-computed database of optimal circuits for functions with up to 5 input variables (Mishchenko, Chatterjee, and Brayton 2006) or on-the-fly by exact synthesis for larger subcircuits (Riener et al. 2019). Crucially, the new implementation need not be logically equivalent to the subcircuit it replaces since its outputs may be irrelevant under certain assignments of inputs. This is either because a particular combination of inputs can never arise in the context of the larger circuit or because other signals mask the output. Exploiting such *don't cares* can help to significantly reduce the size of subcircuits (Savoj and Brayton 1990; Savoj 1992; Mishchenko and Brayton 2005).

Don't cares can be enumerated using a Boolean satisfiability (SAT) solver (Mishchenko and Brayton 2005) or (for circuits with few inputs) by simulation. In subsequent resynthesis, the equivalence of the new implementation and the original subcircuit is not required for don't care assignments. This approach is typically very efficient for single-output subcircuits, and can be used to minimize multi-output subcircuits by considering one output gate and its exclusive circuitry at a time (Riener et al. 2022), but that does not exploit the full implementation flexibility.

Contribution We introduce a new approach capable of optimally resynthesizing multi-output subcircuits. Individual synthesis tasks are encoded as Quantified Boolean Formulas (QBFs) that fully capture implementational flexibility. The encoding is succinct and handles don't cares implicitly, obviating the need for an explicit enumeration.

We use QBF-based resynthesis as a subroutine in an optimization algorithm that scales to circuits of up to several thousand gates. A bespoke local selection strategy repeatedly identifies promising subcircuits for resynthesis. Each synthesis task is encoded as a QBF and passed to an off-the-shelf solver (Janota 2018). A time budget is allocated for individual solver calls to keep overall running time within reason. Since solving times increase with the size of the overall circuit, this timeout is adjusted dynamically based on previously observed running times. The entire circuit is minimized at regular intervals by a sequence of fast logic optimization techniques implemented in `ABC` (Brayton and Mishchenko 2010). This not only further shrinks the circuit, but also sets up new opportunities for exact resynthesis.

Results In an experimental evaluation, the minimization strategy described above achieved substantial size reductions compared to state-of-the-art methods on the following two sets of benchmarks:

- The EPFL combinational benchmark suite was designed to test logic optimization and synthesis tools (Amarú,

[1] Curiously, the *minimum circuit size* problem is not known to be NP-hard. In fact, an NP-hardness proof by a "simple" reduction would settle open problems in computational complexity (Murray and Williams 2017).

Gaillardon, and Micheli 2015). An online repository[2] maintains the smallest implementations found so far. Compared to the best circuits at the time of writing, our approach achieved significant reductions for several small and medium-size circuits (see Tbl. 3). For some of the circuits in question, this was the first improvement seen in years.

- The IWLS'22 programming contest[3] asked participants to synthesize small circuits for single and multi-output Boolean functions given as truth tables. Here, an implementation of our method was able to find the smallest And-Inverter Graph (AIG) for more functions than all competing entries.

Our results demonstrate the potential of applying QBF solvers to challenging design automation tasks. In this instance, QBFs not only allow for a succinct and natural encoding, but unlock new optimization opportunities that help tackle a challenging and practically relevant problem.

Related Work

Methods that fully capture the properties of Boolean functions implemented by circuits (rather than viewing them as polynomials, for instance) are considered the most effective in logic synthesis (Testa et al. 2020). At the same time, they are the most computationally expensive and can only be applied to large circuits as a means to resynthesize small subcircuits (sometimes referred to as *windows*). In particular, that is the case for SAT-based exact synthesis (Haaswijk et al. 2020; Kojevnikov, Kulikov, and Yaroslavtsev 2009) as part of resynthesis workflows (Riener et al. 2019), and for SAT-based resubstitution, which seeks to express the function implemented by a particular gate as a function of few gates already present in the circuit (Mishchenko et al. 2011; Riener, Mishchenko, and Soeken 2020). Since an exhaustive treatment of the wealth of window-based approaches developed in logic synthesis is beyond the scope of this section, we refer to the works cited above and the references therein. Many of these techniques are implemented in the industrial-strength tool `ABC` (Brayton and Mishchenko 2010).

Exact resynthesis of subcircuits has also been considered to find optimum circuits for symmetric functions studied in circuit complexity, albeit without incorporating don't cares (Kulikov, Pechenev, and Slezkin 2022). This is an instance of *SAT-based Local Improvement Method (SLIM)*, a general optimization framework that has recently been applied to several problems in AI and involves repeatedly replacing parts of a structure with optimal solutions computed by a SAT or MaxSAT solver (Fichte, Lodha, and Szeider 2017; Lodha, Ordyniak, and Szeider 2019; Ramaswamy and Szeider 2021, 2022; Schidler and Szeider 2021). The main difference to our local resynthesis approach is that consistency of replacements in previous instantiations of SLIM is ensured by purely local constraints. In contrast, our QBF encoding expresses correctness globally by requiring equivalence of the functions computed by the entire circuit before and after resynthesis.

In the context of logic synthesis, QBFs have been used for bi-decomposition (Chen, Janota, and Marques-Silva 2012), synthesis of reversible quantum circuits (Wille et al. 2008), and synthesis of lookup tables (LUTs) (Fujita et al. 2013; Fujita 2015; Fujita et al. 2020). The latter two problems are more constrained than the setting considered here, in that the topology of the circuits is fixed, whereas the synthesis tasks we encode as QBF also involve deriving a suitable topology.

Preliminaries

Boolean Chains To represent Boolean functions, we use an extension of *Boolean Chains* where each step has exactly k inputs, where $k \geq 2$ is arbitrary but fixed (Knuth 2011; Haaswijk et al. 2020). Let $f : \mathbb{B}^n \to \mathbb{B}^m$ be a Boolean function. For $k \geq 2$, a *k-input Boolean chain of length ℓ* is a sequence $x_1, \ldots, x_{n+\ell}$, where the elements $x_1, \ldots, x_n$ are the inputs of the chain and the remaining elements are denoted as *steps* or *gates*. Whenever the number k is understood, we simply use the term *Boolean chain*. Each step x_{n+i} is a Boolean function that takes k inputs from the set $\{x_1, \ldots, x_{n+i-1}\}$. That is $x_{n+i} = g_i(x_{j_1}, \ldots, x_{j_k})$, where $1 \leq j_1, \ldots, j_k \leq n + i - 1$ and g_i is a k-ary Boolean function. The constant value false is represented by an additional element of the chain x_0. Moreover, for each output index $1 \leq i \leq m$ there needs to be $0 \leq j \leq n + t$ such that $f_i(x_1, \ldots, x_n) = x_j$ or $f_i(x_1, \ldots, x_n) = \neg x_j$. We write $\mathit{steps}(C)$ for the *set of steps* in a Boolean chain C. For $x, y \in \mathit{steps}(C)$ we say that x *depends* on y if y is an input of x or one of the inputs of x depends on y. For a set S of steps in C the *set of successors* of S in C is given by $\mathit{successors}(C, S) = \{x \in \mathit{steps}(C) \mid \exists y \in S.\, x \text{ depends on } y\}$.

A Boolean chain is *optimal* if there is no shorter chain representing the same function. A Boolean function is *normal* if it yields false whenever all its inputs are false. A Boolean chain is *normal* if all of its steps are normal functions. It can be shown that if a normal Boolean function f can be represented by a chain of length ℓ then f can also be represented by a normal chain of length ℓ.[4]

Quantified Boolean Formulas A *quantified Boolean formula (QBF)* $\Phi = \mathbf{Q}.\varphi$ consists of a (quantifier) prefix $\mathbf{Q}$ and a propositional formula φ, called the matrix of Φ. The quantifier prefix is a sequence $\mathbf{Q} = Q_1 X_1 \ldots Q_n X_n$ where the $Q_i \in \{\exists, \forall\}$ are existential ($\exists$) or universal ($\forall$) quantifiers and the X_i are (pairwise disjoint) sets of propositional variables referred to as quantifier blocks. We assume that formulas are closed, so that each variable of φ appears in a quantifier block. The semantics of QBFs can be defined in terms of an evaluation game played between an existential and a universal player. The players take turns choosing variable assignments $\sigma_i : X_i \to \{0, 1\}$ for their quantifier blocks, following the order of the prefix. The existential player wins if the resulting variable assignment satisfies the matrix, and the universal player wins if it falsifies the matrix.

[2] https://github.com/lsils/benchmarks

[3] https://github.com/alanminko/iwls2022-ls-contest

[4] For single output functions we refer to Vol. 4 of *The Art of Computer Programming* (Knuth 2011); the multi-output case is a generalization of the single-output case.

If the existential player has a winning strategy in this game, the QBF is true (or satisfiable); otherwise it is false (or unsatisfiable). The satisfiability problem of QBFs is PSPACE-complete, and QBFs can succinctly encode problems arising in many areas (Shukla et al. 2019). For an overview of QBF, including solving techniques and proof complexity, see the survey by Beyersdorff et al. (2021).

QBF-Based Exact Synthesis

In the following, we will present a QBF-based approach for synthesizing optimal chains from given Boolean functions.

The core idea is to use a QBF encoding to check whether it is possible to represent a Boolean function by a chain of a given length ℓ. If a function can be represented by a chain of length ℓ then the encoding is satisfiable. In this case a realization of the chain can be read off from the model of the encoding. To find an optimal chain we increment the length ℓ until we find a length that allows representing the given function.

Encoding

We will discuss a QBF encoding to check whether a Boolean function can be represented by a k-input Boolean chain of length ℓ. To simplify the presentation, we fix $k = 2$; a generalization to $k > 2$ is straightforward. The encoding is adapted from the *multi selection variable* SAT-encoding (Haaswijk et al. 2020) to exploit the succinctness of QBFs. In particular, we make the following changes:

- By universally quantifying over the inputs, we can represent the values of the individual gates by single existentially quantified variables (in the scope of the universal quantifiers).
- Instead of a truth table, the specification is given as a circuit. In principle, this allows us to synthesize even functions with prohibitively large truth tables. Moreover, this makes it easier to iteratively select and replace subcircuits.

To simplify the encoding, we only consider normal functions. This means that it suffices to only consider normal steps and unnegated outputs. This is without loss of generality, since one can always first normalize a given function and then negate the appropriate outputs in the synthesized chain to obtain the original function.

First, we will introduce the variables used in the encoding. For this purpose, let n be the number of inputs and m the number of outputs of the Boolean function. Moreover, let $1 \leq i \leq \ell$ and $1 \leq j \leq m$.

Selection variables $S_i = \{s_{it} \mid 1 \leq t < i + n\}$. These variables determine the inputs of the i^{th} step. If s_{it} is true and $t \leq n$ then step i depends on the t^{th} input of the chain. If $t > n$ then step i depends on step $t - n$.

Gate definition variables $F_i = \{f^i_{a_1,a_2} \mid 0 \leq a_1 \leq 1, 0 < a_1 + a_2 \leq 2\}$. These variables describe the Boolean function at step i. If $f^i_{a_1,a_2}$ is true then the function yields true for the inputs a_1, a_2, and false otherwise. As we are only considering normal chains we do not need to consider the case $a_1 = 0$ and $a_2 = 0$.

Output variables $O_j = \{o_{tj} \mid 0 \leq t \leq n + \ell\}$. These variables determine the outputs of the chain. If the variable o_{0j} is true then output j is the constant value false. If o_{tj} for $1 \leq t \leq n$ is true then the j^{th} output is given by input t. Finally, if $n < t \leq n + \ell$ and o_{tj} is true then step $t - n$ represents the j^{th} output.

Input variables $I = \{x_t \mid 1 \leq t \leq n\}$. These variables represent the inputs of the chain.

Gate value variables $G = \{g_t \mid 1 \leq t \leq \ell\}$. The variable g_t represent the value of step t under a given assignment to the inputs and an assignment for the gate definition and selection variables.

We define $S = \bigcup_{1 \leq i \leq \ell} S_i$. Similarly, we define the sets F and O.

The matrix of the QBF expresses the subsequent constraints:

- Each step must have exactly k inputs, i.e. at each step exactly k selection variables must be true. We denote the constraint that asserts that λ variables out of a set Var, where $\lambda \in \mathbb{N}$, are assigned to true by $Count(Var, \lambda)$. This constraint can be realized by using a sequential counter (Sinz 2005). Thus, for each $1 \leq i \leq \ell$ we require $Count(S_i, k)$.
- Each output must be uniquely defined, i.e., for each $1 \leq j \leq m$ there is exactly one i such that o_{ij} is true. Thus, for each j we require $Count(O_j, 1)$.
- The assignment of the gate value variables has to be compatible with the assignment for the gate definition variables. Let $1 \leq i \leq n$, $1 \leq u < v < n+i$, $0 \leq a \leq 1$ and $0 < a + b \leq 2$. To establish the compatibility we require $(s_{iu} \wedge s_{iv} \wedge g_u = a \wedge g_v = b) \Rightarrow (g_i = f^i_{ab})$. If $u \leq n$ ($v \leq n$) we replace g_u (g_v) by the corresponding input variable. We denote the conjunction over all u, v, a, b of the above formula by $Comp_i$.
- Let $f_{spec}(x_1, \ldots, x_n)$ denote the function that is to be synthesized. We now require that the chain and the function are equivalent, i.e. they give the same outputs. For this purpose, let $0 \leq i \leq n + \ell$ and $1 \leq j \leq m$. We have to differentiate between three cases: If $i = 0$ we require $o_{ij} \Rightarrow \neg f^j_{spec}(x_1, \ldots, x_n)$. If $1 \leq i \leq n$ we require $o_{ij} \Rightarrow (x_i = f^j_{spec}(x_1, \ldots, x_n))$. Finally, if $n < i \leq n + \ell$ we require $o_{ij} \Rightarrow (g_{i-n} = f^j_{spec}(x_1, \ldots, x_n))$. We denote the conjunction of the above formulas by $Corr_j$.

The encoding is then given by:

$$\exists S, F, O \, \forall I \, \exists G. \bigwedge_{1 \leq i \leq \ell} (Count(S_i, k) \wedge Comp_i) \wedge \bigwedge_{1 \leq j \leq m} (Corr_j \wedge Count(O_j, 1)).$$

Symmetry Breaking

For SAT-based exact synthesis, it was shown (Haaswijk et al. 2020) that symmetry breaking has a significant impact on performance. For this reason, we use a selection of the symmetry-breaking constraints in our QBF encoding. In the following, we provide a brief summary; for a detailed

description, we refer to the original source (Haaswijk et al. 2020).

- Trivial steps in chains are not allowed. This means that a gate must not be a projection of its inputs, and it must not represent a constant value.
- A step is either an output or an input of another step.
- Operations used to represent a step must not be reused. If gate x has inputs a and b and x is an input of gate y then neither a nor b must be an input of y.
- The gates are required to be ordered colexicographically according to their inputs.

Circuit Optimization with Resynthesis

In the following, we describe our new circuit minimization procedure that uses the encoding from the previous section to resynthesize small subcircuits. Because each subcircuit interacts with the rest of the circuit only through a small number of gates, its underlying function can often be simplified without affecting the behavior of the larger circuit. This is where QBFs offer a distinct advantage over propositional logic, since they allow us to fully capture this kind of flexibility in a succinct way.

Starting from an initial circuit obtained by `ABC`, a local selection heuristic repeatedly finds a subcircuit for resynthesis. Each resynthesis problem is encoded as a QBF, which requires additional constraints to ensure consistency with the remaining circuit.

Adapting the Encoding

The QBF encoding used for resynthesis is obtained by applying two major adaptations to the previous encoding. First, new equivalence constraints are used and second additional constraints that rule out cycles are added.

Equivalence Constraints Unlike the encoding presented before we now do not require that the replaced subcircuit and the original subcircuit are equivalent. Instead, we ensure that after replacing the subcircuit the whole circuit is equivalent to the original one.

In the following, we will denote the Boolean chain representing the specification by $f(i_1, \ldots, i_n)$ and the subcircuit that shall be replaced by C. Moreover, $c_1, \ldots, c_p$ denote the outputs of C, where p is the number of outputs of C. For the sake of simplicity, we assume that no input of C depends on an output of C in f. While such dependencies require special attention for cycle prevention, the equivalence constraints can easily be generalized to this case.

To state the equivalence constraint, we introduce a new chain f'. The inputs of f' are given by $i_1, \ldots, i_n, c_1, \ldots, c_p$. The steps of f' are obtained by removing the steps of C from f. We can see that f' is well-defined as for each output c of C, f' has an input c. We let F denote the set of outputs of f and write Ω for the set $successors(f, C) \cap F$. To check if replacing C by a new realization preserves equivalence we can now proceed as follows. We have to consider each assignment σ of $i_1, \ldots, i_n$. Then we compute the assignment γ of the inputs of C in f under σ. Next we determine the assignment ρ of $c_1, \ldots, c_p$ of the new realization of C under γ. We then require that $f_x(\sigma(i_1), \ldots, \sigma(i_n)) = f'_x(\sigma(i_1), \ldots, \sigma(i_n), \rho(c_1), \ldots, \rho(c_p))$ for each $x \in \Omega$.

To obtain the assignment ρ in our encoding, we introduce for each $1 \leq i \leq p$ a new variable ov_i—which we call *output value variable*. These variables give the value of the synthesized subcircuit for the output c_i under an assignment to the inputs of the subcircuit. This means we add for each output value variable ov_i and for each variable $o_{ji} \in O_i$ the constraint: $o_{ji} \Rightarrow (ov_i = g_j)$. For the output variables that indicate that the output is an input, or a constant, the constraint is adapted accordingly. We denote the set of all output value variables by OV.

Finally, we add for each output $x \in \Omega$ the constraint $f_x(i_1, \ldots, i_n) = f'_x(i_1, \ldots, i_n, ov_1, \ldots, ov_p)$. We denote the conjunction over all $x \in \Omega$ of the above formula by $Corr$. Note that in the actual encoding we reuse steps contained in both f and f'. Additionally, we can omit outputs of the chains that do not have an effect on outputs in Ω.

Ensuring Acyclicity While the above adaptations to the original encoding are straightforward, the usage of multi-output subcircuits require a bit more subtle changes. As we allow multi-output subcircuits, there may be a path of an output of the subcircuit to an input of the subcircuit that is not contained in the subcircuit. This is illustrated in Fig. 1a. In the image, the outer box represents the whole circuit that has six inputs and three outputs. The inner box represents the selected subcircuit. We can see that there is a path from an output to an input of the subcircuit that is not contained in the subcircuit.

If we would replace such subcircuits naively cycles could arise after replacing the subcircuit. Let us again consider the example from above: If we replace the subcircuit we could obtain a circuit where the output g_1 depends on i—this is illustrated in Fig. 1b.

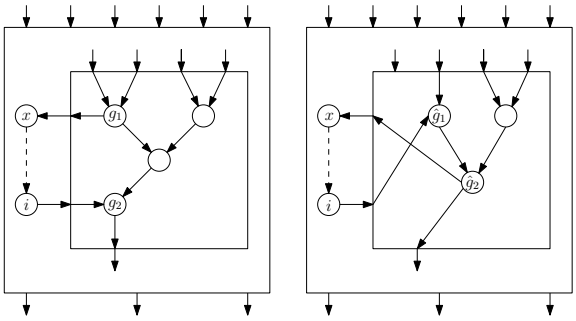
This means that we have to adapt the encoding to rule such cycles out. For this purpose, we compute the set P of all pairs of inputs and outputs of the subcircuit (o, i), s.t. there is a path from o to i that is not contained in the subcircuit. If P is empty we do not have to add any further constraint to rule out cycles. Otherwise, let $I = \{i \mid (o, i) \in P\}$, i.e. the set of all inputs that occur in at least one pair. Next, we add for each gate g in the synthesized subcircuit and each input i in I a *Connection variable* c_{ig}. We denote the set of all connection variables by C. We require that c_{ig} is assigned to true if gate g depends on input i. Finally, let $(o, i) \in P$ and let out be the gate output variable that asserts that gate g serves as output o. Then we require that $out \Rightarrow \neg c_{ig}$. Similarly, we can restrict the gate output variables that assert that an input serves as an output. We denote the cycle constraint by Cyc.

The adapted encoding is then given by:

$$\exists S, F, O\ \forall I\ \exists G, C, OV.\ Corr \wedge Cyc \wedge \bigwedge_{1 \leq j \leq m} Count(O_j, 1) \wedge \bigwedge_{1 \leq i \leq \ell} (Count(S_i, k) \wedge Comp_i).$$

Subcircuit Selection

The encoding presented above requires that a certain subcircuit is given. In order to obtain such a subcircuit we first



(a) Initial circuit (b) Replaced subcircuit

Figure 1: Subcircuit where an input is connected to an output by a path not contained in the subcircuit. Cycle obtained by naively replacing the subcircuit.

select a gate in the circuit—denoted as *root gate*. The root gate is the first gate in the subcircuit. To expand this subcircuit we add inputs of already selected gates to the subcircuit until the size of the subcircuit reaches a certain bound.

To obtain a root gate, we randomly select a gate from the circuit. As the same root gate should not be used repeatedly, we keep a *taboo list* that contains previously selected gates. Gates in this list are not allowed to be selected. As it may be of interest to analyze a gate again if the subcircuit could be reduced, we do not add the gate in this case. If the length of the list reaches a certain ratio of the total number of gates, we remove elements from the list in a first-in-last-out order.

For the expansion of the subcircuit, we evaluated different strategies, like breadth-first or depth-first search. The best results were obtained with a strategy that aims at keeping the number of outputs and, secondarily, the number of inputs small. For this purpose, we select the gate from the available inputs that adds the fewest outputs to the already selected gates. If there are multiple gates that yield the same number of outputs, we select the one that adds the fewest inputs to the selected subcircuit.

Combining the Components

To select and replace subcircuits, a bound for the size of the subcircuits is needed. The initial bound can be given as a parameter to our procedure. This initial bound can then either be increased if the resulting encodings, on average turn out to be easy or decreased if they turn out to be hard. To replace a selected subcircuit, we first check if the subcircuit can be realized by a same-sized circuit. Solving the corresponding resynthesis problem does not result in a smaller subcircuit and serves different purposes. First, as we know that the resulting QBF is true, the solver's running time allows us to adjust the size of subcircuits for resynthesis and calibrate the timeout for subsequent solver calls. In particular, if the solver is unable to find a solution within a more generous time limit, resynthesizing subcircuits of this size is likely infeasible. Second, even if the newly computed implementation is not smaller, its substitution modifies the circuit and can help escape local minima.

Experiments

The primary goal of the empirical evaluation is to assess the effectiveness of the presented techniques for exact synthesis and of circuit reduction. In particular, the evaluation aims to answer the following *research questions*:

RQ1: What is the overhead caused by using QBF instead of SAT for exact synthesis?

RQ2: How significant are the individual features of the base configuration (non-equivalent subcircuits, multi-output subcircuits, inprocessing by `ABC`, and replacements by same-sized subcircuits)?

RQ3: Is the presented approach effective for circuits with non-binary gates?

We implemented the presented synthesis method in Python.[5] Based on preliminary experiments, we selected *QFUN* (Janota 2018) as the backend QBF solver. All experiments were conducted on a cluster with Intel Xeon E5649 processors at 2.53 GHz running 64-bit Linux. Additionally, we used a memory limit of 4 GB.

Experimental Comparison of SAT and QBF Encoding

To assess the higher cost of using a QBF solver instead of a SAT solver (**RQ1**), we compared the runtime of our QBF-based approach with the SAT-based approach on selected benchmarks. To get results for the SAT-based approach, we applied the tool `Percy` (Soeken et al. 2022). For the comparison, we used the four and six-input functions that were previously used for the experimental evaluation of the SAT-based encoding (Haaswijk et al. 2020). As the five-input functions of the aforementioned evaluation were not available to use, we used randomly generated five-input functions instead. For each of the three benchmark families, we applied `Percy` with the optimal configuration according to (Haaswijk et al. 2020).

Tbl. 1 gives the results of the experiments. The first two columns give the worst and the average runtime needed to obtain the size-optimal circuit by using `Percy`, respectively. The remaining two columns give the corresponding information for the QBF-based approach. First, the results for the four-input functions realized by circuits with two input gates are given. Next, the table shows the results for five-input functions using three-input gates and six-input functions using four-input gates.

The table shows that in general the SAT-based approach is up to ten times faster than the QBF-based approach. Nevertheless, the table indicates that the SAT-based approach is not always faster—for the five-input functions, the SAT-based approach needs three times as long as the QBF-based approach in the worst case. The QBF-based approach allows us more flexibility to replace subcircuits, so the longer running time is well compensated.

Iterative Replacement

IWLS2022 To analyze the performance of the iterative replacement of subcircuits and the individual features of our

[5] https://github.com/fxreichl/ciops

	Best SAT		QBF	
Instances	Max time	Mean time	Max time	Mean time
4-input	6.1	0.5	41	5.3
5-input	548.3	7	168	21.7
6-input	1.3	0.5	10.2	3.5

Table 1: Comparison of the runtimes of the SAT-based and the QBF-based approaches for exact synthesis.

approach, we considered the benchmarks for the IWLS2022 programming contest[6]. The benchmarks consist of 100 instances representing Boolean functions with up to sixteen inputs and both single and multi-output functions. The goal is to compute an *And-Inverter Graph* (AIG) with as few gates as possible. As the instances are given as truth tables, and our tool requires that specifications are given as circuits, we have to preprocess the instances first. For this purpose, we apply the tool `ABC` (Brayton and Mishchenko 2010) to generate specifications in the *Berkeley Logic Interchange Format* (BLIF)[7]. Preliminary tests showed a naive transformation of truth tables to circuits by using `ABC` results in relatively large circuits. Thus, we use `ABC` to simplify the generated circuit as far as possible. For this purpose, we apply three combinations of `ABC` commands to the truth tables and select the resulting circuit with the smallest number of gates. As the result has to be an AIG, the encoding needs to be adapted slightly. In order to obtain an AIG, we forbid gates that are not valid AIG gates. As we are only using normal gates, this means that we have to ensure that no gate is a projection to one of its inputs and that no gate is the exclusive or. The non-trivial symmetry-breaking constraint already deals with the first issue. To also deal with the second issue, we add an additional constraint on the gate definition variables for each gate.

In our evaluation setup, we run our reduction tool for one hour, then we apply `ABC` as an inprocessing step using `ABC`-commands that turned out useful in preliminary tests. When we apply `ABC`, we repeat its application until no further improvements can be achieved. The combination of our tool and `ABC` is applied 10 times. We compare 5 different configurations of this approach:

Base As described above.

Equivalent Only replace by equivalent circuits.

1-output Only select subcircuits with a single output.

Reduce Only replace by smaller circuits.

No ABC Do not apply `ABC` between the individual runs.

Instances are grouped into four subsets of 25 based on the initial number of gates. For each configuration and instance group, we determine the mean size reduction (in %) for circuits in that group. We perform 5 independent runs of each configuration and report averages (the standard deviation across independent runs was no more than 0.35). Results are

[6]https://github.com/alanminko/iwls2022-ls-contest

[7]http://www.cs.columbia.edu/~cs6861/sis/blif/index.html

#Gates	Base	Equivalent	1-output	Reduce	No `ABC`
10-58	13.4	4.9	9.7	5.7	12.7
59-182	29.6	11.8	22.4	14.5	25.9
185-777	34.1	9.8	24.6	16.8	24.5
784-7920	20.2	7.0	15.9	9.7	10.2
Overall	24.3	8.4	18.1	11.7	18.3

Table 2: Average reduction (%) of gates compared to the preprocessed IWLS instances, by configuration and initial size.

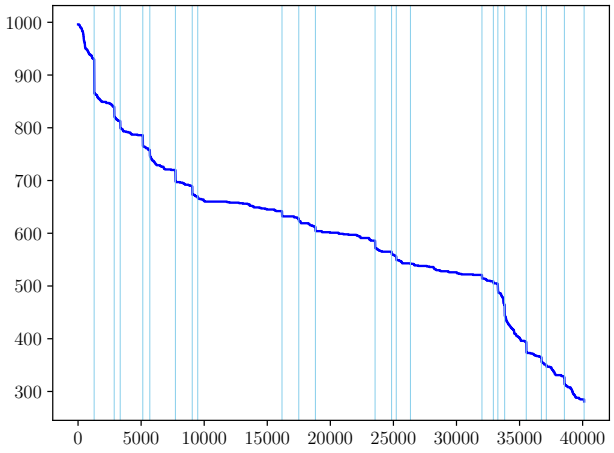


Figure 2: Number of gates in a 48-hour run of our tool applied to the preprocessed IWLS ex07 instance.

shown in Tbl. 2. The base configuration yields the best results for each class. If the replaced subcircuits need to be locally equivalent (*Equivalent*), the results are significantly worse. Also, the requirement to only replace subcircuits with smaller circuits (*Reduce*) has a significant negative impact on overall performance. While the limitation on the selection does not have such a significant impact as the previous restriction, we can still see a clear difference in the number of reduced gates to the base configuration. Finally, not using `ABC` for inprocessing (*No ABC*) leads to a deterioration of a similar magnitude as the previous limitation. All in all, this answers **RQ2** and shows the viability of the base configuration. Later we will illustrate the impact of using `ABC` in a bit more detail.

To further illustrate the behavior of our tool, we selected an instance from the IWLS benchmark set (ex07) and analyzed it in more detail. For this purpose, we applied our tool for two hours with the base configuration. Then we applied `ABC` in the same configuration as before. This combination was then applied 24 times.

The results of these runs are depicted in Fig. 2. The x-axis of the diagram shows the number of iterations. The y-axis shows the number of gates at a given iteration. The vertical lines indicate the applications of `ABC`.

We can see that the curve flattens with an increasing number of iterations. This is expected, as with an increasing number of iterations, the circuits get smaller, and thus

Instance	#Inputs	2016	2017	2018	2019	2020	2021	2022	Ours – 24h	Ours – 48h
Int to float converter	11	34	28	26	26	24	24	24	20	19
Alu control unit	7	29	29	27	27	27	27	27	27	27
Lookahead XY router	60	53	52	50	50	50	50	50	49	49
Coding-cavlc	10	107	101	68	68	68	68	68	59	58
Priority encoder	128	118	110	102	102	102	100	100	98	97
Adder	256	192	192	192	192	192	191	185	183	182
I2c controller	147	230	227	200	200	200	200	200	192	189
Decoder	8	272	270	264	264	264	264	264	264	264
Round-robin arbiter	256	429	409	328	328	313	306	304	297	295
Barrel shifter	135	512	512	512	512	512	512	512	512	512
Max	512	532	523	522	522	522	522	522	522	522
Sine	24	1347	1229	1227	1227	1221	1205	1205	1205	1205
Voter	1001	1515	1301	1297	1297	1293	1281	1279	1279	1279
Memory controller	1204	2399	2354	2041	2041	2041	2019	2019	1995	1975

Table 3: Comparison of the number of gates of the best known implementation for EPFL circuits of the last six years with results obtained from our iterative replacement approach.

it gets harder to reduce them further. Similarly, the reductions obtained by `ABC` tend to decrease. Moreover, we can see that even if the iterative replacement could achieve little improvements, `ABC` could still further improve the circuits. This is most likely due to the replacement of subcircuits, even if they are not smaller. We can also see that the distance between the vertical lines varies substantially. This is mainly due to the usage of different upper bounds on the size of the selected subcircuits. The larger the subcircuits can be, the harder the QBF calls get. This means that each individual iteration takes longer. While a larger bound on the size means that fewer subcircuits can be checked, it often allows reducing the circuit, which was impossible with smaller sizes.

EPFL-instances To analyze the performance of the iterative replacement of subcircuits with non-binary gates, we considered the *EPFL Combinational Benchmark Suite* (Amarú, Gaillardon, and Micheli 2015). This benchmark set consists of twenty circuits. The goal is to represent the specification with a circuit with six-input gates where either the number of gates or the depth of the circuit is as small as possible. We only consider the task of finding representations with few gates. In the benchmark suite also the best known realizations are given. We use the best known realizations as initial specifications for our tool.

In our evaluation, we run our tool for 24 hours. Afterwards we apply our tool to the resulting circuits and let it run for another 24 hours.

Tbl. 3 gives the best known results from the last years and the results of our approach. Note that the table does not list all the benchmarks from the benchmark suite. The six benchmarks with more gates than the *memory controller* instance are too large to be processed by our tool. For this reason, we do not mention them in the table. Also note that the best known results are continuously updated. We use the best known results as of June 30$^{\text{th}}$, 2022 (commit *141d000*).

The results show that the QBF-based approach could further reduce the size for several of the best known realizations. In particular, we want to highlight the results for the *Coding-cavlc* and the *I2c controller* instances. While the last time the two circuits could be improved was five years ago, our approach could still further reduce these circuits.

Moreover, the table shows that even after long runs our tool can achieve further reductions. Overall, Tbl. 3 gives a positive answer to **RQ3**.

Conclusions

We presented a natural QBF encoding for synthesizing an optimal implementation of small circuits. As this encoding captures degrees of freedom not covered by existing methods, we could generalize it to locally improve larger circuits by reducing the size of subcircuits. While this approach is no longer guaranteed to yield optimal implementations, our experiments show that the main features of our approach provide a significant improvement.

While the presented approach is expensive in terms of running time, the running time is not necessarily of major importance. This is illustrated by the IWLS 2022 programming contest and the EPFL 6-LUT circuit collection—in both cases, only the number of gates in the results and not the time needed to compute them matters. Still, there is potential to improve the runtime of the algorithm. First, the algorithm is *anytime* and can be stopped when the desired reduction in size has been achieved or the time budget is exhausted. Second, there is potential for parallelization by running multiple exact synthesis jobs simultaneously. Third, adapting QBF solvers (e.g., better support for incremental solving) could reduce the time spent by the solver. Finally, local resynthesis can be detached from the larger circuit by first computing a Boolean relation capturing the implementation flexibility of multi-output subcircuits (Savoj 1992).

We hope that these tools developed within the larger AI community will lead to further applications and ultimately result in a virtuous cycle as in SAT solving.

Acknowledgements

Supported by the Vienna Science and Technology Fund (WWTF) under the grants [10.47379/ICT19060] and [10.47379/ICT19065], and the Austrian Science Fund (FWF) under the grants W1255 and P32441.

References

Amarú, L.; Gaillardon, P.-E.; and Micheli, G. D. 2015. The EPFL Combinational Benchmark Suite. In *International Workshop on Logic & Synthesis (IWLS)*.

Beyersdorff, O.; Janota, M.; Lonsing, F.; and Seidl, M. 2021. Quantified Boolean Formulas. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, 1177–1221. IOS Press.

Brayton, R. K.; Hachtel, G. D.; and Sangiovanni-Vincentelli, A. L. 1990. Multilevel Logic Synthesis. *Proc. IEEE*, 78(2): 264–300.

Brayton, R. K.; and Mishchenko, A. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *CAV*, volume 6174 of *LNCS*, 24–40. Springer.

Chen, H.; Janota, M.; and Marques-Silva, J. 2012. QBF-Based Boolean Function Bi-Decomposition. In *DATE*, 816–819. IEEE.

De Micheli, G. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw Hill.

Fichte, J. K.; Lodha, N.; and Szeider, S. 2017. SAT-Based Local Improvement for Finding Tree Decompositions of Small Width. In *SAT*, volume 10491 of *LNCS*, 401–411. Springer.

Fujita, M. 2015. Toward Unification of Synthesis and Verification in Topologically Constrained Logic Design. *Proc. IEEE*, 103(11): 2052–2060.

Fujita, M.; Jo, S.; Ono, S.; and Matsumoto, T. 2013. Partial synthesis through sampling with and without specification. In *ICCAD*, 787–794. IEEE.

Fujita, M.; Kimura, Y.; Le, X.; Miyasaka, Y.; and Gharehbaghi, A. M. 2020. Synthesis and Optimization of Multiple Portions of Circuits for ECO based on Set-Covering and QBF Formulations. In *DATE*, 744–749. IEEE.

Haaswijk, W.; Soeken, M.; Mishchenko, A.; and Micheli, G. D. 2020. SAT-Based Exact Synthesis: Encodings, Topology Families, and Parallelism. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 39(4): 871–884.

Janota, M. 2018. Towards Generalization in QBF Solving via Machine Learning. In *AAAI*, 6607–6614. AAAI Press.

Knuth, D. E. 2011. *The Art of Computer Programming. Volume 4A, Combinatorial Algorithms, Part 1*. Addison Wesley, 1st edition. edition.

Kojevnikov, A.; Kulikov, A. S.; and Yaroslavtsev, G. 2009. Finding Efficient Circuits Using SAT-Solvers. In *SAT*, volume 5584 of *LNCS*, 32–44. Springer.

Kulikov, A. S.; Pechenev, D.; and Slezkin, N. 2022. SAT-Based Circuit Local Improvement. In *MFCS*, volume 241 of *LIPIcs*, 67:1–67:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Lodha, N.; Ordyniak, S.; and Szeider, S. 2019. A SAT Approach to Branchwidth. *ACM Trans. Comput. Log.*, 20(3): 15:1–15:24.

Mishchenko, A.; and Brayton, R. K. 2005. SAT-Based Complete Don't-Care Computation for Network Optimization. In *DATE*, 412–417. IEEE Computer Society.

Mishchenko, A.; Brayton, R. K.; Jiang, J. R.; and Jang, S. 2011. Scalable Don't-Care-Based Logic Optimization and Resynthesis. *ACM Trans. Reconfigurable Technol. Syst.*, 4(4): 34:1–34:23.

Mishchenko, A.; Chatterjee, S.; and Brayton, R. K. 2006. DAG-Aware AIG Rewriting a Fresh Look at Combinational Logic Synthesis. In *DAC*, 532–535. ACM.

Murray, C. D.; and Williams, R. R. 2017. On the (Non) NP-Hardness of Computing Circuit Complexity. *Theory Comput.*, 13(1): 1–22.

Ramaswamy, V. P.; and Szeider, S. 2021. Turbocharging Treewidth-Bounded Bayesian Network Structure Learning. In *AAAI*, 3895–3903. AAAI Press.

Ramaswamy, V. P.; and Szeider, S. 2022. Learning Large Bayesian Networks with Expert Constraints. In *UAI*, PMLR, 180:1592–1601.

Riener, H.; Haaswijk, W.; Mishchenko, A.; Micheli, G. D.; and Soeken, M. 2019. On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis. In *DATE*, 1649–1654. IEEE.

Riener, H.; Lee, S.; Mishchenko, A.; and Micheli, G. D. 2022. Boolean Rewriting Strikes Back: Reconvergence-Driven Windowing Meets Resynthesis. In *ASP-DAC*, 395–402. IEEE.

Riener, H.; Mishchenko, A.; and Soeken, M. 2020. Exact DAG-Aware Rewriting. In *DATE*, 732–737. IEEE.

Savoj, H. 1992. *Don't Cares in Multi-Level Network Optimization*. Ph.D. thesis, University of California, Berkeley.

Savoj, H.; and Brayton, R. K. 1990. The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks. In *DAC*, 297–301. IEEE Computer Society Press.

Schidler, A.; and Szeider, S. 2021. SAT-based Decision Tree Learning for Large Data Sets. In *AAAI*, 3904–3912. AAAI Press.

Shukla, A.; Biere, A.; Pulina, L.; and Seidl, M. 2019. A Survey on Applications of Quantified Boolean Formulas. In *ICTAI*, 78–84. IEEE.

Sinz, C. 2005. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In van Beek, P., ed., *CP*, volume 3709 of *LNCS*, 827–831. Springer.

Soeken, M.; Riener, H.; Haaswijk, W.; Testa, E.; Schmitt, B.; Meuli, G.; Mozafari, F.; Lee, S.-Y.; Calvino, A. T.; Marakkalage, D. S.; and De Micheli, G. 2022. The EPFL Logic Synthesis Libraries. *CoRR*, abs/1805.05121v3.

Testa, E.; Amarù, L. G.; Soeken, M.; Mishchenko, A.; Vuillod, P.; Gaillardon, P.; and Micheli, G. D. 2020. Extending Boolean Methods for Scalable Logic Synthesis. *IEEE Access*, 8: 226828–226844.

Wille, R.; Le, H. M.; Dueck, G. W.; and Große, D. 2008. Quantified Synthesis of Reversible Logic. In *DATE*, 1015–1020. ACM.