# eSLIM: Circuit Minimization with SAT Based Local Improvement

**Franz-Xaver Reichl**
TU Wien, Austria

**Friedrich Slivovsky**
University of Liverpool, UK

**Stefan Szeider**
TU Wien, Austria

## Abstract

eSLIM is a tool for circuit minimization that utilizes **E**xact **S**ynthesis and the **S**AT-based **l**ocal **i**mprovement **m**ethod (**SLIM**) to locally improve circuits. eSLIM improves upon the earlier prototype CIOPS that uses Quantified Boolean Formulas (QBF) to succinctly encode resynthesis of multi-output subcircuits subject to don't cares. This paper describes two improvements. First, it presents a purely propositional encoding based on a Boolean relation characterizing the input-output behavior of the subcircuit under don't cares. This allows the use of a SAT solver for resynthesis, substantially reducing running times when applied to functions from the IWLS 2023 competition, where eSLIM placed second. Second, it proposes circuit partitioning techniques in which don't cares for a subcircuit are captured only with respect to an enclosing window, rather than the entire circuit. Circuit partitioning trades completeness for efficiency, and successfully enables the application of exact synthesis to some of the largest circuits in the EPFL suite, leading to improvements over the current best implementation for several instances.

**2012 ACM Subject Classification** Hardware → Circuit optimization; Theory of computation → Automated reasoning

**Keywords and phrases** QBF, Exact Synthesis, Circuit Minimization, SLIM

**Digital Object Identifier** 10.4230/LIPIcs.SAT.2024.23

**Supplementary Material** *Software*: `https://github.com/fxreichl/eSLIM` [29]
archived at `swh:1:dir:9d2862aedfe458cfb06d4d43da6c0f46ed578c91`

**Funding** This work was supported by the Vienna Science and Technology Fund (WWTF) under grant 10.47379/ICT19060, and the Austrian Science Fund (FWF) under grants 10.55776/W1255 and 10.55776/P36420.

**Acknowledgements** The authors thank Alan Mishchenko for suggesting to represent local synthesis tasks with Boolean relations.

## 1 Introduction

Modern integrated circuits have grown increasingly large and complex, making their design and optimization a significant challenge. Automation has become indispensable for the process of circuit design, including logic optimization and logic synthesis, which collectively lead to substantial reductions in the number of gates and circuit depth [6, 10].

Applying exact methods for computing provably minimum size circuits is computationally intractable. Recently, it was shown that for a (multi-output) Boolean function given as a truth table, the task of finding a minimum size circuit consisting of *and*, *or* and *not* gates is NP-complete [17]. This is reflected by the observation that in practice, we can generally not compute minimum size circuits with many more than 10 fanin-2 gates [9, 21]. To deal with larger circuits, one can first partition them into smaller subcircuits, and then minimize these using exact techniques [39].


In practice, this approach is typically restricted to single-output subcircuits [25, 31]. Although efficient, this does not fully exploit the implementation flexibility of multi-output subcircuits. Recently, we proposed a high-effort method for resynthesizing multi-output subcircuits based on Quantified Boolean Formulas (QBFs) [30]. Our prototype CIOPS has shown success in minimizing circuits from the IWLS'22 competition and the EPFL combinational benchmark suite. Specifically due to increasing prices of silicon wafers in recent years [15] such high-effort methods are gaining importance.

In this paper, we describe two improvements for this approach, which we implemented in the new tool eSLIM. First, we present a workflow purely based on SAT instead of QBF. The new SAT workflow relies on computing an input-output relation that yields permissible output assignments for each input assignment. These relations allow representing *don't cares* – input/output combinations of the subcircuit where the outputs can be modified without altering the function computed by the encompassing circuit. In contrast, the QBF encoding captures don't cares implicitly. Our rationale is that the ability to use a SAT solver instead of a QBF solver will more than make up for this extra step when the relation is reasonably small. Second, we incorporate windowing to handle very large circuits. For such circuits, the QBFs encoding the existence of replacements for subcircuits become too hard for QBF solvers, and computing the input-output relation needed for the new SAT encoding takes too much time. To address this, we adapt a strategy from prior work on computing don't cares of single-output subcircuits [25]. Instead of ensuring that a resynthesized subcircuit preserves the Boolean function computed by the full circuit, we only require that it preserves the function computed by a "window" containing the subcircuit to be replaced. In theory, this means don't cares of a subcircuit are no longer fully captured. In practice, with windows containing hundreds of gates, we expect that generally don't cares within the window still allow substantial improvements.

We performed an experimental evaluation of these improvements. The SAT-based workflow proved to be substantially faster for small circuits, showing a significant performance increase for instances from the IWLS'23 competition. For circuits from the EPFL combinational benchmark suite, the SAT and the QBF-based approach showed comparable performance. Using windowing, we were able to scale both approaches to the largest circuits in this set, which had previously been unmanageable.

## 1.1 Related Work

Methods that fully capture the properties of Boolean functions implemented by circuits (rather than considering them as polynomials, for instance) are deemed the most effective in logic synthesis [39]. However, these methods are also the most computationally expensive, and their application to large circuits is limited to resynthesizing small subcircuits. SAT-based exact synthesis [16, 21] and SAT-based resubstitution [26, 32], which aim to represent the function implemented by a specific gate as a function of a few existing gates in the circuit, are examples of this approach. Many of these methods are implemented in the industrial-strength tool ABC [7].

A SAT-based method for capturing don't cares close to ours has been previously considered for single-output subcircuits, including the use of windowing to improve scalability [25]. Boolean relations have been proposed as a means of representing don't cares of multi-output subcircuits [23]. The corresponding optimization workflow relies on a simulation-based under-approximation of don't cares in combination with a divide-and-conquer algorithm for resynthesis, whereas we use a SAT solver for both don't care computation and resynthesis.

Exact resynthesis of subcircuits has also been explored for finding optimal circuits for symmetric functions in circuit complexity research, but without incorporating don't cares [22]. This is an instance of the *SAT-based Local Improvement Method (SLIM)*, a general optimization framework that has been applied to various AI problems [11, 24, 27, 34, 35, 36].

In the realm of logic synthesis, QBFs have been employed for bi-decomposition [8], reversible quantum circuit synthesis [41], and lookup table (LUT) synthesis [12, 13, 14]. The latter two problems impose more constraints than the setting considered in this work, as they maintain a fixed circuit topology. In contrast, our synthesis tasks also involve determining a suitable topology.

## 2 Preliminaries

A *Boolean circuit* is a *directed acyclic graph*. We denote the set of source nodes of a circuit $\mathcal{C}$ as the set of *primary inputs* $\text{in}(\mathcal{C})$ and the non-source nodes as *gates*. The set of *primary outputs* $\text{out}(\mathcal{C})$ is a subset of the set of nodes in $\mathcal{C}$. If there is a directed edge from node $n$ to node $m$ then $n$ is an *input* of $m$. Each gate corresponds to a Boolean function on its inputs. This means that each Boolean circuit naturally induces a Boolean function. Let $x$ be a node in $\mathcal{C}$. Then the *transitive fanin cone* of $x$, $TFI(x)$ is the set of all nodes in $\mathcal{C}$ from which $x$ is reachable. Similarly, the *transitive fanout cone* of $x$ $TFO(x)$ is the set of all nodes in $\mathcal{C}$ which can be reached from $x$. A node $x$ *depends* on a node $y$ if $y \in TFI(x)$. Two circuits $\mathcal{C}$ and $\mathcal{D}$ are logically equivalent ($\mathcal{C} \equiv \mathcal{D}$) if they compute the same Boolean function. *Don't cares* of a subcircuit $\mathcal{S}$ of a circuit $\mathcal{C}$ are certain patterns of its inputs or outputs that do not have an effect on the function computed by the encompassing circuit $\mathcal{C}$. We distinguish between two types of don't cares. *Controllability don't cares* are patterns of the inputs of $\mathcal{S}$ which cannot be attained within $\mathcal{C}$, and *observability don't cares* are patterns of the inputs of $\mathcal{S}$ for which its outputs do not have an effect on $\mathcal{C}$ [10].

A *Quantified Boolean formula (QBF)* is of the form $\forall X_1 \exists X_2 \ldots \forall X_{k-1} \exists X_k . \varphi$, where the $X_i$ are pairwise disjoint sets of variables, and $\varphi$ is a propositional formula called the *matrix*. The quantifiers range over the Boolean domain $\mathcal{B} = \{0, 1\}$, so that existential ($\exists$) quantifiers can be understood as abbreviating a disjunction ($\exists x.\varphi \equiv \varphi[x \leftarrow 0] \vee \varphi[x \leftarrow 1]$), and universal ($\forall$) quantifiers as encoding a conjunction ($\forall x.\varphi \equiv \varphi[x \leftarrow 0] \wedge \varphi[x \leftarrow 1]$). Evaluating QBFs is a PSPACE-complete task, and QBFs can succinctly encode problems arising in many areas [37]. For an overview of QBF, including solving techniques and proof complexity, see [2].

## 3 Exact Synthesis of Subcircuits

Our approach involves replacing a subcircuit with a smaller one, ensuring that the function computed by the encompassing circuit remains unchanged. For a fixed value $\ell$, we use a SAT or QBF solver to determine whether there is a replacement circuit of size $\ell$. To find a smallest possible circuit, the value $\ell$ is decremented until the encoding becomes unsatisfiable.

Throughout this section, let $\mathcal{C}$ denote the encompassing circuit, $\mathcal{S}$ one of its subcircuits, and $\mathcal{T}$ the replacement circuit. Further, $n$ is the number of inputs of the subcircuit $\mathcal{S}$ and $m$ the number of outputs. Moreover, we assume that $\mathcal{C}$ is *$k$-regular*, i.e., each gate in $\mathcal{C}$ has $k$ inputs. Let $\mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}]$ denote the result of substituting the circuit $\mathcal{T}$ for the subcircuit $\mathcal{S}$ in $\mathcal{C}$. Our goal is to find a circuit $\mathcal{T}$ of size $\ell$ with $n$ inputs and $m$ outputs such that $\mathcal{C}$ and $\mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}]$ are logically equivalent. This means that $\mathcal{T}$ and $\mathcal{S}$ do not necessarily need to be equivalent. In particular this also means that the solver can assign an arbitrary behavior to $\mathcal{T}$ for input/output combinations of the subcircuit that do not have an influence on the entire circuits – i.e., we can make use of don't cares implicitly.

We initially set $\ell = |\mathcal{S}|$ and then decrement $\ell$ until the solver determines that no such circuit $\mathcal{T}$ exists, at which point we can conclude that the circuit $\mathcal{T}$ must have at least $\ell + 1$ gates. A circuit $\mathcal{T}$ of this size can be constructed from a model of the last satisfiable encoding, and used to replace $\mathcal{S}$. It may seem unnecessary to initially ask the solver to come up with a circuit of size $\ell = |\mathcal{S}|$, since we already know such a circuit exists. However, the new circuit is typically not equivalent to $\mathcal{S}$, and replacing $\mathcal{S}$ by $\mathcal{T}$ is often beneficial in the overall minimization process even though it does not immediately decrease size [30].

Subsequently, we will describe both the QBF and the SAT encoding. Both are closely related to and inspired by the *multi selection variable* SAT encoding for exact synthesis [16].

## 3.1 The QBF Encoding

The main advantage of a QBF encoding is that one can universally quantify over assignments of primary inputs and encode the value computed by each gate using a single variable, rather than introducing a variable for each line in the truth table representation of $\mathcal{C}$. In addition to these **gate variables** and primary inputs, the encoding contains the following groups of existentially quantified variables that determine the structure of the circuit:

**Selection variables** $S_i = \{s_{it} \mid 1 \leq t < i + n\}$. This set of variables determines the inputs of the $i^{\text{th}}$ gate. If $s_{it}$ is true then the $t^{\text{th}}$ node is an input of the $i^{\text{th}}$ gate[1].

**Function variables** $F_i = \{f^i_{a_1 \ldots a_k} \mid (a_1, \ldots, b_k) \in \mathcal{B}^k\}$. This set of variables describes the Boolean function $f$ at the $i^{\text{th}}$ gate, i.e., the assignment of $f^i_{a_1 \ldots a_k}$ determines the value of $f(a_1, \ldots, a_k)$.[2]

**Output variables** $O_j = \{o_{tj} \mid 0 \leq t \leq n + \ell\}$. This set of variables fixes the output gates of the circuit. If $o_{tj}$ is true then the $j^{\text{th}}$ output is given by the constant value false if $t = 0$ and otherwise by the $t^{\text{th}}$ node.[2]

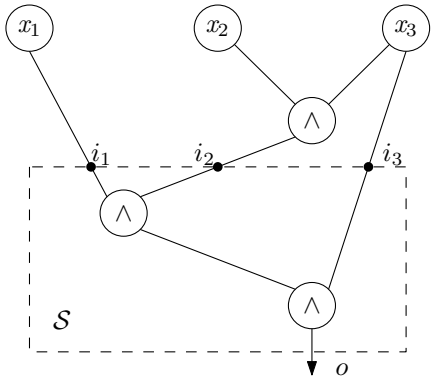
The matrix of the QBF encodes the following constraints:

- Each gate must have exactly $k$ inputs, i.e., at each gate $i$ exactly $k$ selection variables must be true. This can be enforced by using a sequential counter [38]. We denote this constraint by $\mathit{Count}(S_i, k)$.
- Each output must correspond to a single gate, i.e., for each output $j$ exactly one output variable is true. We denote this constraint by $\mathit{Count}(O_j, 1)$.
- For each gate $i$ the assignment of the gate variable must be compatible with the function determined by the function variables. Assume that the assignment for the function variables describes the function $F$ and that the values of the inputs of $i$ are given by $i_1, \ldots, i_k$. This constraint ensures that the gate variable for $i$ is assigned to $F(i_1, \ldots, i_k)$. We denote this constraint by $\mathit{Comp}_i$.
- Replacing $\mathcal{S}$ by $\mathcal{T}$ must preserve the function computed by $\mathcal{C}$. To express this, one can use Tseitin transformation and two sets of gate variables to encode both the specification $\mathcal{C}$ and the circuit $\mathcal{C}' = \mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}]$. For an output gate $o$, let $v_o$ and $v'_o$ denote the gate variable in the encoding of $\mathcal{C}$ and $\mathcal{C}'$, respectively. We add a constraint $v_o \Leftrightarrow v'_o$ and denote their conjunction over all outputs by $\mathit{Corr}$.

---

[1] We index gates from 1 to $\ell$, and we index nodes from 1 to $n + \ell$. The nodes with indices from 1 to $n$ correspond to the primary inputs. For $n < i \leq n + \ell$ the $i^{\text{th}}$ node corresponds to the $(i - n)^{\text{th}}$ gate.

[2] A Boolean function is *normal* if $f(0, \ldots, 0) = 0$, and a circuit is *normal* if each of its gates corresponds to a normal function. In a normal circuit, we can find minimum size normal replacement circuits [20]. We can always normalize a given circuit, reduce the size of the normalized circuit, and finally we can reconstruct the original circuit by flipping output gates if necessary. Thus, $f^i_{0 \ldots 0}$ can be set to false, and outputs yielding the constant value true don't need to be considered.




| $i_1$ | $i_2$ | $i_3$ | $o$ |
|---|---|---|---|
| 0 | 0 | 0 | $\{0\}$ |
| 0 | 0 | 1 | $\{0\}$ |
| 0 | 1 | 0 | $\{0,1\}$ |
| 0 | 1 | 1 | $\{0\}$ |
| 1 | 0 | 0 | $\{0\}$ |
| 1 | 0 | 1 | $\{0\}$ |
| 1 | 1 | 0 | $\{0,1\}$ |
| 1 | 1 | 1 | $\{1\}$ |

**Figure 1** The table on the right gives a Boolean relation for the subcircuit $\mathcal{S}$. One can see that the input pattern $i_2 = 1, i_3 = 0$ is not possible within the entire circuit. The relation given in the table makes use of this controllability don't care by assigning $\{0,1\}$ to the corresponding assignments of the inputs.

Let $S = \bigcup_{1 \leq i \leq k} S_i$ denote the set of selection variables, $F$ the set of gate definition variables, $O$ the set of output variables, $I$ the set of input variables, and $G, G'$ the two sets of gate variables. The complete QBF encoding has the following form:

$$\exists S, F, O\, \forall I\, \exists G, G'.\, \mathit{Corr} \wedge \bigwedge_{1 \leq j \leq m} \mathit{Count}(O_j, 1) \wedge \bigwedge_{1 \leq i \leq \ell} (\mathit{Count}(S_i, k) \wedge \mathit{Comp}_i)$$

In practice, we use the circuit-based QCIR format for the encoding. Thus, we can directly define auxiliary variables by QCIR-gates and so we do not need to add these variables to the prefix.

## 3.2 The SAT Encoding

While the QBF encoding handles don't cares implicitly, the SAT-based approach outlined in this subsection separates the tasks of computing don't cares and synthesizing a subcircuit. More specifically, it first computes a *Boolean relation* [6, 33] representing the input-output behavior of the subcircuit $\mathcal{S}$ on the *care set* (the complement of don't cares), and then uses the *multi selection variable* SAT-encoding [16] to obtain a circuit.

A Boolean relation $R$ for $\mathcal{S}$ maps each input assignment to a set of permissible output assignments, formally $R : \mathcal{B}^{\mathrm{in}(\mathcal{S})} \to (\mathcal{P}(\mathcal{B}^{\mathrm{out}(\mathcal{S})}) \setminus \emptyset)$. A circuit $C$ *implements* the relation $R$ if $C(\sigma) \in R(\sigma)$ for each $\sigma \in \mathcal{B}^{\mathrm{in}(C)}$. In Figure 1 we illustrate a Boolean relation representing a subcircuit $\mathcal{S}$ of a circuit $\mathcal{C}$ that takes the don't cares of $\mathcal{S}$ into account.

The core idea for computing the relation is to determine the set $\Phi \subset \mathcal{B}^{\mathrm{in}(\mathcal{S})} \times \mathcal{B}^{\mathrm{out}(\mathcal{S})}$ of conflicting input/output behaviors. A conflicting input/output behavior is a pair of assignments $(\sigma, \rho)$, where $\sigma$ is an assignment of $\mathcal{B}^{\mathrm{in}(\mathcal{S})}$ and $\rho$ is an assignment of $\mathcal{B}^{\mathrm{out}(\mathcal{S})}$. For each such pair we require that for any circuit $\mathcal{D}$ with inputs $\mathrm{in}(\mathcal{S})$, outputs $\mathrm{out}(\mathcal{S})$ and $\mathcal{D}(\sigma) = \rho$, we have $\mathcal{C}[\mathcal{S} \leftarrow \mathcal{D}] \not\equiv \mathcal{C}$. This means $\Phi$ describes all modifications of $\mathcal{S}$ that would change the function computed by $\mathcal{C}$. The relation $R$ is then given by $R(\sigma) = \{\rho \in \mathcal{B}^{\mathrm{out}(\mathcal{S})} \mid (\sigma, \rho) \notin \Phi\}$. One can easily verify that replacing $\mathcal{S}$ by any circuit implementing this relation preserves the function computed by the entire circuit.

In order to obtain $\Phi$, we first compute a circuit $\mathcal{C}'$ by removing $\mathcal{S}$ from $\mathcal{C}$.[3] Due to the removal of gates, $\mathcal{C}'$ contains an additional primary input for each output of $\mathcal{S}$. We denote the set of these new inputs by $\mathcal{I}$. Next, we compute every assignment $\sigma$ for $\text{in}(\mathcal{C})$ and $\rho$ for $\mathcal{I}$ such that $\mathcal{C}(\sigma) \neq \mathcal{C}'(\sigma \cup \rho)$. Now let $\sigma'$ be the assignment of the inputs of $\mathcal{S}$ ($\text{in}(\mathcal{S})$), which is attained by $\mathcal{C}$ under $\sigma$. Then $\Phi$ is the set of all pairs $(\sigma', \rho)$.

To realize this idea we first compute clausal encodings $\varphi_1$ for $\mathcal{C}$ and $\varphi_2$ for $\mathcal{C}'$ by Tseitin transformation, introducing a propositional variable for each node. For each node $x$ in $\mathcal{C}$ we denote the corresponding variable by $v(x)$ and for each node $x$ in $\mathcal{C}'$ by $v'(x)$. Similarly, we define $v \ / \ v'$ for sets of nodes. Next we introduce for each common primary output $o$ the constraint $v(o) \Leftrightarrow v'(o)$, denoting the set consisting of all these clauses by *equiv*.

This encoding can now be used to compute the relation with incremental SAT solving. The algorithm maintains a set $B$ of blocking clauses, which is empty initially. We now ask the SAT solver for an assignment $\sigma$ that satisfies $\varphi_1 \wedge \varphi_2 \wedge B$ and falsifies *equiv*. Instead of directly adding $\neg\sigma|_{v(\text{in}(\mathcal{S}))} \vee \neg\sigma|_{v'(\mathcal{I})}$ to the clausal representation of the relation, we first try to reduce this clause. For this purpose, we apply a similar approach as it was used by Ravi and Somenzi [28] for reducing the size of assignments. The formula $\varphi_1 \wedge \varphi_2 \wedge \textit{equiv}$ is unsatisfiable under the assumption $\sigma|_{v(\text{in}(\mathcal{C}))} \wedge \sigma|_{v'(\mathcal{I})}$. Using the SAT solver, we can compute a subset $\hat{\sigma}$ of failed assumptions. As $\hat{\sigma}$ suffices to make $\varphi_1 \wedge \varphi_2 \wedge \textit{equiv}$ unsatisfiable it is rather easy to conclude that for each assignment $\mu$ with $\hat{\sigma}|_{v'(\mathcal{I})} \subseteq \mu$, the pair $(\sigma|_{v(\text{in}(\mathcal{S}))}, \mu)$ must not be contained in the relation. Thus, we can add $\neg\sigma|_{v(\text{in}(\mathcal{S}))} \vee \neg\hat{\sigma}|_{v'(\mathcal{I})}$ to the clausal representation of the relation. Moreover, to avoid the same inconsistency in subsequent iterations, we add the blocking clause $\neg\sigma|_{v(\text{in}(\mathcal{S}))} \vee \neg\hat{\sigma}|_{v'(\mathcal{I})}$ to $B$.

As mentioned above, a circuit implementing the relation $R$ can be synthesized using a slight adaptation of the SAT encoding for exact synthesis by Haaswijk et al. [16].

## 4 Minimization by Subcircuit Resynthesis

We use exact synthesis of subcircuits as a subroutine in a circuit minimization algorithm that repeatedly selects subcircuits for resynthesis. To obtain a subcircuit for resynthesis, we start from a *root gate*, and then expand by incorporating successors of previously chosen gates until reaching a predetermined size. Root gates are chosen randomly from the circuit. To expand the root gate, we visit gates that use previously selected gates as inputs in a breadth-first-search manner. We then randomly decide whether to include this gate in the subcircuit. Unlike previous work [30], we use a fixed bound for the size of subcircuits. This bound is decreased in case individual checks timeout.

## 5 Window Selection

Both the SAT and the QBF-based rewriting approach do not only depend on the selected subcircuits but also on the entire circuit. This is necessary to make use of don't cares. Thus, in general, rewriting subcircuits gets harder for larger circuits. To overcome this issue, we only consider don't cares with respect to a *window* (a subcircuit) [25]. The window is chosen such that the size is still manageable for our QBF/SAT-based approach. In this manner, we can minimize windows, and since the functions computed by windows are preserved, the optimized implementation can be used to replace the original window.

[3] Actually, it suffices to consider $TFO(\mathcal{S})$ instead of a copy of the entire circuit

Additionally, using windows allows us to rewrite subcircuits simultaneously. For this purpose, we compute pairwise disjoint windows and rewrite them separately. In the end, we then combine the optimized windows to obtain a new implementation for the original circuit.

## 6 Implementation

We implemented eSLIM mainly in Python and some parts in C++. For using C++ subroutines in Python, we used PYBIND11[4]. As backend solvers, we used QFUN [18] for QBF and CaDiCaL [4] for SAT. To read and write files in the *AIGER* format, we used the AIGER library [5]. To represent arrays of Boolean values, we used the python library BITARRAY[5]. eSLIM is freely available[6] under an MIT license.

### Input/Output Formats

eSLIM can process circuits given in the Berkeley Logic Interchange Format (BLIF) [40] and in the AIGER format [3]. The given circuits have to be purely combinational and thus must not contain any sequential components. Circuits in the BLIF format are required to be sorted topologically, i.e., fanins of a gate must be specified before the gate itself. For the AIGER format, both the ASCII and the binary AIGER format are supported. Similar as for circuits given in the BLIF format, also ASCII AIGER circuits are expected to be sorted topologically. The main output format of eSLIM is BLIF. In case an AIG shall be reduced, also the binary and ASCII AIGER format are supported.

### Parameters of eSLIM

Our tool can be applied as follows.

```
eSLIM <circuit> <result> <budget> [options]
```

We first describe the mandatory arguments:

**circuit** The circuit to be processed in one of the supported formats.

**result** The destination for the minimized circuit (given in the BLIF format).

**budget** The available time in seconds.

Additionally, eSLIM accepts several optional arguments. We only list the most important ones, and we refer to the help information provided by eSLIM for the remaining options.

**--gs** The number of fanins of the synthesized gates.

**--size** The upper bound for considered subcircuit sizes.

**--aig** Synthesize an AIG. This option requires that the fanin size is set to two.

**--aig-out** Write the minimized circuit to a binary or ASCII AIGER file.

**--restarts** Specify the number of applications of our method. Each application can use `budget` seconds.

**--abc** Use ABC for inprocessing after each application of our method.

**--syn-mode** Specify whether the QBF- or the SAT-based approach shall be used.

**--windows** Specify the number of windows that shall be processed concurrently and the reference size for each considered windows.

[4] https://github.com/pybind/pybind11
[5] https://pypi.org/project/bitarray/
[6] https://github.com/fxreichl/eSLIM

### Types of Logic Gates

In its default configuration, eSLIM computes replacement circuits composed of arbitrary $k$-fanin gates – where $k$ is specified by `--gs`. The set of available functions for each gate can be constrained by restricting the assignments for the function variables. This, for example, means that if the option `--aig` is used, then XOR gates are ruled out by requiring $\neg f^i_{01} \vee \neg f^i_{10} \vee f^i_{11}$ for each $1 \leq i \leq \ell$, where $\ell$ denotes the considered circuit size. In addition to computing AIGs, we also used eSLIM to compute XAIGs – AIGs that may contain XOR gates. By adding appropriate constraints for the function variables, additional restrictions can be placed on the set of available functions.

## 7 Experiments

We evaluated our tool on the instances from the IWLS'23 programming contest[7] and the instances from the EPFL benchmark suite [1]. All experiments were conducted on a cluster with AMD EPYC 7402 processors at 2.8 GHz running 64-bit Linux. We used a memory limit of 4 GB. For the parallelized minimization we used a memory limit of 4 GB per thread.

### 7.1 IWLS Instances

The IWLS'23 instances consist of 100 instances, given as truth tables. The goal is to compute an *And-Inverter Graph* (AIG) with as few gates as possible. A preliminary version of eSLIM participated in the competition and took the second place.

Since the instances are given as truth tables, and our tool requires that specifications are given as circuits, we had to preprocess the instances using ABC [7]. As a naive transformation of truth tables to circuits by using ABC results in relatively large circuits, we used ABC to reduce the size of the initial circuit. For this purpose, we used the ABC command DEEPSYN with a timeout of one hour[8]. We chose DEEPSYN as it is one of the most effective optimization strategies for computing compact AIGs [9]. Additionally, unlike many other optimization strategies in ABC, DEEPSYN allows any-time optimization.

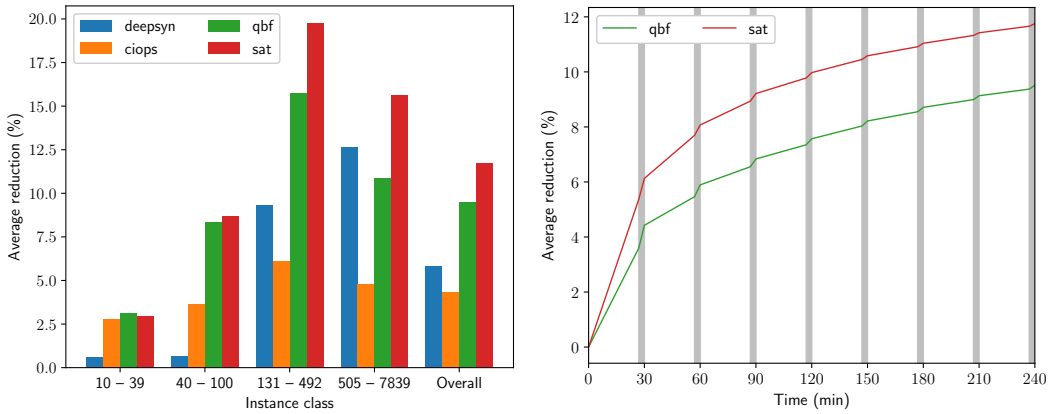
In our evaluation setup, we considered our tool both in the QBF-based and the SAT-based configuration. As only very few of the IWLS instances are sufficiently large for a reasonable application of windowing, we did not evaluate it here. We compared eSLIM with the prototype of the purely QBF-based approach CIOPS [30] and the DEEPSYN procedure[9].

In our experiments, we alternated between 27-minute runs of eSLIM and 3-minute runs of DEEPSYN for inprocessing. Similarly, we alternated between 30-minute runs of CIOPS and exhaustive heuristic minimization with ABC. Here *exhaustive minimization* means that the application of ABC is repeated until no further reductions can be found. We wanted to compare eSLIM with the original version of CIOPS, thus we used different setups for inprocessing in eSLIM and CIOPS. These combinations were applied eight times. We used 30-minute runs of CIOPS as the time needed for inprocessing cannot be controlled. As inprocessing usually only takes a few seconds, still in total roughly 4 hours were available per instance. Finally, DEEPSYN was run for four hours. In general, all tools benefited from

[7] `https://github.com/alanminko/iwls2023-ls-contest`
[8] To run DEEPSYN we used `&deepsyn -T 3600`. By using `-T 3600` we enforced the timeout.
[9] We could not compare our tool against the first and third ranked entry to the IWLS 2023 competition – our tool ranked second – as they were not available to us.

**Table 1** Average reduction (%) of gates compared to the preprocessed IWLS'23 instances, by configuration and initial size and standard deviations of the average reductions per configuration. The best results are marked in boldface.

| | CIOPS | | DEEPSYN | | QBF | | SAT | |
|---|---|---|---|---|---|---|---|---|
| #Gates | mean | stdev | mean | stdev | mean | stdev | mean | stdev |
| 10-39 | 2.8 | 0.28 | 0.59 | 0.4 | **3.11** | 0.19 | 2.93 | 0.28 |
| 40-100 | 3.64 | 0.19 | 0.68 | 0.18 | 8.34 | 0.43 | **8.7** | 0.26 |
| 131-492 | 6.12 | 0.22 | 9.32 | 0.44 | 15.75 | 0.57 | **19.76** | 0.59 |
| 505-7839 | 4.79 | 0.13 | 12.64 | 0.47 | 10.85 | 0.39 | **15.61** | 0.52 |
| Overall | 4.34 | 0.11 | 5.82 | 0.22 | 9.51 | 0.25 | **11.75** | 0.17 |




**(a)** Average reductions.

**(b)** Reductions over time.

**Figure 2** The left figure (a) visualizes the average reductions of gates per instance class and configuration. The right figure (b) visualizes the average reductions among all instances for the SAT and the QBF configuration over time. The areas marked in gray correspond to the application of ABC.

longer runtimes. Nevertheless, we limited the runs to roughly four hours due to constraints on the available computational infrastructure. Additionally, we set the initial bound for the subcircuit size to 6 both for our tool and CIOPS (cf. Section 4).

Instances were grouped into four subsets of 25 based on the initial number of gates. For each configuration and instance group, we determined the average size reduction (in %) for circuits in that group. We performed 5 independent runs for each configuration. First, we calculated the average reduction per run for each subset and for the entire set of instances. Based on these values, we computed average reductions and standard deviations among the individual runs of each configuration for each class of instances. Results are given in Table 1. A visualization of the results is given in Figure 2a.

The experiments show that eSLIM with the QBF configuration clearly outperformed CIOPS. This was mainly due to two simple but apparently effective changes. First, eSLIM uses fixed bounds for the sizes of selected subcircuits, while CIOPS tries to increase the initially given bounds as far as possible. Since larger circuits are usually harder to analyze, this indicates that, in general, it is advantageous to consider more but simpler (smaller) circuits. Second, eSLIM computes subcircuits by expanding root gates in a randomized breadth-first-search manner, while CIOPS applies an expansion strategy that aims at keeping the number of outputs low.

Moreover, the SAT-based strategy outperformed the QBF-based strategy. This was possible as in general the SAT-based approach allowed a faster analysis of subcircuits. Thus, more subcircuits could be analyzed in total. Furthermore, eSLIM could outperform DEEPSYN. This indicates that the combination of DEEPSYN for preprocessing/inprocessing with eSLIM is viable alternative to just applying DEEPSYN alone.

Furthermore, to illustrate the achieved reductions over time we give Figure 2b. The figure shows the average reduction among all instances for the QBF- and the SAT-based configuration over time. We can see that the achieved improvements diminish the longer the tool ran, still on average both approaches could find new improvements until the end. For the QBF- and the SAT-based configuration the DEEPSYN inprocessing steps were responsible for 26%, respectively 18% of the total reductions.

Additionally, if we consider the best implementations for each instance among the 5 SAT runs, we can observe an average reduction of about 14%. This indicates that it may be advantageous to consider multiple runs of eSLIM. As our tool makes use of a randomized subcircuit selection, different runs result in different sequences of replaced subcircuits. Thus, a run might get stuck in a local minimum, which is difficult to escape.

## 7.2 EPFL Instances

To evaluate our tool for circuits with non-binary gates, we considered the *EPFL Combinational Benchmark Suite* [1]. This benchmark set consists of twenty circuits.[10] The goal is to find 6-input lookup table (LUT-6) implementations of the specifications with small size. In addition to a specification given as a circuit with binary gates, the benchmark suite also provides the best known LUT-6 realizations so far. We used the best known realizations as of 2022 (commit *42c1f31*) as initial specifications for our tool.[11]

We ran our reduction tool for 12 hours both with the SAT and the QBF configuration. After each hour we applied the ABC command &MFS as an inprocessing step – we used &MFS as it allows us to directly optimize an LUT-6 circuit. Additionally, we also applied our tool with windowing enabled. Here, we recombined the windows for the inprocessing step and computed new windows afterwards. We compared our tool with CIOPS. We want to point out that the initial realizations have already been highly optimized by different methods, so any improvement can be considered a success.

In addition to a bound on the size of subcircuits, we also used a limit of 10 on the number of inputs of the subcircuits considered for resynthesis. Preliminary tests showed that such a limit is required to reliably generate the Boolean relation for the SAT-based approach within time and memory limits. Additionally, we always set the initial bound for the subcircuit size to 4. In the experiments with windowing, we used two different window sizes, 500 and 1000. First we minimized single windows and second up to 8 windows concurrently. We only applied windowing for instances with at least 1000 gates.

Results for instances with less than 1000 gates are given in Table 2a and results for instances with at least 1000 gates are given in Table 2b. For the results with windowing we selected the best results among the two different window sizes.

Since the initial circuits are already highly optimized by state-of-the-art methods, the relative improvements for the EPFL instances were small compared to the IWLS instances, and it is difficult to draw any definitive conclusion about the superiority of any configuration

[10] We did not consider the MtM instances as the EPFL repository does not contain the best implementations for these circuits.

[11] We did not consider the best results of 2023 as half of them were provided by us. As it is difficult for our tool to further reduce these circuits, we think the circuits from 2022 are better suited for the evaluation.

**Table 2** The table gives for each instance the number of LUT-6 gates of the initial circuit and of the improved circuits per configuration. The best results are marked in boldface.

**(a)** Results for EPFL instances with less than 1000 gates.

| Instance | Initial | CIOPS | QBF | SAT |
|---|---|---|---|---|
| Lookahead XY router | **19** | 19 | 19 | 19 |
| int to float converter | 20 | 20 | **18** | 19 |
| Alu control unit | **25** | 25 | 25 | 25 |
| Coding-cavlc | 54 | 52 | **49** | 53 |
| Priority encoder | 94 | 94 | 93 | **92** |
| Adder | **129** | 129 | 129 | 129 |
| I2c controller | 182 | 178 | 179 | **177** |
| Decoder | **264** | 264 | 264 | 264 |
| Round-robin arbiter | 273 | 273 | 272 | **267** |
| Max | **511** | 511 | 511 | 511 |
| Barrel shifter | **512** | 512 | 512 | 512 |

**(b)** Results for EPFL instances with fewer than 1000 gates.

| | | | No Windowing | | Single Window | | Up to 8 Windows | |
|---|---|---|---|---|---|---|---|---|
| Instance | Initial | CIOPS | QBF | SAT | QBF | SAT | QBF | SAT |
| Sine | 1114 | 1111 | 1095 | 1085 | 1076 | 1069 | 1057 | **1036** |
| Voter | 1217 | 1217 | 1217 | 1179 | 1184 | **1166** | 1177 | 1172 |
| Memory controller | 1735 | 1731 | **1722** | 1727 | 1731 | 1730 | 1724 | 1731 |
| Square-root | 2994 | 2994 | 2994 | 2994 | 2991 | 2985 | 2992 | **2980** |
| Square | 3018 | 3018 | 3014 | 2997 | 2992 | 2994 | **2942** | 2943 |
| Divisor | 3096 | 3096 | 3096 | 3096 | 3096 | **3095** | 3096 | **3095** |
| Multiplier | 4360 | 4360 | 4358 | 4346 | 4346 | 4346 | 4326 | **4317** |
| Log2 | 6133 | 6133 | 6132 | 6109 | 6127 | 6129 | 6078 | **6063** |
| Hypotenuse | 39452 | 39452 | 39452 | 39452 | 39230 | 39251 | **38459** | 38781 |

from these results. Nevertheless, the results suggest that parallel optimization was able to beat single-threaded optimization. Similarly, the results indicate that both configurations of our tool outperformed CIOPS. Moreover, our tool could improve on the best implementation for the majority of instances.

## 8 Conclusion

The experimental analysis shows that eSLIM significantly improves upon CIOPS and that eSLIM is a viable alternative to DEEPSYN. Additionally, the experiments show that the SAT-based approach outperforms the QBF-based version for instances from the IWLS'23 programming contest. For circuits from the EPFL suite, the two variants perform very similarly, with either approach having a slight edge in some instances. Moreover, the experiments show that using windowing allows us to further reduce the largest circuits from the EPFL suite.

The respectable performance of the QBF-based approach on larger circuits hints at the potential of adopting techniques from QBF solving, such as counterexample-guided expansion [19]. Computing the entire Boolean relation upfront can be prohibitive, and generating constraints during the substitution process could be more efficient. Specifically, constraints could be added on-the-fly when the substitution of a synthesized circuit alters the function.

## References


1 Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The EPFL combinational benchmark suite. In *International Workshop on Logic & Synthesis (IWLS)*, 2015. URL: https://github.com/lsils/benchmarks.

2 Olaf Beyersdorff, Mikolás Janota, Florian Lonsing, and Martina Seidl. Quantified Boolean formulas. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1177–1221. IOS Press, 2021.

3 Armin Biere. The AIGER And-Inverter Graph (AIG) format version 20071012. Technical Report 07/1, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2007.

4 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

5 Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and beyond. Technical Report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

6 Robert K. Brayton, Gary D. Hachtel, and Alberto L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proc. IEEE*, 78(2):264–300, 1990.

7 Robert K. Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *CAV*, volume 6174 of *LNCS*, pages 24–40. Springer, 2010. doi:10.1007/978-3-642-14295-6_5.

8 Huan Chen, Mikolás Janota, and João Marques-Silva. QBF-based Boolean function bi-decomposition. In *DATE*, pages 816–819. IEEE, 2012. doi:10.1109/DATE.2012.6176606.

9 Andrea Costamagna, Alan Mishchenko, and Giovanni De Micheli. The combinational-complexity game for symmetric functions. In *IWLS*, 2023. URL: https://si2.epfl.ch/~demichel/publications/archive/2023/AC.pdf.

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

11 Johannes K. Fichte, Neha Lodha, and Stefan Szeider. SAT-based local improvement for finding tree decompositions of small width. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *LNCS*, pages 401–411. Springer, 2017. doi:10.1007/978-3-319-66263-3_25.

12 Masahiro Fujita. Toward unification of synthesis and verification in topologically constrained logic design. *Proc. IEEE*, 103(11):2052–2060, 2015. doi:10.1109/JPROC.2015.2476472.

13 Masahiro Fujita, Satoshi Jo, Shohei Ono, and Takeshi Matsumoto. Partial synthesis through sampling with and without specification. In *ICCAD*, pages 787–794. IEEE, 2013. doi:10.1109/ICCAD.2013.6691203.

14 Masahiro Fujita, Yusuke Kimura, Xingming Le, Yukio Miyasaka, and Amir Masoud Gharehbaghi. Synthesis and optimization of multiple portions of circuits for ECO based on set-covering and QBF formulations. In *DATE*, pages 744–749. IEEE, 2020. doi:10.23919/DATE48585.2020.9116459.

15 Dale Gai. TSMC price hike indicates capacity tightness to persist in 2022, may hit smartphone shipments. online, 2021. URL: https://www.counterpointresearch.com/insights/tsmc-price-hike/.

16 Winston Haaswijk, Mathias Soeken, Alan Mishchenko, and Giovanni De Micheli. SAT-based exact synthesis: Encodings, topology families, and parallelism. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 39(4):871–884, 2020. doi:10.1109/TCAD.2019.2897703.

17 Rahul Ilango, Bruno Loff, and Igor C. Oliveira. NP-hardness of circuit minimization for multi-output functions. In Shubhangi Saraf, editor, *CCC*, volume 169 of *LIPIcs*, pages 22:1–22:36. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.CCC.2020.22.

**18** Mikolás Janota. Towards generalization in QBF solving via machine learning. In *AAAI*, pages 6607–6614. AAAI Press, 2018. doi:10.1609/AAAI.V32I1.12208.

**19** Mikolás Janota, William Klieber, João Marques-Silva, and Edmund M. Clarke. Solving QBF with counterexample guided refinement. *Artif. Intell.*, 234:1–25, 2016. doi:10.1016/j.artint.2016.01.004.

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

**21** Arist Kojevnikov, Alexander S. Kulikov, and Grigory Yaroslavtsev. Finding efficient circuits using SAT-solvers. In *SAT*, volume 5584 of *LNCS*, pages 32–44. Springer, 2009. doi:10.1007/978-3-642-02777-2_5.

**22** Alexander S. Kulikov, Danila Pechenev, and Nikita Slezkin. SAT-based circuit local improvement. In *MFCS*, volume 241 of *LIPIcs*, pages 67:1–67:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.MFCS.2022.67.

**23** Tung-Yuan Lee, Chia-Cheng Wu, Chia-Chun Lin, Yung-Chih Chen, and Chun-Yao Wang. Logic optimization with considering Boolean relations. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 761–766. IEEE, 2018. doi:10.23919/DATE.2018.8342109.

**24** Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. A SAT approach to branchwidth. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *LNCS*, pages 179–195. Springer, 2016. doi:10.1007/978-3-319-40970-2_12.

**25** Alan Mishchenko and Robert K. Brayton. SAT-based complete don't-care computation for network optimization. In *DATE*, pages 412–417. IEEE Computer Society, 2005.

**26** Alan Mishchenko, Robert K. Brayton, Jie-Hong R. Jiang, and Stephen Jang. Scalable don't-care-based logic optimization and resynthesis. *ACM Trans. Reconfigurable Technol. Syst.*, 4(4):34:1–34:23, 2011. doi:10.1145/2068716.2068720.

**27** Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. Turbocharging treewidth-bounded Bayesian network structure learning. In *Proceeding of AAAI-21, the Thirty-Fifth AAAI Conference on Artificial Intelligence*, pages 3895–3903. AAAI Press, 2021. doi:10.1609/AAAI.V35I5.16508.

**28** Kavita Ravi and Fabio Somenzi. Minimal assignments for bounded model checking. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2004. doi:10.1007/978-3-540-24730-2_3.

**29** Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. eSLIM. Software, This work was supported by the Vienna Science and Technology Fund (WWTF) under grant 10.47379/ICT19060, and the Austrian Science Fund (FWF) under grant 10.55776/W1255., swhId: swh:1:dir:9d2862aedfe458cfb06d4d43da6c0f46ed578c91 (visited on 2024-08-05). URL: https://github.com/fxreichl/eSLIM.

**30** Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. Circuit minimization with QBF-based exact synthesis. In *AAAI*. AAAI Press, 2023. doi:10.1609/AAAI.V37I4.25524.

**31** Heinz Riener, Winston Haaswijk, Alan Mishchenko, Giovanni De Micheli, and Mathias Soeken. On-the-fly and DAG-aware: Rewriting Boolean networks with exact synthesis. In *DATE*, pages 1649–1654. IEEE, 2019. doi:10.23919/DATE.2019.8715185.

**32** Heinz Riener, Alan Mishchenko, and Mathias Soeken. Exact dag-aware rewriting. In *DATE*, pages 732–737. IEEE, 2020. doi:10.23919/DATE48585.2020.9116379.

**33** Hamid Savoj and Robert K. Brayton. The use of observability and external don't cares for the simplification of multi-level networks. In *DAC*, pages 297–301. IEEE Computer Society Press, 1990.

**34** Andre Schidler and Stefan Szeider. SAT-boosted tabu search for coloring massive graphs. *J. Exp. Algorithmics*, 2825(1.5):1–19, 2023. doi:10.1145/3603112.

**35** André Schidler and Stefan Szeider. SAT-based decision tree learning for large data sets. *J. Artificial Intelligence Research*, 2024. To appear.

**36** André Schidler and Stefan Szeider. Structure-guided local improvement for maximum satisfiability. In Peter Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Catalonia*, LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. To appear.

**37** Ankit Shukla, Armin Biere, Luca Pulina, and Martina Seidl. A survey on applications of quantified Boolean formulas. In *ICTAI*, pages 78–84. IEEE, 2019.

**38** Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In Peter van Beek, editor, *CP*, volume 3709 of *LNCS*, pages 827–831. Springer, 2005. doi:10.1007/11564751_73.

**39** Eleonora Testa, Luca G. Amarù, Mathias Soeken, Alan Mishchenko, Patrick Vuillod, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. Extending Boolean methods for scalable logic synthesis. *IEEE Access*, 8:226828–226844, 2020. doi:10.1109/ACCESS.2020.3045014.

**40** University of California Berkeley. Berkeley logic interchange format (BLIF), 1992. URL: https://course.ece.cmu.edu/~ee760/760docs/blif.pdf.

**41** Robert Wille, Hoang Minh Le, Gerhard W. Dueck, and Daniel Große. Quantified synthesis of reversible logic. In *DATE*, pages 1015–1020. ACM, 2008. doi:10.1109/DATE.2008.4484814.