SAT-Based Methods for Circuit Synthesis

Roderick Bloem\textsuperscript{1}, Uwe Egly\textsuperscript{2}, Patrick Klampfl\textsuperscript{1}, Robert Könighofer\textsuperscript{1}, and Florian Lonsing\textsuperscript{2}

\textsuperscript{1}Institute for Applied Information Processing and Communications, Graz University of Technology, Austria
\textsuperscript{2}Knowledge-Based Systems Group, Institute of Information Systems, Vienna University of Technology, Austria

Abstract—Reactive synthesis supports designers by automatically constructing correct hardware from declarative specifications. Synthesis algorithms usually compute a strategy, and then construct a circuit that implements it. In this work, we study SAT- and QBF-based methods for the second step, i.e., computing circuits from strategies. This includes methods based on QBF-certification, interpolation, and computational learning. We present optimizations, efficient implementations, and experimental results for synthesis from safety specifications, where we outperform BDDs both regarding execution time and circuit size. This is an extended version of \cite{BloemEtAl:2011}, with an additional appendix.

I. INTRODUCTION

Synthesis is an ambitious design approach: Instead of checking whether an already constructed system satisfies its specification, a correct implementation is derived automatically from the specification \cite{BloemEtAl:2011}. Synthesis is also used in rapid prototyping, automatic repair \cite{BloemEtAl:2010}, and program sketching \cite{BloemEtAl:2014}.

Existing work often focuses on finding strategies to satisfy the specification, or only on deciding realizability. However, computing circuits from strategies is computationally demanding as well. System quality (e.g., circuit size and depth) imposes additional challenges. Synthesized strategies usually allow for much implementation freedom, which needs to be exploited cleverly. Algorithms must also be symbolic (operate on formulas rather than enumerating states) to achieve scalability. These symbolic algorithms are usually implemented with BDDs because they offer existential and universal quantification. Recently, SAT-based synthesis algorithms have been proposed \cite{KampmannEtAl:2015,BloemEtAl:2015} as alternative to BDDs and their scalability issues. However, these works do not address circuit extraction.

We thus present and compare several SAT- and QBF-based circuit synthesis algorithms. The basic algorithms are not new, but we present novel optimizations, combinations, efficient implementations for safety synthesis problems, and extensive experiments. This includes methods based on QBF-certification, computational learning (including the first application of incremental QBF solving in synthesis), and interpolation. We achieve the best results by combining ideas from interpolation \cite{JiangEtAl:2013} with learning \cite{BloemEtAl:2015}, thereby outperforming BDDs both regarding computation time and circuit size.

Related work. It is argued \cite{BloemEtAl:2015} that many circuit synthesis methods are still outperformed by the simple BDD-based cofactor approach \cite{BloemEtAl:2011}. The same work \cite{BloemEtAl:2015} also proposes learning-based techniques, which are implemented with BDDs. This yields smaller circuits, but is slower. We show how learning can be efficiently realized with SAT- and QBF-solvers, and that this realization can outperform the cofactor approach both regarding circuit size and computation time. SAT-based learning is also used in \cite{BloemEtAl:2015}. However, this work only addresses strategy computation and not circuit synthesis. Jiang et al. \cite{JiangEtAl:2013} propose interpolation for circuit extraction, and show how quantifier alternations can be avoided by temporarily treating outputs as inputs. We combine this idea with learning to compute interpolants, thereby achieving a speedup of several orders of magnitude. QBF certification \cite{BloemEtAl:2015} can derive circuits from a completeness proof of the strategy formula. We show how this method can be applied efficiently for safety synthesis.

II. PRELIMINARIES

We assume familiarity with propositional logic, SAT- and QBF-solving (cf. \cite{BiereEtAl:2009}) but repeat the most important concepts.

**Basic Notation.** A literal is a Boolean variable or its negation. A clause (cube) is a disjunction (conjunction) of literals, and a Conjunctive Normal Form (CNF) formula is a conjunction of clauses. We denote variables vectors with overlines, corresponding cubes in bold, and propositional formulas with capital letters. E.g., $x$ is a cube over the variable vector $\overline{x} = (x_1, \ldots, x_n)$, and $F(\overline{\pi})$ is a formula over $\overline{\pi}$. If the variables are irrelevant, we simply write $F$ instead of $F(\overline{\pi})$.

Decision Procedures. A SAT-solver checks if a CNF is satisfiable. We write $(\text{sat}, x) := \text{PSAT}(F(\overline{\pi}))$ for a SAT-solver call, where $\text{sat}$ is assigned true iff the CNF $F$ is satisfiable, and $x$ is a satisfying assignment given as cube over $\overline{x}$. Let $x$ be a cube. We write $y := \text{PCORE}(x, F)$ to denote the extraction of an unsatisfiable core: Given that $x \land F$ is unsatisfiable, $y$ will be a sub-cube of $x$ such that $y \land F$ is still unsatisfiable. Let $A(\overline{\pi}, \overline{\gamma})$ and $B(\overline{\pi}, \overline{\tau})$ be two CNFs such that $A \land B$ is unsatisfiable, and $\overline{\gamma}$ and $\overline{\tau}$ are disjoint. An interpolant is a formula $I(\overline{\pi})$ such that $A \Rightarrow I \Rightarrow \neg B$. Interpolants can be computed from the unsatisfiability proof of $A \land B$ \cite{AtseriasEtAl:2009}. We denote this computation by $I := \text{INT}(A, B)$. A Quantified Boolean Formula (QBF) is a formula $Q_1(\overline{\pi}, Q_2(\overline{\tau}, \ldots F(\overline{\pi}, \overline{\tau}, \ldots))$, where $Q_i \in \{\forall, \exists\}$ and $F$ is a CNF. The quantifiers have their expected semantics. A QBF-solver checks if a QBF is satisfiable. We write $(\text{sat}, a) := \text{QSAT}(\exists \overline{\pi}. Q_1(\overline{\pi}, Q_2(\overline{\tau}, \ldots F(\overline{\pi}, \overline{\tau}, \ldots)))$ for QBF-solver calls. The satisfying assignment $a$ can only be extracted for variables that are quantified existentially on the outermost level. Finally, we write $b := \text{QCORE}(a, \exists \overline{\pi}. Q_1(\overline{\pi}, Q_2(\overline{\tau}, \ldots F(\overline{\pi}, \overline{\tau}, \ldots)))$ to denote the extraction of an unsatisfiable core.
Circuit Synthesis. A strategy is a formula \( S(\pi, \bar{i}, \bar{o}, \bar{x}) \) such that \( \forall \pi, \bar{i}, \exists \bar{o}, \bar{x} \). \( S \). For a given state \( x \) and input \( i \), \( S \) defines allowed output-values \( o \) and next states \( x' \); \( o, x' \) is allowed iff \( x \land i \land o \land x' \) satisfies \( S \). Let \( \pi = \pi \lor i \) and \( \overline{\pi} = \overline{\pi} \lor \overline{i} \). An implementation of \( S(\pi, \overline{\pi}) \) is a function \( f : \mathcal{P}(\mathcal{P}) \rightarrow \mathcal{P}(\mathcal{P}) \) such that \( \forall \pi, S(\pi, f(\pi)) \). This function can be implemented in hardware as shown in Fig. 1.

Strategies for safety specifications are particularly simple: given a winning region \( \pi(\pi) \) from which the specification can be enforced, and a complete and deterministic transition relation \( T(\pi, \bar{i}, \bar{o}, \bar{x}) \) defining the state transitions, the strategy must pick values for \( \overline{\pi} \) such that the next state is in \( W \), i.e., \( \pi(\pi) = (\neg W(\pi)) \lor (T(\pi, \bar{i}, \bar{o}, \bar{x}) \land W(\pi')) \). We only need to synthesize circuits for \( \overline{\pi} \) and define \( \pi' \) using \( T \).

III. Circuit Synthesis Algorithms

A. QBF-Certification

An implementation can be computed as Skolem functions \( f \) for the signals \( \sigma \) and \( \pi' \) in the QBF \( \forall \pi, \exists \bar{i}, \exists \bar{o}, \exists \bar{x}, S(\pi, \bar{i}, \bar{o}, \bar{x}) \). QBF-Cert \( [13] \) computes such functions using DepQBF \( [10] \).

Optimizations for Safety Specifications. We need to find Skolem functions for \( \sigma \) in \( \forall \pi, \exists \bar{i}, \exists \bar{o}, \exists \bar{x}, (\neg W) \lor (T \land W) \). Yet, we achieve better results with QBF-Cert by computing Herbrand functions \( f \) in the unsatisfiable QBF \( \exists \bar{o}, \exists \bar{x}, (\neg W) \lor (T \land W) \). This works because \( T \) is deterministic and complete. In our setting, \( W \) is in CNF, so the conjunctions in the latter formulation are simpler to realize in CNF. Also, the clause resolution proofs required for unsatisfiable QBFs are usually less expensive than the cube resolution proofs for satisfiable ones. Still, the intermediate files produced by QBF-Cert can grow large (hundreds of GB). One reason is that a straightforward CNF encoding of \( \neg W \) requires many auxiliary variables and clauses. We could reduce the size of the files by up to a factor of 30 by learning a CNF representation of \( \neg W \) without introducing auxiliary variables using the following algorithm:

1. procedure SYLearnQBFS(\( S(\pi, \bar{i}, \bar{o}, \bar{x}) \))
2. \( \pi := \pi \lor i \), \( \overline{\pi} := \overline{\pi} \lor \overline{i} \)
3. for \( v \in \pi \) do
4. \( \pi_v := \pi_v \setminus \{v\} \), \( \overline{\pi_v} := \overline{\pi_v} \cup \overline{v} \), \( f_v := \text{true} \), \( R := v \lor \neg S \)
5. while sat, with \( (\text{sat}, u) := \text{QSat}(\exists \pi, \forall \bar{i}, \exists \bar{o}, \exists \bar{x}, (\neg W) \lor (T \land W)) \) do
6. \( u_2 := \text{QCore}(\exists \pi, \forall \bar{i}, \exists \bar{o}, \exists \bar{x}, (\neg W) \lor (T \land W)) \)
7. \( f_v := f_v \land \neg u_2 \), \( R := R \land \neg u_2 \)
8. DUMP Circuit \( f_v \), \( S := S \lor (v \leftrightarrow f_v) \)

SYLearnQBFS builds up circuits in \( f_v \) for one \( v \in \pi \) after the other. Initially, \( f_v = \text{true} \), i.e., the circuit always outputs true. While there exists an input \( u \) for which \( v \) must be false (the QBF in line 5 is satisfiable), \( f_v \) is refined with a clause that maps \( u \) to false. By taking the core in line 6 other inputs are also mapped to false as long as false is allowed by \( S \). The final solution \( f_v \) is dumped, and \( S \) is refined with the implementation of \( v \) before the next circuit is computed. The final \( f_v \) is in CNF, so the circuits have a depth of only two. Even after optimizations and mapping to standard cells, the depth usually remains low [7], which enables fast clock rates.

Once \( \neg S \) is available in CNF, the algorithm only adds clauses to existing CNFs (i.e., to \( R \) and \( f_v \)). Only for the resubstitution in line 8 a CNF encoding of \( \neg f_v \) is needed.

Optimizations for Safety Specifications. As in Sect. III-A, \( \neg S \) is defined as \( W \lor T \land \neg W \). This requires a CNF encoding of \( \neg W \). While computing \( \neg W \) with NegLearn is beneficial for QBF-Cert, it does not pay off for SYLearnQBFS. Hence, we build a CNF for \( \neg W \) with one auxiliary variable per clause of \( W \). Recently, the QBF solver DepQBF was equipped with incremental solving capabilities [11]. SYLearnQBFS is well suited for incremental solving. We use two solver instances for line 5 and 6 respectively. For each \( v \in \pi \), a new incremental session is started. During the inner loop, we only add clauses to the former solver. The QBF of the latter even stays the same.

DepQBF supports unsatisfiable cores natively. The resulting cores are small but not necessarily minimal, so we iterate over the remaining literals to obtain even smaller cores because (slightly) smaller cores typically mean (much) less iterations.

C. Interpolation

Jiang et al. [8] present two interpolation-based approaches to synthesize circuits for one \( v \in \pi \) after the other. The first one...
expands $S$ over $\bar{\pi}$. We consider this intractable in our setting. The second approach circumvents the quantifier alternation and expansion by temporarily treating output signals as inputs:

```
1: procedure SYINT(S(\bar{\pi}, \bar{t}, \bar{\pi}, \bar{\pi}'))
2: \overline{d} := \overline{\pi} \cup \overline{t} \cup \bar{\sigma} \cup \bar{\pi}' \quad \overline{\pi} := \emptyset
3: for v ∈ \overline{d} do
4: \overline{\pi} := \overline{\pi} \setminus \{v\} \quad \overline{\pi} := \overline{\pi} ∪ \{v\}
5: \overline{\pi}_1, \overline{\pi}_2, \overline{\pi}_3, \overline{\pi}_4 := create4FreshCopies(\overline{\pi})
6: M_1(\overline{d}, \overline{\pi}_1, \overline{\pi}_2) := (S \land v)[v ← \overline{\pi}_1] \land (¬S \land ¬v)[v ← \overline{\pi}_2]
7: M_0(\overline{d}, \overline{\pi}_3, \overline{\pi}_4) := (S \land v)[v ← \overline{\pi}_3] \land (¬S \land ¬v)[v ← \overline{\pi}_4]
8: f_v(\overline{d}) := INT(M_1(\overline{d}, \overline{\pi}_1, \overline{\pi}_2), M_0(\overline{d}, \overline{\pi}_3, \overline{\pi}_4))
9: DUMP_CIRCUIT(v, f_v), S := S \land (v ← f_v)
```

In each iteration, \overline{d} contains all variables on which the implementation of the current $v ∈ \overline{\pi}$ can depend, and \overline{\pi} contains the rest. For $\overline{\pi} = (v_1, \ldots, v_n)$, $v_1$ can depend not only on $\overline{\pi}$ but also on $(v_2, \ldots, v_n)$; $v_2$ can depend on $\overline{\pi}$ and $(v_3, \ldots, v_n)$, etc. Yet, when the circuits for all $v ∈ \overline{\pi}$ are built together, the signals $\overline{\pi}$ effectively depend on $\overline{\pi}$ only. The formulas $M_1$ and $M_0$ characterize the $\overline{d}$-vectors for which $v$ must be set to true and false respectively. The syntax $[v ← \overline{\pi}_1]$ means that the variables $\overline{\pi}$ are renamed by fresh copies $\overline{\pi}_1$. Line 8 computes an interpolant between $M_1$ and $M_0$. The property $M_1 \Rightarrow f_v ⇒ ¬M_0$ of the interpolant ensures that (a) $f_v$ is true whenever it must be true, and (b) whenever $f_v$ is true then it does not have to be false. The renaming of the variables $\overline{\pi}$ has the effect that $f_v$ can only depend on the shared signals $\overline{d}$.

**Optimizations for Safety Specifications.** In order to avoid double-negations of $W$ in $S$ by negating $S$, we compute

$$
M_1 := (T ∨ W')[v ← \overline{\pi}_1] \land (T ∨ ¬v \land W \land ¬W')[v ← \overline{\pi}_2]
$$

$$
M_0 := (T ∨ W')[v ← \overline{\pi}_3] \land (T \land v \land W \land ¬W')[v ← \overline{\pi}_4]
$$

Note the difference to a plain substitution of $S = T \land (¬W \lor W')$ and $¬S = T \land W \land ¬W'$ in $\text{SYINT}({¬W \lor W'})$ reduces to $W$ due to the conjunction with $W$ from $¬S$. This is fortunate because disjunctions are expensive in CNF. Since $\text{SYINT}$ allows $v_i$ to depend on other $v_j$ with $j > i$, it is sensitive to the variable order, both regarding execution time and circuit size. We exploit this insight with the following optimization. Once $v_i$ has been synthesized, we analyze on which $v_j$ it actually depends. If $v_i$ does not depend on a particular $v_j$, then $v_j$ is allowed to depend on $v_i$. This gives an increased flexibility without introducing circular dependencies. We simplify all computed interpolants with ABC\textsuperscript{+}[5].

**B. SAT-Based Learning**

Here, we use $\text{SYINT}$ but with a special interpolation procedure (called in line 8) that applies computational learning:

```
1: procedure INTLEARN(M_1(\overline{d}, \overline{\pi}_1, \overline{\pi}_2), M_0(\overline{d}, \overline{\pi}_3, \overline{\pi}_4))
2: f := true
3: while sat, with (sat, d) := PSAT(M_0 \land f) do
4: f := f ∧ ¬PCORE(d, M_1)
5: return f
```

As long as there exists some $d$ for which $f$ is true but must be false, i.e., $M_0 \land f$ is satisfiable, we refine $f$ with an additional clause that excludes the cube $d$ witnessing this insufficiency. By taking the unsatisfiable core, other inputs are also mapped to false as long as false is allowed.

**Optimizations.** We use two SAT solver instances, one for line 3 and one for line 4. A new incremental session is started upon each call of $\text{INTLEARN}$. Using activation variables to set $T$-variables to true, false, or equal to their renamed copy, we consider this intractable in our setting. This can be beneficial because these auxiliary variables often capture the important connections between the variables $\overline{\pi}$, $\bar{\sigma}$, $\bar{\pi}$. When dumping the circuits, we add additional gates that define the referenced auxiliary variables as done by $T$. We also implemented a second minimization pass that tries to remove every clause and literal from every CNF $f$ after $\text{SYINT}$ is done. However, this only results in minor circuit size improvements (around 20%).

**IV. EXPERIMENTAL RESULTS**

**A. Implementation**

We implemented the discussed methods and optimizations in the SAT-based synthesis tool Demiurge\textsuperscript{+}[4]. Demiurge synthesizes $\text{AIGER}$\textsuperscript{+} circuits from safety specifications and complies with the $\text{SyntCom}$\textsuperscript{+} competition rules. The archive of version 1.1.0 contains way more experiments than reported here. E.g., for the SAT-based learning approach alone we implemented 24 variants. Here, we only compare interesting versions, summarized in the following table.

| Name | Engine | Algorithm |
|------|--------|-----------|
| BDD | CuDD 2.4.2 | Cofactor-Based [3] |
| QC | QBFCert 1.0 | QBF-Certification (Sect III-A) |
| SLN | Lingeling ats | SL without dependency opt. |
| SL | MathSAT 5 | $\text{SYINT}$ (Sect III-C) |
| SI | DepQBF 3.02 | $\text{SYINT}+\text{INTLEARN}$ (Sect III-D) |
| QC | SYLEARNQBF (Sect III-B) | |

BDD serves as baseline for our comparison. It was created by students and won a competition held in a synthesis lecture. It implements a cofactor-based approach [3], uses dynamic variable reordering, and forced reorderings at certain points. QC, SL, and SI implement the methods from the previous section with all optimizations. SLN is used to highlight the benefits of the dependency optimization. All our methods use ABC\textsuperscript{+} [5] to minimize the final circuits further. SI uses

[1] We use the command sequence strash; refactor -zl; rewrite -zl; up to 3 times, followed by dfraiag; rewrite -zl; dfraiag;.
MathSAT, which supports several interpolation schemes. We use McMillan’s scheme (see [6]), but the performance is similar with other schemes. We also implemented our own interpolation engine by processing proofs produced by PicoSat. However, the proof files grew prohibitively large.

The limitations of our implementation are that it can only handle safety specifications in AIGER format, it can produce circuit only in AIGER format, and it runs under Linux only.

B. Benchmarks

We use the same benchmarks as [4], but report here only results for the interesting ones. The benchmarks ambajj specify an arbiter for ARM’s AMBA AHB bus 3, where \( i \) is the number of masters, and \( j \in \{c,b,f\} \) indicates the method used to transform the original benchmark [3] into our input format [4]. The benchmarks genbufij, again with \( j \in \{c,b,f\} \), define a generalized buffer [3] connecting \( i \) senders to two receivers. The specifications addi and multij denote \( i \)-bit combinational adders and multipliers.

C. Results and Discussion

Fig. 2 summarizes our results with cactus plots. The y-axis gives the execution time or circuit size, and the x-axis gives the number of benchmarks that can be solved within this time or size limit. Concrete numbers and more plots can be found in the appendix and in the downloadable archive. All experiments were performed on an Intel Xeon E5430 CPU running a 64 bit Linux at 2.66 GHz. The memory limit was set to 8 GB, the time-out to 10,000 seconds. All circuits have been successfully model checked.

Method SL achieves the best results both regarding execution time and circuit size. The dependency optimization (SL vs. SLN) is very beneficial for add and mult, but slower for amba and genbuf. QC, QL, and SI do not perform so well. Using incremental QBF solving in QL gives an average speedup of factor 3.5. The speedup factor compared to using the QBF preprocessor Bloqqer is even 21. Still, QL is not very competitive. BDD is much better, but still outperformed by SL. In particular, SL outperforms SI by many orders of magnitude. Hence, our idea of implementing the interpolation procedure with computational learning is very beneficial. Execution time and circuit size are not in conflict but rather correlate. The time for optimization with ABC is usually insignificant, but only yields moderate size reductions (around 25% for SL). Using method SLN, Demiurge won a track of SyntComp 2014. One reason was the small circuit size compared to other tools.

V. Conclusion

We compared several SAT- and QBF-based methods to synthesize circuits from strategies, and presented optimizations and efficient implementations for safety specifications. Our SAT-based learning method combines the quantifier elimination approach by Jiang et al. [8] with computational learning as proposed by Ehlers et al. [7], and outperforms BDDs both regarding execution time and circuit size in our experiments.

Future research includes preprocessing for incremental QBF solving, exploiting unreachable states, and parallelization.

REFERENCES

[1] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. Handbook of Satisfiability. IOS Press, 2009.
[2] R. Bloem, U. Egly, P. Klampfl, R. Könighofer, and F. Lonsing. SAT-based methods for circuit synthesis. In FMCAD’14. To appear.
[3] R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. Elect. Notes Theor. Comput. Sci., 190(4):3–16, 2007.
[4] R. Bloem, R. Könighofer, and M. Seidl. SAT-based synthesis methods for safety specs. In VMCAI’14. Springer, 2014.
[5] R. K. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In CAV’10. Springer, 2010.
[6] V. D’Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant strength. In VMCAI’10. Springer, 2010.
[7] R. Ehlers, R. Könighofer, and G. Hofferer. Symbolically synthesizing small circuits. In FMCAD’12. IEEE, 2012.
[8] J.-H. R. Jiang, H.-P. Lin, and W.-L. Hung. Interpolating functions from large boolean relations. In ICCAD’09. IEEE, 2009.
[9] B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and fixing faults. J. Comput. Syst. Sci., 78(2):441–460, 2012.
[10] F. Lonsing and A. Biere. DepQBF: A dependency-aware QBF solver. JSAT, 7(2-3):71–76, 2010.
[11] F. Lonsing and U. Egly. Incremental QBF solving. In CP’14. Springer, 2014. To appear.
[12] A. Morgenstern, M. Gesell, and K. Schneider. Solving games using incremental induction. In IJFM’13. Springer, 2013.
[13] A. Niemetz, M. Preiner, F. Lonsing, M. Seidl, and A. Biere. Resolution-based certificate extraction for QBF. In SAT’12. Springer, 2012.
[14] A. Solar-Lezama. The sketching approach to program synthesis. In APLAS’09. Springer, 2009.
Table I contains more performance results. “T” indicates a time-out of 10,000 seconds. The suffix $k$ stands for a multiplication with 1000. The size column $G$ gives the number of AIGER gates defining $T$. The column “files” in QC gives the size of the intermediate files (the QBF trace) produced by QBFCert; we aborted at 20 GB. “M” indicates that ABC ran out of memory (because QC produces huge circuits). More details can be found in the downloadable archive.

### TABLE I

**PERFORMANCE RESULTS.**

| Size | BDD | QC | QL | SI | SL | SLN |
|------|-----|----|----|----|----|-----|
| $| | | | | | |
| add2 | 4 | 2 | 17 | 0.1 | 2 | 1 | 13 | 0.1 |
| add4 | 8 | 2 | 45 | 0.1 | 4 | 1 | 9 | 0.3 |
| add6 | 12 | 6 | 73 | 0.1 | 6 | 11 | 20k | 11 |
| add8 | 16 | 8 | 101 | 0.1 | 8 | 164 | M | 738 |
| add10 | 20 | 10 | 129 | 0.1 | 10 | T | - | >17k |
| add12 | 24 | 12 | 157 | 0.1 | 12 | - | - | >20k |
| add14 | 28 | 14 | 185 | 0.1 | 14 | - | - | >20k |
| add16 | 32 | 16 | 213 | 0.1 | 16 | - | - | >20k |
| add18 | 36 | 18 | 241 | 0.1 | 18 | - | - | >20k |
| add20 | 40 | 20 | 269 | 0.2 | 20 | - | - | >20k |
| mult2 | 4 | 4 | 24 | 0.1 | 4 | 1 | 8 | 0.1 |
| mult4 | 8 | 8 | 128 | 0.1 | 8 | 1 | 10.4k | 6.1 |
| mult5 | 10 | 10 | 217 | 0.2 | 10 | 12 | 49k | 142 |
| mult7 | 14 | 14 | 455 | 0.2 | 14 | 380 | 23k | 57k |
| mult8 | 16 | 16 | 604 | 0.2 | 16 | 21k | 51k | 132k |
| mult10 | 20 | 20 | 964 | 0.2 | 20 | 61k | 151k | 766k |

**Table Contents:**

- **BDD**: Size of intermediate BDDs in bytes.
- **QC**: Size of intermediate QC files in bytes.
- **QL**: Size of intermediate QL files in bytes.
- **SI**: Size of intermediate SI files in bytes.
- **SL**: Size of intermediate SL files in bytes.
- **SLN**: Size of intermediate SLN files in bytes.

The table shows the size of various intermediate files produced by different circuits and operations, with annotations for time-outs and memory limitations.
Fig. 3 contains cactus plots summarizing the execution time and circuit size per benchmark. For mult, the dependency optimization has an even stronger positive impact than for add, both regarding execution time and circuit size (SL vs. SLN). For genbuf, it results in smaller circuits, but is slower. For amba, it is not beneficial in either metric. The interpolation-based method SI is competitive for add, but has troubles with the other benchmarks. The difference in circuit size can grow very large (more than three orders of magnitude). Our a-posteriori circuit minimization using ABC can only compensate an insignificant fraction of this difference. Investing more effort in post-processing can be expensive, especially for large circuits. We therefore conclude that circuit size is best considered during the synthesis process already. The fact that circuit size and execution time correlate is not surprising. Most of our methods compute circuits for one output signal after the other. The strategy is refined with the circuit for one signal before continuing with the next one. This is necessary in order to prevent uncoordinated choices. If the computed circuits are complicated, then this re-substitution makes the strategy formula for the next output complicated, which results in higher computation times. This may also explain why implementing an interpolation procedure with computational learning is very beneficial (SL vs. SI): computational learning seems to find smaller circuits, and this also pays off in terms of the overall computation time.