A fast symbolic transformation based algorithm for reversible logic synthesis

Mathias Soeken[1], Gerhard W. Dueck[2], and D. Michael Miller[3]

[1] Integrated Systems Laboratory, EPFL, Switzerland
[2] University of New Brunswick, NB, Canada
[3] University of Victoria, BC, Canada
mathias.soeken@epfl.ch

Abstract. We present a more concise formulation of the transformation based synthesis approach for reversible logic synthesis, which is one of the most prominent explicit ancilla-free synthesis approaches. Based on this formulation we devise a symbolic variant of the approach that allows one to find a circuit in shorter time using less memory for the function representation. We present both a BDD based and a SAT based implementation of the symbolic variant. Experimental results show that both approaches are significantly faster than the state-of-the-art method. We were able to find ancilla-free circuit realizations for large optimally embedded reversible functions for the first time.

Keywords: Reversible circuit synthesis, symbolic methods, binary decision diagrams, Boolean satisfiability

1 Introduction

The most important application areas of reversible logic are quantum computing and low power design. Due to the requirement of reversibility, only n-input and n-output Boolean functions that represent permutations can be considered. One of the most important problems to solve is *synthesis*, which is the problem of finding a circuit that realizes a given reversible function $f : \mathbb{B}^n \to \mathbb{B}^n$.

Due to the reversibility, a reversible circuit cannot have fanout. Therefore, it is composed as a cascade of reversible gates. The circuit has $r \geq n$ circuit lines. If $r = n$, i.e., no additional ancilla line is required to realize f, the synthesis is called ancilla-free. So far, almost all presented ancilla-free synthesis approaches (e.g., [9, 4, 7]) use an explicit representation of f, e.g., as a truth table or a permutation, which grows exponentially with n. Consequently, the approaches are not applicable to large reversible functions. Recently, two ancilla-free synthesis approaches [15, 14] have been presented that work on a symbolic representation of f (using decision diagrams) and therefore overcome the limitation and are applicable to much larger functions.

In this paper, we present a symbolic ancilla-free synthesis approach based on the most prominent explicit ancilla-free synthesis approach, called transformation based synthesis [9]. We reformulate the algorithm in a more concise way

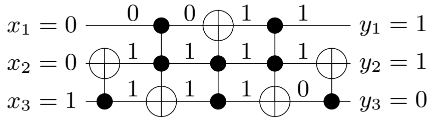


Fig. 1. Example reversible circuit with sample simulation

and derive properties which we exploit in the symbolic variant. In addition to a binary decision diagram (BDD) based implementation that follows principles from [15] and [14], we also present an implementation based on Boolean satisfiability (SAT) of the symbolic synthesis approach for the first time. So far Boolean satisfiability was only used for minimal circuit synthesis [6, 20], which is only applicable to very small functions. Due to the symbolic description of the algorithm, it can be performed using fewer computation steps and using lower memory requirements for the representation of f. An experimental evaluation shows that the SAT based implementation outperforms the BDD based approach and both approaches outperform the previously presented symbolic approaches significantly.

The contributions of the paper can be summarized as follows: (1) a more concise formulation for the transformation based synthesis approach presented in [9], (2) a generic symbolic variant of the algorithm, and (3) two implementations, one based on BDDs and one based on SAT. With these contributions we were able to find ancilla-free circuit realizations for several benchmarks for the first time (including reversible functions with 68 variables). All these contributions make our approach particularly interesting for hierarchical reversible logic synthesis to ensure local optimal results with respect to the number of ancilla lines.

2 Preliminaries

A reversible function is a Boolean multi-output function $f : \mathbb{B}^n \to \mathbb{B}^n$ that is bijective, i.e., every possible input pattern corresponds to a unique output pattern. Let X be a set of lines identified as $\{1, \dots, n\}$. In this work, we consider the family of multiple-controlled Toffoli gates. A Toffoli gate, denoted $\mathrm{T}(C, t)$, inverts a target line $t \in X$ if, and only if, the value of each control line in $C \subseteq X \setminus \{t\}$ is 1. All other lines remain unchanged. If $|C| = 0$ or if $|C| = 1$, we refer to the gate as a NOT gate or a CNOT gate, respectively. As an example, the gate $\mathrm{T}(\{1, 2\}, 3)$ inverts the value of line 3 if, and only if, the first two lines are set to 1. We use the customary notation of solid circles to denote control lines and the '$\oplus$' symbol to denote the target line. Fig. 1 shows an example circuit using this notation with a simulation of the assignment $001 \mapsto 110$.

The assignment of a variable x_i in a Boolean function $f(x_1, \dots, x_n)$ is referred to as the *co-factor* of f with respect to x_i. If x_i is assigned 1, the co-factor is called positive and denoted f_{x_i}. Otherwise, it is called negative and denoted $f_{\bar{x}_i}$. Existential quantification of a variable in a Boolean function, also called smoothing, is defined as $\exists x_i\, f = f_{\bar{x}_i} \vee f_{x_i}$. The effect is that all occurrences of x_i and $\bar{x}_i$ are removed from an expression representing f.

Let $f(x_1, \dots, x_n) = (y_1, \dots, y_m)$ be a multiple-output function, where each output is specified by a Boolean function $y_i = f_i(x_1, \dots, x_n)$. Then the *characteristic function* of f is

$$F(x_1, \dots, x_n, y_1, \dots, y_m) = \bigwedge_{i=1}^{m} \left(\bar{y}_i \oplus f_i(x_1, \dots, x_n)\right). \tag{1}$$

Note that $\bar{a} \oplus b = \overline{a \oplus b}$ is the XNOR operation.

Due to space limitations we refer the reader to the relevant literature for binary decision diagrams (e.g., [3]) and Boolean satisfiability (e.g., [2]).

3 Truth table based algorithm

The transformation based synthesis algorithm [9] is one of the first and most popular ancilla-free synthesis algorithms for reversible functions. It works on the truth table representation of a reversible function $f : \mathbb{B}^n \to \mathbb{B}^n$. In each step, f is updated by applying a gate $g = T(C, t)$:

$$f \leftarrow f \circ g \tag{2}$$

This step is repeated until f is the identity function and at completion the composition of all collected gates $g_1, \dots, g_k$ realizes the original f.

The gates are selected such that they *transform* output patterns to input patterns in an assignment $x \mapsto y$ with $x = x_1 \dots x_n$ and $y = y_1 \dots y_n$. The step in (2) is repeatedly applied to transform y in order to match x. Each gate will change one bit position y_i that differs from x_i in an order that first 0's are changed to 1's and then 1's are changed to 0's. Let $X_p = \{i \mid x_i = p\}$ and $Y_p = \{i \mid y_i = p\}$ partition the bits in x and y according to their polarities. The sets $X_1 \cap Y_0$ and $X_0 \cap Y_1$ characterize the bit positions in which x and y differ. Inserting the gate sequence

$$\bigcirc_{i \in X_1 \cap Y_0} \mathrm{T}(Y_1, i) \circ \bigcirc_{i \in X_0 \cap Y_1} \mathrm{T}(X_1, i) \tag{3}$$

in reverse order to the front end of the circuit will transform the output pattern such that it matches the input pattern. Here, '$\bigcirc$' denotes the accumulation symbol for functional decomposition. Besides transforming y to match x, the gates also transform other output patterns in the truth table. However, the following essential property holds. All output patterns $y' \neq y$ such that $y' \leq x$ are not affected by any gate in (3). This property was first observed in [1]. Hence, applying this transformation to all input/output assignments $x^{(i)} \mapsto y^{(i)}$ for $1 \leq i \leq 2^n$ will result in the identity function if the input patterns $x^{(i)}$ are ordered such that

$$x^{(i)} \neq x^{(j)} \text{ for all } i \neq j \text{ and } x^{(i)} \not\leq x^{(j)} \text{ for all } i > j, \tag{4}$$

$x_1x_2x_3$	$y_1y_2y_3$						$X_1 \cap Y_0$	$X_0 \cap Y_1$
000	111	$\underline{000}$	000	000	000	000	$\emptyset$	$\{1,2,3\}$
001	000	111	$\underline{001}$	001	001	001	$\emptyset$	$\{1,2\}$
010	110	001	111	$\underline{010}$	010	010	$\emptyset$	$\{1,3\}$
011	100	011	101	101	$\underline{011}$	011	$\{2\}$	$\{1\}$
100	010	101	011	110	100	100		
101	001	110	110	011	111	$\underline{101}$	$\emptyset$	$\{2\}$
110	011	100	100	100	110	110		
111	101	010	010	111	101	111		

Fig. 2. Example application of the transformation based synthesis method. The synthesis adjusts five input/output assignments whose values after gate application are underlined. The two rightmost columns show the values of the sets $X_1 \cap Y_0$ and $X_0 \cap Y_1$ in each of these steps.

where '$\leq$' refers to bitwise comparison. Ordering the input patterns with respect to their integer representation, i.e., $0 \dots 00, 0 \dots 01, \dots, 1 \dots 11$ satisfies (4). The following algorithm formalizes the synthesis approach using this order.

Algorithm T. (*Transformation based synthesis*). Given an n-variable reversible function f, this algorithm computes a reversible circuit C that realizes f by transforming output patterns in numerical order of their corresponding input patterns.

T1. [Initialize.] Let C be empty and set $x \leftarrow 0$.

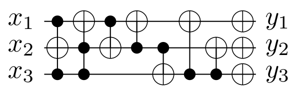
T2. [Prepend gates.] Compute X_0, X_1, Y_0, and Y_1 for x and $y = f(x)$. Set

$$f \leftarrow f \circ \bigcirc_{i \in X_1 \cap Y_0} \mathrm{T}(Y_1, i) \circ \bigcirc_{i \in X_0 \cap Y_1} \mathrm{T}(X_1, i) \tag{5}$$

according to (3) and prepend the gates in reverse order to C.

T3. [Terminate?] If $x = 2^n - 1$, terminate. Otherwise, set $x \leftarrow x + 1$ and return to step 2.

The function f is updated in (5) by adjusting all output patterns in f that match the control lines of the gates. An example application of the algorithm using this order is given in Fig. 2 that results in the reversible circuit

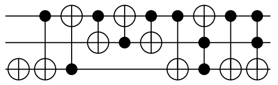
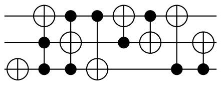


The time complexity of Algorithm T is exponential in the number of variables for all functions f since x is incremented by 1 in step 3 until all 2^n input patterns have been considered. One can check whether $f(x) \neq x$ before computing X_0, X_1, Y_0, and Y_1. However, the gain in efficiency is negligible. The problem is that there is no way to skip a whole sequence of assignments and *jump* to the next one that requires adjustment. In Section 5, we will present a symbolic variant of Algorithm T that allows such jumps, enabling a linear time complexity for certain classes of functions in the best case.

Since gates are appended to the front end of the circuit at each step, the algorithm is referred to as backward-directed transformation based synthesis. The algorithm can also be applied in the forward direction by adjusting input patterns to match their output patterns. For this purpose, gates

$$\bigcirc_{i \in X_0 \cap Y_1} \mathrm{T}(X_1, i) \circ \bigcirc_{i \in X_1 \cap Y_0} \mathrm{T}(Y_1, i) \tag{6}$$

are appended to the back end of the circuit at each step and the output patterns $y^{(i)}$ are ordered with respect to the constraints that are obtained by replacing x with y in (4). The two approaches can be combined into a bidirectional approach by fixing a valid order of patterns $z^{(1)}, \dots, z^{(2^n)}$ and at each step i either inserting gates according to (3) in the backward direction to adjust the assignment $z^{(i)} \mapsto f(z^{(i)})$ or inserting gates according to (6) in the forward direction to adjust the assignment $f^{-1}(z^{(i)}) \mapsto z^{(i)}$. A good heuristic for choosing the direction is to select the assignment with the smaller Hamming distance, which directly corresponds to the number of gates. The circuits for the function in Fig. 2 obtained by applying the algorithm in the forward direction and bidirectional are

 and 

respectively. In the case of a tie for the Hamming distance in the bidirectional approach, backward direction was chosen.

The two circuits obtained from the unidirectional approaches each consist of 10 gates, whereas using the bidirectional algorithm a circuit consisting of 8 gates can be obtained. An optimal realization for the function using Tofolli gates with positive control lines requires 7 gates.

It is worth noting that the Toffoli gates in (3) to change 0's to 1's have the same set of control lines Y_1 and the Toffoli gates to change 1's to 0's have the same set of control lines X_1. This fact can be emphasized and the representation can be made more concise when allowing Toffoli gates to have multiple targets by passing a set of lines instead of a line as the second parameter for 'T':

$$\mathrm{T}(Y_1, Y_0 \cap X_1) \circ \mathrm{T}(X_1, X_0 \cap Y_1) \tag{7}$$

4 Symbolic representation of reversible functions

We make use of binary decision diagrams to symbolically manipulate and evaluate a reversible function $f : \mathbb{B}^n \to \mathbb{B}^n$. When representing f using a BDD over n variables and n start vertices, the reversibility of f is not explicitly represented. Such a BDD representation corresponds to considering each column of the truth table representation of f individually. Instead, we use the BDD representation for the characteristic function of f (see [15, 14]), which in the remainder of the

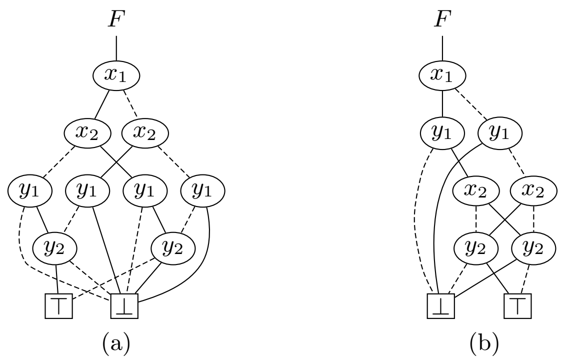


Fig. 3. BDDs for F in (8) using a) the natural and b) the interleaved variable order

paper is denoted F. Each one-path in the BDD of F represents one input/output assignments in f.

As an example consider the CNOT gate $\mathrm{T}(\{1\}, 2)$ over 2 variables. Its characteristic function is

$$\begin{aligned} F(x_1, x_2, y_1, y_2) &= (\bar{y}_1 \oplus x_1)(\bar{y}_2 \oplus x_2 \oplus x_1) \\ &= \bar{x}_1\bar{x}_2\bar{y}_1\bar{y}_2 \lor \bar{x}_1x_2\bar{y}_1y_2 \lor x_1\bar{x}_2y_1y_2 \lor x_1x_2y_1\bar{y}_2. \end{aligned} \tag{8}$$

The first expression emphasizes the functional behavior of the gate, i.e., $y_1 = x_1$ and $y_2 = x_2 \oplus x_1$, whereas the second expression lists all input/output assignments explicitly.

The order of the variables in the BDD of a characteristic function F for a reversible function f is crucial. If inputs are evaluated before outputs, e.g., in their natural order $x_1 < \cdots < x_n < y_1 < \cdots < y_n$, the size of F will always be exponential. After all inputs have been evaluated each output pattern must be represented by a node $\textcircled{y_1}$, and there are 2^n different output patterns. The same effect can be observed if all outputs are evaluated before inputs due to the function reversibility. However, if we interleave the inputs with the outputs, e.g., $x_1 < y_1 < \cdots < x_n < y_n$, compact representations are possible [16]. Fig. 3 shows the BDDs for F in (8) both in the natural and the interleaved order. Solid and dashed lines refer to high and low edges, respectively.

The symbolic representation of the reversible function f by the characteristic function F allows several operations that can be implemented efficiently using BDDs. The most important one is functional composition which is reduced to multiplication of the permutation matrices represented by the respective BDDs. Let $f_1(x_1, \ldots, x_n) = (z_1, \ldots, z_n)$ and $f_2(z_1, \ldots, z_n) = (y_1, \ldots, y_n)$ be two reversible functions and F_1 and F_2 their characteristic functions. Also, let $h = f_1 \circ f_2$ be the composition of f_1 and f_2. Then

$$H = \exists z (F_1 \land F_2) \tag{9}$$

is the characteristic function of h (see, e.g., [18]). We demonstrate how this operation works illustrated on the disjunctive normal forms that are represented by the BDDs. Note that existential quantification can be implemented using Bryant's APPLY algorithm [3] in the conventional manner. Let

$$F_1 = \bar{x}_1\bar{x}_2\bar{z}_1\bar{z}_2 \vee \bar{x}_1 x_2\bar{z}_1 z_2 \vee x_1\bar{x}_2 z_1 z_2 \vee x_1 x_2 z_1\bar{z}_2$$

and

$$F_2 = \bar{z}_1\bar{z}_2 y_1\bar{y}_2 \vee \bar{z}_1 z_2 y_1 y_2 \vee z_1\bar{z}_2\bar{y}_1\bar{y}_2 \vee z_1 z_2\bar{y}_1 y_2$$

be the characteristic functions of the CNOT gate $\mathrm{T}(\{1\},2)$ and the NOT gate $\mathrm{T}(\{\},1)$, respectively. The operation $F_1 \wedge F_2$ pairs up those minterms for which the polarities of z_1 and z_2 are equal; all other combinations evaluate to false and therefore vanish from the expression, i.e.,

$$F_1 \wedge F_2 = \bar{x}_1\bar{x}_2\bar{z}_1\bar{z}_2 y_1\bar{y}_2 \vee \bar{x}_1 x_2\bar{z}_1 z_2 y_1 y_2 \vee x_1\bar{x}_2 z_1 z_2\bar{y}_1 y_2 \vee x_1 x_2 z_1\bar{z}_2\bar{y}_1\bar{y}_2,$$

and existentially quantifying over z_1 and z_2 removes these *gluing variables* from the expression:

$$\exists z_1 \exists z_2 (F_1 \wedge F_2) = \bar{x}_1\bar{x}_2 y_1\bar{y}_2 \vee \bar{x}_1 x_2 y_1 y_2 \vee x_1\bar{x}_2\bar{y}_1 y_2 \vee x_1 x_2\bar{y}_1\bar{y}_2.$$

If the variable names don't match, one can also use existential quantification to locally rename them. To rename a function $F(x,y)$ to $F(x,z)$, denoted $F_{y\to z}$, one computes

$$\exists y \left(F \wedge \textstyle\bigwedge_{i=1}^{n} \bar{y}_i \oplus z_i\right). \tag{10}$$

5 Symbolic algorithm

The truth table based variant of the transformation based algorithm visits all 2^n assignments. It is not possible to only visit the assignments that need adjustment, i.e., for which the output pattern differs from the input pattern. In this section we discuss a symbolic variant of the algorithm. Besides a symbolic representation of the function, which can decrease the space requirements, a major difference of the symbolic variant is the order in which the assignments are visited. The truth table based variant of the algorithm, Algorithm T, visits all assignments in numerical order of the input patterns. For example, if $n = 4$, the order is

$$\begin{array}{c}0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111,\\ 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111.\end{array} \tag{11}$$

Ordering the input patterns according to their Hamming weight, i.e., the number of ones, is also valid. The symbolic variant of the transformation based algorithm makes use of this property. For $n = 4$, the order is

$$0000, \quad \begin{array}{c}0001,\\0010,\\0100,\\1000,\end{array} \quad \begin{array}{c}0011,\\0101,\\0110,\\1001,\\1010,\\1100,\end{array} \quad \begin{array}{c}0111,\\1011,\\1101,\\1110\end{array} \quad 1111 \tag{12}$$

where the order of patterns within a set of patterns of the same Hamming weight can be arbitrary. The key difference in the symbolic variant is that it iterates through these sets (of which there are $n+1$, i.e., linearly many) instead of iterating through all patterns individually (of which there are 2^n, i.e., exponentially many). For each set the algorithm extracts assignments (in a possibly arbitrary order) that are not matched yet and disregards those that are already matched. The constraint (4) ensures that the inserted gates do not affect any other assignment within the current set or previously considered sets.

For an n-variable reversible function f, we can symbolically represent all input patterns x with Hamming weight k such that $f(x) \neq x$ with the expression

$$F(x, y) \wedge S_{=k}(x) \wedge D(x, y) \tag{13}$$

where F is the characteristic function of f, $x = x_1, \dots, x_n$, and $y = y_1, \dots, y_n$. The symmetric function

$$S_{=k}(x) = [x_1 + \cdots + x_n = k] \tag{14}$$

restricts the assignments to those that have input patterns of Hamming weight k. The function

$$D(x, y) = \bigvee_{i=1}^{n} x_i \oplus y_i \tag{15}$$

further restricts the assignments such that the input patterns and output patterns differ in at least one bit. These are all requirements to describe the symbolic transformation based algorithm.

Algorithm S. (*Symbolic transformation based synthesis*). Given the characteristic function F to an n-variable reversible function f, this algorithm finds a circuit C that realizes f. In this algorithm, f and F always refer to the same function in different representations.

S1. [Initialize.] Let C be empty and set $k \leftarrow 0$.

S2. [Terminate?] If $k = n + 1$, terminate.

S3. [Increment k.] If $F \wedge S_{=k} \wedge D = \bot$, set $k \leftarrow k + 1$ and go to step 2.

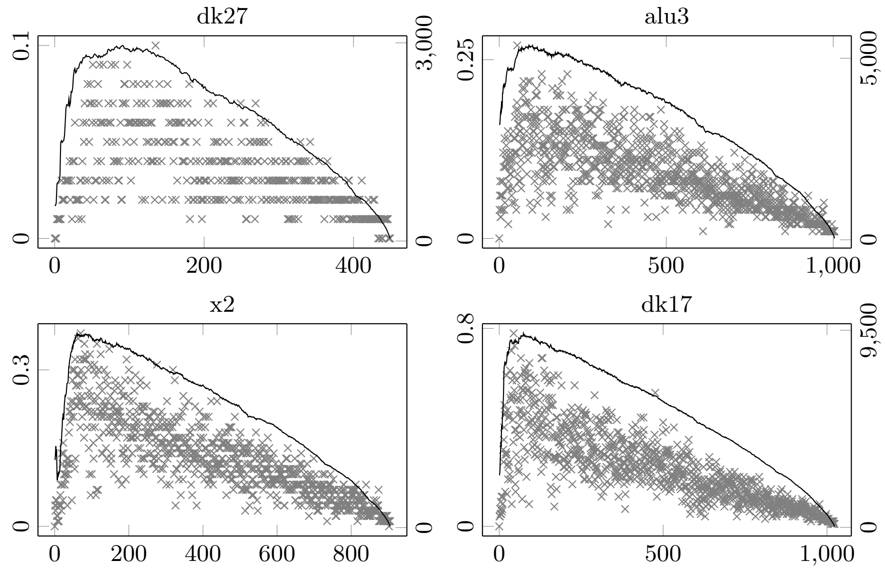
S4. [Extract assignment and prepend gates.] Extract $x \mapsto y$ by picking any minterm from $F \wedge S_{=k} \wedge D$ and compute X_0, X_1, Y_0, and Y_1. Set

$$f \leftarrow f \circ \bigcirc_{i \in X_1 \cap Y_0} \mathrm{T}(Y_1, i) \circ \bigcirc_{i \in X_0 \cap Y_1} \mathrm{T}(X_1, i) \tag{16}$$

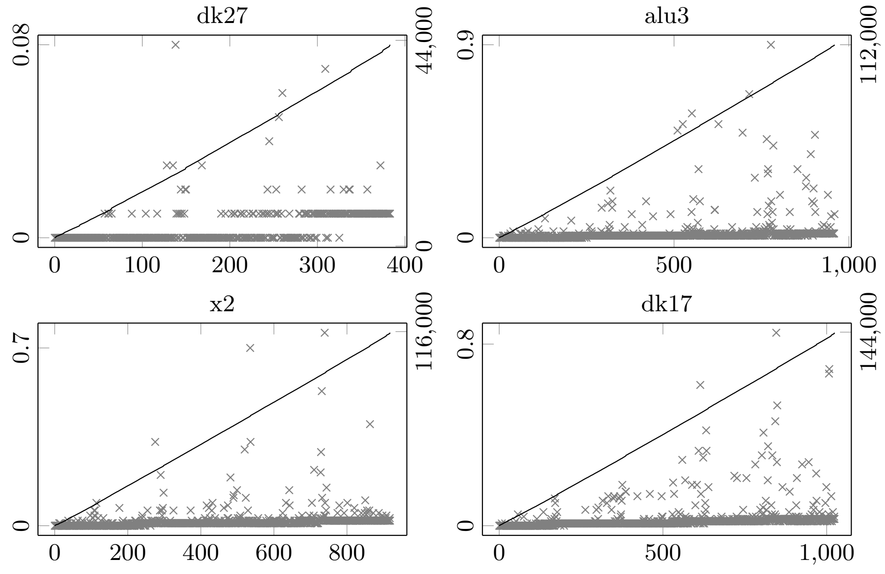
and prepend the gates in reverse order to C. Return to step 3.

The linear runtime complexity is readily verified by inspecting step 2. The possible exponential complexity comes with step 4 as it is executed $\binom{n}{k}$ times in the worst case for each $k \in \{0, \dots, n\}$ and therefore $\sum_{i=0}^{n} \binom{n}{k} = 2^n$ times in total.

In the following, two implementations of Algorithm S are described. The first uses BDDs while the second uses SAT. There are two parts in the algorithm that require individual attention depending on the underlying technique: (i) solving $F \wedge S_{=k} \wedge D$ in step 3, extracting a solution in step 4 in case of the expression being satisfiable, and (ii) updating F in step 4 according to (16).



(a)



(b)

Fig. 4. Runtime behavior of (a) the BDD based and (b) the SAT based approach. The x-axis shows the number of assignments, the left y-axis (marks) shows the runtime to solve each assignment in seconds, the right y-axis (line) shows the size of the BDD (nodes) or SAT instance (clauses)

5.1 BDD based implementation

Solving $F \wedge S_{=k} \wedge D$ is done in a straightforward way by checking whether its BDD is not equal to $\boxed{\bot}$. If that is the case, a satisfying solution can be extracted by picking any path from the start vertex to $\boxed{\top}$. Every such path visits all variables and therefore represents a minterm. This is true because each path in F already represents a minterm and $S_{=k}$ and D only restrict F further.

The BDD of F is updated by composing it with a BDD that represents the characteristic function of a gate from the right. In order to execute fewer BDD operations we make use of the fact that the gates in (16) can be expressed as two multiple-target gates as described in (7). The characteristic function of a multiple-target gate $\mathrm{T}(C,T)$ is

$$G = \bigwedge_{i \in T} \left(\bar{y}_i \oplus x_i \oplus \textstyle\bigwedge_{j \in C} x_j \right) \wedge \bigwedge_{i \in \overline{T}} (\bar{y}_i \oplus x_i) \tag{17}$$

where $\overline{T} = \{1, \dots, n\} \setminus T$. The gate can then be multiplied to F by computing $F \leftarrow \exists z \, (F_{y \to z} \wedge G_{x \to z})$ as described in (10).

5.2 SAT based implementation

For the SAT based implementation a satisfiability check is performed on the formula $F \wedge S_{=k} \wedge D$. For this purpose, the formula needs to be represented in conjunctive normal form. The characteristic function F is initially represented as a BDD as in the BDD based implementation. Each node $\textcircled{x_i}$ with children h and l represents the function $f_v = x_i \,?\, h : l$ and is translated to

$$(\bar{x}_i \vee \bar{h} \vee f_v)(\bar{x}_i \vee h \vee \bar{f}_v)(x_i \vee \bar{l} \vee f_v)(x_i \vee l \vee \bar{f}_v). \tag{18}$$

Similar clauses are added for each node $\textcircled{y_i}$. To enforce only valid input/output assignments the variable representing the start vertex f_{v_0} is added as a unit clause.[4] The subexpression $S_{=k}$ is called a *cardinality constraint* and several ways to encode such constraints as clauses have been proposed. One such encoding has been proposed in [11], which is also used in this implementation. Finally, the formula D can be translated using the Tseytin encoding for representing gates as clauses. Some of the clauses can be saved by making use of blocked clause elimination [10, 8].

The tricky part in Algorithm S is updating F in step 4. As the aim is to avoid BDD operations, we cannot just compute a new F by multiplying it with the gates that are computed from the extracted assignment. Instead, we extend the SAT formula by further constraints that represent the gates, and—since gates update the output patterns of f—compute new outputs y_i. Let $\mathrm{T}(C,T)$

[4] We also tried to write F to an AIG, perform circuit optimization, and obtain the CNF from the optimized AIG, however, no improvement in runtime could be observed, although the number of clauses can be decreased this way.

be a multiple-target gate that is added in step 4. For each $i \in T$ we add a new variable y'_i and clauses for the constraint

$$G = \bigwedge_{i \in T} \left(\bar{y}'_i \oplus y_i \oplus \textstyle\bigwedge_{j \in C} y_j \right) \tag{19}$$

using the Tseytin encoding. In this manner all created gates can be encoded and are added to the subsequent SAT calls, i.e., one updates F by setting $F \leftarrow F \wedge G_1 \wedge G_2$ where G_1 and G_2 are the two encoded multiple-target gates added in step 4. It is important to take care of the updated output variables which occur in D, see (15), and each G, see (19). For this purpose, we first introduce a set of variables $\tilde{y}_1, \dots, \tilde{y}_n$ initially set to $\tilde{y}_i \leftarrow y_i$ for $i \leq 1 \leq n$. Then, we replace y_i by $\tilde{y}_i$ and y_j by $\tilde{y}_j$ in equations (15) and (19). Finally, we update $\tilde{y}_i \leftarrow y'_i$ for each $i \in T$ in each added multiple-target gate $\mathrm{T}(C, T)$.

In order to speed up the solving process, we have implemented the SAT based approach in an incremental manner. Constraints for $S_{=k}$ and D are added using activation literals to the solver whenever updated versions are required and enforced by assuming the respective activation literals in the SAT calls (see, e.g., [5]).

5.3 Runtime behavior

A thorough experimental evaluation is given in the next section. In order to understand the differences of the BDD based and the incremental SAT based implementations of the transformation based synthesis approaches better, this section presents the results of a simple runtime evaluation experiment.

For each assignment, we recorded the runtime it took to obtain and apply the assignment (i.e., performing steps 3 and 4 in Algorithm S) as well as the size of F at that moment. In the case of the BDD implementation, the size of F is the number of nodes in the BDD, and in the case of the SAT implementation, the size of F is the number of clauses in the SAT instance. Four selected benchmarks, namely *dk27*, *alu3*, *x2*, and *dk17* serve as representatives. Other benchmarks show similar effects.

The results of the experimental evaluation are provided in the plots in Fig. 4. The x-axis shows the number of adjusted assignments, i.e., how often step 4 has been applied, the marks (left y-axis) show the required runtime in seconds to obtain and apply each assignment and the solid line (right y-axis) shows the current size of F. The four plots on the left hand side show the results for the BDD based implementation. It can be seen that the runtime to obtain and adjust an assignment correlates with the size of F. Also, the size of F is initially small, increases very quickly and then decreases towards the size of the identity function. All considered benchmark functions show this same effect; only four of them are depicted here as representatives. The effect may be explainable as follows: It is well-known that BDDs have an exponential size in the average case [19] when considering random Boolean functions, but often show reasonable space requirements for the very small subset of “nonrandom functions” that are often considered in realistic applications. The characteristic function that is input to Algorithm S is not random, and neither is the identity characteristic function which is obtained after the last step. However, applying the Toffoli gates in the course of the algorithm can depart this nonrandom space.

A completely different behavior is observed for the SAT based implementation, shown in the plots on the right hand side. The size of F increases linearly due to the addition of clauses for $S_{=k}$, D, and the gates. The runtime does not correlate with the size, although larger runtimes are only observed once the SAT instance has many clauses—yet many assignments can be obtained and adjusted in a very short time even when the number of clauses is large.

This experiment demonstrates that the SAT based approach in general is advantageous compared to the BDD based approach: i) the size of F cannot explode, since it increases linearly, and ii) the runtimes are not overly affected by the size of F. An improved encoding of the SAT instance therefore has a significant effect on the overall solution time. Also note that the size of F, both for BDDs and SAT, does not impact the size of the resulting reversible circuit, but only the number of assignments that need to be adjusted.

6 Experimental evaluation

We have implemented both symbolic transformation based synthesis approaches in C++ on top of RevKit [13] in the command '*tbs*'.[5] We have compared the approach to the state-of-the-art symbolic ancilla-free synthesis approach presented in [14] that is based on functional decomposition (DBS). It was shown that the DBS approach is faster than the approach presented in [15]. Benchmarks were taken from *www.revlib.org* as PLA files and optimally embedded using the approach presented in [16] which returns a BDD of the characteristic function. The experimental results are shown in Table 1. Besides the benchmark function, their number of inputs and outputs are listed together with the minimum number of lines after optimum embedding. The columns list the number of gates and the runtime in seconds (with a timeout of one hour, referred to as TO) required for each synthesis approach. For both transformation based synthesis approaches, also the number of adjusted assignments, i.e., how often step 4 is executed in Algorithm S, is reported.

Our main concern in this work is scalability and the possibility to obtain a circuit with a minimum number of lines. Therefore, we compare the approaches with respect to runtime. Other metrics such as gate count and quantum cost can be improved using post optimization approaches. The SAT based variant outperforms the BDD based approach, in particular for *alu1*, *apex4*, and *ex5p*. Both approaches outperform the decomposition based synthesis approach; that algorithm did not terminate for the majority of the benchmarks within the one hour timeout. In very few cases (see *parity* and *urf6*) the DBS approach was able to find a solution in which both TBS approaches did not find a solution.

[5] The code can be downloaded at https://www.github.com/msoeken/cirkit. Check the file `addons/cirkit-addon-reversible/demo.cs` for a usage demonstration.

Table 1. Experimental Results

			DBS [14]		TBS (BDD)			TBS (SAT)		
Function	I / O	lines	gates	time	assignments	gates	time	assignments	gates	time
add6	12 / 7	13	10989	997.58	4088	26479	867.21	4053	26731	837.76
alu1	12 / 8	18		TO	4012	35655	2488.80	4036	35664	414.12
alu2	10 / 6	14		TO	1021	6691	54.21	1006	6471	19.15
alu3	10 / 8	14		TO	1004	6785	65.38	959	6297	19.05
apex4	9 / 19	26		TO	452	3946	80.38	452	4007	6.76
apla	10 / 12	22		TO	1021	10351	349.09	1022	10234	28.90
cm152a	11 / 1	11	2660	21.35	2030	10644	138.68	2019	10434	84.68
cm85a	11 / 3	13	9489	836.93	1890	12324	186.89	1706	11305	62.37
dk17	10 / 11	19		TO	1023	9169	180.13	1024	8955	22.64
dk27	9 / 9	15		TO	449	2730	10.52	383	2391	1.41
ex1010	10 / 10	18		TO	1023	8874	161.13	1024	8719	21.02
ex5p	8 / 63	68		TO	253	3067	301.32	251	2943	4.83
parity	16 / 1	16	44	31.31			TO			TO
sym10	10 / 1	11	2409	36.60	1017	5445	27.07	1011	5574	16.24
urf4	11 / 11	11	2641	25.48	2027	10651	132.63	2029	10574	90.88
urf6	15 / 15	15	2164	7.00			TO			TO
x2	10 / 7	16		TO	905	7012	86.26	918	6946	14.85

When both DBS and TBS find a circuit, DBS shows a comparable performance and is sometimes even faster. Particularly, the reported gate costs for the circuits obtained by DBS is significantly lower compared to the circuits obtained from TBS. This effect is also observed when comparing the truth-table based implementations of these two algorithms. One reason for this improvement may be the ability of DBS to support mixed-polarity control lines, which cannot easily be utilized in transformation-based synthesis.

We have not compared our approach with heuristic hierarchical approaches that allow the use of additional lines. Such approaches are typically faster and can be applied to larger functions—however, with the disadvantage of adding a possibly large number of additional lines. For some functions, particularly arithmetic components, hand-crafted synthesis results exist that lead to significantly better results (see, e.g., [17])—however, we intend to have a general purpose algorithm that is not tailored to a specific function type.

7 Conclusions

In this paper we have presented a symbolic variant of the transformation-based synthesis approach for reversible logic. The approach allows the realization of large reversible functions without additional ancilla lines. It exploits a property considering the ordering in which assignments need to be considered for adjustment. Both a BDD and a SAT based implementation of the symbolic synthesis algorithm have been presented. So far, SAT has not been used for the synthesis of large reversible functions. An experimental evaluation shows that it significantly outperforms the state-of-the-art ancilla-free symbolic synthesis approaches w.r.t. runtime. For some benchmarks, ancilla-free realizations have were found for the first time. In future work, we want to integrate further optimizations that have been proposed for the truth-table variant of the algorithm, such as bidirectional adjustment and the consideration of a larger gate library [12].

Acknowledgments. This research was supported by H2020-ERC-2014-ADG 669354 CyberCare.

References

1. Alhagi, N., Hawash, M., Perkowski, M.A.: Synthesis of reversible circuits with no ancilla bits for large reversible functions specified with bit equations. In: ISMVL. pp. 39–45 (2010)
2. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
3. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE-TC 35(8), 677–691 (1986)
4. De Vos, A., Van Rentergem, Y.: Young subgroups for reversible computers. Adv. in Math. of Comm. 2(2), 183–200 (2008)

5. Eén, N., Mishchenko, A., Amla, N.: A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. In: FMCAD. pp. 181–188 (2010)
6. Große, D., Wille, R., Dueck, G.W., Drechsler, R.: Exact multiple-control Toffoli network synthesis with SAT techniques. TCAD 28(5), 703–715 (2009)
7. Gupta, P., Agrawal, A., Jha, N.K.: An algorithm for synthesis of reversible logic circuits. TCAD 25(11), 2317–2330 (2006)
8. Järvisalo, M., Biere, A., Heule, M.: Blocked clause elimination. In: TACAS. pp. 129–144 (2010)
9. Miller, D.M., Maslov, D., Dueck, G.W.: A transformation based algorithm for reversible logic synthesis. In: DAC. pp. 318–323 (2003)
10. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. JSC 2(3), 293–394 (1986)
11. Sinz, C.: Towards an optimal CNF encoding of boolean cardinality constraints. In: CP. pp. 827–831 (2005)
12. Soeken, M., Chattopadhyay, A.: Fredkin-enabled transformation-based reversible logic synthesis. In: ISMVL. pp. 60–65 (2015)
13. Soeken, M., Frehse, S., Wille, R., Drechsler, R.: RevKit: A toolkit for reversible circuit design. Multiple-Valued Logic and Soft Comp. 18(1), 55–65 (2012)
14. Soeken, M., Tague, L., Dueck, G.W., Drechsler, R.: Ancilla-free synthesis of large reversible functions using binary decision diagrams. JSC 73, 1–26 (2016)
15. Soeken, M., Wille, R., Hilken, C., Przigoda, N., Drechsler, R.: Synthesis of reversible circuits with minimal lines for large functions. In: ASP-DAC. pp. 85–92 (2012)
16. Soeken, M., Wille, R., Keszocze, O., Miller, D.M., Drechsler, R.: Embedding of large Boolean functions for reversible logic. JETC (2015), accepted, pre-print available at arXiv:1408.3586
17. Takahashi, Y., Tani, S., Kunihiro, N.: Quantum addition circuits and unbounded fan-out. Quantum Information & Computation 10(9&10), 872–890 (2010)
18. Touati, H.J., Savoj, H., Lin, B., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Implicit state enumeration of finite state machines using BDDs. In: ICCAD. pp. 130–133 (1990)
19. Wegener, I.: The size of reduced OBDDs and optimal read-once branching programs for almost all Boolean functions. IEEE Trans. Computers 43(11), 1262–1269 (1994)
20. Wille, R., Große, D., Dueck, G.W., Drechsler, R.: Reversible logic synthesis with output permutation. In: VLSI Design. pp. 189–194 (2009)