Checking Reversibility of Boolean Functions

Robert Wille[1,2], Aaron Lye[3], and Philipp Niemann[3]

[1] Institute for Integrated Circuits, Johannes Kepler University, A-4040 Linz, Austria
[2] Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[3] Institute of Computer Science, University of Bremen, D-28359 Bremen, Germany
robert.wille@jku.at {lye,pniemann}@informatik.uni-bremen.de

Abstract. Following the reversible computation paradigm is essential in the design of many emerging technologies such as quantum computation or dedicated low power concepts. The design of corresponding circuits and systems heavily relies on information about whether the function to be realized is indeed reversible. In particular in hierarchical synthesis approaches where a given function is decomposed into sub-functions, this is often not obvious. In this paper, we prove that checking reversibility of Boolean functions is indeed coNP-complete. Besides that, we propose two complementary approaches which, despite the complexity, can tackle this problem in an efficient fashion. An experimental evaluation shows the feasibility of the approaches.

1 Introduction

Reversible circuits realize an alternative computation paradigm which, in contrast to conventional circuits, employs n-input n-output functions that map each possible input vector to a *unique* output vector. In other words, bijections are realized. This provides an essential characteristic for many emerging technologies such as

- quantum computation [15], which allows for solving many practical relevant problems (e.g. factorization [18] or database search [11]) exponentially faster and relies on quantum operations that are inherently reversible or
- certain aspects in low-power design motivated by the fact that reversible computation is information loss-less and, hence, the absence of information loss (at least theoretically) helps avoiding energy dissipation during computations[1].

Besides that, superconducting quantum interference devices [16], nanoelectromechanical systems [12, 13], adiabatic circuits [2], and many further technologies utilize this computation paradigm. Even for conventional design tasks, useful

[1] Initial experiments verifying the underlying link between information-loss and thermodynamics have been reported in [3].

applications have been proposed recently, e.g. for the design of efficient on-chip interconnect codings [28].

Because of this steadily increasing interest, also the design of reversible circuits and systems is gaining interest. Here, the inherent reversibility constitutes a major obstacle. In order to not violate the paradigm, each reversible function has to be realized by a sequence or cascade of (atomic) reversible operations or gates, respectively. To this end, established gate libraries (see e.g. [24]) or assembly-like software instructions (see e.g. [23]) have been introduced in the past. But how to realize (complex) reversible functionality in terms of these atomic operations remains a major problem.

To this end, complementary approaches have been introduced in the past. One set of solutions requires a fully reversible function as input (e.g. [10, 14, 17, 20]). As frequently also irreversible functionality is to be realized, a pre-synthesis process called *embedding* is conducted before (see e.g. [21,27]). As an alternative, hierarchical solutions e.g. based on decision diagrams or two-level representations have been proposed e.g. in [25] or [8], respectively. Here, large functionality is decomposed into smaller sub-functions from which the respectively desired atomic representations can be derived.

However, both directions suffer from the fact that it is often not known whether the respectively considered (sub-)function is indeed reversible. In fact, this causes that approaches such as proposed e.g. in [10, 14, 17, 20] are usually applicable to rather small functions only, while solutions e.g. proposed in [8, 25] yield designs of very large costs (this is discussed in more detail later in Section 3). As a consequence, the non-availability of solutions for efficiently checking the reversibility of a given function constitutes a major obstacle in the design of reversible circuit and systems[2].

In this work, we are addressing this problem. We first consider the underlying problem from a theoretical perspective showing that checking reversibility for a given function is coNP-complete. Afterwards, we provide efficient solutions which tackle this problem despite the proven complexity. More precisely, two complementary approaches are proposed: one utilizing the efficient function manipulation capabilities provided by decision diagrams and another which exploits the deductive power of solving engines for Boolean satisfiability.

In an experimental evaluation we demonstrate the applicability of the proposed approaches. While both complementary strategies can efficiently handle the problem, also differences between them are unveiled. Overall, the solution based on satisfiability solvers is capable of checking the reversibility of functions in negligible run-time even for some of the largest function considered in the design of reversible circuits and systems thus far.

[2] Note that this problem has been recognized in other works concerning embedding (e.g. [21]) and synthesis (e.g. [19]). But, thus far, the issue has only been addressed peripherally and without a theoretical consideration, explicit algorithms, or an experimental evaluation.

The remainder of this work is structured as follows: The next section provides preliminaries, i.e. definitions of the different function representations utilized in this work. Section 3 discusses the importance of checking for reversibility and, hence, provides the motivation of this work. Afterwards, the complexity of the considered problem is considered in Section 4 before the two complementary approaches are introduced in Section 5 and Section 6. Results of the experimental evaluation are summarized in Section 7. Section 8 concludes this paper.

2 Preliminaries

Logic computations can be defined as functions over Boolean variables. More precisely:

Definition 1. *A* Boolean function *is a mapping* $f\colon \mathbb{B}^n \to \mathbb{B}$ *with* $n \in \mathbb{N}$. *A function* f *is defined over its* primary input *variables* $X = \{x_1, x_2, \ldots, x_n\}$ *and, hence, is also denoted by* $f(x_1, x_2, \ldots, x_n)$. *The concrete mapping may be described in terms of Boolean algebra with expressions formed over the variables from* X *and operations like* $\wedge$ *(*AND*),* $\vee$ *(*OR*), or* $\bar{\cdot}$ *(*NOT*).*

A multi-output Boolean function *is a mapping* $f\colon \mathbb{B}^n \to \mathbb{B}^m$ *with* $n, m \in \mathbb{N}$. *More precisely, it is a system of Boolean functions* $f_i(x_1, x_2, \ldots, x_n)$. *The respective functions* f_i $(1 \leq i \leq m)$ *are also denoted as* primary outputs.

The set of all *Boolean functions with* n *inputs and* m *outputs is denoted by* $\mathcal{B}_{n,m} = \{f \mid f\colon \mathbb{B}^n \to \mathbb{B}^m\}$.

In this work, we consider the design of circuits and systems realizing reversible functions. Reversible functions are a subset of multi-output functions and are defined as follows:

Definition 2. *A multi-output function* $f\colon \mathbb{B}^n \to \mathbb{B}^m$ *is* reversible *iff* f *is a bijection.*

In other words, its number of inputs is equal to the number of outputs, i.e. $f \in \mathcal{B}_{n,n}$, and it performs a permutation of the set of input patterns. A function that is not reversible is termed *irreversible*.

Besides the representation in Boolean algebra, (reversible) functions can also be represented in terms of set relations.

Definition 3. *A function* $f\colon \mathbb{B}^n \to \mathbb{B}^m$ *is by definition a relation* $F \subset \mathbb{B}^n \times \mathbb{B}^m$ *of all possible input patterns to the set of possible output patterns. For a* reversible *function, this relation additionally inherits the property that each input pattern is related to exactly one output pattern, i.e.* $\forall y \in \mathbb{B}^m : |\{x \in \mathbb{B}^n \mid (x, y) \in F\}| = 1$. *The composition of two set relations* F *and* G *(i.e. two functions* f *and* g*) is defined by* $(G \circ F) \subset \mathbb{B}^n \times \mathbb{B}^k$ *so that*

$$G \circ F = \{(x, y) \mid \exists z \in \mathbb{B}^m : (x, z) \in F \wedge (z, y) \in G\}.$$

Finally, the input/output mapping of a (reversible) function can also be represented in terms of a characteristic function.

Definition 4. *The characteristic function for a Boolean relation F is defined as $\chi_F : \mathbb{B}^n \times \mathbb{B}^m \to \mathbb{B}$ where $\chi_F(x, y) = 1$ if and only if $(x, y) \in F$.*

3 Motivation

Although never explicitly considered thus far, knowing whether a given function is reversible is an important information in the design of reversible circuits and systems. This section briefly reviews the current state-of-the-art synthesis approaches and discusses why the non-availability of corresponding checking methods constitutes a major obstacle in the development of design methods for reversible circuits and systems.

3.1 Obstacles to the Embedding Process

Not surprisingly, many design methods for reversible circuits (e.g. those proposed in [10,14,17,20]) require a fully reversible function as input. As frequently also irreversible functionality is to be realized in reversible logic, a pre-synthesis process called *embedding* is conducted before (see e.g. [21, 27]).

To this end, additional outputs (so-called *garbage outputs*) are added to the considered function $f \in \mathcal{B}_{n,m}$. More precisely, $\lceil \log_2(\mu(f)) \rceil$ additional outputs are required, whereby μ is the maximal number of times an output pattern is generated by f, i.e. $\mu(f) = \max_{y \in \mathbb{B}^m}(|\{x \mid y = f(x)\}|)$. In order to keep the number of inputs and outputs equal, this may also result in the addition of further inputs. That is, an irreversible function $f\colon \mathbb{B}^n \to \mathbb{B}^m$ is embedded into a function $f'\colon \mathbb{B}^{m+\lceil \log_2(\mu) \rceil} \to \mathbb{B}^{m+\lceil \log_2(\mu) \rceil}$. While f' is to be specified in a fully reversible fashion, the desired target functionality can be employed by setting the additionally added inputs to a constant value and recognizing only the non-garbage outputs. An example illustrates the idea.

Example 1. Consider the Boolean function $f\colon \mathbb{B}^2 \to \mathbb{B}^1$ with $f(x_1, x_2) = x_1 \wedge x_2$ to be synthesized as a reversible circuit. Obviously, f is irreversible. The maximal number of times an output pattern is generated by f is $\mu(f) = 3$ (namely 0 for the input patterns 00, 01, and 10). Hence, in order to realize f using a synthesis approach as e.g. proposed in [10, 14, 17, 20], this function has to be embedded into a function $f'\colon \mathbb{B}^{2+1} \to \mathbb{B}^{1+2}$ with $\lceil \log_2(3) \rceil = 2$ additional outputs and $1+\lceil \log_2(3) \rceil - 2 = 1$ additional input. The resulting function f'can be specified as

- $f'_1(x_1, x_2, x_3) = x_1$
- $f'_2(x_1, x_2, x_3) = x_2$
- $f'_3(x_1, x_2, x_3) = (x_1 \wedge x_2) \oplus x_3$.

This function is reversible (as can be checked by applying all $2^3 = 8$ possible input assignments) and realizes the target functionality f by setting x_3 to a constant zero value, i.e. $f = x_1 \wedge x_2 = (x_1 \wedge x_2) \oplus 0 = f'_3(x_1, x_2, 0)$.

However, generating an embedding as sketched above is an exponentially complex tasks: In order to determine μ, all 2^n output patterns generated by the inputs have to be inspected. Previous work tried to avoid this complexity by not aiming for a minimal result with respect to the number of additionally required outputs, but a heuristic one: In fact, since μ can never exceed 2^n, at most $\lceil \log_2(2^n) \rceil = n$ additional garbage outputs are required [27], i.e. *any* irreversible function can be embedded into a function $f' \colon \mathbb{B}^{n+m} \rightarrow \mathbb{B}^{m+n}$. But also here, the question remains how to specify the functionality of the newly added garbage outputs. Although heuristics assigning the additional outputs with a dedicated functionality as e.g. done in Example 1 are very promising, no solutions are available yet which guarantee that the resulting function f' is indeed reversible. As a consequence those heuristics did not become established and, hence, the design methods from [10, 14, 17, 20] mostly remain applicable to small functions only.

3.2 Obstacles to the Synthesis Process

In order to overcome the problems sketched above, researchers considered alternative synthesis schemes (see e.g. [8, 25]) relying on conventional decomposition methods (e.g. according to Shannon). Here, a given function f is decomposed with respect to an input variable x_i into two sub-functions $f_{x_i=0}$ and $f_{x_i=1}$ such that $f = (\overline{x}_i \wedge f_{x_i=0}) \vee (x_i \wedge f_{x_i=1})$ holds. The sub-functions are called co-factors of f and are obtained by assigning x_i to 0 and 1, respectively. The resulting co-factors are further decomposed until sub-functions result for which a building block is available. Plugging the resulting building blocks together eventually yields a circuit realizing the desired function. Because of this, no explicit embedding scheme is required, but the function is implicitly embedded.

In these approaches, information about the reversibility of the respectively considered (sub-)functions is essential to the quality of the resulting circuits. In fact, the decomposition almost always yields sub-functions which are not reversible anymore (even if the originally given function is). Hence, again garbage outputs and constant inputs are required in order to derive building blocks for them. Since this is conducted for each single sub-function (out of which a significant amount exists for a originally given function to be synthesized), this eventually leads to a significant amount of additional circuitry which is far beyond upper bounds (as evaluated by a corresponding study in [27]).

Being able to check whether a (sub-)function is reversible may offer the prospect of performing a decomposition such that not two arbitrary Boolean functions, but two *reversible* Boolean functions result. Since they can be realized with no additionally required outputs, significantly more compact circuits may be derived from that.

Either way, the non-availability of methods for checking the reversibility of a given function poses a major obstacle to the design of reversible circuits and systems. It prevents the application of (heuristic) embedding methods allowing to efficiently synthesize the desired function with dedicated approaches and it prevents the alternative, namely approaches based on decomposition, from generating compact circuits.

4 Theoretical Consideration

The previous section discussed why checking the reversibility of Boolean functions is of high importance. Now, we are considering the complexity of this problem. More precisely, the following decision problem is considered:

Definition 5. *Let $f \in \mathcal{B}_{n,n}$ be a Boolean function with n inputs and n outputs*[3]*. Moreover, let Rev_n denote the set of all reversible functions with n inputs and n outputs, i.e. $Rev_n = \{g \in \mathcal{B}_{n,n} \mid g \text{ is reversible}\}$. Then,* REV *is the decision problem asking whether $f \in Rev_n$.*

Note that the means of representing f is essential. For example, if f is given as a truth table, the check can be performed in linear time on the exponential input representation. In the following, we will consider the complexity with respect to the number of inputs/outputs. For this, we will prove the following:

Proposition 1. *REV is coNP-hard.*

The complexity of *REV* is shown by a reduction from the embedding problem which has been investigated in [21]. Using the notation of [21], let $f \in \mathcal{B}_{n,m}$, let $\mu(f) = max\{|f^{-1}(\{y\})| \mid y \in \mathbb{B}^m\}$ denote the number of occurrences of the most frequent output pattern, and let $l(f) = \lceil log_2 \mu(f) \rceil$ denote the minimal number of additional variables required to embed f. Then, it was shown that:

Lemma 1 (Proposition 4.3 in [21]). *For each fixed $l \geq 0$, it is coNP-hard to decide for a given $f \in \mathcal{B}_{n,m}$ whether $l(f) = l$.*

In order to apply this in our context, we have to consider the case $l = 0$ for $m = n$. Then, we immediately obtain the following:

Corollary 1. *Let $f \in \mathcal{B}_{n,n}$ ($n \geq 1$). It is coNP-hard to decide whether $l(f) = 0$.*

Proof. (adapted from [21]) The basic idea to proof this corollary is to provide a polynomial time many-one reduction from the validity problem for propositional formulas. This problem asks whether a propositional formula is a tautology and itself is known to be coNP-complete [6]. To this end, for a fixed

[3] Since functions $f\colon \mathbb{B}^n \to \mathbb{B}^m$ with $n \neq m$ are not reversible by definition, we are assuming an equal number n of inputs and outputs in the following.

propositional formula ϕ over the variables $\{x_1, \ldots, x_n\}$, we compute the function $f = (f_1, \ldots, f_n) \in \mathcal{B}_{n,n}$ by defining the component functions f_i by means of the propositional formulas

$$f_i(x_1, \ldots, x_n) := x_i \wedge \phi(x_1, \ldots, x_n) \wedge \phi(0, \ldots, 0).$$

Clearly, this computation can be performed in polynomial time. Now, as we have the equivalence

$$\begin{aligned} l(f) = 0 &\iff \lceil log_2 \mu(f) \rceil = 0 \\ &\iff max\{|f^{-1}(\{y\})| \mid y \in \mathbb{B}^n\} = 1 \\ &\iff f \text{ is injective,} \end{aligned}$$

it remains to show that the original formula ϕ is valid if and only if f is injective. Now, if ϕ is valid, we have $f_i = x_i$ and f turns out to be the identity function on $\mathbb{B}^n$ which is indeed injective. On the other hand, if ϕ is not valid there is an assignment $\tilde{x} = \{\tilde{x}_1, \ldots, \tilde{x}_n\}$ such that $\phi(\tilde{x}) = 0$. For the case $\tilde{x} = \{0, \ldots, 0\}$, all f_i are contradictions by construction and f always evaluates to 0^n. For the other case, $\tilde{x} \neq \{0, \ldots, 0\}$, we obtain $f(\tilde{x}) = 0^n = f(0, \ldots, 0)$. In both cases, f is not injective which proves the corollary. □

As we have seen in the proof, $l(f) = 0$ is equivalent to the injectivity of f. Moreover, as f has the same (finite) domain and codomain $\mathbb{B}^n$, injectivity is equivalent to bijectivity and, thus, reversibility. Consequently, we obtain the following corollary from which Proposition 1 can be implied immediately:

Corollary 2. *Let $f \in \mathcal{B}_{n,n} (n \geq 1)$. It is coNP-hard to decide whether $f \in Rev_n$.*

Note that, in order to show coNP-completeness, a counterexample for reversibility is only polynomially sized (two inputs that provide the same output) and can also be checked in polynomial time (by evaluating the function for the two inputs). This inheritance in coNP together with the coNP-hardness proves, in fact, the coNP-completeness of REV.

Knowing the complexity of the considered problem, in the remainder of this work, we focus on how to solve it as efficient as possible. To this end, two complementary approaches are introduced and discussed.

5 Checking for Reversibility Using Decision Diagrams

Graph-based representation and manipulation of (Boolean) functions became very popular in computer-aided design after the initial work on *Binary Decision Diagrams* (BDDs) by Bryant [5]. Graph-based representations of Boolean functions have – besides others – two major advantages: (1) they describe the entire function in a compact manner and (2) they allow for efficiently applying logical manipulations (e.g. computing $f \wedge g$). Accordingly, they can be utilized for the problem considered in this work. In this section, a corresponding solution based on BDDs is introduced. To this end, we first sketch the general idea before details about the implementation are provided.

5.1 General Idea

While BDDs allow for an efficient representation of Boolean functions, the characteristic of reversibility cannot directly be derived from them. Consequently, our aim is to use the various possibilities for (efficient) function manipulation in order to transform a given function $f \in \mathcal{B}_{n,n}$ in such a way that its (non)-reversibility becomes clearly evident.

To this end, we exploit the fact that the composition of a reversible function with its inverse yields the identity function. Hence, the general idea of the proposed approach is to

1. determine the inverse function f^{-1} and, afterwards,
2. check whether the composition of f and f^{-1} is equivalent to the identity mapping, i.e.
$$f^{-1} \circ f = \mathrm{id}_{\mathbb{B}^n} .$$

If the check for identity holds, the considered function is reversible. Otherwise, it can be concluded that the considered function is not reversible.

However, while checking equivalence of two functions using their BDD representation is straight-forward, there are two main issues of this procedure that are non-trivial:

(1) how to create the inverse of f (especially: what if f is irreversible?) and
(2) how to perform the composition?

These issues will be addressed in the following.

5.2 Generating the Inverse Function

Unfortunately, an inverse *function* can only be constructed if the original function is reversible. To overcome this and to develop a procedure that can also be applied to irreversible functions, we consider the graph of the function, i.e. the underlying set relation, and perform the reversibility check at the level of relations (cf. Definition 3 from Section 2). Here, an inverse can easily be created by swapping the first and the second component of each pair.

Example 2. Consider the function $f \in \mathbb{B}_{2,2}$ shown in Fig. 1(a). The corresponding set relation $F \subset \mathbb{B}^2 \times \mathbb{B}^2$ is shown in Fig. 1(b) in terms of a complete list of related pairs. The inverse relation F^{-1} which is obtained by swapping the first and second component of each pair is shown in Fig. 1(c). Apparently, the composition $F^{-1} \circ F$ (as shown in Fig. 1(d)) is clearly different from the identity relation (as shown in Fig. 1(e)), since the pattern 10 is not only related to itself, but also to 11. Consequently, f is not reversible.

However, set relations are – in contrast to BDDs – not a very efficient representation of a function. But the same concept can similarly be applied to characteristic functions (cf. Definition 4 from Section 2) and, hence function representations for which BDDs are applicable.

x_1	x_2	f_1	f_2
0	0	0	1
0	1	1	0
1	0	1	1
1	1	1	1

(a) f

$(00,01)$	$(01,00)$	$(10,10)$	$(00,00)$
$(01,10)$	$(10,01)$	$(10,11)$	$(01,01)$
$(10,11)$	$(11,10)$	$(11,10)$	$(10,10)$
$(11,11)$	$(11,11)$	$(11,11)$	$(11,11)$

(b) F (c) F^{-1} (d) $F^{-1} \circ F$ (e) id

Fig. 1. Set relations of Boolean functions

Outputs \ Inputs	00	01	10	11
00	0	0	0	0
01	1	0	0	0
10	0	1	0	0
11	0	0	1	1

(a) M_F

Outputs \ Inputs	00	01	10	11
00	0	1	0	0
01	0	0	1	0
10	0	0	0	1
11	0	0	0	1

(b) $M_{F^{-1}}$

Fig. 2. Matrix representations of characteristic functions.

Example 3. The characteristic function of a relation F, χ_F can be represented by a matrix M_F with entries $m_{ij} = \chi_F(i,j)$, i.e. the columns denote the possible input patterns and the rows denote the possible output patterns. Thus, a matrix entry is 1 if and only if the corresponding input pattern is related to the corresponding output pattern. The corresponding matrices for the relations from Figs. 1(b) and 1(c) are shown in Figs. 2(a) and 2(b), respectively. Note that the matrix for the inverse relation can be obtained by transposing the matrix M_F, i.e. $M_{F^{-1}} = M_F^T$.

Now, given χ_F, the representation for $\chi_{F^{-1}}$ can be obtained from the one for χ_F by simply swapping input and output variables and re-labelling the corresponding nodes. However, a multi-output Boolean function $f = (f_1, \ldots, f_n)$ is usually not given in terms of its characteristic function, but rather by a set (forest) of individual BDDs describing the component functions. Consequently, we have to compute χ_F as a pre-processing step in the first place. This is done by first computing the characteristic functions for the components $\chi_{F_i} = f_i \odot y_i$ (where $\odot$ denotes the XNOR-operation) and then combining these to $\chi_F = \chi_{F_1} \wedge \ldots \wedge \chi_{F_n}$.

5.3 Computing the Composition

Given the characteristic functions, χ_F and $\chi_{F^{-1}}$, we have to compute $\chi_{F^{-1} \circ F}$. In order to conduct this, we recall that the corresponding set relation is given

by $F^{-1} \circ F = \{(x, y) \mid \exists z : (x, z) \in F \wedge (z, y) \in F^{-1}\}$. Returning to the level of characteristic functions, this translates to

$$\chi_{F^{-1} \circ F}(x, y) = \exists z : \chi_F(x, z) \wedge \chi_{F^{-1}}(z, y)$$

In order to construct this function, we use an (established) logic operation called *existential quantification*.

Definition 6. *Given $f \in \mathcal{B}_{n,m}$ over variables $x_1, \ldots, x_n$, we define the (Boolean) function $(\exists x_i : f) \in \mathcal{B}_{n-1,m}$ by $(\exists x_i : f) := f_{x_i=0} \vee f_{x_i=1}$, where $f_{x_i=0}$ and $f_{x_i=1}$ denote the* co-factors *of f restricted to the respective value of x_i. That means $(\exists x_i : f)$ evaluates to true for an input assignment $(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ if and only if x_i can be chosen such that $f(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) = 1$.*

This operation can be employed for our purpose of composing χ_F and $\chi_{F^{-1}}$ as follows: we define a (Boolean) helper function

$$H(x, y, z) = \chi_F(x, z) \wedge \chi_{F^{-1}}(z, y)$$

and can then obtain $\chi_{F^{-1} \circ F}$ by existentially quantifying z:

$$\chi_{F^{-1} \circ F} = (\exists z : H).$$

After this, the characteristic function for the identity function $\mathrm{id}_{\mathbb{B}^n}$ has to be created. This can easily be done by constructing a BDD representing the function $\chi_{id} = x_1 \odot y_1 \wedge \ldots \wedge x_n \odot y_n$ (again, note that $\odot$ denotes the XNOR-operation). This states that $\chi_{id}(x, y) = 1$ if and only if $x = y$.

Finally, the resulting BDD representing $\chi_{F^{-1} \circ F}$ and the BDD representing χ_{id} have to be checked for equivalence. After constructing both BDDs this test can be performed in constant time. If both are equivalent, the considered function f is reversible. Otherwise, it has been shown that f is irreversible.

Another way to employ existential quantification for checking reversibility, as sketched in [19], is to quantify over all input variables of the characteristic function, i.e. to compute $\exists x : \chi_F$, which yields the disjunction of all output patterns. The resulting function is a tautology if and only if f is surjective/reversible. However, as existential quantification is the most expensive BDD operation used in the proposed flow, this alternative approach will not perform significantly different.

6 Checking for Reversibility Using Satisfiability Solvers

As an alternative to the BDD-based approach, we additionally propose a complementary solution to the problem considered in this work which is based on search methods. More precisely, solvers for the *Boolean satisfiability problem* (SAT problem) are utilized. In this section, we again sketch the general idea first before details on the implementation are provided.

6.1 General Idea

The SAT problem itself is simple to describe: For a given Boolean formula Φ, the SAT problem is about determining an assignment α to the variables of Φ such that $\Phi(\alpha)$ evaluates to true or to prove that no such assignment exists.

In the past years, tremendous improvements have been achieved in the development of corresponding solving engines (so-called SAT solvers). Instead of simply traversing the complete space of assignments, powerful techniques such as intelligent decision heuristics, conflict based learning schemes, and efficient implication methods e.g. through *Boolean Constraint Propagation* (BCP) are applied (see e.g. [7,9]). These techniques led to effective search procedures which can handle instances composed of thousands of variables and constraints. Furthermore, the SAT problem has been proven to be NP-complete [6], i.e. every problem in NP can be reduced in polynomial time to the SAT problem.

However, checking whether a given Boolean function is reversible does not obviously look like a satisfiability problem at a first glance: A certain property (namely unique output patterns) has to be checked *for all* possible input patterns. But this problem can easily be reformulated to a SAT problem: Instead of checking the uniqueness of all output patterns, we can negate the problem formulation and ask whether two input patterns *exist* which yield the same output pattern. This is a classical satisfiability problem.

Note that, by using this formulation, we are not considering the REV-problem (cf. Definition 5 in Section 4) anymore, but its negation (denoted by $NOTREV$ in the following). Since REV is in coNP, the complementary problem $NOTREV$ is in NP and, hence, can be solved as a SAT problem[4]. More formally, the following problem is left to be solved: Let $f \in \mathcal{B}_{n,n}$. Then, the SAT solver is asked for two patterns x, y, $x \neq y$ such that $f(x) = f(y)$ holds.

6.2 Implementation

In order to implement the proposed idea, the question "Do two input assignments $x, y \in \mathbb{B}^n$ exist so that $f(x) = f(y)$?" has to be formulated in terms of a SAT instance Φ which can be handled by corresponding solvers. Often, satisfiability solvers require the respectively given function Φ for which an assignment has to be determined in *Conjunctive Normal Form*, in *bit-vector logic*, or similar. In order to generate this formulation, the following steps have to be performed:

- *Introduce (SAT-)variables which symbolically represent all possible assignments:* A symbolic formulation for all possible assignments that have to

[4] A similar idea has been employed for equivalence checking in the domain of verification (see e.g. [1, 4]). In our context, instead of two different functions, the same function is considered twice and, instead of applying the same pattern on both functions, we apply different patterns.

	f	$=$	$(a$	$\wedge$	$b)$	$\oplus$	c
Φ_1	x_6		x_1	x_4	x_2	x_5	x_3
Φ_2	y_6		y_1	y_4	y_2	y_5	y_3

(a) Variables

$$\Phi_1 = ((x_4 = (x_1 \wedge x_2)) \wedge (x_5 = (x_4 \oplus x_3)) \wedge (x_6 = x_5))$$
$$\Phi_2 = ((y_4 = (y_1 \wedge y_2)) \wedge (y_5 = (y_4 \oplus y_3)) \wedge (y_6 = y_5))$$

(b) Constraints

$$\Phi_{obj} = ((x_1 \neq y_1) \vee (x_2 \neq y_2) \vee (x_3 \neq x_3)) \wedge (x_6 = y_6)$$

(c) Objective

Fig. 3. SAT formulation

be checked has to be created. In the problem considered here, this is accomplished by introducing a new free (Boolean) variable for each primary input, primary output, and internal signal of the considered function f. Since two different assignments are to be determined, all these variables have to be created twice (in the following distinguished between x-variables and y-variables). Fig. 3(a) exemplary provides the respectively needed variables for checking the function $f = (a \wedge b) \oplus c$.

- *Introduce constraints in order to allow for valid solutions only:* Obviously, just passing the newly created variables to a solving engine does not lead to any useful result – without further constraints, the solver would just generate arbitrary assignments. Hence, in another step, constraints must be added which restrict the solving engine to determining *valid* solutions only. In the scenario considered here, this particularly includes constraints ensuring a valid input-output mapping of the considered function, i.e. depending on the representation of f, the internal signals and, by this, the primary outputs are restricted. This has to be done for both "copies" eventually leading to a sub-instance Φ_1 and a sub-instance Φ_2. This is illustrated in Fig. 3(b) for the function from above.
- *Employ the objective*: With the formulation thus far, a symbolic representation of the evaluation of the given function f for two arbitrary assignments is available. Finally, constraints have to be employed which enforce the considered objective. For the problem considered here, this includes constraints enforcing that both inputs are not equal, while their primary outputs must be equal (leading to a sub-instance Φ_{obj}). This is illustrated in Fig. 3(c) for the function from above.

Passing the conjunction of all sub-instances, i.e. $\Phi = \Phi_1 \wedge \Phi_2 \wedge \Phi_{obj}$, to a SAT solver, a satisfying assignment of the SAT variables is derived if indeed two input assignments exist for which f yields the same output. Then, these input assignments can be obtained from the solution determined by the SAT solver

and serve as a witness for the non-reversibility. If in contrast the SAT solver proved that no satisfying assignment for the considered instance exists, it can be concluded that the function is reversible.

7 Experimental Evaluation

The approaches presented above provide non-trivial solutions to the coNP-complete problem of checking whether a given Boolean function is reversible. In order to evaluate how they eventually cope with the underlying complexity (as discussed in Section 4), both approaches have been implemented and throughly evaluated. In this section, we summarize and discuss the results of these evaluations.

7.1 Setup

The BDD-based approach from Section 5 and the SAT-based approach from Section 6 have been implemented in C/C++. To this end, *CUDD* [22] and *MiniSAT* [7] have been utilized as existing libraries for BDD construction and satisfiability solvers, respectively.

As benchmarks we considered functions from *RevLib* [26] and the well-known *LG-Synth* package. Since most of the functions are not reversible (particularly for large functions; see also the discussion of the obstacles for the design of these functions in Section 3), some irreversible functions have been made reversible using the implicit embedding from the BDD-based synthesis as proposed in [25]. More precisely, the originally given (irreversible) functions have been synthesized and, afterwards, functional descriptions have been derived from the resulting circuits using a restricted set of input/output mappings (ignoring constant inputs and garbage outputs). This way, a variety of reversible as well as irreversible functions of different sizes became available for evaluation.

Finally, all resulting functions have been processed with the implemented solutions. All experiments have been conducted on a 3 GHz Dual Opteron 2222 with 32 GB of main memory.

7.2 Results and Discussion

A selection of the obtained results are summarized in Table 1. The first columns provide the name of the respectively considered benchmarks (*Benchmark*), its number of inputs/outputs (n), as well as the desired information of whether it is reversible or not (denoted by ✓and ✗, respectively, in column *REV?*). Afterwards, the results obtained by the proposed approaches are summarized. Since both always provided the correct result on whether the function is reversible, only performance values are listed: For the BDD-based approach, the maximum number of nodes required to represent the (characteristic) function (*Nodes*) is

Table 1. Experimental evaluation

			BDD-based		SAT-based		
Benchmark	n	REV?	Nodes	Time (s)	Vars	Clses	Time (s)
9sym	27	✓	16304	0.51	9731	8269	<0.01
9sym	9	✗	961	<0.01	353	545	<0.01
cordic	52	✓	31948	2.23	27601	20993	0.08
cordic	23	✗	2849	<0.01	879	1281	<0.01
revsyn_9sym	27	✓	233020	12.97	1929	1931	<0.01
revsyn_cordic	52	✓	$> 10^8$	>3600	1941	1573	<0.01
revsyn_xor5	6	✓	214	<0.01	161	109	<0.01
xor5	6	✓	178	<0.01	161	109	<0.01
xor5	5	✗	214	<0.01	123	185	<0.01
add64_184	193	✓	12606	0.53	7403	11171	0.02
add64_184	129	✗	$> 10^8$	>3600	4693	6743	0.01
bw	87	✓	$> 10^8$	>3600	13345	9417	0.03
bw	28	✗	12480	16.23	3143	3493	<0.01
dk17_224	21	✓	7544	0.08	1449	2077	<0.01
in0_235	26	✓	26246	1.15	6427	10019	<0.01
in0_235	15	✗	361	<0.01	299	271	<0.01

given, while, for the SAT-based approach, the number of variables (*Vars*) and clauses (*Clses*) required to formulate the corresponding satisfiability problem is given. For both approaches additionally the required run-time (*Time*, in CPU seconds) is provided.

The results clearly show that both approaches are successful in efficiently solving the considered problem. Considering the coNP-hardness of the task, functions composed of more than 100 variables (constituting one of the largest functions currently considered in the design of reversible circuits and systems) can be handled quite efficiently.

Comparing both (complementary) solutions against each other, it is obvious that the SAT-based approach performs significantly better than the BDD-based approach. This can be explained by the "memory explosion" of the BDD representation. In fact, BDDs are known for their efficient representation of Boolean functions, but eventually require exponential space in the worst case. In the scenario considered here, this worst case is often approached because characteristic functions are considered. Building these often requires the BDD package to fold up the entire functionality before reductions e.g. due to sharing can be exploited. This obviously harms the efficiency of the approach.

In contrast, the SAT-based approach can handle the respective search space in a more efficient fashion. Even for larger functions, always negligible run-time

is required. Hence, the SAT-based solution clearly constitutes itself as a very efficient solution for checking the reversibility of a given function.

8 Conclusions

In this work, we considered how to check whether a given function is reversible. Although never explicitly considered thus far, the absence of corresponding solutions constitutes a major obstacle in the design of reversible circuits and systems. We proved that the underlying problem is coNP-complete and proposed two complementary approaches addressing it – one based on decision diagrams and another exploiting satisfiability solvers. The experimental evaluation showed that, despite the complexity, both solutions can handle the problem. In fact, the SAT-based solution is even capable of solving the task in negligible run-time even for some of the largest functions considered in the design of reversible circuits and systems thus far.

Acknowledgments

This work has partially been supported by the EU COST Action IC1405.

References

1. Amarú, L., Gaillardon, P.E., Wille, R., De Micheli, G.: Exploiting inherent characteristics of reversible circuits for faster combinational equivalence checking. In: Design, Automation and Test in Europe (2016), to appear
2. Athas, W., Svensson, L.: Reversible logic issues in adiabatic CMOS. In: Proc. Workshop on Physics and Computation, 1994. PhysComp '94. pp. 111–118 (1994)
3. Berut, A., Arakelyan, A., Petrosyan, A., Ciliberto, S., Dillenschneider, R., Lutz, E.: Experimental verification of Landauer's principle linking information and thermodynamics. Nature 483, 187–189 (2012)
4. Brand, D.: Verification of large synthesized designs. In: Int'l Conf. on CAD. pp. 534–537 (1993)
5. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. on Comp. 35(8), 677–691 (1986)
6. Cook, S.: The complexity of theorem-proving procedures. In: Symp. on Theory of Computing. pp. 151–158. ACM (1971), http://doi.acm.org/10.1145/800157.805047
7. Eén, N., Sörensson, N.: An extensible SAT solver. In: SAT 2003. LNCS, vol. 2919, pp. 502–518 (2004)
8. Fazel, K., Thornton, M., Rice, J.: ESOP-based Toffoli gate cascade generation. In: IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PacRim 2007). pp. 206 –209. IEEE (2007)
9. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Int'l Joint Conference on Artificial Intelligence. pp. 386–392 (2007)

10. Große, D., Wille, R., Dueck, G.W., Drechsler, R.: Exact multiple control Toffoli network synthesis with SAT techniques. IEEE Trans. on CAD 28(5), 703–715 (2009)
11. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Theory of computing. pp. 212–219 (1996)
12. Houri, S., Valentian, A., Fanet, H.: Comparing CMOS-based and NEMS-based adiabatic logic circuits. In: Dueck, G., Miller, D. (eds.) Reversible Computation, LNCS, vol. 7948, pp. 36–45. Springer (2013)
13. Merkle, R.C.: Reversible electronic logic using switches. Nanotechnology 4(1), 21–40 (1993)
14. Miller, D.M., Maslov, D., Dueck, G.W.: A transformation based algorithm for reversible logic synthesis. In: Design Automation Conf. pp. 318–323 (2003)
15. Nielsen, M., Chuang, I.: Quantum Computation and Quantum Information. Cambridge Univ. Press (2000)
16. Ren, J., Semenov, V., Polyakov, Y., Averin, D., Tsai, J.S.: Progress towards reversible computing with nSQUID arrays. IEEE Transactions on Applied Superconductivity 19(3), 961–967 (2009)
17. Saeedi, M., Zamani, M.S., Sedighi, M., Sasanian, Z.: Synthesis of reversible circuit using cycle-based approach. J. Emerg. Technol. Comput. Syst. 6(4), 13:1–13:26 (2010)
18. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. Foundations of Computer Science pp. 124–134 (1994)
19. Soeken, M., Tague, L., Dueck, G.W., Drechsler, R.: Ancilla-free synthesis of large reversible functions using binary decision diagrams. J. Symb. Comput. 73, 1–26 (2016), http://dx.doi.org/10.1016/j.jsc.2015.03.002
20. Soeken, M., Wille, R., Hilken, C., Przigoda, N., Drechsler, R.: Synthesis of reversible circuits with minimal lines for large functions. In: ASP Design Automation Conf. pp. 85–92 (2012)
21. Soeken, M., Wille, R., Keszocze, O., Miller, D.M., Drechsler, R.: Embedding of large boolean functions for reversible logic. J. Emerg. Technol. Comput. Syst. 12(4), 41:1–41:26 (2015), http://doi.acm.org/10.1145/2786982
22. Somenzi, F.: Efficient manipulation of decision diagrams. Software Tools for Technology Transfer 3(2), 171–181 (2001)
23. Thomsen, M.K.: Describing and optimising reversible logic using a functional language. In: Implementation and Application of Functional Languages, pp. 148–163. Springer (2012)
24. Toffoli, T.: Reversible computing. In: de Bakker, W., van Leeuwen, J. (eds.) Automata, Languages and Programming, LNCS, vol. 85, pp. 632–644. Springer (1980)
25. Wille, R., Drechsler, R.: BDD-based synthesis of reversible logic for large functions. In: Design Automation Conf. pp. 270–275 (2009)
26. Wille, R., Große, D., Teuber, L., Dueck, G.W., Drechsler, R.: RevLib: an online resource for reversible functions and reversible circuits. In: Int'l Symp. on Multi-Valued Logic. pp. 220–225 (2008), RevLib is available at http://www.revlib.org
27. Wille, R., Keszöcze, O., Drechsler, R.: Determining the minimal number of lines for large reversible circuits. In: Design, Automation and Test in Europe. pp. 1204–1207. IEEE (2011)
28. Wille, R., Drechsler, R., Osewold, C., Garcia-Ortiz, A.: Automatic design of low-power encoders using reversible circuit synthesis. In: Design, Automation and Test in Europe. pp. 1036–1041. IEEE (2012)