2016 IEEE 46th International Symposium on Multiple-Valued Logic

Re-writing HDL Descriptions for Line-aware Synthesis of Reversible Circuits

Zaid Alwardi*† Robert Wille‡§ Rolf Drechsler*§

*Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
†Collage of Engineering, Al-Mustansiriya University, Baghdad, Iraq
‡Institute for Integrated Circuits, Johannes Kepler University Linz, 4040 Linz, Austria
§Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{alwardi,drechsle}@informatik.uni-bremen.de robert.wille@jku.at

***Abstract*—Reversible computing is a promising research field due to its applications in several emerging technologies. Accordingly, several approaches for the design of reversible circuits have been introduced – including solutions realizing functionality provided in terms of hardware description languages. Their main drawback is, however, that they require a substantial amount of additional circuit lines. While some solutions addressing this problem have been proposed in the past, the contribution of the respectively given HDL code to this drawback has hardly been considered yet. In this work, we are considering this problem from this angle: Observations have been conducted which, eventually, led to a set of re-writing rules for a line-aware synthesis of reversible circuits from HDL descriptions. Case studies show the benefits of these rules – in total, substantial reductions in the number of circuit lines have been observed.**

I. Introduction

Reversible computing is an interesting research field due to its promising applications in many emerging technologies, such as quantum computing [1], [2], low power design [3], [4], [5], or the design of encoders/decoders [6]. Accordingly, researchers heavily investigated how to synthesize a given function in terms of a reversible circuit while, at the same time, keeping the costs of the resulting netlists small. Usually, costs are measured in terms of gate costs which, depending on the design objective, e.g. represents quantum costs [7], [8] or transistor costs [4]. Besides that, it is tried to keep the number of circuit signals (circuit lines) minimal. This is mainly motivated by the fact that, in the domain of quantum computation, each circuit line has to be represented by a qubit – usually considered a very limited resource [2].

First synthesis approaches e.g. based on permutations [9], transformations [10], positive-polarity Reed-Muller spectra [11], Boolean satisfiability [12], and others accordingly aimed for keeping the number of circuit lines minimal. Since this always required a truth table-like description of the function to be synthesized, this frequently led to an exponential explosion and, hence, an applicability for rather small functions only. As an alternative, hierarchical approaches have been introduced which decompose a given function to be synthesized into smaller sub-functions and, hence, apply a *divide-and-conquer* scheme. To this end, binary decision diagrams [13] or two-level function descriptions such as ESOPs [14] have been utilized. Moreover, even initial approaches based on *Hardware Description Languages* (HDL) have been proposed [15], [16]. These solution indeed allow for the synthesis of large functionality. But at the same time, they usually require a significant amount of additional circuit lines – a serious drawback.

In this work, we aim to improve on this drawback for HDL-based synthesis of reversible logic. To this end, we focus on the HDL SyReC as well its corresponding synthesis scheme introduced in [15]. SyReC is capable of realizing large functionality in terms of reversible circuits, but suffers from a seriously large number of additionally required circuit lines. Although significant improvements on that have recently been achieved by employing a so-called garbage-free synthesis (see [17]) or a dedicated consideration of the synthesis of expressions (see [18]), the number of additionally required circuit lines is still far beyond the known theoretical bounds [19]. Also the application of post-synthesis optimization techniques such as proposed in [20], [21] does not reduce this number of additional lines to a moderate level.

As an alternative, we now try to additionally address this issue from a different angle: Instead of optimizing the synthesis scheme itself, we consider how the originally given HDL description affects the number of additionally required circuit lines. To this end, we analyze statements that generate additional lines and replaced them with equivalent statements causing less additional lines. Based on our observations, we derived re-writing rules that allow for a simple pre-synthesis optimization of a given HDL description. Case studies confirm that the fashion in which HDL code is provided, has a significant effect on the total number of additional lines. In total, substantial reductions can been observed.

The reminder of this work is structured as follows: The next section reviews the basics on reversible circuits, the reversible HDL SyReC, as well as the corresponding synthesis scheme. Afterwards, the motivation of this work is provided, i.e. it is shown how HDL descriptions may affect the number of circuit lines. From this, corresponding rules are derived which are presented and illustrated in Section IV. Their application is eventually discussed and evaluated in Sections V and VI, respectively, before the paper is concluded in Section VII.

0195-623X/16 $31.00 © 2016 IEEE
DOI 10.1109/ISMVL.2016.36

II. Background

This section provides a brief background on reversible circuits, the reversible HDL SyReC, as well as the corresponding synthesis scheme which is required to keep the paper self-contained.

A. Reversible Circuits

Reversible circuits realize functions $f : \mathbb{B}^n \rightarrow \mathbb{B}^n$ with a unique input/output mapping, i.e. bijections. A reversible circuit $G = g_1 \dots g_d$ is composed as a cascade of reversible gates g_i [2]. The inverse of G (representing the function f^{-1} and denoted by G^{-1}) can be obtained by cascading $g_d^{-1} g_{d-1}^{-1} \cdots g_1^{-1}$, where g_i^{-1} is the inverse gate of g_i. Since the gates considered in this paper are self-inverse (i.e. $g_i = g_i^{-1}$), G^{-1} can simply be obtained by reversing the order of the gates of G.

For a set of signals $X = \{x_1, \dots, x_n\}$, a *reversible gate* has the form $g(C, T)$, where $C = \{x_{i_1}, \dots, x_{i_k}\} \subset X$ is the set of *control lines* and $T = \{x_{j_1}, \dots, x_{j_l}\} \subseteq X$ with $C \cap T = \emptyset$ is the non-empty set of *target lines*. The gate operation is applied to the target lines if, and only if, all control lines meet the required control conditions. Control lines and unconnected lines always pass through the gate unaltered.

In the literature, several types of reversible gates have been introduced. Usually, circuits realized by *Toffoli gates* and *Fredkin gates* are considered. A Toffoli gate has a single target line x_j and uniquely maps the input $(x_1, x_2, \dots, x_j, \dots, x_n)$ to the output $(x_1, x_2, \dots, x_{i_1} x_{i_2} \cdots x_{i_k} \oplus x_j, \dots, x_n)$. That is, a Toffoli gate inverts the target line if, and only if, all control lines are assigned the logic value 1. An Toffoli gate with no (one) control line is also denoted as *NOT* (*CNOT*) gate. A Fredkin gate has two target lines x_{j_1} and x_{j_2} and interchanges their values if, and only if, the conjunction of all control lines evaluates to 1.

By definition, reversible circuits can only realize reversible functions. In order to realize non-reversible functions, *additional circuit lines* with constant inputs and garbage outputs (i.e. don't care outputs) are applied (see e.g. [11], [19]). Furthermore, additional circuit lines are also used frequently in hierarchical synthesis approaches (e.g. [14], [13], [15]).

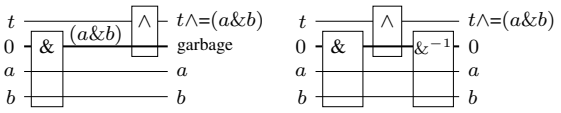
B. The Reversible HDL SyReC

SyReC has been proposed as a reversible HDL in [15][1]. A SyReC program is composed of modules in which the desired I/O functionality is specified. Fig. 1 shows such a module. The first line defines its name (*example*) as well as the involved signals with its type ($in/out/inout$), name (a, b, x, f), and bit-width (32). Line 2 is an explicit declaration of an internal signal (*wire*) which is supposed to be used only within this module. The remaining lines define the statements to be executed on these signals. More precisely, the signals on the *left-hand side* (LHS) (i.e. t,x,f) are updated according to a respectively applied *reversible* operation ($\hat{}=$, $+=$) and the expression on the *right-hand side* (RHS).

[1]SyReC is partially reviewed within this paper. For more details, we refer to [15] as well as to the detailed documentation provided at the RevLib benchmark webpage [22].

```
1 module example(in a(32), in b(32), inout x(32), out f(32))
2 wire t(32)
3 t ^= (a & b)
4 x += (((a * b) + (a / b)) - ((a + b) / t))
5 f ^= (((t + b) ^x) * (a - b))
```

Fig. 1. Simple SyReC program



(a) Original Synthesis Scheme (b) Garbage-free Synthesis Scheme

Fig. 2. SyReC synthesis for t $\oplus=$ (a$\odot$b)

In SyReC, only the reversible operations XOR, increase, and decrease, i.e. $\{\hat{}=, +=, -=\}$, are allowed for signal updates. An expression $\mathbb{E}$ can either be a simple signal identifier or a binary-expression in the form $(\mathbb{E}_{left} \odot \mathbb{E}_{right})$, where $\mathbb{E}_{left}$ and $\mathbb{E}_{right}$ are again (sub-)expression. Binary-operations $\odot$ are not restricted to reversible ones, but can assume a wide range of binary-operators including arithmetic, bit-wise, logical, relational and shift operators. Reversibility is guaranteed by asserting that signals on the LHS must not occur on the right-hand side. Then, signal values are updated only through reversible operations while the values of the signal on the RHS always keep their value.

C. SyReC-based Synthesis

SyReC programs allow for the definition of complex reversible functionality. In order to realize this, a synthesis scheme as sketched in Fig. 1 for the statement t $\hat{}=$ (a & b) (Line 3 of Fig. 1) is applied. Here, the target signal t is updated with the result of the binary expression (a & b) using the reversible update operator $\hat{}=$.

Fig. 2(a) shows the resulting circuit structure. First, the binary-operator & is realized[2]. Since & is non-reversible, constant-input lines are added to the circuit to preserve reversibility [15]. This eventually results in a garbage output. Afterwards, the result of this operation is applied to the respective reversible update $\hat{}=$, which eventually realizes this statement (see Fig. 2(a)).

Following this scheme, a new additional circuit lines are added with each statement – resulting in a large number of lines. These accumulated lines are the main drawback of the SyReC-based synthesis approach. Hence, an alternative has been proposed in [17]. Here, a garbage-free synthesis is proposed which is sketched in Fig. 2(b). The first steps are identical to the previous approach. But after the reversible update has been conducted, a third block is added which inverses the computation of the first block. Because of that, the (intermediate) result of the binary operation (which is not needed anymore) is re-computed to its original value 0, allowing the same line to be reused by other statements. This

[2]Operators are defined for various bit-widths. Both signals must have the same bit-width.

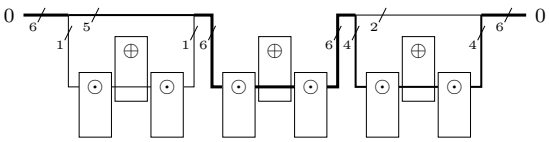


Fig. 3. Effect of expression size

eventually leads to a total number of additionally required circuit lines, which is bounded by the largest bit-width an expression in the given SyReC code requires. An example illustrates the idea:

Example 1. *Recall the SyReC code in Fig. 1 and its three statements in lines 3, 4, and 5. The garbage-free synthesis results for this program is sketched in Fig. 3. The three statements have an expressions with 1, 6, and 4 operations, respectively. Accordingly, 1, 6, and 4 additional circuit signals (each with the corresponding bit-width) are required. Since previously used signals can be re-used, the overall circuit eventually requires* $\max(1, 6, 4) = 6$ *additional circuit signals.*

As a drawback, garbage-free synthesis almost doubles gate costs. But instead, it significantly reduces the number of circuit lines (as also experimentally confirmed in [17]). Nevertheless, the number of additionally required circuit lines remain high for many HDL programs.

III. Motivation

Garbage-free synthesis as reviewed above already leads to substantial reductions in the number of additionally required circuit signals. But as shown in Fig. 3, circuits with a rather high number of additional lines still result. Moreover, it shows that this number heavily relies on the size of the expressions (i.e. its number of operations) to be synthesized. This motivates a consideration from a different angle: Instead of optimizing the synthesis approach itself, also the fashion in which reversible HDL descriptions are given provides potential for improvement. In this section, we illustrate observations which can be exploited for this purpose. This eventually results in re-writing rules, which are proposed and applied in Section IV and Section V.

A. Split Expressions

As reviewed in Section II-C, garbage-free synthesis adds a number of lines equal to the number of binary-operators in the largest expression of a given HDL description. Consequently, reducing the number of operations in this particular expression leads to a reduction in the number of lines of the overall circuit. The example illustrates the idea:

Example 2. *Consider the statement in Line 4 of Fig. 1:*

$$\mathrm{x} \mathrel{+}= (((\mathrm{a} * \mathrm{b}) + (\mathrm{a} \;/\; \mathrm{b})) - ((\mathrm{a} + \mathrm{b}) \;/\; \mathrm{t}))$$

This statement has the largest expression in the module that contains 6 binary-operators, which means that 6 lines will implicitly be added to realize the circuit. This expression is composed of two sub-expression operands for the operator -: The left operand is $((\mathrm{a} * \mathrm{b}) + (\mathrm{a} \;/\; \mathrm{b}))$ *with 3 operators and the right operand is* $((\mathrm{a} + \mathrm{b}) \;/\; \mathrm{t})$ *with only two. The exact value of the target signal* x *can calculated with, even shorter expressions, as follows:*

$$\mathrm{x} \mathrel{+}= (\mathrm{a} * \mathrm{b})$$
$$\mathrm{x} \mathrel{+}= (\mathrm{a} \;/\; \mathrm{b})$$
$$\mathrm{x} \mathrel{-}= ((\mathrm{a} + \mathrm{b}) \;/\; \mathrm{t}))$$

Although equivalent to the original statement, the new code reduces the size of this expression from 6 to only 2. As a consequence, only two additional circuit signals are needed anymore for this statement. Considering the overall structure as shown in Fig. 3, this allows for realizing the circuit with 4 additional circuit signals only (due to the new bound set by the statement in line 5).

B. Internal Wires

Unfortunately, splitting expressions as discussed above is only possible if the respective top-level operation of the expression can be realized by a reversible update. In other cases, no reversible computation is guaranteed anymore. However, this problem can be circumvented if wires are applied. They introduce a constant signal to the circuit and, since $(0 \;\hat{}\; \mathrm{E}) = \mathrm{E}$, allow to copy the result of an expression E. Although wires also introduce additional circuit lines in the worst case, this may prove beneficial as illustrated next:

Example 3. *Consider the statement in Line 5 from Fig. 1:*

$$\mathrm{f} \mathrel{\hat{}}= (((\mathrm{t} + \mathrm{b}) \;\hat{}\; \mathrm{x}) * (\mathrm{a} - \mathrm{b}))$$

This statement assigns the result of an expression with 4 binary-operators to the target signal. The root binary operator of this expression is not reversible ($*$*), i.e. a splitting as done before is not possible directly. However, a new wire (*w*) can be declared for this purpose. Then, the intermediated result can be buffered and applied to non-reversible operations. The value of the target signal (*f*) is then computed as follows:*

$$\mathrm{w} \mathrel{\hat{}}= ((\mathrm{t} + \mathrm{b}) \;\hat{}\mathrm{x})$$
$$\mathrm{f} \mathrel{-}= (\mathrm{w} * (\mathrm{a} - \mathrm{b}))$$

*In this case the new statements have expressions with only 2 operators each. As the wire signal (*w*) requires an "own" circuit signal, the total number of signals for the overall circuit is reduced to 3 (compared to the original 6 and the 4 obtained by splitting expressions only).*

Of course, the internal signal should be re-computed as well in order to allow for re-using the corresponding circuit signal for another statement. For this purpose, the following inverse statements can be applied:

1) $\mathrm{S} \mathrel{\hat{}}= \mathrm{E}$ is a self inverse statement as long as $((\mathrm{S} \;\hat{}\; \mathrm{E}) \;\hat{}\; \mathrm{E}) = \mathrm{S}$, which means that the initial value of the target signal S is re-computed by repeating this statement twice[3].
2) The inverse statement of $\mathrm{S} \mathrel{+}= \mathrm{E}$ is $\mathrm{S} \mathrel{-}= \mathrm{E}$.

This leads to an undo for internal wires as shown in the following example.

[3]The value of the RHS is assumed to be equal (unchaged) in both statements.

Example 4. *Consider again the code from Example3. Adding the following code re-computes signal* w *to its initial value* 0*.*

```
wire w
w ^= ((t + b) ^x)
f -= (w * (a - b))
w ^= ((t + b) ^x)                    // re-compute w
```

As a consequence, the corresponding circuit signals can be re-used for another statement.

IV. Derived Re-writing Rules

The observations from the previous section show that the respectively given HDL code may have a significant impact to the overall number of additionally required circuit lines. Hence, optimizing an HDL code in this regard provides a promising approach to correspondingly optimize a resulting circuit. To this end, several re-writing rules have been obtained which are introduced in this section. They application and evaluation is, afterwards, discussed in Section V and Section VI, respectively.

In order to describe the re-writing rules, the following notation is applied:

- S denotes a general explicit signal identifier. Other upper case letters may also be used for the same purpose if needed.
- W denotes an internal wire signal identifier.
- $\odot$ denotes a binary operator, such as arithmetic, logical, relational, shift, and bit-wise operators.
- $\oplus$ denotes a reversible binary operator, which is a special case of $\odot$, i.e. $\{\hat{}, +, -\}$.
- $\mathbb{E}$ denotes an expression that can be a simple signal identifier or has the general form $(\mathbb{E}_{left} \odot \mathbb{E}_{right})$ where the operands $\mathbb{E}_{left}$ and $\mathbb{E}_{right}$ are expressions.
- **S**$\mathbb{S}$ denotes a statement to update the value of a signal S with the value of the binary expression $\mathbb{E}$ by using the reversible operator $\oplus=$, i.e. **S**$\mathbb{S}$ denotes $S \oplus= \mathbb{E}$.
- **S**$\mathbb{S}^{-1}$ denotes the inverse statement of **S**$\mathbb{S}$ with the inverse update operator $\oplus^{-1}=$ of the original operator $\oplus=$, where $(- \equiv +^{-1})$ and $\hat{}$ is self-inverse.

Using this notation, the following rules for re-writing HDL statements are proposed:

R1: A signal update statement **S**$\mathbb{S}$ of the form

$$S \oplus= (\mathbb{E}_{left} \oplus \mathbb{E}right)$$

can be re-written to

$S \oplus= \mathbb{E}_{left}$
$S \oplus= \mathbb{E}_{right}$

if $\oplus=$ and $\oplus$ are both either $\hat{}$, +, or -.

R2: A statement $S \mathrel{+}= \mathbb{E}$ can be re-written to $S \mathrel{\hat{}}= \mathbb{E}$, if $(S = 0)$ (since $(0 + E)$ = $(0 \hat{} E)$ = E).

R3 If $\bigoplus$ and $\oplus$ are two reversible operators and S is a signal that appears just once within the combined expression within the statement **S**$\mathbb{S}$ such as

$$T \bigoplus= (\ \ldots\ (S \oplus \mathbb{E})\ \ldots\ )$$

then **S**$\mathbb{S}$ can be re-written to

$S \oplus= \mathbb{E}$
$T \bigoplus= (\ \ldots\ S\ \ldots\ )$
$S \oplus^{-1}= \mathbb{E}$

R4: Any statement **S**$\mathbb{S}$ with any arbitrary binary expression $\mathbb{E}$ on the RHS can be re-written to two sub-expressions by assigning one of its operands into a wire $(W = 0)$ as follows:

$W \mathrel{\hat{}}= \mathbb{E}_{left}$
$S \oplus= (W \odot \mathbb{E}_{right})$

The value of W can be re-computed to 0 using the inverse statement $W \mathrel{\hat{}}= \mathbb{E}_{left}$ if the wire is needed for reuse in the rest of the program.

R5: An internal wire W is redundant and can be substituted by the signal S, if $(S = 0)$ and W is last accessed to initialize S with statement: $S \mathrel{\hat{}}= W$.

V. Application of the Re-writing Rules

The re-writing rules described above can be applied to optimize a given HDL code with respect to additionally required lines needed by the synthesis. In this section, the application of the respective rules is discussed and illustrated by examples.

Applying the rule *R1* is useful with respect to many issues. First of all, when a binary operator is substituted by a reversible update operator, an additionally required circuit signal is saved. But besides that, the circuit $G_{S \oplus= \mathbb{E}}$ is also synthesized with lower cost as compared to $G_{(S \oplus \mathbb{E})}$ (since splitting the expression also reduces the size of it). Hence, two improvements are accomplished at once. A corresponding example illustrating this has already been provided by Example 2. Note that small re-writings have additionally to be considered, e.g. when the decrease operator is applied, e.g. $S-=(\mathbb{E}_{left} + \mathbb{E}_{right})$ has to be mapped to $S-=\mathbb{E}_{left}; S-=\mathbb{E}_{right}$ (changing the addition from the original expression to a subtraction).

The rule *R2* is conditionally applicable for 0-valued signals, such as (wire/out) signals before they have been initialized or after they have been re-computed. The statement $G_{S \wedge= \mathbb{E}}$ is synthesized with lower cost than the statement $G_{S+= \mathbb{E}}$, which means that the rule can be used to reduce the cost. Besides that, further simplifications may be achieved in some cases, if *R2* is combined with *R1* as illustrated in the following example.

Example 5. R1 *is not applicable for statement:* $S \mathrel{\hat{}}= (\mathbb{E}_{left} + \mathbb{E}_{right})$*, because the two operators are not equivalent. However, if* (S=0)*, then* R2 *along with* R1 *can be applied to split the statement into* $S \mathrel{\hat{}}= \mathbb{E}_{left}$; $S+= \mathbb{E}_{right}$.

Rule *R3* is illustrated by means of the following example:

Example 6. R3 *can be applied in order to re-write the statement*

$$\text{y } \hat{}= ((\text{a} * ((\text{b} * \text{c}) + \text{x}) / \text{d})$$

to

```
1 x += (b * c)
2 y ^= ((a * x)) / d)
3 x -= (b * c)                       // re-compute x
```

It is not possible to reduce the size of this expression using R1 or R2, because the root operator is not reversible (/). Instead, R3 is applicable for this statement to update signal x

because it appears only once in the expression. The value of x *is increased by the value of the sub-expression* (b * c) *in the first line. The signal* x *is updated to be* (b * c) + x)*, then used to calculate the required expression in the second line. Finally the original value of* x *is re-computed in the last line.*

Rule *R4* is illustrated by means of the following example:

Example 7. *The following statement*
y ^= (((a * b) / (a + b)) * ((c / d) * (c - d)))
can be re-written to

```
1 wire L
2 L ^= ((a * b) / (a + b))
3 f ^= (L * ((c / d) * (c - d)))
4 L ^= ((a * b) / (a + b))          //re-compute L
```

In this example, the original expression contains 7 operators, while the equivalent program reduces them to only 4 in the largest expression, plus the declared wire L, i.e. the total number of required signals is reduced from 7 to 5.

Hence, R4 can be applied to assign the result of a sub-expression to a signal. If this is done for the sub-expression of a statement with a larger number of operations, a reduction in the number of required circuit lines is achieved.

Note that the re-computations of wires as conducted in rules R3 and R4 increase the gate costs. Hence, this constitutes a trade-off between circuit lines and gate costs as e.g. also observed in [21]. Of course, these re-computations can be omitted if the original signal values are not required for further statements anymore

The rule *R5* can be used, in some cases, to remove wires as illustrated by the following example:

Example 8. *Consider the following code:*

```
1 module rule5(in a(8) in b(8), inout x(8), out s(8))
2 wire w(8)
3 w ^= (a * b)
4 y += (w / 2)
5 s ^= w
```

Signal s *is declared as an output signal with initial value* s=0*. Line 5 sets* s *to the value of the internal wire* w*. Hence, R5 can be applied to remove the wire* w *as follows:*

*(1) Erase the declaration statement of the wire (*w*), i.e. line 2.*

(2) Erase the statement initializing signal s*, i.e. line 5.*

(3) Substitute the wire w *with the signal* s*, as follows:*

```
1 module rule5(in a(8), in b(8), inout x(8), out s(8))
2 s ^= (a * b)
3 x += (s / 2)
```

Removing w *reduces one signal. Additionally, removing line 5 decreases the gate cost.*

Applying these rules result in programs with larger number of statements but with shorter expressions. These programs are optimized for garbage-free synthesis to produce circuits with less additionally required circuit lines. On the downside, this resulting programming style is usually less readable compared to the originally given code.

Example 9. *The following SyReC program represent the a re-written equivalent to the program in Fig.1:*

```
1 module example(in a(32), in b(32), in x(32), out f(32))
2 wire t(32)
3 t ^= (a & b)
4 x += (a * b)
5 x += (a / b)
6 a += b
7 x -= (a / t)
8 a -= b
9 t += b
10 t ^= x
11 a -= b
12 f ^= (t * a)
```

Expressions in this code contain one or no operator at all. Consequently, only one line is implicitly added to the circuit, in comparison to the original code which adds 6 lines.

VI. Experimental Evaluation

The effect of the proposed re-writing rules to the number of lines as well as the costs of the resulting circuits has been evaluated. To this end, we considered HDL descriptions (taken from *RevLib* [22] and provided in SyReC notation) and optimized them using the rules from above. Afterwards, we realized the resulting HDL descriptions using the synthesis approach proposed in [15] together with its optimizations as proposed in [17]. In this section, the obtained results are summarized and discussed.

Table I provides a numerical overview of the obtained results[4]. The first column denotes the name of the HDL description as well as the total number inputs/outputs which have to be derived according to the HDL code. Afterwards, the number of actually derived circuit lines and the resulting quantum costs are reported for the synthesis approach originally proposed in [15], the garbage-free synthesis approach proposed in [17], and the results obtained by applying the proposed re-writing rules and, afterwards, synthesizing the resulting HDL code with the garbage-free synthesis approach. For the number of circuit lines, we did not only list the total number of lines but also the percentage difference of them with respect to the number of input/output lines (i.e. 50% in this column states that, with respect to the input/output lines which are needed anyway, 50% additional lines are introduced because of the synthesis scheme). Finally, columns denoted by *Diff.* list the percentage difference of the results obtained by the proposed method compared to the results of the state-of-the-art proposed in [17] (i.e. 50% in this column states that the proposed scheme yields a circuit which has 50% of the additional lines/quantum costs than the circuit derived from [17]).

The numbers clearly show that the proposed re-writing rules lead to substantial reductions in the number of additionally required circuit lines for almost all considered HDL descriptions. In some cases, the number can be reduced to just a fraction of what was needed before. Moreover, all these improvements

[4]The first row shows the results of the HDL description from Fig. 1 which has not been taken from *RevLib*.

TABLE I
EXPERIMENTAL EVALUATION

Benchmark		Number of Circuit Lines							Quantum Cost			
		Orignal [15]		Garb.free [17]		Proposed		Diff.	Orignal [15]	Garb.free [17]	Proposed	Diff.
name	L_{io}	L_t	$L_{ad}\%$	L_t	$L_{ad}\%$	L_t	$L_{ad}\%$	wrt. [17]				wrt. [17]
Fig. 1	128	512	300%	352	175%	192	50%	29%	2460912	4921296	4919024	100%
Acc	160	352	120%	288	80%	256	60%	75%	2435006	4569948	4869820	107%
Var10	96	376	292%	208	117%	128	33%	29%	44626	87168	85888	99%
Fact4	48	152	217%	104	117%	72	50%	43%	4841	9640	6956	72%
Fact8	80	280	250%	168	110%	120	50%	45%	9665	19264	16060	83%
Poly4	56	136	143%	88	57%	80	43%	75%	10836	21248	17384	82%
Poly4-loop	56	160	186%	104	86%	80	43%	50%	9153	17856	17384	97%
Poly8	88	376	327%	152	73%	128	45%	63%	38144	75488	41300	55%
Poly8-loop	88	288	227%	168	91%	128	45%	50%	18289	35696	41300	116%
alu-233	98	229	134%	133	36%	133	36%	99%	1704912	3407588	3401244	100%
alu-flat-239	50	118	136%	67	34%	67	34%	100%	181662	363012	361172	99%
cpu-alu-16bit-242	55	404	635%	142	158%	142	158%	100%	662531	1281717	1229192	96%
cpu-alu-32bit-243	103	756	634%	254	147%	252	145%	99%	2235491	4381653	4222827	96%
cpu-control-unit-244	232	391	69%	290	25%	272	17%	69%	40433	80142	79093	99%
lu-238	98	197	101%	133	36%	132	35%	97%	6557	10462	7966	76%
simple-alu-234	98	229	134%	133	36%	133	36%	100%	144791	287346	278504	97%
varops-250	96	224	133%	192	100%	192	100%	100%	2801	4176	3120	75%

L_{io}: Total number of input/output lines according to the HDL descriptions L_t: Total number of resulting circuit lines
$L_{ad}\%$: Percentage difference in the number of resulting circuit lines with respect to the number of input/output lines

can be obtained while, at the same time, hardly increasing the resulting quantum costs. In fact, in almost all cases the quantum costs remain the same or even get reduced. This is particularly remarkable since previous studies frequently showed a trade-off between the number of circuit lines and the quantum costs [21].

Overall, the evaluation shows that re-writing HDL description for reversible circuit synthesis is a promising direction to get more compact results than the current state-of-the-art.

VII. CONCLUSIONS

In this work, an improvement for the synthesis of HDL descriptions for reversible circuits has been proposed. Here, the substantial amount of additionally required circuit lines constitutes a major problem. Instead of further optimizing the synthesis scheme itself, we proposed to change the respectively given HDL code instead. This motivated an observation which eventually resulted in the derivation of re-writing rules. Case studies and experimental evaluations showed that applying the re-writing rules yields substantial improvement with respect to the number of additionally required circuit lines. At the same time, the costs of the circuit remain stable or are even also improved.

ACKNOWLEDGMENTS

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

REFERENCES

[1] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," *Foundations of Computer Science*, pp. 124–134, 1994.

[2] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.

[3] V. V. Zhirnov, R. K. Cavin, J. A. Hutchby, and G. I. Bourianoff, "Limits to binary logic switch scaling – a gedanken model," *Proc. of the IEEE*, vol. 91, no. 11, pp. 1934–1939, 2003.

[4] B. Desoete and A. D. Vos, "A reversible carry-look-ahead adder using control gates," *INTEGRATION, the VLSI Jour.*, vol. 33, no. 1-2, pp. 89–104, 2002.

[5] A. Berut, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, and E. Lutz, "Experimental verification of Landauer's principle linking information and thermodynamics," *Nature*, vol. 483, pp. 187–189, 2012.

[6] R. Wille, R. Drechsler, C. Oswald, and A. Garcia-Ortiz, "Automatic design of low-power encoders using reversible circuit synthesis," in *Design, Automation and Test in Europe*, 2012, pp. 1036–1041.

[7] A. Barenco, C. H. Bennett, R. Cleve, D. DiVinchenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter, "Elementary gates for quantum computation," *The American Physical Society*, vol. 52, pp. 3457–3467, 1995.

[8] D. M. Miller, R. Wille, and Z. Sasanian, "Elementary quantum gate realizations for multiple-control toffoli gates," in *Int'l Symp. on Multi-Valued Logic*, 2011, pp. 288–293.

[9] V. V. Shende, A. K. Prasad, I. L. Markov, and J. P. Hayes, "Synthesis of reversible logic circuits," *IEEE Trans. on CAD*, vol. 22, no. 6, pp. 710–722, 2003.

[10] D. M. Miller, D. Maslov, and G. W. Dueck, "A transformation based algorithm for reversible logic synthesis," in *Design Automation Conf.*, 2003, pp. 318–323.

[11] D. Maslov and G. W. Dueck, "Reversible cascades with minimal garbage," *IEEE Trans. on CAD*, vol. 23, no. 11, pp. 1497–1509, 2004.

[12] R. Wille, H. M. Le, G. W. Dueck, and D. Große, "Quantified synthesis of reversible logic," in *Design, Automation and Test in Europe*, 2008, pp. 1015–1020.

[13] R. Wille and R. Drechsler, "Effect of BDD optimization on synthesis of reversible and quantum logic," in *Conference on Reversible Computation*, 2009, pp. 33–45.

[14] K. Fazel, M. Thornton, and J. Rice, "ESOP-based Toffoli gate cascade generation," in *Communications, Computers and Signal Processing, 2007. PacRim 2007. IEEE Pacific Rim Conference on*, 2007, pp. 206 –209.

[15] R. Wille, E. Schonborn, M. Soeken, and R. Drechsler, "SyReC: A hardware description language for the specification and synthesis of reversible circuits," *INTEGRATION, the VLSI Jour.*, vol. 53, no. 3, pp. 39 – 53, 2016.

[16] M. K. Thomsen, "A functional language for describing reversible logic," in *Forum on Specification and Design Languages*, 2012, pp. 135–142.

[17] R. Wille, M. Soeken, E. Schönborn, and R. Drechsler, "Circuit line minimization in the HDL-based synthesis of reversible logic," in *IEEE Annual Symposium on VLSI*, 2012, pp. 213–218.

[18] Z. Al-Wardi, R. Wille, and R. Drechsler, "Towards line-aware realizations of expressions for HDL-based synthesis of reversible circuits," in *Conference on Reversible Computation*, 2015, pp. 233–247.

[19] R. Wille, O. Keszöcze, and R. Drechsler, "Determining the minimal number of lines for large reversible circuits," in *Design, Automation and Test in Europe*, 2011, pp. 1204–1207.

[20] R. Wille, M. Soeken, and R. Drechsler, "Reducing the number of lines in reversible circuits," in *Design Automation Conf.*, 2010, pp. 647–652.

[21] R. Wille, M. Soeken, D. M. Miller, and R. Drechsler, "Trading off circuit lines and gate costs in the synthesis of reversible logic," *INTEGRATION, the VLSI Jour.*, vol. 47, no. 2, pp. 284–294, 2014.

[22] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler, "RevLib: an online resource for reversible functions and reversible circuits," in *Int'l Symp. on Multi-Valued Logic*, 2008, pp. 220–225, RevLib is available at http://www.revlib.org.