Towards HDL-based Synthesis of Reversible Circuits with No Additional Lines

Robert Wille*, Majid Haghparast†, Smaran Adarsh‡, and Tanmay M‡

*Institute for Integrated Circuits, Johannes Kepler University Linz, Austria
†Yadegar-e-Imam Khomeini (RAH) Shahre Rey Branch, Islamic Azad University, Tehran, Iran
‡People’s Education Society University, Bangalore, India

robert.wille@jku.at http://iic.jku.at/eda/research/quantum/

Abstract—Reversible circuits are needed in different emerging technologies, but their design is still mainly conducted on low abstraction levels thus far. Hardware Description Languages (HDLs) provide suitable description means to lift the design process to higher levels of abstractions. However, synthesis of HDL descriptions thus far still relies on non-reversible building blocks even if the corresponding statements are purely reversible. This leads to reversible circuits with additional circuit lines (i.e., circuit signals)—rendering HDL-based synthesis infeasible for many applications such as quantum computing. In this work, we present a synthesis method which realizes many of the HDL statements with no additional lines at all. To this end, we consider the respective (reversible) HDL statements as an entirety rather than breaking it down into (possibly non-reversible) building blocks. For the first time, this allows to realize many HDL descriptions with no additional circuit lines.

I. INTRODUCTION

Reversible circuits realize an unconventional form of computing in which operations can be employed in both directions, i.e., from the inputs to the outputs and vice versa. The resulting properties are very useful for designing and realizing different emerging technologies. Most prominently, this finds application in the design of quantum circuits [1] which are inherently reversible and, caused by the current momentum in this area, many design approaches first realize reversible circuits which, afterwards, are mapped to corresponding quantum realizations [2], [3]. But also other areas such as adiabatic circuits [4], [5], [6], the design of encoders [7], on-chip interconnects [8], certain aspects of low power design [9], [10], or even verification [11] heavily utilize principles of reversible circuits.

Accordingly, a substantial amount of work has been spent on the efficient synthesis of reversible circuits. Originally, most synthesis approaches focused on realizing the desired circuit with a minimal number of circuit lines1 (e.g., approaches based on truth-tables [12], permutations [13], positive-polarity Reed-Muller spectra [14], and Boolean satisfiability [15]). But since they relied on exponential function representations, their scalability remained limited. In contrast, hierarchical synthesis approaches have been proposed which e.g., employed a divide and conquer approach, i.e., they decompose the function to be realized into smaller functions for which corresponding building blocks are available. Afterwards, these building blocks are cascaded together with respect to the applied decomposition so that eventually the desired circuit results. Approaches relying on two level descriptions [16], [17] or decision diagrams [18], [19] fall in this category. Besides that, also hybrid approaches exist [20]. However, in all these approaches, the design is still mainly conducted on low abstraction levels.

1 Note that, in the domain of reversible circuits, the respective circuit signals are usually called circuit lines.

Hardware Description Languages (HDLs) provide a suitable alternative to those approaches. Accordingly, several reversible HDLs have been introduced in the past (see e.g., [21], [22]). They offer description means which are similar to conventional HDLs (such as Verilog [23] or VHDL [24]), but, at the same time, respect restrictions and rules needed to describe reversible circuits. While this allows to lift the design of those circuits to higher levels of abstractions, the corresponding synthesis approaches still suffer from the main problem of generating too many additional circuit lines. This is, because the corresponding HDL statements are decomposed into sub-statements which may not be reversible anymore. Because of this, additional circuit lines are introduced to embed non-reversible functionality into a reversible one (this is described in more detail later in Section III-A). This leads to reversible circuits with additional circuit lines—rendering HDL-based synthesis infeasible for many applications such as quantum computing where the number of circuit lines is crucial.

Although recent approaches investigated e.g., on re-writing the given HDL description [25], undoing computations [26] [27], or realizing single operations in a more line-aware fashion [28], no solution exists yet, which is capable of realizing HDL descriptions with no additional circuit lines at all. This is particularly unsatisfactory because the originally given HDL is always purely reversible and, hence, at least in principle, allows for a fully reversible description of the circuit to be realized.

In this work, we provide a solution that addresses this problem. The proposed approach rests on a rather simple but not yet investigated idea: Rather than breaking down reversible statements into (possibly non-reversible) building blocks (which cause additional circuit lines), we aim to synthesize those statements as an entirety. In order to handle larger statements (for which purely reversible building blocks are not available and/or cannot efficiently be generated), a translation scheme is provided which maps many of those statements into smaller ones—without “loosing” reversibility and, hence, the need to introduce additional circuit lines. For the first time, this allows to realize many HDL descriptions with no additional circuit lines at all. By this, the proposed synthesis scheme provides the first feasible HDL-based synthesis method for emerging technologies relying on reversible circuits. Experimental evaluations confirm the benefits of the resulting approach. For HDL descriptions which also have been used in the past to evaluate HDL-based synthesis, reversible circuits are generated which do not require a single additional circuit line anymore.

—Reversible circuits are needed in different emerging technologies, but their design is still mainly conducted on low abstraction levels thus far. Hardware Description Languages (HDLs) provide suitable description means to lift the design process to higher levels of abstractions. However, synthesis of HDL descriptions thus far still relies on non-reversible building blocks even if the corresponding statements are purely reversible. This leads to reversible circuits with additional circuit lines (i.e., circuit signals)—rendering HDL-based synthesis infeasible for many applications such as quantum computing. In this work, we present a synthesis method which realizes many of the HDL statements with no additional lines at all. To this end, we consider the respective (reversible) HDL statements as an entirety rather than breaking it down into (possibly non-reversible) building blocks. For the first time, this allows to realize many HDL descriptions with no additional circuit lines.

I. INTRODUCTION

Reversible circuits realize an unconventional form of computing in which operations can be employed in both directions, i.e., from the inputs to the outputs and vice versa. The resulting properties are very useful for designing and realizing different emerging technologies. Most prominently, this finds application in the design of quantum circuits [1] which are inherently reversible and, caused by the current momentum in this area, many design approaches first realize reversible circuits which, afterwards, are mapped to corresponding quantum realizations [2], [3]. But also other areas such as adiabatic circuits [4], [5], [6], the design of encoders [7], on-chip interconnects [8], certain aspects of low power design [9], [10], or even verification [11] heavily utilize principles of reversible circuits.

Accordingly, a substantial amount of work has been spent on the efficient synthesis of reversible circuits. Originally, most synthesis approaches focused on realizing the desired circuit with a minimal number of circuit lines1 (e.g., approaches based on truth-tables [12], permutations [13], positive-polarity Reed-Muller spectra [14], and Boolean satisfiability [15]). But since they relied on exponential function representations, their scalability remained limited. In contrast, hierarchical synthesis approaches have been proposed which e.g., employed a divide and conquer approach, i.e., they decompose the function to be realized into smaller functions for which corresponding building blocks are available. Afterwards, these building blocks are cascaded together with respect to the applied decomposition so that eventually the desired circuit results. Approaches relying on two level descriptions [16], [17] or decision diagrams [18], [19] fall in this category. Besides that, also hybrid approaches exist [20]. However, in all these approaches, the design is still mainly conducted on low abstraction levels.

1 Note that, in the domain of reversible circuits, the respective circuit signals are usually called circuit lines.

Hardware Description Languages (HDLs) provide a suitable alternative to those approaches. Accordingly, several reversible HDLs have been introduced in the past (see e.g., [21], [22]). They offer description means which are similar to conventional HDLs (such as Verilog [23] or VHDL [24]), but, at the same time, respect restrictions and rules needed to describe reversible circuits. While this allows to lift the design of those circuits to higher levels of abstractions, the corresponding synthesis approaches still suffer from the main problem of generating too many additional circuit lines. This is, because the corresponding HDL statements are decomposed into sub-statements which may not be reversible anymore. Because of this, additional circuit lines are introduced to embed non-reversible functionality into a reversible one (this is described in more detail later in Section III-A). This leads to reversible circuits with additional circuit lines—rendering HDL-based synthesis infeasible for many applications such as quantum computing where the number of circuit lines is crucial.

Although recent approaches investigated e.g., on re-writing the given HDL description [25], undoing computations [26] [27], or realizing single operations in a more line-aware fashion [28], no solution exists yet, which is capable of realizing HDL descriptions with no additional circuit lines at all. This is particularly unsatisfactory because the originally given HDL is always purely reversible and, hence, at least in principle, allows for a fully reversible description of the circuit to be realized.

In this work, we provide a solution that addresses this problem. The proposed approach rests on a rather simple but not yet investigated idea: Rather than breaking down reversible statements into (possibly non-reversible) building blocks (which cause additional circuit lines), we aim to synthesize those statements as an entirety. In order to handle larger statements (for which purely reversible building blocks are not available and/or cannot efficiently be generated), a translation scheme is provided which maps many of those statements into smaller ones—without “loosing” reversibility and, hence, the need to introduce additional circuit lines. For the first time, this allows to realize many HDL descriptions with no additional circuit lines at all. By this, the proposed synthesis scheme provides the first feasible HDL-based synthesis method for emerging technologies relying on reversible circuits. Experimental evaluations confirm the benefits of the resulting approach. For HDL descriptions which also have been used in the past to evaluate HDL-based synthesis, reversible circuits are generated which do not require a single additional circuit line anymore.
The resulting synthesis scheme is described in Section IV. The proposed in this work are discussed in Section III. Afterwards, the remainder of this paper is structured as follows: The next section reviews reversible circuits and corresponding circuit lines. However, sometimes additional circuit lines (usually employing a constant input) are added, e.g., to realize non-reversible functions (an aspect which becomes relevant and is discussed in more detail later in Section III-A).

II. BACKGROUND

To keep the paper self-contained, this section briefly reviews the basics on reversible circuits as well as corresponding hardware description languages.

A. Reversible Circuits

Reversible circuits realize functions \( f : \mathbb{B}^n \rightarrow \mathbb{B}^m \) over variables \( X = \{x_1, \ldots, x_n\} \) which have the same number of inputs and outputs (i.e., for which \( n = m \)) and which employ a unique mapping for all input/output patterns. That is, those circuits realize bijections. A reversible circuit \( G = g_1 \ldots g_d \) is composed as a cascade of reversible gates \( g_i \). Each reversible gate in the circuit has the form \( g_i(C, \overline{T}) \), where \( C = \{x_1, \ldots, x_d\} \subseteq X \) are control lines and \( T = \{x_1, \ldots, x_d\} \subseteq X/C \) are target lines. The most important gates (which are also employed in this work) are the Toffoli gate [29] and the Fredkin gate [30]. The Toffoli gate has only one target line \( T = \{x_1\} \) whose value is inverted, iff all positive (negative) control lines are assigned \( 1 \) (0). The Fredkin gate has two target lines \( T = \{x_2, x_3\} \), whose values are swapped, iff all positive (negative) control values are assigned to \( 1 \) (0). In both cases, the value of all remaining lines pass the gate unchanged.

Example 1. Fig. 1 shows a reversible circuit realizing a reversible function over three inputs \( X = \{x_0, x_1, x_2\} \) and with three gates (two Toffoli gates and one Fredkin gate). Each variable of the realized function is represented by a gate.

Since reversible circuits realize bijections which can be computed in both directions, the inverse of a reversible circuit \( G \) (denoted \( G^{-1} \) in the following and realizing the function \( f^{-1} \)) can be easily obtained by \( G^{-1} = g_d^{-1} \cdots g_2^{-1} g_1^{-1} \), where \( g_i^{-1} \) is the inverse gate of \( g_i \). Since Toffoli and Fredkin gates are self-inverse, \( g_i = g_i^{-1} \) holds and, thus, \( G^{-1} \) can simply be obtained by reversing the order of the gates of \( G \).

In order to measure the costs of a circuit, different metrics can be applied which often depend on the respectively defined application area. In this work, we consider metrics called quantum costs and transistor costs as defined in [2], [3] and [31], respectively.\(^2\) They depend on the gates used in the circuit and, hence, are also called gate costs in the following. Besides that, and usually much more important, the number of circuit lines constitute a crucial cost metric. In principle, a reversible function over \( n \) variables can be realized using \( n \) circuit lines. However, sometimes additional circuit lines (usually employing a constant input) are added, e.g., to realize non-reversible functions (an aspect which becomes relevant and is discussed in more detail later in Section III-A).

B. Reversible Hardware Description Languages

Reversible Hardware Description Languages (reversible HDLs) allow for the description and realization of complex and large reversible circuits which would be hard to design manually. In this work, we consider the language SyReC which has been introduced in [21] and constitutes a suitable representative for this work.

SyReC utilizes the concept of reversible assignments which are of the form \( v = e \), \( \overline{v} = e \), \( v 

\overline{v} = e \), where the left-hand side (LHS) variable \( v \) must not appear in the right-hand side (RHS) expression \( e \). Those assignments conduct a so-called reversible update, i.e., update the variable \( v \) using a reversible operation such as XOR (\( \hat{=} \)), increment (\( + = \)), or decrement (\( - = \)) to keep the entire statement reversible.\(^3\) The RHS expression \( e \) may be non-reversible and can either be a signal identifier or a binary expression of the form \( v = e_{left} \otimes e_{right} \), where \( \otimes \) is an arbitrary binary operation and \( e_{left} \) and \( e_{right} \) are again expressions or sub-expressions. A list of the most common binary operations which are directly applicable include arithmetic \((+, −, \ast, /, \% , \ast)\), bitwise \& , | , ~), logical \& , | , ~), and relational \(<, >, <=, >=, ! = \) operations. Although the RHS expression can be non-reversible, reversibility is always ensured because the input values remain unchanged and only reversible updates (for which inverses are available) are conducted—allowing to always employ a statement in both directions.

Example 2. Consider the SyReC description as shown in Fig. 2. Here, the module signature first defines the signal names \((x_0, x_1, x_2, x_3, x_4, x_5)\), types \((in/inout/out)\), and bitwidths \((64)\). The next lines define the statements to be executed. Each statement consists of an LHS signal and an RHS expression. The LHS signal is updated with the reversible operation and the RHS expression. For example, line 2 states that \( x_0 \) is XORed with the results of \((x_1 + x_2 - x_3)\) and, then, the result is stored in \( x_0 \) (i.e., \( x_{i=1} = x_0 \oplus ((x_1 + x_2) - x_3) \)).

For a complete treatment of the reversible HDL considered here (including a complete grammar), we are referring to [26].

\(^2\)Note that we choose these cost metrics as they are also applied in the respective related work. However, the considerations conducted here do not rely on a particular cost metric and can be applied to any other metric as well.

\(^3\)Note that further operations \( f \) can be used for the reversible update as long as there is an inverse operator \( f^{-1} \) with \( f = f^{-1} \). Furthermore, note that XOR is self-inverse, while the inverse of the increment (\( + = \)) is the decrement (\( − = \)) and vice versa.
module main(inout x0(64), in x1(64), in x2(64), in x3(64), in x4(64), out x5(64))
x0 := ((x1 + x2) − x3)
x0 := (((x1 − x2) − x3) + (x1 − x4))
x0 := (((x1 + x2) − (x1 − x4)))
x0 := (((x0 * x1) + x1) + ((x2 * x1) + x4))

Fig. 2: A Simple HDL code

III. MOTIVATION AND GENERAL IDEA

Synthesis of reversible circuits described in terms of a
 reversible HDL constitutes a non-trivial task which has intensely
 been considered in the past years. A major problem in all
 these endeavors was that, even though the respectively given
 HDL is purely reversible in principle, all synthesis approaches
 proposed thus far still required additional circuit lines. This
 section illustrates the problem and briefly discusses why
 this remained to be a challenge until today. Afterwards, we sketch
 an idea to eventually overcome this problem.

A. Considered Problem

Reversible HDLs allow for the definition of complex re-
versible functionality. However, in order to eventually realize
 those descriptions in terms of reversible circuits, a hierarchi-
cal synthesis method has been applied thus far [26]. Here,
 realizations of the respective operations (both, the reversible
 assignment operations (\(\sim\)), (\(+\)) and (\(-\)) as well as
 all considered binary operations (\(+\), \(-\), &\(\&\), |\(\|\), &\(\&\)\(|\(\|\), etc.)
 are first pre-computed or taken from literature such as [32],
 [33], [34]. Then, these building blocks are taken and used to
 realize each statement of the given HDL description towards
 the corresponding circuit realization. To this end, two steps
 are conducted as illustrated in Fig. 3a:

1) Realize the RHS expression \(e\), which yields a
 sub-circuit \(G_e\) composed of existing building blocks
 realizing the respective binary operations.

2) Realize the reversible update, which yields a
 sub-circuit \(G_{e}^{-1}\) composed of existing building blocks
 realizing the respective reversible assignment operation
 and, by this, the entire statement.

The application of this hierarchical flow in which the overall
 (reversible) statement is decomposed into sub-functions causes
 a significant drawback: Since the expression \(e\) (as well as
 possible sub-expressions \(e_{\text{left}}\) and \(e_{\text{right}}\) out of which \(e\)
 is composed of) may describe non-reversible functions (i.e.,
 may include non-reversible binary operations), corresponding
 buildings block for those require additional circuit lines with
 constant inputs. The following example illustrates the issue.

Example 3. Let’s assume the statement \(x_0 \sim x_1 + x_2\) shall be
 realized. Following the hierarchical flow sketched above, first
 a building block \(G_e = G_{x_1+x_2}\) realizing the addition of \(x_1\)
 and \(x_2\) is required. But since addition is not reversible (e.g.,
 the sum 3 may originate from two different input assignments
 \(x_1 = 1/2, x_2 = 2/4\) or \(x_1 = 2/4, x_2 = 1\)), additional circuit
 lines are required to make this building block reversible. A
 possible realization is shown in the left-hand side of Fig. 3c
 (the sub-circuit labeled \(G_{x_1+x_2}\)). However, additional lines
 with constant inputs are used to store the desired output while
 the originally given circuit lines keep the input values—making
 the computation reversible\(^4\). Using this building block in addition

\(^4\) Note that Fig. 3c shows a circuit where all signals are realized with
 bitwidth 2. Accordingly, \(x_1[0]\) and \(x_1[1]\) denote the least significant and most
 significant bit, respectively.

\(^{5}\) Note that \(G_{x_1+x_2}^{-1}\) can be easily determined by just inverting the gates
 from \(G_{x_1+x_2}\).

In the worst case, each non-reversible binary operation that
 occurs in the expression \(e\) causes a further set of additional
 circuit lines. In order to reduce this amount of additional
 lines, several improvements have been proposed. The most
 effective one [35] is to undo all operations conducted in order
 to determine the result of the expression \(e\), after the reversible
 update has been executed. More precisely, after the two steps
 from above have been conducted, a third step as illustrated in
 Fig. 3b is employed:

3) Undo the realization of expression \(e\), i.e., add a
 sub-circuit \(G_{e}^{-1}\).

Example 4. Additionally conducting the third step for the
 statement \(x_0 \sim x_1 + x_2\) considered in the previous example
 eventually yields the complete circuit shown in Fig. 3c (in-
 cluding the sub-circuit shown in the right-hand side of Fig. 3c
 and labelled \(G_{x_1+x_2}^{-1}\)).

Although adding \(G_{x_1+x_2}^{-1}\) substantially increases the gate costs
 of the circuit (by almost doubling it), it sets the additional
 circuit lines back to their constant values. By this, those
 constant values can be used again for the realization of the
 next statement—preventing the need for additional circuit lines
 for each single statement. Motivated by that, several methods
 have been proposed (see e.g., [25], [26], [27], [28]) to keep the
 length of the statement small and/or to optimize expressions so
 that the number of additionally required circuit lines are kept
 as small as possible. However, despite these efforts, no solution
 is known yet, which is capable of realizing these reversible
 statements with no additional circuit lines at all.
B. General Idea

The current state-of-the-art in HDL-based synthesis of reversible circuits clearly is unsatisfactory. Although a fully reversible description of the circuit to be realized is available in terms of an HDL that allows to define purely reversible statements, existing synthesis schemes still rely on non-reversible building blocks introducing additional circuit lines. An obvious way to avoid that is to always consider a reversible statement in its entirety, i.e., without breaking it down into possibly non-reversible building blocks. That this works in principle is illustrated by the following example:

Example 5. Consider again the realization of the statement \( x_0 = x_1 + x_2 \). However, rather than following a hierarchical approach which combines the building block for the +-operation and the =-operation (yielding additional circuit lines), the statement can also be synthesized in its entirety at once—yielding, e.g., the circuit as shown in Fig. 3d.

However, a big problem with this approach is that considering a statement in its entirety requires the availability of building blocks for all possible statements. That is, instead of utilizing and composing building blocks only for single operations such as \( = \), \( + \), \( - \), \( \& \), \( \| \), \( < \), \( > \), etc., such an approach would additionally require building blocks for all possible combinations of them—an infinite amount which obviously leads to an infeasible amount of building blocks to pre-compute. Moreover, even an “on-the-fly”-synthesis often is not feasible—in particular when statements are getting complex and, hence, often cannot be efficiently synthesized anymore (e.g., already a statement such as \( x_0 = (x_1 + x_2) \times (x_3 - x_4) + x_5 \) is highly non-trivial to synthesize in its entirety for a decent bitwidth). Because of these problems, researchers and engineers still opt back to the hierarchical synthesis scheme reviewed above—even if they yield additional circuit lines.

In this work, we propose an alternative that eventually addresses this problem for many of the cases. Our approach rests on a rather simple but not yet investigated idea: Many of the combinations between operations can be easily translated into a sequence of simpler statements. As a very simple example, a statement such as \( x_0 = x_1 + x_2 + x_3 \) (requiring additional circuit lines for each of the binary operations when following the established hierarchical synthesis flow) can be translated into an equivalent sequence of \( x_0 = x_1 + x_2 \) and \( x_0 = x_3 \) (for which building blocks with no additionally circuit lines are available and/or can be generated easily). Following this premise, an HDL-based synthesis for reversible circuits with no additional circuit lines is possible if

1) arbitrary statements (combining various operations) can be translated into a sequence of simple statements (i.e., statements with at most one non-reversible operation only), and
2) building blocks for those simple statements are available and/or can be determined easily.

IV. RESULTING HDL-BASED SYNTHESIS SCHEME

In this section, we provide details of the proposed HDL-based synthesis scheme. More precisely, we first discuss methods to translate arbitrary statements into a universal subset of simple statements. Then, we discuss why determining building blocks realizing those simple statements in its entirety and, hence, without additional circuit lines is a feasible task.

A. Translating Arbitrary Statements to Simple Statements

HDLs such as SyReC reviewed in Section II-B allow for arbitrary definitions of statements. This is defined through a corresponding grammar available in [26]. Having that, statements can take any form of \( v \oplus = e \) (see also Section II-B). The RHS-expression \( e \) (with \( e = e_{\text{left}} \circ e_{\text{right}} \)) can be represented in a tree-like fashion, i.e., an Abstract Syntax Tree (AST), where \( e_{\text{left}} \) and \( e_{\text{right}} \) can themselves be composed of sub-expressions (i.e., represent either operators or operands).6 The root node of the AST represents the operator \( \circ \) of the expression \( e \).

Having this description, term re-writing techniques can be employed to translate a given statement to a corresponding sequence of simple statements. This works particularly well in cases where the statement or the (sub-)expression(s) are composed of operations that are associative. But also certain combinations of operations work well, which are not per se associative but can be adjusted accordingly (e.g., statements/expressions involving several combinations of + and − operations can be translated into simple statements by translating a subtraction into an addition with inverted inputs). In contrast, a few combinations of operations such as − and \( ^\sim \) are harder to translate. But since those combinations rarely occur in existing HDL descriptions (as also confirmed by evaluations summarized in Section V), it is acceptable to ignore those combinations for now.

As a result, many of the arbitrary statements can be translated into a subset of simple statements. This is described for many of the frequently occurring cases in the following. In all following descriptions, we assume thereby that the RHS-expression is provided in terms of an AST in which each internal node corresponds to an operator and each leaf node corresponds to an operand.

Then, the simplest case is where the expression \( e \) is solely composed of \( ^\sim \), +, and − (i.e., operations which are also available as reversible assignment operations). If additionally the signals belonging to \( e_{\text{left}} \) of the expression are not repeated more than once on every level of the AST, apply the following steps:

1) Recursively traverse the AST in a post order fashion and add statements \( e_{\text{left}} \circ -1 = e_{\text{right}} \) for each operation node.
2) Once the overall expression \( e \) is realized, add the statement \( v \oplus = e \) which describes the assignment of the root expression \( e \) to the LHS.
3) Add statements which inverse the statements from Step 1, i.e., add \( e_{\text{left}} \circ -1 = e_{\text{right}} \) for each depth.

Example 6. Consider again the HDL description shown in Fig. 2. The statement in Line 2 (whose RHS-expression is represented by an AST as shown in Fig. 4a) can be translated to the following sequence of statements:

\[
\begin{align*}
x_1 &= x_2 \\
x_1 &= x_3 \\
x_0 &= x_1 \\
x_1 &= x_3 \\
x_1 &= x_2
\end{align*}
\]

Otherwise, if the expression \( e \) is solely composed of \( ^\sim \), +, and − (i.e., operations which are also available as reversible assignment operations), but the signals belonging

\[^6\text{Note that, according to the grammar proposed in [26], } e_{\text{right}} \text{ is never larger than } e_{\text{left}}.\]
1) Traverse the AST in post order fashion and, for the deepest operation node with \( \circ \not\in \{ ^-, +, - \} \), add the statement \( v \circ = e_{\text{left}} \circ e_{\text{right}} \), where \( e_{\text{left}} \) and \( e_{\text{right}} \) are either signals or results of the respective sub-expression.

2) For the rest of the AST:
   
   - If the currently considered operator node has two leaf nodes, add the statement \( v \circ = e_{\text{left}} \circ e_{\text{right}} \) (if \( \circ \in \{ ^-, +, - \} \) or \( v \circ^{-1} = e_{\text{left}} \circ e_{\text{right}} \) (otherwise)
   - If the currently considered operator node has one leaf node and if \( \circ \in \{ ^-, +, - \} \), add the statement \( v \circ = e_{\text{left}} \circ e_{\text{right}} \) (if \( \circ \in \{ ^-, +, - \} \) or \( v \circ^{-1} = e_{\text{left}} \circ e_{\text{right}} \) (otherwise)
   - If the currently considered operator node \( \circ \not\in \{ ^-, +, - \} \), add the statement \( v \circ = e_{\text{left}} \circ e_{\text{right}} \) (if \( \circ \not\in \{ ^-, +, - \} \) or \( v \circ^{-1} = e_{\text{left}} \circ e_{\text{right}} \) (otherwise) where \( \circ_{\text{top}} \) refers to the next top level operation of the currently considered node.

3) Adding corresponding inverse statements for all statements which have been added in Step 2 with \( \circ \in \{ ^-, +, - \} \), i.e. add \( e_{\text{left}} \circ^{-1} e_{\text{right}} \) for all these cases.

**Example 8.** Consider again the HDL description shown in Fig. 2. The statement in Line 4 (whose RHS-expression is represented by an AST as shown in Fig. 4c) can be translated to the following sequence of statements:

\[
\begin{align*}
  x_5 &= (x_0 \cdot x_1) \cdot x_1 \\
  x_4 &= x_2 \\
  x_3 &= x_4 \\
  x_0 &= (x_1/x_3) \\
  x_3 &= x_4 \\
  x_1 &= x_2
\end{align*}
\]

The cases discussed so far cover a huge amount of frequently occurring statements. However, for the statements which do not fall into any of the cases mentioned above, no translation down to simple statements that can be easily realized with no additional circuit lines has been found. Nevertheless, applying the rules from above, still yield reductions with respect to the number of lines as illustrated by the following example.

**Example 9.** Consider again the HDL description shown in Fig. 2. The statement in Line 5 (whose RHS-expression is represented by an AST as shown in Fig. 4d) can be translated to the following sequence of statements:

\[
\begin{align*}
  x_5 &= (x_0 \cdot x_1) \cdot x_1 \\
  x_4 &= x_2 \\
  x_3 &= x_1 + x_4 \\
  x_2 &= x_3
\end{align*}
\]

Here, all the statements except \( x_5 = ((x_0 \cdot x_1) \cdot x_1) \) can be easily realized with no additional circuit lines. Compared to the number of additional lines required when realizing the original statement from Line 6, this allows for a substantial reduction.

Overall, the general idea and the rules proposed above indeed allow for a translation of arbitrary statements into simple statements in most of the cases (this is also confirmed by experimental evaluations summarized later in Section V). By this, the main basis for the proposed synthesis scheme is laid out. Based on that, only a restricted set of building blocks for those simple statements is needed anymore.
Using those benchmarks, we realized corresponding reversible

scriptions provided in [36] which also have been used in the

ensive evaluations have been conducted using HDL de-

realize arbitrary HDL descriptions with no additional circuit

C++ and resulted in a synthesis tool which is capable to

⊕ most one binary operation

fashion. Here, Fig. 6a and Fig. 6b provide corresponding

such as AND and XOR can be realized in an even simpler

inverse addition (i.e., subtraction) are designed using the

methods proposed in [34].

Finally, the sub-circuit

G e operation (as also done in the original HDL-based synthesis).

Having an HDL description translated into simple state-

ments, the determination of the correspondingly needed building

blocks gets much easier. Indeed rather than to realize arbitrary combinations of expressions, now only building blocks for a restricted amount of combinations is needed.

If we restrict ourselves to the statements with at most one non-reversible operator in the right-hand side expression, we end up with a total of 51 different combinations (3 possible assignment operations times 17 possible binary operations). By providing purely reversible building blocks for these combinations, the realization of many statements with no additionally required circuit lines is possible.

In fact, most of these building blocks have a structure which is similar than the one sketched in Fig. 5.

Here, the sub-circuit $G_e$ now realizes the respective binary operations in expression $e$, with the results stored in one of the input lines (rather than an additional circuit line). Then, the sub-circuit $G_B$ realizes the respective reversible assignment operation (as also done in the original HDL-based synthesis). Finally, the sub-circuit $G_e^{-1}$ applies the inverse again. Since the sub-circuit $G_e$ is now guaranteed to be composed of at most one binary operation $\oplus$ only, it can be realized easily. Fig. 3d shows a corresponding example for the statement $x_0 = x_1 \lor x_2$, where the sub-circuits for addition and the inverse addition (i.e., subtraction) are designed using the methods proposed in [34].

Besides that, building blocks realizing Boolean operations such as AND and XOR can be realized in an even simpler fashion. Here, Fig. 6a and Fig. 6b provide corresponding examples for $x_0 = x_1 \land x_2$ and $x_0 = x_1 \lor x_2$, respectively.

B. Determining Building Blocks for Simple Statements

Having an HDL description translated into simple statements, the determination of the correspondingly needed building blocks gets much easier. Indeed rather than to realize arbitrary combinations of expressions, now only building blocks for a restricted amount of combinations is needed. If we restrict ourselves to the statements with at most one non-reversible operator in the right-hand side expression, we end up with a total of 51 different combinations (3 possible assignment operations times 17 possible binary operations). By providing purely reversible building blocks for these combinations, the realization of many statements with no additionally required circuit lines is possible.

In fact, most of these building blocks have a structure which is similar than the one sketched in Fig. 5.

Here, the sub-circuit $G_e$ now realizes the respective binary operations in expression $e$, with the results stored in one of the input lines (rather than an additional circuit line). Then, the sub-circuit $G_B$ realizes the respective reversible assignment operation (as also done in the original HDL-based synthesis). Finally, the sub-circuit $G_e^{-1}$ applies the inverse again. Since the sub-circuit $G_e$ is now guaranteed to be composed of at most one binary operation $\oplus$ only, it can be realized easily. Fig. 3d shows a corresponding example for the statement $x_0 = x_1 \lor x_2$, where the sub-circuits for addition and the inverse addition (i.e., subtraction) are designed using the methods proposed in [34].

Besides that, building blocks realizing Boolean operations such as AND and XOR can be realized in an even simpler fashion. Here, Fig. 6a and Fig. 6b provide corresponding examples for $x_0 = x_1 \land x_2$ and $x_0 = x_1 \lor x_2$, respectively.

V. Experimental Evaluation

The concepts introduced above have been implemented in C++ and resulted in a synthesis tool which is capable to realize arbitrary HDL descriptions with no additional circuit lines in almost all cases. In order to confirm the benefits, intensive evaluations have been conducted using HDL descriptions provided in [36] which also have been used in the past to evaluate HDL-based synthesis of reversible circuits. Using those benchmarks, we realized corresponding reversible circuits using the resulting synthesis tool and compared them to circuits generated

- using the original synthesis tool presented in [21] and
- using the state-of-the-art solution available thus far (additionally employing line-aware synthesis as proposed in [26]).

All evaluations have been conducted on an 64-bit Intel machine with 2.66 GHz and 8 GB of main memory. In the following, the main results are presented and discussed.

Table I summarizes the obtained results. The first column provides the name of the respectively considered benchmark as well as the considered bitwidth (for each benchmark, we realized circuits for bitwidth of 16 and 32). Afterwards, the number of additionally required circuit lines and the number of gates as well as the respectively resulting gate costs (based on the metrics reviewed in Section II-A) are provided for each resulting circuit.

The numbers clearly show that the proposed synthesis scheme realizes the HDL descriptions with no additional circuit lines. Considering that all previous work only managed to reduce the number of circuit lines (compare the results from the original synthesis tool to the results from the line-aware synthesis tool) but never where able to completely get rid of them, this confirms the main benefit of the synthesis scheme proposed in this work. Moreover, this improvement can be obtained while, at the same time, hardly further increasing the resulting costs. In fact, in almost all cases the costs are much lower than that of line-aware synthesis scheme. This is particularly remarkable since previous studies frequently showed a trade-off between the number of circuit lines and the quantum costs.

VI. Conclusions

In this paper, we proposed to optimize HDL based synthesis of reversible circuits by considering reversible HDL statements as an entity, instead of breaking it down into non-reversible blocks. Since, there are infinite number of possible statements (leading to an infeasible amount of building blocks), we propose to translate arbitrary statements into a universal subset of simple statements (i.e., statements with at most one non-reversible operation) for which building blocks can be determined easily. Following this approach, the number of additional lines can be reduced to zero, whenever arbitrary description can be fully translated to simple statements. But also in all other cases, the descriptions can be translated and, hence, a reduction becomes possible. As the number of circuit lines is usually considered to be a very limited resource (e.g., in the domain of quantum computation), these achievements are promising.

ACKNOWLEDGMENTS

This work has partially been supported by the LIT Secure and Correct System Lab funded by the State of Upper Austria.

REFERENCES

**TABLE I: Experimental Results**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>new_alu</td>
<td>16</td>
<td>32</td>
<td>306</td>
<td>1874</td>
<td>5087</td>
<td>17</td>
<td>516</td>
<td>3524</td>
<td>9168</td>
<td>0</td>
<td>514</td>
<td>3266</td>
<td>8640</td>
</tr>
<tr>
<td>new_alu</td>
<td>32</td>
<td>64</td>
<td>626</td>
<td>3858</td>
<td>10584</td>
<td>33</td>
<td>1060</td>
<td>7268</td>
<td>18896</td>
<td>0</td>
<td>1058</td>
<td>6754</td>
<td>17856</td>
</tr>
<tr>
<td>for</td>
<td>16</td>
<td>128</td>
<td>1218</td>
<td>7490</td>
<td>20552</td>
<td>17</td>
<td>2052</td>
<td>14084</td>
<td>36624</td>
<td>0</td>
<td>2050</td>
<td>13058</td>
<td>34560</td>
</tr>
<tr>
<td>for</td>
<td>32</td>
<td>256</td>
<td>2498</td>
<td>15426</td>
<td>41856</td>
<td>33</td>
<td>4226</td>
<td>29060</td>
<td>75536</td>
<td>0</td>
<td>4226</td>
<td>27010</td>
<td>71424</td>
</tr>
<tr>
<td>parity</td>
<td>16</td>
<td>30</td>
<td>65</td>
<td>313</td>
<td>1000</td>
<td>16</td>
<td>127</td>
<td>615</td>
<td>1976</td>
<td>0</td>
<td>65</td>
<td>313</td>
<td>1000</td>
</tr>
<tr>
<td>parity</td>
<td>32</td>
<td>62</td>
<td>129</td>
<td>633</td>
<td>2024</td>
<td>32</td>
<td>255</td>
<td>1255</td>
<td>4024</td>
<td>0</td>
<td>129</td>
<td>633</td>
<td>2024</td>
</tr>
<tr>
<td>call</td>
<td>16</td>
<td>32</td>
<td>514</td>
<td>3266</td>
<td>8640</td>
<td>17</td>
<td>692</td>
<td>4884</td>
<td>12464</td>
<td>0</td>
<td>482</td>
<td>3106</td>
<td>8128</td>
</tr>
<tr>
<td>call</td>
<td>32</td>
<td>64</td>
<td>1085</td>
<td>6758</td>
<td>17856</td>
<td>33</td>
<td>1428</td>
<td>10100</td>
<td>25776</td>
<td>0</td>
<td>994</td>
<td>6434</td>
<td>16832</td>
</tr>
<tr>
<td>parity_check</td>
<td>16</td>
<td>32</td>
<td>69</td>
<td>333</td>
<td>1064</td>
<td>17</td>
<td>135</td>
<td>655</td>
<td>2104</td>
<td>0</td>
<td>69</td>
<td>333</td>
<td>1064</td>
</tr>
<tr>
<td>parity_check</td>
<td>32</td>
<td>64</td>
<td>133</td>
<td>653</td>
<td>2088</td>
<td>33</td>
<td>263</td>
<td>1295</td>
<td>4152</td>
<td>0</td>
<td>133</td>
<td>653</td>
<td>2088</td>
</tr>
<tr>
<td>gray_binary</td>
<td>16</td>
<td>30</td>
<td>94</td>
<td>462</td>
<td>1472</td>
<td>2</td>
<td>156</td>
<td>764</td>
<td>2448</td>
<td>0</td>
<td>64</td>
<td>312</td>
<td>992</td>
</tr>
<tr>
<td>gray_binary</td>
<td>32</td>
<td>62</td>
<td>190</td>
<td>942</td>
<td>3008</td>
<td>2</td>
<td>316</td>
<td>1564</td>
<td>5008</td>
<td>0</td>
<td>128</td>
<td>632</td>
<td>2016</td>
</tr>
<tr>
<td>deutsch1</td>
<td>16</td>
<td>16</td>
<td>48</td>
<td>48</td>
<td>384</td>
<td>16</td>
<td>80</td>
<td>80</td>
<td>640</td>
<td>0</td>
<td>32</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>deutsch2</td>
<td>32</td>
<td>32</td>
<td>96</td>
<td>96</td>
<td>768</td>
<td>32</td>
<td>160</td>
<td>160</td>
<td>1280</td>
<td>0</td>
<td>64</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td>deutsch3</td>
<td>16</td>
<td>16</td>
<td>64</td>
<td>64</td>
<td>768</td>
<td>16</td>
<td>96</td>
<td>96</td>
<td>768</td>
<td>0</td>
<td>48</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>Single_long_statement</td>
<td>32</td>
<td>80</td>
<td>504</td>
<td>844</td>
<td>4240</td>
<td>80</td>
<td>864</td>
<td>1584</td>
<td>8352</td>
<td>0</td>
<td>800</td>
<td>1520</td>
<td>7072</td>
</tr>
<tr>
<td>Multiple_statement</td>
<td>32</td>
<td>160</td>
<td>1032</td>
<td>1776</td>
<td>8720</td>
<td>160</td>
<td>1776</td>
<td>3264</td>
<td>17184</td>
<td>0</td>
<td>1152</td>
<td>2144</td>
<td>9664</td>
</tr>
<tr>
<td>Multiple_statement</td>
<td>32</td>
<td>48</td>
<td>560</td>
<td>1040</td>
<td>4926</td>
<td>16</td>
<td>768</td>
<td>1488</td>
<td>7584</td>
<td>0</td>
<td>768</td>
<td>1488</td>
<td>6816</td>
</tr>
<tr>
<td>inputs_repeated</td>
<td>32</td>
<td>96</td>
<td>816</td>
<td>1536</td>
<td>7456</td>
<td>48</td>
<td>1296</td>
<td>2496</td>
<td>12768</td>
<td>0</td>
<td>1144</td>
<td>1984</td>
<td>9808</td>
</tr>
<tr>
<td>inputs_repeated</td>
<td>32</td>
<td>192</td>
<td>1680</td>
<td>3168</td>
<td>15392</td>
<td>96</td>
<td>2672</td>
<td>5152</td>
<td>26336</td>
<td>0</td>
<td>2360</td>
<td>4096</td>
<td>20304</td>
</tr>
<tr>
<td>operators_repeated</td>
<td>32</td>
<td>32</td>
<td>272</td>
<td>512</td>
<td>2656</td>
<td>16</td>
<td>424</td>
<td>784</td>
<td>4112</td>
<td>0</td>
<td>240</td>
<td>480</td>
<td>2400</td>
</tr>
<tr>
<td>operators_repeated</td>
<td>32</td>
<td>64</td>
<td>560</td>
<td>1056</td>
<td>5472</td>
<td>32</td>
<td>872</td>
<td>1616</td>
<td>8464</td>
<td>0</td>
<td>496</td>
<td>992</td>
<td>4960</td>
</tr>
</tbody>
</table>

Add. Lines: Number of additionally required circuit lines
Gates: Number of gates
QCTC: Quantum costs and transistor costs (cf. Section II-A)