Chapter 13
SyReC: A Programming Language for Synthesis of Reversible Circuits

Robert Wille, Sebastian Offermann, and Rolf Drechsler

Abstract Reversible logic serves as a basis for emerging technologies like quantum computing and additionally has applications in low-power design. In particular, since conventional technologies like CMOS are going to reach their limits in the near future, reversible logic has been established as a promising alternative. Thus, in the last years this area started to become intensely studied by researchers. In particular, how to efficiently synthesize complex reversible circuits is an important question. So far, only synthesis approaches are available that rely on Boolean function representations, like e.g., truth tables or decision diagrams.

In this chapter, we propose the programming language SyReC that allows to specify and afterwards to automatically synthesize reversible circuits. Using an existing programming language for reversible software design as basis, we introduce new concepts, operations, and restrictions allowing the specification of reversible hardware. Furthermore, a hierarchical approach is presented that automatically transforms the respective statements and operations of the new programming language into a reversible circuit. Experiments show that with the proposed method, complex circuits can be easily specified and synthesized while with previous approaches this often is not possible due to the limits caused by truth tables or decision diagrams.

13.1 Introduction

In the last decades, great achievements have been made in the development of computing machines. However, due to the exponential growth of transistor density and in particular due to the tremendously increasing power dissipation, researchers

R. Wille (✉) • S. Offermann • R. Drechsler
Institute of Computer Science, University of Bremen, 28357 Bremen, Germany
e-mail: rwille@informatik.uni-bremen.de; offerman@informatik.uni-bremen.de; drechsle@informatik.uni-bremen.de

T.J. Kaźmierski and A. Morawiec (eds.), *System Specification and Design Languages: Selected Contributions from FDL 2010*, Lecture Notes in Electrical Engineering 106,
DOI 10.1007/978-1-4614-1427-8_13,

expect that conventional technologies like CMOS will reach their limits in the near future [38]. To further satisfy the needs for more computational power, alternatives are needed.

Reversible logic marks a promising new direction where all operations are performed in an invertible manner. This enables applications e.g., in the domain of low-power design since only reversible circuits can have close to zero power dissipation [3, 9]. In [4], first reversible circuits driven by their input signals only (and accordingly without additional power supplies) already have been implemented. Besides that, also the growing area of quantum computation [14] established itself as a promising application of reversible logic. Quantum computers allow to solve practical relevant problems (e.g., factorization) much faster than conventional circuits [21, 26]. Since every quantum operation is inherently reversible, progress in the domain of reversible logic can directly be applied to quantum logic.

As a result, reversible logic became an intensely studied research area leading to first approaches for synthesis [8, 11, 20, 29, 34], optimization [5, 13, 39], verification [27, 28, 31], testing [16, 17, 19, 36], and even debugging [30]. But so far only simple circuits have been considered since most of the respective synthesis approaches still rely on Boolean function descriptions like permutations [20], truth tables [11], binary decision diagrams [29], or positive-polarity Reed-Muller expansion [8]. They do not allow the specification of complex reversible systems.

However, it can be expected that significantly larger reversible or quantum circuits become (physically) realizable in the near future. Thus, synthesis of reversible logic must reach a level which allows the description of complex systems at higher abstractions. For this purpose, programming languages can be exploited. Considering conventional synthesis, approaches using languages like VHDL [10], SystemC [7], or SystemVerilog [22] have been established to specify and subsequently synthesize circuits. Even if first programming languages are also available in the reversible domain (see e.g., [1, 18, 37]), so far they only have been used to design reversible software. Similar approaches for reversible circuit synthesis are still missing.

In this chapter, we propose the programming language SyReC intended to specify and afterwards to automatically synthesize reversible logic. For this purpose, *Janus* [37] – an existing language designed to specify reversible software – is used as basis. We introduce new concepts as well as operations and describe restrictions leading to a programming language that allows the specification of reversible circuits. Afterwards, the respective steps needed to synthesize a program written in this language are described. Therefore, a hierarchical approach is presented that automatically transforms the respective statements and operations of the new programming language into a reversible circuit. Experiments show that complex circuits can be efficiently generated using our approach. A comparison to a previously presented synthesis approach for large reversible functions shows the advantages of our technique.

The remainder of this chapter is structured as follows: Sect. 13.2 introduces reversible logic and therewith provides the basis for the rest of this work. Afterwards, the SyReC programming language as well as the new concepts, operations, and

restrictions applied for hardware synthesis are introduced in Sect. 13.3. Section 13.4 describes the hierarchical synthesis approach and explains in detail how reversible circuits specified in SyReC can be generated. Finally, experimental results and conclusions are given in Sects. 13.5 and 13.6, respectively.

13.2 Reversible Logic

To keep the chapter self-contained, this section introduces the basics of reversible functions and reversible circuits. For a more detailed insight we refer to the respective publications.

Definition 13.1. A multiple-output function $f : \mathbb{B}^n \to \mathbb{B}^m$ is a *reversible function* iff (1) its number of inputs is equal to the number of outputs (i.e., $n = m$) and (2) it maps each input pattern to a unique output pattern.

In other words, each reversible function is a bijection that perform permutations of the set of input patterns. A function that is not reversible is termed *irreversible*. Frequently, (irreversible) multi-output Boolean functions should be represented by reversible circuits. This necessitates the irreversible function to be *embedded* [33] into a reversible one which requires the addition of *constant inputs* and *garbage outputs*.

To realize reversible logic some restrictions must be considered, i.e., fanouts and feedback are not allowed [14]. This is reflected in the definition of reversible circuits.

Definition 13.2. A *reversible circuit* G over inputs $X = \{x_1, x_2, \ldots, x_n\}$ is a cascade of reversible gates g_i, i.e., $G = g_1 g_2 \cdots g_d$ where d is the number of gates. A *reversible gate* has the form $g(C,T)$, where $C = \{x_{i_1}, \ldots, x_{i_k}\} \subset X$ is the set of *control lines* and $T = \{x_{j_1}, \ldots, x_{j_l}\} \subset X$ with $C \cap T = \emptyset$ is the set of *target lines*. C may be empty. The gate operation is applied to the target lines iff all control lines meet the required control conditions. Control lines and unconnected lines always pass through the gate unaltered.

In the past, Toffoli and Fredkin gates have been established as a universal gate library for reversible circuits:

- A Toffoli gate [25] has a single target line x_j and maps the input $(x_1, x_2, \ldots, x_j, \ldots, x_n)$ to the output $(x_1, x_2, \ldots, x_{i_1} x_{i_2} \cdots x_{i_k} \oplus x_j, \ldots, x_n)$. That is, a Toffoli gate inverts the target line iff all control lines are assigned to 1.
- A Fredkin gate [6] has two target lines x_{j_1} and x_{j_2}. The gate interchanges the values of the target lines iff the conjunction of all control lines evaluates to 1.

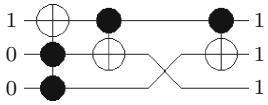
To determine the effort to realize a reversible circuit, the following cost models are used depending on the addressed technology:

- *Gate count* denotes the number of gates the circuit consists of (i.e., d).

Table 13.1 Quantum cost for Toffoli gates

	Quantum cost	
No. of control lines	Of a Toffoli gate	Of a Fredkin gate
0	1	3
1	1	7
2	5	15
3	13	28, if at least 2 lines are unconnected 31, otherwise
4	26, if at least 2 lines are unconnected 29, otherwise	40, if at least 3 lines are unconnected 54, if 1 or 2 lines are unconnected 63, otherwise
5	38, if at least 3 lines are unconnected 52, if 1 or 2 lines are unconnected 61, otherwise	52, if at least 4 lines are unconnected 82, if 1, 2 or 3 lines are unconnected 127, otherwise
6	50, if at least 4 lines are unconnected 80, if 1, 2 or 3 lines are unconnected 125, otherwise	64, if at least 5 lines are unconnected 102, if 1, 2, 3 or 4 lines are unconnected 255, otherwise

Fig. 13.1 Reversible circuit



- *Quantum cost* denotes the effort needed to transform a reversible circuit to a quantum circuit. Table 13.1 shows the quantum cost for a selection of Toffoli and Fredkin gate configurations as introduced in [2] and further optimized e.g., in [12]. As can be seen, gates of larger size are considerably more expensive than gates of smaller size. The sum of the quantum cost for all gates defines the quantum cost of the whole circuit.
- *Transistor cost* denotes the effort needed to realize a reversible circuit in CMOS according to [24] (with slight extensions). The transistor cost of a Toffoli or Fredkin gate is $8 \cdot s$ where s is the number of control lines.

Example 13.1. Figure 13.1 shows a reversible circuit consisting of Toffoli and Fredkin gates. The control lines are thereby denoted by ●, while the target lines are denoted by $\oplus$ (for Toffoli gates) and $\times$ (for Fredkin gates), respectively. The annotated values illustrate the computation of the respective gates. This circuit has a gate count of 4, quantum cost of 10, and transistor cost of 32.

13.3 The SyReC Language

In this work, we use *Janus* [37] as basis for the programming language SyReC to specify reversible systems to be synthesized as circuits. This section briefly reviews the syntax of the Janus language. Afterwards, the new concepts and operations added to address circuit synthesis are introduced.

13.3.1 The Software Language Janus

Janus is a reversible language that is simple but yet powerful enough to design practical reversible software systems [37]. It provides fundamental constructs to define control and data operations while still preserving reversibility.

Figure 13.2 shows the syntax of Janus. Each Janus program (denoted by P) consists of variable declarations (denoted by D) and procedure declarations. The variables have non-negative integer values and are denoted by strings. They can be grouped as arrays. New variables are initially assigned to 0. Constants are denoted by c. Each procedure consists of a name (id) and a sequence of statements (denoted by S) including operations, reversible conditionals, reversible loops, as well as call and uncall of procedures (lines 4–7 in Fig. 13.2). Variables within statements are denoted by V.

In the following, we distinguish between *reversible assignment operations* (denoted by $\oplus$) and (not necessarily reversible) *binary operations* (denoted by $\odot$). The former ones assign values to a variable on the left-hand side. Therefore, the respective variable must not appear in the expression on the right-hand side. Furthermore, only a restricted set of assignment operations exists, namely increase (+=), decrease (-=), and bit-wise XOR (^=), since they preserve the reversibility (i.e., it is possible to compute these operations in both directions). In particular, the bit-wise XOR is of interest because $a \hat{} = b$ is equal to an assignment $a = b$ if a is equal to 0.

In contrast, binary operations, i.e., arithmetic (+, *, /, %, */), bit-wise (&, |, ^), logical (&&, ||), and relational (<, >, =, !=, <=, >=) operations, may not be reversible. Thus, they can only be used in right-hand expressions which preserve

(1) P ::= D^* (**procedure** *id* S^+)$^+$
(2) D ::= x | $x[c]$
(3) V ::= x | x[E]
(4) S ::= V $\oplus$= E |
(5) **if** E **then** S **else** S **fi** E |
(6) **from** E **do** S **loop** S **until** E |
(7) **call** *id* | **uncall** *id* | **skip**
(8) E ::= c | V | (E $\odot$ E)
(9) $\oplus$ ::= **+** | **-** | ^
(10) $\odot$ ::= $\oplus$ | * | **/** | **%** | ***/** | **&** | | | **&&** | || |
(11) < | > | = | **!=** | <= | >=

Fig. 13.2 Syntax of the software language Janus

Fig. 13.3 Syntax of the hardware language SyReC

(1) P ::= D^* (**procedure** *id* S^+)$^+$
(2) D ::= *x* | *x* (*c*)
(3) V ::= *x* | *x*.*N*:*N* | *x*.*N*
(4) N ::= *c* | #V
(5) S ::= V <=> V | V $\oplus$= E |
(6) **if** E **then** S **else** S **fi** E |
(7) **from** *N* **do** S **loop** S **until** *N* |
(8) **for** *N* **do** S **until** *N* |
(9) **call** *id* | **uncall** *id* | **skip**
(10) E ::= *N* | V | (E $\odot$ E) | (E $\circledcirc$ *N*)
(11) $\oplus$ ::= **+** | **-** | ^
(12) $\odot$ ::= $\oplus$ | ***** | **/** | **%** | ***/** | **&** | | | **&&** | || |
(13) < | > | = | **!=** | <= | >=
(14) $\circledcirc$ ::= << | >>

(i.e., do not modify) the values of the respective inputs. In doing so, all computations remain reversible since the input values can be applied to revert any operation. For example, to specify a multiplication (i.e., $a * b$) in Janus, a new free variable c must be introduced which is used to store the product (i.e., $c \hat{=} a * b$ is applied). In comparison, to common (irreversible) programming languages this forbids statements like $a = a * b$. Having this as basis, Janus can be used to specify reversible programs and execute them in a reversible manner (i.e., forward and backward).

13.3.2 The Hardware Language SyReC

In the following, we describe the programming language SyReC for synthesis of reversible circuits.[1] We used Janus as basis and extended this language by further concepts (e.g., declaring circuit signals of different bit-width) and operations (e.g., bit-access and shifts). Besides that, some restrictions must be applied (e.g., dynamic loops are forbidden in hardware). Incorporating all these aspects, a syntax of a programming language for reversible circuit synthesis as depicted in Fig. 13.3 results. More precisely, the following extensions and restrictions have been applied:

- The declaration of variables has been extended so that the designer can declare variables with different bit-widths (line 2).
- Arrays are not allowed.
- Operators to access single bits (*x*.*N*), a range of bits (*x*.*N*:*N*), as well as the size (#V) of a variable, respectively, have been added (line 3 and line 4).
- Since loops must be completely unrolled during synthesis, the number of iterations has to be available before compilation. That is, dynamic loops (defined by expressions) are not allowed (line 7).

[1]Note that we focus thereby on the concepts of the language. A detailed technical definition of SyReC can be found at RevLib.org [32].

Fig. 13.4 SyReC example: simple ALU

```
op ( 2 ) x0 x1 x2
procedure alu
if (op = 0) then
    x0 ^= (x1 + x2)
else
    if (op = 1) then
        x0 ^= (x1 - x2)
    else
        if (op = 2) then
            x0 ^= (x1 * x2)
        else
            x0 ^= (x1 ^ x2)
        fi (op = 2)
    fi (op = 1)
fi (op = 0)
```

- Macros for the SWAP operation (<=>) (line 5) as well as for the for-loop statement (line 8) have been added.[2]
- Further operations used in hardware design (e.g., shifts ⋘) have been added (line 10 and line 14).

Example 13.2. Figure 13.4 shows a simple *Arithmetic Logic Unit* (ALU) illustrating the core concept of the resulting hardware programming language. Thereby, the basic arithmetic operations can easily be applied. Furthermore, control variables can be defined with a lower bit-width than data variables.

In contrast to previous approaches, this allows a much easier specification of (complex) reversible circuits. Having this, the next section describes how circuits can be synthesized from this representation.

13.4 Synthesis of Circuits

Using the language introduced above it is possible to specify reversible circuits on a higher level. As demonstrated by Example 13.2 this particularly allows to design complex circuits in an easier way than e.g., by truth tables or decision diagrams. Nevertheless, the specified circuits still need to be synthesized. To this end, we propose a hierarchical synthesis method that uses existing realizations of the individual operations (i.e., basic blocks) and combines them so that the desired circuit results. More precisely, our approach (1) traverses the whole program and (2) adds cascades of reversible gates to the circuit to be synthesized for each statement or expression, respectively.

[2]These extensions are not necessarily needed (i.e., they can also be expressed by the existing operations), but they allow a more intuitive programming of reversible circuits.

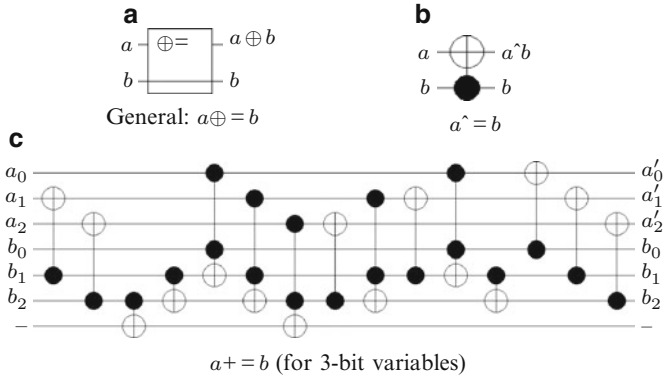


Fig. 13.5 Realization of reversible assignment operations

In the following, the individual mappings of the operations and expressions to the respective reversible cascades are described in detail. Due to page limitations it is not possible to give the concrete mappings for *all* operations of the language. Thus, we describe the general idea for the different kinds and give concrete examples where appropriate.

13.4.1 Reversible Assignment Operations

As introduced in Sect. 13.3, reversible assignment operations include those which are reversible even if they assign a new value to the variable on the left-hand side of a statement. In the following, we use the notation depicted in Fig. 13.5a to denote such an operation in a circuit structure.[3] Solid lines represent the variable(s) on the right-hand side of the operation, i.e., the variable(s) whose values are preserved.

The simplest reversible assignment operation is the bit-wise XOR (e.g., a^=b). For Boolean variables, this operation can be synthesized by a single Toffoli gate as shown in Fig. 13.5b. If variables with a bit-width greater than 1 are applied, then a Toffoli gate has to be applied analogously for each bit.

To synthesize the increase operation (e.g., $a+=b$), a modified addition network is added. In the past, several realizations of addition in reversible logic have been investigated. In particular, it is well known that the minimal realization of a one-bit adder consists of four Toffoli gates (see e.g., [32]). Thus, cascading the required number of one-bit adders is a possible realization. But since every one-bit adder also requires one constant input, this is a very poor solution with respect to circuit lines. In contrast, heuristic realizations exist that require a fewer number of additional

[3]Figure 13.5a shows the notation for a single bit operation. For larger bit-widths the notation is extended accordingly.

lines (see e.g., [23]). Our approach uses a realization with only one additional line (which additionally can be reused for any further addition operation). A cascade showing this realization for a 3-bit addition is depicted in Fig. 13.5c. Nevertheless, any other adder realization can be applied as well.

Finally, the mapping for the decrease operation is left (e.g., $a- = b$). Here, also the realization from Fig. 13.5c is applied, which is fed with a negated variable value.

13.4.2 Binary Operations

Binary operations include operations that are not necessarily reversible so that its inputs have to be preserved to allow a (reversible) computation in both directions. To denote such operations, in the following the notation depicted in Fig. 13.6a is used. Again, solid lines represent the variable(s) whose values are preserved (i.e., in this case the input variables).

Synthesis of irreversible functions in reversible logic is not new so that for most of the respective operations already reversible circuit realizations exist. Additional lines with constant inputs are thereby applied to make an irreversible function reversible (see e.g., [33]). As an example, Fig. 13.6b shows a reversible cascade that realizes an AND operation. As can be seen, this requires one additional circuit line with a constant input 0. Similar mappings exist for all other operations.

Since binary operations can be applied together with reversible assignment operations (e.g., $c\hat{} = a\&b$), sometimes a more compact realization is possible. More precisely, additional (constant) circuit lines can be saved (at least for some operations), if the result of a binary operation is applied to a reversible assignment operation. As an example, Fig. 13.6c shows the realization for $c\hat{} = a\&b$ where no constant input is needed but the circuit line representing c is used instead. However, such a "combination" is not possible for all operations. As an example, Fig. 13.6d shows a two-bit addition whose result is applied to a bit-wise XOR, i.e., $c\hat{} = a + b$. Here, removing the constant lines and directly applying the XOR operation on the lines representing c would lead to a wrong result. This is because intermediate results are stored at the lines representing the sum. Since these values are reused later, performing the XOR operation "in parallel" would destroy the result. Thus, to have a combined realization of a bit-wise XOR and an addition, a concrete embedding for this case must be generated. Since finding and synthesizing respective embeddings for all affected operations and combinations, respectively, is a non-trivial task, a more detailed consideration of this aspect is left for future work. So far, constant lines are applied to realize the respective functionality.

Besides that, while for most of the binary operations (in particular for the bit-wise, logical, and relational operations) many realizations already exist (see e.g., [32]), reversible logic realizations of multiplication (and similar operations, like modulo) is still subject of current research. Thus, Fig. 13.6e briefly shows how multiplication is realized by our synthesis method. As can be seen, we apply partial products. Considering one of the factors a, each time a respective bit of this factor

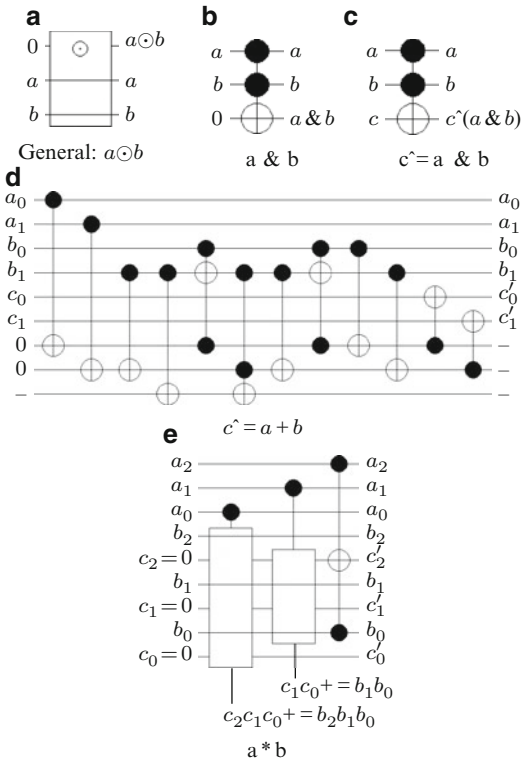


Fig. 13.6 Realization of binary operations

(denoted by a_i) is equal to 1, the respective partial product is added to the product. This allows to reuse the increase realization introduced in the previous section (see [15] for a more detailed treatment).

13.4.3 Conditional Statements, Loops, Call/Uncall

Finally, the realization of control operations as reversible cascades is considered. Loops and procedure calls/uncalls can be realized in a straightforward way. More precisely, loops are realized by simple cascading (i.e., unrolling) the respective statements within a loop block for each iteration. Since the number of iterations must be available before synthesis (see Sect. 13.3), this results in a finite number of statements which is subsequently processed. Call and uncall of procedures are handled similarly. Here, the respective statements in the procedures are cascaded together.

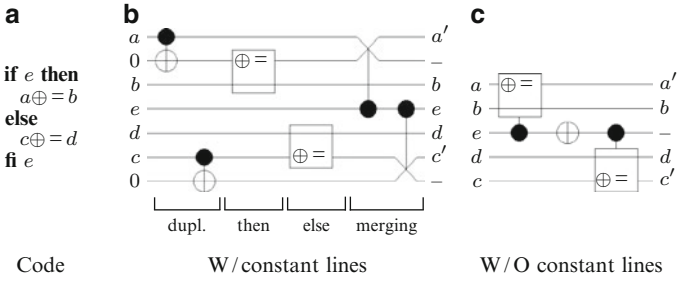


Fig. 13.7 Realization of an if-statement

To realize conditional statements (e.g., the one shown in Fig. 13.7), two variants are proposed. Figure 13.7 shows the first one, which is realized in three steps:

1. All variables in the then- or else-block, respectively, which potentially are assigned to a new value (i.e., that are on the left-hand side of a reversible assignment operation) are duplicated. This respectively requires an additional circuit line with constant input 0.
2. The statements in the respective blocks are mapped to reversible cascades. The duplications introduced in the last step are thereby applied to intermediately store the results of the then-block and the original values of the variables in the else-block, respectively.
3. Depending on the result of the if-statement e, the respective values of the duplicated lines and the original lines are swapped. More precisely, in the example of Fig. 13.7 the value of a is swapped with its (newly assigned) duplication iff e evaluates to 1. Analogously, iff e evaluates to 0 the (newly assigned) value of c is passed through.

The second realization of a conditional statement is depicted in Fig. 13.7. In contrast to the previous one, here no duplications (and therewith no additional circuit lines) are required. Instead, control lines are added to all gates in the realization of the respective then- and else-block. Thus, the operations are computed iff the expression e is assigned to 1 or 0, respectively. A NOT gate (i.e., a Toffoli without control lines) is thereby used to flip the value of e so that the gates of the else block can be "controlled" as well.

Having both realizations, it is up to the designer which one should be used during synthesis. Using the first realization leads to additional circuit lines (particularly in quantum logic a restricted resource). This is not in case in the second realization; however, here due to the additional control lines both the quantum cost and the transistor cost of the circuit significantly increase. Besides other aspects, this is also evaluated in the experiments in the next section.

13.5 Experiments

The proposed synthesis approach for the programming language SyReC has been implemented in C++. In this section, we provide experimental results obtained by this approach. In particular, we evaluate the different realizations of conditional statements in more detail. Furthermore, we compare the results obtained by the proposed approach to a recently published synthesis method based on binary decision diagrams [29].

As benchmarks we use programs including a simple arithmetic logic unit (denoted by *alu*; see also Fig. 13.4), a program computing the average of 8 or 16 values (denoted by *avg8* and *avg16*), a logic unit applying bit-wise operations instead of arithmetic (denoted by *lu*), as well as an arbiter with 8 clients (denoted by *arb8*). Thus, results obtained by programs including arithmetic (*alu*, *avg8*, and *avg16*) as well as bit-wise operations (*lu* and *arb8*) have been evaluated. All experiments have been carried out on an AMD DualCore Athlon 3GHz machine with 32 GB of main memory. The time-out was set to 500 CPU seconds.

In a first evaluation the effect of different if-statement realizations is considered in detail. The results are presented in Table 13.2a. The first column gives the name of the benchmark followed by the applied bit-width of data variables (denoted by BW) and the resulting number of primary inputs (denoted by PI). The following columns give the number of constant input lines (CI), the number of gates (d), the quantum cost (qc), and the transistor cost (tc) of the circuits obtained using the if-realization with additional circuit lines (denoted by IF-STM. W/ ADD. LINES) or without additional circuit lines (denoted by IF-STM. W/O ADD. LINES), respectively. Finally, the run-time is reported for both approaches in the columns denoted by TIME.

The results confirm the discussion from the last section. If additional circuit lines are applied, the respective cost can be significantly reduced. In comparison to the realization without additional circuit lines for if-statements, approx. 80% of the quantum costs and at least for *alu* and *lu* more than 50% of the transistor costs can be saved (this does not hold for the *avg* benchmarks since they do not include if-statements). In contrast, this leads to a significant increase in the number of constant inputs.

Finally, the results are compared to previous work, namely the BDD-based synthesis approach proposed in [29]. Here, a function given as binary decision diagram is used as input. Thus, we extracted the circuits obtained by our approach as BDDs and re-synthesized them based on the concepts of [29].[4] The results are given in Table 13.2b using the same denotation as described above. As can be clearly seen, the proposed approach outperforms the BDD-based synthesis in all objectives: Circuits with significantly less number of gates, quantum cost, and transistor cost, respectively, are synthesized in much less run-time (only the small *arb8* is an

[4] A similar comparison to alternative approaches (e.g., [8, 11]) was not possible since due to memory limitations the respective benchmarks cannot be represented in terms of truth tables which is required by these approaches.

Table 13.2 Experimental results

(a) Applying the programming language SyReC

			IF-STM. W/ ADD. LINES					IF-STM. W/O ADD. LINES				
BENCH	BW	PI	CI	*d*	*qc*	*tc*	TIME	CI	*d*	*qc*	*tc*	TIME
alu	8	26	65	453	2069	5840	0.03s	41	408	11125	13208	0.02s
alu	16	50	121	1345	7377	19856	0.03s	73	1252	40749	45240	0.03s
alu	32	98	233	4473	27785	72464	0.03s	137	4284	155677	166136	0.03s
avg8	8	72	11	405	885	4200	0.01s	11	405	885	4200	0.01s
avg8	16	144	19	861	1853	8872	0.01s	19	861	1853	8872	0.01s
avg8	32	288	35	1773	3789	18216	0.01s	35	1773	3789	18216	0.01s
avg16	8	136	12	754	1654	7832	0.01s	12	754	1654	7832	0.01s
avg16	16	272	20	1602	3462	16536	0.01s	20	1602	3462	16536	0.01s
avg16	32	544	36	3298	7078	33944	0.01s	36	3298	7078	33944	0.01s
lu	8	26	64	164	392	1328	0.02s	40	119	1960	2960	0.02s
lu	16	50	120	308	744	2544	0.02s	72	215	3768	5584	0.02s
lu	32	98	232	596	1448	4976	0.03s	136	407	7384	10832	0.02s
arb8	1	16	37	80	296	640	0.45s	1	24	746	800	0.44s

(b) BDD-based synthesis

			BDD-based synthesis				
BENCH	BW	PI	CI	*d*	*qc*	*tc*	TIME
alu	8	26	768	3560	11196	70792	0.06s
alu	16	50	541099	2842702	9380494	57530752	283.63s
alu	32	98	–	–	–	–	>500.00s
avg8	8	72	2933	10449	36581	217240	4.61s
avg8	16	144	–	–	–	–	>500.00s
avg8	32	288	–	–	–	–	>500.00s
avg16	8	136	7410	25454	89938	532424	9.61s
avg16	16	272	–	–	–	–	>500.00s
avg16	32	544	–	–	–	–	>500.00s
lu	8	26	111	331	823	5928	0.01s
lu	16	50	215	651	1623	11688	0.03s
lu	32	98	423	1291	3223	23208	0.08s
arb8	1	16	15	49	101	824	0.01s

exception). Moreover, in particular for the benchmarks including arithmetic (i.e., *alu* and *awg*) for large bit-widths no circuit can be synthesized within the given time-out. This can be explained by the fact, that in particular for the multiplication no efficient representation as BDD exists. Thus, for these examples the BDD-based approach suffers from memory explosion.

Altogether, SyReC allows the specification of complex circuits that are hard to describe in terms of a decision diagram or truth table, respectively. Afterwards, the specified circuits can efficiently be synthesized.

13.6 Conclusions and Future Work

In this chapter, we proposed the programming language SyReC to synthesize reversible circuits. Based on the software language Janus, we introduced new concepts, operations, and restrictions allowing the specification of reversible hardware. A hierarchical approach has been proposed that uses "basic blocks" to transform the respective statements into cascades of reversible gates. The steps to synthesize the given program as a reversible circuit have been described.

The experiments show that, using this approach, it is possible to efficiently synthesize complex reversible circuits. Both, the SyReC-codes as well as the resulting circuits of the experiments are available online at RevLib.org [32].

Since this work addresses synthesis of reversible circuits by means of programming languages for the first time, the proposed approach also builds the basis for further research. In particular, the reduction of the number of circuit lines is important (in particular for applications in quantum logic where qubits and, therewith, the number of lines are restricted. For this purpose, optimization approaches as e.g., introduced in [35] can be applied. But beyond that also more dedicated solutions need to be explored. In this context, also determining better embeddings of the binary operations is a promising task for future work.

Acknowledgment This work was supported by the German Research Foundation (DFG) (DR 287/20-1).

References

1. S. Abramsky. A structural approach to reversible computation. *Theor. Comput. Sci.*, 347(3):441–464, 2005.
2. A. Barenco, C. H. Bennett, R. Cleve, D.P. DiVinchenzo, N. Margolus, P. Shor, T. Sleator, J.A. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *The American Physical Society*, 52:3457–3467, 1995.
3. C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Dev*, 17(6):525–532, 1973.
4. B. Desoete and A. De Vos. A reversible carry-look-ahead adder using control gates. *INTEGRATION, the VLSI Jour.*, 33(1-2):89–104, 2002.
5. D. Y. Feinstein, M. A. Thornton, and D. M. Miller. Partially redundant logic detection using symbolic equivalence checking in reversible and irreversible logic circuits. In *Design, Automation and Test in Europe*, pages 1378–1381, 2008.
6. E. F. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3/4):219–253, 1982.
7. T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
8. P. Gupta, A. Agrawal, and N. K. Jha. An algorithm for synthesis of reversible logic circuits. *IEEE Trans. on CAD*, 25(11):2317–2330, 2006.
9. R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.*, 5:183, 1961.
10. R. Lipsett, C.F. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design*. Kluwer Academic Publishers, Intermetrics, Inc., 1989.

11. D. Maslov, G. W. Dueck, and D. M. Miller. Toffoli network synthesis with templates. *IEEE Trans. on CAD*, 24(6):807–817, 2005.
12. D. Maslov, C. Young, G. W. Dueck, and D. M. Miller. Quantum circuit simplification using templates. In *Design, Automation and Test in Europe*, pages 1208–1213, 2005.
13. D. M. Miller, R. Wille, and R. Drechsler. Reducing reversible circuit cost by adding lines. In *Int'l Symp. on Multi-Valued Logic*, pages 217–222, 2010.
14. M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.
15. S. Offermann, R. Wille, G. W. Dueck, and R. Drechsler. Synthesizing Multiplier in Reversible Logic. In *IEEE Symp. on Design and Diagnostics of Electronic Circuits and Systems*, 2010.
16. K. N. Patel, J. P. Hayes, and I. L. Markov. Fault testing for reversible circuits. *IEEE Trans. on CAD*, 23(8):1220–1230, 2004.
17. M. Perkowski, J. Biamonte, and M. Lukac. Test generation and fault localization for quantum circuits. In *Int'l Symp. on Multi-Valued Logic*, pages 62–68, 2005.
18. A. Di Pierro, C. Hankin, and H. Wiklicky. Reversible combinatory logic. *Mathematical. Structures in Comp. Sci.*, 16(4):621–637, 2006.
19. I. Polian, T. Fiehn, B. Becker, and J. P. Hayes. A family of logical fault models for reversible circuits. In *Asian Test Symp.*, pages 422–427, 2005.
20. V. V. Shende, A. K. Prasad, I. L. Markov, and J. P. Hayes. Synthesis of reversible logic circuits. *IEEE Trans. on CAD*, 22(6):710–722, 2003.
21. P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. *Foundations of Computer Science*, pages 124–134, 1994.
22. S. Sutherland, S. Davidmann, and P. Flake. *System Verilog for Design and Modeling*. Kluwer Academic Publishers, 2004.
23. Y. Takahashi and N. Kunihiro. A linear-size quantum circuit for addition with no ancillary qubits. *Quantum Information and Computation*, 5:440–448, 2005.
24. M. K. Thomson and R. Glück. Optimized reversible binary-coded decimal adders. *J. of Systems Architecture*, 54:697–706, 2008.
25. T. Toffoli. Reversible computing. In W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, page 632. Springer, 1980. Technical Memo MIT/LCS/TM-151, MIT Lab. for Comput. Sci.
26. L. M. K. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang. Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414:883, 2001.
27. G. F. Viamontes, I. L. Markov, and J. P. Hayes. Checking equivalence of quantum circuits and states. In *Int'l Conf. on CAD*, pages 69–74, 2007.
28. S.-A. Wang, C.-Y. Lu, I-M. Tsai, and S.-Y. Kuo. An XQDD-based verification method for quantum circuits. *IEICE Transactions*, 91-A(2):584–594, 2008.
29. R. Wille and R. Drechsler. BDD-based synthesis of reversible logic for large functions. In *Design Automation Conf.*, pages 270–275, 2009.
30. R. Wille, D. Große, S. Frehse, G. W. Dueck, and R. Drechsler. Debugging of Toffoli networks. In *Design, Automation and Test in Europe*, pages 1284–1289, 2009.
31. R. Wille, D. Große, D. M. Miller, and R. Drechsler. Equivalence checking of reversible circuits. In *Int'l Symp. on Multi-Valued Logic*, pages 324–330, 2009.
32. 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*, pages 220–225, 2008. RevLib is available at http://www.revlib.org.
33. R. Wille, O. Keszöcze, and R. Drechsler. Determining the Minimal Number of Lines for Large Reversible Circuits. In *Design, Automation and Test in Europe*, pages 1204–1207, 2011.
34. R. Wille, H. M. Le, G. W. Dueck, and D. Große. Quantified synthesis of reversible logic. In *Design, Automation and Test in Europe*, pages 1015–1020, 2008.
35. R. Wille, M. Soeken, and R. Drechsler. Reducing the Number of Lines in Reversible Circuits. In *Design Automation Conf.*, pages 647–652, 2010.

36. R. Wille, H. Zhang, and R. Drechsler. ATPG for reversible circuits using simulation, Boolean satisfiability, and pseudo Boolean optimization. In *IEEE Annual Symposium on VLSI*, pages 120–125, 2011.
37. T. Yokoyama and R. Glück. A reversible programming language and its invertible self-interpreter. In *Symp. on Partial evaluation and semantics-based program manipulation*, pages 144–153, 2007.
38. 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*, 91(11):1934–1939, 2003.
39. J. Zhong and J.C. Muzio. Using crosspoint faults in simplifying Toffoli networks. In *IEEE North-East Workshop on Circuits and Systems*, pages 129–132, 2006.