Extensions to the Reversible Hardware Description Language SyReC

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, A-4040 Linz, Austria
§Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{alwardi,drechsle}@informatik.uni-bremen.de robert.wille@jku.at

***Abstract*—*Hardware Description Languages* (HDL) are proposed to facilitate the design of complex circuits and to allow for scalable synthesis. While rather established for conventional circuits, HDLs for the design and synthesis of reversible circuits are at the beginning. SyReC is a representative of such an HDL which already has successfully be applied to realize complex functionality in reversible logic. Nevertheless, the grammar and, by this, the functional scope of this language is rather limited. In this work, we propose extensions to the SyReC HDL which will enhance the usability of the language. For each extension, we additionally provide corresponding synthesis schemes. Overall, this yields a new (extended) SyReC HDL, which will simplify the design and realization of corresponding circuits.**

I. Introduction

The reversible computing paradigm is receiving increasing attention (in particular for so-called emerging technologies) and provides the basis for several applications including but not limited to *quantum computation* [1], [2], certain aspects of *low-power design* [3], the design of *adiabatic circuits* [4], interconnect design [5], [6], and *encoding and decoding* devices [7]. Reversible circuits can only realize bijective operations, i.e. functions that map each possible input vector to a *unique* output vector. Then, computations can be conducted in either direction.

The design of reversible circuits and systems significantly differs from the conventional design flow. This is seen in all abstraction levels down to the applied building blocks and gate libraries. Already a simple standard operation like the logical AND illustrates the differences to the reversible computing paradigm: Although it is possible to uniquely compute the inputs of an AND gate whose output is 1 (then both inputs must have been 1 as well), it is not possible to determine the input values if the AND outputs 0.

As a consequence, new methods for the design and synthesis of reversible circuits have been introduced. Thus far, the majority of them focused on the realization of reversible circuits derived from functional descriptions provided in terms of truth tables [8], [9], two-level descriptions [10], [11], decision diagrams [12], [13], or similar (Boolean) function representations. Obviously, these approaches are limited by their restricted scalability and are not competitive to the state-of-the-art design flows available for conventional circuits and systems.

In order to address this problem, hierarchical approaches have been introduced which decompose a given function to be synthesized into a set of smaller sub-functions and, hence, apply a *divide-and-conquer* scheme. Approaches based on *Hardware Description Languages* (HDL) have been proposed [14], [15]. These solution indeed allow for the synthesis of large functionality.

SyReC is a reversible HDL, which has been introduced to facilitate the description of reversible circuits by means of simple high level codes [14]. A corresponding synthesis scheme showed the ability to describe and synthesize complex functionality, such as a reversible CPU [16]. SyReC, as a tool for reversible circuit synthesis, witnessed some enhancements, which focused mainly on optimizing SyReC-based synthesis [17], [14], [18] or the SyReC programming style [19]. On the other hand, the language itself still maintains the same grammar since its first release.

In this work, we propose syntactical extensions to the SyReC language to simplify the programming with this HDL. To this end, grammatical rules of the language are modified to be simpler, new operators are defined, and new statements are added to the grammar. In order to keep the synthesizability of the HDL, we additionally provide corresponding synthesis schemes for each proposed extension. These extensions facilitate SyReC-based design of reversible circuits.

The remainder of this work is structured as follows: The next section provides an overview on the reversible computing paradigm. In Sec. III, the reversible HDL SyReC is briefly reviewed to explain the main features of the language. Afterwards, the proposed extensions are introduced and illustrated by means of examples which compare the original programming style with the newly proposed one. Besides that, how to synthesize the new descriptions is described. Finally, the work is concluded in Sec. V.

II. Preliminaries

To keep the paper self-contained, this section introduces necessary definitions of reversible logic and circuits.

A. Reversible Functions

A propositional or Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}^n$ over the *variables* $X = \{x_1, \ldots, x_n\}$ is called *reversible* if it is bijective. Clearly, many Boolean functions of practical interest are not reversible (e.g. the conjunction of two propositional variables with a truth table represented by the bit-string 0001). In order to realize such functionality as reversible circuit, the corresponding functions are *embedded* [20], [21]. This is achieved by adding so-called *garbage outputs* which are used to distinguish equal output patterns, thus making the function injective. As a last step in the embedding *constant inputs* are added to equalize the number of input variables and output variables of the function, thus making it bijective.

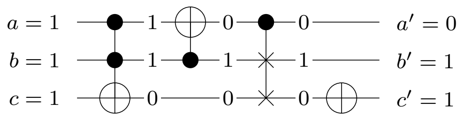


Fig. 1. Reversible circuit

B. Reversible Circuits

Reversible functions can be realized by reversible circuits in which each variable of the function is represented by a *circuit line*. To maintain the bijectivity property of the reversible function, fan-out and feedback are not directly allowed in reversible circuits. As a consequence, reversible circuits can be built as a cascade of reversible gates $G = g_1 \dots g_d$. There exist different gate libraries that are being used to build reversible circuits. However, in the scope of this work we restrict ourselves to the most commonly used ones containing the *Toffoli gate* [22] and the *Fredkin gate* [23]. For this purpose each gate g_i in the circuit is denoted by $t(C, T)$ with

- a gate type $t \in \{\mathrm{T}, \mathrm{F}\}$,
- *control lines* $C \subset X$, and
- *target lines* $T \subseteq X \setminus C$.

Each gate g_i realizes a reversible function $f_i : \mathbb{B}^n \to \mathbb{B}^n$. If $t = \mathrm{T}$, i.e. the gate is a Toffoli gate, we have $T = \{x_t\}$ and f_i maps

$$(x_1, \dots, x_n) \mapsto (x_1, \dots, x_{t-1}, x_t \oplus \bigwedge_{c \in C} c, x_{t+1}, \dots, x_n),$$

i.e. the value on line x_t is inverted if and only if all control values are assigned 1. A Toffoli gate is called a NOT gate if $|C| = 0$. For a Fredkin gate, i.e. $t = \mathrm{F}$, we have $T = \{x_s, x_t\}$ and f_i maps

$$(x_1, \dots, x_n) \mapsto \\ (x_1, \dots, x_{s-1}, x'_s, x_{s+1}, \dots, x_{t-1}, x'_t, x_{t+1}, \dots, x_n),$$

with $x'_s = \bar{c}' x_s \oplus c' x_t$, $x'_t = \bar{c}' x_t \oplus c' x_s$, and $c' = \bigwedge_{c \in C} c$, i.e. the values of the target lines are interchanged (swapped) if and only if all control values are assigned 1. A Fredkin gate is also referred to as SWAP gate if $|C| = 0$. The function realized by the circuit is the composition of the functions realized by the gates, i.e. $f = f_1 \circ f_2 \circ \dots \circ f_d$.

In addition to the constant inputs and garbage outputs that are added to a function in the process of embedding, for circuits we are also considering so-called *ancilla lines*. Ancilla lines hold a constant input assigned some Boolean value v and are used in such a way that their output is always v. Moreover, when considering circuits that realize a complex functionality some lines may be semantically grouped as a *signal*, e.g. if the circuit realizes the addition of two 32-bit values.

Example 1. *Fig. 1 shows a reversible circuit with three lines and four gates. The first, second, and fourth gates are Toffoli gates with a different number of control lines. The target line is denoted by $\oplus$ whereas the control lines are denoted as solid black dots. The third gate is a Fredkin gate which target lines are denoted by $\times$.*

III. The Reversible HDL SyReC

A major motivation of research in the domain of reversible circuit synthesis is the striving for better scalability in order to enable the efficient design of complex functionality. Consequently, HDLs became a focus of ongoing research. One of the first versions of an HDL for reversible circuits named *SyReC* has been introduced in [14]. SyReC is based on the reversible software language *Janus* [24], which has been enriched by further concepts (e.g. declaring circuit signals of different bit-widths), new operations (e.g. bit-access and shifts), and some restrictions (e.g. the prohibition of dynamic loops). In the following, we briefly review the main concepts of this HDL by means of Fig. 2 which depicts a SyReC specification of a simple arithmetic logic unit (for a more detailed treatment, we refer to [14]).

```
1  module simple_alu(in op(2), in a, in b, out c)
2  if (op = 0) then
3    c ^= (a + b)
4  else
5    if (op = 1) then
6      c ^= (a - b)
7    else
8      if (op = 2) then
9        c ^= (a * b)
10     else
11       c ^= (a / b)
12     fi (op = 2)
13   fi (op = 1)
14 fi (op = 0)
```

Fig. 2. SyReC description of a simple ALU

As can be seen in Fig. 2, a SyReC description includes the declaration of modules and signals of the circuit to be specified (Line 1). Signals represent non-negative integers as their sole data type. Furthermore, a variety of statements and expressions are available to specify the functionality of the circuit and, in order to ensure reversibility, these statements must satisfy certain criteria. For example, in each conditional statement, the *if-expression* has to be terminated by a corresponding *fi-expression* (see e.g. Line 8 and 12). Furthermore, statements and expressions are distinguished between *reversible assignment operations* (denoted by $\oplus$=) and not necessarily reversible *binary operations* (denoted by $\odot$).

Reversible assignment operations assign values to a signal on the left-hand side. Therefore, the respective signal 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 (^=). These operations preserve the reversibility (i.e. it is possible to compute these operations in both directions).

In contrast, binary operations, e.g. arithmetic, bit-wise, logical, or relational operations, may not be reversible. Thus, they can only be used in right-hand expressions which preserve the values of the respective inputs. In doing so, all computations remain reversible since the input values can be applied to reverse any operation. For example, to specify the multiplication $(a * b)$ in Fig. 2 Line 9, a new free signal c in combination with a reversible assignment operation is applied.

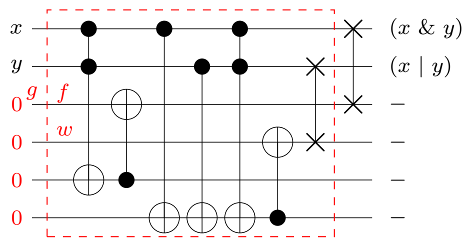
The entire grammar of SyReC as originally proposed in [14] is provided at the end of this paper in Fig. 11.

IV. Extensions to the SyReC Grammar

This section introduces the extensions to the SyReC grammar. For each extension, a comparison to the original programming style is provided. Besides that, we also respectively discuss the corresponding synthesis scheme, which allows to still automatically realize the resulting new HDL descriptions.

```
1  module sub_circuit(inout a(1), inout b(1), out f(1))
2  wire w(1)
3  f ^= (a & b)
4  w ^= (a | b)
5  w <=> b
6
7  module main(inout x(1), inout y(1))
8  wire g(1)
9  call sub_circuit(x,y,g)
10 x <=> g
```

3.a. SyReC code



3.b. Resulting circuit

Fig. 3. SyReC description with sub-module

A. Import of Alternative Circuit Descriptions

SyReC descriptions are modular, e.g. the main module is composed of statements and other modules. When SyReC parses the code, it generates a tree-structure for the main module, which contains sub-modules as sub-trees. The next phase is to convert this tree-structure into a reversible circuit object. This process is a done in a bottom-up fashion, where sub-modules and statements are synthesized first and then interconnected together to generate the main circuit definition.

Example 2. *Consider Fig. 3.a which shows a SyReC program composed of two modules* `main` *and* `sub_circuit`*. The first is the top level module that calls the later in Line 9 using a call-statement. Applying the synthesis method, this yields the circuit as shown the bottom of Fig. 3.b .*

However, thus far, this modularity has only been used to import circuit descriptions provided in the SyReC language itself. But often, building blocks are available (and could be used) which are provided e.g. in gate level descriptions. These descriptions could not been used in the current version of SyReC. This frequently prevents the realization of more compact circuits, because often cheaper building blocks cannot be described in SyReC, but e.g. only in terms of gate level descriptions.

In order to avoid this problem, we propose an extension to support the import of alternative circuit descriptions (as an example of an alternative descriptions, circuits described in the `.real`-format as introduced in [25] are considered). To this end, we extend the grammar from Fig. 11 by allowing for an `<import-list>` as defined in Fig. 12 (Lines 26-27, 33).

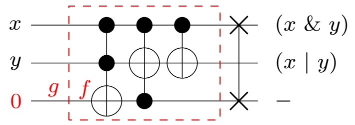
Example 3. *Fig. 4.a shows a circuit description which is functionally equivalent to the description considered before in Fig. 3.a. However, instead of relying on a SyReC description for* `sub_circuit`*, the corresponding buiding block is now provided in terms of a gate level description (provided in the* `.real`*-format) as shown in Fig. 4.b and imported using the newly added statement (Line 1). Overall, this yield a significantly smaller circuit as shown in Fig. 4.c.*

```
1 import sub_circuit from circuit_file.real
2
3 module main(inout x(1), inout y(1))
4 wire g(1)
5 call sub_circuit(x,y,g)
6 x <=> g
```

4.a. SyReC code

```
# ----- circuit_file.real ------
# This file has been generated using
# RevKit 1.3-snapshot (www.revkit.org)
.version 2.0
.numvars 3
.variables x0 x1 x2
.inputs a b const_0
.outputs  a b f
.constants --0
.garbage --1
.begin
t3 x0 x1 x2
t3 x0 x2 x1
t2 x0 x1
.end
```

4.b. Gate level description of the sub-circuit



4.c. Resulting circuit

Fig. 4. SyReC description with gate level sub-module

This extension does not change the call statement in the main code and also does not add any additional complexity to the synthesis scheme. In fact, it simplifies the synthesis process as some parts of the circuit are already synthesized and only have to be accordingly interconnected. However, because the functions of imported circuits are not explicitly described in the code, their behaviors must be verified before importing them.

B. Bit-wise Rotation

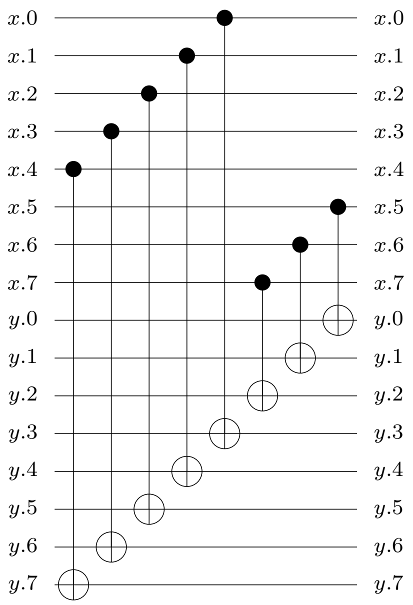
Reversible HDLs are supposed to facilitate the description of reversible circuits and, by this, allow for a more efficient design. To this end, it should provide as much as possible building blocks. While the current version of SyReC already provides several of the corresponding description means, it misses bit-wise rotation operations. This is a problem, since bit-wise rotation is obviously a reversible operation and, additionally, receives special significance in cyclic-coding applications and cryptography [26].

Using the original grammar of SyReC, programmers need to write a sophisticated code to perform rotation opetation bit-by-bit in SyReC. To this end, they can re-use the related shift operators << and >>, which, however, are not reversible and, hence, require additional circuit lines. This is not only counter-intuitive (and, hence, against the aim of facilitating the design process) but also yields significantly larger circuits.

Example 4. *Fig. 5.a shows a SyReC code to update the 8-bit signal* `y` *using the* ^= *operator with the value of the signal* `x` *rotated 3-bits to the left. This code shows that the operation is defined bit-by-bit. Obviously, the resulting code is rather unintuitive. Moreover, the resulting circuit (shown in Fig. 5.b) is rather large due to the need to embedd this shift operation with an additional signal* `x`*.*

```
module test(inout x(8), inout y(8))
y.7:3 ^= x.4:0
y.2:0 ^= x.7:5
```

5.a. SyReC code



5.b. Resulting circuit

Fig. 5. Original SyReC description realizing a signal rotation

To overcome this problem, we propose to extend the SyReC grammar with the corresponding rotation operation <| and |> (for rotate-left and rotate-right, respectively). These new operators are added to the extended SyReC grammar as shown in Fig. 12 (Line 28, 31, 32). Since a rotation operation only swaps bits by a respectively given number, both newly added operations can be synthesized by a set of swap statements.

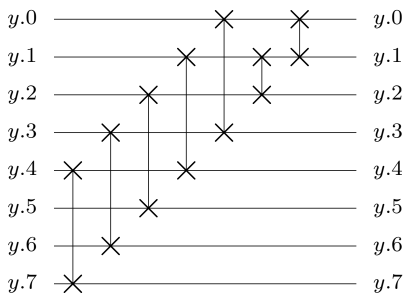
Example 5. *Using the extended grammar, the rotation operation from Fig. 5.a can be described in a significantly more efficient fashion as shown in Fig. 5.a. Moreover, also the resulting circuit (shown in Fig. 5.b) is more compact, because the reversible rotation operation is now directly realized at the 8-bit signal* y *using a cascade of SWAP gates.*

C. Optional fi-conditions

In order to guarantee reversibility of the descriptions of reversible HDLs, a reversible control flow has to be implemented [14]. Consequently, conditional statements do not only require an if-condition (in order to decide which of the then- or the else-block is to be executed next), but also a so-called fi-condition (for the same reason, if the computation is conducted in reverse direction). This was first introduced in the reversible software language *Janus* [24], where the fi-condition is called an assertion. SyReC, which is based on Janus, inherited this assertion. Moreover, HDL descriptions do occur from which it is not possible to realize a reversible control flow at all. Hence, designers of reversible circuits and systems are not only faced with the problem of properly describing a reversible control flow, but also the uncertainty whether such a control flow is even possible.

```
module test(inout x(8), inout y(8))
y.7:3 <=> y.4:0
y.2:1 <=> y.1:0
```

6.a. SyReC code



6.b. Resulting circuit

Fig. 6. Improved SyReC description realizing a signal rotation

Example 6. *Fig. 2 shows a SyReC description with several fi-conditions that are identical to the corresponding if-conditions. In this specific example, the value of* op *(which is used in the if-condition) is not updated within the conditional statement and, hence, this case represents an example of a completely reversible if-statement which can be computed in both directions. In contrast, Fig. 7 shows a SyReC description where the value of the operands* x, y *(which are used in the if-condition) are updated within the conditional statement. Accordingly, a respectively adjusted fi-condition is required. Moreover, it is not obvious anymore whether the entire SyReC description is still fully reversible.*

Generating a fi-condition and/or checking for full reversibility is not always an intuitive task. Accordingly, automatic solutions have recently been proposed in [27] for this purpose.

Example 7. *Consider the conditional statement in Fig. 7. It works for most of the possible assignments of x, y in both directions. However, a problem occurs if e.g. x = 4 and y = 1 are considered. In forward direction, this would not satisfy the if-condition and, hence, would trigger the execution of the else-block (leading to x = 4 and y = 3). This assignment however would satisfy the fi-condition, i.e. if executed in reverse direction, the then-block would reversibly be executed (leading to x = 3 and y = 3). In other words, the two input states (x, y) = (4, 1) and (x, y) = (3, 3) both map to the output state (4, 3) – a clear violation of the reversible computing paradigm.*

```
module partial_fi(inout x(32), inout y(32))
if (x = y) then
  x += 1
else
  y += 2
fi ((x - 1) = y)
```

Fig. 7. Partially reversible program

Now, this partially-reversible description can be checked using the method proposed in [27]. However, then it still remains the question how to fix the problem, i.e. how to transfer this description into a reversible one. In fact, it is not always possible or desirable to generate a fully-reversible condition. But relying on the original grammar of SyReC, this is mandatory.

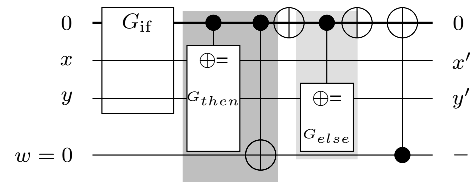


Fig. 8. Realization of a partially reversible program

```
module partial_fi(inout x(32), inout y(32))
wire w(1)
if (x = y) then
  x += 1
  ~= w
else
  y += 2
fi w
```

Fig. 9. Improved SyReC description

As an alternative, we propose to accept this partial reversibility and, instead, apply additional circuit lines (similarly to the embedding process [20], [21] where additional circuits lines are employed to realize non-reversible functions). More precisely, an additional one-bit wire signal is applied which has a default initial value of 0 and gets set to 1 if and only if the if-condition is satisfied. Then, this signal is used to trigger either the respective realization of the then-block or the else block as shown in Fig. 8. This solution guarantees a correct computation at the extra-cost of a single bit only (which can be re-computed to its initial value).

However, in order to make this solution consistent to the definition of SyReC (where partially reversible descriptions are not allowed), we have to modify the grammar again. In fact, we simply have to make the fi-statement optional as shown in Fig. 12 (Line 29). Whenever the fi-condition is provided, a synthesizer simply realizes the corresponding expression. Whenever the condition is omitted, a synthesizer uses the algorithms proposed in [27] to generate a suitable fi-condition. If this is possible (i.e. the conditional statement is completely-reversible), the resulting fi-condition is realized. Otherwise, a single bit wire is automatically declared and applied as described above.

Example 8. *Consider again the conditional statement shown in Fig. 7. As this is a partially reversible description (as determined using the approach from [27]), this code can automatically be transformed to a functionally equivalent description shown in Fig. 9 (with the fi-condition replaced by an additional wire w).*

D. Reversible Case-statement

Considering the same approach in dealing with fi-conditions, other control statements can be proposed. In fact, a reversible case-statement is now possible to be realized. A case-statement is a special form of nested if-statement structure, where all if-conditions have the same form `(choice_expression = <number>)`. Instead of repeating the choice expression with each if-condition, it is provided once at the beginning of the case statement. This will tangibly enhance the readability of the code and simplify the structure. It is a very intuitive task to map those case-statements into equivalent nested-if structures and, hence, also does not constitute a serious obstacle for the synthesie process. Overall, this motivates extensions to the SyReC grammar as shown in Fig. 12 (Line 28, 30).

```
module simple_alu(in op(2), in a, in b, out c)
with  op select
case 0: c ^= (a + b)
case 1: c ^= (a - b)
case 2: c ^= (a * b)
case default: c ^= (a / b)
endcase
```

Fig. 10. SyReC description with case-statement

Example 9. *Fig. 2 and Fig. 10 show two functionally equivalent SyReC representation, where the former is written using the original SyReC grammar (i.e. as a cascade of conditional statements) and the latter is written using the proposed case statement.*

V. Conclusions

In this work, we considered the design of reversible circuits based on the hardware description language SyReC. Although SyReC-based design and synthesis has been considered from various aspects in the past years, the language itself still maintains the same gammas since its first release in 2010. As shown in this work, this causes several shortcomings which makes it hard to describe the desired behavior and often even yields more expensive circuits. To address these problems, we introduced extensions to the language and proposed a revised grammar (shown in Fig. 12) which further facilitates the design of complex reversible circuits. Examples and discussions illustrated the benefit of the extensions. In future work, we will focus on supporting further description means such as PLA and truth tables, BDDs, or even codes written in conventional HDLs. Another concern is to support further data types for numbers and signals.

References

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

[2] 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.

[3] 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.

[4] A. De Vos, *Reversible Computing: Fundamentals, Quantum Computing and Applications*. Weinheim: Wiley-VCH, 2010.

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

[6] R. Wille, O. Keszocze, S. Hillmich, M. Walter, and A. G. Ortiz, "Synthesis of approximate coders for on-chip interconnects using reversible logic," in *Design, Automation and Test in Europe*, pp. 1140–1143, 2016.

[7] A. Zulehner and R. Wille, "Taking one-to-one mappings for granted: Advanced logic design of encoder circuits," in *Design, Automation and Test in Europe*, 2017.

[8] D. M. Miller, D. Maslov, and G. W. Dueck, "A transformation based algorithm for reversible logic synthesis," in *Design Automation Conf.*

[9] M. Soeken, R. Wille, C. Hilken, N. Przigoda, and R. Drechsler, "Synthesis of reversible circuits with minimal lines for large functions," in *ASP Design Automation Conf.*, pp. 85–92, 2012.

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

[11] Y. Sanaee and G. W. Dueck, "ESOP-based Toffoli network generation with transformations," in *Int'l Symp. on Multi-Valued Logic*, pp. 276–281, 2010.

[12] R. Wille and R. Drechsler, "BDD-based synthesis of reversible logic for large functions," in *Design Automation Conf.*, pp. 270–275, 2009.

[13] C.-C. Lin and N. K. Jha, "RMDDS: Reed-Muller decision diagram synthesis of reversible logic circuits," *J. Emerg. Technol. Comput. Syst.*, vol. 10, no. 2, p. 14, 2014.

Program and Modules

```
1  ⟨program*⟩ ::= ⟨module⟩ {⟨module⟩}
2  ⟨module⟩ ::= 'module' ⟨identifier⟩ '(' [⟨parameter-list⟩] ')' {⟨signal-list⟩} ⟨statement-list⟩
3  ⟨parameter-list⟩ ::= ⟨parameter⟩ {',' ⟨parameter⟩}
4  ⟨parameter⟩ ::= ('in' | 'out' | 'inout') ⟨signal-declaration⟩
5  ⟨signal-list⟩ ::= ('wire' | 'state') ⟨signal-declaration⟩ {',' ⟨signal-declaration⟩}
6  ⟨signal-declaration⟩ ::= ⟨identifier⟩ {'['⟨int⟩']'} ['('⟨int⟩')']
```

Statements

```
7  ⟨statement-list⟩ ::= ⟨statement⟩ {';' ⟨statement⟩}
8  ⟨statement*⟩ ::= ⟨call-statement⟩ | ⟨for-statement⟩ | ⟨if-statement⟩ | ⟨unary-statement⟩ | ⟨assign-statement⟩ | ⟨swap-statement⟩ | ⟨skip-statement⟩
9  ⟨call-statement⟩ ::= ('call' | 'uncall') ⟨identifier⟩ '(' (⟨identifier⟩ {',' ⟨identifier⟩}) ')'
10 ⟨for-statement⟩ ::= 'for' [['$' ⟨identifier⟩ '='] ⟨number⟩ 'to'] ⟨number⟩ ['step' ['-'] ⟨number⟩] ⟨statement-list⟩ 'rof'
11 ⟨if-statement*⟩ ::= 'if' ⟨expression⟩ 'then' ⟨statement-list⟩ 'else' ⟨statement-list⟩ 'fi' ⟨expression⟩
12 ⟨assign-statement⟩ ::= ⟨signal⟩ ('^' | '+' | '-') '=' ⟨expression⟩
13 ⟨unary-statement⟩ ::= ('~' | '++' | '--') '=' ⟨signal⟩
14 ⟨swap-statement⟩ ::= ⟨signal⟩ '<=>' ⟨signal⟩
15 ⟨skip-statement⟩ ::= 'skip'
16 ⟨signal⟩  ::= ⟨identifier⟩ {'[' ⟨expression⟩ ']'} ['.' ⟨number⟩ [':' ⟨number⟩]]
```

Expressions

```
17 ⟨expression⟩ ::= ⟨number⟩ | ⟨signal⟩ | ⟨binary-expression⟩ | ⟨unary-expression⟩ | ⟨shift-expression⟩
18 ⟨binary-expression⟩ ::= '(' ⟨expression⟩ ('+' | '-' | '^' | '*' | '/' | '%' | '*>' | '&&' | '||' | '&' | '|' | '<' | '>' | '=' | '!=' | '<=' | '>=')
                        ⟨expression⟩ ')'
19 ⟨unary-expression⟩ ::= ('!' | '~') ⟨expression⟩
20 ⟨shift-expression*⟩ ::= '(' ⟨expression⟩ ('<<' | '>>') ⟨number⟩ ')'
```

Identifier and Constants

```
21 ⟨letter⟩     ::= ('A' | ...| 'Z' | 'a' | ...| 'z')
22 ⟨digit⟩      ::= ('0' | ...| '9')
23 ⟨identifier⟩ ::= ('_' | ⟨letter⟩) {('_' | ⟨letter⟩ | ⟨digit⟩)}
24 ⟨int⟩        ::= ⟨digit⟩ {⟨digit⟩}
25 ⟨number⟩ ::= ⟨int⟩ | '#' ⟨identifier⟩ | '$' ⟨identifier⟩ | ('(' ⟨number⟩ ('+' | '-' | '*' | '/') ⟨number⟩ ')')
```

Fig. 11. Grammar of the hardware description language SyReC. Rules in Line 1 and 8 (highlighted by *) are extended in the new grammar

Program and Modules

```
26 ⟨program⟩ ::= [⟨import-list⟩] ⟨module⟩ {⟨module⟩}
27 ⟨import-list⟩ ::= 'import' ⟨identifier⟩ 'from' ⟨file⟩ {',' ⟨identifier⟩ 'from' ⟨file⟩ }
```

Statements

```
28 ⟨statement⟩ ::= ⟨call-statement⟩ | ⟨for-statement⟩ | ⟨if-statement⟩ | ⟨unary-statement⟩ | ⟨assign-statement⟩ | ⟨swap-statement⟩ | ⟨skip-statement⟩ |
                ⟨case-statement⟩| ⟨rotate-statement⟩
29 ⟨if-statement⟩ ::= 'if' ⟨expression⟩ 'then' ⟨statement-list⟩ 'else' ⟨statement-list⟩ 'fi' [⟨expression⟩]
30 ⟨case-statement⟩ ::= 'with' ⟨identifier⟩ 'select' { 'case' ⟨number⟩ ':' ⟨statement-list⟩} ['case' 'default' ':' ⟨statement-list⟩] 'endcase'
31 ⟨rotate-statement⟩ ::= ⟨signal⟩ ('<|' | '|>') '=' ⟨number⟩
```

Expressions

```
32 ⟨shift-expression⟩ ::= '(' ⟨expression⟩ ('<<' | '>>'| '<|' | '|>') ⟨number⟩ ')'
```

Identifier and Constants

```
33 ⟨file⟩       ::= ⟨identifier⟩ ['.real']
```

Fig. 12. Extension to the grammar of the hardware description language SyReC

[14] R. Wille, E. Schönborn, M. Soeken, and R. Drechsler, "SyReC: A hardware description language for the specification and synthesis of reversible circuits," *INTEGRATION, the VLSI Jour.*, vol. 53, pp. 39–53, 2016.

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

[16] R. Wille, M. Soeken, D. Große, E. Schönborn, and R. Drechsler, "Designing a RISC CPU in reversible logic," in *Int'l Symp. on Multi-Valued Logic*, pp. 170–175, 2011.

[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*, pp. 213–218, 2012.

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

[19] Z. Alwardi, R. Wille, and R. Drechsler, "Re-writing HDL descriptions for line-aware synthesis of reversible circuits," in *Int'l Symp. on Multi-Valued Logic*, pp. 31–36, 2016.

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

[21] 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.

[22] T. Toffoli, "Reversible computing," in *Automata, Languages and Programming* (W. de Bakker and J. van Leeuwen, eds.), p. 632, Springer, 1980. Technical Memo MIT/LCS/TM-151, MIT Lab. for Comput. Sci.

[23] E. F. Fredkin and T. Toffoli, "Conservative logic," *International Journal of Theoretical Physics*, vol. 21, no. 3/4, pp. 219–253, 1982.

[24] T. Yokoyama and R. Glück, "A reversible programming language and its invertible self-interpreter," in *Symp. on Partial evaluation and semantics-based program manipulation*, pp. 144–153, 2007.

[25] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler, "RevLib: an online resource for reversible functions and reversible circuits," pp. 220–225, 2008. RevLib is available at http://www.revlib.org.

[26] K. Datta, V. Shrivastav, I. Sengupta, and H. Rahaman, "Reversible logic implementation of AES algorithm," in *International Conference on Design Technology of Integrated Systems in Nanoscale Era*, pp. 140–144, 2013.

[27] R. Wille, O. Keszöcze, L. Othmer, M. K. Thomsen, and R. Drechsler, "Generating and checking control logic in the hdl-based design of reversible circuits," in *Reversible Computation*, pp. 160–166, 2016.