Generating and Checking Control Logic in the HDL-based Design of Reversible Circuits

Robert Wille[1,2] Oliver Keszocze[2,3] Lars Othmer[2] Michael Kirkedal Thomsen[4] Rolf Drechsler[2,3]
[1]Institute for Integrated Circuits, Johannes Kepler University, Linz, Austria
[2]Cyber Physical Systems, DFKI GmbH, Bremen, Germany
[3]Institute of Computer Science, University of Bremen, Bremen, Germany
[4]DIKU, Department of Computer Science, University of Copenhagen, Denmark
robert.wille@jku.at {keszocze,lothmer,drechsle}@informatik.uni-bremen.de kirkedal@acm.org

***Abstract*—Although different from the conventional computing paradigm, *reversible computation* received significant interest due to its applications in various (emerging) technologies. Here, computations can be executed not only from the inputs to the outputs, but also in the reverse direction. This leads to significantly different design challenges to be addressed. In this work, we consider problems that occur when describing a reversible control flow using *Hardware Description Languages* (HDLs). Here, the commonly used conditional statements must, in addition to the established *if*-condition for forward computation, be provided with an additional *fi*-condition for backward computation. Unfortunately, deriving correct and consistent *fi*-conditions is often not obvious. Moreover, HDL descriptions exist which may not be realized with a reversible control flow at all. In this work, we propose automatic solutions which generate the required *fi*-conditions and check whether a reversible control flow indeed can be realized. The solution utilizes *predicate transformer semantics* based on *Hoare logic*. This has exemplary been implemented for the reversible HDL *SyReC* and evaluated with a variety of circuit description examples. The proposed solution constitutes the first automatic method for these important designs steps in the domain of reversible circuit design.**

I. Introduction

In the vast majority of today's circuits and systems, a conventional computing paradigm is employed in which computations are performed in a single direction only. In contrast to that, circuits and systems following a *reversible* computing paradigm cannot *only* be employed in one direction (i.e. from the inputs to the outputs), but also in the reverse direction (i.e. from the outputs to the inputs).

This paradigm received increasing attention (in particular for so-called emerging technologies) and provides the basis for several applications including but not limited to

- quantum computation [13], since corresponding quantum circuits inherently realize reversible functionality and, hence, can be derived from reversible circuits (cf. [2]),
- certain aspects of low-power design (as experimentally observed e.g. in [3]), since reversible computations are information-lossless and, hence, do not result in power dissipation because of lost bits (a property which is not crucial for today's technologies but may become relevant if feature sizes are continuing to shrink),
- the design of adiabatic circuits (cf. [6]), since reversible circuits are particularly suited for the underlying physical constraints, and
- encoding and decoding devices (cf. [18]), since they do realize one-to-one mappings which inherently follow the reversible computing paradigm.

However, the design of corresponding reversible circuits and systems significantly differs from the conventional design flow. This shows all the way 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. In contrast, reversible circuits and systems can only realize bijective operations, i.e. functions that map each possible input vector to a *unique* output vector.

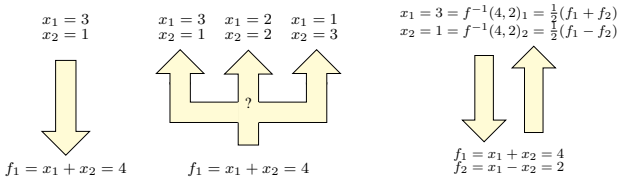
As a consequence, new methods for design and synthesis of reversible circuits have been introduced. Thus far, the majority of them focused on the realization of reversible circuits derived from functional descriptions provided in terms of truth tables [12], [21], two-level descriptions [8], [14], decision diagrams [17], [11], 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, researchers and engineers started the elaboration of *Hardware Description Languages* (HDLs) following the reversible computing paradigm [20], [16]. Similar to established (conventional) languages, such as *Verilog* [10] and *VHDL* [1], they enable the designers to express the intended functionality on a significantly higher level of abstraction, but at the same time also respecting the restrictions of the reversible computing paradigm[1]. Unfortunately, these restrictions often cause further design challenges for the implementer.

In fact, in order to guarantee reversibility of the HDL descriptions, a reversible control flow has to be implemented. For example, 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)[2]. Moreover, HDL descriptions does occur that is not possible to realize using 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 even is possible. Section III describes and illustrates these issues in more detail.

[1]Note that similar developments can be observed in the design of reversible software (see e.g. [22]). Although those are out of the scope of this work, contributions presented here may be advantageous for the design of reversible software languages as well.

[2]In the domain of reversible imperative programming languages, this is also called *fi*-assertions; named because this in forward execution has to be asserted true or false depending on the branch taken [22].



(a) Conventional computation (b) Reversible computation

Fig. 1: Conventional logic vs. reversible logic

In this work, we propose a solution to these two problems. A methodology is presented which applies symbolic simulation in order to automatically generate a representation of all system states that originated from the execution of a conditional statement. From that, the respectively desired *fi*-condition can be derived. Moreover, the symbolic simulation (together with some solving engines) can also be utilized to check whether a given HDL description allows for a fully reversible control flow at all; in other words, whether the control flow of the description is total. Experiments and case studies demonstrate the efficiency and applicability of the proposed solution. As a result, some manual and time-consuming tasks for the design of reversible circuits and systems can now be automated.

The remainder of this work is structured as follows: The next section provides an overview on the reversible computing paradigm and respective HDLs. How to describe control logic in this paradigm is covered in Section III – including a brief discussion of the resulting design challenges. Afterwards, the proposed solutions to address these challenges are presented in Section IV. Finally, results of our experimental evaluation are described in Section V before the paper is concluded in Section VI.

II. Reversible Computation and HDLs

Reversible circuits realize bijective functions, i.e. functions with inputs $X := \{x_1, \ldots, x_n\}$ and outputs $F := \{f_1, \ldots, f_m\}$, where

1) the number of inputs is equal to the number of outputs (i.e. $n = m$) and
2) each input pattern maps to a unique output pattern.

This significantly differs from what conventional circuits realize, as illustrated by the addition operation shown in Fig. 1. Following a conventional interpretation of the adder function (denoted by f_1 in Fig. 1(a)), computations can only be performed in one direction; mapping e.g. an output $f_1 = 4$ back to the inputs does not lead to a unique assignment. In contrast, a reversible circuit would e.g. realize the addition as defined in Fig. 1(b). Here, an additional output is added (f_2) which can be used in order to define an f_1^{-1} as well as an f_2^{-1} and, hence, allows for a reversible computation.

Obviously, these differences have to be reflected in hardware description languages such as [20], [16] which are dedicated to reversible circuits. The basic syntax of these languages, e.g. for the declaration of modules or signals, are similar to conventional languages such as VHDL or Verilog. But data operations have to be defined in a purely reversible fashion. Consequently, the elementary language construct of an arbitrary variable assignment (such as used in the majority of the existing languages) can not be used as they are clearly non-reversible. In contrast, reversible hardware description languages are making use of so-called *reversible assignments* (also known as *reversible updates* [22]).

Reversible assignments have the form $v \oplus= e$ such that the variable v does not appear in the right-hand side expression e. In general, the operator $\oplus$ can realize any function f, as long as there exists an inverse operator g such that

$$v = g(f(v, e), e) \tag{1}$$

for all variables v and for all expressions e. For example, in [20], $\oplus$ is defined by $\oplus \in \{\hat{}, +, -\}$. Here, "+" (addition) is inverse to "−" (subtraction) and vice versa, while "ˆ" (bitwise exclusive OR) is inverse to itself. When executing the description in reverse order, all reversible assignment operators are replaced by their inverse operators.

Using reversible assignments, arbitrary (even non-reversible) expressions e can be applied on the right-hand side. This is possible since the input values to the operation are also given to the inverse operation when reverting the assignment. For example, to specify a multiplication $a * b$, a new free signal c can be introduced which is used to store the result (i.e. c ˆ= a ∗ b is applied). By this, syntactical expressiveness of the language is guaranteed, while all operations are conducted in a reversible fashion.

Example 1. *Following the principles sketched above, the following code (using the* SyReC *syntax*[3]*) describes a reversible circuit computing the value $p(x)$ of the parameterized polynomial $ax^2 + bx + c$.*

```
module pol(in a, in b, in c, in x, out res)
  res ^= a*x*x
  res += b*x
  res += c
```

The first line declares all signals, while the following lines include the statements to be realized. Since all statements perform a reversible assignment, computations can be performed in either direction, i.e. are reversible.

III. Control Logic in Reversible HDLs and Resulting Design Challenges

Relying on reversible assignments the reversible data flow is ensured. However, in a similar fashion the control flow has to be made reversible. This is clearly manifest in conditional statements. Here, in contrast to non-reversible languages, is has to be guaranteed that the correct block (either, the *then*-block or the *else*-block) is executed when performing the computations in reverse direction. To this end, an additional *fi*-condition has to be provided for each conditional statement. If computations are performed in forward direction, the *fi*-condition can be applied as an assertion. If computations are performed in reverse direction, the *fi*-condition decides whether the *then*-block or the *else*-block is supposed to be executed next and the *if*-condition can be used as the end-assertion. The following example illustrates the idea.

Example 2. *Consider the following two conditional statements*

```
if (b = 5) then
  x += y // executed if b = 5
else
```

[3]Note that, although we are illustrating our contributions by means of SyReC descriptions, the proposed methodology is applicable to control logic described in various languages. The syntax of SyReC has been sketched in [20], while a complete definition is available in a technical document provided in [19].

```
  x −= y // executed if b != 5
fi (b = 5);
```

and

```
if ((x % 2) = 1) then
// executed if x % 2 (fwd) or (x − 3) % 2 (bwd)
  x += 3;
else
  x += 1;
  y += c;
fi (((x − 3) % 2) = 1)
```

The first does not modify any of the signals of the conditional expression (signal b in this case). Hence, the if- *and the* fi-*condition are identical. In contrast, the* then-*block of the second conditional statement modifies the value of signal x. Hence, a suitable* fi-*condition different from the* if-*condition has to be provided in order to ensure correct execution semantics in both directions.*

The examples above are very simple, but in general it is not obvious to derive a correct *fi*-condition. In particular when more complex or even nested conditional statements have to be considered, the generation of a correct control logic for a reversible circuit becomes a hard and error-prone task, which has been conducted manually thus far.

Besides that, another problem poses an obstacle to the correct generation of control logic for reversible circuits. Statements in the *then*/*else*-blocks could prevent the generation of a fully reversible control logic; in other words, the *if*-conditions together with two statements might not implement a total and injective (bijective) function. Then, only *fi*-conditions that satisfy parts of the range can be derived.

The following example illustrates the problem[4].

Example 3. *Consider the following conditional statement*

```
if (x = 6) then
  x −= 3;
else
  x += y;
fi (x = 3)
```

This statement works for most of the possible assignments of x and y in both directions. However, a problem occurs if e.g. $x = 1$ and $y = 2$ are considered. In forward direction, this would not satisfy the if-*condition and, hence, would trigger the execution of the* else-*block (leading to $x = 3$ and $y = 2$). This assignment however would satisfy the* fi-*condition, i.e., if executed in reverse direction, the* then-*block would reversibly be executed (leading to $x = 6$ and $y = 2$). In other words, the two input states $(x, y) = (1, 2)$ and $(x, y) = (6, 2)$ both map to the output state $(3, 2)$ – a clear violation of the reversible computing paradigm.*

Cases like this are called *partially reversible control statements* in the following, as they only implement a partial reversible function. Often the conditional statements become partial reversible only because of a very small set of possible signal assignments[5], so detecting such signals becomes even harder than generating the *fi*-condition. Again, no automatic support is available to the designers thus far.

[4]Note that, for sake of clarity, we will restrict our observations to non-negative variables only. The solution presented in this paper is applicable to integers of arbitrary bitwidth with the usual over- and underflow behavior.

[5]In Example 3, this only occurs for inputs $(6, y^*)$ and (x^*, y^*) with $x^* + y^* = 3$.

Overall, this leads to two major challenges to be addressed when designing control logic in HDL-based synthesis of reversible circuits, namely

- how to efficiently generate a correct *fi*-condition for a given control statement and
- how to efficiently check whether a control statement is partially or fully reversible.

In the remainder of this work, we propose a methodology which automatically solves these tasks for the designer.

IV. Proposed Solution

Here, we propose a methodology that relies on the symbolic simulation of a given HDL description to automatically address the challenges discussed above. Specifically, we utilize *predicate transformer semantics* that is based on *Hoare logic*. In the following, we will review the semantics of Hoare logic and, by this, provide the basis for the proposed solution. Afterwards, we detail how this semantics is actually applied for *fi*-generation as well as the automatic check for partial reversibility.

A. Hoare Logic and Rules for Symbolic Simulation

Hoare logic [9] is based on so-called *Hoare triples*, which have the form

$$\{P\}\ S\ \{R\}\ , \tag{2}$$

where S is a statement (or a sequence of statements) and P as well as R are pre- and post-conditions, respectively, i.e. assertions in predicate logic. This triple is to be interpreted as follows: If the pre-condition P holds, then executing the statement S ensures that the result R holds.

The logic comes with two basic axioms

$$\{P\}\ \text{skip}\ \{P\} \tag{3}$$

$$\{P[E/x]\}\ x \leftarrow E\ \{P\} \tag{4}$$

where $P[E/x]$ states that, in P, every free occurrence of the variable x is replaced by the expression E. The first axiom simply states that performing nothing does not change the system state (the pre-condition is equal to the post-condition). The second axiom formalizes a variable assignment as illustrated in the following example.

Example 4. *The Hoare triple*

$$\{x + 4 < 42\}\ y \leftarrow x + 4\ \{y < 42\}$$

describes the situation that, if $x + 4 < 42$ holds, the value of y is less than 42 when being assigned the value of $x + 4$.

Using these axioms, further rules can be derived. Two further rules are of particular importance for the proposed solution. The first one,

$$\frac{\{P\}\ S\ \{R'\} \quad \{R'\}\ S'\ \{R\}}{\{P\}\ S; S'\ \{R\}}, \tag{5}$$

defines the symbolic simulation of two consecutive statements (i.e. the composition of statements). The second one,

$$\frac{\{B \wedge P\}\ S\ \{R\} \quad \{\neg B \wedge P\}\ S'\ \{R\}}{\{P\}\ \text{if } B \text{ then } S \text{ else } S'\ \{R\}} \tag{6}$$

defines the symbolic formulation of an if-statement (note the additional assertion B and its negation $\neg B$ which are used to distinguish whether the *then*-block S or the *else*-block S' is simulated).

Building upon this, it is possible to symbolically simulate a given HDL description (according to [7]). More precisely, given a pre-condition $\{P\}$ and a sequence of statements S, a so-called *strongest post-condition* (denoted by $sp(S, P)$) can be derived, which describes all possible system states that can be reached by executing S from system states satisfying P. To this end, the axioms and rules from Eq. 3 to Eq. 6 are formulated as *transformation rules* for skip (S), assignment (A), composition (C), and if-statement (I):

$$sp(\text{skip}, P) := P \quad \text{(S)}$$

$$sp(x \leftarrow E, P) := \exists k : x = E[k/x] \wedge P[k/x] \quad \text{(A)}$$

$$sp(S; S', P) := sp(S', sp(S, P)) \quad \text{(C)}$$

$$sp(\text{if } B \text{ then } S \text{ else } S', P) := sp(S, P \wedge B) \vee sp(S', P \wedge \neg B) \quad \text{(I)}$$

Note that the variable k in (A) needs to be a free variable.

Example 5. *Consider the statement $S = x \leftarrow x + 2$ and the pre-condition $P = x \neq y$. Applying the rule (A) as defined above yields the strongest post-condition*

$$sp(x \leftarrow x + 2, x \neq y) = \exists k : x = k + 2 \wedge k \neq y \ ,$$

which can easily be reduced to $x - 2 \neq y$. This symbolically describes all possible system states that can be reached by executing $x \leftarrow x + 2$ from system states satisfying $x \neq y$.

Based on these rules, the challenges sketched in Section III can now be addressed.

B. Generation of fi-conditions

The *if*-condition of a conditional statement is a symbolic description of all system states which are supposed to enter the *then*-block. In a similar fashion, the *fi*-condition is a symbolic description of the system states which originated from executing the *then*-block. Hence, in order to automatically derive a *fi*-condition, it is sufficient to perform a symbolic simulation as described in the previous section. More formally, for a given *if*-condition B and a *then*-block composed of statements S_{then}, the desired *fi*-condition is equivalent to the strongest post-condition $sp(S_{\text{then}}, B)$.

However, in order to become applicable for the purposes considered here, some additional adjustments and assumptions have to be employed. In order to describe those properly, we first assume the notation of a reversible conditional statement to be

$$S_{\text{if}} := \textbf{if}\ (B)\ \textbf{then}\ S_{\text{then}}\ \textbf{else}\ S_{\text{else}}\ \textbf{fi}\ (). \quad (7)$$

where S_{if}, B, S_{then}, and S_{else} denote the entire conditional statement, the *if*-condition, the statements of the *then*-block, and the statements of the *else*-block, respectively. Note that the *fi*-condition is intentionally left empty as it is about to be generated. Furthermore, we assume that S_{then} and S_{else} are fully reversible (sequences of) statements which, however, may be empty (i.e. $S_{\text{then}} = \text{skip}$ or $S_{\text{else}} = \text{skip}$ is possible). Finally, we firstly assume that there are no nested if-statements[6].

The overall procedure for *fi*-generation is given in Algorithm 1. Initially, it is assumed that all system states are allowed to execute the statements; hence the pre-condition P is set to $true$ (line 1). Afterwards, all statements of the HDL description are traversed (line 2). If the currently considered statement S is *not* a conditional statement, P is accordingly updated using rules (S) or (A) (line 4). Rule (C) is implicitly employed by iteratively updating the condition P. Otherwise, the rule (I) is applied which splits the determination of the post-condition into two steps (lines 6/7), leading to a post-condition P_{then} obtained for the *then*-block and a post-condition P_{else} obtained for the *else*-block. The disjunction of both yields the updated description for P (line 8). Moreover, the post-condition P_{then} additionally yields the *fi*-condition for the currently considered if-statement and can accordingly be updated (line 9).

Algorithm 1: Generation of *fi*-conditions

Data: Reversible HDL description HDL given as a list of statements S
Result: Reversible HDL description with *fi*-conditions

```
1 P ← true
2 foreach S ∈ HDL do
3     if S is not an if-statement then
4         P ← sp(S, P)
5     else
6         P_then ← sp(S_then, P ∧ B)
7         P_else ← sp(S_else, P ∧ ¬B)
8         P ← P_then ∨ P_else
9         add P_then as fi-condition to S
```

Using Algorithm 1, *fi*-conditions can be automatically generated for many HDL descriptions. However, problems remain when nested if-statements occur. Then, two further issues have to be dealt with:

1) *Inner if-statements would be skipped*
   This is because an entire conditional statement S_{if} is always considered to be a single statement $S \in HDL$ as defined in Eq. 7. Hence, strictly following Algorithm 1 would indeed generate a *fi*-condition for S_{if} but, afterwards, move directly on with the next statement $S' \in HDL$ – leaving possible further if-statements within S_{then} and S_{else} unconsidered.
2) *Inner if-statements are subject to restricted system states*
   In order to correctly determine the strongest post-condition and, hence, the *fi*-condition, P is constantly updated in Algorithm 1. However, if a *fi*-condition for an inner if-statement is to be generated, the *if*-conditions of the respective outer if-statements have to be additionally employed. This is not yet incorporated in Algorithm 1.

Obviously, the first issue can easily be handled by modifying Algorithm 1 such that not only top level statements are traversed, but also all statements within the respective *then*- and *else*-blocks. Dealing with the second issue, however, requires a more elaborated adjustment. In order to describe that properly, let's consider again the underlying problem using the following example.

Example 6. *Consider the following conditional statements:*

```
if( ((3 <= x) && (x <= 6)) && (y = 1) ) then
  if( x = 6 ) then
    x −= 3;
  else
    x += y;
  fi (...)
else
  skip
fi (...)
```

[6]How to deal with nested if-statements is discussed after the main procedure has been introduced.

Algorithm 2: *fi*-generation for nested if-statements

Data: If-statement S_{if}, pre-condition P valid before S_{if}
Result: Returns post-condition of provided if-statement; recursively adds valid *fi*-conditions to all if-statements visited in the process (including itself)

```
/* Initialize block conditions                    */
```
1 $P_{\text{then}} \leftarrow P \wedge B$
2 $P_{\text{else}} \leftarrow P \wedge \neg B$
```
/* Iterate over statements                        */
```
3 **foreach** $S \in S_{\text{then}}$ **do**
4 **if** S *is if-statement* **then**
5 $P_{\text{then}} \leftarrow$ result of Algorithm 2 with S and P_{then}
6 attach P_{then} as *fi*-condition to S
7 **else**
8 $P_{\text{then}} \leftarrow sp(S, P_{\text{then}})$
9 **foreach** $S \in S_{\text{else}}$ **do**
10 **if** S *is if-statement* **then**
11 $P_{\text{else}} \leftarrow$ result of Algorithm 2 with S and P_{else}
12 attach P_{else} as *fi*-condition to S
13 **else**
14 $P_{else} \leftarrow sp(S, P_{\text{else}})$
15 **return** $P_{\text{then}} \vee P_{\text{else}}$

The pre-condition P to be applied for the innermost then*-block is not just $x = 6$, but must additionally take the condition from the outer if-statement (i.e. $3 \leq x \wedge x \leq 6 \wedge y = 1$) into account*[7]*. This becomes particularly obvious in this example, since the inner if-statement alone is partially reversible (in fact, it represents the same situation as discussed in Example 3). Only due to the additional consideration of the condition from the outer if-statement, a* fi*-condition for a fully reversible HDL description can be generated.*

To solve with this problem, we revise the part of Algorithm 1 that is responsible for dealing with if-statements (lines 6-8). Instead of directly deriving the strongest post-conditions with fixed $P \wedge B$ and $P \wedge \neg B$, the procedure shown in Algorithm 2 is called. Also here, P is updated depending on the respectively considered if-statement and its condition B (see lines 1/2). But in contrast to Algorithm 1, P is further updated whenever the procedure encounters another if-statement. This is realized by recursively calling Algorithm 2 as shown in lines 5/11. The result of this recursive call derives the *fi*-condition for the currently considered if-statement, whereas the overall return value is used by Algorithm 1 to update the current system state P. Following these schemes, also cases as discussed in Example 6 are fully supported.

C. Check for Partial Reversibility

As discussed in Section III, checking whether a given reversible HDL description indeed is fully reversible remains the second challenge designers have to address when creating control logic for reversible circuits and systems. A (sequence of) statements S is partially reversible, if there exist two different input states whose execution of S yields the same output state. Since assignment statements are by definition fully reversible, they can never be the reason for a partial reversible HDL description. In contrast, conditional statements allow for the execution of two different sequences of statements (the *then*-block and the *else*-block) and, hence, may indeed transform two different input states to the same output states (as illustrated in Example 3).

In order to check that, the method for *fi*-generation as introduced above can be re-used and accordingly extended. Recall that a (generated) post-condition $sp(S_{\text{then}}, P \wedge B)$ is a symbolic representation of all system states that originate from the execution of all statements in the *then*-block. Accordingly, a (generated) post-condition $sp(S_{\text{else}}, P \wedge \neg B)$ is a symbolic representation of all system states that originate from the execution of all statements in the *else*-block. Hence, if there exists an output state which originated from two different input states, the conjunction

$$sp(S_{\text{then}}, P \wedge B) \wedge sp(S_{\text{else}}, P \wedge \neg B) \tag{8}$$

must evaluate to true.

This constitutes a typical *satisfiability problem* (SAT, cf. [4]): If an assignment to all variables of an HDL description exists which satisfies Eq. 8, a system state showing the partial reversibility can be derived. If it has been shown that no such assignment exists, the HDL description has been proven to be fully reversible. In order to conduct those checks, various powerful solving engines (so called *SAT solvers*) have been proposed in the past and can be utilized for this purpose. To this end, Eq. 8 has to be converted into a proper format and, afterwards, simply passed to a SAT solver.

Example 7. *Consider again the conditional statements from Example 3. Applying the method for* fi*-generation as introduced above for $sp(S_{\text{then}}, x = 6)$ yields the post-condition $x = 3$. For $sp(S_{\text{else}}, x \neq 6)$, the post-condition $x - y \neq 6$ is generated. Passing the conjunction of both conditions, i.e.*

$$x = 3 \wedge x - y \neq 6,$$

to a SMT solver, yields a satisfying assignment $x = 3$ and $y = 0$. This assignment indeed represents an output state showing the non-reversibility of the conditional statement (as already discussed in Example 3).

V. Experimental Evaluation

In the previous section, we proposed a methodology to both automatically check whether a conditional statement given in a reversible HDL is indeed fully reversible and automatically derives a corresponding (correct) *fi*-condition. In order to evaluate the performance of the obtained hypotheses, the proposed approach has been thoroughly tested. In this section, the obtained results are summarized and discussed.

For this purpose, we implemented the proposed approach on top of RevKit [15]. To conduct the check for partial reversibility as described in Section IV-C, the SAT solver *Z3* [5] has been utilized. All evaluations have been conducted on an Intel Core 2 Duo machine with 2.4 GHz and 4 GB of main memory.

As benchmarks, we applied a variety of HDL descriptions known from the literature. This includes different versions of an arithmetic logic unit (*alu*, *cpu_alu*, *simple_alu*), an arbiter that allocates access to shared resources (*arb*), as well as a control unit (*cpu_control_unit*), a program counter (*cpu_pc*), and a register bench (*cpu_register*) of a CPU[8]. Besides that, special examples have been implemented that have been

[7] Note that, in a similar fashion, the negation of $3 \leq x \wedge x \leq 6 \wedge y = 1$ must be taken into account, if an inner if-statement existed in the *else*-block.

[8] All these HDL descriptions have been taken from [19].

explicitly constructed to evaluate certain corner-case scenarios of the approach. This includes benchmarks with if-statements including an empty *then*-block (*empty_then*) and an empty *else*-block (*empty_else*), nested if-statements including various read/write statements in the corresponding *then*/*else*-blocks (*nested_if*), examples realizing decoders/encoders (*decoder*) or determining the maximum value of a variable (*max_value*), and further examples representing corner cases (*misc*). For selected descriptions, also corresponding versions including a partially reversible if-statement to be detected have been considered (denoted by the addition *_partial*).

Table I summarizes the obtained results. The first column gives the name of the respective HDL descriptions (following the naming convention as introduced above), while the three following columns provide the total number of statements, the total number of if-statements, and the maximal depth of nested if-statements for each example. Column REVERSIBLE? states whether an HDL description has been proven to be fully reversible (✓) or not (×, i.e. the HDL description is partially reversible). The final column provides the total runtime (in CPU seconds) required to check the *entire* HDL description for partial reversibility and to generate *all fi*-conditions.

The results confirm the benefits of the proposed approach. For an error-prone and time-consuming design task, which has been conducted manually thus far, an automatic and efficient solution has been presented. In fact, all desired *fi*-conditions can be generated in negligible run-time, i.e. within up to a bit more than a CPU second only. Moreover, at the same time, it can be checked whether the given HDL descriptions are indeed fully reversible. Because of this, we were able to prove that the HDL descriptions used in literature thus far are indeed fully reversible (see column REVERSIBLE? for the first 10 HDL descriptions in Table I).

TABLE I: Experimental Evaluation

HDL DESCRIPTION	STMTS.	IF-STMTS.	MAX. DEPTH	REVERSIBLE?	TIME
alu_233	7	3	2	✓	0.08
alu_flat_23	12	4	0	✓	0.04
cpu_alu_16bit_242	80	25	17	✓	0.48
cpu_alu_32bit_243	80	25	17	✓	0.36
lu_238	8	3	2	✓	0.03
simple_alu_234	7	3	2	✓	0.03
arb8_235	17	8	7	✓	0.08
cpu_control_unit_244	41	7	1	✓	0.12
cpu_pc_246	5	2	1	✓	0.02
cpu_register_247	3	1	0	✓	0.01
empty_then_partial	3	1	1	×	0.01
empty_then	3	1	0	✓	0.01
empty_else_partial	3	1	1	×	0.01
empty_else	3	1	1	✓	0.01
nested_if1_partial	15	7	2	×	1.10
nested_if2	18	4	2	✓	0.04
nested_if3	7	3	1	✓	0.03
decoder1_partial	9	4	3	×	0.04
decoder2	9	4	3	✓	0.04
decoder3_partial	17	8	7	×	0.09
decoder4	17	8	7	✓	0.08
decoder5_partial	15	7	6	×	0.07
max_value	15	7	2	✓	0.08
misc1_partial	6	2	0	×	0.03
misc2_partial	3	1	1	×	0.01
misc3	3	1	0	✓	0.02
misc4_partial	10	1	0	×	0.35
misc5_partial	6	2	0	×	0.03
misc6_partial	9	3	0	×	0.03

VI. CONCLUSIONS

In this work, we considered the generation of control logic in hardware description languages following the reversible computing paradigm. Here, obstacles occur since (1) corresponding descriptions may not necessarily be reversible and (2) conditional statements in reversible logic require a *fi*-condition in addition to the established *if*-condition. Both issues resulted in new design tasks which have been addressed manually thus far. We proposed a solution which applies symbolic simulation as well as solvers for satisfiability problems in order to automatically tackle these tasks. Experimental evaluations on HDL descriptions used in literature as well as explicitly designed corner cases confirmed the applicability of the proposed methods. Using the approaches presented in this work, the required *fi*-conditions can automatically be generated in negligible run-time. Moreover, automatic checks whether an HDL description indeed is reversible are possible. By this, the important tasks for the design of reversible circuits and systems eventually got automated. Methods like this will be an important part of future design tools.

ACKNOWLEDGMENTS

This work has partially been supported by the European Union through the COST Action IC1405.

REFERENCES

[1] IEEE Standard VHDL Language Reference Manual Amendment 1: Procedural Language Application Interface, 2007.

[2] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *Physical Review A*, 52(5):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*, 483:187–189, 2012.

[4] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability*. IOS Press, 2009.

[5] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. Z3 is available at http://z3.codeplex.com/.

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

[7] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer, 1990.

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

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[10] IEEE Std. 1800. *IEEE SystemVerilog*, 2005.

[11] C.-C. Lin and N. K. Jha. RMDDS: Reed-muller decision diagram synthesis of reversible logic circuits. *J. Emerg. Technol. Comput. Syst.*, 10(2):14, 2014.

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

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

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

[15] M. Soeken, S. Frehse, R. Wille, and R. Drechsler. RevKit: A toolkit for reversible circuit design. In *Workshop on Reversible Computation*, pages 69–72, 2010. RevKit is available at http://www.revkit.org.

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

[17] R. Wille and R. Drechsler. BDD-based synthesis of reversible logic for large functions. In *Design Automation Conf.*, pages 270–275, 2009.

[18] 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*, pages 1036–1041, 2012.

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

[20] 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.*, 53:39–53, 2016.

[21] R. Wille, M. Soeken, N. Przigoda, and R. Drechsler. Exact synthesis of Toffoli gate circuits with negative control lines. In *Int'l Symp. on Multi-Valued Logic*, pages 69–74. IEEE, 2012.

[22] T. Yokoyama, H. B. Axelsen, and R. Glück. Principles of a reversible programming language. In *Conference on Computing Frontiers. Proceedings*, pages 43–54. ACM, 2008.