Efficient Simulation-based Debugging of Reversible Logic

Stefan Frehse Robert Wille Rolf Drechsler
Institute for Computer Science
Bibliothekstraße 1
28359 Bremen, Germany
{sfrehse,rwille,drechsle}@informatik.uni-bremen.de

***Abstract*—Reversible logic has become an active research area due to its various applications in emerging technologies, like quantum computing, low power design, optical computing, DNA computing, or nanotechnologies. As a result, complex reversible circuits containing thousands of gates can be efficiently synthesized, today. However, this also increases the probability of design errors. While for the detection of errors already a couple of simulation-based or formal verification techniques have been proposed for reversible logic. Research in the domain of debugging is still at the beginning.**

In this paper, we present an automatic debugging approach for reversible logic which is based on simulation. We show that a particular error in a gate always requires a counterexample leading to a concrete gate input pattern. By simulating all counterexamples and checking for these input patterns, irrelevant gates (i.e. gates that do not contain an error) can be excluded. Experiments show, that applying the proposed approach leads to speed-ups of up to five orders of magnitude. Furthermore, the number of error candidates can be reduced in comparison to previous work.

I. Introduction

While nowadays circuit technologies more and more start to suffer from the increasing miniaturization and the exponential growth of the number of transistors, reversible logic [1], [2], [3] offers a promising alternative. Here, functions are realized that map each input pattern to a unique output pattern (i.e. bijections are realized). The resulting reversibility enables applications particularly in areas like e.g. quantum computation [4] or low power design [5], but also in optical computing [6], DNA computing [2], and nanotechnologies [7].

However, in contrast to traditional CMOS circuits, reversible logic is subject to some important restrictions. For example, fanout and feedback are not allowed [4]. Therefore, a complete new gate library and circuit structure was introduced [3]. More precisely, reversible circuits are cascades of reversible gates. This also affects the design flow which has to be reorganized and where single steps need considerable modifications to support reversible logic.

In the last years, researchers particularly focused on synthesis (see e.g. [8], [9], [10], [11], [12]). While at the beginning, synthesis of reversible logic was only possible for very small functions (i.e. for functions with up to 30 variables), in the meanwhile approaches have been introduced that handle 100 and more variables [12]. As a consequence, also the resulting reversible circuits increased in their size and complexity, so that the focus more and more moves from synthesis also to further steps in the design flow such as simulation [13], [14], optimization [15], [16], testing [17], [18], [19], and formal verification [20], [21], [22]. In particular, automatic methods for checking the correctness become more important as with increasing circuit sizes this cannot be manually ensured any longer.

However, while methods for simulation, testing, or verification can only be used to detect the existence of errors, they provide no support to locate the source of an error. Thus, recently also a first approach for automatic debugging of errors has been proposed [23]. Here, given an erroneous circuit and a set of counterexamples, a set of gates (so called *error candidates*) is returned, whose replacements with other gates fix the counterexamples, The debugging problem has been thereby encoded as an instance of Boolean satisfiability (inspired by circuit debugging for non-reversible circuits [24]) and solved by an efficient SAT solver [25].

In this paper, we introduce an alternative debugging approach that relies on the simulation of counterexamples. An observation is thereby exploited stating that a particular error in a gate always requires a counterexample leading to a concrete gate input pattern. Thus, by simulating all available counterexamples and checking for these input patterns, irrelevant gates (i.e. gates that definitely do not contain an error) can be excluded.

This leads to the following advantages:

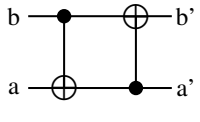
- The number of gates which have to be further considered is significantly reduced. Sometimes, already this leads to a single error candidate, i.e, the concrete error location.
- Due to the simulation-based nature of the approach this reduction can be achieved in very low run-time.
- The proposed approach can be additionally used as preprocessor for further debugging methods. More precisely, in a first step the number of gates to be considered is significantly reduced by the proposed simulation-based approach. If then still a notable number of error candidates remains, this set can be further refined e.g. by the approach from [23]. In doing so, for the first time debugging becomes feasible even for circuits containing hundreds of thousands of gates.

Our experiments confirm these advantages as speed-ups of up to five orders of magnitude are achieved. Furthermore, the number of error candidates can be significantly reduced in many cases in comparison to previous presented SAT-based approach [23].

0195-623X/10 $26.00 © 2010 IEEE
DOI 10.1109/ISMVL.2010.37

a	b	a'	b'
0	0	0	0
0	1	1	1
1	0	0	1
1	1	1	0

(a) Truth table



(b) Toffoli circuit

Fig. 1. Reversible logic

The remainder of this paper is structured as follows. First, the basics needed in the rest of this paper are introduced in Section II. The general idea including the motivating observations is proposed in Section III while the resulting algorithm is described in Section IV. All descriptions are thereby done by means of the missing control error model. How further error models can be applied is described in Section V. Finally, experimental results and conclusions are given in Section VI and Section VII, respectively.

II. Background

To keep the remaining paper self-contained, this section briefly introduces the basics of reversible logic, the debugging problem, and the automatic debugging approach of [23].

A. Reversible Logic

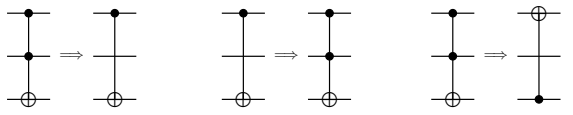
Reversible Logic realizes functions $f : \mathbb{B}^n \rightarrow \mathbb{B}^n$ over the variables $X \in \mathbb{B}^n$, that map each input vector to a unique output vector. A reversible circuit G consists of a cascade of reversible gates g_i, i.e. $G = g_0 g_1 \dots g_{d-1}$, where d represents the number of gates. In the past, several reversible gates have been introduced. In this work, circuits composed of *Multiple control Toffoli gates* [3] are considered. A multiple control Toffoli gate $\mathrm{TOF}(C,t)$ consists of a set $C = \{x_{i_1}, \cdots, x_{i_k}\} \subset X$ of *control lines* and a single *target line* $t = \{x_j\}$ with $C \cap t = \emptyset$. The gate performs the mapping $(x_1, \dots, x_n) \mapsto (x_1, \cdots, x_{j-1}, x_j \oplus \wedge_{l=1}^{k} x_{i_l}, x_{j+1}, \cdots, x_n)$, i.e. the value of the target line is inverted if all control lines are assigned to 1.

Example 1: Fig. 1 shows a Toffoli circuit representing the function specified by the given truth table. The circuit is drawn in standard notation (see e.g. [4]) and includes $d = 2$ gates.

In the following, the notation G_i^j with $i < j$ is used to represent the sub-circuit $g_i g_{i+1} \dots g_j$ of G. Furthermore, $G[\vec{v}]$ evaluates an input vector $\vec{v} \in \mathbb{B}^n$ on G, i.e. $G[\vec{v}] = f(\vec{v})$, iff G realizes $f : \mathbb{B}^n \rightarrow \mathbb{B}^n$.

B. The Debugging Problem

During the design of reversible logic, errors may occur e.g. due to bugs in synthesis as well as optimization tools, or manual modifications, respectively. Errors can be detected e.g. by various verification methods like [20], [21], [22], [13], [14] leading to a *counterexample* showing the erroneous behavior. However, to find the source of an error, the circuit must be *debugged* – often a manual and time-consuming process.



(a) *Missing* control (b) *Additional* control (c) *Wrong target*

Fig. 2. Considered errors

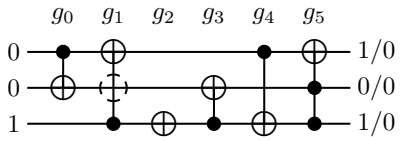


Fig. 3. Network 3_17 with a missing control error

Thus, recently an automatic approach has been introduced that support the designer to reduce the possible error locations [23]. Therefore, error models are used abstracting frequently occurred errors. In this work, we distinguish between three single error models:

Definition 1: Let $g = \mathrm{TOF}(C,t)$ be a Toffoli gate of a circuit G. Then,

1) a *Missing Control Line Error* (MCE) appears if g is replaced by $\mathrm{TOF}(C',t)$, whereas $C' = C \setminus \{x_i\}$ with $x_i \notin C \cup t$ (i.e. a control line is removed),
2) an *Additional Control Line Error* (ACE) appears if g is replaced by $\mathrm{TOF}(C',t)$, whereas $C' = C \cup \{x_i\}$ with $x_i \notin C \cup t$ (i.e. a control lines is added), and
3) a *Wrong Target Line Error* (WTE) appears if g is replaced by $\mathrm{TOF}(C',t')$, whereas $t' \neq t$ and C' may be different from C (i.e. g is replaced by a gate with another target line).

Fig. 2 illustrates the respective models.

Given an error model together with an erroneous circuit G as well as a set of counterexamples, the goal of automatic debugging approaches is to determine a set of *error candidates* that may explain the erroneous behavior of G. An error candidate is thereby a set of gates g_i that can be replaced by other gates (according to the error model) such that for each counterexample the correct output values result. The size of an error candidate is given by the number of gates (later denoted by k).

Example 2: Fig. 3 shows an erroneous circuit G together with a counterexample (applied to the inputs of G). At the outputs, the wrong values (determined by the counterexamples) as well as the expected values are annotated. For this example, gate g_1 is an error candidate since replacing g_1 with another gate (namely a gate with one additional control line) would correct the output values. In this case, the counterexample detects a missing control error.

The set of error candidates determined, the debugging process can be significantly accelerated since only a small part of the circuit must be inspected. Moreover, in many cases determining error candidates directly leads to the error position.

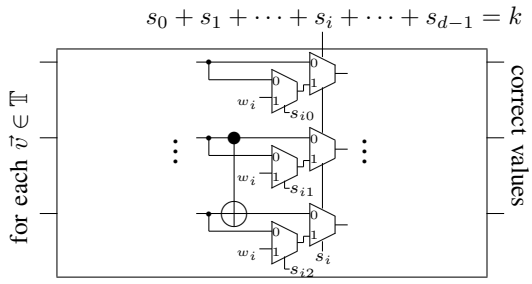


Fig. 4. SAT-based debugging approach

C. SAT-based Debugging

To determine error candidates, recently an automatic approach based on *Boolean satisfiability* (SAT) has been proposed in [23]. Here, a SAT instance as depicted in Fig. 4 is constructed. For each counterexample, a copy of the circuit is created, whose inputs are assigned to values provided by the counterexamples (denoted by $\vec{v} \in \mathbb{T}$). The outputs are assigned to the correct values. Furthermore, each gate g_i is extended by additional logic: First, a multiplexor with select line s_i is added. If s_i is assigned to 0, then the output values of gate g_i are passed through, i.e. the gate remains unchanged. Otherwise (if $s_i = 1$), an output of an arbitrary different Toffoli gate is generated using a new (unrestricted variable) w_i as well as further multiplexors with s_{ij} $(0 \leq j < n)$ as select lines. By the additional constraint $s_{i0} + \cdots + s_{in-1}$ it is ensured that in fact at least one line may be different from the input of the respective gate while all other lines must pass through. Altogether, this formulation allows to modify the behavior of a gate g_i (and therewith to achieve the correct output values) just by assigning $s_i = 1$.

Additionally, the number of select signals s_i that are set to 1 is limited to k. Starting with $k = 1$, k is iteratively increased until the instance becomes satisfiable. In doing so, only the behavior of potential error candidates is modified while the remaining (probably correct) gates work as usual. Furthermore, the select signal s_i is used for a gate g_i with respect to all duplications (i.e. for all counterexamples). Also this ensures that the behavior of a gate g_i is only modified if this leads to correct output values for *all* counterexamples. Then, each satisfying assignment yields an error candidate of size k. All gates with s_i set to 1 are contained in the set of error candidates.

For a more detailed description of the SAT-based debugging for reversible logic, we refer to [23].

III. Excluding Error Candidates Using Simulation

As shown in the last section, SAT-based debugging additional logic requires to *all* gates g of the circuit G. In particular for large circuits, this results in a significant overhead[1]. In this section we show how (under the single error assumption) gates can be excluded from consideration by applying (i.e. simulating) the available counterexamples and checking for appropriate input patterns of g. We motivate and describe the approach by the following observations for the case of missing control line errors. In Section V, the respective findings are also extended to other error models.

Observation 1: Let G be a circuit and $g_i(C,t)$ be a reversible gate of G. To identify a missing control line $x_m \in X \setminus (C \cup t)$ in g_i one of the following input patterns has to be applied to g_i:

$$P_{\text{MCE}}(g_i, x_m) \quad = \quad \big(x_0, \cdots, x_{n-1}\big) \in \{0,1,-\}^n$$
$$\text{with}$$
$$\forall x_i \in C.x_i = 1 \wedge x_m = 0 \wedge \forall x_j \in X'.x_j = -$$

where $X' = X \setminus (C \cup \{x_m\})$.

In other words, to identify a single missing control error in g_i all control lines C of g_i must be assigned to 1, while the (potential) missing control line x_m must be assigned to 0 (the remaining lines are don't care). Only these input patterns detect the difference between the correct and the erroneous circuit.

Example 3: Consider the circuit G shown in Fig. 3. The marked missing control line of g_1 is only identified if one of the patterns $P_{\text{MCE}}(g_1, \{x_1\}) = (-, 0, 1)$ (i.e. $(0,0,1)$ or $(1,0,1)$) are applied to g_1.

As a result of this observation, if g_i includes a single missing control line error, then *all* counterexamples must stimulate g_i by $P_{\text{MCE}}(g_i, x_m)$ since only these input patterns show the difference between the correct and the erroneous circuit. Conversely, if at least one counterexample does not include such a pattern, x_m can not be a missing control line of g_i. If this can be shown for all x_m of g_i, then g_i is not an error candidate and thus can be excluded from consideration. More formally:

Lemma 1: Let G be a erroneous circuit with the gate cascade $G = g_0 \dots g_{d-1}$ and $\mathbb{T}$ be the set of counterexamples. The gate $g_i(C,t)$ is not a single error candidate with respect to the missing control error model, if

$$\forall x_m \in X \setminus (C \cup t). \exists \vec{v} \in \mathbb{T}. G_0^{i-1}[\vec{v}] \neq P_{\text{MCE}}(g_i, x_m).$$

Proof: Without loss of generality, let G be a circuit and x_m a possible missing control line at gate g_i. To detect this error, (1) all control lines of g_i have to be assigned to 1 and (2) another line of g_i (the missing control line) has to be assigned to 0. That is, for all counterexamples $\vec{v} \in \mathbb{T}$, $G_0^{i-1}[\vec{v}] = P_{\text{MCE}}(g_i, x_m)$ must hold. Conversely, if there is at least one counterexample $\vec{w}$ with $G_0^{i-1}[\vec{w}] \neq P_{\text{MCE}}(g_i, x_m)$, then this counterexample detects another error. But since single errors are assumed, no missing control line x_m in g_i is possible. If such a case occurs for all possible missing control lines x_m of g_i, then g_i is not an error candidate. ■

Example 4: Again, consider the circuit G from Fig. 3 and additionally the set of counterexamples $\mathbb{T} = \{(0,0,1),(1,1,1)\}$. By Lemma 1, it can be shown that gate g_0 can not have a missing control line x_2 since such an error must be detected by patterns $P_{\text{MCE}}(g_0, x_2) = (1, -, 0)$. Obviously, $G_0^0[(0,0,1)] = (0,0,1) \neq (1,-,0)$, i.e. at least one counterexample does not identify such an error. Since additionally no further missing control lines are possible for

[1]Only in some few cases (i.e. if a restricted error model is assumed and if additionally the number of *all* counterexamples are known) some extra logic can be omitted as also discussed in [23].

Algorithm 1: patternMatchingMissingControl

Data: Erroneous circuit $G = g_0 \cdots g_{d-1}$, counterexamples $\mathbb{T}$
Result: Set of error candidates

1 $G_{\text{mc}} = \{g_0, \cdots, g_{d-1}\}$;
2 $M =$ newVector;
3 **foreach** $g_i(C,t) \in G$ **do**
4 $A = \emptyset$;
5 **foreach** $x_m \in X \setminus (C \cup t)$ **do**
6 $A = A \cup \{(g_i, x_m)\}$;
7 $M[i] = A$;
8 **foreach** $\vec{v} \in \mathbb{T}$ **do**
9 $\vec{w} = \vec{v}$;
10 **for** $i = 0$ *to* $d-1$ **do**
11 $g_i(C,t) = G_i^{i+1}$;
12 **foreach** $x_m \in X \setminus (C \cup t)$ **do**
13 **if** $\vec{w} \neq P_{\text{MCE}}(g_i, x_m)$ **then**
14 $M[i] = M[i] \cap \{(g_i, x_m)\}$;
15 **if** $M[i] = \emptyset$ **then**
16 $G_{\text{mc}} = G_{\text{mc}} \cap g_i$
17 $\vec{w} = G_i^{i+1}[\vec{w}]$;
18 **return** (G_{mc})

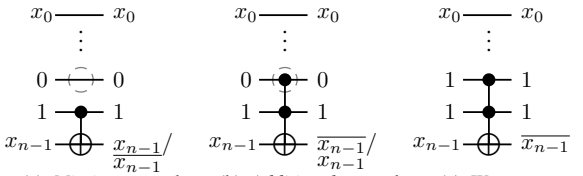
gate g_0, this gate can be completely excluded as an error candidate.

Using these observations, under the missing control line error model many gates can be excluded as possible error candidates just by applying (i.e. simulating) the available counterexamples and checking whose respective input pattern of each g_i matches $P_{\text{MCE}}(g_i, x_m)$ for all possible missing control line locations x_m. Before this observation is extended for other error models (namely additional line errors and wrong target line errors), the next section describes the algorithm that applies Lemma 1 for efficient exclusion of error candidates.

IV. Algorithm

This section presents the algorithm applying Lemma 1 to exclude gates that do not have to be considered any longer during debugging. The general idea is based on the observations described in the last section, i.e. counterexamples are simulated while for each gate it is checked whether the input of the gate matches the respective patterns described above. The complete procedure is given in Algorithm 1 for the missing control error model (the application for the other error models is similar as discussed in Section V).

The algorithm gets the erroneous circuit G and a set of counterexamples $\mathbb{T}$ as input. First, for each gate g_i of the circuit all possible missing control lines errors x_m are collected and stored as a tuple (g_i, x_m) in $M[i]$ (lines 3–7). Then, the available counterexamples are simulated and it is checked for which gates the respective patterns do not match (lines 8–17) according to Lemma 1. Each time, a pattern does not match (i.e. each time a particular error is not identified by the current counterexample), the respective missing control line error is removed from $M[i]$. If all missing controls for one gate g_i are removed (i.e. $M[i]$ is empty), then g_i is no longer an error candidate under the missing control error. In this case, for each possible missing control line at least one counterexample exists which does not identify this error. After all available counterexamples have been considered, the remaining set of error candidates is returned.



(a) *Missing* control (b) *Additional* control (c) *Wrong target*

Fig. 5. Pattern for respective errors

For simulation, a parallel simulation engine for reversible circuits has been used. This has been implemented on a 64-bit machine and thus allows to simulate 64 counterexamples in parallel.

As the experiments in Section VI show, already this leads to a significant reduction of the gates that have to be considered during debugging. Moreover, the proposed approach can be additionally used as pre-processor e.g. for the SAT-based debugging formulation described in Section II-C. Then, in a first step the number of gates to be considered is significantly reduced by the simulation-based approach. If then still a notable number of error candidates remains, this set can be further refined e.g. by the approach from [23].

V. Supporting Other Error Models

So far, the proposed approach has been introduced for missing control errors only. However, the general idea can also be applied to other error models. Then, only a slight modification of Algorithm 1 is necessary. More precisely, another set of patterns to be checked must be applied. The following observation gives the respective patterns for additional control line errors and wrong target line errors, respectively.

Observation 2: Let G be a circuit and $g_i(C,t)$ be a reversible gate of G. To detect an additional control line $x_a \in C$ in g_i one of the following input patterns has to be applied to g_i:

$$P_{\text{ACE}}(g_i, x_a) = \big(x_0, \cdots, x_{n-1}\big) \in \{0,1,-\}^n \text{ with } \forall x_i \in C \setminus \{x_a\}.x_i = 1 \wedge x_a = 0 \wedge \forall x_j \in X'.x_j = -$$

where $X' = X \setminus (C \cup \{x_a\})$. That is, in contrast to missing control errors one of the control variables must be set to 0.

To detect a wrong target line in g_i, the following input patterns must be applied:

$$P_{\text{WTE}}(g_i) = \big(x_0, \cdots, x_{n-1}\big) \in \{0,1,-\}^n \text{ with } \forall x_i \in C.x_i = 1 \wedge \forall x_j \in X \cup t.x_j = -$$

Fig. 5 shows an example for each of the considered error models, respectively.

TABLE I
EXPERIMENTAL RESULTS FOR CONTROL LINE ERRORS

ERRONEOUS CIRCUIT			MISSING CONTROL LINE ERRORS									ADDITIONAL CONTROL LINE ERRORS								
				SIM.-BASED[a]		+SAT [23]		ONLY SAT [23]		QUALITY			SIM.-BASED		+SAT [23]		ONLY SAT [23]		QUALITY	
NAME	L	D	$\lvert\mathbb{T}\rvert$	REMAIN.	EC	TIME [s]	EC	TIME [s]	ΔEC	SPUP	$\lvert\mathbb{T}\rvert$	REMAIN.	EC	TIME [s]	EC	TIME [s]	ΔEC	SPUP
3_17	3	6	2	2	66.67%	2	< 0.01	2	< 0.01	0	–	2	1	83.33%	1	< 0.01	2	< 0.01	1	–
4_49	4	16	4	1	93.75%	1	< 0.01	2	< 0.01	1	–	2	6	62.50%	2	< 0.01	2	< 0.01	0	–
4gt4	5	6	8	1	83.33%	1	< 0.01	3	< 0.01	2	–	4	4	33.33%	3	< 0.01	3	< 0.01	0	–
4mod5	5	9	8	4	55.56%	1	< 0.01	4	< 0.01	3	–	4	1	88.89%	–	< 0.01	4	< 0.01	–	–
ham3	3	5	4	1	80.00%	1	< 0.01	1	< 0.01	0	–	2	2	60.00%	1	< 0.01	1	< 0.01	0	–
ham7	7	23	16	5	78.26%	2	< 0.01	2	< 0.01	0	–	8	6	73.91%	1	< 0.01	2	< 0.01	1	–
ham15	15	132	40	5	96.21%	1	0.01	29	0.05	28	5.00	40	41	68.94%	11	0.11	29	0.05	18	0.45
hwb4	4	17	8	1	94.12%	1	< 0.01	2	< 0.01	1	–	4	2	88.24%	1	< 0.01	2	< 0.01	1	–
hwb5	5	55	16	1	98.18%	1	< 0.01	7	0.01	6	> 1	8	4	92.73%	1	< 0.01	3	0.01	2	> 1
hwb6	6	126	16	1	99.21%	1	< 0.01	3	0.02	2	> 1	8	2	98.41%	1	< 0.01	1	0.02	0	> 1
hwb7	7	289	8	1	99.65%	1	< 0.01	3	0.10	2	> 1	4	9	96.89%	2	< 0.01	3	0.04	1	> 1
hwb8	8	637	32	1	99.84%	1	0.01	2	0.52	1	52.00	16	11	98.27%	2	0.12	2	0.37	0	3.08
hwb9	9	1544	32	4	99.74%	1	0.11	2	1.11	1	10.00	16	4	99.74%	1	0.02	2	1.88	1	94.00
plus63mod4096	12	429	40	3	99.30%	1	0.05	26	0.82	25	16.04	32	263	38.69%	26	1.15	47	0.75	21	0.65
plus63mod8192	13	492	32	3	99.39%	1	0.04	92	2.01	91	50.25	9	338	31.30%	73	1.24	92	1.65	19	1.33
plus127mod8192	13	910	16	29	96.81%	1	0.04	202	7.85	201	196.25	8	648	28.79%	170	2.90	204	6.53	34	2.25
urf1	9	11554	40	1	99.99%	1	0.20	2	28.52	1	142.60	8	17	98.88%	1	0.18	1	24.98	0	138.78
urf2	8	5030	40	1	99.89%	1	0.08	3	12.02	2	150.25	32	1	99.97%	1	0.05	4	12.24	3	244.80
urf3	10	26468	40	1	$>$ 99.99%	1	0.48	4	1126.07	3	2345.98	32	1	99.99%	1	4.96	4	509.25	3	102.67
urf5	9	10276	40	1	99.99%	1	0.17	4	51.52	4	303.06	40	35	99.66%	2	3.87	4	23.44	2	6.06
prng9	10	7617	40	7	99.91%	1	0.58	5	23.06	4	39.76	40	1	99.99%	1	0.15	7	34.81	6	232.07
prng10	10	16997	40	1	99.99%	1	0.31	3	175.45	2	565.97	40	1	99.99%	1	0.34	6	56.86	5	167.24
prng11	11	36910	40	1	$>$ 99.99%	1	0.70	1	266.53	0	380.76	40	1	$>$ 99.99%	1	0.80	1	1100.13	0	1375.20
prng12	13	79225	40	1	$>$ 99.99%	1	1.76	2	1113.92	1	632.91	40	1	$>$ 99.99%	1	1.88	2	1200.20	1	638.40
prng15	15	716934	40	44	99.99%	1	72.21	716934	> 5000	716933	> 69.24	40	5	$>$ 99.99%	5	18.93[b]	716934	> 5000	716929	> 264.13

[a] The run-time of the simulation-based approach is negligible (always less than 20 CPU seconds) and thus is omitted due to space restrictions.
[b] The SAT-based debugging approach has been aborted. Thus, the run-time of the simulation-based approach only is given.

VI. EXPERIMENTAL RESULTS

The proposed approach was implemented in C++ and evaluated on an Intel Xeon 3Ghz (32GB RAM) running Linux. As described above, a 64-bit parallel simulation engine was applied to simulate the counterexamples. Several circuits from [26] have been used where different single errors (according to the considered error models) have be randomly injected. The counterexamples have been generated using the SAT-based equivalence checker proposed in [22].

In the experiments, the proposed pattern matching approach has been evaluated as stand-alone method first. Afterwards, SAT-based debugging as described in Section II-C has been additionally applied to the remaining set of gates. Finally, for comparison the SAT-based debugging approach has been solely applied to the benchmarks. The time out for all evaluations has been set to 5000 CPU seconds. This section presents the obtained experimental results.

A. Control Lines Errors

In a first evaluation, the efficiency of the proposed approach is considered for control line errors. Table I provides the results of the evaluations, broken down by missing control line errors and additional control line errors, respectively. The first columns give the name (NAME), the number of circuit lines (L), and the number of gates (D) of the circuit, respectively. Column $|\mathbb{T}|$ denotes the number of counterexamples that have been applied. The following columns (SIM.-BASED) provide the results obtained by the proposed simulation-based approach, i.e. the number of error candidates and the resulting reduction Column +SAT shows the number of error candidates and the run-time if additionally SAT-based debugging is applied. The results obtained if SAT-based debugging is solely applied (i.e. without simulation-based approach as pre-processing) are given in Column ONLY SAT. Finally, a comparison of SIM.-BASED + SAT and ONLY SAT with respect to number of resulting error candidates and the run-time is given in column ΔEC and column SPUP, respectively.

The results clearly show that the proposed pattern matching approach is quite effective. Already with this method the number of gates that still have to be considered can be reduced by up to 99.99%. In particular for large circuits, significant reductions are achieved. For example, the error location of circuit *prng15* containing approx. 700k gates can be reduced to 44 gates. If additionally SAT-based debugging is applied, the results can be further improved. Then, for nearly all circuits a single error candidate returns. Thus, the error location of *prng15* can be identified in only sometimes more than a minute. In contrast, if SAT-based debugging is solely applied, this circuit cannot be handled within the given time limit, i.e. all gates remain as error candidates. Also for the other circuits, significant improvements are observed. In total, the run-time can be improved by up to four orders of magnitude (*urf3*, *prng11*) and in the best case the number of error candidates is reduced from 202 to 1 (*plus127mod8192*).

B. Wrong Target Line Errors

In a second evaluation, target line errors have been injected into the considered circuits. The results obtained by the respective debugging approaches are given in Table II. The column descriptions are equal to the ones from Table I.

It can be seen that the detection of wrong target errors is harder. As a result, the application of the SAT-based debugging approach results in time outs for larger circuits. In contrast, even for large circuits, the simulation-based approach can be efficiently applied leading to significant reductions in the number of error candidates that have to be considered (again up to 99.9% of the gates have been classified as non-relevant within some seconds). The run-time can be improved by up to five orders of magnitude (e.g. *urf1*).

TABLE II
EXPERIMENTAL RESULTS FOR WRONG TARGET LINE ERRORS

NAME	L	D	|T|	SIM.-BASED[a]		+SAT [23]		ONLY SAT [23]		QUALITY	
				REMAIN.		EC	TIME [s]	EC	TIME [s]	ΔEC	SPUP
3_17	3	6	4	2	66.67%	2	< 0.01	2	< 0.01	0	–
4_49	4	16	4	3	81.25%	2	< 0.01	1	0.01	1	> 1
4gt4	5	6	8	2	66.67%	1	< 0.01	1	< 0.01	0	–
4mod5	5	9	8	5	44.44%	2	< 0.01	2	< 0.01	0	–
ham3	3	5	4	1	80.00%	1	< 0.01	1	< 0.01	–	–
ham7	7	23	16	8	65.22%	2	0.01	2	0.02	0	2
ham15	15	132	40	11	91.67%	1	4.35	1	1.31	0	0.30
hwb4	4	17	8	1	94.12%	1	< 0.01	1	< 0.01	0	–
hwb5	5	55	16	1	98.18%	1	< 0.01	1	0.06	–	> 1
hwb6	6	126	16	2	98.41%	1	0.12	1	0.28	0	2.33
hwb7	7	289	8	3	98.96%	1	0.13	1	2.73	0	21
hwb8	8	637	32	4	99.37%	1	14.13	1	22.55	0	1.60
hwb9	9	1544	32	7	99.55%	1	146.14	1	189.66	0	1.30
plus63mod4096	12	429	40	12	97.20%	1	28.43	1	21.84	0	0.77
plus63mod8192	13	492	18	11	97.76%	1	31.02	1	27.49	0	0.88
plus127mod8192	13	910	16	62	93.19%	2	83.85	2	145.03	0	1.73
urf1	9	11554	40	4	$> 99.99\%$	*4*	0.20[b]	11554	> 5000	11550	> 25000.00
urf2	8	5030	40	1	$> 99.99\%$	1	0.09	2	2471.20	1	3045.80
urf3	10	26468	40	8	$> 99.99\%$	*8*	0.50[b]	26468	> 5000	26460	> 10000.00
urf5	9	10276	40	6	99.99%	*6*	0.18[b]	10276	> 5000	10270	> 27777.78
prng9	10	7617	40	300	96.06%	*300*	0.14[b]	7617	> 5000	7517	> 35714.29
prng10	10	16997	40	672	96.05%	*672*	0.32[b]	16997	> 5000	16325	> 15625.00
prng11	11	36910	40	1580	95.72%	*1580*	0.76[b]	36910	> 5000	35330	> 6578.95
prng12	13	79225	40	3352	95.77%	*3352*	1.75[b]	79225	> 5000	75873	> 2857.14
prng15	15	716934	40	30393	95.76%	*30393*	18.17[b]	716934	> 5000	686541	> 275.18

[a] The run-time of the simulation-based approach is negligible (always less than 20 CPU seconds) and thus is omitted due to space restrictions.
[b] The SAT-based debugging approach has been aborted. Thus, the run-time of the simulation-based approach only is given.

VII. CONCLUSION

In this paper, we introduced an alternative to the SAT-based debugging approach proposed in [23]. Instead of encoding the debugging problem as an instance of Boolean satisfiability (leading to complexity issues), a simulation-based approach with pattern matching is suggested. As the experiments show, in many cases already this leads to significant reductions of the number of error candidates. Moreover, the results can be further refined by applying SAT-based debugging afterwards. Since then many gates are already excluded from consideration, this enables the application of automated debugging approaches also for circuits containing hundreds of thousands of gates. Our experiments showed, that applying our approach leads to run-time improvements by up to five orders of magnitude and a smaller set of error candidates.

ACKNOWLEDGMENT

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

REFERENCES

[1] R. Landauer, "Irreversibility and heat generation in the computing process," *IBM J. Res. Dev.*, vol. 5, p. 183, 1961.
[2] C. H. Bennett, "Logical reversibility of computation," *IBM J. Res. Dev*, vol. 17, no. 6, pp. 525–532, 1973.
[3] T. Toffoli, "Reversible computing," in *Automata, Languages and Programming*, W. de Bakker and J. van Leeuwen, Eds. Springer, 1980, p. 632, technical Memo MIT/LCS/TM-151, MIT Lab. for Comput. Sci.
[4] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.
[5] B. Desoete and A. D. Vos, "A reversible carry-look-ahead adder using control gates," *INTEGRATION, the VLSI Jour.*, vol. 33, no. 1-2, pp. 89–104, 2002.
[6] R. Cuykendall and D. R. Andersen, "Reversible optical computing circuits," *Optics Letters*, vol. 12, no. 7, pp. 542–544, 1987.
[7] R. C. Merkle, "Reversible electronic logic using switches," *Nanotechnology*, vol. 4, pp. 21–40, 1993.
[8] V. V. Shende, A. K. Prasad, I. L. Markov, and J. P. Hayes, "Synthesis of reversible logic circuits," *IEEE Trans. on CAD*, vol. 22, no. 6, pp. 710–722, 2003.
[9] D. Maslov, G. W. Dueck, and D. M. Miller, "Toffoli network synthesis with templates," *IEEE Trans. on CAD*, vol. 24, no. 6, pp. 807–817, 2005.
[10] P. Gupta, A. Agrawal, and N. Jha, "An algorithm for synthesis of reversible logic circuits," *IEEE Trans. on CAD*, vol. 25, no. 11, pp. 2317–2330, 2006.
[11] R. Wille, H. M. Le, G. W. Dueck, and D. Große, "Quantified synthesis of reversible logic," in *Design, Automation and Test in Europe*, 2008, pp. 1015–1020.
[12] R. Wille and R. Drechsler, "BDD-based Synthesis of Reversible Logic for Large Functions," in *Design Automation Conf.*, 2009, pp. 270–275.
[13] G. F. Viamontes, M. Rajagopalan, I. L. Markov, and J. P. Hayes, "Gate-level simulation of quantum circuits," in *ASP Design Automation Conf.*, 2003, pp. 295–301.
[14] D. Goodman, M. A. Thornton, D. Y. Feinstein, and D. M. Miller, "Quantum logic circuit simulation based on the QMDD data structure," in *Int'l Reed-Muller Workshop*, 2007.
[15] J. Zhong and J. Muzio, "Using crosspoint faults in simplifying Toffoli networks," in *IEEE North-East Workshop on Circuits and Systems*, 2006, pp. 129–132.
[16] 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*, 2008, pp. 1378–1381.
[17] K. N. Patel, J. P. Hayes, and I. L. Markov, "Fault testing for reversible circuits," *IEEE Trans. on CAD*, vol. 23, no. 8, pp. 1220–1230, 2004.
[18] M. Perkowski, J. Biamonte, and M. Lukac, "Test generation and fault localization for quantum circuits," in *Int'l Symp. on Multi-Valued Logic*, 2005, pp. 62–68.
[19] I. Polian, T. Fiehn, B. Becker, and J. P. Hayes, "A family of logical fault models for reversible circuits," in *Asian Test Symp.*, 2005, pp. 422–427.
[20] G. F. Viamontes, I. L. Markov, and J. P. Hayes, "Checking equivalence of quantum circuits and states," in *Int'l Conf. on CAD*, 2007, pp. 69–74.
[21] S.-A. Wang, C.-Y. Lu, I.-M. Tsai, and S.-Y. Kuo, "An XQDD-based verification method for quantum circuits," *IEICE Transactions*, vol. 91-A, no. 2, pp. 584–594, 2008.
[22] R. Wille, D. Große, D. M. Miller, and R. Drechsler, "Equivalence checking of reversible circuits," in *Int'l Symp. on Multi-Valued Logic*, 2009.
[23] R. Wille, D. Große, S. Frehse, G. W. Dueck, and R. Drechsler, "Debugging of Toffoli networks," in *Design, Automation and Test in Europe*, 2009, pp. 1284–1289.
[24] A. Smith, A. G. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.
[25] N. Eén and N. Sörensson, "An extensible SAT solver," in *SAT 2003*, ser. LNCS, vol. 2919, 2004, pp. 502–518.
[26] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler, "RevLib: An online resource for reversible functions and reversible circuits," in *Int'l Symp. on Multi-Valued Logic*, 2008, pp. 220–225, RevLib is available at http://www.revlib.org.