UDK621.3:(53+54+621+66), ISSN0352-9045 Informacije MIDEM 32(2002)3, Ljubljana

EXPLOITING SYMBOLIC MODEL CHECKING FOR SENSING STUCK-AT FAULTS IN DIGITAL CIRCUITS

Aleš Časar, Zmago Brezočnik, Tatjana Kapus
University of Maribor, Faculty of Electrical Engineering and Computer Science, Slovenia

Keywords: stuck-at faults, symbolic model checking, automatic test pattern generation, testing, CTL formulas, finite state machine, binary decision diagrams

Abstract: This paper presents algorithms for automatic test pattern generation for discovering stuck-at faults in sequential digital circuits or proving that there are no stuck-at faults in the given circuit. A circuit is represented as a finite state machine. Properties for stuck-at faults expressed with CTL formulas which are valid in the circuit with stuck-at faults and generally not valid in the good circuit are generated. Validity of the formulas is checked by symbolic model checking, and for invalid formulas counterexamples are constructed which guide the circuit to the states which prove the absence of stuck-at faults. Test patterns guide the circuits exactly as the counterexamples. Experimental results for a set of benchmark circuits together with the time and space complexity analysis of the algorithms are also given.

Uporaba simboličnega preverjanja modelov pri zaznavanju zatičnih napak v digitalnih vezjih

Ključne besede: zatične napake, simbolično preverjanje modelov, avtomatsko generiranje testnih vzorcev, testiranje, formule CTL, končni avtomat, binarni odločitveni grafi

Povzetek: V članku predstavljamo algoritme za avtomatsko generiranje testnih vzorcev, s pomočjo katerih pri sekvenčnih digitalnih vezjih odkrivamo zatične napake oziroma pokažemo, da zatičnih napak v danem primerku vezja ni. Vezje predstavimo kot končni avtomat. Za zatične napake generiramo lastnosti v obliki formul CTL, ki so veljavne v vezju z zatičnimi napakami in praviloma neveljavne v dobrem vezju. S simboličnim preverjanjem modelov preverimo veljavnost formul in za neveljavne formule skonstruiramo protiprimere, s katerimi vezje pripeljemo v stanja, ki dokažejo odsotnost zatičnih napak. Testni vzorci so sestavljeni tako, da izvajanje vezja vodijo po poti protiprimerov. Teoretične raziskave so podkrepljene z eksperimentalnimi rezultati. Prikazana je tudi analiza časovne in prostorske zahtevnosti.

1 Introduction

Testing of newly produced digital circuits is a necessity. Since circuits are becoming larger and larger, it is impossible to perform exhaustive testing of the circuits nowadays. Therefore, a suitable trade-off between exhaustive testing and speed of testing (length of test patterns) should be made. We tend to discover (or prove their absence) as many circuit faults as possible but at moderate test pattern length. Many different faults can occur in a circuit, but stuck-at faults are the most common ones. Hence, we introduce the method which will find all possible single stuck-at faults in the circuit or prove that there is no stuck-at fault present in the circuit.

Because enumeration methods [10] do not perform well with large circuits, we propose to use symbolic methods [5, 6]. A circuit is represented as a finite state machine (FSM). FSMs are represented as Boolean functions and these with binary decision diagrams (BDDs). Properties of FSMs which are to be checked by symbolic model checking are expressed with CTL formulas.

For every possible single stuck-at fault, a property which is valid in the circuit with that stuck-at fault and generally invalid in the good circuit can be generated. When the property is invalid, a counterexample can be found. If the test pattern guides the circuit exactly as the counterexample, the absence of the treated stuck-at fault is proved if testing with this test pattern ends successfully.

We neither deal with other types of possible faults nor with multiple stuck-at faults in this paper. Practice suggests that also multiple stuck-at faults can be discovered in most cases, but we did not prove that. A mentionable limitation of the proposed method is also

the necessity of the insight into the circuits (logic values stored in flip-flops).

In Section 2 we briefly show how to represent FSMs with BDDs, describe searching of reachable states, and symbolic model checking in CTL. Section 3 describes the methods of searching counterexamples and witnesses. The main part of the paper is Section 4 where we present algorithms for generation of properties for stuck-at faults. Experimental results for benchmark circuits with time and space complexity analysis are given in Section 5. We conclude with some discussion and plans for future work.

2 Preliminaries

Binary decision diagrams (BDDs) are compact canonical representations of Boolean functions [2]. BDDs can be used for representing and manipulating sets if we represent sets by means of their characteristic functions [5, 6].

A deterministic *finite state machine* (FSM) M is a sixtuple $M = (\Sigma, \mathcal{S}, \mathcal{O}, \delta, \lambda, s_0)$, where Σ is a finite set of input symbols, $\mathcal{S}$ a finite set of states, $\mathcal{O}$ a finite set of output symbols, $\delta : \mathcal{S} \times \Sigma \to \mathcal{S}$ a state transition function, $\lambda : \mathcal{S} \times \Sigma \to \mathcal{O}$ an output function, and $s_0 \in \mathcal{S}$ an initial state.

If we want to realize a FSM by a digital circuit, we have to encode the sets $\mathcal{S}$, Σ, and $\mathcal{O}$ by binary symbols (e.g. 0 and 1). States are encoded by state variables. At least $n = \lceil \log_2 |\mathcal{S}| \rceil$ state variables, $m = \lceil \log_2 |\Sigma| \rceil$ input variables, and $l = \lceil \log_2 |\mathcal{O}| \rceil$ output variables are needed. Let $\mathcal{Y}$, $\mathcal{X}$, and $\mathcal{Z}$ represent the set of state variables, the set of input variables, and the set of output variables, respectively.

Once the states and the input symbols of the circuit are encoded, next state variables are functions of present state variables and input variables. We denote next state variables by an added prime ($'$) and write a transition function of a state variable y_i as

$$y_i' = \delta_i(y_0, y_1, \ldots, y_{n-1}, x_0, x_1, \ldots, x_{m-1}) \qquad (1)$$

for $i = 0, 1, \ldots, n-1$. We rather introduce transition relations

$$T_i = y_i' \Leftrightarrow \delta_i(y_0, y_1, \ldots, y_{n-1}, x_0, x_1, \ldots, x_{m-1}). \qquad (2)$$

Namely, relations have much greater expressive power than functions [3]. Transition relations T_i can be combined by taking their conjunction to form the *monolithic transition relation* $T = T_0 \cdot T_1 \cdot \ldots \cdot T_{n-1}$. After the encoding, output variables are functions of present state variables and input variables. We write

$$z_i = \lambda_i(y_0, y_1, \ldots, y_{n-1}, x_0, x_1, \ldots, x_{m-1}) \qquad (3)$$

for $i = 0, 1, \ldots, l-1$.

Let us very briefly show how we search reachable states of a FSM [1, 5, 6]. Let $\mathcal{S}_i$ denote a set of states reachable in at most i steps. $\mathcal{S}_0$ represents a set of initial states. In our case we have $\mathcal{S}_0 = \{s_0\}$. In general, a set of states reachable in at most i steps is represented by

$$\mathcal{S}_i = \mathcal{S}_{i-1} \cup \left\{ s' \mid \exists a \exists s [a \in \Sigma \wedge s \in \mathcal{S}_{i-1} \wedge \delta(s, a) = s'] \right\}. \qquad (4)$$

We continue with this procedure until in a step k no new state is reached. In any case, this happens sooner or later, because we deal with FSMs, where the set of states $\mathcal{S}$ is finite. Then, $\mathcal{S}_k = \mathcal{S}_{k-1}$ is a set of all reachable states.

The logic which we use to specify properties of FSMs is a propositional temporal logic of branching time, called *Computation Tree Logic* — CTL [9]. In this logic, each of the usual future time operators of linear-time temporal logic (***G*** — *globally* or *invariantly*, ***F*** — *sometimes in the future*, ***X*** — *next time*, ***U*** — *until*) must be immediately preceded by *path quantifier* ***A*** (for all computation paths) or ***E*** (for some computational path). We thus obtain eight different CTL operators: ***AG***, ***EG***, ***AF***, ***EF***, ***AX***, ***EX***, ***AU***, ***EU*** [1, 5, 6].

CTL formulas are constructed from *atomic propositions* using Boolean connectives and CTL operators. In the case of circuit verification, the set of atomic propositions is equal to the set $\mathcal{Y}$ of state variables of the circuit.

3 Searching for Counterexamples and Witnesses

One of the most important extensions of symbolic model checking is the ability to search counterexamples for some invalid and witnesses for some valid CTL formulas [8]. Counterexamples explain why a given CTL formula is invalid in a FSM, and witnesses show why a given CTL formula is valid.

3.1 Counterexamples

Counterexample is a path in the computation tree which shows why a given CTL formula is invalid. Actually, this is evident from the last state on the path, but FSM must be guided to this state to demonstrate invalidity of the formula.

It is impossible to find counterexamples for all invalid formulas. According to the definition of CTL formulas [5], they are constructed from atomic propositions, Boolean operators, and CTL operators. Let us look how these three constructs affect searching of counterexamples.

Searching of a counterexample for an atomic proposition is a trivial task. Such a formula represents the characteristic function of the set of states where the formula is valid. Because counterexamples are searched only for invalid formulas, it suffices to check if the current state of the FSM is not in that set.

Although there are 16 binary Boolean operators, all of them can be expressed by negation, conjunction, and disjunction. Searching for a counterexample for negation of a function means the same as searching for a witness for that (non-negated) function. Therefore, searching of a counterexample for $\overline{f}$ is equivalent to searching of witness for f. Searching of witnesses will be described in Section 3.2.

When dealing with conjunction, a counterexample for a formula of the form $f \cdot g$ should be found. Because the formula is invalid (counterexamples are searched only for invalid formulas), at least one of the functions f or g is invalid. To find a counterexample for the whole function $f \cdot g$, it is enough to find either a counterexample for f or a counterexample for g. Of course, in the case one of the formulas f or g is valid, we should find a counterexample for the other one, which is invalid.

It is not possible to find a counterexample for disjunction in all cases. If a formula of the form $f + g$ is invalid, then both formula f and formula g are invalid. To prove that, one should prove both simultaneously. Generally this is impossible to do with just one counterexample, but there are exceptions. If one of the formulas f or g is of such a kind that the counterexample for it is not a path but just a single state (this happens when a formula does not contain any CTL operators), then also a counterexample for disjunction can be found. We search a counterexample for the other formula and at the end also check that the current state is not in the set of states which our formula without CTL operators is the characteristic function for.

Let us now look at another interesting exception. Triple Boolean operator $ite(f, g, h)$ can be written also as $f \cdot g + \overline{f} \cdot h$. If f represents a Boolean formula (without temporal operators) in such a construction, then exactly one of the disjunctives is invalid because of f in every state. There are two cases:

1. Formula f is valid in the given state. Operand $\overline{f} \cdot h$ is invalid then and we have to show that operand $f \cdot g$ is also invalid. It is known that factor f is valid, therefore we have to show that factor g is invalid. Actually we have to search for a counterexample for g.

2. Formula f is invalid in the given state. In that case operand $f \cdot g$ is invalid because of f, and the only thing we have to prove is that operand $\overline{f} \cdot h$ is also invalid. Because $\overline{f}$ is valid here, h is invalid accordingly. Therefore, a counterexample for h should be searched for.

After explaining Boolean operators, let us devote now to CTL operators. Counterexamples can be searched only for CTL operators with universal path quantifier $\boldsymbol{A}$. Namely, only these operators state that on *every* computation path a formula is valid, and a counterexample is *one* computation path where this formula is invalid.

$\boldsymbol{AX}\, f$ states that formula f is valid in all successors of the present state. Problem of searching a counterexample for an invalid formula of such a kind is to find a successor of the present state where formula f is invalid. When an adequate successor is found, a counterexample for formula f should be found.

According to formula $\boldsymbol{AF}f$, every computation path from the present state should lead to a state where formula f is valid. If this is not true, we have to find a path where we will *never* reach a state where formula f is valid. Since computation paths are infinite, how at all can we show that something never happens along a given path? Paths are infinite indeed, but they lead over finite set of states. Therefore, at least one state on the path must occur repeatedly. When an already visited state is reached, the path from this state back to itself can safely be repeated infinitely often. Actually, the path ends with a cycle. It is not necessary that the initial part of the path is part of the cycle. In that case, the path is composed of the prefix and the cycle as shown in Fig. 1. It is enough to find a prefix

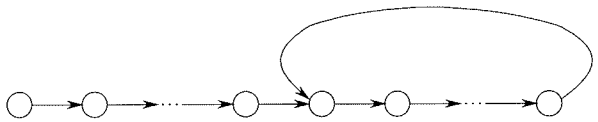

Figure 1: A path with prefix and cycle

and a cycle to find such a path. When we return to an already visited state at traversing the states, we have found a counterexample as a proof that formula $\boldsymbol{AF}f$ is invalid indeed. Formula f must be invalid in every state along that path, therefore, counterexamples for itself should be found in every state on that path. However, those "nested" counterexamples must not go off the path, since our computation path would transform to a computation tree otherwise. To avoid such situations, formula f should not contain any CTL operators.

Formula $\boldsymbol{AG}\, f$ says that on all the paths, formula f is valid all the time. In order to prove its invalidity, a path leading to a state where formula f is invalid has to be found. As with operator $\boldsymbol{AX}$, from there on a counterexample for formula f has to be found in order to confirm its invalidity in the last state of the path for $\boldsymbol{AG}\, f$.

The last CTL operator containing the universal path quantifier is $\boldsymbol{A}\,[f\, \boldsymbol{U}\, g]$, which says that on all the paths, formula f is valid until a state where formula g is valid. In order to refute validity of formula $\boldsymbol{A}\,[f\, \boldsymbol{U}\, g]$, either an infinite path on which formula g is never valid or a path which leads to a state where formula f is invalid and on which formula g is never valid has to be found. The case of the infinite path is similar to the counterexample construction with operator $\boldsymbol{AF}$ — only a prefix and a cycle terminating the infinite path have to be found. As in the case of $\boldsymbol{AF}$, formula g must not contain any CTL operators. If one decides to search for a path leading to a state where formula f is invalid, one must,

of course, continue with a counterexample for f.

3.2 Witnesses

Witness is a path in the computation tree which indicates why a CTL formula considered is valid. In fact, the validity is evident from the last state of the path, but the FSM has to be led to that state in order for the path to demonstrate the validity.

As with counterexamples, witnesses cannot be found for every valid CTL formula. Since witnesses are in a sense dual to counterexamples, problems occur exactly with the CTL formulas dual to those CTL formulas, validity of which cannot be demonstrated by counterexamples. We now make an overview of the structural elements of CTL formulas and their influence on searching of witnesses.

Searching of a witness for an atomic proposition is in fact the same as searching of a counterexample. The difference is that witnesses are searched for valid formulas. Therefore, it must be checked if the current state of the FSM *is* in the set determined by the characteristic function in the form of the given atomic proposition.

Searching of a witness for negation of a function means the same as searching of a counterexample for the (non-negated) function. It follows that searching of a witness for $\overline{f}$ is equivalent to searching of a counterexample for f. Searching of the counterexamples is described in detail in Section 3.1.

Another example is searching of a witness for a disjunction $f + g$. Since the formula is valid (as witnesses are searched only for valid formulas), at least one of the functions f and g is valid. In order to find a witness for function $f + g$, it therefore suffices to find either a witness for f or a witness for g. If one of the formulas is invalid, a witness for the valid one has to be found, of course.

With counterexamples, there were some problems with disjunction, whereas due to duality of witnesses, similar problems now occur with conjunction, which is dual to disjunction. It is not always possible to find a witness for conjunction. If a formula of the form $f \cdot g$ is valid, then formula f and formula g must be valid. In order to demonstrate that, validity of both should be demonstrated *simultaneously*. As a witness must be a path, which must not have branches, this is generally not possible. Exceptions, however, exist also in this case. If one of the formulas f and g is such that its witness is a path containing just one state (this is the case when the formula does not contain any CTL operators), then a witness for the conjunction can be found as well. A witness for the other formula has to be found, and at the end, it has to be checked if the current state is also in the set of states whose characteristic function is the formula without CTL operators.

Finally, let us look at the CTL operators and their influence on searching of witnesses. Witnesses can only be found for the CTL operators containing existential path quantifier $\boldsymbol{E}$. They say that there *exists* a computation path where a formula is valid, and a witness is simply *one* path which confirms the existence and validity of the formula.

Formula $\boldsymbol{EX}f$ says that there exists a successor of the current state where formula f is valid. In order to find a witness for a valid formula of this form, a successor of the current state where formula f is valid has to be found. A witness for f must then be found from the successor state on.

Formula $\boldsymbol{EF}f$ says that there exists at least one path leading to a state where formula f is valid. In order to prove its validity, a path leading to such a state has to be found and from there on, a witness for formula f has to be found.

Formula $\boldsymbol{EG}f$ says that there exists an infinite path on which formula f is valid all the time. We already know how to deal with infinite paths. A prefix and a cycle at the end of an infinite path have to be found, such that formula f is valid in every state of the path. When during passing from state to state, an already visited state is reached for the first time, the searching is finished. Since validity of formula f must be confirmed by the witness along the whole path, which must not be abandoned, the confirmation is possible only if formula f does not contain any CTL operators.

We already know that $\boldsymbol{E}[f\,\boldsymbol{U}\,g]$ says there exists a path where formula f is valid until a state is reached where formula g is valid. In order to find a witness, one of such paths must, therefore, be found. Formula f must again not contain any CTL operators, and starting from the last state, a witness for the validity of formula g in that state must be found.

3.3 Realization

In order to implement symbolic model checking, we implemented only resolution of three CTL operators, $\boldsymbol{EX}f$, $\boldsymbol{E}[f\,\boldsymbol{U}\,g]$, and $\boldsymbol{EG}f$, whereas the other five were expressed with them [5]. The same approach can be followed to realize searching of counterexamples and witnesses. For unrealized operators the searching can be realized as follows:

- searching of a counterexample for $\boldsymbol{AX}f$ is replaced by searching of a witness for $\boldsymbol{EX}\overline{f}$,
- searching of a witness for $\boldsymbol{EF}f$ is replaced by searching of a witness for $\boldsymbol{E}[1\,\boldsymbol{U}\,f]$,
- searching of a counterexample for $\boldsymbol{AF}f$ is replaced by searching of a witness for $\boldsymbol{EG}\overline{f}$,
- searching of a counterexample for $\boldsymbol{AG}f$ is replaced by searching of a witness for $\boldsymbol{E}[1\,\boldsymbol{U}\,\overline{f}]$,
- searching of a counterexample for $\boldsymbol{A}[f\,\boldsymbol{U}\,g]$ is replaced either by searching of a witness for $\boldsymbol{E}[\overline{g}\,\boldsymbol{U}\,\overline{f}\cdot\overline{g}]$ or by searching of a witness for $\boldsymbol{EG}\overline{g}$.

To find out if formula $\boldsymbol{EX}f$ is valid in a given state s using symbolic model checking, we start by the set $\mathcal{S}_f$ of states in which formula f is valid. This set will also be used to find a witness for validity of formula $\boldsymbol{EX}f$ in state s, which can be done if it is found that formula $\boldsymbol{EX}f$ is valid in state s. In order to find this witness, we have to find a successor of state s in which formula f is valid. In accordance with formula (4), the set of all successors of state s is calculated as follows:

$$\mathcal{S}' = \left\{ s' \mid \exists a[a \in \Sigma \wedge \delta(s,a) = s'] \right\} . \qquad (5)$$

In the set $\mathcal{S}'$, we have to choose a successor of s in which formula f is valid. It follows that state s' must not only be in set $\mathcal{S}'$, but also in $\mathcal{S}_f$, i.e. $s' \in \mathcal{S}' \cap \mathcal{S}_f$ must hold. This is illustrated in Fig. 2. The witness for

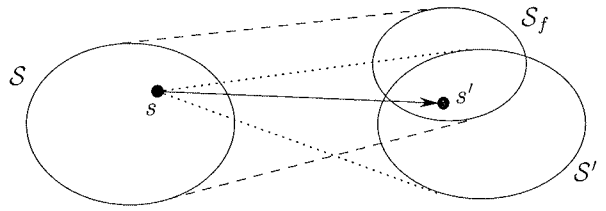

Figure 2: Witness for $\boldsymbol{EX}f$

formula $\boldsymbol{EX}f$ in state s is, therefore, composed of two states, (s, s'), and is found in one step.

When checking validity of formula $\boldsymbol{E}[f\,\boldsymbol{U}\,g]$, we start with the set $\mathcal{S}_g$ of all states where formula g is valid. By gradually adding all such predecessors where formula f is valid, we obtain the set of all states where formula $\boldsymbol{E}[f\,\boldsymbol{U}\,g]$ is valid. Let $\mathcal{S}_f$ be the set of all states where formula f is valid, and let $\mathcal{S}^i$ denote the set of all states obtained until the i-th step of the procedure, the step included. Note that $\mathcal{S}^0 = \mathcal{S}_g$. The situation is shown in Fig. 3. The sets have the follow-

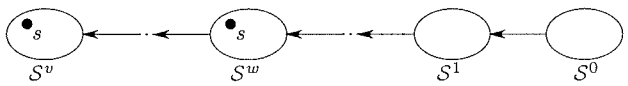

Figure 3: Set of states at checking formula $\boldsymbol{E}[f\,\boldsymbol{U}\,g]$

ing characteristics:

1. $\mathcal{S}^i \subset \mathcal{S}_f \cup \mathcal{S}_g$ for $i = 0, 1, \ldots, v$.
2. $\mathcal{S}^{i-1} \subset \mathcal{S}^i$ for $i = 1, 2, \ldots, v$.
3. $\mathcal{S}^i \setminus \mathcal{S}^{i-1} \neq \emptyset$ for $i = 1, 2, \ldots, v$. For all states in this set difference there exists a path to a state in $\mathcal{S}^0$ of length i and no shorter path exists to any of the states in $\mathcal{S}^0$.
4. $\mathcal{S}^w$ is the first set on the construction path (and also the smallest set) which contains state s.
5. For $\mathcal{S}^v$, there does not exist any successor in which formula f would be valid but would not already be in $\mathcal{S}^v$.

In order to find a witness for formula $\boldsymbol{E}[f\,\boldsymbol{U}\,g]$ in a state s, we must find a path from the state s in the set $\mathcal{S}^w$ to a state in the set $\mathcal{S}^0$. A path of the form $(s_w, s_{w-1}, \ldots, s_0)$ will contain exactly w steps[1], where for all states, $s_i \in \mathcal{S}^i$ and $s_w = s$. It is clear that states from the sets cannot be chosen arbitrarily. For any pair of states s_i and s_{i-1}, a transition from s_i to s_{i-1} must exist. We, therefore, choose in each step a state s_{i-1} which will also be in the set of all successors of state s_i. If the set is denoted by $\mathcal{S}^{i'}$, then we can write $s_{i-1} \in \mathcal{S}^{i-1} \cap \mathcal{S}^{i'}$. An example for $w = 3$ is shown in Fig. 4.

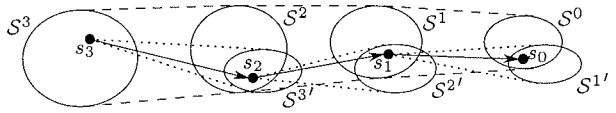

Figure 4: Witness for $\boldsymbol{E}[f\,\boldsymbol{U}\,g]$

When checking a formula $\boldsymbol{EG}f$ in a state s, we calculate the set of all states where the formula is valid. Let the set be denoted by $\mathcal{S}$. From every state in this set there leads an infinite path where f is valid all the time. Since any state on such a path is also a starting state of an infinite path where formula f is valid all the time, all the states on the path are in the set $\mathcal{S}$ and the complete path runs within $\mathcal{S}$. The construction of a witness begins in the state s. In the i-th step we choose a state from the intersection of the set $\mathcal{S}_{i-1}{}'$, which contains all successors of the current state s_{i-1}, with the set $\mathcal{S}$. The general construction step is shown in Fig. 5. If the intersection contains some already visited state,

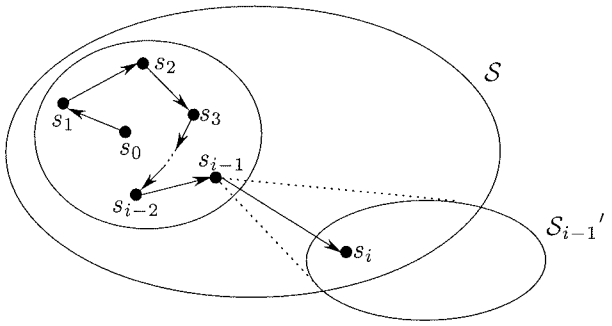

Figure 5: General construction step of the witness for $\boldsymbol{EG}f$

we choose such a state and the construction of a witness is ended; otherwise, we choose an arbitrary successor and continue the procedure. The algorithm will end anyway, as the set $\mathcal{S}$ contains a finite number of states. We will reach an already visited state in the $|\mathcal{S}|$-th step at the latest. For a path $(s_0, s_1, \ldots, s_v, \ldots, s_w)$ which is a witness, the following is true:

- $s_0 = s$;

[1]Of course, a longer path could also serve as a witness. It is, however, useful to have as short witnesses and counterexamples as possible.

- there exists a transition from state s_{i-1} to state s_i for $i = 1, 2, \ldots, w$;
- $s_w = s_v$ where $v \in \{0, 1, \ldots, w-1\}$;
- $s_i \neq s_j$ for $i, j = 0, 1, \ldots, w-1$ and $i \neq j$;
- $w \leq |\mathcal{S}|$.

The complete witness and the last construction step are shown in Fig. 6.

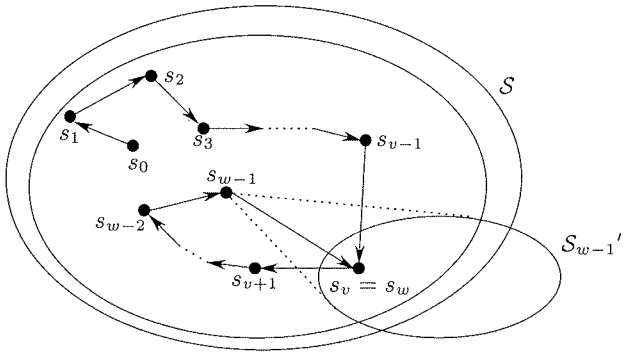

Figure 6: Witness for $\boldsymbol{EG}\, f$ with last construction step

4 Generation of Properties for Stuck-at Faults

4.1 Stuck-at Faults

Digital sequential circuits consist of flip-flops, logic gates, and lines between them. Generally, some lines lead to the circuit, i.e. they are inputs, whereas some are outputs, which lead out of the circuit. Such a circuit can be looked upon as a realization of a binary encoded FSM. Every flip-flop has its state variable, circuit inputs are the inputs of the FSM, and outputs are also its outputs. The state transition function is determined by logic gates and lines which connect them with the flip-flops, and similarly for the output function.

For example, let us look at an arbiter which chooses the request with the highest priority from among the three requests on its inputs. The arbiter is shown in Fig. 7. The smaller the number of a requesting device, the higher its priority. State variables of the FSM realized by the arbiter in Fig. 7 are IN_0, IN_1, IN_2, OUT_0, OUT_1, and OUT_2. Its inputs are REQ_0, REQ_1, and REQ_2, and its outputs are GR_0, GR_1, and GR_2. Here

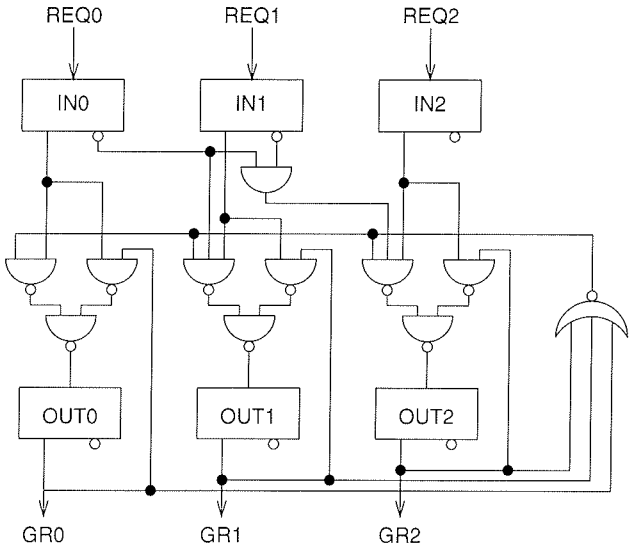

Figure 7: Digital circuit schematics of a 3-input arbiter

are its state transition functions:

$$\begin{aligned}
IN_0' &= REQ_0 \\
IN_1' &= REQ_1 \\
IN_2' &= REQ_2 \\
OUT_0' &= \overline{OUT_0} \cdot \overline{OUT_1} \cdot \overline{OUT_2} \cdot IN_0 + IN_0 \cdot OUT_0 \\
OUT_1' &= \overline{OUT_0} \cdot \overline{OUT_1} \cdot \overline{OUT_2} \cdot \overline{IN_0} \cdot IN_1 + \\
&\quad IN_1 \cdot OUT_1 \\
OUT_2' &= \overline{OUT_0} \cdot \overline{OUT_1} \cdot \overline{OUT_2} \cdot \overline{IN_0} \cdot \overline{IN_1} \cdot IN_2 + \\
&\quad IN_2 \cdot OUT_2
\end{aligned} \tag{6}$$

Its output functions are as follows:

$$\begin{aligned}
GR_0 &= OUT_0 \\
GR_1 &= OUT_1 \\
GR_2 &= OUT_2
\end{aligned} \tag{7}$$

In general, a circuit gives a FSM with a set of state variables $\mathcal{Y} = \{y_{n-1}, y_{n-2}, \ldots, y_0\}$, inputs $\mathcal{X} = \{x_{m-1}, x_{m-2}, \ldots, x_0\}$, and outputs $\mathcal{Z} = \{z_{l-1}, z_{l-2}, \ldots, z_0\}$. The FSM has the transition functions (1) and the output functions (3).

A stuck-at fault is caused by a short circuit between a line connecting two elements and the logical 1 or 0. If the short circuit is with the logical 0, there is an SA0 fault ("stuck-at 0"). If the short circuit is with the logical 1, there is an SA1 fault ("stuck-at 1").

The question is how a stuck-at fault in a circuit is manifested in the FSM whose realization it is. In the circuit, all the flip-flops as well as external inputs and outputs remain the same. It follows that the FSM whose realization is the circuit *with* a stuck-at fault will have the same state variables, inputs, and outputs as the FSM whose realization is the good circuit without stuck-at faults. However, some transition and output function can change due to changed connections.

Each line connects exactly one output of a logic gate or flip-flop or an input of the circuit with, in general,

many inputs of logic gates, flip-flops, or outputs of the circuit. Since the source of the line is uniquely determined, a stuck-at fault can cannonically be denoted by $G = 0$, respectively $G = 1$, which means that an output of a logic gate or a flip-flop, or a circuit input G has stuck at 0, respectively to 1. With the transition functions (1) and the output functions (3), we obtain for the FSM corresponding to the circuit with a stuck-at fault $G = b$ (G can be equal to any of the possible fault locations and $b \in \{0, 1\}$) the following transition functions:

$$y_i' = f_i|_{G=b}(y_0, y_1, \ldots, y_{n-1}, x_0, x_1, \ldots, x_{m-1}) \quad (8)$$

for $i = 0, 1, \ldots, n-1$ and output functions:

$$z_i = g_i|_{G=b}(y_0, y_1, \ldots, y_{n-1}, x_0, x_1, \ldots, x_{m-1}) \quad (9)$$

for $i = 0, 1, \ldots, l-1$. It can happen that not all transition and output functions are changed as a consequence of a stuck-at fault. It is perfectly possible that $f_i = f_i|_{G=b}$, respectively $g_j = g_j|_{G=b}$, for some values of i and j. This is in fact quite usual with real circuits. If a stuck-at fault occurs in a redundant part of the circuit, it can even happen that all the transition and output functions remain the same.

Suppose that in the circuit in Fig. 7 the output of the logic gate NOR would stick at logical 1. The circuit would become a realization of a FSM similar to the original one. Only the state transition functions for OUT_0, OUT_1, and OUT_2 would change:

$$\begin{aligned} OUT_0' &= IN_0 \\ OUT_1' &= \overline{IN_0} \cdot IN_1 + IN_1 \cdot OUT_1 \\ OUT_2' &= \overline{IN_0} \cdot \overline{IN_1} \cdot IN_2 + IN_2 \cdot OUT_2 \end{aligned} \quad (10)$$

The rest of transition and output functions would remain the same.

4.2 Extension of FSM

Any circuit property expressed in the form of a CTL formula is valid in some states of the FSM corresponding to the circuit. The states are determined by state variable values. Input and output values do not determine the current state of the FSM.

Since stuck-at faults generally also affect the output values, they must be covered by properties as well. As outputs cannot be included in CTL formulas, a possible solution is to extend the FSM with additional state variables. For every output z_i, a state variable y_{n+i} is added. The variable's transition function is defined as $y_{n+i}' = f_{n+i} = z_i$ for $i = 0, 1, \ldots, l-1$. The new outputs of the extended FSM are defined as $z_{z,i} = g_{z,i} = y_{n+i}$ for $i = 0, 1, \ldots, l-1$.

Stuck-at faults do not affect the circuit input values. However, the input values affect the circuit (FSM) state transitions in the future. For this reason, the inputs must also be covered by properties. The solution is similar to the one for outputs, only that here, additional state variables are added at the inputs of the original circuit. For every input x_i, a state variable y_{n+l+i} is added. Transition functions for the added state variables are defined as $y_{n+l+i}' = f_{n+l+i} = x_{x,i}$ for $i = 0, 1, \ldots, m-1$, where $x_{x,i}$ is a new input of the extended circuit and m the number of inputs. In all formulas f_i and g_j where the original circuit inputs x_k occur, we take into account that $x_k = y_{n+l+k}$ for $k = 0, 1, \ldots, m-1$.

Graphically, a FSM extension can easily be shown in the corresponding circuit schematics. If we extend the arbiter from Fig. 7 with additional state variables (i.e. flip-flops in the circuit) in the way just described, we obtain the circuit shown in Fig. 8. The added elements are drawn with dotted lines.

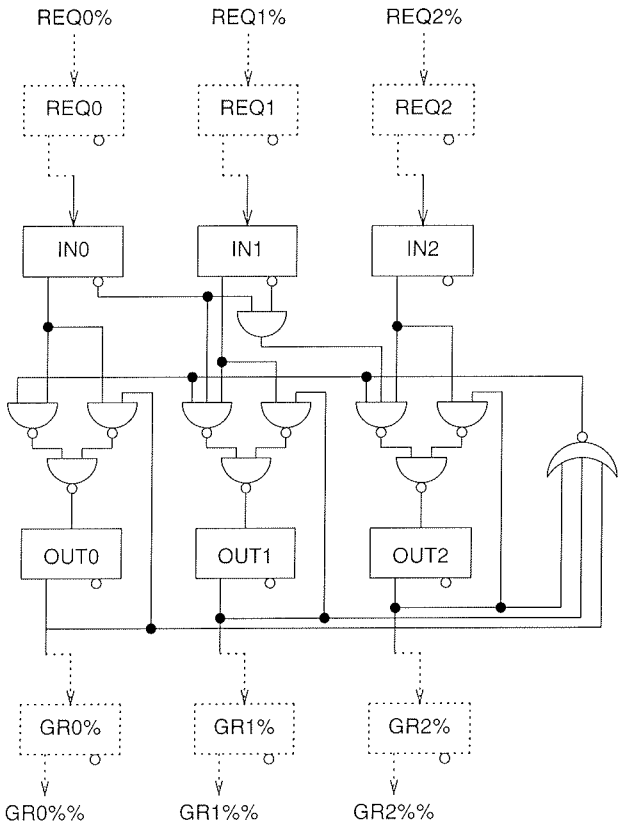

Figure 8: Extended arbiter with additional state variables

The newly added state variables y_i (for $i = n, n+1, \ldots, n+l+m-1$) can be used in CTL formulas like any other state variable. The new variables can be used to express properties about the original FSM inputs and outputs. It should be noticed that stuck-at faults can only occur on the lines in the original circuit and, consequently, in the corresponding places in the original and extended FSM.

4.3 Properties

In every FSM, many properties are valid. One of the most basic types of properties, which are valid in every

FSM, are properties of the form:

$$\boldsymbol{AG}(f_i \cdot \boldsymbol{AX}\, y_i + \overline{f_i} \cdot \boldsymbol{AX}\, \overline{y_i}) \qquad (11)$$

The formula says that on all possible computation paths and all the time it holds that if in some state the state transition function f_i of the state variable y_i has the value 1, then in every successor of the state, the value of y_i will also be 1; respectively, if the value of the transition function f_i of the state variable y_i in the state is 0, then the variable y_i will also be 0 in every successor state. This holds for all the state variables, including the added ones.

The properties cannot help us to find stuck-at faults. Since they are valid, counterexamples do not exist, of course. Also, for this type of formulas, even a witness cannot be constructed according to the explanation in Section 3.2.

However, following the above pattern we can write a CTL formula which will be valid in the FSM that corresponds to the circuit with a stuck-at fault. If an output of a logic gate or a flip-flop, or a circuit input G sticks at a logical value b, then the property has the form:

$$\boldsymbol{AG}(f_i|_{G=b} \cdot \boldsymbol{AX}\, y_i + \overline{f_i|_{G=b}} \cdot \boldsymbol{AX}\, \overline{y_i}) \qquad (12)$$

By rule, such a formula cannot be valid in a circuit (FSM) without a stuck-at fault. It follows that a counterexample can be constructed for such a CTL formula. The formula namely does not contain CTL operators with existential path quantifier. It contains a disjunction, but in such a way that a counterexample can be found anyway.

When a stuck-at fault $G = b$ is inserted into a FSM, the properties (12) are valid for all the state variables, i.e. for $i = 0, 1, \ldots, n+l+m-1$. The single properties can be applied to form the common one:

$$\bigwedge_{i=0}^{n+l+m-1} \boldsymbol{AG}(f_i|_{G=b} \cdot \boldsymbol{AX}\, y_i + \overline{f_i|_{G=b}} \cdot \boldsymbol{AX}\, \overline{y_i}) \qquad (13)$$

Due to distributivity of operator $\boldsymbol{AG}$ and conjunction, formula (13) can be rewritten as

$$\boldsymbol{AG} \bigwedge_{i=0}^{n+l+m-1} (f_i|_{G=b} \cdot \boldsymbol{AX}\, y_i + \overline{f_i|_{G=b}} \cdot \boldsymbol{AX}\, \overline{y_i}) \qquad (14)$$

Some of the conjunctives in the CTL formula (14) can be valid always and everywhere. Finding all such factors is generally a complex problem, but certainly, all the factors for which the stuck-at fault $G = b$ does not affect the corresponding transition function and consequently, $f_i|_{G=b} = f_i$, are among them. If a CTL formula is valid always and everywhere, then it is equivalent to formula 1 ("true"). It follows that such factors can be left out in the conjunction. Only those have to be included, for which $f_i|_{G=b} \neq f_i$. The factors corresponding to the variables added at the original inputs y_i $(i = n+l, n+l+1, \ldots, n+l+m-1)$ are also true. This is because all the lines which affect the variables are added only in the abstract FSM, whereas they do not exist in the real circuit, in which stuck-at faults can occur. If we take into account both facts, we get the following CTL formula (property):

$$\boldsymbol{AG} \bigwedge_{\substack{0 \le i < n+l \\ f_i|_{G=b} \neq f_i}} (f_i|_{G=b} \cdot \boldsymbol{AX}\, y_i + \overline{f_i|_{G=b}} \cdot \boldsymbol{AX}\, \overline{y_i}) \qquad (15)$$

The generation of properties continues by generating properties (15) for all possible stuck-at faults $G = b$ in the circuit, for G equal to all possible flip-flop and logic gate outputs[2] and for b equal to logical values 0 in 1.

Generally, properties (15) are not valid in a FSM without stuck-at faults. If some of them is valid anyway, it means that the stuck-at fault $G = b$ considered has no effect on the circuit behaviour and consequently, the fault need not be refuted during testing. Only the invalid properties (15) are, therefore, interesting, and we continue the work with them alone.

4.4 Searching of Counterexamples

When searching counterexamples for invalid CTL formulas of the type (15), we in fact search for a witness of the following CTL formula because of the chosen way of searching and the use of DeMorgan's law:

$$\boldsymbol{E}\left[1\, \boldsymbol{U} \bigvee_{\substack{0 \le i < n+l \\ f_i|_{G=b} \neq f_i}} \overline{f_i|_{G=b} \cdot \overline{\boldsymbol{EX}\, \overline{y_i}} + \overline{f_i|_{G=b}} \cdot \overline{\boldsymbol{EX}\, y_i}}\right] \qquad (16)$$

It means that we search for a path leading from the initial state to a state in a state set with the following characteristic. For each state, it is not the case that a transition function f_i is equal to 1 (respectively, 0) in the state and that at the same time, there does not exist a successor state where the corresponding state variable y_i is equal to 0 (respectively, 1).

When a witness for the operator $\boldsymbol{EU}$ in formula (16) is found, we still must find a witness from that state on for the formula

$$\bigvee_{\substack{0 \le i < n+l \\ f_i|_{G=b} \neq f_i}} \overline{f_i|_{G=b} \cdot \overline{\boldsymbol{EX}\, \overline{y_i}} + \overline{f_i|_{G=b}} \cdot \overline{\boldsymbol{EX}\, y_i}}$$

We can choose one factor in the big disjuction and find a witness for it. Of course, we must choose a factor which is valid in the state reached when searching for a witness of the operator $\boldsymbol{EU}$. It follows that a witness of

$$\overline{f_i|_{G=b} \cdot \overline{\boldsymbol{EX}\, \overline{y_i}} + \overline{f_i|_{G=b}} \cdot \overline{\boldsymbol{EX}\, y_i}}$$

is searched for, where i is such that the formula is valid there. This can be done by finding a counterexample for the CTL formula

$$f_i|_{G=b} \cdot \overline{\boldsymbol{EX}\, \overline{y_i}} + \overline{f_i|_{G=b}} \cdot \overline{\boldsymbol{EX}\, y_i}$$

[2] The extended circuit inputs need not be considered since the inputs are added artificially and stuck-at faults cannot occur on them.

There are two possibilities. The transition function $f_i|_{G=b}$ can either be valid or invalid in the current state. We, therefore, continue along one of the following paths:

1. If $f_i|_{G=b}$ is valid, we search for a counterexample of the formula $\overline{\boldsymbol{EX}\overline{y_i}}$, which consequently means searching for a witness of the CTL formula $\boldsymbol{EX}\overline{y_i}$. A successor of the current state in which the value of state variable y_i is 0 has to be found.

2. If $f_i|_{G=b}$ is invalid, we search for a counterxample of the formula $\overline{\boldsymbol{EX}y_i}$, which consequently means searching for a witness of the CTL formula $\boldsymbol{EX}y_i$. A successor of the current state in which the value of state variable y_i is 1 has to be found.

A counterexample found is given in the form of a sequence of state variable values of the extended FSM in the states starting with the initial one and continuing along the counterexample path. Every state variable which is equal to the one in the original FSM has the same meaning in the latter and in the extended FSM. The state variables added at the inputs of the original FSM indicate the values we have to assign to the circuit inputs in order for the circuit execution to follow the counterexample path. The state variables added at the outputs of the original FSM indicate the values the real outputs of the good circuit will have in the states on the path.

It should be noted that the circuit outputs in the form of the added variables of the extended FSM are delayed for one step. The current circuit output values occur in the added variables in the next step. This is because a state variable gets a value determined by its state function in the next state, whereas the outputs get their value in a moment.

The test pattern which belongs to the counterexample found is a sequence of values of the state variables added at the inputs. The values in the last state on the counterexample path can be left out because the circuit input values in the last step do not matter. The circuit is tested as follows. The input values from the test pattern are set on the circuit inputs one after another. When its end is reached, we check if values of the state variables (flip-flops) and circuit outputs are equal to those in the last state of the counterexample.

If all the values are equal, the absence of the stuck-at fault considered is confirmed. Afterwards, we reset the circuit and repeat the whole procedure with the test pattern for the next stuck-at fault, and so on until the last possible stuck-at fault has been considered.

5 Experimental results

Experiments were done on a server with *AMD Athlon/800* processor, 512 MB of physical memory, and 1.4 GB of virtual memory under *Linux* operating system. We have used our own BDD package ([7]), which is an efficient *ite*-based implementation of reduced ordered binary decision diagrams with complemented edges.

We generated test patterns for some ISCAS benchmark circuits. Results are shown in Table 1. From left to right the columns in Table 1 refer to the circuit name, number of potential stuck-at faults, joint length of all test patterns, maximal length of test patterns, and the maximal number of BDD nodes whenever generated with the CPU time in seconds.

Table 1: Test pattern generation for ISCAS benchmark circuits

circuit	# stuck-at faults	joint length	max length	# BDD nodes	CPU time [s]
s27	38	58	3	604	0.00
s208.1	248	5474	257	50606	8.17
s298	296	1158	11	41582	1.92
s344	412	932	8	48208	7.67
s349	414	932	8	48087	6.69
s382	388	5387	102	99775	31.01
s386	372	1034	10	45819	2.29
s444	434	5990	102	182597	39.05
s526	458	7749	102	286215	79.22
s526n	460	7771	102	286183	80.24
s641	962	1787	5	1698296	362.29
s713	986	1807	5	1698421	363.47
s820	700	3090	12	257285	185.28
s832	696	3056	12	256203	179.73
s953	972	4667	11	224760	336.28
s1196	1178	2359	4	139827	69.34
s1238	1136	2254	4	134801	60.12
s1488	1410	6847	22	107989	154.61
s1494	1398	6743	22	108057	170.19

The number of potential stuck-at faults is approximately proportional to the number of flip-flops and gates in the circuits. The maximal length of test patterns depends on the number of steps necessary to set such values in flip-flops that stuck-at fault will demonstrate at computation of the next state. Joint length of all test patterns is a plain sum of lengths of single test patterns. It is very difficult to say anything general about the joint length.

We examined time and space complexity of the test pattern generation on a series of parametric up/down counters. Results we obtained are shown in Table 2, where the number of potential stuck-at faults, joint length of all test patterns, and maximal number of BDD nodes whenever generated with the CPU time in seconds for every counter size n are shown. Both time and space complexities are *less* than exponential.

We did not compare our results obtained with test pattern generation with results of other authors since we did not manage to find any contribution where authors would have generated test patterns for searching stuck-at faults with symbolic model checking. In the most similar example [4] they used symbolic state space traversal but not symbolic model checking.

Table 2: Test pattern generation for parametric counter

n	# stuck-at faults	joint length	# BDD nodes	CPU time [s]
10	102	156	5232	0.06
20	202	316	39312	0.42
30	302	476	78499	1.58
40	402	636	129859	4.31
50	502	796	213819	10.32
60	602	956	338379	60.19
70	702	1116	511539	250.02
80	802	1276	741299	488.44
90	902	1436	1045840	823.04
100	1002	1596	1424382	1287.52
110	1102	1756	1884720	1890.39
120	1202	1916	2434862	2709.84
130	1302	2076	3082801	3781.46
140	1402	2236	3836544	5160.15
150	1502	2396	4704084	6926.01
160	1602	2556	5693423	9169.76
170	1702	2716	6812565	11867.43
180	1802	2876	8069514	15189.49

6 Conclusions

We developed methods for fully automatic test pattern generation (ATPG) for discovering single stuck-at faults in synchronous sequential digital circuits. Test patterns are based on counterexamples obtained by symbolic model checking. For every possible stuck-at fault, a suitable property is generated. When the property is invalid a counterexample is found.

The described algorithms are realized in the form of a computer program for ATPG for discovering of single stuck-at faults or proving their absence. The program is based on a home-made package for manipulating FSMs which is also based on a fully home-made very efficient package for manipulating Boolean functions represented by BDDs [7].

We illustrated the usage of presented algorithms by generating test patterns for some ISCAS benchmark circuits and a parametric up/down counter. Results from the latter one also indicate time and space complexity of the algorithms.

There are a lot of possibilities for future work. The most interesting would be development of methods where stuck-at faults would manifest only at circuit outputs. Since also other types of faults can occur in a circuit, it would be interesting to discover also them. It might be useful to find out also which stuck-at fault is present in the circuit if presence of one is detected.

References

[1] Zmago Brezočnik, Aleš Časar, and Tatjana Kapus. Efficient Symbolic Traversal Algorithms using Partitioned Transition Relations. In Zmago Brezočnik and Tatjana Kapus, editors, *Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design*, pages 146–155, Maribor, Slovenia, June 1996.

[2] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[3] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.

[4] Gianpiero Cabodi, Paolo Camurati, and Stefano Quer. Symbolic forward/backward traversals of large finite state machines. *Journal of Systems Architecture*, 46:1137–1158, 2000.

[5] Aleš Časar. Verification of finite state machines with symbolic model checking. Master's thesis, University of Maribor, Faculty of Electrical Engineering and Computer Science, Maribor, Slovenia, June 1998. In Slovene.

[6] Aleš Časar, Zmago Brezočnik, and Tatjana Kapus. Formal Verification of Digital Circuits using Symbolic Model Checking. *Informacije MIDEM*, 30(3(95)):153–160, September 2000.

[7] Aleš Časar, Robert Meolic, Zmago Brezočnik, and Bogomir Horvat. Representation of Boolean Functions with ROBDDs. *Electrotechnical Review*, 59(5):299–307, December 1992. In Slovene.

[8] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, October 1994.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[10] Bogdan Dugonik. Metode za iskanje optimalnih vektorjev za diagnostično testiranje digitalnih vezij s pomočjo modela napak. Master's thesis, University of Maribor, Faculty of Electrical Engineering and Computer Science, Maribor, Slovenia, 1995. In Slovene.

mag. Aleš Časar, univ. dipl. inž. rač. in inf.
izr. prof. dr. Zmago Brezočnik, univ. dipl. inž. el.
izr. prof. dr. Tatjana Kapus, univ. dipl. inž. el.
Univerza v Mariboru
Fakulteta za elektrotehniko, računalništvo in informatiko
Smetanova 17
2000 Maribor
tel.: +386–2–22–07–211
fax: +386–2–25–11–178
email: {casar,brezocnik,kapus}@uni-mb.si

Prispelo (Arrived): 23.04.2002 *Sprejeto (Accepted): 28.06.2002*