FORMAL VERIFICATION OF DISTRIBUTED MUTUAL-EXCLUSION CIRCUITS

Robert Meolic, Tatjana Kapus, Bogdan Dugonik, Zmago Brezočnik
Faculty of Electrical Engineering and Computer Science, University of Maribor,
Smetanova ulica 17, SI-2000 Maribor, Slovenia

Keywords: Asynchronous circuit, Fundamental mode, Process algebra, Model checking, ACTL

Abstract: Distributed mutual-exclusion (DME) circuits are an interesting example of asynchronous circuits. They are composed of identical DME cells connected in a ring of arbitrary size. Each DME cell provides a connection point for one user, and all users compete for exclusive access to a shared resource. This paper reports about formal verification of two well-known DME circuit implementations. External behaviour of the circuits is described with a simple process, whereas the required properties are expressed with temporal logic ACTL. We were able to detect hazards and verify correctness of external behaviour of the circuits under the fundamental mode of operation.

Formalna verifikacija vezij za porazdeljeno medsebojno izključevanje

Ključne besede: Asinhrono vezje, Fundamentalni način, Procesna algebra, Preverjanje modelov, ACTL

Povzetek: Vezja za porazdeljeno medsebojno izključevanje (DME) so zanimiv primer asinhronih vezij. Sestavljena so iz enakih celic DME, povezanih v obroč poljubne velikosti. Vsaka od celic DME ponuja priključno točko za enega uporabnika in vsi uporabniki med seboj tekmujejo za izključen dostop do skupnega vira. V članku obravnavamo formalno verifikacijo dveh znanih izvedb vezja DME. Obnašanje vezij opišemo s preprostim procesom, zahtevane lastnosti pa s temporalno logiko ACTL. Na ta način smo lahko odkrili hazarde ter verificirali pravilnost obnašanja vezij v fundamentalnem načinu delovanja.

1 Introduction

Asynchronous circuits have been built and used for decades, and nowadays, large and efficient circuits can be constructed [5, 7, 17, 19, 20]. Techniques and methodologies for designing asynchronous circuits differ from those used with the synchronous approach. An important issue in asynchronous design is hazard removal. Because synchronization is performed without a global clock, unwanted signal changes can kill the circuit. Well known techniques for hazard-free synthesis, decomposition and verification of asynchronous circuits are based on modelling with flow tables [4], asynchronous finite state machines (AFSM), burst-mode state machines (BM), signal transition graphs (STG), state graphs (SG) [17], and also process algebrae. Some of algebraic approaches to verification of asynchronous circuits are Circal agents [2], CCS-like burst-mode specification [20], and DILL specification based on LOTOS [8]. An overview of the state-of-the-art in tools for asynchronous design can be found in [1].

This paper describes an algebraic approach to detecting hazards and verifying correctness of asynchronous circuits. Muller's model is used for modelling, and fundamental mode of operation is assumed. Section 2 gives an overview of asynchronous design and introduces Muller's model and hazards. Section 3 describes a simple process algebra and shows how it can be used for modelling individual gates. Section 4 describes the procedure for verification of asynchronous circuits. Section 5 introduces ACTL model checking. Section 6 presents two well-known implementations of DME circuits and reports about the results of their verification. In the conclusion we evaluate our work.

2 Asynchronous design

Circuits are composed of *gates* and *wires*. In this paper, the term gate refers to simple or complex elements for which only external behaviour is considered, and the term wire refers to connections between gates carrying binary signals. Regarding their operation, circuits can be classified into *combinational* and *sequential*. In a combinational circuit, output values of all gates are logic functions of current circuit input values. In a sequential circuit, some gate outputs depend also on a history of circuit input values. The memory effect is achieved with feedback loops.

Fundamental to an asynchronous design are assumptions about gate and wire delays. If delays are overestimated, the resulting circuit is likely to be inefficient and expensive. If they are underestimated, the design may not guarantee correct circuit operation. Delays can be *bounded* or *unbounded*. For bounded delay the upper bound is given, while the magnitude of unbounded delay is only known to be positive and finite. The delays can also be *pure* or *inertial*. In the latter case, short pulses are filtered out.

With regard to assumptions about delays, there are two widely used models for designing asynchronous circuits. *Huffman's model* supposes that gate and wire delays are bounded and known. Circuits designed with this model are called Huffman circuits. On the other hand, *Muller's model* supposes that wires have negligible delays in comparison to gates, which have inertial unbounded delays. Muller's model is typically used to design speed-independent circuits. Because of negligible wire delays, all *forks* in Muller's model are isochronic. This means that if a signal splits, all instances of this signal have equal delays. Thus, an output signal produced by a gate is equally delayed for all gates which consume it. By explicitly adding non-isochronic FORK elements, Muller's model can be extended to produce self-timed, delay-insensitive, and quasi-delay-insensitive circuits, where wire delays are important. Note that there are also other design methodologies for asynchronous circuits not based on the mentioned models (e.g. self-clocked circuits and micropipelines)

A simple Muller's model is presented in Figure 1. It represents the C-element, a standard building block used in many asynchronous systems, which was also introduced by Muller. The C-element changes its output only if both inputs are changed to 0 or 1. In the figure, the small boxes labeled with d_1, d_2, d_3, and d_4 are delay elements attached to the gate outputs, which are the only components in the circuit delaying signals. The figure clearly shows that in the Muller's model an output signal produced by a gate is equally delayed for all gates which consume it.

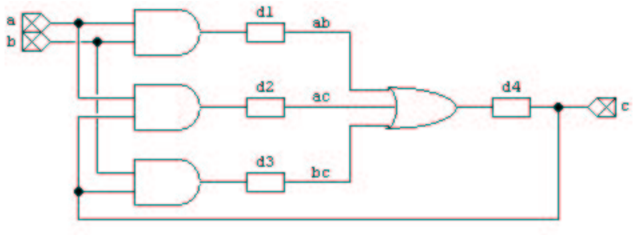


Figure 1: A gate-level implementation of the C-element

An important concept related to delays is circuit's mode of operation, which characterizes the interaction between a circuit and its environment. *Fundamental mode* of operation assumes that the environment will change the value of only one input signal at once and then wait until the circuit becomes stable. An asynchronous circuit is *stable* if no internal or output signal value can be changed without changing some input signal value. The opposite of fundamental mode is *input/output mode* of operation. In this mode, the environment can change values of input signals at any time. A third type of circuit's mode of operation is *generalized fundamental mode* or *burst mode*. There, signal changes are segregated in time forming input bursts (intervals where input signal values change) and output bursts (intervals where output signal values change). An output burst can be empty, whereas an input burst must contain at least one signal. Input and output bursts must alternate. Within a burst, the ordering of signal changes is not determined.

There are some anomalous types of asynchronous circuit behaviour, which the designers try to avoid. An example of a usually unwanted behaviour is a possibility that the circuit enters a closed loop of transitions without becoming stable. This can result in the circuit with oscilating outputs. A simple example of such a circuit and its simulation run are presented in Figure 2.

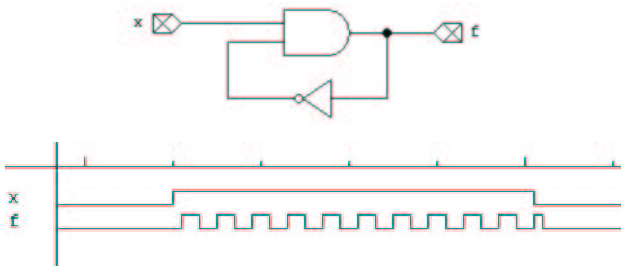


Figure 2: A circuit with oscilating output

If due to internal delays, a circuit can make an unwanted pulse called a *glitch* or can become stable with an unwanted combination of values on internal or output lines, we have a *hazard*. Hazards reflecting in glitches are classified with regard to their shape into *static* and *dynamic* hazards. Static hazard occurs when the signal is momentarily changed although it should remain the same. Dynamic hazard occurs if the signal oscillates before changing its value. A circuit which operates without hazards in the fundamental mode is called a *fundamental-mode circuit*.

For each hazard, there is a reason for its existence. In combinational circuits, three types of hazards are distinguished. *Logic hazards* are a property of particular implementation. A logic hazard exists in the circuit because for the same signal, two or more parallel paths through the circuit exist, which then reconverge. *Functional hazards* are a property of the logic functions which do not change monotonically during a sequence of particular input changes. The hazard arises when such inputs change simultaneously. Logic and functional hazards can both be either static or dynamic. A much different type of hazard is *delay hazard*. It occurs because a new input signal is applied before the circuit becomes stable. Logic hazards can always be avoided by redesigning the circuit. Functional and delay hazards can be removed only by engineering delays. Under fundamental and burst mode of operation, functional and delay hazards do not have an impact. In combinational circuits, hazardous behaviour is a transitory phenomenon, and if no new inputs are applied until the circuit stabilizes, then the correct outputs will be produced.

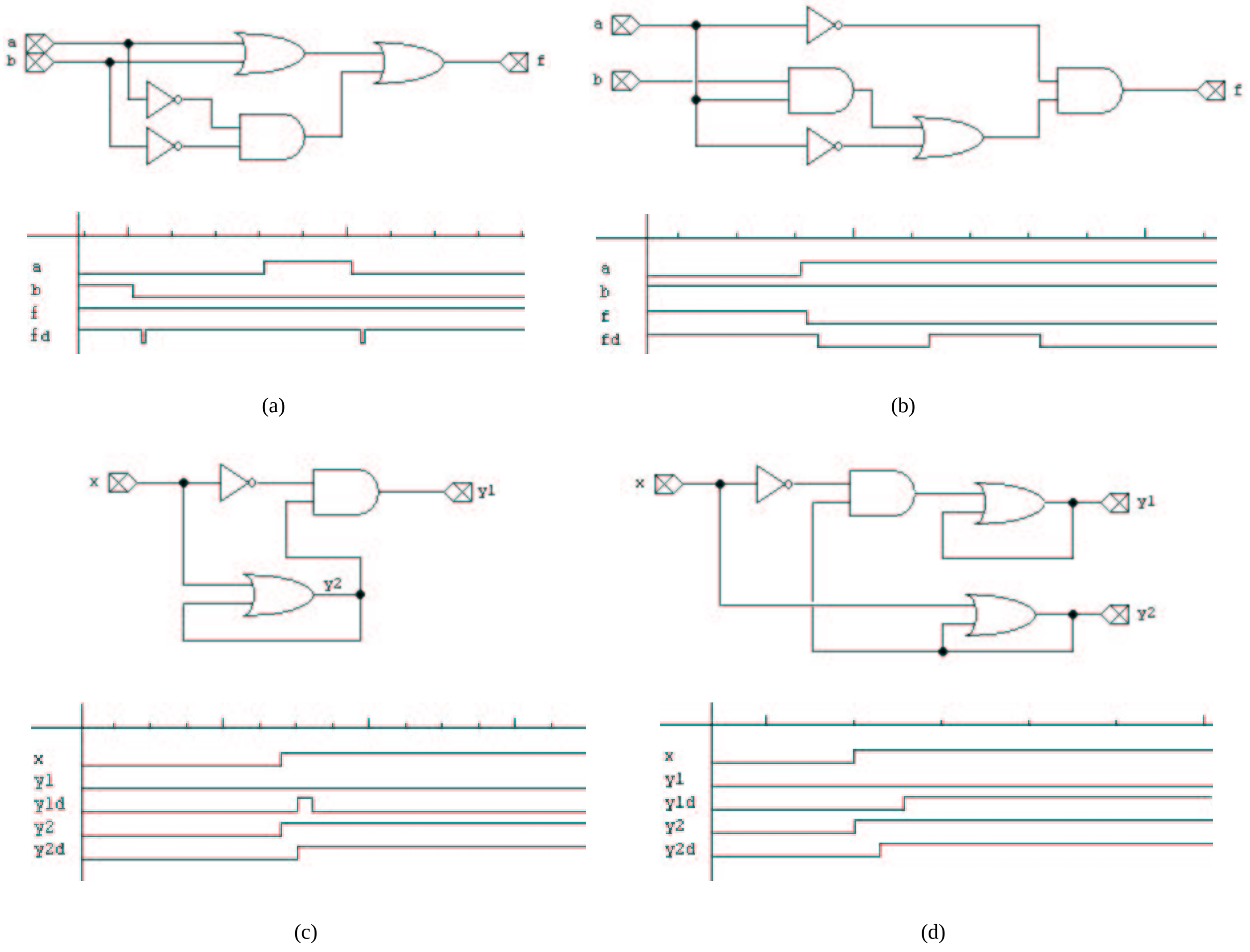


Figure 3: Simple circuits containing (a) static hazard, (b) dynamic hazard, (c) transient hazard, and (d) steady-state hazard.

In sequential circuits, additional *sequential hazards* can exist as a consequence of the order in which input signals and feedback signals are considered. Sequential hazard which results in a glitch is *transient hazard*. On the other hand, if due to delays the circuit can become stable with incorrect values of internal or even output signals, we have *steady-state hazard*. Both types of sequential hazards are an inherent property of sequential functions and not of the particular circuit implementation. Sequential hazards appear despite of fundamental and burst mode constraints.

Figure 3 shows circuits containing different types of hazards and their simulation runs. Signals f, $y1$, and $y2$ represent the behaviour of output signals without delays in the circuit, while signals fd, $y1d$, and $y2d$ represent their behaviour after introducing significant delays. Figures 3(a) and 3(b) show logic hazards. The static hazard in Figure 3(a) is obtained by using delayed inverters. The dynamic hazard in Figure 3(b) appears when the left AND gate is delayed and the top inventer is even more delayed. Figure 3(c) represents a transient hazard which appears if the AND gate gets the new value from the feedback line earlier than the new input value. The circuit in Figure 3(d) has steady-state hazard because after changing input x, the circuit without delays produces outputs $y1 = 0$, $y2 = 1$, whereas using a delayed inverter, it produces outputs $y1d = 1$, $y2d = 1$.

3 Representing circuits with processes

Process algebrae are widely used formalisms for modelling and verification of concurrent systems, e.g. communication protocols. In a process algebra, a system is described as a set of communicating processes. Among others, well known process algebrae are CCS (*Calculus of Communicating Systems*) introduced by R. Milner in 1980 and CSP (*Communicating Sequential Processes*) introduced by C. A. R. Hoare in 1985. We will use an algebraic approach which is similar to CCS and has some notation from CSP.

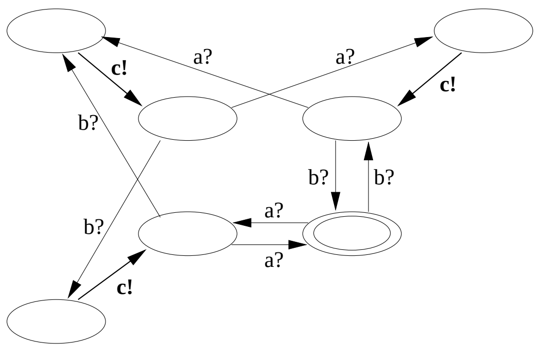
In our approach, processes are labelled directed graphs. Graph nodes are called states. An edge from state p to state q labelled with action α is called an α*-transition* or shortly a *transition* from state p to state q. If there exists an α*-transition* from a given state, we say that in this state the process can *perform* α-transition or that it can *perform* action α. The set of all actions which a process can perform is called the *alphabet* of the process. A sequence of transitions in the process is called a *path*. The alphabet of a process always includes a special action τ, which is used to model internal communications, not visible to an external observer. A transition with action τ is called *silent transition*. All actions others than τ are called *visible* actions. Visible actions are divided into input and output actions. The name of an *output* *action* always terminates by '!', e.g. $a!, b!, ...$ The name of an *input action* always terminates by '?', e.g. $a?, b?, ...$ Two actions whose names differ only in the last sign, e.g. $a?$ and $a!$, are called *complementary actions*. An action complementary to the given action α is denoted by $\overline{\alpha}$. A sequence of visible actions is called a *trace*. Two states p and q have *equivalent traces* if from them the same traces can be performed. Processes are *trace-equivalent* if their initial states have equivalent traces. A state without outgoing transitions is a *deadlock* state. If there exists a sequence of transitions from the initial state of a process to a state p, then state p is *reachable* in this process. Otherwise, the state is *unreachable* in this process. Other important concepts in process algebrae are strong equivalence, observational equivalence, and determinacy of processes [16].

Processes are used to represent external behaviour of circuits. Each transition in the process represents a change of a signal value and we do not distinguish whether the value changes from 0 to 1 or vice versa. A transition with an input action represents change of an input signal value, caused by the environment. A transition with an output action represents change of an output signal value, caused by the circuit. We will use interleaving semantics, i.e. a simultaneous occurence of two or more signal changes will be modeled by including multiple sequences of transitions, one for each permutation of changing signals.

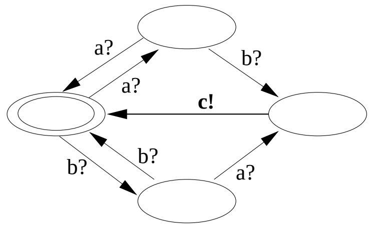
The process represents an external behaviour of the circuit if each of its traces starting in the initial state of the process corresponds to a possible sequence of changes of input and output signal values during the operation of the circuit in the given environment considering given initial values of all input and output signals. We are not interested in exact timing when the value of signals changes. If two circuits have the same possible sequences of changes, they have the same external behaviour, although one of them is much faster than other. Therefore, two processes represent equal external behaviour iff they are trace-equivalent. However, delays are important because different sequences of changes can be possible in the same asynchronous circuit if the delay of gates changes. Some traces in the process may correspond to sequences of changes possible only with particular arrangements of gate delays and not all of them. Note that we do not require fixed delays. Gate delays can change during the operation of the circuit. Because each action represents just a change of signal value, we cannot uniformly associate a circuit to the given process without knowing the initial value of all input and output signals. Namely, to determine whether performing a particular transition represents a rising or falling edge, one must know the initial value of that signal and follow its changes.

With regard to the level of abstraction, the process can represent more or less details on the circuit's internal behaviour and its connections with the enviroment. For the purpose of finding hazards and verification of asynchronous circuits under the fundamental mode of operation, we introduce a special form of processes called circuit models. A *circuit model* is determinate process without unreachable states and without silent transitions. All circuit models representing external behaviour of the same circuit in the same environment and with the same initial values of input and output signals are strongly equivalent. Among them, the one with minimal number of states will be identified and used for verification.

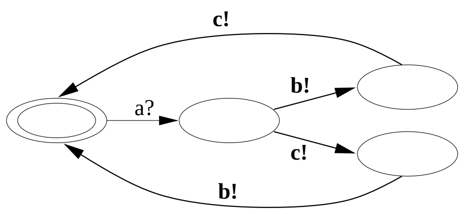
Circuit models in Figure 4 represent external behaviour of an AND gate, the C element, and the FORK element, respectively. Further, they will be used to build DME cells. The AND gate and the C element have two input signals (actions $a?$ and $b?$) and one output signal (action $c!$). The FORK element is a frequently used gate in asynchronous circuits. It has one input signal (action $a?$) and two output signals (actions $b!$ and $c!$). The value of both outputs changes simultaneously after the change of the input value. Simultaneous change of two outputs is modelled by the ability of performing corresponding output actions in both orders. The presented circuit models describe external behaviour of gates under the fundamental mode of operation with all input and output signals initially set to 0.



(a)



(b)



(c)

Figure 4: Circuit model of (a) two-input AND gate, (b) the C-element, and (c) the FORK element

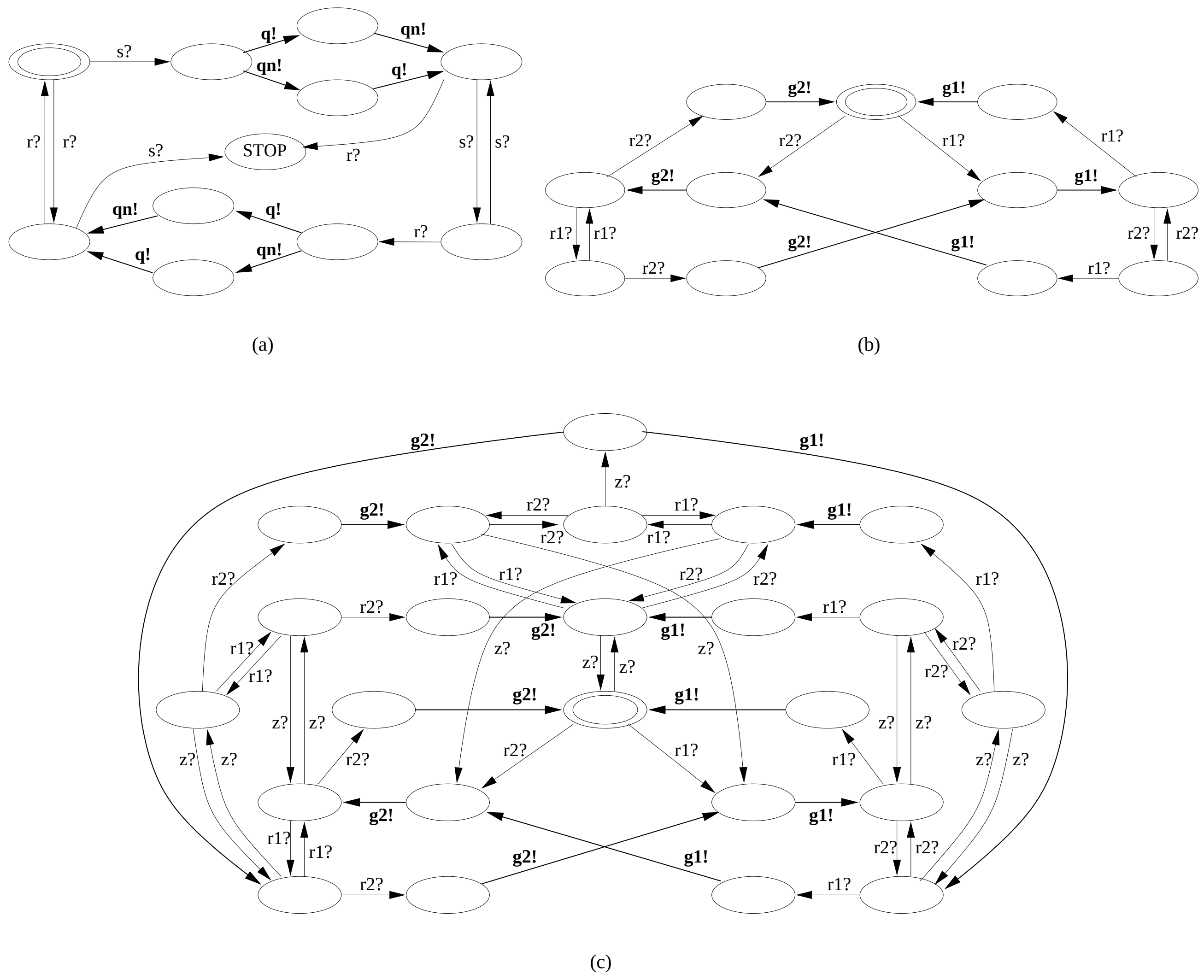


Figure 5: Circuit model of (a) RS flip-flop, (b) ME element, and (c) ME element with inhibit signal.

DME circuits contain some more complex gates, too. The circuit model in Figure 5(a) describes external behaviour of RS flip-flop with two inputs denoted by s (set) and r (reset) and two outputs denoted by q and qn, where qn is inverted q. Initially, s, r, and q are equal to 0. When input signal s is set to 1, output q becomes 1. When input signal r is set to 1, then output q becomes 0. Inputs r and s must never be set to 1 at the same time. The circuit model contains a deadlock state, which is not reached if the RS flip-flop is properly driven by its environment.

Mutual exclusion (ME) element is a non-deterministic element used in asynchronous circuits as an arbiter. Two different versions of ME elements appear in DME circuits. In the first one, the ME element has two inputs denoted by $r1$ and $r2$ and two outputs denoted by $g1$ and $g2$. If input signal $r1$ is set to 1 and output $g2$ is not set to 1, then output $g1$ becomes 1. When input signal $r2$ is set to 1 and output $g1$ is not set to 1, then output $g2$ becomes 1. If inputs $r1$ and $r2$ are simultaneously set to 1, then the ME element non-deterministically chooses one output and sets it to 1. In another version, the ME element has an additional input z intended for an inhibit signal, which prevents the grant to new requests until the request that caused the previous grant is removed. The circuit models we used to describe external behaviour of the ME element without and with inhibit signal, where all input and output signals are initially equal to 0, are presented in Figures 5(b) and 5(c), respectively.

4 Verification of asynchronous circuits

The verification of an asynchronous circuit starts by representing all necessary gates with circuit models. Then, they are composed together using *parallel composition with multi-way synchronisation* [8, 20]. The compound process represents external behaviour of the circuit. The behaviour of gates in the circuit is asynchronous, but they are not completely independent. Complementary actions in different circuit models must be performed simultaneously as the value of a signal cannot change only in one part of a wire. Simultaneous performance of complementary actions is called synchronisation between circuit models. During the composition, two types of transitions with visible actions are distinguished. The transitions used for synchronisation with the system's environment are *external transitions*, whereas the transitions serving for synchronisation between processes in the system are *internal transitions*.

The parallel composition of circuit models will not return a meaningful result if:

- signals do not have unique names,
- the outputs of two or more gates are connected together,
- there exists a gate directly driving itself,
- an output signal observable by the environment is used as a feedback signal into the circuit.

The first two situations are straighforward. A gate is not allowed to drive itself because it is expected that at least two different circuit models cooperate in synchronisation. An output signal observable from the environment is not allowed to be a feedback signal into the circuit because the same visible action cannot be used in internal and external transitions. The last two requirements have an impact on the preciseness of the verification. For example, circuits in Fig. 3(c) and Fig. 3(d) cannot be verified in their original form without adding FORK elements after the OR gates which produce signals $y1$ and $y2$.

The next verification step is a transformation of the obtained compound process into a circuit model which represents the external behaviour of the assembled circuit under the fundamental mode of operation. This transformation consists of two steps, removal of redundant traces and determinization of the process. We call the first operation *fundamental-mode reduction* and it removes:

- all transitions with an input action from states where a silent transition can be performed,
- all transitions with an input action from states where a transition with an output action can be performed.
- all silent transitions from states where a transition with an output action can be performed. ■

An example of the circuit model obtained by composing circuit models of individual gates is presented in Figure 6. It is a circuit model which represents the external behaviour of the circuit with oscilating output under the fundamental mode of operation. In the initial state, it can perform input action $x?$. After changing input x, output f begins to oscilate. Afterwards, the circuit cannot change the value of input signal again because it never becomes stable.

One of the goals of our approach was hazard detection. It can be done by finding particular patterns in the circuit models which represent unwanted external behaviour. This patterns are simpler for the circuits containing only one output signal. Therefore, in the case of a circuit with many outputs we made the verification separately for each output. A hazard resulting in a glitch is present in the circuit if in its circuit model, after performing an output action, the same output action can be performed again before any input action is performed. Hence, most hazards can be easily detected by looking for such sequences of transitions, although in this

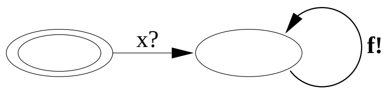


Figure 6: Circuit model of the circuit with oscilat. output

way they cannot be classified into logic, functional and transient hazards. However, static and dynamic hazards can be distinguished, as a static hazard results in an output action successively repeated an even number of times, whereas in the case of a dynamic hazard, the output action is repeated an odd number of times.

It is more complicated to detect steady-state hazards. A steady-state hazard is present in the circuit if after a particular sequence of input signals with some arrangements of delays an output signal appears, but with other arrangements of delays it does not. This always results in a state in the circuit model where both input and output actions can be performed. However, not all such states are a consequence of steady-state hazards:

- it can also indicate a glitch which appears with some arrangements of delays, but not with all of them,
- if some actions representing output signals have been abstracted from the model, then it can also indicate a non-deterministic behaviour of the circuit, which in particular situations produces the retained output signal and sometimes the abstracted one.

Glitches which appear only with some arrangements of delays and not with all of them are avoidable by engineering delays. We will call them *avoidable* hazards. Not all hazards are avoidable (i.e. they are *unavoidable*) because delays can be set only to gates and not to wires. An example of avoidable and unavoidable static logic hazard is shown in Figure 7. An example of a circuit with non-deterministic behaviour is the ME element with inhibit signal in Figure 5(c).

Circuit models in Figure 8 represent the external behaviour of circuits with hazards. They were obtained by composing circuits in Figure 3. The circuit model in Figure 8(a) shows that in the circuit in Figure 3(a), if the value of a and b is initially assumed to be 0, after changing any of them, output f may change two times consecutively. This represents a static hazard. Afterwards, if the same input is changed again, another static hazard may appear. All hazards in this circuit are avoidable. The circuit model in Figure 8(b) shows that in the circuit in Figure 3(b), if a and b have initial value 0 and the value of a changes, no hazards will occur. However, the situation is quite different after b is changed to 1. Then, each change of a may be followed by three consecutive changes of output f, which is a dynamic hazard. This hazard is avoidable, too. The circuit model in Figure 8(c) indicates a transient hazard in the circuit in Figure 3(c). If x is initially assumed to be 0 and then changes to 1, a static hazard may appear on ouput $y1$. The hazard is avoidable and it is possible only after the first change of x. The circuit models in Figure 8(d), 8(e), and 8(f) represent the behaviour of the circuit in Figure 3(d). The initial value of all signals is assumed to be 0. After x changes to 1, the

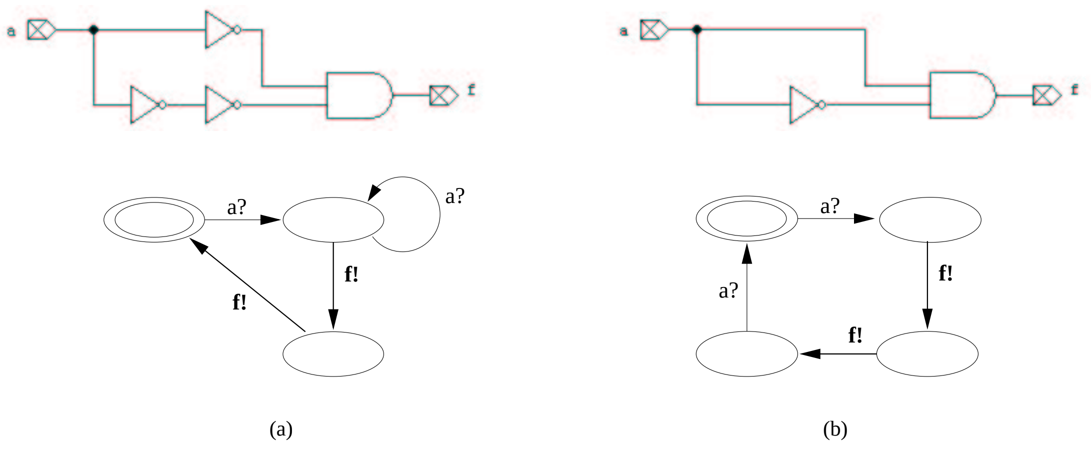


Figure 7: A circuit with (a) avoidable and (b) unavoidable static logic hazard

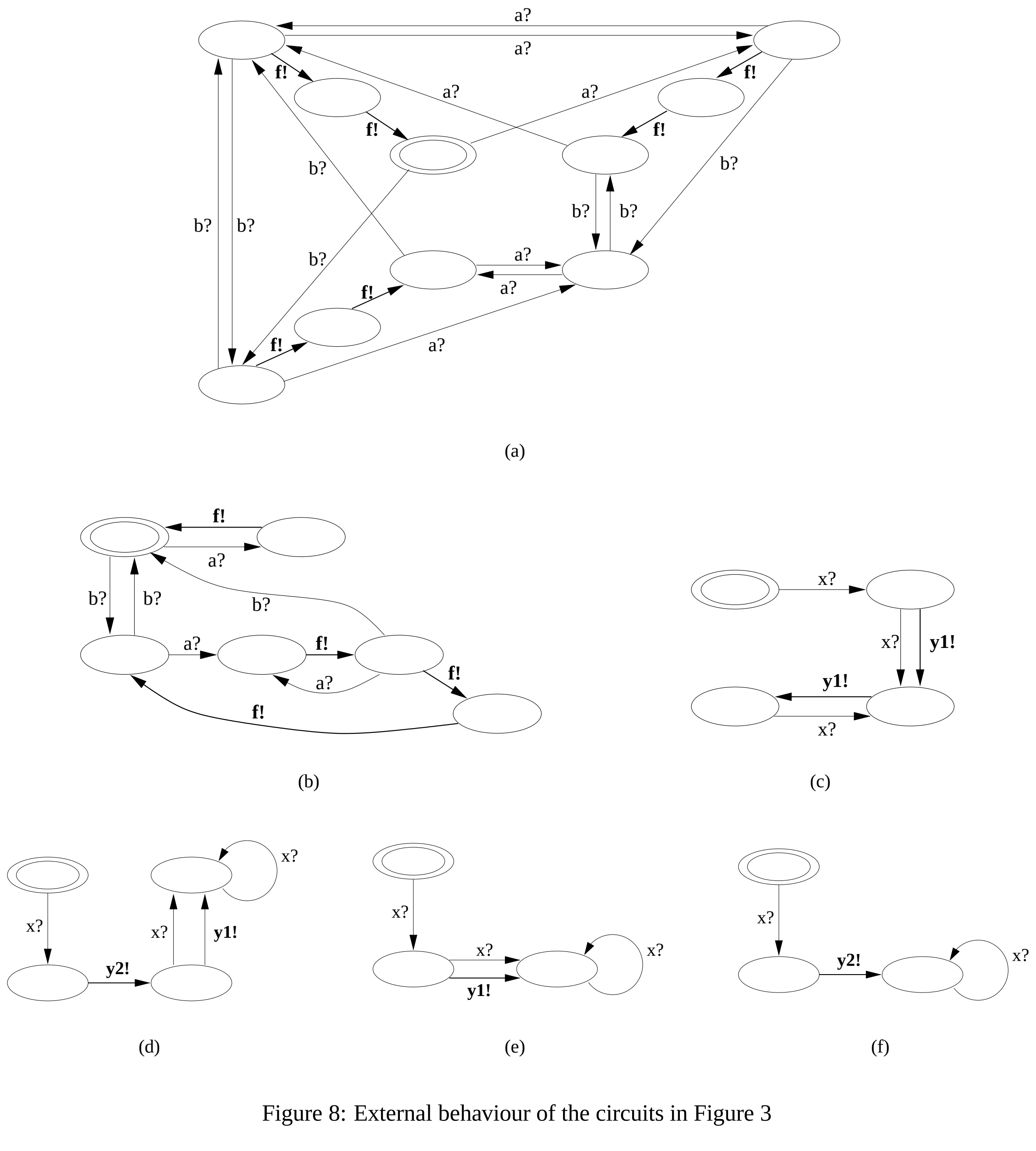


Figure 8: External behaviour of the circuits in Figure 3

value of $y1$ with some arrangements of delays changes and with some arrangements it does not. In both cases, further changes of x do not affect $y1$ anymore. There are no hazards on line $y2$. Figure 8(d) shows the circuit model containing both output signals, while in the other two circuit models one output signal is abstracted away.

5 ACTL model checking

In Section 4 we detected hazards in asynchronous circuits by looking for particular patterns in the circuit models. This step of verification can be effectively done by *model checking*, which is a powerful technique for checking properties of processes. It is also used for verification of liveness and safety properties of asynchronous circuits.

We will specify circuit properties with *action computation tree logic* (ACTL), which is a propositional branching time temporal logic [6, 14]. The syntax of ACTL formulae includes constants $true$ and $false$, standard Boolean operators **NOT**, **AND**, **OR**, *path quantifiers* **E** ("there exists a path") and **A** ("for all paths"), and *temporal operators* **U** ("until"), **W** ("unless"), **X** ("for the next transition"), **F** ("for some transition in the future"), and **G** ("for all transitions in the future").

ACTL formulae are state formulae. A state where ACTL formula φ is valid will be called φ-state. ACTL formulae are constructed from action and path formulae. An action for which action formula χ is valid will be called χ-action. A transition from state p to state q where action formula χ is valid for the action executed during this transition and ACTL formula φ is valid in state q will be called (χ, φ)-*transition*. In a process, path formulae are evaluated as follows:

- Path formula $\mathbf{X}\{\chi\}\ \varphi$ is valid on path π iff the first transition on this path is a (χ, φ)-transition.
- Path formula $\mathbf{F}\{\chi\}\ \varphi$ is valid on path π iff there exists a (χ, φ)-transition on this path.
- Path formula $\mathbf{G}\ \varphi\{\chi\}$ is valid on path π iff ACTL formula φ is valid in the first state of this path and all transitions on the path are (χ, φ)-transitions.
- Path formula $[\varphi\ \{\chi\}\ \mathbf{U}\ \{\chi'\}\ \varphi']$ is valid on path π iff ACTL formula φ is valid in the first state of this path and the path begins with a finite sequence of (χ, φ)-transitions followed by a (χ', φ')-transition.
- Path formula $[\varphi\ \{\chi\}\ \mathbf{W}\ \{\chi'\}\ \varphi']$ is valid on path π iff formula $[\varphi\{\chi\}\mathbf{U}\{\chi'\}\varphi']$ is valid on this path or formula $\mathbf{G}\ \varphi\{\chi\}$ is valid on this path.

The given rules are used for finite and infinite paths. In ACTL, each path formula is always immediately preceded by a path quantifier. Path quantifier **E** requires that the property expressed by the path formula is valid for at least one path starting in the given state. On the other hand, path quantifier **A** requires that the property expressed by the path formula is valid for all paths starting in the given state. In a deadlock state, formulae $\mathbf{EG}\ \varphi\{\chi\}$, $\mathbf{AG}\ \varphi\{\chi\}$, $\mathbf{E}\ [\varphi\ \{\chi\}\ \mathbf{W}\ \{\chi'\}\ \varphi']$, and $\mathbf{A}\ [\varphi\ \{\chi\}\ \mathbf{W}\ \{\chi'\}\ \varphi']$ are valid only if the state is a φ-state. Formulae $\mathbf{EX}\ \{\chi\}\ \varphi$, $\mathbf{AX}\ \{\chi\}\ \varphi$, $\mathbf{EF}\ \{\chi\}\ \varphi$, $\mathbf{AF}\ \{\chi\}\ \varphi$, $\mathbf{E}\ [\varphi\ \{\chi\}\ \mathbf{U}\ \{\chi'\}\ \varphi']$, and $\mathbf{A}\ [\varphi\ \{\chi\}\ \mathbf{U}\ \{\chi'\}\ \varphi']$ are invalid in all deadlock states. We will take that an ACTL formula is valid in process P iff it is valid in its initial state.

In ACTL formulae, the constant *true* can be omitted in many cases, for example:

$$\begin{aligned} \mathbf{E}[true\ \{\chi\}\ \mathbf{U}\ \{\chi'\}\ \varphi'] &= \mathbf{E}\ [\{\chi\}\ \mathbf{U}\ \{\chi'\}\ \varphi'] \\ \mathbf{A}[\varphi\ \{true\}\ \mathbf{U}\ \{true\}\ \varphi'] &= \mathbf{A}\ [\varphi\ \mathbf{U}\ \varphi'] \end{aligned}$$

There are also two widely accepted abbreviations of ACTL operators:

$$\begin{aligned} <\chi>\varphi &= \mathbf{EX}\{\chi\}\ \varphi \\ [\chi]\ \varphi &= \neg\mathbf{EX}\{\chi\}\ \neg\varphi \end{aligned}$$

ACTL formula $<\alpha>\ \varphi$ is valid in the given state iff there exists a transition with action α from that state to a state where ACTL formula φ is valid. ACTL formula $[\alpha]\varphi$ is valid in the given state iff all transitions with action α from that state lead to a state where ACTL formula φ is valid.

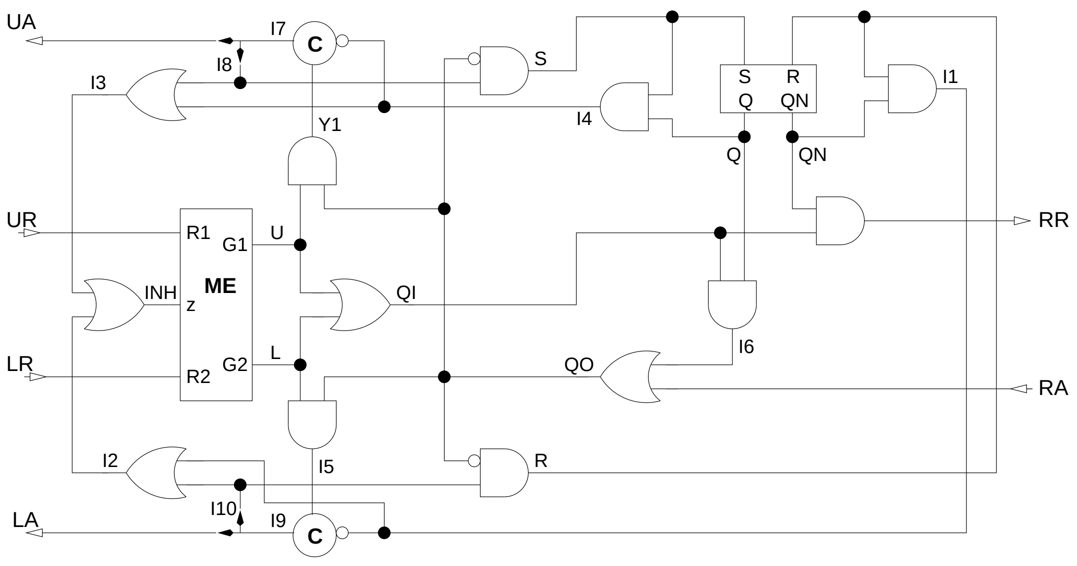
Suppose that the alphabet of the process contains actions $a?$, $b?$, and $f!$. Here are some ACTL formulae which can be used for checking properties of this process:

- There is no deadlock state: $\mathbf{AG}\ \mathbf{AF}\ \{true\}$
- At any moment, ouput action $f!$ will be performed in the future: $\mathbf{AG}\ \mathbf{AF}\ \{f!\}$
- There is no state where output action $f!$ can be performed succesively two times:
  $\mathbf{NOT}\ \mathbf{EF}\ \{f!\}\ <f!>\ true$
- There is no state where output action $f!$ and also input action $a?$ or $b?$ can be performed:
  $\mathbf{NOT}\ \mathbf{EF}\ ((<f!>\ true)\ \mathbf{AND}\ (<a?\ \mathbf{OR}\ b?>\ true))$

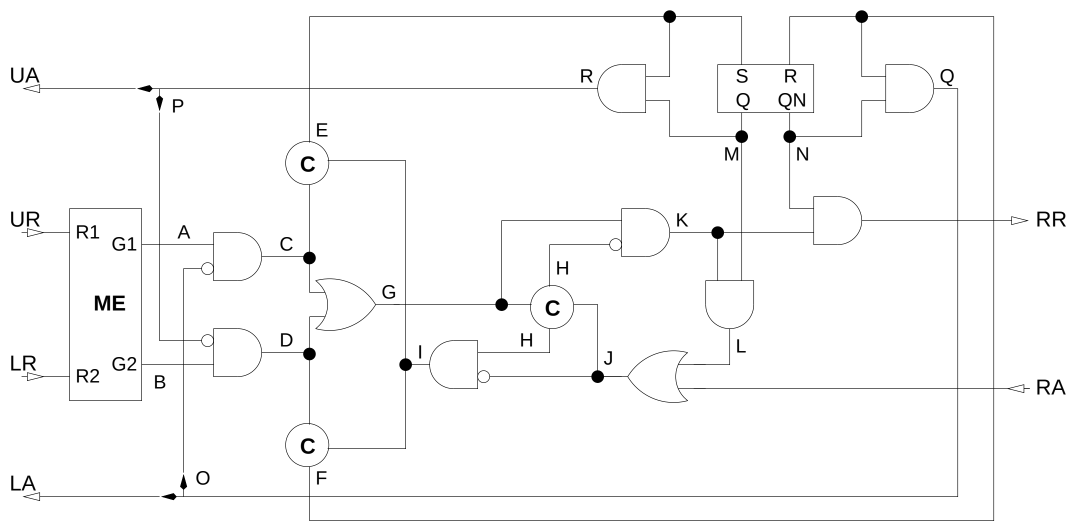
6 Results from verification of DME circuits

We verified *distributed mutual-exclusion* (DME) circuits. They are composed of DME cells connected in a ring. DME cells work by passing a token around the ring. The ownership of the token is determined by output signal q of the RS flip-flop. The token is exchanged via the request and acknowledge signals with the left cell (LR and LA) and with the right cell (RR and RA). The users gain exclusive access to the resource via the request and acknowledge signals UR and UA. The DME circuit was originally proposed by Martin in 1985 [11]. Martin's design does not work correctly under the input/output mode of operation [10]. In 1988, Burns gave a simpler design of the DME cell [3]. It was later slightly modified by McMillan and became a standard benchmark for asynchronous design verification tools [12, 13, 18]. The DME cells from Martin and McMillan are presented in Figure 9.

In Martin's design, the token indicates which user has last accessed the shared resource. If a DME cell receives a request but does not have the token, it notices this to its right



(a)

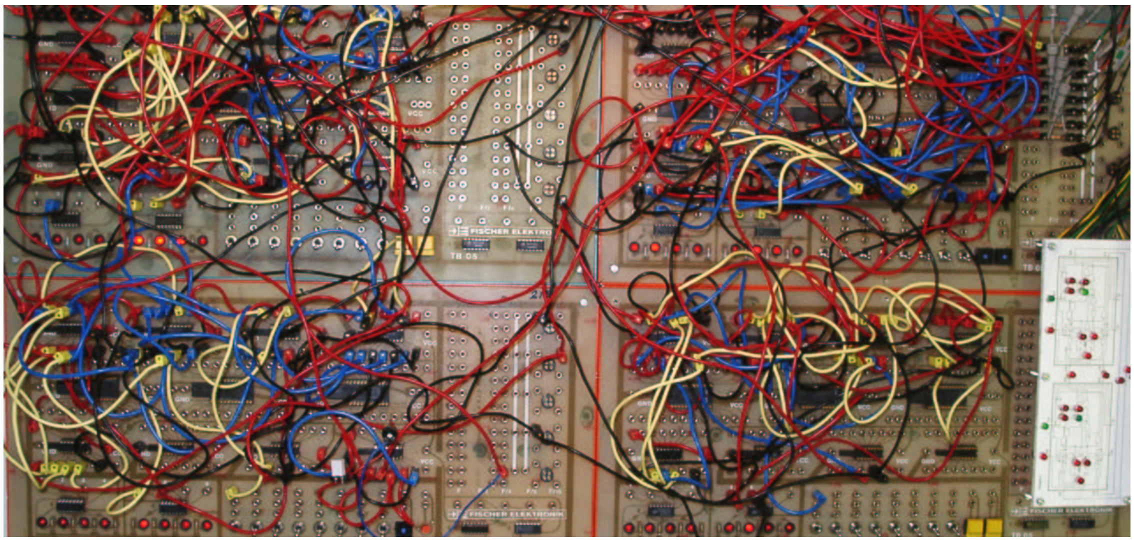


(b)

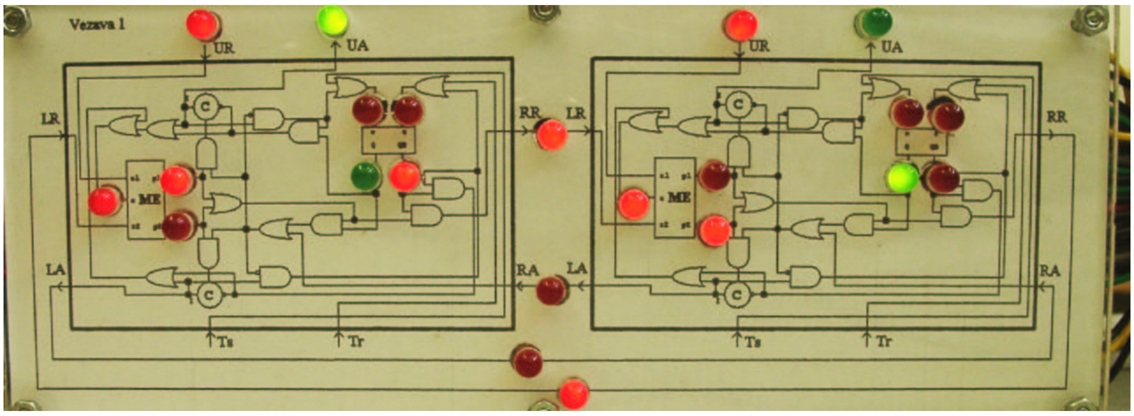
Figure 9: A cell of the DME circuit (a) as proposed by Martin [11], and (b) as proposed by McMillan [12]

neighbour via the RR signal. When a DME cell gets a request, either from the user via the UR signal or from its left neighbour via the LR signal, the DME cell attempts to satisfy the demand. If a DME cell has the token and no granted request is outstanding, then it sends an acknowledgement, either via the UA or LA signal, as appropriate. When a DME cell establishes it can approve user access, it immediately sends the UA signal to the user. If this DME cell does not have the token, then the token is transferred to it after the user removes the request. In McMillan's design, a user request is never acknowledged by a DME cell which does not possess the token. The token is transferred first, which makes the response to a user request slower. Moreover, McMillan's design has slower response times regardless of the token position because there the request signal has a longer path through the decision logic.

The verification was done with *Efficient Symbolic Tools* (EST), our BDD-based tool for symbolic verification of concurrent systems [15]. We started by modelling all necessary gates and composing them in DME cells. Afterwards, possible hazards in each DME cell were examined. Finally, we composed DME cells into rings of different sizes and checked correctness of external behaviour of the obtained circuits. To confirm the results of formal verification we also implemented the circuits on the prototype board (Figure 10) and tested their behaviour by measurements with HP 1652B Logic Analyzer. In Figure 10(b), the reader may notice that some logic for initialisation of the RS flip-flops was added for testing. The test runs obtained for the ring composed of two DME cells are given in Figure 11. Signals UR, UA, Q, RR, LA, Z, S, R, G1, and G2 belong to the first DME cell, while others belong to the second one.

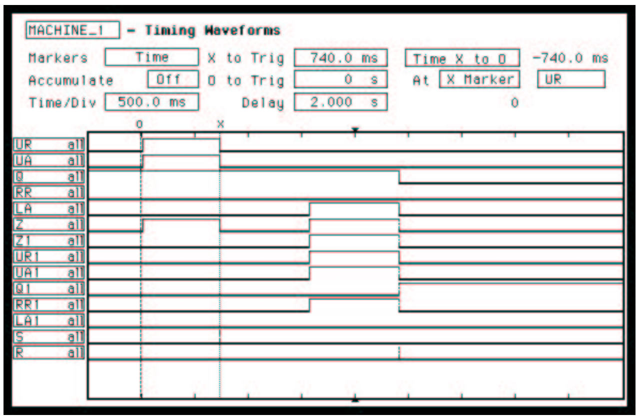


(a)



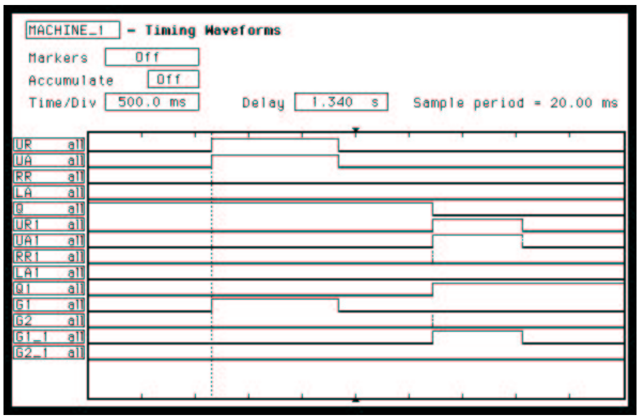
(b)

Figure 10: The ring of two Martin’s DME cells: (a) implementation with gates and wires, (b) LED indicators for testing



```
MACHINE_1  - Timing Waveforms
Markers     Time       X to Trig  740.0 ms   Time X to 0   -740.0 ms
Accumulate  Off        0 to Trig  0 s        At X Marker   UR
Time/Div    500.0 ms   Delay      2.000 s                  0
```

(a)



```
MACHINE_1  - Timing Waveforms
Markers     Off
Accumulate  Off
Time/Div    500.0 ms   Delay  1.340 s   Sample period = 20.00 ms
```

(b)

Figure 11: Test run of the ring composed of (a) two Martin’s DME cells, (b) two McMillan’s DME cells

In the circuit model of DME cell, input signal changes were represented with actions `ur?`, `lr?`, and `ra?`, and output signal changes were represented with actions `ua!`, `la!`, and `rr!`. We created 3 different circuit models for each DME cell. Each circuit model represents the behaviour of one output signal, whereas the other two are abstracted away. This makes the verification simpler. To find hazards, we used the following ACTL formulae, which are for simplicity here presented using macros, although EST does not support them yet:

```
\define IN (ur? OR lr? OR ra?)
\define OUT (ua! OR la! OR rr!)

# Static hazards
NOT EF {IN} <OUT> <OUT> <IN> true

# Dynamic hazards
NOT EF {IN} <OUT> <OUT> <OUT> <IN> true

# Steady-state hazards
NOT EF {IN}
  ((<IN> true) AND (<OUT> <IN> true))
```

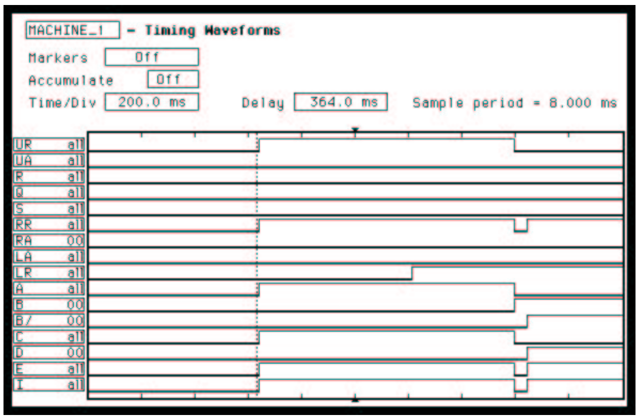
For any formula which is invalid for a given model, EST shows a counterexample. To find all hazards on the particular output signal effectively, model checker should find all counterexamples (e.g. tree-like counterexamples [9]). Unfortunately, EST is not capable of that, and therefore we helped us with an iteration method. For each hazard found, we deleted outgoing transitions from the state where the hazard began and then checked the same ACTL formula again. We repeated this until the formula became valid. With the presented formulae only hazards containing two or three successive changes of an output signal can be detected, but we also verified that the model contains no other hazards. In Martin's DME cell, which initially does not possess the token, we found static hazards on output signals UA (1-4) and RR (5-8) and steady-state hazards on output signals UA (9), LA (10-11), and RR (12-20):

1. ra?,ur?,ua!,ra?,ua!,ua!
2. ra?,ur?,ua!,lr?,ra?,ua!,ua!
3. ra?,ur?,ua!,ur?,ur?,ra?,ua!,ua!
4. ra?,ur?,ua!,ur?,ur?,lr?,ra?,ua!,ua!
5. ur?,rr!,lr?,ur?,rr!,rr!
6. lr?,rr!,ur?,lr?,rr!,rr!
7. ra?,lr?,rr!,ur?,lr?,rr!,lr?,ra?,rr!,ur?,rr!,rr!
8. ra?,lr?,rr!,ur?,lr?,rr!,lr?,ra?,rr!,lr?,rr!,rr!
9. ra?,lr?,ur?,lr?,lr?,ra?,ra?,[ua!]
10. ra?,ur?,ur?,ur?,lr?,ra?,[la!]
11. ra?,lr?,la!,ur?,lr?,lr?,ra?,la!,ra?,[la!]
12. ra?,ur?,rr!,ur?,rr!,ur?,lr?,ra?,lr?,[rr!]
13. ra?,lr?,rr!,ur?,lr?,rr!,lr?,ra?,rr!,ra?,ra?,[rr!]
14. ra?,lr?,rr!,ur?,lr?,rr!,lr?,ra?,rr!,ra?,ur?,[rr!]
15. ra?,lr?,rr!,ur?,lr?,rr!,lr?,ra?,rr!,ra?,lr?,[rr!]
16. ra?,ur?,rr!,ur?,rr!,ur?,lr?,ra?,ra?,lr?,lr?,ra?,[rr!]
17. ra?,ur?,rr!,ur?,rr!,ur?,lr?,ra?,ra?,ur?,lr?,ra?,lr?,[rr!]
18. ra?,ur?,rr!,ur?,rr!,ur?,lr?,ra?,ra?,ur?,ur?,lr?,lr?,ra?,[rr!]
19. ra?,ur?,rr!,ur?,rr!,ur?,lr?,ra?,ra?,ur?,lr?,ra?,ra?,ur?,[rr!]
20. ra?,ur?,rr!,ur?,rr!,ur?,lr?,ra?,ra?,ur?,lr?,ra?,ra?,lr?,[rr!]

In McMillan's DME cell, which initially does not possess the token, we found only static hazards on output signal RR:

1. ra?,ur?,rr!,rr!
2. ra?,lr?,rr!,rr!
3. ur?,rr!,lr?,ur?,rr!,rr!
4. lr?,rr!,ur?,lr?,rr!,rr!

Due to the separate verification for each output signal, counterexamples do not describe the behaviour of all output signals, which can be a drawback. A test run corresponding to the hazard no. 3 in McMillan's DME cell is given in Figure 12. Initially, all input and ouput signals are set to 0 and the DME cell does not possess the token. When the DME cell gets a user request, it immediately sends signal RR to its right neighbour. Afterwards, if LR and UR change before other signals, we get static hazard on signal RR.


MACHINE_1 - Timing Waveforms

Markers Off

Accumulate Off

Time/Div 200.0 ms

Delay 364.0 ms

Sample period = 8.000 ms

Figure 12: Hazard on signal RR in McMillan's DME cell

The results of the verification show that both designs of DME cell contain hazards even under the fundamental mode of operation. However, McMillan's design is much cleaner. Table 1 gives the number of hazards found. Note that we distinguish two hazards only if they occur in different situation in the circuit. A DME cell can first acquire the token and then deliver it forward, and afterwards it is found in the same situation as in the beginning. Moreover, static hazard in Martin's DME cell described with the sequence ur?,ra?,ua!,ra?,ua!,ua! is treated to be the same as the first one in the given list because the situation in the circuit is the same regardless of the order in which the value of input signals ur? and ra? changes.

Table 1: Hazards in DME cells

	Martin [11]			McMillan [12]		
	UA	**LA**	**RR**	**UA**	**LA**	**RR**
Static hazards	4	0	4	0	0	4
Dynamic hazards	0	0	0	0	0	0
Steady-state hazards	1	2	9	0	0	0

In the next step of the verification we checked, whether the DME circuits of different sizes satisfy safety and liveness properties proposed in [13]:

1. An acknowledgement is not given without a request.
2. An acknowledgement is not removed while a request persists.
3. All requests are eventually acknowledged.
4. No two users are acknowledged simultaneously.

After composing DME cells, only signals UR and UA from and, respectively, to different users remain in the model. They were represented with actions `ur1?`, `ua1!`, `ur2?`, `ua2!`, etc. Because in a circuit model the same action represents either the change of signal value from 0 to 1 or vice versa, the first two properties can be verified with the same ACTL formulae. On the other hand, the last and the most important property cannot be directly expressed with one ACTL formula. We can only express mutual exclusion after a user gets the acknowledgement for a given number of times. Here are the formulae used for verification of the DME circuit composed of two DME cells:

```
# After an acknowledgement is sent
# (removed), it will not be removed
# (sent) before the user requests this.

AG [ua1!] A[{NOT ua1!} UU {ur1?}]
AG [ua2!] A[{NOT ua2!} UU {ur2?}]

# All requests will be acknowledged.

AG [ur1?] AF {ua1!}
AG [ur2?] AF {ua2!}

# After a user gets the acknowledgement
# for the first time (second time etc.),
# other users will not get an
# acknowledgement until his acknowledgement
# is removed.

\define UA1 {NOT ua1!} UU {ua1!}
\define UA2 {NOT ua2!} UU {ua2!}

A[UA1 A[{NOT ua2!} UU {ua1!}]]
A[UA2 A[{NOT ua1!} UU {ua2!}]]

A[UA1 A[UA1 A[UA1 A[{NOT ua2!} UU {ua1!}]]]]
A[UA2 A[UA2 A[UA2 A[{NOT ua1!} UU {ua2!}]]]]
```

All listed formulae were valid for the circuit composed of two Martin's DME cell and also for the circuit composed of two McMillan's DME cell. Afterwards, we verified DME circuits composed of three and more DME cells, too. To do this, ACTL formulae must have been adequately adapted. Because no incorrect behaviour was detected, we may conclude that both designs of DME cell operate correctly under the fundamental mode of operation. Evidently, with the presented approach, we were not able to detect malfunction of DME circuits composed of Martin's cells because it is the result of delay hazards.

The size of circuit models used during the verification is given in Table 2. The most complex task during the verification was the composition of processes. Table 3 and Table 4 give some statistics about the complexity of parallel composition obtained on a system composed of 800 MHz Athlon processor, 512 MB RAM and Linux OS. The size of DME cells in Table 2 refers to the circuit model describing external behaviour of DME cells which initially do not possess the token. The sizes reported in the last two tables refer to the circuit model describing external behaviour of rings where internal behaviour of DME cells is abstracted away. We were not able to compose more than 4 Martin's and 5 McMillan's DME cells although the resulting process is supposed not to be enormous. Hence, an "on the fly" model checker would be of great interest here.

Table 2: The size of circuit models

Circuit	**Inputs / Outputs**	**States / Transitions**	**BDD nodes**
C element	2/1	4/7	32
RS flip-flop	2/2	11/16	75
ME element [11]	3/2	24/51	164
ME element [12]	2/2	11/16	73
DME cell [11]	3/3	72/148	474
DME cell [12]	3/3	48/104	304

Table 3: Parallel composition of Martin's DME cells

Circuit	**States / Transitions**	**BDD nodes**	**Time for composition**
2 DME cells	11/16	62	0.1s
3 DME cells	57/117	365	2.4s
4 DME cells	236/632	1752	100.2s

Table 4: Parallel composition of McMillan's DME cells

Circuit	**States / Transitions**	**BDD nodes**	**Time for composition**
2 DME cells	11/16	66	0.1s
3 DME cells	40/72	251	0.8s
4 DME cells	145/316	974	12.9s
5 DME cells	596/1545	4444	1299.1s

7 Conclusion

Asychronous circuits are an important class of digital circuits used as autonomous devices or just a part of otherwise synchronous circuits. They have many nice properties, but they are also difficult to design and verify, especially in an ad hoc fashion. Emerging tools for formal verification promise a solution for many problems in this area.

This paper introduces a method for formal verification of asychronous circuits modelled with Muller's model under the fundamental mode of operation. It is suitable for detecting hazards in the given circuit and for verification of safety and liveness properties. The method is based on the representation of external behaviour of the circuit with processes. A special form of processes called circuit models is used to describe gates. A parallel composition with the ability of multi-way synchronisation and an operation called fundamental-mode reduction serve for construction of circuits from single gates. Determinization of processes assures a canonical form of specifications. The properties of circuits can be checked by ACTL model checking. The results obtained by verification and measurements on real circuits convince us about the correctness of our approach.

We used presented method for the verification of DME circuits. We were able to verify the external behaviour of DME circuits composed of up to 5 DME cells. For larger circuits, BDD-based tool EST exceeded memory limits of 512 MB during the parallel composition of circuit models. Maybe, the impact of state explosion could be moderated by using "on the fly" model checking. Alternatively, parallel composition and fundamental-mode reduction could be united in a more efficient operation.

There are many topics for further research. The ability of model checking to find tree-like counterexamples was mentioned in the paper. We are also interested in experiments with *input-quasi-receptive* circuit models, which describe the behaviour of the circuit under input/output mode of operation. They enable checking semi-modularity. Input-quasi-receptive models accept input signals in all states and this leads to more complex specifications. It is a challenge how to apply such an approach to larger asynchronous circuits.

References

[1] ACiD-WG. Report on Design, Automation and Test for Asynchronous Circuits and Systems, January 2002.

[2] A. Bailey. Automatic Verification of Speed-Independent Circuit Designs Using the Circal System. In *Correct Hardware Design and Verification Methods (CHARME '93)*, volume 683 of *LNCS*, pages 167–178. Springer-Verlag, May 1993.

[3] S. M. Burns. Automated Compilation of Concurrent Programs into Self-timed Circuits. Master's thesis, California Institute of Technology, 1988.

[4] F.-C. Cheng. Exact Essential-Hazard-Free State Minimization of Incompletely Specified Asynchronous Sequential Machines. Technical report CUCS-033-94.

[5] A. Davis and S. M. Nowick. An Introduction to Asynchronous Circuit Design. Technical Report UUCS-97-013, University of Utah, September 1997.

[6] A. Fantechi, S. Gnesi, F. Mazzanti, R. Pugliese, and E. Tronci. A Symbolic Model Checker for ACTL. In *Proceedings of FM-Trends'98*, volume 1641 of *LNCS*, pages 228–242. Springer-Verlag, October 1998.

[7] S. Hauck. Asynchronous Design Methodologies: An Overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.

[8] J. He. Formal Specification and Analysis of Digital Hardware Circuits in LOTOS, August 2000. Technical Report CSM-158. University of Stirling.

[9] Y. Lu. *Automatic Abstraction in Model Checking*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 2000.

[10] A. R. Martello. *Temporal Analysis for Time-Bounded Causal Digital Systems*. PhD thesis, University of Pittsburgh, April 1993.

[11] A. J. Martin. The Design of a Self-timed Circuit for Distributed Mutual Exclusion. In *Proceedings of the 1985 Chapel Hill Conference on VLSI*, pages 245–260, 1985.

[12] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, May 1992. Technical report CMU-CS-92-131.

[13] K. L. McMillan. The SMV system, November 2000. http://www-2.cs.cmu.edu/∼modelcheck/smv.html.

[14] R. Meolic. Checking correctness of concurrent systems behaviour. Master's thesis, University of Maribor, 1999. In Slovene.

[15] R. Meolic, T. Kapus, and Z. Brezočnik. The Efficient Symbolic Tools Package. In *Proceedings of the Soft-COM 2000*, volume I, pages 147–156, Split, Croatia, October 2000. http://www.el.feri.uni-mb.si/est/.

[16] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[17] C. Myers. *Asynchronous Circuit Design*. John Wiley & Sons, 2001.

[18] O. Roig, J. Cortadella, and E. Pastor. Conservative Symbolic Model-Checking of Petri Nets for Speed-independent Circuit Verification, 1994. DAC/UPC Technical Report No. RR-94.

[19] M. Shams, J. C. Ebergen, and M. I. Elmasry. Asynchronous Circuits. John Wiley's Encyclopedia of Electrical Engineering.

[20] K. S. Stevens. *Practical Verification and Synthesis of Low Latency Asynchronous Systems*. PhD thesis, Dept. of Computer Science, University of Calgary, Canada, September 1994.