Journal of ELECTRICAL ENGINEERING, VOL. 59, NO. 1, 2008, 14–22

TEMPORAL LOGIC IN VERIFICATION OF DIGITAL CIRCUITS

Daniela Kotmanová [*]

Verification of hardware circuits is a fundamental part of the hardware design. The paper presents temporal logic as a very powerful verification tool. Temporal logic is a classical logic perceived as a function of time. Now, the truth values of logical propositions change as the time flows on. We fetch up in a space of logical assertions in which, we emphasize, the time is flowing, that is, it is a variable, not a constant. Temporal logics, in principle, are linear or branching where linear means linear-time temporal logic (LTL) and branching branching-time temporal logic (CTL). Both can be used in SMV (Symbolic Model Verifier), a model checking-based verification system for digital devices. In this paper, we describe the results we obtained in our verification of a design of the binary comparator.

K e y w o r d s: temporal logic, automatic verification, model checking, digital design, Symbolic Model Verifier (SMV)

1 INTRODUCTION

Finite state reactive systems, including digital circuits, are an important constituent of computer science. As soon as the question is about the correctness of the design many problems arise. The most widely used verification techniques up to now are based on extensive simulation, possibly testing. They cannot be exhaustive, notably when the number of possible states of the digital system is very large. Thus, logical errors in the design part may emerge unexpectedly and menace all the correct device performance, and even may be dangerous, in view of the unpredictability of the behaviour of such a device.

In this paper we give some notion of temporal logics and how they can be used in digital design, specifications and mainly in verification processes.

The principal method for verifying these systems is model checking. Broadly speaking, temporal logic model checking is, generally, an automatic algorithmic procedure for verifying finite-state reactive systems, and in particular, digital systems. We show in our paper how to apply this automatic verification method, based on temporal logics, to a digital system design, such as, for instance, a binary comparator.

We demonstrate how to describe the behaviour of a digital circuit using temporal logic and how to establish temporal specifications of the circuit design. As illustration, we introduce in detail the design of the binary comparator with two inputs, and describe its behaviour in temporal specifications, using linear-time temporal logic.

The design, specification and verification of the comparator are an application of our own. We also present an automatic verifying tool — a model checker SMV (Symbolic Model Verifier) — and give an example of the SMV verification program, written by us in the SMV Language. Our verification program is, of course, relative to the digital design of the comparator we made.

In the end of the paper, the results of the verification we arrived at are presented.

2 TEMPORAL LOGIC, FORMAL VERIFICATION METHODS, MODEL CHECKING

2.1 Temporal logic

Temporal logic is a logic where necessarily the time plays a role. In classical logic, such as propositional or predicate 1-st order logic, the notion of truth is independent of the time. Truth values of formulas remain the same regardless the time flowing. It is not the case for the temporal logics. A formula is not statically true, but it may change its truth value dynamically, with the time running. Thus, formulas applied to states of a system which evolves from state to state in the time swap their truth values from 0 to 1 and vice versa, depending on current state the system is in.

In a concrete state transition diagram with subsets of atomic propositions assigned to each state, this alternation of truth values of a proposition is perceptible as either a presence of the proposition in a given state, or its absence in the state.

There are two fundamental temporal logics that differ by their view of time. Linear-time temporal logic (LTL) considers time as a linear and single succession of time points (or time- instances) while branching-time temporal logics (*eg* CTL) allow several alternative future paths at any given point of time. The path in branching-time temporal logics is perceived as a trajectory in the unwound state transition diagram in form of tree (thereof Computation Tree Logic, CTL). It consists of a set of reachable states of the system. Branching-time temporal logic is also used in modelling non-deterministic systems, computations included. The path in linear-time temporal logic is conceived as a trajectory consisting of a linear

[*] Institute of Computer Systems and Networks, Faculty of Informatics, Slovak University of Technology, Ilkovičova 3, 812 19 Bratislava; kotmanova@fiit.stuba.sk

ISSN 1335-3632 © 2008 FEI STU

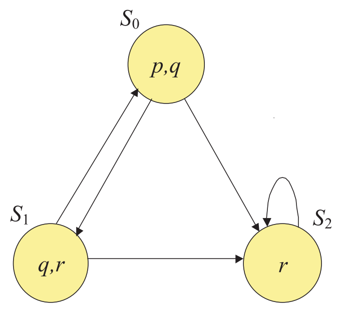


Fig. 1. An oriented state graph with the initial state s_0. Each of states (s_0, s_1, s_2) contains a set of atomic propositions $\{p,q\}$, $\{q,r\}$, $\{r\}$ respectively).

succession of reachable states. We will see it later more in details.

Linear-time temporal logic LTL

A dynamic system which evolves in time necessarily moves from one state to an another. Each state of the system model corresponds to a certain point of time. Thus, each linear sequence of time points is equivalent to a linear sequence of states of the system. That is, the sequence also constitutes a path in the model. Applying linear-time temporal logic we will always have only an unique path in temporal dynamics of the system.

- **Syntax of the language used**

Linear time temporal logic has the following syntax in Backus Naur form [3]:

$$\Phi := p \,|\, \neg\,\Phi \,|\, \Phi \wedge \Phi \,|\, X\Phi \,|\, F\Phi \,|\, G\Phi \,|\, \Phi U \Phi\,,$$

p is any atomic proposition.

- **Semantic of the language used, Model for LTL, Kripke structure, Satisfaction relation**

Linear time temporal logic model LTL or CTL

We take as model $\mathcal{M}$ for linear-time temporal logic a labelled state transition diagram characterized by the triplet (S,σ,L), where

1. S is a finite set of states, $S = \{s_1, s_2, s_3, s_4, \dots\}$
2. σ is a binary relation on the set S (its graph $\mathcal{G}_\sigma$ satisfies the assertion $\mathcal{G}_\sigma \subseteq S \times S$):

$$\sigma\colon S \to S\,, \quad s \mapsto s' = \sigma(s)\,.$$

For each $s \in S$ there is an $s' \in S$:

$$\forall s \in S \quad \exists s' \in S \quad [(s,s') \in \mathcal{G}_\sigma]\,,$$

s' is a successor of the state s in S.

3. L is a labelling function:

$$L\colon S \to \mathcal{P}(\text{Atoms})\,, \quad s \mapsto L(s)\,,\ L(s) \in \mathcal{P}(\text{Atoms})$$

where:

Atoms $= \{p, q, \dots\}$ are the set of all atomic propositions that may be true in single states of the system,

$\mathcal{P}(\text{Atoms}) = \{emptyset, \text{Atoms}, \{p\}, \{q\}, \{p,q\}, \dots\}$ is the power set of the set Atoms

$S = \{s_1, s_2, s_3, \dots\}$ represents the set of all possible states of the system modelled.

σ is a transition relation, it determines potential transitions between states, *ie* it says us how the system evolves ahead in time.

L assigns to each state s a given set $L(s)$ which is different for every particular state and attached to the state it is valid in. $L(s)$ is thus a set of atomic propositions true in the particular given state s.

R e m a r k . The model for LTL is identical with the model used for CTL, see in [3].

Kripke structure

The model $\mathcal{M}$, represented by the triplet (S,σ,L), is called a Kripke structure, more in [3]. A Kripke structure is thus an oriented labelled state-transition diagram, with a set of true atomic propositions assigned to (*ie* present in) states, as illustrated in Fig. 1. The nodes of the diagram represent global states of the system and they contain propositional atoms valid in each respective particular state. The edges represent atomic global state transitions, arrows showing directions of a transition. In Fig. 1, such a Kripke structure is drawn, with sets of logical propositions true in particular global states:

$$L(s_0) = \{p,q\}\,,\ L(s_1) = \{q,r\}\,,\ L(s_2) = \{r\}\,.$$

Path

A path π is a infinite sequence of states $(s_1, s_2, s_3, \dots, s_i, \dots)$ such that

$$\forall i \in N \qquad [(s_i, s_{i+1}) \in \mathcal{G}_\sigma]$$

or a finite sequence of states $(s_1, s_2, s_3, \dots, s_n)$ such that

$$\forall i \in \{1, 2, \dots, n-1\} \qquad [(s_i, s_{i+1}) \in \mathcal{G}_\sigma]\,.$$

The notation $s_0 \to s_1 \to s_2 \to s_3 \to \dots$ is also used to emphasize the state transition character of a path, as a sequence of successors states. The notation π^i, $i \in N_0$, is used for path sections such that $s_i \to s_{i+1} \to s_{i+2} \to s_{i+3} \to \dots$.

Satisfaction relation

Let $\mathcal{M} = (S,\sigma,L)$ be a model for LTL.
For a given path $\pi\colon s_0 \to s_1 \to s_2 \to \dots \to s_i \dots \to$, $\forall i \in N_0$, and an LTL formula Φ we define the satisfaction relation $\models$ valid in the model $\mathcal{M}$ and along a path π beginning in state s_0 of the model:

$$\mathcal{M}\,,\ \pi \models \Phi\,.$$

Taking into consideration the notation adopted above and supposing Φ is an atomic proposition we can put:

$$\Phi \in L(s_0)\,.$$

Examples

An unary linear temporal formula $\Phi = Z\varphi$ constructed with an arbitrary unary linear-time temporal operator denoted generally Z, representing arbitrarily one of linear operators X, F, G; and with a proposition φ, is said valid along a path $\pi\colon s_0 \to s_1 \to s_2 \to s_3 \to$ in a model $\mathcal{M}$ iff φ is valid on corresponding path sections π^i, $i \in N_0$, according to the character of the concrete temporal operator used:

$$\mathcal{M}, \pi \models Z\varphi \iff \mathcal{M}, \pi^i \models \varphi\,, \quad i \in N_0$$

what means

$$CalM, \pi \models X\varphi \iff \models \varphi\,,$$
$$\mathcal{M}, \pi \models F\varphi \iff \exists i \in N, i \geq 1 \quad \mathcal{M}, \pi^i \models \varphi\,,$$
$$\mathcal{M}, \pi \models G\varphi \iff \forall i \in N, i \geq 1 \quad \mathcal{M}, \pi^i \models \varphi\,.$$

A *binary linear temporal formula* $\Phi = \varphi U \psi$ constructed with the binary linear-time temporal operator U, and with propositions φ, ψ, is said *valid* along a path $\pi\colon s_0 \to s_1 \to s_2 \to s_3 \to \dots$ in a model $\mathcal{M}$ iff φ, ψ are valid on corresponding path sections π^i, π^j, according to the equation below:

$$\mathcal{M}, \pi \models \varphi U \psi \iff$$
$$(\exists j \in N\ \pi^j \models \psi\ \wedge\ \forall i \in N,\ 1\ \leq i \leq j-1\ \ \mathcal{M},\ \pi^i \models \varphi)$$

The notation adopted has to be understood in the following meaning:

$\pi\colon s_0 \to s_1 \to s_2 \to \dots \to s_i \to \dots$

path beginning in state s_0 (initial state)

$\pi^i\colon s_i \to s_{i+1} \to s_{i+2} \to s_{i+3} \to \dots$

path beginning in state s_i (path section)

Resumé

An atomic formula φ is true along any generalized path $\pi^i\colon s_i \to s_{i+1} \to s_{i+2} \to s_{i+3} \to \dots$, $i \in N_0$, if it is true in the given initial state s_i of the path π^i. The following equivalence is obviously true:

$$\mathcal{M}\,,\ \pi^i \models \varphi \iff \mathcal{M}\,,\ s_i \models \varphi \ \ \textit{ie}\ \ \varphi \in L(s_i)\,,\ i \in N_0\,.$$

For $i = 0$, $\pi^i = \pi$ and $s_i = s_0$ (initial state). $L(s_i)$ remaining a set of formulas true in the state s_i.

An LTL formula Φ is satisfied in a *state* s of the model $\mathcal{M}$ iff Φ is satisfied along every path $\pi\colon s_0 \to s_1 \to s_2 \to s_3 \to \dots$ starting at the state s_0 of the model $\mathcal{M}$.

2.2 Verification and Checking

Formal Verification Methods

In the sphere of verification of reactive systems, generally two principal formal verification methods have been developed:

• *deduction method*

which is applied to concurrent reactive systems. The method is proof-based. System description is made in some formal language and leads to a set Γ of appropriate formulas in an appropriate logic. The set Γ constitutes a formal logical inference system for deduction.

A system specification is an another formula φ of a chosen logic. The verification consists of finding a proof within the given formal logical system, which would demonstrate that the specification formula φ is inferred from the axioms and inference rules of the formal system Γ, *ie* $\Gamma \models \varphi$. The formal system Γ is assumed to be sound and complete.

This method is for the time being laborious and only slightly computer assisted. Nevertheless, it is much more general than the model checking method — it works also for infinite state systems. Another advantage is that it is a pre-development method. Such this, it is used during the system development, not after.

• *model checking*

which is, as its name predicts, model-based. The system is represented by a finite model $\mathcal{M}$ for an appropriate logic and specifications are formulas of this logic. The model $\mathcal{M}$ comprises a set of states, a transition relation and a labelling function, as shown yet. The verification consists then of checking whether the model $\mathcal{M}$ satisfies the specification φ, *ie* $\mathcal{M} \models \varphi$.

This method is fully automatic, intended, however, only for finite systems. Anyway, another inconvenient is that it is a post-development method, used after the system has been developed.

Model Checking Verification Tools

As mentioned, model checking is an algorithmic automatic method for verifying finite-state reactive systems, among them digital circuits designs, and concurrent systems, such as communication protocols and parallel processes as well. It is based on a kind of temporal logic, see for more details in [2], [3], [5], [6].

The system being modelled as a state transition diagram with global states as nodes and transition relations as edges, its states are labelled each with an appropriate set of propositions.

Thus, every state is assigned its own set of atomic propositions expressed in classical Propositional Logic.

Temporal formulas, Specifications

Temporal operators, linear or branching-time, always applied to *propositions in particular states*, allow to define *temporal formulas* over a path or sequences of states. Validity of temporal formulas over a path (LTL formulas) or in an initial state of a path (CTL formulas) is obviously determined by the character of the temporal operator used and, in last consequence, by the validity or non-validity (presence or absence) of the atomic propositions in relevant particular states.

Temporal formulas capture dynamic properties of the system and as such form *specifications* to be checked.

Once the system has been modelled and temporal specifications established, the model is described and transferred into the computer verification system by means of some special language which includes also temporal specifications. We say more about it in Chapter 3.

The verification based on model checking consists then precisely in finding out whether or not the chosen model satisfies the specifications. The verification tools based on model checking are called *model checkers* and they produce an answer at the end of the process of checking. This answer is in fact the final goal of any practical usage of model checking. If the model satisfies the given specification the model checker generates the answer "true"; if not, the answer "false" and an error tracing is simultaneously produced.

3 SYMBOLIC MODEL VERIFIER (SMV)

Symbolic Model Verifier is a verification system especially for digital circuit designs. SMV is a formal verification tool, *ie* it verifies all the possible behaviours that a given system can have and that satisfy required specifications. It was created by McMillan in 90's [5], [6], [7].

We restate that a specification is, in general, a collection of properties that a system must observe. Properties of the system, or temporal formulas, are formulated in some logics the model checker would be able to recognize. Symbolic Model Verifier has been constructed for using both, the linear temporal logic as well as the branching temporal logic.

The SMV system provides a special language, called *SMV Language*. The language incorporates a set of special tools for describing models of the system and translating temporal specifications about desired properties to be checked (temporal formulas).

SMV accepts as input a file consisting of a program. The verification program for the SMV model checker, written in the SMV Language, contains the SMV description of the system model: states, inputs and outputs, and specifications established to capture temporal properties of the system. The program may contain one or some more modules. As in other programming languages one of the modules is fundamental and called *main*. In modules, there are declarations of variables and assignments to the variables, either with initial and next values or with a non-deterministic expressions in braces, containing some elements to be random chosen among them and assigned. Such aspect of the SMV language — as the non-determinism — allows especially to model the influence of the environment.

Formal verification of the system design with the SMV Model Checker

3.1 Input file to the model checker

We can summarize the necessary steps in verification of a digital circuit design, carried out by the SMV model checker:

1. Model the digital system, usually by creating a finite state-transition diagram, with defined states and state-transitions (FSM); afterwards, an appropriate semantic model $\mathcal{M}$ for LTL or CTL logic, with its appropriate states called global, and its state transitions; and with propositions valid in the global states (state space model; Kripke structure).
2. Describe, using the description language of the model checker, the model drawn. This description made in SMV Language will tell the computer what system is going on.
3. Express the dynamic properties Φ of the system — specifications — in some language of temporal logic and incorporate them into the SMV input file using the specification language of the model checker SMV.

Dynamic properties of the system — its behaviour in the time — are captured, we underline, by temporal operators applied to the propositions valid in various states, creating in this way temporal formulas. Temporal formulas, applied along a path or in the initial states, will indeed have a meaning of dynamic properties Φ, as it will be shown below. Having incorporated formulas Φ of temporal logic to the input file the model checker will code them to make them intelligible to the computer.

Essential parts of an input file for the model checker SMV are:

- Keyword main with formal parameters and declarations of the variables
- Description, in the SMV language, of the system modelled as a state transition diagram first, then as a global state transition diagram $\mathcal{M}$ (state space model)
- Temporal properties Φ of the system to verify, formulated as specifications in the model $\mathcal{M}$

This way, to verify a model with the model checker SMV amounts to run the model checker with the model $\mathcal{M}$ and temporal specifications Φ included in the input file, launching the SMV command *Verify all*.

3.2 Output

The output of the model checker, as said, is done by the word "*true*" or "*false*", shown on the display, according to the results found by the model checker — "*true*" notifying that a desired property of the system is verified in the model $\mathcal{M}$ (*ie* in a path including corresponding states the model can reach, according to temporal operator used) while "*false*" signifying the opposite. What is worth attention is that usually, the model checker produces a counter-example to demonstrate why and where the property becomes false. The counter example consists of tracing the path in the model leading to the place where the property would fail. The counter-example helps

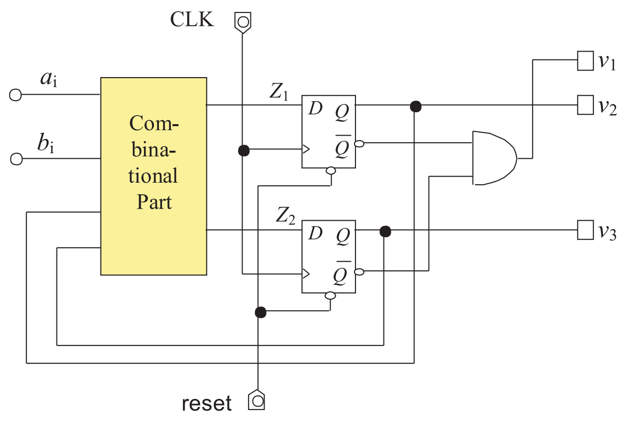


Fig. 2. Binary Comparator — a hardware circuit.

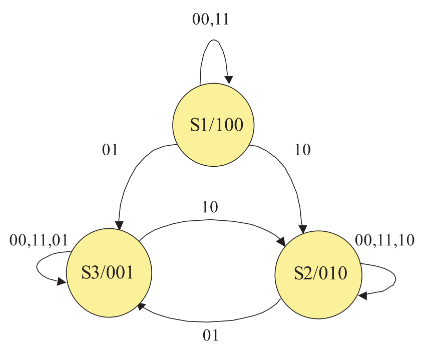


Fig. 3. The Moore's automaton

us to understand why the formula does not hold, giving the path and the state where the failure happens.

The SMV works on Binary Decision Diagrams (BDDs.), see [1], [3] and [5]–[7].

4 DIGITAL DESIGN: A BINARY COMPARATOR

4.1 Binary Comparator with two inputs — hardware circuit

The binary comparator we chose to demonstrate temporal properties and their verification is a simple hardware circuit to compare two binary numbers. The comparator, with two inputs, carries out a strict comparison between two n-place binary numbers arriving as serial strings of binary signals on the inputs, starting with the bits of low order. In fact, this results in realizing a strict ordering relation between the two n-bit numbers, determining their mutual relation, *ie* in a distinction among the cases of either equality of the two numbers, either superiority or inferiority of the one number over the another. The hardware realization of the circuit consists of a combinational part and some memory elements. In Fig. 2, two clocked D Flip-Flops are used. There are two inputs a_i, b_i for two binary signals, and three outputs y_1, y_2, y_3, each with a particular significance to characterize the mutual size relation of the numbers on the input. Two binary signals arriving at the input necessarily cause one – and only one – of the three device outputs to light up, the first of them being reserved for the equality of the two numbers, the second for the superiority and the third for the inferiority, of the first number over the second.

4.2 Modelling Binary Comparator as a finite state machine

To draw a state-transition diagram we have to determine what states and transitions inducing the change of behaviour the device could produce. Sometimes, it is not easy to define. What is remembered in the case of a comparator as described above is, of course, the comparison. Thus, to model the comparator as a finite state machine (FSM) we have to pick up three different states in its behaviour. They are defined by the mutual quantitative relation of the two numbers permanently compared at given points. As a FSM we use Moore's automaton.

Defining States

- State $S1$ – "equality" (the two numbers are equal)
- State $S2$ – "superiority" (the first number is greater than the second)
- State $S3$ – "inferiority" (the first number is less than the second)

The transition between two different states is taken whenever signals arriving on inputs are in corresponding relations between them mutually and between them and the previous ordering, as shown on the Moore's automaton in Fig. 3. If the comparison's result does not change, *ie* the ordering of the two numbers is preserved the system remains in the same state and the transition function keeps its previous value, *ie* it is reflexive.

With each state of the Moore's automaton a triple output is joined to differentiate between states. Only one of three outputs will be active at a time and this way, characterize just exactly one of three different states the system could currently be in signalizing them uniquely, by a LED for example.

We can derive a state-transition-and output table from the state-transition diagram (Tables 2 and 3).

Table 1. State-Transition Table and Output Table

Current state	Next state				Output		
st	a_i b_i				y_1	y_2	y_3
$S1$	$S1$	$S2$	$S1$	$S3$	1	0	0
$S2$	$S2$	$S2$	$S2$	$S3$	0	1	0
$S3$	$S3$	$S2$	$S3$	$S3$	0	0	1

trans(st, a_i, b_i) $o(st)$

trans(st, a_i, b_i) – transition function with the variable st ("Current state") and the input variables a_i, b_i $o(st)$ – output function with the variable st (depending only on the current state)

Coding states

state $S1$	00	*ie*	$z_1 = 0$	$z_2 = 0$
state $S2$	01	*ie*	$z_1 = 1$	$z_2 = 0$
state $S3$	10	*ie*	$z_1 = 0$	$z_2 = 1$

Table 2. Coding map

		z_2
	$S1$	$S3$
z_1	$S2$	x

Coded Transition- and Output functions

Transition function and Output function coded with the code defined above (Karnaugh map)

Table 3. Coded Transition Function (Forcing functions Z_1, Z_2)

		a_i	a_i, b_i	b_i
	00	10	00	10
z_2	01	10	01	01
z_2, z_1	xx	xx	xx	xx
z_1	01	10	01	01

Z_1, Z_2

Table 4. Coded Output Function (Coded outputs y_1 , y_2 , y_3)

		z_2
	100	001
z_1	010	xxx

y_1, y_2, y_3

Maximal regular configurations of 1's in Karnaugh maps Tabs. 3, 4 lead to the following coded transition and output functions :

$$Z_1 = a_i\overline{b}_i + a_i z_i + \overline{b}_i z_i$$
$$Z_2 = \overline{a}_i b_i + b_i z_2 + \overline{a}_i z_2$$
$$y_1 = \overline{z}_1\overline{z}_2$$
$$y_2 = z_1$$
$$y_3 = z_2$$

The variables z_1 , z_2 act as state variables and the variables Z_1 , Z_2 as forcing functions.

5 VERIFICATION OF A DIGITAL CIRCUIT DESIGN

5.1 Behaviour of a Binary Comparator

To determine the behaviour of a digital system we have to take into account what task the device is assumed to realize. The binary comparator must accomplish the operation of comparing two binary numbers, *ie* realize a relation of strict ordering of the two numbers. We use the terms "*equality*", "*superiority*" and "*inferiority*" introduced above, to catch the different possibilities of an act of strict comparing. They determine various potential results of the operation.

We now enumerate all the possibilities that can arise to demonstrate how temporal properties of the comparator have been established in the program, in order to be verified.

For that reason we introduce the words "*ordering*" and "*new ordering*" to characterize the current state and the succeeding state of the comparison of two binary strings A, B.

We write the appropriate relations :

If *ordering* is $A = B$ and arriving signals a, b are such that $a = b$, then *new ordering* will be $A' = B'$.

If *ordering* is $A = B$ and arriving signals a, b are such that $a > b$, then *new ordering* will be $A' > B'$.

If *ordering* is $A = B$ and arriving signals a, b are such that $a < b$, then *new ordering* will be $A' < B'$.

If *ordering* is $A > B$ and arriving signals a, b are such that $a = b$, then *new ordering* will be $A' > B'$.

If *ordering* is $A > B$ and arriving signals a, b are such that $a > b$, then *new ordering* will be $A' > B'$.

If *ordering* is $A > B$ and arriving signals a, b are such that $a < b$, then *new ordering* will be $A' < B'$.
If *ordering* is $A < B$ and arriving signals a, b are such that $a = b$, then *new ordering* will be $A' < B'$.

If *ordering* is $A < B$ and arriving signals a, b are such that $a > b$, then *new ordering* will be $A' > B'$.

If *ordering* is $A < B$ and arriving signals a, b are such that $a < b$, then *new ordering* will be $A' < B'$.

Now, we transform these conditional propositions from the usual language to the mathematical notation and will deduce the logical rapports existing between assertions to establish correct temporal specifications.

We can now be sure they will cover all possible behaviours of the modelled system since we have viewed all possibilities that could arise.

Mathematical notation

Let have two binary numbers A and B, of length n. Each of them consists of a set of n bits, represented by $a_{n-1}, a_{n-2}, \dots, a_0$ and $b_{n-1}, b_{n-2}, \dots, b_0$ respectively, starting with the lower orders. Thus, we have

$$A = A_n = a_{n-1}a_{n-2}\dots a_0$$
$$B = B_n = b_{n-1}b_{n-2}\dots b_0$$

In a given moment, on the device inputs we have had a couple of partial strings Aj and B_j , of length j , $j \le n$, such that

$$A_j = a_{j-1}a_{j-2}\dots a_0\,,$$
$$B_j = b_{j-1}b_{j-2}\dots b_0\,.$$

In the next moment, we will have had, on the device inputs, a couple of partial strings A_{j+1} and B_{j+1} , of length $j+1$, $j \le n$, such that

$$A_{j+1} = a_j a_{j-1}\dots a_0\,,$$
$$B_{j+1} = b_j b_{j-1}\dots b_0\,.$$

The mathematical description of the device performance is the following:

$$A_j = B_j \wedge a_j = b_j \rightarrow A_{j+1} = B_{j+1}$$
$$A_j = B_j \wedge a_j > b_j \rightarrow A_{j+1} > B_{j+1}$$
$$A_j = B_j \wedge a_j < b_j \rightarrow A_{j+1} < B_{j+1}$$

$$A_j > B_j \wedge a_j = b_j \rightarrow A_{j+1} > B_{j+1}$$
$$A_j > B_j \wedge a_j > b_j \rightarrow A_{j+1} > B_{j+1}$$
$$A_j > B_j \wedge a_j < b_j \rightarrow A_{j+1} < B_{j+1}$$

$$A_j < B_j \wedge a_j = b_j \rightarrow A_{j+1} < B_{j+1}$$
$$A_j < B_j \wedge a_j > b_j \rightarrow A_{j+1} > B_{j+1}$$
$$A_j < B_j \wedge a_j < b_j \rightarrow A_{j+1} < B_{j+1}$$

5.2 Temporal specifications

Behaviour of the model - a set of temporal properties

To describe the behaviour of the system and capture its properties we use temporal operators X and G, applied to some proposition p, Xp being valid over a path π if p valid over π^2 (in the second state of the path π) and Gp being valid over a path π if p valid in all underlying paths π^i, ergo in all states of the system ("*globally*"). As proposition p we can have various atomic propositions, *eg* $\mathit{state} = S1$, or $a > b$, *etc*, always bound up with a state. If the system is really in state $S1$, naturally the value of p: $\mathit{state} = S1$ is true, if not, false. Applying temporal operator to a proposition we evaluate the truth value of the temporal formula not in the state the proposition is associated with but over a path (or in initial states) given by the operator.

To affirm the assertions written above, assumed to be true for all the time the system is running, are indeed true during this time we put before every assertion the operator G.

From the equations above we draw following temporal properties, these are precisely the properties which will be verified by means of the model checker SMV:

$$G(a = b \wedge \mathit{state} = S1 \rightarrow X\mathit{state} = S1);$$
$$G(a = b \wedge \mathit{state} = S2 \rightarrow X\mathit{state} = S2);$$
$$G(a = b \wedge \mathit{state} = S3 \rightarrow X\mathit{state} = S3);$$
$$G(a \neq b \rightarrow (X\mathit{state} = S2 \wedge X\mathit{state} = S3));$$
$$G(a > b \rightarrow X\mathit{state} = S2);$$
$$G(a < b \rightarrow X\mathit{state} = S3)\,.$$

Temporal assertions about the output are expressed by the equivalences required of the design, using the output functions y_1, y_2, y_3:

$$G(\mathit{state} = S1 \leftrightarrow y1 \wedge \neg y2 \wedge \neg y3)\,,$$
$$G(\mathit{state} = S2 \leftrightarrow y2 \wedge \neg y1 \wedge \neg y3\,,)$$
$$G(\mathit{state} = S3 \leftrightarrow y3 \wedge \neg y1 \wedge \neg y2)\,.$$

5.3 Verification program in the SMV Language

The SMV program takes form of standard programs:

```
module main(a,b,y1,y2,y3)
  {
      input a,b: boolean;
      output y1,y2,y3: boolean;

/*
  a,b     - independent input variables (boolean)
  y1,y2,y3- dependent output variables; outputs y1,y2,y3
            take on the values 0 and 1 (boolean);
  state   - variable ranging over a set of state values,
            ie the set {S1, S2, S3}
*/
  state : S1,S2,S3;
  init (state) := S1;
  next (state) :=
    switch(state, a, b) {

    (S1,    0,0) : S1;
    (S1,    0,1) : S3;
    (S1,    1,0) : S2;
    (S1,    1,1) : S1;
    (S2,    0,0) : S2;
    (S2,    0,1) : S3;
    (S2,    1,0) : S2;
    (S2,    1,1) : S2;
    (S3,    0,0) : S3;
    (S3,    0,1) : S3;
    (S3,    1,0) : S2;
    (S3,    1,1) : S3;
  };
        y1 := (state = S1);
        y2 := (state = S2);
        y3 := (state = S3);
/*
  Verification of temporal properties
Verification of temporal properties relative to states
*/
    temp_prop1_1 : assert G(a=b state=S1 -> X state= S1);
    temp_prop1_2 : assert G(a=b state=S2 -> X state= S2);
    temp_prop1_3 : assert G(a=b state=S3 -> X state= S3);

    temp_prop2 : assert G(a~=b -> G(X state=S2^X state=S3));
    temp_prop3 : assert G(a>b -> X state=S2);
    temp_prop4 : assert G(a<b -> X state=S3);
/*
Supplement - equivalence of temporal propositions
  temp_prop2 and temp_prop2a
*/
    temp_prop2a: assert G(a~ =b -> G X(state=S2^state=
  S3));
    temp_prop2b: assert G(a~=b -> G(X state=S2^X
  state=S3)) <-> G(a~=b -> G X(state=S2^state=S3));
    temp_prop2c: assert G(X state=S2^X state=S3
  <-> X (state=S2^state=S3));
    temp_prop2d: assert G(X state=S2 | X state=S3
  <-> X (state=S2 | state=S3));
/*
We can obviously conclude, from the properties temp_prop2
and temp_prop2a, and using temp_prop2b, that there is an
equivalence between the two assertions, and finally, claim
the distributivity of the operator X over the logical
connective +^(EXOR) (temp_prop2c)

   Remark1
   In this specific case, we may also use | (logical con-
nective OR) instead of ^ (logical connective EXOR) but ^
is more precise; in this case connectives | and ^ are
equivalent just because of the practical impossibility of
a simultaneous realization of truth values ''true'' of the
two assertions, state=S2 and state=S3 (temp_prop2d)
   Remark2
   It is even possible to put the assertions temp_prop2,
temp_prop3 and temp_prop4 together in a single proposition
temp_prop5, as follows:
*/
    temp_prop5: assert G(a~=b -> G X((a>b ->
  X state=S2)|(a<b -> X state=S3)));

/*
Verification of temporal properties relative to outputs
/*
    temp_prop6: assert G(state=S1 <-> y1&~y2&~y3);

/*
N.B. However, the proposition
    temp_prop6a : assert G state=S1 <->y1&~y2&~y3;
is false ; already the proposition
    temp_prop6b : assert G(y1&~y2&~y3 -> state=S1);
is true
*/

    temp_prop6b : assert G(y1&~y2&~y3 -> state=S1);

    temp_prop7a : assert G state=S2 <-> y2&~y1&~y3;
    temp_prop7b : assert G(state=S2 <-> y2&~y1&~y3);

    temp_prop8a : assert G state=S3 <-> y3&~y1&~y2;
    temp_prop8b : assert G(state=S3 <-> y3&~y1&~y2);
}
```

Some remarks.

1. We emphasize the program is written for linear temporal logic. The program for branching time temporal logic would have another structure since the syntax of SMV Language for CTL logic is different.
2. As showed above, there are used two sorts of temporal operators in the program: Operator G means *Globally* — the logical assertion the temporal operator G is applied to is true for all states that the system can reach. Operator X meaning *Next* is fundamental in modelling and specifying state machines: Basically, it concerns the successor state after a transition between two states (different or identical) has been made.
3. Validity of propositions `temp_prop7a` and `temp_prop7b`, `temp_prop8a` and `temp_prop8b` stems from the fact that the initial state of the system is the state $S1$.

5.4 SMV Model checking

The SMV system based on model checking algorithms works over a check structure consisting of binary decision diagrams, for more details see [1]. The model checker, using the efficient algorithms, check every symbolic state the system could get in.

In fact, the *state space*, we could say the global space, in the SMV verification tool, is formed with all the variables that occur in the verification program for SMV, and with their reciprocal combinations.

Thus, in our case, we have two input variables a, b that are naturally boolean, and a classical state variable called state with three possible values $S1$, $S2$ and $S3$. The values of the state variable are put into 1-of-N code. So, we have now three additional boolean variables, artificially created. All the boolean variables with their respective values create now a state space with 2^n global states $s_1, s_2, s_3, \dots, s_n, \dots, s_{2^n}, n$ number of all (original as well as transformed) variables. This is these states precisely that are characterized by all the combinations of the values the state space variables range over, and therefore called *global*.

A *state of the model* is thus an assignment of truth values to a set of state space 'boolean' variables a, b, $S1, S2, S3$. However, we manage only practically realizable states.

Figure 4 shows how this space is constructed. We have now, really reachable, 12 global states in the state space model, with propositions valid (present) in appropriate states (input signals $a, \overline{a}, \dots$ representing propositions φ: $a = 1$ resp. $\overline{a} = 1, \dots$ *etc*; and states $S1, S2, \dots$ representing propositions ψ: *state* $= S1, \dots$ *etc*; propositions about $\overline{S1}, \overline{S2}, \dots$ futile to consider).

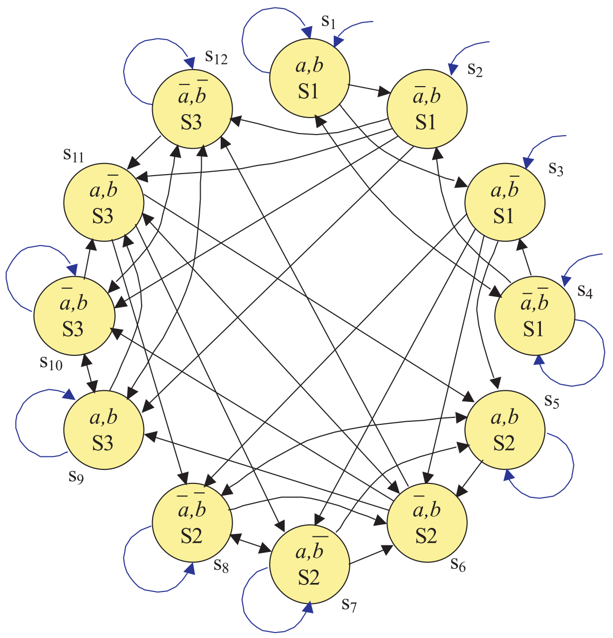


Fig. 4. State space model of a binary comparator with global states.

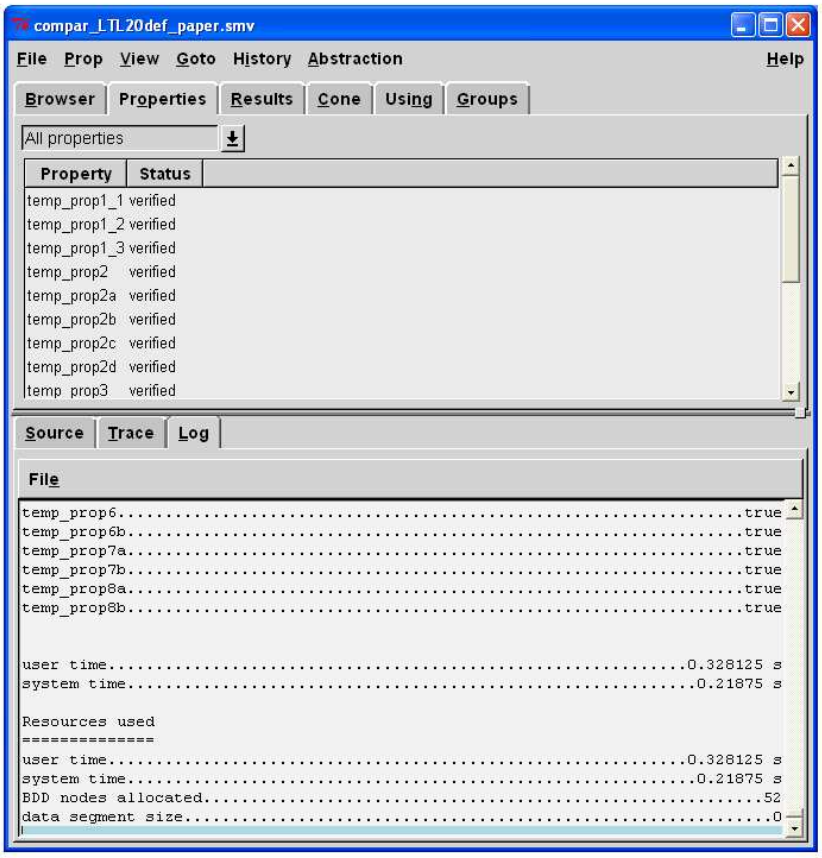
The state space we constructed in Fig.4 corresponds to the Kripke structure defined in **2.1.**

5.5 Results obtained

The model checker yields the following results:

The verification is total since it covers all possible behaviours of the system checked. Model checking was launch on our processor AMD Duron 758 MHz, 524 MB RAM. The resources used were:

User time	0.328125 s
System time	0.21875 s


compar_LTL20def_paper.smv
File Prop View Goto History Abstraction Help
Browser Properties Results Cone Using Groups
All properties
Source Trace Log
File

Property	Status
temp_prop1_1	verified
temp_prop1_2	verified
temp_prop1_3	verified
temp_prop2	verified
temp_prop2a	verified
temp_prop2b	verified
temp_prop2c	verified
temp_prop2d	verified
temp_prop3	verified

```
temp_prop6.........................................................true
temp_prop6b........................................................true
temp_prop7a........................................................true
temp_prop7b........................................................true
temp_prop8a........................................................true
temp_prop8b........................................................true


user time.....................................................0.328125 s
system time....................................................0.21875 s

Resources used
==============
user time.....................................................0.328125 s
system time....................................................0.21875 s
BDD nodes allocated.................................................52
data segment size....................................................0
```

Fig. 5. The final SMV Results window with results of checking

6 CONCLUSION

Logical errors in the design of digital systems, in transformational and concurrent programs and other such systems are becoming an increasingly important problem. They may, in particular, cause the failure of a critical device yet in use, or the loss of money and time if launched on the marquet with "bugs".

Methods for verifying such systems, as mentioned in the introduction, have been so far based on simulation and testing, so they can miss significant errors. If the systems can be viewed as having only a finite number of states, the alternative technique of verification, consisting in model checking application we showed in this paper can be advantageously used.

Acknowledgement

This work was supported by Slovak Science Grant Agency within the project No. VG 1/3104/06: "Systems of grid computing and its components".

References

[1] BRYANT, RANDALE. E. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams : ACM Computing Surveys **24** No. 3 (September 1992), 293–318.

[2] CLARKE, E. M.—GRUMBERG, O.—LONG, D. E. : Model Checking, NATO ASI Series F, Volume 152 (A survey on model checking, abstraction and composition), Springer-Verlag, 1996.

[3] HUTH, M. R. A.—RYAN, M. D. : Logic in Computer Science. Modelling and Reasoning about Systems, Cambridge University Press, 2004.

[4] MANNA, Z.—PNUELI, A. : The Temporal Logic of Reactive and Concurrent Systems. Specifications, Springer-Verlag, New York Inc, 1992.

[5] MCMILLAN, K. L. : The Model Checking System, In: SMV Reference Manual, Cadence Berkeley Labs, Berkeley, USA, 2002.

[6] MCMILLAN, K. L. : Cadence. Getting started with SMV, In: SMV Reference Manual, Cadence Berkeley Labs, Berkeley, USA, 1999.

[7] MCMILLAN, K. L. : The SMV Language, In: SMV Reference Manual, Cadence Berkeley Labs, Berkeley, USA, 1999.

[8] VARDI, M. Y. : Branching vs. Linear Time: Final Showdown, (Invited) European Joint Conferences on Theory and Practice of Software, Genova, Italy (April 2001).

Received 12 November 2006

Daniela Kotmanová, born in Bratislava, graduated from the Slovak University of Technology in Bratislava, Faculty of Electrical Engineering in 1984. She was a research worker at the Academy of Sciences, Institute of Control Theory and Robotics, now at the Faculty of Informatics and Information Technologies (FIIT) of the Slovak University of Technology in Bratislava. Her main area of research includes verification of reactive and concurrent systems, software systems, deductive methods, model checking, binary decision diagrams, nonclassical logics (temporal logics, modal logics).