APPLICATION OF MODEL CHECKING TO THE VERIFICATION OF DIGITAL SYSTEMS

Daniela KOTMANOVÁ, Milan KOLESÁR
Faculty of Informatics and Information Technologies,
Slovak University of Technology in Bratislava, Ilkovičova 3, 842 16 Bratislava 4, Slovak Republic,
tel. +421 2 60291359
E-mail: kotmanova@fiit.stuba.sk, kolesar@fiit.stuba.sk

ABSTRACT

The article presented here deals with the verification of a digital design using temporal logic and model checking. Temporal logic has been used as a specification language to catch the behaviour of the system design and model checking has been chosen as a verification method to evaluate the design accuracy.

As a digital system to be modelled and verified we have chosen the sequence detector; the detected string is 010. To compare the way how to establish temporal model and its properties and to see better coincidences and differences in describing their behaviour we made up the model of the sequence detector in two variants: without and with overlapping. We took, as finite state machine (FSM), the both automata, the Moore as well as the Mealy, each considered without and with overlapping.

After the temporal properties of the four drawn models had been defined we wrote respective programs corresponding to each of models, in SMV language, to pass them as input files into the SMV model checker. Finally, we tested the correctness of the models without and with overlapping also by submitting temporal properties of the weaker sequence detector to the stronger – the result was satisfying because all the fundamental properties of the Moore FSM without overlapping were verified also by the weaker Moore FSM with overlapping. Given the limited space available we made only this experiment, the other - with the Mealy FSM, could not be performed.

Keywords: *digital system design and model specification, temporal logic, verification, model checking*

1. INTRODUCTION

Temporal logics, in contrast to classical logics, include a notion of time. A classical proposition can be true or false independently of the time flowing. This is not the case for temporal logics. As the time flows truth values of propositions vary, in other words, formulas applied to states of a system, which evolves from state to state over time, swap their truth values from 0 to 1 and vice versa, depending on current state the system is in. See in details [1], [2], [8].

A system, modelled as a Kripke structure (explained later), with states labelled with propositions, has a dynamic behaviour which can be expressed in temporal formulas (so-called specifications of the model). Such system models and their dynamic behaviours are then described in a language proper to the verifying tool - a model checker - and verified. The verification is exhaustive since model checking, the method the model checker works with, is based on a total exploration of all the global states in the state space model of the design.

Simulation is so far a most widely spread hardware verification technique. Based on stimuli, provided by a user or automatically, execution paths of the chip model are imitated using a simulator. The simulator's output is compared with the required output. Unlike model checking verification method simulation suffers from the same limitation as testing: a restrictive set of input stimuli.

2. TEMPORAL LOGIC

2.1. Temporal operators

Temporal connectives according to their character can occur as single symbols such as X, F, G, U (linear time temporal logic LTL) or as a pair of symbols where each symbol of X, F, G, U must be preceded by an A or an E (branching time temporal logic CTL), for instance AX, EX, AF, EF etc.

Temporal connectives, singles or composed, usually precede, if unary, the propositions they are applied too. If not, an infix notation is used (binary temporal connectives, like U for example). For more, see [1], [2], [6].

The various temporal operators allow us to relate properties of the current state of a given digital system model with the properties of succeeding states of the model. We give some examples of the diverse applications of our logic to the sequence detectors modelled without and with overlapping. The logic used is the LTL logic.

In the paper, we applied the LTL temporal formulas to a sequence of states produced by the sequence detector. They indicate the character of the modifications the system has been exposed to.

We use essentially operators *X (next)* and *G (globally)*.

2.2. Linear and Non-linear Temporal Logic. Syntax

o Linear Time Temporal logic (LTL)

Temporal logic where the time is linear.

Syntax.
Formulas Φ in linear time temporal logic have the following syntax (in Backus-Naur form BNF):

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

p is any arbitrary atomic propositional formula

o **Non-linear Time Temporal Logic (CTL)**

Temporal logic with a time which is non-linear; called also Branching-Time Temporal Logic or Computer Tree Logic CTL).

Syntax.

Formulas Φ in branching time temporal logic have the following syntax (in Backus-Naur form BNF):

$$\Phi ::= \bot | \top | p | \neg\Phi | \Phi\wedge\Phi | \Phi\vee\Phi | \Phi\rightarrow\Phi | AX\Phi | EX\Phi | AG\Phi | EG\Phi | AF\Phi | EF\Phi | A[\Phi U\Phi] | E[\Phi U\Phi]$$

2.3. Model for temporal logics LTL and CTL (Semantics of LTL and CTL)

A model $\mathcal{M}$ for LTL and CTL is a labelled state-transition graph (a reachability graph), characterized by the triplet $(S,\mathcal{R},L)$, where

1. S is a finite set of states
2. $\mathcal{R}$ is a binary relation on S (underlying set $\mathcal{G}_{\mathcal{R}} \subseteq$ S x S is a graph of the relation):

$$\mathcal{R}: S \rightarrow S$$
$$s \mapsto s' \quad \text{s.t.} \quad s\mathcal{R}s'$$

and every $s \in S$ is reachable from the initial state and has some $s' \in S$, that is to say $\mathcal{R}$ must be total on S:

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

s' being a successor state of s in S

3. L is a labelling function:

$$L: S \rightarrow \wp(\text{Atoms})$$
$$s \mapsto L(s),\ L(s)\in \wp(\text{Atoms})$$

Ad 1: S represents a set of possible states of the system.

2. $\mathcal{R}$ gives the possible transitions between states, e.g. it says us how the system evolves, how it can move from state to state. Every state must be reachable from an initial state.

$\mathcal{R}$ is total means no state in the system can deadlock.

3. L assigns to each state s a set L(s) (which is associated with the state), i.e. a set of atomic propositions true in that particular state s.

Atoms = {p,q,...} is a set of all atomic propositions which can hold in the system

$\wp$(Atoms) = {∅, Atoms, {p}, {q}, ..., {p, q},…} is a power set of Atoms and

$$\text{Card}\,\wp(\text{Atoms}) = 2^{\text{Card Atoms}}$$

2.4. Path. Safisfaction relation.

Path.

A path π is an infinite sequence of states $(s_o, s_1, \ldots, s_i, \ldots)$ in a model $\mathcal{M}$ such that

$$\forall i \in \square \quad [(s_i, s_{i+1})\in \mathcal{G}_{\mathcal{R}}]$$

or a finite sequence of states $(s_o, s_1, \ldots, s_i, \ldots, s_n)$ in the model $\mathcal{M}$ such that

$$\forall i \in \{1,2,...,n\} \quad [(s_i, s_{i+1})\in \mathcal{G}_{\mathcal{R}}]$$

Satisfaction relation.

Let $\mathcal{M} = (S,\mathcal{R},L)$ be a model for LTL or CTL.

Given any path π and an LTL formula Φ, resp. any $s \in S$ and a CTL formula Φ , we denote the satisfaction relation $\models$ by

$$\mathcal{M}, \pi \models \Phi \quad \text{resp.} \quad \mathcal{M}, s \models \Phi$$

It says us whether a LTL formula Φ holds along a path π, resp. a CTL formula Φ in state s of the model $\mathcal{M}$. See [1], [2],[6] for more details.

2.5. Kripke structure (Oriented state graph)

An oriented labeled state-transition graph is a graphic representation of the model $\mathcal{M}$. Its nodes constitute global states of the state space of the design and contain all the propositional atoms, which are true in that particular state. The edges of the graph are oriented global state transitions.

This is called a *Kripke structure*. In Fig. 1 such a Kripke structure is shown, with the propositions valid in each state.

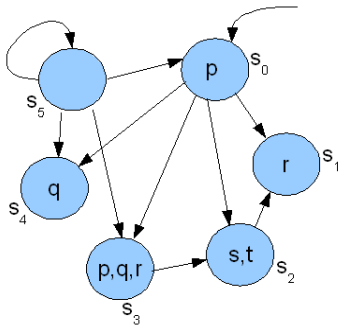


Fig. 1 An oriented state graph with the initial state s_o (s_o,s_1,...,s_5 are six states of the model containing each a set of atomic propositions {p}, {r}, {s,t}, {p,q, r}, {q}, {∅} respectively).

3. DIGITAL DESIGN VERIFICATION – MODEL CHECKER SMV

Schematic representation of the development

The relation between simulation and model checking can be seen in Fig. 2. We modified the schema in [2] Katoen, Principles of model checking.

Model checker SMV (Symbolic Model Verifier)

The design of sequence detectors we made will be verified using the SMV model checker – Symbolic Model Verifier [3], [4], [5].

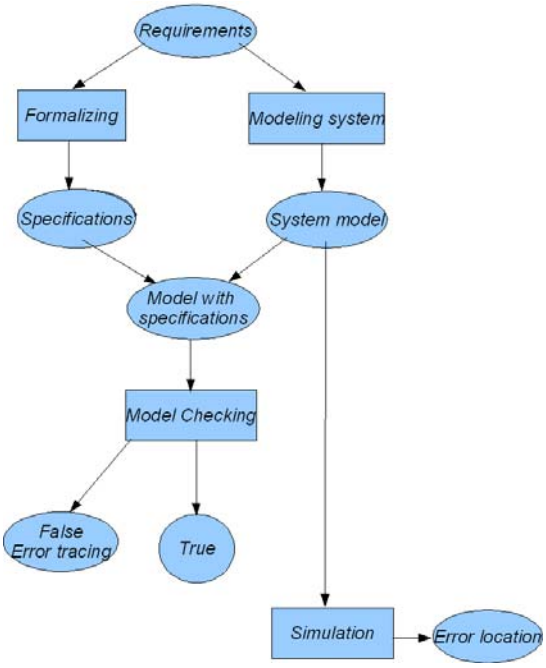


Fig. 2 Schema of progressive steps in verification of a digital design

The SMV provides its own language for describing the models built as state-transition diagrams, and includes syntax of languages for linear and branching time temporal logics, for describing model behavior.

Specifications, composed of classical logical propositions and accompanied by temporal operators form temporal formulas. Logical propositions are atomic propositions which hold in various single global states of the state space model built by the model checker. Then a model checking consists of verifying the validity of temporal formulas in all the global states the system can reach.

The input to the SMV verification system is given by a program in SMV language, which describes the model of the system to verify and also includes its specifications. The SMV system produces as output either the word „true“ if the temporal formulas hold for all initial states, or shows a trace which indicates why the specification fails for the chosen model.

The SMV model checker verifies, by working on the principle of BDD´s (Binary Decision Diagrams), that every possible behavior of the system satisfies the specification. This is also the main disadvantage of the use of SMV model checker, considering the state explosion problem in the state space model – there is too many variables to check (the number of states increases exponentially with the number of variables in the state space).

In this context, the complexity of the algorithms used is very important. The time of the verification can be indeed very long, and the capacity of the computer SMV model checker is run on, could be insufficient for the system to be verified.

The SMV system offers some tools to reduce the verification of such large and complex systems, such as compositional verification, refinement, symmetry reduction, temporal case splitting, data type reduction, induction. More in [3], [4],[5], [6].

4. EXAMPLE OF VERIFICATION – SEQUENCE DETECTOR

A sequence detector is a state machine which outputs a logic 1 whenever the required input pattern is detected, and outputs a logic 0 otherwise. The input to the device is supplied serially, bit after bit, one bit per time. The state machine can or cannot accept overlapping.

To illustrate how such a detector works we consider both Moore and Mealy model of finite state machine (FSM), each with and without overlapping. Input sequences are chosen arbitrarily and are the same for all four cases. Generated outputs, as shown below, mutually differ in spite of the inputs are identical.

We have chosen the input pattern 010. The sequence detection will be effectuated without and with overlapping. The chronological input-output successions below offer an example of how a sequence detector would work if distinguished whether an overlapping is applied or not.

The principal difference in temporal behaviour between non-overlapping and overlapping sequences is illustrated on the following sequences:

Moore

o without overlapping

input	0	1	1	0	0	1	0	1	0	1	0	0	1
output	0	0	0	0	0	0	0	1	0	0	0	1	0

o with overlapping

input	0	1	1	0	0	1	0	1	0	1	0	0	1
output	0	0	0	0	0	0	0	1	0	1	0	1	0

Mealy

o without overlapping

input	0	1	1	0	0	1	0	1	0	1	0	0	1
output	0	0	0	0	0	0	1	0	0	0	1	0	0

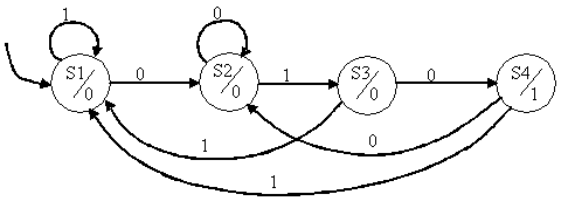
o with overlapping

input	0	1	1	0	0	1	0	1	0	1	0	0	1
output	0	0	0	0	0	0	1	0	1	0	1	0	0

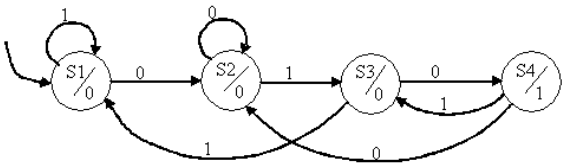
4.1. Models of the sequence detectors (Finite State Machine, FSM)

Moore

o without overlapping

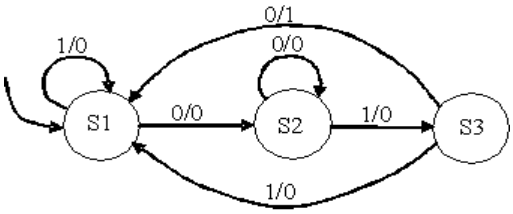


o with overlapping

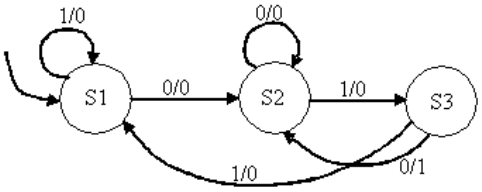


Mealy

o without overlapping



o with overlapping



4.2. Behaviors of the Models and Specifications

When a required sequence is detected the output is set to 1.

If we consider *a* to be an input and *y* an output the following assertions give a true picture of the dynamics of the system:

Moore

o without overlapping

If we have, in the zero time unit, on the device input a=0, in the first time unit a=1, and in the second time unit a=0 then the output *y* will be set to 1 in the time unit which comes immediately after.

If, closely after having detected the required sequence, a new 1 followed by a 0 occurs the output may not be set to 1 (to guarantee non-overlapping).

o with overlapping

If we have, in the zero time unit, at the device input a=0, in the first time unit a=1 and in the second time unit a=0 then the output *y* will be set to 1 in the time unit which comes immediately after.

If, closely after having detected the required sequence, a new 1 followed by a 0 occurs at the device input the output must be set to 1 again (to assure overlapping).

Mealy

o without overlapping

If we have, in the zero time unit, at the device input a=0, in the first time unit a=1, and in the second time unit a=0 then the output *y* will be set to 1 in the same time unit as the last 0 appears at the input.

If, closely after having detected the required sequence, a new 1 followed by a 0 occurs the output may not be set to 1 (to guarantee non-overlapping).

o with overlapping

If we have, in the zero time unit, at the device input a=0, in the first time unit a=1 and in the second time unit a=0 then the output *y* will be set to 1 as soon as the last *a*=0 appears at the input..

If, closely after having detected the required sequence, a new 1 followed by a 0 occurs at the device input the output must be set to 1 either (to assure overlapping).

4.3. Temporal properties

Moore

o without overlapping

$G(a=0 \wedge X(a=1) \wedge XX(a=0 \wedge state=S3) \leftrightarrow XXX(y=1))$

$G(y=1 \wedge a=1 \wedge X(a=0) \rightarrow XX(y \neq 1))$

o with overlapping

$G(a=0 \wedge X(a=1) \wedge XX(a=0) \leftrightarrow XXX(y=1))$

$G(y=1 \wedge a=1 \wedge X(a=0) \rightarrow XX(y=1))$

Mealy

o without overlapping

$G(a=0 \wedge X(a=1) \wedge XX(a=0 \wedge state=S3) \leftrightarrow XX(y=1))$

$G(y=1 \wedge X(a=1) \wedge XX(a=0) \rightarrow XX(y \neq 1))$

o with overlapping

$G(a=0 \wedge X(a=1) \wedge XX(a=0) \leftrightarrow XX(y=1))$

$G(y=1 \wedge X(a=1) \wedge XX(a=0) \rightarrow XX(y=1))$

4.4. Program in SMV Language

Moore

o without overlapping

```
module main (start,done)
{
   start,done: boolean;
   state: {A1,A2,A3,A4};

   init(state):=A1;
      next(state):=
       switch(state){
```

```
         A1:start?A1:A2;
         A2:start?A3:A2;
         A3:start?A1:A4;
         A4:start?A1:A2;
       };
       done:=(state=A4);

/*Verification of temporal properties*/

  temp_prop1: assert G(start=0&X
start=1)&XX(start=0&state=A3)<->
XXX(done=1));

  temp_prop1a : assert G(start=0&X(
start=1&state=A2&XX(start=0&state=
A3)<-> XXX(done=1));      /*true*/

  temp_prop1b : assert F(start=0&X(
start=1&XX(start=0)<->XXXdone=1);
  /*true but F not sufficient*/

/*temp_prop1c: assert G(start=0&X(
start=1)&XX(start=0)<->XXX(done=1) );
                          false*/

  temp_prop2: assert G(done=1&start
=1&X(start=0)-> XX(done~=1));

/* temp_prop2a: assert G(done=1&
start=1&X(start=0)<->XX(done~=1));
            false, naturally*/
}
```

o with overlapping

```
module main (start,done)
{
     start,done: boolean;
     state: {A1,A2,A3,A4};

     init(state):=A1;
       next(state):=
       switch(state){
         A1:start?A1:A2;
         A2:start?A3:A2;
         A3:start?A1:A4;
         A4:start?A3:A2;
       };
       done:=(state=A4);

/*Verification of temporal properties*/

  temp_prop1: assert G(start=0&X(
start=1)&XX(start=0)<->XXX(done=1) );

  temp_prop2: assert G(done=1&
start=1&Xstart=0-> XX(done=1));

/*However,bi-implication is false*/

/* temp_prop2a: assert G(done=1&
start=1&Xstart=0<->XX(done=1));
                        false

  temp_prop2b: assert G(XX(done=1) -
>(done=1&start=1&X(start=0)));
                     false*/

/*Supplement*/

/* temp_prop1a : assert G(start=0&X(
start=1&state=A2)&XX(start=0&state=A3)<
->XXX(done=1));
  false,but unilateral -> is true*/

  temp_prop1b: assert G(start=0&X(
start=1&state=A2)&XX(start=0&state=A3)-
>XXX (done=1));       /*true*/

/* temp_prop1c: assert G(XXX(done= 1)-
>start=0&X(start=1&state=A2)&XX(
start=0&state=A3));          false*/

  temp_prop1d: assert G(start=0&X(
start=1)&XX(start=0&state=A3)<->XXX
(done=1));                /*true*/
}
```

Mealy

o without overlapping

```
module main (start,done)
{
     start,done: boolean;
     state: {A1,A2,A3};

     init(state):=A1;
       next(state):=
       switch(state){
         A1:start?A1:A2;
         A2:start?A3:A2;
         A3:start?A1:A1;
       };
       done:=(~start&state=A3);

/*Verification of temporal properties*/

  temp_prop1: assert G(XX(done=1) <-
>(start=0)&X(start=1)&XX(start=0&
state=A3));

  temp_prop1a: assert G(XX(done=1) <-
>(start=0)&X(start=1&state=A2)&
XX(start=0&state=A3));      /*true,
but too strong - inutile*/

/* temp_prop1 and temp_prop1a are, as
it happens, equivalent ...*/

  temp_prop1b: assert (G(XX(done=1) <-
>(start=0)&X(start=1)&XX(start=0
&state=A3)))<->(G(XX(done=1)<->(
start=0)&X(start=1&state=A2)&XX
(start=0&state=A3)));

/* temp_prop1c: assert G(XX(done=1) <-
>(start=0&X(start=1)&XX(start=0)) );
               false, but F true */
```

```
  temp_prop1d: assert F(XX(done=1) <-
>(start=0&X(start=1)&XX(start=0)) );
                      /*true*/

  temp_prop2: assert G(done=1&X(
start=1)&XX(start=0)->XX(done~=1));
}
```

o with overlapping

```
module main (start,done)
{
     start,done: boolean;
     state: {A1,A2,A3};

     init(state):=A1;
       next(state):=
       switch(state){
         A1:start?A1:A2;
         A2:start?A3:A2;
         A3:start?A1:A2;
       };
       done:=(~start&state=A3);

/* Verification of temporal
properties*/

  temp_prop1: assert G(XX(done=1) <-
>(start=0&X(start=1)&XX(start=0)) );

  temp_prop1a: assert G(X(done=1) <-
>(start=1&state=A2)&X(start=0&
state=A3));               /*true*/

/* Relation existing between 1 and 1a
(equivalence)*/

  temp_prop1b: assert (G(X(done=1) <-
>(start=1&state=A2)&X(start=0&
state=A3))<->G(XX(done=1)<->(start=
0&X(start=1)&XX(start=0))));

  temp_prop1c: assert (G(XX(done=1) <-
>start=0&X(start=1)&XX(start=0))) -
>(G(X(done=1)<->(start=1&state=A2)
&X(start=0&state=A3)));

  temp_prop1d: assert (G(X(done=1) <-
>(start=1&state=A2)&X(start=0&
state=A3)))->(G(XX(done=1)<->start=
0&X(start=1)&XX(start=0)));

  temp_prop2: assert G(done=1&X(
start=1)&XX(start=0)->XX(done=1));
}
```

4.5. Comparing two models of Moore

In the end we took the Moore's model with overlapping and subjoin the specifications from the Moore's model without overlapping to its specification.

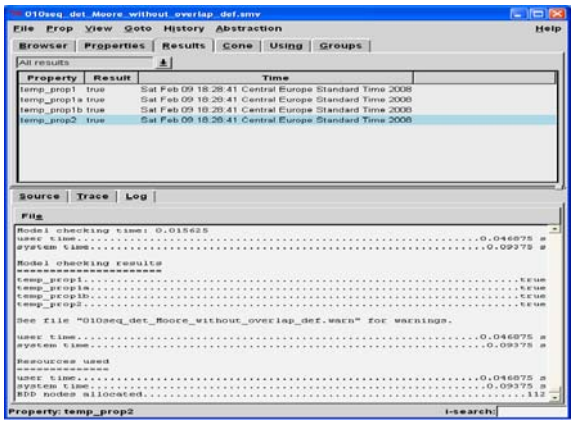
```
module main (start,done)
{
     start,done: boolean;
     state: {A1,A2,A3,A4};

     init(state):=A1;
       next(state):=
       switch(state){
         A1:start?A1:A2;
         A2:start?A3:A2;
         A3:start?A1:A4;
         A4:start?A3:A2;
       };
       done:=(state=A4);

/*Verification of temporal properties -
with overlapping; the temporal
properties here are the same as in the
original program, except for the suffix
"with", to distinguish them from the
ones in the file without overlapping*/

  temp_prop1_with: assert G(start=0
&X(start=1)&XX(start=0)<->XXX(done=
1));
  temp_prop2_with: assert G(done=
1&start=1&X(start=0)->XX(done=1));

/*However,bi-implication remains false
  temp_prop2a_with: assert G(done=
1&start=1&X(start=0)<->X(done=1));
                          false;

  temp_prop2b_with: assert G(XX(
done=1)->done=1&start=1&X(start=0) );
                            false*/

/*Supplement*/

/* temp_prop1a_with: assert G(start
=0&X(start=1&state=A2)&XX(start=0&
state=A3)<->XXX(done=1));
          /*remains false and
unilateral->remains true either*/

  temp_prop1b_with: assert G(start
=0&X(start=1&state=A2)&XX(start=0&
state=A3)->XXX(done=1));
               /*remains true*/

/* temp_prop1c_with : assert G(XXX(
done=1)->start=0&X(start=1&state=A2
)&XX(start=0&state=A3));
               remains false*/

  temp_prop1d_with: assert G(start
=0&X(start=1)&XX(start=0&state=A3) <-
>XXX(done=1));   /*remains true*/

/* Verification of temporal properties
- without overlapping; the temporal
properties here are the same as in the
primitive program, except for the
suffix "without" to distinguish them
from the ones in the file without
overlapping*/
```

```
  temp_prop1_without: assert G(
start=0&X(start=1)&XX(start=0&state=A3)
<->XXX(done=1));

/* temp_prop1a_without: assert G
(start=0&X(start=1&state=A2)&XX(
start=0&state=A3)<->XXX(done=1));
                  now false!*/

  temp_prop1b_without: assert F
(start=0&X(start=1)&XX(start=0)<->
XXX(done=1));              /*remains
true but F already insuffisant*/

  temp_prop1c_without: assert G
(start=0&X(start=1)&XX(start=0)<->
XXX(done=1));
  /*now true, evidently, see the 1st
prepos.temp_prop1_with*/

/* temp_prop2_without: assert G( done=1
& start=1 & X(start=0) -> X X done~=1);
now false, naturally*/

/* temp_prop2a_without: assert G(
done=1&start=1&X(start=0)<->XX(done
~=1));        false,in principle*/
}
```

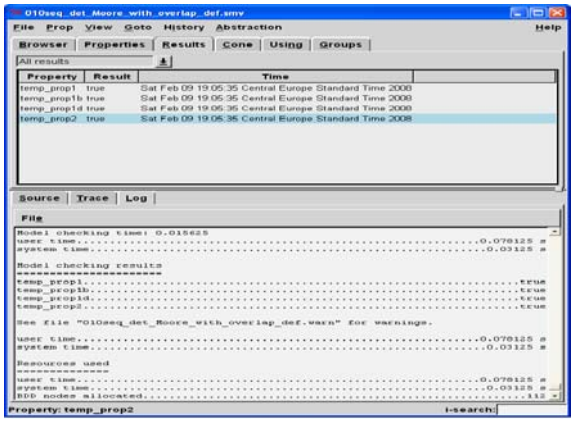
5. VERIFICATION AND RESULTS

Moore

o without overlapping

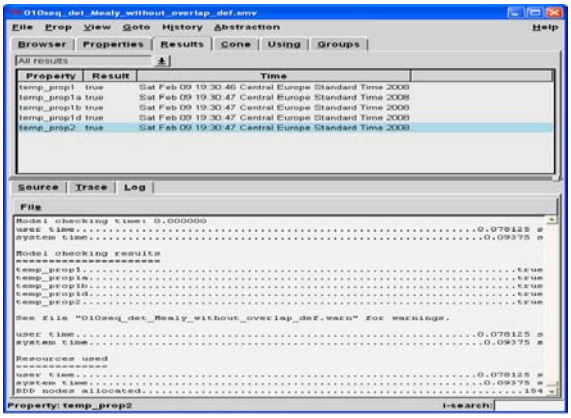


o with overlapping

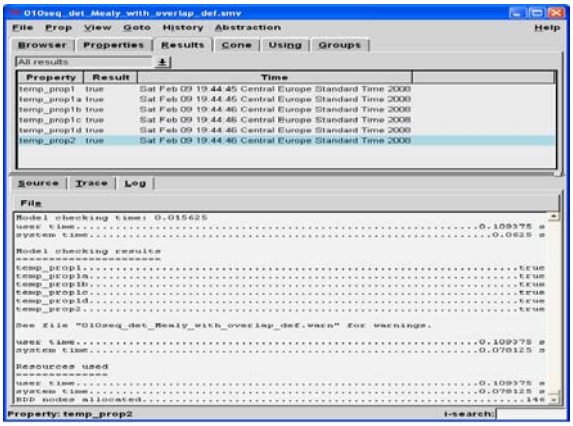


Mealy

o without overlapping



o with overlapping



6. CONCLUSION

We mention also resources used, at least orientationally. Model checking was launch on our processor AMD Duron 758 MHz, 524 MB RAM.
The resources used were:

User time	0,109375 s
System time	0,0625 s

The main disadvantage of model checking is not only the state explosion problem but also a finite number of states in the system. If the design can be assumed to have a finite number of states the algorithmic verification technique of model checking we showed in this paper can be advantageously applied.

ACKNOWLEDGEMENT

This paper was created within the research project VEGA 1/3104/06 supported by Scientific Grant Agency of the Ministry of Education of Slovak republic and the Slovak Academy of Sciences.

REFERENCES

[1] Huth, M. - Ryan, M.: Logic in Computer Science. Modelling and Reasoning about Systems. Cambridge University Press, 2nd Ed. 2004 Cambridge, 427 pp.

[2] Katoen, J.-P.: Principles of Model Checking, Formal Methods and Tools Group, University of Twente. Lecture Notes 2004-5.

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

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

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

[6] McMillan, K. L.: Symbolic Model Checking. Kluwer Academic Publishers, 1993

[6] Clarke, E.M. - Grumberg, O. - Long, D.E.: Verification Tools for Finite-State Concurrent Systems, LNCS 803. In: *The proceedings of REX school/symposium on A decade of concurrency: reflections and perspectives*, Noordwijkerhout, The Netherlands, June 1993

[7] Clarke, E.M. – Grumberg, O. – Long, D.E. (1996).: Model Checking, In Springer-Verlag NATO ASI Series F, Volume 152, 1996 .(A survey on model checking, abstraction and composition).

[8] Vardi, Moshe Y.: Linear vs. Branching Time: A Complexity-Theoretic Perspective. In: *Proc. 13th IEEE Symposium on Logic in Computer Science*, pp. 394-405, 1998.

Received April 25, 2008, accepted November 11, 2008

BIOGRAPHIES

Daniela Kotmanová graduated (MSc.) at the Faculty of Electrical Engineering and Informatics at Slovak Technical University of Bratislava. She worked at the Institute of Control Theory and Robotics, in Slovak Academy of Sciences in Bratislava. Her works concerned domains of Temporal and Modal Logics, especially those related to Specification and Verification of Digital Systems, and to Models of Knowledge in Multi-agent Systems. Since 2003 she has been working as a research worker at the Institute of Computer Systems and Networks at the Faculty of Informatics and Information Technologies in Bratislava. His scientific research is centred on verification of digital systems designs, linear and branching time temporal logic, model checking and related issues.

Milan Kolesár received the diploma of electrical engineering and the Ph.D. degree in control engineering and computer science, both from Slovak University of Technology in Bratislava, Slovak Republic, in 1964 and 1979, respectively. He is engaged in teaching and research in the areas of switching circuits and digital systems design. He has published several scores of papers on switching theory and digital system design. He has four patents. In 1981 he authored a book titled Programmable Logic Controllers (in Czech) and 1990 a textbook titled Logical Systems (in Slovak).