SYMBOLIC MODEL VERIFIER – AN EDUCATIONAL FORMAL VERIFICATION TOOL

S. MĂTASE[*] **A. CAŢARON**[*] **A. CAŢARON**[**]

Abstract: *The modern verification of the integrated circuits is based on the a priori logical verification, in the design phase, instead of testing a prototype. This method was proved to lead to significant costs decreases. The paper presents the Symbolic Model Verifier (SMV), an educational formal verification tool. We present the new type of formal verification named model checking and we present a practical verification example.*

Key words: *educational tool, formal verification, Symbolic Model Verifier.*

1. Introduction

The verification has a very significant role in the life of any system. There is no product to not be tested prior to utilization. These very simple principles are also available for ASIC. The design of an integrated circuit has become a very complex operation since the dramatic increase of the complexity of integrated circuits. In the early years, design started with connecting the elementary gates in a prototype. The first testing was done by electrical measurements of the resulted product. The design has moved now into the area of the logical part of the circuit. There are specialized applications that automatically produce the layout starting from a high-level language code. These languages actually allow the designer to describe the circuit functionality.

It is obvious that to test a circuit is not necessary anymore to physically realize a prototype. The technology allows us now to make coherent tests of the software description of the circuit. These tests confirm that logically, the designed structure agrees with the specification constraints. The verification role is extremely important. Avoiding the errors before producing the physical circuit by describing it in a hardware description language (Verilog, VHDL etc.) reduces the very high financial efforts, even millions of US dollars.

Formal verification is a generic name for a collection of techniques to test the correctness of a system or program. Formal verification can be used to mathematically prove that the system is in accordance with the specifications. Formal verification needs a formal description of the system and a formal specification of its desired behaviour.

The purpose of this paper is to present a formal verification solution applied to a part of a network processor, specifically to the block implementation of the round-

[*]Dept. of Electronics and Computers, *Transilvania* University of Braşov.
[**]High School of Building and Constructions, Braşov.

robin mechanism. In order to achieve the intended goal, a formal verification tool named Symbolic Model Verifier (SMV) will be employed.

2. Traditional verification / formal verification

In the traditional approach, using the same hardware description language does the functional verification of a circuit. The inputs flow through the circuit producing outputs. The verifier has to generate the input signals according to *a priori* schemes, which may have to be applied during the circuit use. More importantly, he has to study the output signals generated by the system. Let us consider a circuit with 2^{200} Boolean inputs. The total number of different input combinations is 2^{200}. To verify the circuit for all these possible combinations is an impossible task.

From the designer point of view, this verification is not sufficient. Many errors can be hidden, representing *corner cases*. These errors lead to unpredictable evolutions of the system, far from the design specifications. This is the reason why it was necessary to introduce a safer cover of the input signals space.

The formal verification is a new technique to test the circuits described in a high level language. Its goal is to cover the input signals space by providing a mathematical proof that a circuit performs correctly, according to the design specifications.

3. SMV – an academic formal verification application

At this moment, many professional formal verification tools are available. For example, IBM developed RuleBase, Cadence developed FormalCheck. RuleBase employs a formal verification method called Symbolic CTL Model Checking. This paper will shortly present SMV (Symbolic Model Verifier), a verification tool developed by a team from Berkley University, U.S., coordinated by Ken McMillan. This software application uses the verification model named *model checking*. This is equivalent with an exhaustive verification of the possible input signals space.

In *model checking* the specifications are sets of properties to be verified. These properties are presented with *temporal logic*, a special notation used to simply express temporal relations between signals. The properties have to meet some specifications connected to "legal" input signals, the allowed combinations of the circuit inputs. The combinations are generated by a set of interconnected finite state machines, with the result of producing all possible legal inputs for the tested circuit. An important principle implemented by SMV to produce the legal inputs is the non-determinism. Using this concept, the finite state machines generating the inputs are able to produce the whole area of values. This exhaustive verification model is appropriate for small size circuits. In practice, the circuits have much bigger dimensions.

To verify such a circuit, the user has to decompose it in small enough modules, which can be explored exhaustively with *model checking*. The name of this method is *compositional verification*. SMV integrates some other techniques to adjust complex circuits in order to verify them both by *model checking*: symmetry reduction, temporal case splitting, data type reduction, induction, etc.

4. Case study: a round-robin block verification

We will suppose that we have to verify the component circuit of a network processor. Its role is to serve concurrent requests for accessing a unique transmission line. The negotiating procedure is *round-robin*.

The circuit has request inputs (`req1`, `req2`, `req3`, `req4`), `clock` and `reset` inputs and an output, corresponding to each a request (`ack1`, `ack2`, `ack3`, `ack4`). We assume that initially the system is trying to serve request no. 1. At each step it tries to serve the request with the closest order number to the last served request, using the following algorithm:

```
...
Step i: serve request k
Step i+1: serve the request
(k+l mod 4) if active, where
l is an integer from 1 to 4
and the serving priority is
ascending order of l. If no
request is active, no answer
is issued.
...
```

The SVM code contains:

a) The designed circuit code (written in SMV or Synchronous Verilog)

b) The SMV code with the input generation

c) The SMV code for properties verification.

We will present the code corresponding to b) and c).

The generation of the input signals can be viewed as an "environment" generation.

The `clock` signal is Boolean and is active at each moment, equivalent to the system clock.

The `reset` signal is also Boolean, and is active only at the initial moment, then becomes inactive. This signal can have a non-deterministic behaviour, but this restriction does not influence the generality of the verification. The request signals are the four components of the array `req`, for an easier processing.

```
clk : boolean;
clk := 1;

reset : boolean;
init(reset) := 1;
next(reset) := 0;

req : array 0..3 of boolean;
init(req[0]) := 0;
init(req[1]) := 0;
init(req[2]) := 0;
init(req[3]) := 0;
for (i=0; i<4; i=i+1)
next(req[i]) :=
 case{
  req[i]=1 & gnt[i]=1
           : {0, 1};
  req[i]=1 : 1;
  default  : {0,1};
 };
```

These code lines produce the legal inputs corresponding to the requests.

How are the inputs generated corresponding to the 4 requests?

Initially, we will consider all the requests on 0, which is the system is in the reset state. Non-deterministic delayed Mealy automata, as the following, model the evolution of the requests:

a) When emitting the request `req[i]` and it receives an acknowledgement signal `gnt[i]` that is it is served, the evolution of the request is not determined and the value at the next clock can be 0 or 1;

b) When the request was asserted and the acknowledgement signal has not been received yet, its value at the next clock is the same as the current one, 1;

c) If the request was not asserted at the current step, at the next step it will have a not-determined value, 0 or 1, as in the first situation.

For the purpose of verification, the code has to be extended by a set of user variables, used in a more visible and compact writing of the testing properties.

We introduce the variable served in this module. It actually implements the arbitration mechanism. This variable stores the index of the last served request. This value is used to compute the next request to be served.

```
served : 0..3;

init(served) := 3;
next(served) :=
 case{
  reset : 3;
  gnt[(served + 1) mod 4]
       : (served + 1) mod 4;
  gnt[(served + 2) mod 4]
       : (served + 2) mod 4;
  gnt[(served + 3) mod 4]
       : (served + 3) mod 4;
  gnt[(served + 4) mod 4]
       : (served + 4) mod 4;
  default : served;
 };
```

We initially consider the line "parked" on request 3, which is assigned value 1. In other words, this is the last served request. The next request to be served will be chosen from requests 0, 1, 2, 3, in this order, corresponding to the associated request value.

The automata implementing this behaviour were presented in this section and cover all the situations in a real-world use.

The testing code is composed by one or more assertions or rules, implementing the necessary properties of the given specifications. These properties are:

a) The arbitration to be fair, that is each asserted request to finally receive an acknowledgement signal;

b) The sending of the acknowledgement signals is accordance with the arbitration mechanism;

c) It is not possible to issue more acknowledgement signals at one moment;

d) It is not possible to issue an acknowledgement signal when the associated request has not been asserted.

The code that implements these properties is the following:

```
rule_fairness1  : assert G req[0] -> F gnt[0];
rule_fairness2  : assert G req[1] -> F gnt[1];
rule_fairness3  : assert G req[2] -> F gnt[2];
rule_fairness4  : assert G req[3] -> F gnt[3];

rule_arb0  : assert G gnt[0] -> X served=0;
rule_arb1  : assert G gnt[1] -> X served=1;
rule_arb2  : assert G gnt[2] -> X served=2;
rule_arb3  : assert G gnt[3] -> X served=3;

rule_no_more_than_one_grant_at_the_time:
           : assert G gnt[0]+gnt[1]+gnt[2]+gnt[3] <= 1;

rule_grant0_with_request0
           : assert G gnt[0] -> req[0];
rule_grant0_with_request1
```

```
           : assert G gnt[1] -> req[1];
rule_grant0_with_request2
           : assert G gnt[2] -> req[2];
rule_grant0_with_request3
           : assert G gnt[3] -> req[3];
```

To write these rules, we used an instructions subset of SMV that implements the temporal logic. Three new operators, G, F and X represent this type of logic:

a) G - the *global* operator: in each state of the system evolution, the suffixing property has to be true. *G m* means that *m* will be true at all times in the future;

b) F - the *future* operator: there is a state in the system evolution where the suffixing property is true. *F m* is true if *m* will be true at some time later;

c) X - the *next* operator: in any of the states possible to occur at in the next clock state, the suffixing property is true. *X m* means that *m* will be true at the next time.

For example, the following rule:

```
rule_fairness4  :
 assert   G   req[3]   ->   F
gnt[3];
```

has to be read: "In any state of the system evolution, if request 3 is asserted, starting with this state, there is a state in the future where the acknowledge signal will be asserted."

5. Conclusions

In this paper we have presented Symbolic Model Verifier, a formal verification tool, suited for educational use in digital circuit design. We also presented the new type of formal verification named model checking and a practical verification example.

The paper shows the methodology of applying SMV verification on a round-robin block. The results proved the improved performance of the formal verification techniques compared to the regular verification as follows: the verification is exhaustive, the input space modeling is simple and it frees the verification engineer from building functional scenarios used in the traditional verification.

The complete verification of a system can be done only by formal verification. It offers the certitude that the tested system performs correctly in every possible case.

Though SMV was developed for academic purposes, it is possible to use it for quite complicate circuits. Especially for designs including large data path components, the user must split them into small enough parts for SMV to verify. This technique is known as compositional verification.

From the testing engineer's point of view, SMV is easy to use, as long as the base syntax is similar to the known HDLs and the temporal logic is easy to understand. This allows concise specifications about temporal relationships between signals. The main problem in the industrial use of formal verification is the high cost of the software tools and the hardware needs. The costs with the smaller number of errors in the logic design of the circuit bring the main advantages.

This tool was designed for use in an academic environment. It offers an easier design and functional verification of integrated circuits. This recommends it as a good option for implementing training in formal verification.

References

1. McMillan, K.: *The SMV language.* Berkley University, CA, USA, 1998.

2. McMillan, K.: *Getting started with SMV*. Berkley University, CA, USA, 1998.
3. http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/index.html *RuleBase presentation*.
4. http://www.cadence.com/datasheets/formalcheck.html *FormalCheck presentation*.

System Model Verifier – o aplicație educațională pentru verificarea formală

Rezumat: *Verificarea modernă a circuitelor integrate se bazează pe verificarea logică a priori, încă din faza de proiectare, înlocuind verificarea şi testarea unui prototip. Această metodă, aşa cum s-a dovedit practic, conduce la scăderi semnificative ale costurilor globale de proiectare şi realizare a unui produs. Acest articol prezintă System Model Verifier (SMV), o aplicație de verificare formală cu caracter educațional. Este prezentat noul tip de verificare formală, numit model checking, şi este dezvoltat un model practic de verificare formală.*

Cuvinte cheie: *software educațional, verificare formală, Symbolic Model Verifier.*

Recenzent: Conf. dr. ing. Dan Nicula.