Università degli Studi di Verona, Facoltà di Scienze MM. FF. NN.
Dipartimento di Informatica
EDALab: Embedded System Design Center

Ca' Vignal, Strada Le Grazie 15
37134 Verona, Italia
Tel. +39 045 8027069
Fax +39 045 8027068

Verona, 16/03/2007

An Introduction to SMV

Nicola Bombieri

Ca' Vignal, Strada Le Grazie 15
37134 Verona, Italia
Tel. +39 045 8027069
Fax +39 045 8027068

1 Required Background

Students interested in learning SMV are required to know the fundamentals of mathematical logic and basic concepts related to modeling and synthesizing digital systems. Moreover, it is advisable that students are familiar with at least one among the following traditional hardware description languages (HDL): VHDL, SystemC, Verilog.

2 Goal

The goal of this lecture consists of describing the basic concepts related to the use of the SMV model checker to formally verify the correctness of a digital system description with respect to the initial specification. Students will learn to:

- modeling a digital device by using the SMV language;
- defining temporal properties to formally describe the initial specification.

3 What is SMV?

SMV is a model checker, i.e., a tool for formal verification of finite state systems, like for example, hardware devices [1]. In particular, SMV is used to exhaustively verify that each behavior of the design under verification (DUV) satisfy the formal specification defined by means of temporal properties. On the contrary, dynamic verification allows to verify the DUV behavior only for the set of input stimuli provided during simulation.

SMV has been created to verify HW designs. In particular, it can be used to verify the correctness of RTL or gate-level models of HW components as reported in Figure 1. However, it is worth to note that model checking can be used to formally verify the correctness of SW programs too.

To use SMV it is required to provide:

- An SMV language-based description of the DUV.
- A set of CTL (Computation Tree Logic) or LTL (Linear Time Logic) temporal properties to verify.

Generally, SMV provides a response related to the validity of each verified property with respect to the DUV model. When the property fails, SMV generates a counterexample (a set of values for DUV signals involved in the property checking) to show the evidence of the verification failure. However, sometimes, SMV can be unable to complete the checking when the temporal/space resources (memory, computational time) provided to the tool are insufficient to manage very complex DUV models.

4 DUV Modeling

Generally, the mathematical model adopted by model checking tools is know as Kripke structure. According to this formalism, the DUV is represented as a triple $M = (S, R, L)$ where:

- S is a finite set of states;
- $R \subseteq S \times S$ is the next state relation;

Ca' Vignal, Strada Le Grazie 15
37134 Verona, Italia
Tel. +39 045 8027069
Fax +39 045 8027068

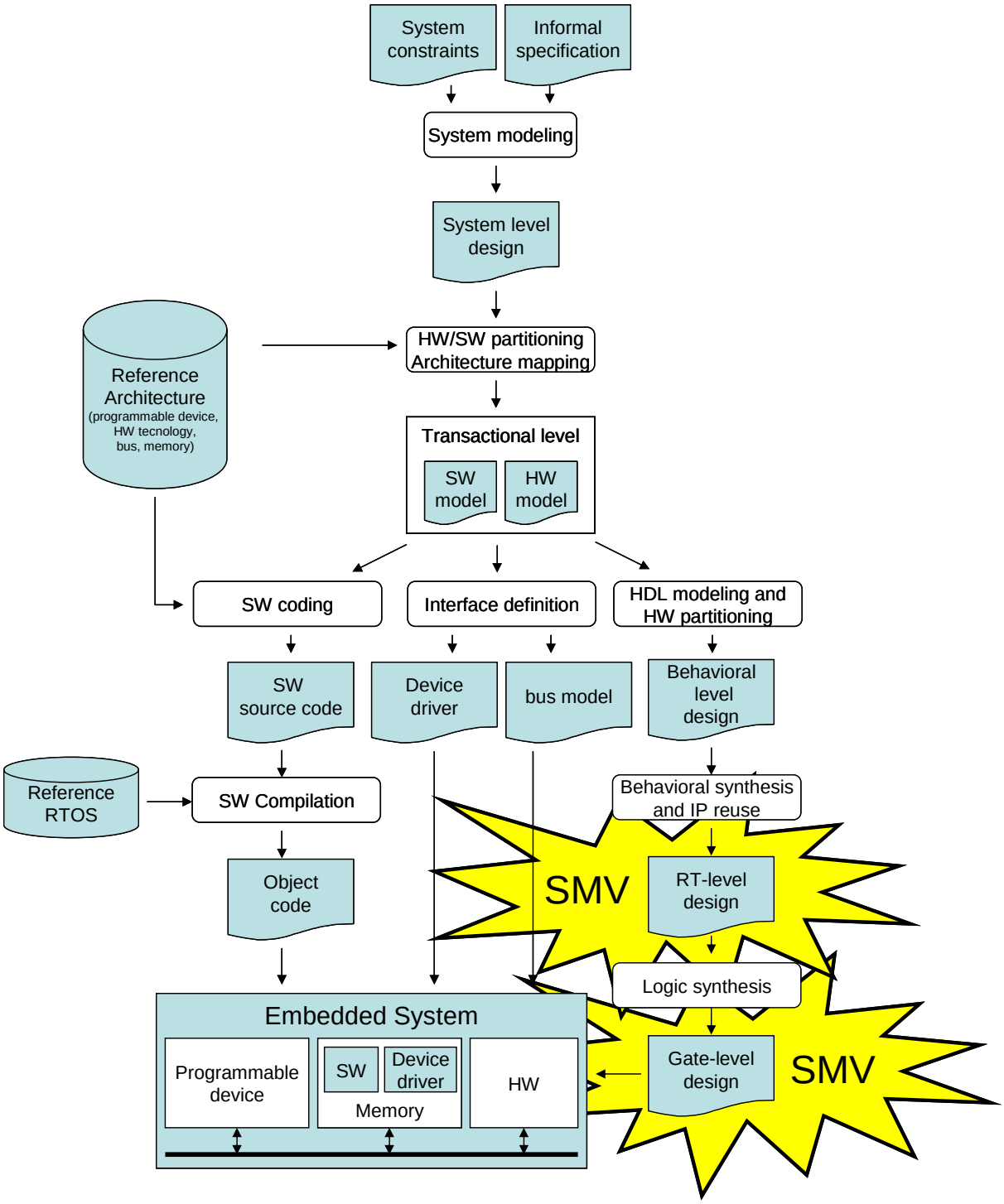

Figure 1: Embedded system design flow.

Ca' Vignal, Strada Le Grazie 15
37134 Verona, Italia
Tel. +39 045 8027069
Fax +39 045 8027068

- $L: S \rightarrow 2^{AP}$ is a function that labels each state in S with the set of atomic propositions that hold on it.

However, from the practical point of view, SMV users are not required to explicitly define the Kripke structure corresponding to the DUV. On the contrary, the DUV must be modeled by using a programming language similar to traditional HDLs, e.g. VHDL or Verilog. The details about the SMV language syntax can be found on the tool documentation [2,3], whilst the semantic details should be already present on the students background. For this reason, we report, in the following, only a commented example of a simple *bus arbiter* to provide a rapid panoramic view of the main constructs of the SMV language. The role of the *bus arbiter* consists of enabling accesses to the bus according to the requests sent by the devices which share it (in our example, let us suppose that the bus is shared between two devices). The arbiter has two inputs (for access requests) and two outputs (for access acknowledgments). In case of concurrent requests, the arbiter assigns the bus alternatively (initially the device with the highest priority is the one connected to input `req1`).

```
module main(req1,req2,ack1,ack2)    --main module declaration
{
  input req1,req2 : boolean;        --primary inputs declarations
  output ack1,ack2 : boolean;       --primary outputs declarations
  bit : boolean;                    --declaration of a state variable

  next(bit) := ack1;                --the value of bit at time t+1 will
                                    --equal the value of ack1 at time t

  if (bit) {                        --bit is used to alternate the priority
    ack1 := req1 & ~req2;
    ack2 := req2;
  }
  else {
    ack1 := req1;
    ack2 := req2 & ~req1;
  }
```

Expressiveness of the SMV language is the same of the HDLs previously cited. However, the SMV language is rarely used to model the DUV. Generally, the DUV is written by using VHDL, Verilog or SystemC, since such languages are supported by the majority of tools for automatic design adopted in the industry. In such cases, to avoid a manual translation of the code from the adopted HDL to the SMV language, it is sufficient synthesizing the HDL code to obtain a gate-level Verilog description mapped on equations (logical ports must not be mapped on any technology library, but they must be represented by means of the corresponding logical operators of Verilog). Then, the gate-level code can be automatically converted in SMV language by using the *vl2smv* command included in the SMV distribution. Finally, it is necessary to define a *main* module that instantiates the description of the DUV provided by the *vl2smv* as follows.

```
#include "DUV_filename.smv"   --to include the converted gate-level code
module main ()
{

--declarations of variable necessary to instantiate the DUV module
--one variable for each I/O signals must be declared
```

```
VAR
      param_1 : type;
      param_n : type;

--instantiation of the module(s) of the DUV

label : module_name (
      in_1 <== param_1,
      … …
      out_m <== param_n
);

-- definition of properties
}
```

It must be observed that *vl2smv* allows to avoid a manual translation from HDL to SMV, but the obtained code must be accurately checked. In some cases, the code generated by *vl2smv* must be manually corrected to remove syntax errors introduced during the translation.

Finally, it is worth to note that some synthesis tools (e.g., Mentor Graphics LeonardoSpectrum) generates assignment in the format:

```
operator (operand_1, operand_2, operand_3);
```

In such a case, it is necessary to modify the syntax of the assignment as follows:

```
operand_3 := operand_1 operator operand_2;
```

5 Temporal Properties Definition

To use SMV, properties derived from the specification must be defined by using LTL or CTL. Both such logics include the traditional operators of the first order logic (*and, or, not, implication*) and the four *temporal operators F, G, U, X,* necessary to specify properties dependent on the time. The distinction between the two logics derives from the mode they allow to manage the evolution of the DUV according to the time. In particular, in LTL the time is linear along a single computational path. On the contrary, in CTL, given a state, the past before it is unique, but the future can follow different computational paths. Then, in CTL the temporal operators are always headed by a *path quantifier* between *A* and *E* that are not included in LTL. Syntax and semantics of LTL and CTL are exhaustively defined in [1], thus, in this lecture we report only the intuitive meaning of temporal operators and path quantifiers. For this reason, it is necessary to define what *computational path* means with respect to a Kripke structure: it is an infinite sequence of states $\pi = (s_0, s_1, s_2, \ldots)$ such that $R(s_i, s_{i+1})$ hold for $i \geq 0$. In the following definitions, let us assume that *p, q* represent LTL properties, while *u* represents a CTL property.

5.1 Temporal Operators

- *F (eventually):* it is used to define a condition that must be true eventually in the future. For example, the property *Fp* is true if *p* is eventually true in the future.
- *G (always):* it is used to define a condition that must be always true. Thus, the property *Gp* is true if *p* is always true.

- *U (until):* it is used to define a condition whose truth depends on another condition. Thus, the property *pUq* (*p* until *q*) is true if *q* is true at some time *t* and *p* is true for each $t' < t$.
- *X (next):* it is used to define a condition that will be true on the next time. For example, *Xp* is true at time *t* if *p* is true at time *t+1*.

5.2 Path Quantifier

- *A (for all computational paths):* it is used to define that, given a state *s*, a property is always true whatever the computational path followed starting from *s* is. For example, the property *AGu* is true if *u* is always true in each computational path.
- *E (for some computational path):* it is used to define that, given a state *s*, a property is true in at least one computational path starting from *s*. For example, the property *EFu* is true if *u* eventually holds at some future time in at least one computational path.

5.3 Liveness and Safety

Independently from the adopted logics, two kinds of properties can be defined from the semantics point of view: *safety* and *liveness*. Safety properties are used to express the necessity that a bad condition never happens (e.g., deadlock, mutual exclusion failure, …). On the contrary, liveness properties are used to express the eventuality that some good condition happens in the future (e.g., service warrantee, program termination, …).

Now, let us consider the bus arbiter previously defined. In the following four LTL safety properties and one LTL liveness property are defined to check the correctness of the design.

```
--ack1 e ack2 must not be concurrently asserted (mutual exclusion)
mutex : assert G ~(ack1 & ack2);

--if the arbiter receives a request, it must answer to accord the bus
--access
serve : assert G ((req1 | req2) -> (ack1 | ack2));

--the bus access must be accorded to the device connected to ack1 only
--if this sent a request
waste1 : assert G (ack1 -> req1);

--the bus access must be accorded to the device connected to ack2 only
--if this sent a request
waste2 : assert G (ack2 -> req2);

--the bus access must be eventually assigned to the device with the lowest
--priority connected to req2 and ack2, or req2 is always deasserted
--(to avoid starvation of the device with the lowest priority)
no_starve : assert G F (~req2 | ack2);
```

To verify the previous properties, it is sufficient to include them in the same file of the bus arbiter model before the closed bracket of the *main* module.

6 SMV Limitation and Bounded Model Checking

SMV, like the majority of other model checkers, verifies a property by traversing the transition state graph of the DUV. During the verification of a property, SMV generates 2^n

states, where *n* represents the number of state variables (registers) included in the logic cone of the property (i.e., *n* is the number of state variables necessary to check if the property is true or false). Thus, the computational effort required to verify a property rapidly increases as the number of state variables raises, in particular when the states are explicitly enumerated. This can lead to the well known state explosion problem which causes the incapability of model checkers to complete the verification of complex systems without using advanced reduction techniques like *symmetry reduction, temporal case splitting, data type reduction, ...* These techniques are not covered in this lecture. To partially reduce complexity problems, SMV uses BDD (Binary Decision Diagrams) to implicitly represent the state graph [4]. However, SMV cannot manage very complex designs (e.g., with more than 100 state variables on a Sun Fire 280R equipped with a 750MHz dual UltrasparcIII processor and 4.0GB of RAM) without using the previously cited techniques related to compositional verification.

Instead of using the BDD-based version of SMV, a valuable alternative is represented by *Bounded Model Checking* (BMC). When the BMC mode is enabled, SMV accomplishes a semi-decidable verification of properties, i.e., it tries to generate a counterexample of finite length (the length is set by the user) by exploiting the capability of a SAT-solver to confute the validity of a Boolean function. If a counterexample is founded, then the user is guaranteed that the property is false. On the contrary, the incapability of generating a counterexample of the desired length is not a proof of the validity of the property, since a longer counterexample could exist.

SMV is arranged to be automatically interfaced to the *zchaff* SAT-solver [5]. In this way, the failure of a property can be verified more quickly with respect to the use of the exhaustive verification mode based on BDD.

7 SMV Commands

Command line

`smv` *`filename.smv`*	to run SMV in text mode.
`smv -bmc -l` *`val filename.smv`*	to run SMV in BMC mode with counterexample of length *`val`*.
`vw`	to run SMV in graphic mode.
`smv --help`	to show the help.

Graphic mode

Menu **`Prop → Verify all`**	to verify all the defined properties.
Menu **`Prop → Verify prop_nome`**	to verify only the property `prop_name` (after `prop_name` has been selected in the property frame).
Menu **`Prop → Options`**	to enable the BMC mode select `Use SAT-based bounded model checking` and set the counterexample length in the `Trace length bound` text area.
Tab **`Cone`**	to see the number of state variables involved in the verification of the property.

Ca' Vignal, Strada Le Grazie 15
37134 Verona, Italia
Tel. +39 045 8027069
Fax +39 045 8027068

8 Case Study

As a case study, some properties for the ADPCM component will be presented. The module is inserted into the V-CLIP system showed in Figure 2. It presented an API

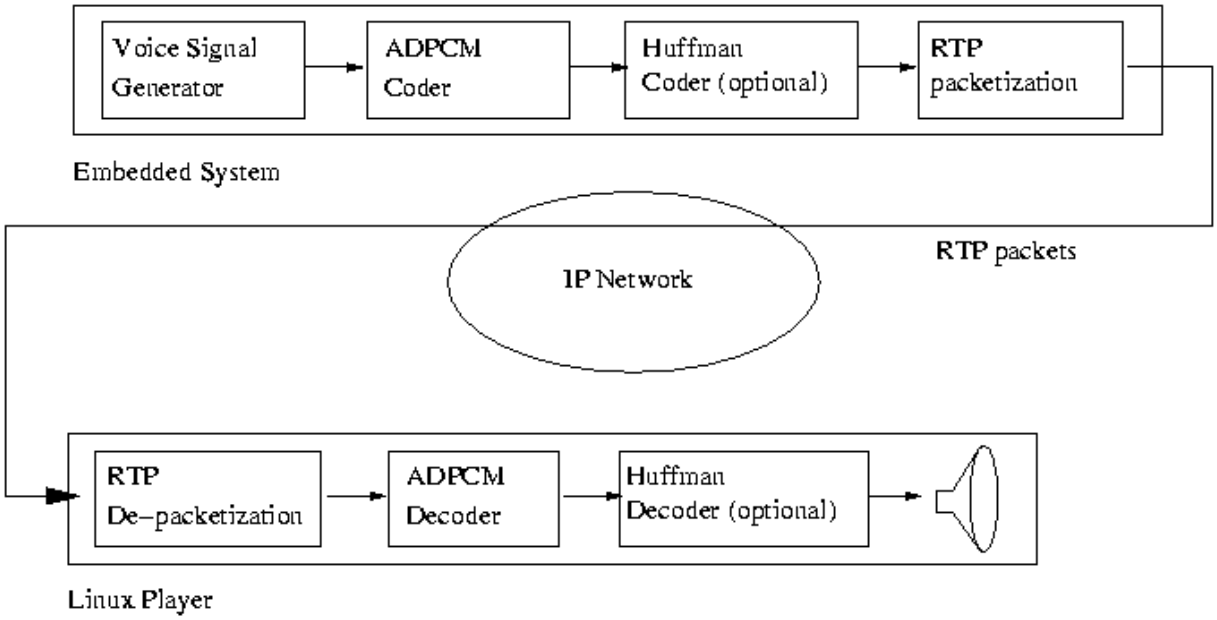

Figure 2: V-CLIP system.

9 References

[1] E.M. Clarke, O. Grumberg, D.A. Patel. *"Model Checking"*, MIT Press, 2000.
[2] K.L. McMillan. *"Getting Started with SMV"*, Cadence Berkley Labs, 1999.
[3] K.L. McMillan. *"The SMV Language"*, Cadence Berkley Labs, 1999.
[4] K.L. McMillan. *"Symbolic Model Checking"*, Kluwer Academic Publishers, 1993.
[5] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik. "*Chaff: Engineering an efficient sat solver"*. In: Proc. of the Design Automation Conference (DAC), pagg. 530—535, 2001.