An Automatic Design Flow from Formal Models to FPGA

Judson S. Santiago*
Advisor: David B.P. Déharbe

Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Campus Universitário
Lagoa Nova
59072-970 Natal, RN, Brazil
judson@dimap.ufrn.br

Resumo SMV [McM93] é uma linguagem apropriada para desenvolvimento de circuitos integrados e otimizada para verificação formal. VHDL [IEE93] é um formato de desenvolvimento apropriado para simulação e síntese de circuitos, mas não possui suporte para os propósitos de verificação formal. A contribuição deste artigo é a integração das duas abordagens através da definição de regras sistemáticas para traduzir programas SMV em descrições VHDL, provendo assim um componente importante para um ambiente de desenvolvimento automático de circuitos que faz uso eficiente dos métodos formais. Nosso processo de tradução tem como alvo um subconjunto específico de VHDL voltado à síntese de circuitos. Conseqüentemente, a descrição VHDL produzida pode ser automaticamente mapeada para dispositivos FPGA usando ferramentas de síntese comerciais.

Abstract SMV [McM93] is a language suitable for integrated circuit design and optimized for formal verification. VHDL [IEE93] is a design format suitable for simulation and synthesis, but poorly designed for formal verification purposes. The contribution of this paper is the integration of the two approaches through the definition of systematic rules to translate SMV programs into VHDL descriptions, providing thus an important component for an automated circuit design environment making efficient use of formal methods. Moreover, our translation process targets a synthesis-specific subset of VHDL language. Consequently, the produced VHDL descriptions may be automatically mapped to standard FPGA devices using commercial synthesis tools.

Keywords Formal methods, Circuit design automation, FPGA

1 Introduction

In the last 30 years, the size of digital circuits has sustained an exponential growth. The task of designing hardware systems has become incredibly complex and, consequently, is now extremely costly and error-prone. To cope with the continued challenge of using the continuously increasing quantity of available silicon to design more complex systems, the design automation comunity has seen whole new areas emerge in the last decade. High-level synthesis [Ber93] has allowed a higher level of abstraction in the design process and has become mainstream. Formal verification methods [CW96] (such as theorem proving, equivalence checking and model checking [BCMD90]) aiming to prove the correctness of the designs and commercial tools are now available.

* This research is sponsored by the Centro Nacional de Pesquisa (CNPq) under Iniciação Científica (IC) and Programa Temático Multi-institucional em Ciência da Computação (ProTem-CC) programs. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of CNPq, or the Brazilian Government.

However, because the computational complexity of formal verification is so huge, in practice these methods are used to find errors in critical parts of larger systems and they are today seen as a potential complement to traditional verification methods, such as test generation and simulation.

Therefore, the integration of traditional integrated circuits and systems design with formal methods in a coherent process is strategic for the production of error-free systems. Unfortunately, most formal verification tools have their own input language and are in some way incompatible with other IC design software, which are usually based on standard hardware description languages, such as VHDL [IEE93] or Verilog HDL. Most developments in formal verification are mainly realized in the realm of universities, and are available in academic tools, such as SMV [McM93]. SMV is used to fully automatically prove properties of the system beeing designed, but requires input in its own language. On the one hand, SMV is optimized for formal verification, but is not compatible with any CAD tools. On the other hand, VHDL may be used for a wide variety of purposes, including simulation and synthesis, but the intrinsic complexity of the language makes it difficult to verify formally [BCS95]. In this paper, we propose a tool that permits a design flow based on SMV, one of the most successful and widely used formal verification tool, and VHDL, a dominant hardware-description language. This design flow had originally been used manually to produce a VHDL description of a crossbar switch interconnection system. This experience showed that the translation is potentially automatable.

This paper contributes with a translator from SMV to VHDL. The produced VHDL description is at the register transfer level, and obeys common description rules at this level [IEE98]. This output can then be used to synthesize, place and route the product on FPGA devices using off-the-shelf CAD tools. In this paper we present the concepts that are used to build a translation tool from SMV to VHDL-RTL.

This paper is organized as follows. Section 2 briefly presents the SMV language and tool capabilities. Section 3 frames the VHDL language within this context, and Section 4 explains the translation process between both languages and presents some results of this work under progress. Section 5 concludes and plans for future developements of this work.

2 The symbolic model checker SMV

In general, model checking is an algorithm to check that a given structure is a model for a given formula expressed in some logic. SMV is indeed a type of model checker that verifies if a finite-state system, called a Kripke structure, satisfies properties written in the temporal logic CTL [CE81]. The verification algorithm, known as symbolic model checking, is based on exhaustive search algorithm in the state space of the given systems and uses binary decision diagrams [Bry86] to represent the structure state variables and transitions. SMV proofs are fully automatic, and provide counterexamples when a specification property is not specified by the system.

Synchronous as well as asynchronous systems can be described in the input language of SMV. Examples of designs that have been described and checked with SMV include protocols, software and hardware designs. The specification logic CTL allows to express important classes of concurrency properties, such as safety, fairness, liveness and freedom of deadlock. In addition, SMV can also compute quantitative properties, such as to check that an event occurs within a certain time interval after another occurs.

2.1 Kripke structures

SMV input is a special-purpose language, in which the user can describe the system under design and its desired properties. The semantics of the SMV description is defined in terms of Kripke structures, and the properties are expressed in the temporal logic CTL. SMV then checks the specified properties against the corresponding Kripke structure.

Let P be a finite set of boolean propositions, a Kripke structure over P is a quadruple $M = (S, T, I, L)$ where:

- S is a finite set of states.
- $T \subseteq S \times S$ is a transition relation, such that $\forall s \in S, \exists s' \in S, (s, s') \in T$.

– $I \subseteq S$ is the set of initial states.
– $L : S \rightarrow 2^P$ is a labeling function and associates with each state a set of boolean propositions true in the state.

A formula f is valid in structure M if it is valid for all initial states:

$$M \models f \text{ iff } \forall s \in I, M, s \models f.$$

Kripke structures form a suitable type of models to characterize a concurrent system, and may be used to describe software, protocols, synchronous or asynchronous hardware, etc.

2.2 The temporal logic CTL

CTL temporal logic formulas can be embedded in SMV source code and are interpreted as the specification of the described system. CTL (Computation Tree Logic) formulas can describe properties of branching structures, such as the execution tree of a Kripke structure. It is therefore possible to assert properties about all possible behaviors of a system modeled in SMV.

Examples of properties that can be expressed in CTL are:

– **AG**$\neg(g_1 \wedge g_2)$ which stands for g_1 and g_2 are never true simultaneously, i.e. mutual exclusion.
– **AG**$(r \Rightarrow \mathbf{AF}g)$ every r is eventually followed by a g, i.e. freedom of starvation.

2.3 SMV: the language

A SMV description consists of a hierarchy of interconnected modules, the top-most module is named `main`. Figure 1 presents an example of SMV description. As the described systems shall be finite, all data types are also finite (enumerated and bounded integer types).

Figure 1 SMV description of 2 cascated inverters.

```
1    MODULE main
2    VAR
3        gate1 : inverter(gate2.output);
4        gate2 : inverter(gate1.output);
5    SPEC
6        (AG AF gate1.output) & (AG AF !gate1.output)
7    MODULE inverter(input)
8    VAR
9        output : boolean;
10   ASSIGN
11       init(output) := 0;
12       next(output) := !input;
```

Each module may contain several of the following components:

– The interface lists the formal parameters of the module. The formal parameters are instanciated with the actual parameters in the body.
– The local state variables follow keyword `VAR` and can be of type boolean, enumerated, integer interval, a module, or an array of these.
– The keyword `ASSIGN` starts the definition of initial and next-state value for the local state variables, respectively denoted `init` and `next`.

– Abbreviations for expressions can be introduced via keyword `DEFINE`.
– Temporal logic specifications follow keyword `SPEC`. See Section 2.2 for further details on the specification logic.

An interesting feature of SMV (line 5 in Figure 2) is the possibility to assign nondeterministically a value from a set of values listed between braces. Nondeterminism is both a very useful tool at the early stages of the modeling activity, and a powerful method to build simplified models of components. However there is no equivalent construct in VHDL.

Figure 2 A piece of SMV code with a nondeterministic expression.

```
1    VAR
2        request : boolean;
3        state : ready,busy;
4    ASSIGN
5        next(state) := {ready,busy};
```

3 VHDL

VHDL and Verilog HDL are the two major textual hardware design languages today and are still evolving to accomodate demands of their user bases. Numerous commercial CAD environments are based on at least one of them. VHDL is used mostly for simulation and synthesis. It is very versatile and allows a mix of different description styles, usually classified as behavioral, data flow and structural descriptions.

A VHDL description of a system is generally composed by entities and architectures. An entity (Figure 3, lines 1-5) describes the interface between the chip and the external world. This interface is defined by a clock signal (Figure 3, lines 2) and a set of input and output signals. The architecture (Figure 3, lines 6-15) contains one or more processes that describes the internal behaviour of the chip. This behaviour is achieved by a set of statements in the VHDL language and defines how the output values and other internal signals are calculated from the input and internal signals. VHDL has the ability to use entities/architectures as components of other systems and to define a library of components for future reference. To achieve this, a component declaration (Figure 4, lines 2-6) , configuration (Figure 5, lines 12-15) and instantiation (Figure 5, lines 8-9) is necessary.

VHDL semantics was originally defined in function of a simulation engine, called the kernel. The scope of the language is general and cannot be straightforwardly synthesized to hardware. Consequently, synthesis tools put important syntactic and semantic restrictions on the VHDL descriptions they handle. These restrictions have been unified into a draft standard for VHDL register transfer level (VHDL-RTL) synthesis, known as IEEE 1076.6, and currently under process of approval [IEE98]. Since the main goal of our work is the production of an actual chip, we shall use a VHDL subset that is compatible with these restrictions, providing a totally automated pathway from SMV to actual hardware.

3.1 VHDL-RTL

VHDL-RTL[IEE98] is a subset of VHDL defined by a group of industry and academic experts that aims at defining a greatest common denominators to most, if not all, commercial synthesis tools. The definition of this subset is now achieved, and its approval in a standardization body well advanced. Register transfer level synthesis tools have each their own strengths and limitations and each of them handles different VHDL subsets. Consequently, tool interoperability is limited and team work efficiency suffers from these incompatibilities.

[IEE98] defines how hardware elements, such as combinational logic, or edge-sensitive and level-sensitive sequential logic shall be described in VHDL. Moreover guidelines for verification of both combinational and sequential

elements are given and a precise syntax is included too. Since one of the aim of the VHDL-RTL standardization process was to define a subset common to the most popular tools, the result has been a lowest common denominator and lacks important features of the VHDL language, such as component support.

All these restrictions are addressed in the translation process to make the generated VHDL code compatible with most synthesis tools.

4 Correspondence between SMV and VHDL-RTL

The VHDL data flow style has a lot of similarities with SMV, even though it lacks some abstract features such as nondeterminism. In SMV, a description basically consists of variables which are assigned expressions that depends on other variables, which is about the same that is done in VHDL data flow. However, in SMV, it is implicit that all transitions are taken synchronously within a module, whereas this is not the case in VHDL. For instance, when describing at the register transfer level, one shall explicitly declare a clock signal, and all registers shall fetch their data on an edge of this signal.

In the remainder of this section, we shall detail how were handled the concepts of modularity (Section 4.1), and we then present how the different parts of a SMV module description may be translated: parameters and variables declarations in Section 4.2, assignments and macro definitions in Section 4.3, and we discuss how nondeterminism is handled in Section 4.4. The specification part (line 6 in Figure 1) shall not be translated since the temporal logic specification is not used in the register transfer level synthesis.

4.1 Modules and components

The module structure of SMV is hierarchical. The root module usually controls and interconnects other modules that describe the different components of a system. A systematic translation of such hierarchy into a VHDL component hierarchy is possible and we outline a strategy based on the SMV example of Figure 1. Each SMV module is translated into an entity-architecture pair and a component declaration. For instance, the translation of the module *inverter* of the example is given in Figure 3. Figure 4 contains the component declaration corresponding to our example. This approach makes each SMV module a potential instantiable component in VHDL and gives flexibitity to the user.

Figure 3 VHDL equivalent for a SMV leaf module.

```
1    entity inverter is
2        port (clock: in STD_ULOGIC;
3              input: in STD_ULOGIC;
4              output: out STD_ULOGIC);
5    end inverter;
6    architecture arch_inverter of inverter is
7    begin
8        process
9            variable var_output: STD_ULOGIC;
10       begin
11           wait until clock = '1';
12           var_output := not input;
13           output <= var_output;
14       end process;
15   end arch_inverter;
```

Figure 4 VHDL package with component declarations.

```
1    package declarations is
2        component inverter_comp
3            port (clock: in STD_ULOGIC;
4                  input: in STD_ULOGIC;
5                  output: out STD_ULOGIC);
6        end component;
7    end declarations;
```

When a module instantiates other modules, the corresponding VHDL entity and architecture shall make use of the components previously created for the other modules, as shown in line 8 of Figure 5. However, a configuration declaration is necessary to bind the components to their corresponding units (line 12 in Figure 5) and the VHDL-RTL subset does not accept component configuration.

Figure 5 VHDL equivalent for SMV main module.

```
1    entity main is
2        port (clock: in STD_ULOGIC;
3              outsig : out STD_ULOGIC);
4    end main;
5    architecture arch_main of main is
6        signal s1, s2: STD_ULOGIC;
7    begin
8        gate1:inverter port map (clock=>clock, input=>s2, output=>s1);
9        gate2:inverter port map (clock=>clock, input=>s1, output=>s2);
10       outsig <= s1;
11   end arch_main;
12   configuration conf_main of WORK.main(arch_main) is
13       for all: inverter use entity WORK.inverter(arch_inverter);
14       end for;
15   end conf_main;
```

Our solution is to copy the entire body of the component being instantiated instead of making the instantiation itself. It may look a mindless solution, but it is the only possible solution without the use of component configuration and that is certainly not a problem since the user will have the option to create two version of the same code, a VHDL and a VHDL-RTL compatible one, thus making it possible for the user to choose which parts he wants to synthesize to hardware.

4.2 Variables and signals

In SMV, data carrier objects are variables whereas, in VHDL, they can be either signals, and allow interprocess communication, or variable and are restricted to local process access.

SMV has two variable declaration mechanisms:

- A formal parameter that can be instantiated with arbitrary expressions.
- A local variable that can be arbitrarily read and written inside and outside of its definition scope.

The signal declarations in VHDL are quite different. Firstly, signals cannot be referenced outside of their scope, and have a mode which indicates the direction of the data flow they carry. Signals of mode `out` can be assigned but their value cannot be read, signals of mode `in` can be read but they cannot be assigned.

In summary, when translating SMV variables to VHDL objects, the following rules are obeyed:

- A variable that is only readed in the scope of a given module (a parameter of the module) is translated to an input signal (mode `in`).
- A variable that receives assignments is translated to an internal variable, an output signal(mode `out`) and an assignment of the internal variable to the output signal.

SMV also has a concept of macros, or abbreviations. These macros are translated as VHDL signals, and the analysis of the corresponding mode is the same. The difference between SMV variables and macros, is that variables are storage elements, whereas macros are not.

4.3 Assignments

Once we have shown how to handle variables and macro declarations in VHDL, it is necessary to analyze assignments and macro definitions which are grouped under the section ASSIGN and DEFINE.

There are three types of assignments in SMV: an initial assignment (`init`), an next assignment (`next`) and a direct assigment that is not introduced by a keyword. The `init` statement is translated to VHDL as an initial value on the declaration of that variable. The other two assignments are embedded as signal assignments into process statements sensitive to the clock signal edges.

The source expressions used in macro definitions and variable assignments are usually trivial to traduce. Most SMV expression constructs have a direct VHDL counterpart. A notable expression is the case expression, which can be translated as sequence of cascaded `if` and `elsif` or as a conditional signal assignment statement. Another important exception is nondeterminism, which is discussed in the next section.

4.4 Nondeterminism

SMV input language has two forms of nondeterminism. The first form is introduced with the keyword `process`. When a SMV module instantiation is in the `process` mode, their execution is asynchronous. That is, at each step, exactly one module is scheduled nondeterministically, which may incur starvation for some of the module instantiations. SMV makes it possible to state a fairness constraint that eliminates this type of starvation. This first type of nondeterminism is mainly used to model at the system level (another possible use is in the description of asynchronous circuits). However VHDL has no similar constructs, most probably because it has not been designed to describe at the system level of abstraction. We have identified two ways to resolve this discrepancy. The first solution is to code the nondeterminism in VHDL, introducing auxiliary signals and a scheduler process, and the algorithm this process shall execute remains an open issue. The second possible approach, which we have temporarily adopted, is not handling descriptions that make use of this type of nondeterminism.

The second form of nondeterminism occurs in expressions (see line 5 in Figure 2). Syntactically, such expressions consist of a list of subexpressions and semantically, the value is the value of one of these subexpressions, randomly chosen. This type of nondeterminism may be used when some implementation decision has not yet been made. VHDL has not this type of expressions, and we have identified several approaches to resolve this translation problem. Suppose that a SMV variable is assigned nondeterministically one of n values: $v_1, \ldots, v_n$. We propose the following 3 solutions to the translation problem:

1. The translator choses one of the values v_i, once for all. Note that this approach cuts off parts of the execution tree of the Kripke structure. Consequently, liveness properties true of the SMV description may turn false in the produced VHDL description. This phenomenon is common to most refinement techniques that usually only preserve universal properties [GL94].

2. The user choses a value, also once for all. As previously, some properties may turn false in the produced VHDL description.
3. The only possible way to model nondeterminism in VHDL are inputs [Cho74]. One possible approach is to introduce a signal of mode `in` that can take n different values to the interface of the VHDL description, and to translate the nondeterministic expression as a conditional expression, depending on the value of the introduced signal.

4.5 Preliminary results

We defined a complete set of translation rules from SMV to VHDL-RTL. We used this rules to translate manually several small examples available in the SMV distribution (a bit counter, a semaphore and the bit inverters shown in this paper) and thus validated our set of translation rules. A prototype tool has been implemented, based on SMV own front-end scanner and parser, which makes it easy to integrate both the verifier and the translation tools. The VHDL output produced by the translator for a small SMV description has been synthesized using a commercial tool [Max00].

5 Conclusion and future work

We defined a methodology to translate a synchronous subset of SMV, a language suitable for integrated circuit design and optimized for formal verification, into a subset of VHDL, known as VHDL register transfer level, an emerging standard for circuit synthesis. We showed that this methodology could be automated into a prototype tool. We claim that this is the first step towards the definition of a circuit automated design flow that would start with the definition of the model in a language that is suitable for formal verification, such as SMV. Once the designer has proved that his initial model satisfies the requirement properties, a HDL description of the system could be automatically, or semi-automatically, produced. This HDL description would then be fed as input to further stages of the circuit design, using available synthesis technologies.

A prototype tool implementing the ideas presented in this paper has been implemented. Based on this prototype, we plan to further validate our approach based on larger examples and case studies from the industry. The SMV description of a circuit design will be translated to VHDL-RTL and ultimately synthesized to a Field Programmable Gate Array (FPGA) using a set of commercial tools.

References

[BCMD90] J.R. Burch, E.M. Clarke, K.L. Mc Millan, and D.L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference, DAC'90*, 1990.

[BCS95] R.K. Brayton, E.M. Clarke, and P.A. Subrahmanyam, editors. *Formal Methods in System Design*, volume 7, number 1/2. Kluwer, August 1995. Special Issue on VHDL Semantics – Guest Editor: D. Borrione.

[Ber93] R. Bergamaschi. High-level synthesis in a production environment: methodology and algorithms. In J. Mermet, editor, *Fundamentals and Standards in Hardware Description Language*. Kluwer, 1993.

[Bry86] R.E. Bryant. Graph-based algorithm for boolean function manipulation. *IEEE Transactions on Computers*, C(35):1035–1044, 1986.

[CE81] E.M. Clarke and E. A. Emerson. Synthesis of synchronization skeleletons for branching time temporal logic. In *Logics of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heigths, NY, May 1981. Springer Verlag, New York.

[Cho74] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

[CW96] E. Clarke and J. Wing. Formal methods: state of the art and future directions, 1996.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[IEE93] IEEE. *IEEE Standard VHDL Language Reference Manual*, 1993. Std 1076-1993.

[IEE98] IEEE. *IEEE P1076.6/D2.0 — Draft standard for VHDL register transfer level synthesis*, 1998.

[Max00] Max+plus ii. Altera Corporation, 2000. `www.altera.com`.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.