Constrained Symbolic Simulation with Mathematica and ACL2

Ghiath Al Sammane, Diana Toma, Julien Schmaltz, Pierre Ostier, and Dominique Borrione

TIMA Laboratory, VDS Group, Grenoble, France
http://tima.imag.fr

Abstract. We use symbolic simulation for the verification of high level circuit specifications. We combine Mathematica for algebraic computation and ACL2 for branching decision to increase the efficiency of the method.

1 Introduction

Symbolic simulation, proposed as early as 79 by J.Darringer, is intermediate between conventional simulation and mathematical reasoning, to verify abstract, pre-RTL design specifications. Instead of simulating a design with numerical values, symbolic inputs are given to the symbolic simulator, which produces an algebraic expression for the memory and output variables, as a function of the initial state and of the inputs. These difficulties arise: (1) the symbolic expressions may become exponentially large in the number of simulation cycles; (2) in the presence of conditional statements, when the condition is a symbolic term, all alternative paths must be explored. The simulator generates a simulation tree, which may also grow exponentially; (3) the automatic simplification and reduction of the computed symbolic expressions is needed, else the outputs of symbolic simulation are unreadable.

Previous works have tackled one or more of the above difficulties: e.g. GSTE [9] at switch and gate-level, PVS [7] and ACL2 [4] at the initial abstract design levels. To simplify symbolic simulation by reducing algebraic expressions and controlling the expansion of the simulation tree, most proposed solutions use an automated reasoning tool.

A systematic approach for using ACL2 as a symbolic simulation engine was proposed by J. Moore [6]. On this base, the semantics of a subset of VHDL [3] were defined in ACL2 in order to simulate a VHDL design symbolically [2]. In this paper we propose a different approach based on the separation of *algebraic computation* and *branching decision*. We combine Mathematica [8] a *computer algebra system* and ACL2 [4] an *automatic theorem prover* to perform what we call *constrained symbolic simulation*. This association increases the efficiency of the symbolic simulation by using two tools, each one being powerful in its domain.

D. Geist and E. Tronci (Eds.): CHARME 2003, LNCS 2860, pp. 150–157, 2003.

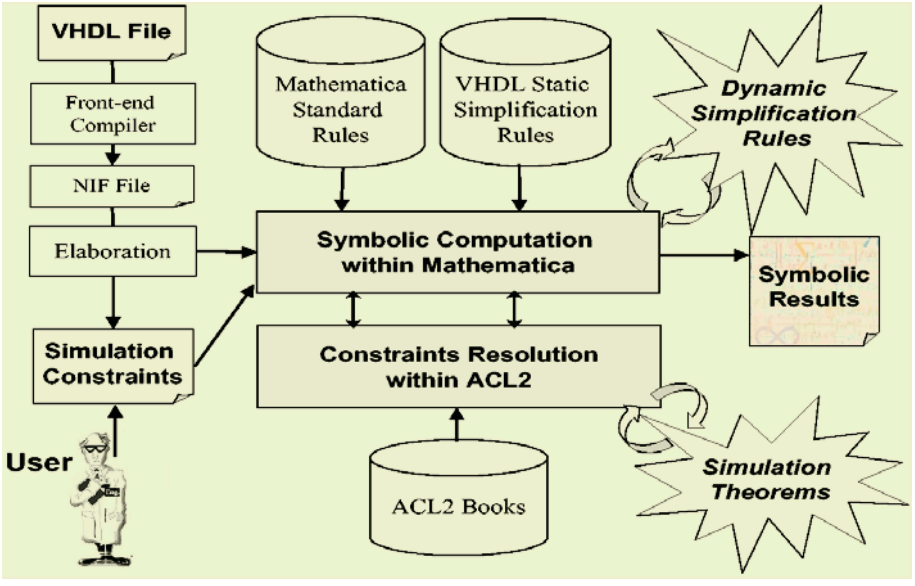

Fig. 1. Overview of the method

2 Overview of the Method

Figure 1 shows the overall combined verification system taking VHDL inputs. The front-end compiler performs syntactic and static semantics checks, and serves as common starting point to all EDA tools. NIF is an intermediate format developed by our group. The elaboration of the Mathematica model, called M-Code, is performed on the NIF file. During this step, data type restrictions are extracted as constraints. Before starting the simulation, the user, who is not necessary a proof expert, can add constraints on the inputs. Those are inequalities or equalities between expressions composed of design variables or input signals and arithmetic operators $(+, -, /, \times)$. M-code and constraints are submitted to Mathematica for n simulation cycles, n is user defined. During simulation, symbolic expressions are simplified using rewrite rules. Standard Mathematica simplification rules are algebraic axioms like $(x - x \longrightarrow 0)$ and arithmetic simplifications like $(n+n \longrightarrow 2n)$, for terms defined on real or integer types. VHDL simplification rules were defined by us for the hardware types unknown to Mathematica (e.g. Bit). To reduce the simulation tree, whenever path conditions are encountered, ACL2 is called as a reasoning engine. ACL2 evaluates a given condition under simulation constraints using pre-proved theorems. Depending on the ACL2 answer, Mathematica chooses a path. After each simulation cycle, the values of all variables and signals are stored in a file. This is the result of the *constrained symbolic simulation* of the VHDL description.

Table 1. Example of stabilizing concurrent assignments

cycle	VHDL expressions
1	a <=(d and not(c)) or (b and c); b <=(a and not(c)) or (d and c);
2	a <=(d and not(c)) or ((a and not(c)) or (d and c) and c); b <=((d and not(c)) or (b and c) and not(c)) or (d and c);
3	a <= d; b <= d;

3 Modeling VHDL in Mathematica

The VHDL supported by our tool is based on the standard subset for Register Transfer Level (RTL) synthesis [3], enlarged with full arithmetic types. Combinational logic and clock-edge synchronized sequential logic may be described using a behavioral, structural or dataflow style, or any combination thereof. A model is a component, i.e. an `entity` coupled with its associated `architecture`.

Due to the absence of explicit time [3], the simulation algorithm is simplified, as described in [2]: the driver of a signal only holds one current and one next value, since right hand side waveforms are a single zero delay expression (the `after` clause is not recognized in the subset). Concurrent signal assignments and combinational processes are stabilized by performing delta computation cycles between each two clock simulation cycles. In this context, the model is observable only at the clock cycle level.

In the M-code, a VHDL component built from a (entity *Ent*, architecture *A*) pair is modeled by a Mathematica function named: *EntA*. Its arguments are all the objects declared in the corresponding entity-architecture: input, output and local signals, and local variables. All are named Mathematica `blank patterns`, i.e. no data type is defined. However, the information about data types is not lost: it will serve as simulation constraints.

Two Mathematica variables are necessary to model each local or output signal: one for the current value, passed to *EntA* as argument; one for the next value, declared as temporary variable inside the body of *EntA*. Input signals, that cannot be modified in the architecture, only have a current value.

The body of *EntA* is the Mathematica model for the VHDL statements inside the architecture. All processes are flattened inside the body. To eliminate simulation delta cycles, we perform a symbolic fixed point computation during M-code generation. We repeat the execution, symbolically and sequentially, of all concurrent signal assignments, and simplify the expressions, until they stabilize. The next values of all signals can then be computed in one step.

Table 1 displays the three cycles needed to stabilize the symbolic value for the concurrent assignments shown at cycle 1. In the M-code symbolic delta cycles are no more needed. The corresponding M-code for this example is:

```
NextSig[a,d];
NextSig[b,d];
```

Table 2. Examples of M-code assignment functions

VHDL	M-code
A <= d + g;	NextSig[Anext, Plus[d, g]];
V := 2 + j;	ChangeVar[V, Plus[2, j]];
Q := V + 1;	ChangeVar[Q, Plus[V, 1]];

Table 3. Syntax of VHDL branching statements in M-code

VHDL	M-code
If B then state-bloc-1 else state-bloc-2 end if	If [B, state-bloc1 ,state-bloc-2 ,decideACL2]
For I in start to end loop Statements End loop;	For [Set[I,start], Equal[I,end], Incr[I] decideACL2, Statements] (*Comment: B = Equal[I,end] *)

At this stage, the body of $EntA$ contains only sequential statements: assignments, conditionals or instantiations of components. Each one of them is represented by a function in Mathematica syntax.

An assignment is modeled by $NextSig$ for signals and $ChangeVar$ for variables (Table 2). $NextSig$ assigns the next value of the signal while $ChangeVar$ assigns the variable directly. $NextSig[Sig, terms]$ or $ChangeVar[Var, terms]$ also create rewrite rules [5] that transform Sig or Var to $terms$. These rules are not applied during M-code generation, but during simulation.

Branching statements are modeled by functions in which their semantics consider a three state logic (Table 3). When B is a symbolic formula that cannot be evaluated to true or false by Mathematica, ACL2 is called to decide B under constraints. Details about the decision procedure are discussed in the next section.

4 Simulation Algorithm

First, all objects are initialized with their values according to their VHDL declaration. The consistency of simulation constraints is verified by ACL2. After that, the M-code function is executed `NbCYCLE` times (`NbCYCLE` is user defined).

At each simulation cycle, the function *Test-vectors* can be customized to generate specific inputs; for instance, reset signals can be active in the first simulation cycle, inactive otherwise. Then, the $EntA$ function is interpreted in Mathematica, where two operations are performed: simplification of terms and branch decision. At the end of each cycle an execution tree is generated, which contains all symbolic values for each signal and variable in the design.

4.1 Computation of Terms

When assignment functions $NextSig[Sig, terms]$ or $ChangeVar[Var, terms]$ are encountered, right hand side $terms$ are simplified into $terms\prime$, using standard

```
Initialize(Sin,Sout,Slocal,Vlocal)
Verify-by-acl2(Constraints)
For cycle:=1 to NbCYCLE do
   Test-vectors(Reset,cycle,Sin)
   EntA(Reset,Sin,Sout,Slocal,Vlocal)
   Print-Tree(Sin,Sout,Slocal,Vlocal)
End for;
```

Fig. 2. Simulation algorithm

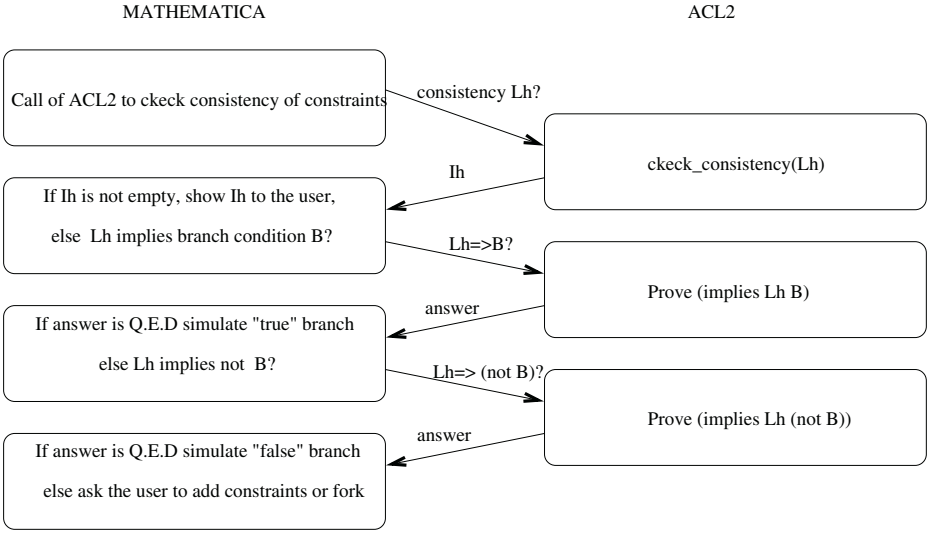

Fig. 3. Branch decision scheme

Mathematica and static VHDL rules. Then, the left hand side Sig or Var is assigned with $terms\prime$ and the rewriting rule $Sig \longrightarrow terms\prime$ or $Var \longrightarrow terms\prime$ is added to a library called dynamic VHDL simplification rules. Those rules are now available to simplify all successive assignments. This on the fly simplification of terms is essential for time and memory efficiency.

In Table 2, $ChangeVar[V, Plus[2, j]]$ assigns V with $Plus[2, j]$ and creates the rewrite rule $V \longrightarrow 2 + j$. In the next assignment $(V + 1)$ is simplified using $(V \longrightarrow 2 + j)$. Then, Q is assigned with $3 + j$. Finally, the rewrite rule $(Q \longrightarrow 3 + j)$ is created.

4.2 Branch Decision

During simulation, Mathematica, whenever it cannot decide a branch condition, calls ACL2. Figure 3 shows the principle of their interaction.

First, Mathematica asks ACL2 to check the consistency of the set of simulation constraints L_h. Function *check_consistency* takes L_h as input and returns a

minimal set of contradictory hypothesis I_h, or the empty set. If I_h is not empty, the simulation is stopped and the contradiction is shown to the user.

If I_h is empty, Mathematica sends $L_h \Rightarrow B$ to ACL2. If ACL2 finds a proof, it returns $Q.E.D$; the "true" branch is considered for simulation. If ACL2 fails or is not able to find a proof in a given time, it returns *Failed*. In this case, Mathematica sends $L_h \Rightarrow \neg B$. If it succeeds, the "false" branch is considered for simulation. Otherwise, the simulation stops and the user is asked for more constraints. If more constraints are given, simulation is reinitialized. Otherwise, the symbolic simulation forks into two branches, one assuming the branch condition is true and the other its negation.

Branch decision is generally not decidable. However, most cases are limited to equalities and inequalities formulae, and resolved by using some pre-proved theorems on them (written as ACL2 books). At each cycle the proved theorems are added to the ACL2 database and they are available for the future proofs.

Example Euclid's GCD algorithm (Table 4):

Table 4. Euclid's GCD algorithm

VHDL	M-code

P1: process begin
 	
wait until clk='1';
if RST='1' then
a0:=a;
b0:=b;
ok<=False;
elsif a0=b0 then
ok<=True;
res<=a0;
elsif a0>b0 then
a0:=a0-b0;
else b0:=b0-a0;
end if;
end process P1;
	GCDmath[CLK_,RST_,a_,b_,OK_,
	res_,a0_,b0_,c0_]:=
	Module[,
	If[RST==1,
	ChangeVar[a0,a];
	ChangeVar[b0,b];
	NextSig[OK,False]
	,If[Equal[a0,b0]
	,NextSig[OK,True];
	NextSig[res,a0]
	,If[a0>b0
	,ChangeVar[a0,a0-b0]
	,ChangeVar[b0,b0-a0]
	,decideACL2]
	,decideACL2]
	,decideACL2]]

Before beginning the simulation, the function *Test_vectors* has been customized to generate an active reset at the first simulation cycle and inactive hereafter. The initial values are $a = 3n$ and $b = n$ and the constraints are $L_h = \{n \in \mathcal{N}^*\}$. The simulation of four cycles runs as follows.

At Cycle1, RST has the numeric value 1 and a_0 and b_0 are assigned with initial values $3n$ and n. In all subsequent cycles, RST is set to 0 and Mathematica will always decide to simulate the "false" branch of the first $if-then-else$ statement. We do not mention it anymore. At Cycle2, Mathematica cannot decide if a_0 is equal to b_0, i.e. if $3n$ is equal to n. So, it calls $decideACL2$, which works as shown on Figure 3. The constraint $\{n \in \mathcal{N}^*\}$ is transformed into the ACL2

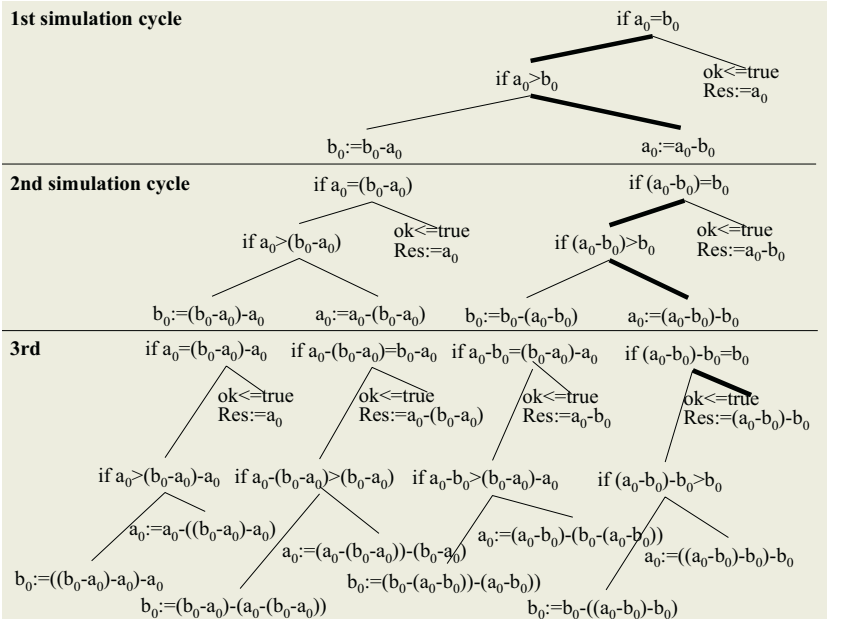

Fig. 4. Execution tree of the GCD example

list $((integerp\ n)\ (< 0\ n))$ and its consistency is checked. As ACL2 returns an empty list of contradictions I_h, Mathematica sends the following "defthm event" to ACL2 :

```
(defthm branch-1
   (implies (and (integerp n) (< 0 n))
            (equal (* 3 n) n)))
```

Because the ACL2 answer is "Failed", Mathematica sends the event :

```
(defthm branch-1-negation
   (implies (and (integerp n) (< 0 n))
            (not (equal (* 3 n) n))))
```

ACL2 answers "Q.E.D", Mathematica considers the "false" branch for simulation and simplifies $a_0 - b_0$ to $2n$. The reader may be surprised by the simplicity of the theorems, but without ACL2 Mathematica is not able to prove them. At Cycle3, a_0 is simplified to n and at Cycle4 ACL2 answers "Q.E.D" to the event:

```
(defthm branch-4
   (implies (and (integerp n) (< 0 n))
            (equal n n)))
```

As four cycles have been simulated, the simulation is stopped. Figure 4 shows the execution tree without any constraints. With constraints, only the bold path is simulated (reset has been omitted).

5 Discussion and Conclusions

Our prototype system for *Constrained Symbolic Simulation* takes advantage of the best qualities of two powerful automatic systems: Mathematica to simplify algebraic expressions, and ACL2 to decide the truth value of expressions under a set of hypotheses. Clock synchronized sequential circuits and delay-free combinational circuits, written in a synthesizable VHDL subset, are automatically translated into a M-code file, its simulation model.

The automatic generation of proof obligations for ACL2, under the form of "defthm events" is implemented. Mathematica and ACL2 are executed as concurrent processes, and communicate via a pipeline. Our technique efficiently prunes the execution tree, and proves VHDL assert statements [1] on small circuit blocks; we are working on bigger systems, like the AMBA architecture. We intend to extend our method to more abstract specifications, as describable in the next version of the VHDL subset for system-level synthesis, or SystemC.

References

1. Al Sammane G., Toma D., Schmaltz J., Ostier P., Borrione, D.: Constrained Symbolic Simulation with Mathematica and ACL2. ISRN TIMA-RR–03/07-03–FR, $http://tima.imag.fr/publications/files/rr/css_177.pdf$
2. Borrione, D., Georgelin, P., Moraes Rodrigues, V.: Symbolic Simulation and Verification of VHDL with ACL2. In: Ashenden, P.J., Mermet, J.P., Seepold, R. (eds.): System-on-chip Methodologies and Design Languages. Kluwer, 2001, 59–70
3. 1076.6-1999 IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis. IEEE, 1999
4. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided reasoning : ACL2 An approach. Kluwer Academic Press, 2000
5. Maeder, R.: Term Rewriting and Programming Paradigms. In: Keränen, V.(ed): Mathematics with a Vision: Proceedings of the First International Mathematica Symposium (1995). Computational Mechanics Publications
6. Moore, J.S.: Symbolic Simulation: An ACL2 Approach. In: Formal Methods in Computer-Aided Design (FMCAD '98) (1998), 334–350
7. Shankar, N., Owre S., Rushby J.M., Stringer-Calvert D.W.J.: PVS Prover Guide Computer Science Laboratory, SRI International (1999) Menlo Park, CA, USA
8. Wolfram, S.: The Mathematica Book. Cambridge University Press and Wolfram Research (2000). 100 Trade Center Drive, Champaign, IL 61820-7237, USA
9. Yang, J., Seger, C.I.: Generalized Symbolic Trajectory Evaluation Abstraction in Action. In: Formal Methods in Computer-Aided Design (FMCAD'02) (2002). Springer, LNCS 2517, Portland, Or, USA, 70–87