Combining ACL2 and Mathematica for the Symbolic Simulation of Digital Systems

AL SAMMANE Ghiath BORRIONE Dominique OSTIER Pierre
SCHMALTZ Julien TOMA Diana

Abstract

We combine Mathematica, a computer algebra system, and ACL2 to perform what we call constrained symbolic simulation. This association increases the efficiency of the symbolic simulation by using the automated reasoning capabilities of ACL2 and the powerful symbolic computation of Mathematica. The communication between the two systems is automated.

1 Introduction

The *constrained symbolic simulation* relies on the separation of *algebraic computation* and *branching decision*. It uses the efficiency of Mathematica [Wol00] at reducing symbolic expressions and the ACL2 reasoning capabilites to decide conditional expressions under constraints.

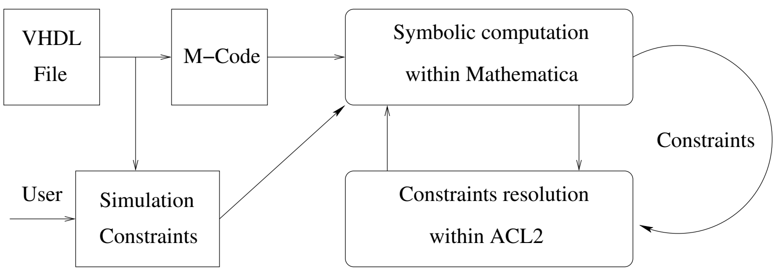

Figure 1: Overview of the method

Figure 1 shows the overall combined verification system for the VHDL standard. The M-Code is the translation of the VHDL description file in the Mathematica syntax. 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. In our system, due to some restrictions on the VHDL, a simulation cycle is identical to a clock cycle. Indeed, we consider sequential processes synchronized on a single clock and concurrent instructions which are stabilized before simulation using, the Mathematica fixed point feature. During simulation, symbolic expressions are simplified using Mathematica rewrite rules, and whenever path conditions cannot be resolved by Mathematica, ACL2 is called. 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. This document concentrates on the use of ACL2 to prune the execution tree. Details on the simplification of the symbolic expressions by Mathematica are given in [STSB03].

2 ACL2 as a reasoning engine

2.1 ACL2 and Mathematica communication

Mathematica calls ACL2 through the function *callAcl2["string"]*. It sends *string* to ACL2 via a pipe and gets back the last line of the ACL2 response.

```
callAcl2["(defthm foo (equal x x) :rule-classes nil)"]
FOO
callAcl2["(defthm foo (not (equal x x)) :rule-classes nil)"]
******** FAILED ******** See :DOC failure ******** FAILED ********
```

Our tool in Mathematica uses ACL2 as shown on Figure 2. First, Mathematica asks ACL2 to check the consistency of the set of simulation constraints L_h. This is done by a function *check_consistency* that takes L_h as input and returns a minimal set I_h of unsatisfiable constraints [1] or t if L_h is consistent. This is done by *callAcl2["(check-consistency L state)"]* (see next subsection).

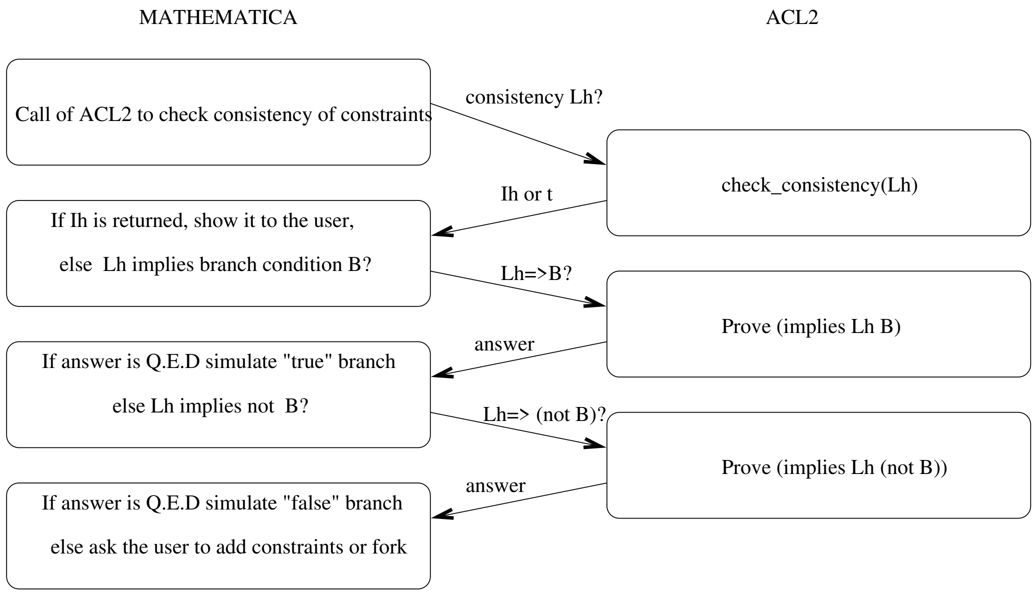

Figure 2: Branch decision scheme

If I_h is not empty, the simulation is stopped and the contradiction is shown to the user. If L_h is consistent, Mathematica sends $L_h \Rightarrow B$ to ACL2.

```
callAcl2[‘‘(mv-let (erp val state)
                   (defthm foo (implies Lh B))
                   (declare (ignore val))
                   (if erp
                       (value nil)
                       (value T)))’’]
```

If ACL2 finds a proof, the "true" branch is considered for simulation. If ACL2 fails or is not able to find a proof in a given time, Mathematica sends $L_h \Rightarrow \neg B$. If it succeeds, the "false" branch is considered for simulation. Otherwise, the simulation is stopped and the user is asked to add more constraints. If more

[1] minimal in the sense that every strict subset of constraints is satisfiable

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.

Note that ACL2 automatically resolves most of the equalities and inequalities formulae of branch decisions by using some pre-proven theorems on them (written as ACL2 books). At each cycle the proven theorems are added to the ACL2 database and they are available for the future proofs.

2.2 Checking constraints consistency

The function *check_consistency* uses some functions of the book *books/misc/expander* of the ACL2 distribution, particularly *tool1-fn*, which tries to simplify a list of hypotheses. This function can decide if a list of hypotheses is consistent, *check_consistency* exhibits a set of contradictory hypotheses I_h. Function *check_consistency(L state)* returns *nil* in case of errors, *t* if L is consistent, else it calls *consistency(L nil 1 state)*.

```
(defun check-consistency (L state)
   (if (true-listp L)
       (cond ((endp L) (value nil))
             (t (mv-let (erp val state)
                  (tool1-fn L state nil t nil t t)
                  (if erp
                     (value nil)
                     (if (nth 1 val)
                         (value t) ; L contains no contradictions
                         (consistency L nil 1 state))))))
       (value nil)))
```

Function *consistency* takes as inputs an inconsistent set of constraints L, the initial values of I_h, an index i, and the ACL2 state. It returns *nil* in case of errors, else it returns the list of the contradictory hypotheses I_h of L.

```
(defun consistency (L Ih i state)
     (if (and (true-listp L)
              (true-listp Ih)
              (integerp i)
              (< 0 i))
         (cond ((endp L) ; last step of the algorithm
                 (value Ih)); now Ih is the minimal set
                ((< (length L) i) (value nil)) ;error:  i out of L range
                ((endp Ih) ; first step(s) of the algorithm (at call Ih is empty)
                  (mv-let (erp val state)
                    (tool1-fn (subseq L 0 i) state nil t nil t t)
                    (if erp
                       (value nil) ; tool1-fn error case
                       (if (nth 1 val) ; is either a list of consistent constraints or nil
                           (consistency L Ih (+ i 1) state)
          ; if no contradictions in L[0 ..  i], proceed with L[0 ..  i+1]
          ; else the added constraint is removed from L and added to Ih
                           (consistency (remove (nth (- i 1) L) L)
                                        (cons (nth (- i 1) L) Ih) 1 state)))))
               (t (mv-let (erp val state) ; one step of the algorithm
                   (tool1-fn Ih state nil t nil t t)
                   (if erp
                      (value nil) ; tool1-fn error case
                      (if (nth 1 val)
                          (mv-let (erp1 val1 state)
                             (tool1-fn (append Ih (subseq L 0 i))
                                       state nil t nil t t)
```

```
(29)         ; check of the consistency of the union of Ih and L[0 ..  i]
(30)                          (if erp1
(31)                              (value nil) ; tool1-fn error case
(32)                              (if (nth 1 val1)
(33)                                  (consistency L Ih (+ i 1) state)
(34)                                  (consistency (remove (nth (- i 1) L) L)
(35)                                               (cons (nth (- i 1) L) Ih) 1 state))))
(36)                          (value Ih))))))
(37)       (value nil)))
```

Lines 2 to 5 insure the type of the inputs. On lines 6 and 7, L is the empty list and I_h is returned. If the index is greater than the length of L, *nil* is returned (8). When I_h is empty (9 to 16), if L[0 .. i] is consistent, we proceed with L[0 .. i+1], else the constraint L[i] is removed from L and added to I_h. The algorithm restarts with $i = 1$. When neither L nor I_h are the empty list (21 to 36), if I_h is not consistent, the algorithm terminates and I_h is returned; else, it proceeds in a similar way to the third case of the "cond" (9 to 16), except that *tool1-fn* is called on the concatenation of I_h and L[0 .. i].

3 Applications

3.1 Reduction of the execution tree

Let us illustrate simulation tree reduction through a $VHDL$ process that implements Euclid's GCD algorithm:

```
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;
```

For instance, one wants to simulate this circuit for $a = 3n$ and $b = n$ under the constraint $L_h = \{n \in \mathcal{N}^*\}$ and for four cycles. At the first cycle, RST has the numeric value 1 and a_0 and b_0 are assigned with initial values $3n$ and n, respectively. In all the next cycles, RST is set to 0 and Mathematica will always treat the "false" branch of the first $if - then - else$ statement. At the next cycle, Mathematica cannot decide if a_0 is equal to b_0, i.e. if $3n$ is equal to n. So, it calls ACL2 :

```
callAcl2[``(mv-let (erp val state)
                   (defthm branch-1
                     (implies (and (integerp n) (< 0 n))
                     (equal (* 3 n) n)))
            (declare (ignore val)) ...'']
```

Because the answer is *"nil"*, Mathematica sends the negation of *branch-1*. As ACL2 answers *"t"*, 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 the third cycle, a_0 is simplified to n and at the fourth cycle ACL2 answers *"t"* to the event:

```
callAcl2[``(mv-let (erp val state)
                  (defthm branch-4
                    (implies (and (integerp n) (< 0 n))
                    (equal n n)))
            (declare (ignore val)) ...'']
```

As four cycles have been simulated, the simulation is stopped. If ACL2 were not used, eight paths would have been simulated instead of the single explained one.

3.2 Symbolic evaluation of assertions

The proposed combination can be used for the verification of digital systems by proving properties by symbolic evaluation of assertions. VHDL assert statements insure that a given condition *bool_expr* is never violated. If it happens, the "message" is printed and, depending on the severity level (*e.g.* error, warning), the simulation is stopped or not. These statements will be translated as If function calls in Mathematica.

VHDL assert statement	Mathematica if function

`Label1:assert bool_expr`
`report "message"`
`severity severitylevel;`
	`If[bool_expr`
	`,ChangeVar[Label1,True]`
	`,ChangeVar[Label1,False]`
	`,decideACL2]`

Label1 is translated to a variable, which is assigned by function ChangeVar with value *true* or *false* according to the truth value of *bool_expr*. It is possible that the truth value of the assertion remains under the form of a symbolic expression during one or more simulation cycles. If it evaluates to false at cycle C, the path from the root leading to the node in the simulation tree constitutes a counter-example for the assertion.

4 Conclusion

We have presented a new approach for the symbolic simulation of high level circuit specifications called *Constrained Symbolic Simulation.* This method makes use of typing information and user constraints to prune the execution tree. It is implemented on top of two powerful automatic systems, taking advantage of the best qualities of each one: Mathematica to simplify algebraic expressions, and ACL2 to decide the truth value of expressions under a set of hypotheses.

Up to now, we have applied our technique on small circuit blocks and we are working on bigger systems. Finally 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

[STSB03] G. AL Sammane, D. Toma, J. Schmaltz, and D. Borrione. Symbolic Simulation of Digital Circuits with an Automatic Theorem Prover and a Computer Algebra System. *Journal of Symbolic Computation*, 2003. (Submitted for publication).

[Wol00] S. Wolfram. *The Mathematica Book.* Cambridge University Press and Wolfram Research, 100 Trade Center Drive, Champaign, IL 61820-7237, USA, 2000. fourth ed.