Electronic Notes in Theoretical Computer Science 65 No. 5 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html 8 pages

Symbolic simulation of synchronous programs

David Garriou [1,2]

IRCCyN
Université de Nantes - Ecole Centrale de Nantes
1 rue de la Noe, BP 92101, 44321 NANTES cedex 3, France

Abstract

This paper proposes an intermediate approach between simulation and formal verification. This method called symbolic simulation allows to simulate a reactive program by giving it the set of its possible inputs. An important aspect of this work is the fact that we deal with values of unbounded numerical variables and take them into account while computing properties of control. This has been done for synchronous languages, (e.g. Lustre, Esterel and Signal), as these languages are known to provide precise mathematical description of critical embedded systems with real time constraints.

1 Introduction

The increasing complexity of softwares embedded in critical systems is a major challenge for programming methods, simulation and formal verification tools.

Among many approaches for programming embedded softwares, we are interested in synchronous languages [3,10,2], which propound clean mathematical semantics allowing formal proofs on programs.

These languages are now widely used in industrial designs, especially in critical fields such as avionics, nuclear plant control.

Validation of software is there a major issue. The most widely used approach in industrial designs is still simulation and test, which is, up to date, the only way to validate very large software (often more than 1000 inputs/outputs).

Formal verification using model-checking [6], had success in validating hardware or protocols, especially when control mainly depends on logical computations. Data structure such as BDDs [1,4] gives the algorithmic power to handle systems with hundred of boolean state variables.

[1] Thanks to anonymous referees who suggested ameliorations of this paper
[2] David.Garriou@irccyn.ec-nantes.fr

Much work have been done in the challenging area of taking into account values of numerical variables to check properties or to compute the reachable states space of a system [16,5,18]. The major drawback of these techniques is their computational cost, which forbids to apply them on "real world" large software. Recently some refinement techniques [12], based on abstract interpretation have been proposed to deal with numerical linear properties of wide synchronous reactive programs.

The purpose of our approach is to propose an intermediate analysis of reactive systems between simulation and formal verification. We try to offer the programmer the ability to choose the better compromise between efficiency and power. Roughly speaking, we want to give him several choices between the following extremal solutions: quickly simulating a system with only one particular input pattern, and trying to verify the whole system with all possible input patterns.

Our proposal is strongly related to what James C. King proposed in 1976 under the name of "Symbolic Execution" in an other context : analysis of imperative (PL1 like) programs. In the field of reactive programming, we are used to talk about "simulation" instead of "execution", but the underlying concepts are similar. The major difference is that we deal with infinite behaviors. We also use more recent works to solve logical and numerical constraints.

The paper is organized as follows : we first set the context of synchronous programs through an example. Then we describe and compare simulation and symbolic simulation trough the same example. Then we describe verification with symbolic simulation, and last we give some references to the technical issues underlaying symbolic simulation. We conclude giving some directions for future work.

2 Synchronous programs

Synchronous programs are reactive programs that react in zero-time to their environment. This hypothesis allows clean mathematical semantics for synchronous languages [8]. Synchronous languages allow different programming style - imperative with Esterel - declarative with Lustre and Signal, graphical with SyncCharts and Argos.

The underlying execution model for a synchronous program is an infinite loop program which at each step : scans its inputs variables or is waked up by input events, then computes next values for its states variables, and last outputs values to control its environment.

For example, the following Lustre [10] program controls a boiler watching its temperature **Temp**. It computes estimated values for internal state variables **Pressure** and **Volume**.

It controls two pumps (**In**, **Out**) that may be **Open** or not. The program emits diagnosis with signal **Ok** which is true if and only if one and only one pump is opened, and with signal **Alarm** emitted when both pumps are closed.

```
node controller(Temp:int)returns(OpenIn,OpenOut,Alarm,Ok:bool);
var Pressure, Volume : int;
let
  OpenOut = true -> if Temp>10
                  then (Pressure>5 or pre(OpenOut)) and Volume<20
                  else not pre(OpenIn);
  OpenIn = false -> if Temp>10 then (Volume>=20 or pre(OpenIn))
                                  and Pressure<=5
                  else not pre(OpenOut);
  Pressure = 7 -> if Temp <=10 then if pre(Pressure) < 1
                                  then pre(Pressure)
                                  else pre(Pressure)-1
                             else if pre(Volume) < 1
                                  then pre(Pressure)
                                  else pre(Pressure)+1;
  Volume = 10 -> if Temp <=10  then if pre(Pressure) < 1
                                  then pre(Volume)
                           else pre(Volume)+5
                             else if pre(Volume) < 1
                                  then pre(Volume)
                           else pre(Volume)-5;
  Ok = if OpenOut then not OpenIn else OpenIn;
  Alarm = not OpenOut and not OpenIn;
tel.
```

Each equation is an invariant statement that defines value of a state variable or output variable at each time step. A variable has in fact a discrete flow of values (one value at each time step). The "Pre" operator refers to the previous value of a variable.

To understand behavior of this program, we may look at its generated sequential code as it would be generated by LUSTRE compiler [17] :

```
program controller (Temp: int) returns (OpenIn,OpenOut,Alarm,Ok : bool;
                                        Pressure, Volume : int);
var x,y : bool
begin
 OpenOut := true;  OpenIn := false;
 Pressure := 7;    Volume := 10;
 while true do
    read(Temp);
    if temp <= 10 then if Pressure >= 1
                       then Pressure := Pressure-1; Volume := Volume+5;
                  else if Volume >= 1
                       then Pressure := Pressure+1; Volume := Volume-5;
    if Temp > 10 then x := Pressure > 5 or OpenOut and Volume < 20;
                     y := (Volume >= 20 or OpenIn) and Pressure <= 5;
                 else x := not OpenIn ;  y := not OpenOut;
    OpenIn := x;    OpenOut := y;
    Ok := if OpenOut then not OpenIn else OpenIn;
    Alarm := not OpenOut and not OpenIn;
    write(OpenIn,OpenOut,Alarm,Ok,Pressure, Volume);
 done;
end
```

For the purpose of simulation, we now consider this kind of programs : infinite loop programs reading their inputs and computing at each cycle next values for internal state variables and output variables.

We consider only boolean and unbounded integer variables. These programs may be modelled as interpreted input/output automata as used in [9].

Depending on what we want to do, programs may be viewed as :

- single state automata with boolean and integer variables,
- or as automata with states encoding boolean variables and with integer variables interpreted,
- or as infinite state systems with states encoding values of both kinds of variables.

3 From usual simulation to symbolic simulation

Simulating a reactive system usually means executing it for a valuation of its input variables at each cycle. Symbolic simulation consists in executing a system for a set of valuations of its input variables at each cycle.

3.1 Simulation

A simulation is a particular execution. In usual sense, the previous example could be simulated in the following way : at each time step, the program asks values for its input variables and writes values for output variables.

```
temp? 8
openin=false, openout=true, pressure=6, volume=15, alarm=false, ok=true
temp? 9
openin=false, openout=true, pressure=5, volume=20, alarm=false, ok=true
temp? 10
openin=false, openout=true, pressure=4, volume=25, alarm=false, ok=true
temp? 11
openin=false, openout=true, pressure=3, volume=30, alarm=false, ok=true
temp? 12
openin=true, openout=false, pressure=4, volume=25, alarm=false, ok=true
```

In order to test different behaviors, we have to execute many simulations. As we allow unbounded integer variables, simulation can't be exhaustive.

3.2 Symbolic execution

J.C.King defines symbolic execution of PL-1 like programs in [13]. It consists in executing a program whose values of input variables have been replaced by identifiers not belonging to the set of variables identifiers of the program. He calls such identifiers "input symbols".

Execution of such a program consists in propagating input symbols along an execution path, and building a relation upon the input symbols (by collecting instructions constraints). Building the relation is semi-automatic, because one must sometimes ask the user which path he should choose if the current instruction is a choice instruction.

We may symbolicaly execute the previous example giving it at each time a new input symbol for the variable **temp** :

```
temp? t1
openin=false, openout=true, pressure=if t1>10 then 8 else 6,
volume=if t1>10 then 5 else 15, alarm=false, ok=true
temp? t2
...
```

Results are given as formulas on input symbols. This approach has been introduced for finite sequential programs. For reactive programs, it's not realistic to introduce free variables at each time step. However it's a first way to deal with set of input valuations and it is really interesting for debugging reactive programs.

3.3 Symbolic simulation

We call symbolic simulation a simulation where :

- a set of input values is implicitely given by its charasteristic function,
- all behaviors allowed by some of these input valuations are simultaneously computed and are described by a characteristic function defining a relation on state and ouput variables.

For the previous example, we may assert the following relation about variable **temp** and simulate all behaviors. We obtain the following relations :

```
? temp >= 5 and temp <= 15
not openin and openout and 5*pressure+volume=45 and
volume<=15 and volume>=5 and ok and not alarm
? temp >= 5 and temp <= 15
not openin and openout and 5*pressure+volume=45 and
volume<=20 and volume>=0 and ok and not alarm
```

Giving at the first time the relation "temp >= 5 and temp <= 15" allows to simulate this system for all inputs satisfying this relation.

The result we obtain, may give unique values for some variables :
"not openin" means that the only possible value of openin is false,
"openout" means that the only possible value of openout is true

It may give general relations upon variables :
"volume <= 15 and volume >= 5" gives an assertion on possible values of volume at the first time.

This simulation also discovers a linear relation that stands between the 2 variables **Pressure** and **Volume** : "5*pressure+volume=45" at the first two time steps.

We may conclude about this example that a symbolic simulation allows to execute many concrete simulations at the same time.

In fact, usual simulation is only a particular case of symbolic simulation as we may see on the following simulation :

```
? temp=8
not openin and openout and pressure=6 and volume=15 and ok and not alarm
? temp=9
not openin and openout and pressure=5 and volume=20 and ok and not alarm
? temp=10
not openin and openout and pressure=4 and volume=25 and ok and not alarm
? temp=11
not openin and openout and pressure=3 and volume=30 and ok and not alarm
? temp=12
openin and not openout and pressure=4 and volume=25 and ok and not alarm
```

Giving input relation "temp=8" corresponds to a deterministic simulation with only one allowed input value. We then obtain a deterministic result for all output and state variables written as a conjunction. This result is the same as the one shown in previous section about usual simulation.

3.4 Discovering linear relations and verification

As reactive systems are often critical systems, we are interested in finding their properties. While simulating a program for the set all its possible inputs at each cycle, e.g with input relation `true`, we may discover linear relations that are invariants of the program. As we deal with numerical values, convergence may be enforced using upper-approximation and widening [11].

On previous example, simulating the program with always the same input relation : `true`, allows to compute all reachable behaviors for all input values of variable **temp**. This calculus reaches the following fix-point after a few steps :

```
if openin
then not openout and 5*pressure+volume=45 and volume<=45 and volume>=20
else openout and 5*pressure+volume=45 and volume<=45 and 0<=volume+4
     and not alarm and ok
```

This says that : **ok** variable is always true; **alarm** variable is always false; the linear relation : "5*pressure+volume=45" is an invariant of this program.

This may be a way to verify properties of synchronous programs.

Safety properties may be first classified according to the type of variables involved : boolean or numerical. But the way to prove them is more interesting : some properties may be proved by static analysis or theorem-proving (for example :`ok => not alarm`).

Some more difficult properties need to compute the set of reachable states on the underlying automaton (involving boolean variables).

Most difficult properties are those depending on reachability upon the underlying infinite state system (e.g. reachability analysis involving boolean and numerical variables). Property: `alarm=false`, is such a property, because it depends on numerical values reached by `Pressure` and `Volume`.

4 Implementation

We have implemented a simulator which reads Lustre programs. The underlying engine is based on binary decision diagrams (Bdd) and linear constraints.

We use a hybrid data stucture [14], merging linear constraints as atomic variables in Bdds. Unfortunately, we then loose canonicity, because a convex polyhedron has no single representation as a conjunction of constraints.

We have then developped algorithms which perform simplifications needed to decide if a relation is equivalent to the universal relation.

We also worked on variable ordering which has a great influence on the size of Bdds.

Our more recent work is about definition of a widening operator on Bdds with constraints, derived from Halbwachs's widening operator[7] on polyhedra.

5 Conclusion

We have presented symbolic simulation of synchronous programs involving boolean and numerical variables. Symbolic simulation is another way to analyze systems between test and formal verification. We showed it covers a wide range of use as it may perform usual simulation and also verification of safety properties.

Our first experiments on verification show that it may handle really difficut numerical properties, but is less efficient than tools especially dedicated to verification such as [12].

We have built a prototype [14,15] of a symbolic simulator, called YoYo.

Our goal is now mainly a practical issue than a theoretical one. Work in progress mainly concerns abstraction techniques, that we strongly need to test efficiency of our approach on industrial size case studies. Use of assertions on input variables (constraining the environment) should allow us to perform modular verification of large systems.

References

[1] Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6), August 1978.

[2] A. Benveniste, P. Caspi, N. Halbwachs, and P. Le Guernic. Dataflow synchronous languages. In *A Decade of Concurrency, reflexions and perspectives, REX School/Symposium*, volume 803 of *Lecture Notes in Computer Science*, pages 1–45. Springer Verlag, June 1993.

[3] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[4] R. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM computing surveys*, 24(3), September 1992.

[5] W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining constraints solving and symbolic model-checking for a class of systems with non-linear constraints. In *Proceedings of CAV'97*, volume 1254 of *LNCS*, pages 316–327. Springer-Verlag, June 1997.

[6] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite state concurrent systems using temporal logic specification. *ACM Trans. Programming Languages*, 8(2), April 1986.

[7] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[8] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.

[9] N. Halbwachs. Bac: A boolean automaton checker (version 1). 1994.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[11] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[12] Bertrand Jeannet. *Partitionnement Dynamique dans l'Analyse de Relations Linéaires et Application à la Vérification de Programmes Synchrones.* Thèse de doctorat, INPG (Institut National Polytechnique de Grenoble), 2000.

[13] James C. King. Symbolic execution and program testing. *ACM*, 19(7):385–394, July 1976.

[14] C. Mauras. Symbolic simulation of interpreted automata. In *3rd Workshop on Synchronous Languages*, December 1996.

[15] C. Mauras and D. Garriou. Simulation symbolique de programmes réactifs synchrones : application à la mise au point et à la vérification du controle d'une chaudière. In Hermes, editor, *MSR'99*. ENS Cachan, Mars 1999.

[16] C. Courcoubetis R. Alur and D.L. Dill. Model-checking for real-time systems. In *Procedings of* 5^{th} *Logic Computer Science*, pages 414–425. IEEE, June 1990.

[17] P. Raymond. *Compilation efficace d'un langage dclaratif synchrone : le gnrateur de code Lustre - V3.* Thèse de doctorat, I.N.P.G., Novembre 1991.

[18] Richard Gerber Tevfik Bultan and William Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *Proceedings of CAV'97*, volume 1254 of *LNCS*, pages 400–411. Springer-Verlag, June 1997.