Model checking CSM automata with PVS

Jakub Ratajczak
e-mail: J.Ratajczak@ii.pw.edu.pl
Instytut Informatyki
Politechnika Warszawska
ul. Nowowiejska 15/19, 00-665 Warszawa

Abstract

This paper shows a way of examining CSM (concurrent, finite state) automata with the PVS tool. It focuses on temporal logic properties of automata. They may be investigated with symbolic model checker built in the PVS. The way that allows translating CSM system into PVS is proposed. This paper also describes briefly symbolic model checking possibilities of the PVS tool. Showing coherency of different formal methods of software/hardware design is an interesting aspect of this work.

1. Introduction

This paper was prepared to show possible coherency of different formal methods (or tools)[1]: CSM and PVS. The CSM is a methodology that helps specifying and model checking concurrent systems. It is based on theory of concurrent finite state automata [5], [6]. The PVS system is an integrated prover with a model checker [11]. The paper shows a way of using PVS for model checking CSM automata with

1 The coherency between different formal methods was postulated on FM'99. I.e., see also [3]

PVS's model checker. This method is implemented in the Grapher tool [7] - part of the COSMA environment.

The CSM methodology is based on theory of finite state automata. It allows specifying and verifying concurrent systems. Talking very briefly it is a way of preparing one finite state automaton from a set of concurrent automata in the way that all properties of the system are preserved. The generated automaton may be then verified and results apply to the whole source system.

The PVS (Prototype Verification System) [11] is a tool that was originally designed as a semi-automatic theorem prover. The prover is based on the Gentzen's logic system. The application has implemented very powerful proving algorithms.

In the latest version, among other new features, a couple of model checking abilities were added[2]. They are based on fixpoints for sets, μ-calculus and CTL operators. It allows analyzing finite state automata in terms of temporal behavior like reaching states or producing certain output in the future (on possible paths) [1]. Because a CSM automaton is also a finite state one it allows model checking CSM systems.

2. Converting an automaton from CSM to PVS

The easiest way of preparing an automaton for PVS system is converting a CXL automaton's description to a PVS file. The source file (CXL) is a XML-based text description of an CSM automaton. Appropriate tags stand for states, transitions, conditions etc (see [8]). This file may be prepared either by hand or with the Grapher tool [7]. Grapher may convert CXL to PVS automatically.

Automaton below (Fig. 1.) comes from [4]. It is a sample of wrong designed concurrent system with two producers and one consumer. This is a product of joining three independent automata (2 producers and a consumer). Each producer produces 'put1' or 'put2' signal accordingly when the consumer is ready (signal 'empty'). The consumer is to intercept these signals and send appropriate ('take1' or 'take2') signal depending on which producer generated 'put_' signal. The system goes wrong because it is possible to loose a producer's signal. The following sections will use this example for illustrating a way using PVS as a model checker.

[2] Model checking is done by entering prover (M-x pr on certain formula) and typing '(model-check)' command.

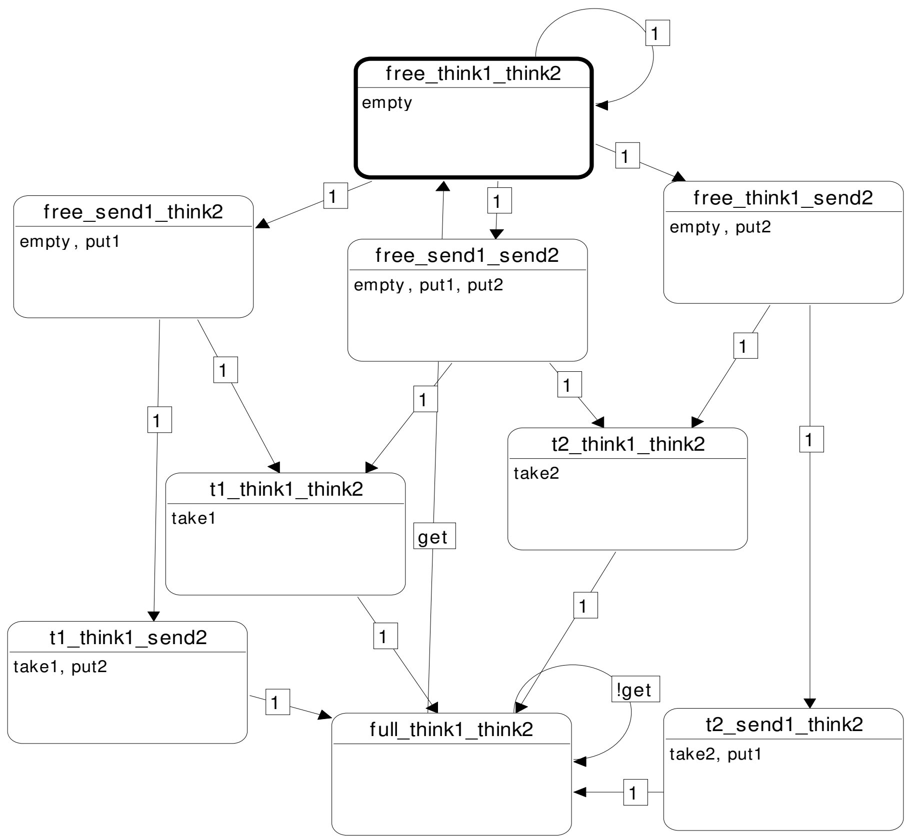

Fig. 1. Sample automaton

The target file consists of three parts: Declarations, An automaton's definition, and Universal formulae. Each part is described in following paragraphs.

2.1. Declarations

The very first line is declaration of PVS theory. The theory is named the same as the automaton with the '_th' suffix. The following keyword 'BEGIN' declares theory's body beginning.

```
prod_cons_th: THEORY
BEGIN
```

The ‘Environment’ type is a record of all possible signals automaton generates or uses in transitions’ clauses. The ‘e’ variable is declared to help building formulas later on. Access to a record’s field may be done either with brackets ‘e(take1)’ or with ‘`’ operator ‘e`take1’ (symbol ASCII: 96).

```
Environment: TYPE = [# take1:bool, put1:bool, empty:bool,
    put2:bool, take2:bool, get:bool #]
e: VAR Environment
```

It is easy to introduce other environment variable types apart from boolean. It may be any countable, limited type[3]. It has to be remembered that the size of model checked relation grows exponentially with number of environment variables. In above sample it is 2^6.

The ‘State’ type is a set of the automaton’s states. Two variables s and s1 are declared for later use.

```
State: TYPE = {full_think1_think2, t2_send1_think2,
    t1_think1_send2, t2_think1_think2, t1_think1_think2,
    free_send1_send2, free_think1_send2, free_send1_think2,
    free_think1_think2}
s, s1: VAR State
```

It is very important to understand what is in fact model-checked. These are not only automaton’s states. These are states and environment (environment’s signals). It allows constructing formulas that are able to check signals. I.e., check if always after signal s1 signal s2 occurs.

Special type is introduced to allow model checking. It connects the automaton’s states with environment variables. The ‘MState’ (M*odel checked* State) type is a record consisting of two fields:

```
MState: TYPE = [#st: State, env: Environment#]
start, ms, ms1, ms2, ms3, me, me1, me2: VAR MState
```

The ‘TransitionTable’ type is introduced to express an automaton’s transition table. It is a boolean function of start state, environment and target state. An automaton’s graph is translated into variable of this type (see below).

```
TransitionTable: TYPE = [State, Environment, State -> bool]
```

The variable of ‘ProductTable’ type represents an automaton’s product. It is relation between automaton states and environment signals.

3 In the PVS model checker only limited types may be processed.

```
ProductTable: TYPE = [State, Environment -> bool]
```

An artificial type 'MCRelation' (M*odel* C*hecked* Relation) is a relation of ' MState' that is model checked. A variable of this type is constructed from the automaton's transition table with conversion function 'tbl2mc'. This variable is the first parameter for all CTL operators. The conversion function constructs ' MCRelation' in the following way: Each two 'MState' variables are related if and only if automaton's state (st(ms)) and environment signals (env(ms)) from the first argument are related with the second argument's state (st(me)) according to automaton's transition table (mt). It doesn't matter what are the second argument environment signals.

```
MCRelation: TYPE = [MState, MState -> bool]

tbl2mc(mt: TransitionTable): MCRelation =
       (LAMBDA (ms, me): mt(st(ms), env(ms), st(me)))
```

The last section of this part is the clause importing 'ctlops' theory. It imports all CTL operators. Its parameter is the 'MState' type so operators will work on this type.

```
IMPORTING ctlops[MState]
```

Following types don't depend on actual automaton (they are common): ' MState', 'TransitionTable', 'ProductTable', 'MCRelation'. Also the 'tbl2mc' function is the same in all files.

2.2. An automaton's definition

After declaration section there is actual automaton transition table. Below is part of the sample automaton's table:

```
prod_cons_trans: TransitionTable = (LAMBDA(s, e, s1):
  % --------- full_think1_think2 --------
   (s=full_think1_think2 AND ( NOT e`get) AND
                             s1=full_think1_think2) OR
   (s=full_think1_think2 AND (e`get) AND
                             s1=free_think1_think2) OR
  % --------- t2_send1_think2 --------
   (s=t2_send1_think2 AND s1=full_think1_think2) OR
  % --------- t1_think1_send2 --------
   (s=t1_think1_send2 AND s1=full_think1_think2) OR
```

```
% ...
 % --------- free_think1_think2 --------
  (s=free_think1_think2 AND s1=free_think1_think2) OR
  (s=free_think1_think2 AND s1=free_think1_send2) OR
  (s=free_think1_think2 AND s1=free_send1_send2) OR
  (s=free_think1_think2 AND s1=free_send1_think2)
) % End of prod_cons_trans transition table
```

Start states are indicated by the ‘s’ variable, environment that provides signals allowing certain transitions is represented by the ‘v’ variable and target states by the ‘ s1’ variable. The types of these variables were defined earlier. The name of the variable that keeps this table is the same as the automaton’s name with suffix ‘ _trans’.

The automaton’s product table is a variable of ‘ProductTable’ type:

```
prod_cons_prod: ProductTable = (LAMBDA(s, e):
  (s=t2_think1_think2 AND (e`take2)) OR
  (s=t2_send1_think2 AND (e`take2 OR e`put1)) OR
  (s=t1_think1_send2 AND (e`take1 OR e`put2)) OR
  (s=t1_think1_think2 AND (e`take1)) OR
  (s=free_send1_send2 AND (e`empty OR e`put1 OR e`put2)) OR
  (s=free_think1_send2 AND (e`empty OR e`put2)) OR
  (s=free_send1_think2 AND (e`empty OR e`put1)) OR
  (s=free_think1_think2 AND (e`empty))
) % End of prod_cons_prod product table
```

The first parameter of the function is an automaton’s state while the second one is an environment. Each line describes which signals are active in different states.

Next goes predicate that is true for the initial state. It helps building formulas:

```
init(ms): bool = st(ms)=free_think1_think2
```

At the end a model checkable relation is built over the automaton’s transition table with the‘tbl2mc’ function. It is assigned to a variable called the same as the automaton:

```
prod_cons: MCRelation = tbl2mc(prod_cons_trans)
```

2.3. Universal formulae

The last section of the file is filled with sample predicates that fulfills two functions. The first is to check some universal properties of the automaton. The second is being an example for building new formulae. These formulae are described later on.

3. Automata model checking with PVS

3.1. CTL operators

All CTL operators are defined in the prelude file in the theory ‘ctlops[state: TYPE]’. The theory’s parameter ‘state’ must be provided while instantiating theory. Then all operators work with states of this type.

There are three basic CTL operators. Two of them (‘EG’ and ‘EU’) are defined with fixpoints operators. Their meaning is as follows:

- EX(N, f)(u) - there is a very next state (existence) after a start state u which holds the f predicate;
- EG(N, f) - there is in the future (existence) an infinite path that holds the f predicate;
- EU(N, f, g) - the f predicate is held (always) until the g predicate holds.

Other CTL operators are built on top of above three ones:

- EF(N, f) - there is a path (existence) on which ultimately the f predicate holds;
- AX(N, f) - the f predicate isn’t held (always) in any very next state of the start state;
- AF(N, f) - on all possible paths the f predicate is held infinitely often (but there may be states that don’t hold f);
- AG(N, f) - on all possible paths the f predicate is held always;

The very first parameter of a CTL operator is a relation on which model checking is to be done. In our case it is always a variable of type ‘MCRelation’. Normally it is called the same as the investigated automaton (I.e., ‘prod_cons’). Next one or two (‘EU’ and ‘AU’) predicates follow. They describe states we are interested in. Predicated are constructed with following syntax (see [9]):

LAMBDA *mstate: condition(mstate)*

Each operator must be used with a parameter indicating a path's start state. I.e. AF(...,...)(start).

Whole logical sentence has to start with an implication. On the left side there are requirements for start states. Normally it is the automaton's initial state (in fact it may set of 'MState' states). I.e., init(start) => AG(..., ...)(start).

The sentence must be kept in PVS syntax so it's name and THEOREM keyword is needed: cycle_strong: THEOREM init(start) => AG(..., ...)(start).

3.2. Universal formulae

The first property of an automaton that should be checked is if it cyclic or not. Cyclic means that if the initial state is left it will be reached again. There are two kinds of cyclic automata. The first when for sure an automaton will go back to the initial (or any other) state. So there are no infinite loops possible: all paths from each state lead back to the initial state. It is a strong condition. The second case where an automaton in each state has chance to go back to its initial state. It is a weak condition. This case in fact checks if there is no deadlock. There are two formulas prepared for checking above situations.

The formula that checks if given automaton in all possible paths (AG) always returns (AF) to its start state (st(start)=st(me)) is shown below:

```
cycle_strong: THEOREM init(start) =>
  AG(prod_cons, LAMBDA ms:
    AF(prod_cons, LAMBDA me:st(start)=st(me))(ms)
    )(start)
```

It was strong case. The weak one is covered by following formula: For all states in the future (AG(...)(start)) a path exists (EF) leading back to the start state. So even if a loop appears there is a way to escape.

```
cycle_weak: THEOREM init(start) =>
  AG(prod_cons, LAMBDA ms:
    EF(prod_cons, LAMBDA me:st(start)=st(me))(ms)
  )(start)
```

In our sample automaton the first formula is false (automaton has possibility of passing into infinite loops) while the second one is true (automaton does not have deadlocks).

Notify that initial state in above formulas may be exchanged with any other state and it will allow checking parts of the original automaton.

Other general properties may be validated with formulae built in the similar way. I.e. Grapher export tool defines also formulae testing deadlock.

3.3. Other formulae

Each system has its own properties that have to be checked while proving correctness. After converting an automaton to the PVS system it is designer's role to prepare appropriate formulae. In our automaton (2 producers and 1 consumer) it is important that after producing data by any producer consumer intercepts it for sure. It is true if after each 'put1' signal appears 'take1'and after each 'put2' goes 'take2'. The put1 signal comes from the first producer while 'take1'comes from the consumer and it means that it had read data from the producer 1. Similar with'put2' and 'take2' signals.

The following formula checks the first condition (with put1 and take1):

```
var1: THEOREM prod_cons_prod(start`st, (e WITH [put1:=true])) =>
  EU(prod_cons, LAMBDA ms1: prod_cons_prod(ms1`st,
                                    (e WITH [put1:=true])),
                LAMBDA ms2: prod_cons_prod(ms2`st,
                                    (e WITH [take1:=true]))
  )(start)
```

It means: Check if after any number (EU) of states (ms1) with the 'put1' signal active (e WITH [put1:=true][4]) is always a state (ms2) with the 'take1' signal (e WITH [take1:=true]). There is a different start state than in previous formulas. It is because we consider only paths that starts with a state where signal 'put1' is active.

4. Conclusions

Because PVS was not originally designed as a model checker the proposed way should be rather treated as an example of possible collaboration between different formal tools. Interesting issues may appear if use PVS as a theorem prover and treating model checked CSM automata as logic objects meeting certain criteria and

4 The following clause 'e WITH [put1:=true]' means: A record variable 'e' (in this case 'e' is ' Environment' type) with field 'put1' set to true.

proving theorems on them. This paper shows also that such a conversions may be prepared for other model checking tools (like SPIN or SMV) in the similar manner.

Bibliography

[1]. Owre S., Rushby J., Shankar N., Integration in PVS: Tables, Types and Model Checking, LNCS 1217 (366-383), Springer-Verlag 1997,

[2]. McMillan K.L., Symbolic model checking, Kluwer Academic Publishers, 1993,

[3]. Sifakis J., Integration, the Price of Success Extended Abstract, LNCS 1708 (52-55), Springer-Verlag 1999,

[4]. Mieścicki J,. Metodologia i narzędzia CASE do tworzenia i modyfikacji współbieżnego oprogramowania do celów projektowania urządzeń elektronicznych, ICS WUT Research Report 24/98 Institute of Computer Science, WUT, June 1998, [in Polish],

[5]. Mieścicki J., Hierarchical Concurrent State Machines: a framework for concurrent systems' behavioral specification and verification, ICS WUT Research Report 13/99 Institute of Computer Science, WUT, June 1999,

[6]. Mieścicki J., A methodology for concurrent systems design and analysis based on the CSM, ICS WUT Research Report 12/99 Institute of Computer Science, WUT, June 1999,

[7]. Ratajczak J., Grapher: an interactive tool for edition of Concurrent State Machines, ICS WUT Research Report 14/99 Institute of Computer Science, WUT, 1999,

[8]. Ratajczak J., CXL, a XML-based language for the description of Hierarchical Concurrent State Machines, ICS WUT Research Report 15/99 Institute of Computer Science, WUT, June 1999,

[9]. Prototype Verification System Manual,

[10]. A CSM home page at www.ii.pw.edu.pl

[11]. A PVS home page at pvs.csl.sri.com