On-the-Fly Verification with PROD

Kimmo Varpaaniemi

Helsinki University of Technology, Digital Systems Laboratory
Otakaari 1, FIN-02150 Espoo, Finland
Kimmo.Varpaaniemi@hut.fi

1 Introduction

On-the-fly verification of a property means that the property is verified during state space generation, in contrary to the traditional approach where properties are verified after state space generation. As soon as it is known whether the property holds, the generation of the state space can be stopped. Since an erroneous system can have a much larger state space than the intended correct system, it is important to find errors as soon as possible. On the other hand, even in the case that all states become generated, the overhead caused by on-the-fly verification, compared to non-on-the-fly verification, is often negligible.

It has turned out that many of the methods that have originally been designed to avoid the generation of all states in non-on-the fly verification can be applied to on-the-fly verification as well. Valmari's *stubborn set method* [3] is one such method.

In this paper we describe how the Pr/T-net reachability analysis tool PROD [1] verifies properties on-the-fly.

2 Verification

The on-the-fly verification algorithm in PROD is based on the algorithm of Valmari [3]. PROD can thus verify a property on-the-fly if the negation of the property can be expressed by means of a *tester* of the form defined in [3].

In [3], Valmari considers the computation of a parallel composition of labelled transition systems. However, it is very easy to transfer the theory to concern the generation of the reachability graph of a Pr/T-net.

A tester in a Pr/T-net is a unique place together with the arcs connected to the place. At any reachable marking, the tester place contains exactly one tuple, and such tuple is unary. The value inside the tuple is the *state* of the tester. An *action* is a transition instance, i.e. a transition with a single combination of values of the variables of the transition. An action is *visible* if and only if the action is connected to the tester place, i.e. there is at least one arc between the tester place and the transition of the action in such a way that the expression on the arc has a nonempty value. (In an arc expression of PROD, a tuple can be multiplied by a non-constant expression.)

We can now associate special meanings with the states of the tester and proceed as in [3]. The algorithm in [3] is directly applicable to the generation of the reachability graph of the net since a reachable marking of the net can imagined to be a pair of the state of the tester and the state of the actual system. If stubborn sets are wanted, they are computed by a Petri net oriented algorithm which satisfies the conditions mentioned in [3].

An example is given in the appendix. There we have the classical dining philosopher example with 1994 philosophers and demonstrate the usage of a so called *livelock monitor state.* The stubborn set method is used in the example.

3 Discussion

It would be tempting to say that any linear time temporal property can be expressed by means of a tester. This is true in the sense that a Büchi automaton is a special case of a tester. However, if we actually construct a tester corresponding to a given formula by using any of the known algorithms, the alphabet of the resulting tester consists of atomic propositions or sets of atomic propositions. Consequently, a linear time temporal formula can be expressed by means of a tester of the form used in PROD if the atomic propositions in the formula are actions, whereas nothing can be guaranteed if some atomic proposition in the formula is not an action.

This is a practical problem since it is often more intuitive to describe a property of a Petri net in terms of places than in terms of transitions. In private discussions with Valmari it turned out that the theory in [3] can be extended to handle state-oriented information. On the other hand, implementing such extension in PROD seems to require more description power to the net description language and extending the reachability graph generation algorithms to cover the increased description power.

Peled [2] has presented an on-the-fly verification algorithm which is slightly similar to Valmari's algorithm but works with any atomic propositions. Some of the ideas in [2] could perhaps be utilized in the development of PROD.

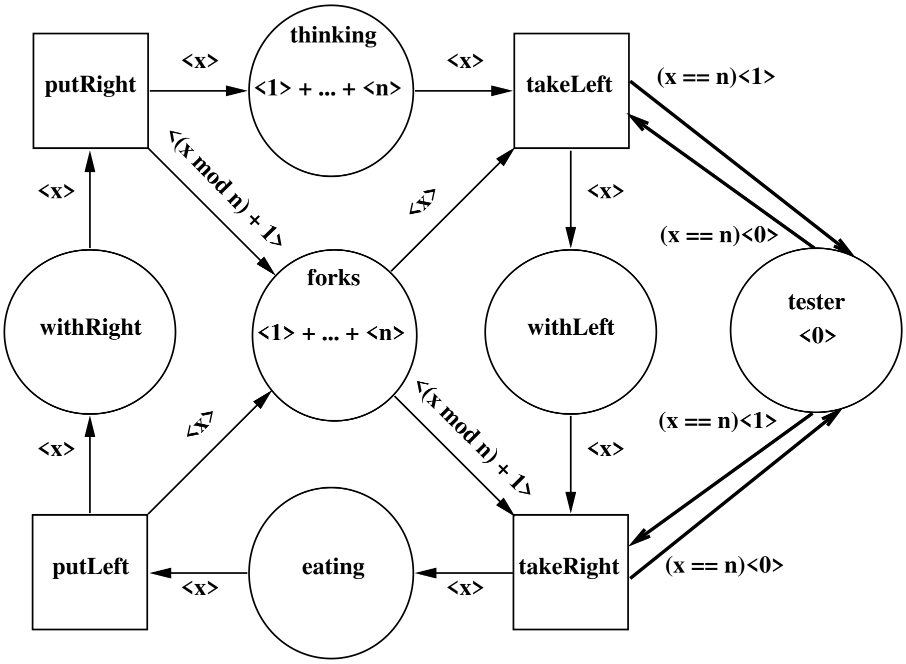
Acknowledgements

This work has been funded by the Technology Development Centre of Finland.

References

1. Grönberg, P., Tiusanen, M., and Varpaaniemi, K.: *PROD—A Pr/T-Net Reachability Analysis Tool.* Helsinki University of Technology, Digital Systems Laboratory Report B 11, Espoo 1993, 44 p.
2. Peled, D.: *Combining Partial Order Reductions with On-the-Fly Model-Checking.* Dill, D. (Ed.), Proceedings of CAV '94, Stanford CA, June 1994. LNCS 818, Springer-Verlag, Berlin 1994.
3. Valmari, A.: *On-the-Fly Verification with Stubborn Sets.* Courcoubetis, C. (Ed.), Proceedings of CAV '93, Elounda, Greece, June/July 1993. LNCS 697, Springer-Verlag, Berlin 1993, pp. 397–408.

Appendix



To find a loop where philosopher n holds his left-hand fork, we search for an invisible loop where tester is marked by <1>. The only visible actions are takeLeft(x == n) and takeRight(x == n).

```
ws4% cat dining.net
#define n 1994
#define LEFT(x)  (x)
#define RIGHT(x) (1 + ((x) % n))
#place thinking  lo(<.1.>) hi(<.n.>) mk(<.1..n.>)
#place forks     mk(<.1..n.>)
#place withLeft  lo(<.1.>) hi(<.n.>)
#place eating    lo(<.1.>) hi(<.n.>)
#place withRight lo(<.1.>) hi(<.n.>)
#tester tester
#place tester    lo(<.0.>) hi(<.1.>) mk(<.0.>)
#monitor 1 livelock
#trans takeRight
   in  { forks: <.RIGHT(x).>; withLeft: <.x.>;
         tester: (x == n)<.1.>; }
   out { eating: <.x.>; tester: (x == n)<.0.>; }
#endtr
#trans takeLeft
   in  { thinking: <.x.>; forks: <.LEFT(x).>;
         tester: (x == n)<.0.>; }
```

```
   out { withLeft: <.x.>; tester: (x == n)<.1.>;}
#endtr
#trans putLeft
   in  { eating: <.x.>; }
   out { withRight: <.x.>; forks: <.LEFT(x).>; }
#endtr
#trans putRight
   in  { withRight: <.x.>; }
   out { thinking: <.x.>; forks: <.RIGHT(x).>; }
#endtr
ws4% prod dining.net
Compiling...
Generating reachability graph
Livelock reached
Loop 1994 [1> 1996 [0> 1997 [0> 1999 [0> 1994
For more information, start "probe dining" and look at the set %2
ws4% probe dining
0#goto 1994
1994#calc withLeft
<.1994.>
1994#succ arrow 1
Arrow 1: transition takeLeft, precedence class 0
  x = 1
to node 1996
------------------------------------------------
1994#next 1
1996#succ arrow 0
Arrow 0: transition takeRight, precedence class 0
  x = 1
to node 1997
------------------------------------------------
1996#next 0
1997#succ arrow 0
Arrow 0: transition putLeft, precedence class 0
  x = 1
to node 1999
------------------------------------------------
1997#next 0
1999#succ arrow 0
Arrow 0: transition putRight, precedence class 0
  x = 1
to node 1994
------------------------------------------------
```