Real-Time Systems Design with PEP

Christian Stehno

Fachbereich Informatik,
Carl von Ossietzky Universität Oldenburg,
D-26111 Oldenburg, Germany
Christian.Stehno@informatik.uni-oldenburg.de

Abstract. The PEP tool provides an integrated development and verification environment for parallel systems. Beginning with version 2.0 it also offers use of timed systems. This paper describes a sample session for the design and the verification by partial order based techniques of a simple real-time model for a mutual exclusion algorithm.

1 Overview of PEP

The PEP tool is a **P**rogramming **E**nvironment based on **P**etri nets[1]. It supports the development, the simulation and the verification of parallel programs by a variety of tools and techniques through its graphical user interface.

While PEP is mainly an integrated development environment for the parallel programming language $B(PN)^2$ [3], including a C code generator [11] and allowing programs to be transformed into Petri nets for verification, it also includes a complete Petri net tool providing editors, simulators and a large number of analysis and verification tools. An overview of the capabilities of PEP can be found in Sect. 2, while Sect. 3 describes a sample modelling session and Sect. 4 concludes the paper.

2 Components of the PEP Tool

Development Level. Due to the compactness and simplicity of notations used, this level is frequently used for large systems specification and design. In addition to an editor for $B(PN)^2$, there are also ones for the Specification and Description Language SDL [7] and for parallel finite automata [9]. The C code generator produces plain C code which supports the parallel execution of $B(PN)^2$ programs by the means of Unix processes. This code can be extended by a programmer to complete the algorithm written in $B(PN)^2$ and produce an executable for some target machine.

1 http://parsys.informatik.uni-oldenburg.de/~pep

Net Level. On the net level, the Petri net tool part of PEP is located. This level provides editors and simulators for high and (timed) low level nets, and more. The net editor contains a composition interface for nets supporting system design with Petri nets. All systems designed on the development level can be compiled into a Petri net representing their semantics. Along with this transformation, references are generated between representations providing the ability of PEP to simulate not only the nets but also the higher level systems directly.

Analysis Level. The analysis level includes Petri net specific verification checks and model checking of all models used within PEP. The behaviour of a net can be examined in many ways, ranging from state space analysis to model checking with linear and branching time logics. The next section will focus on those verifiers capable of using timed systems as input.

3 Timed Systems Analysis

3.1 Preliminaries

For the time being, PEP supports timing information within the Petri Box Calculus (PBC, [2, 13]), on the low level nets (according to [15]), within all programs that support the finite prefix (which can be built for Time Petri nets, cf. [4, 8]), and the INA tool [16]. The net editor supports time only on the editing level, simulation is rudimentary, i.e. only for the display of counterexamples.

We will model a variant of Fischer's protocol (cf. [1]) for two processes. This protocol provides a very simple mutual exclusion algorithm for an arbitrary number of processes, such that only one of them may enter the critical section and then halts, although the protocol may be extended for more than one round. The crucial cause of the correctness of the protocol are time intervals for writing and reading a global variable, making this an interesting timed protocol.

3.2 Modelling the Petri Net

The lack of high level timed models within PEP forces a system designer to start at the net level. As the PBC offers a simple and structured interface to (Time) Petri nets, we use it as the starting point for our example. The PBC provides a basic set of operations for combining nets into more complex ones. Most notably there are sequential (`;`), parallel (`||`) and choice operators (`[]`), synchronous communication mechanisms with optional block encapsulation and some primitive variable and data type facilities, including (finite) integer numbers, stacks and channels.

For the example, the shared variable `v` is needed. The macro for a variable is `D_BOX(v,0,{0..2})`, which creates a variable `v` with type {0,1,2} and initial value `0`. Access to this variable is provided by action symbols of the form `v_pre_post`, changing `v` from `pre` to `post`. This box is put in parallel with two processes, each of which consists of three sequential steps (see Fig. 1, lines 4 and

6). Each step is given by the action symbol to be executed and a time interval (attached to it by an @) defining earliest and latest execution times after enabling. Thus, an interval $[m, n], m \leq n, m, n \in \mathbb{N}$ allows to execute the action after waiting at least m (integral) time steps; after n time steps the action has to occur, if still possible.

In each process the variable is initially checked for value zero, which has to be done immediately (interval `[0,0]`). After waiting at most one time step, the variable is set to value i for the ith process, which is done in a choice as other processes might have changed the value of `v` already. In a third step, the variable is checked for a value of i, in which case process i is allowed to enter its critical section[2] after waiting strictly longer than one time unit.

Finally, all components are glued together by the scoping operation over all used symbols, creating the correct links between the processes and the variable.

```
[{v,v_0_0,v_0_1,v_0_2,v_1_0,v_1_1,v_1_2,v_2_0,v_2_1,v_2_2}:
D_BOX(v,0,{0..2})
||
(v_0_0@[0,0]; (v_0_1@[0,1][]v_1_1@[0,1][]v_2_1@[0,1]); {v_1_1,cs}@[2,4])
||
(v_0_0@[0,0]; (v_0_2@[0,1][]v_1_2@[0,1][]v_2_2@[0,1]); {v_2_2,cs}@[2,4])
]
```

Fig. 1. Fischer's protocol for 2 processes in PBC syntax

The program in Fig. 1 can be compiled into the Time Petri net shown in Fig. 2. The transformation is almost a 1:1 correspondence, mapping actions and their accompanying time restrictions onto transitions. Arcs are created depending on the control flow of the program and the communication structure with the global variable. Further action will take place on the net rather than the PBC code.

3.3 Verification

With the latest version of its unfolder, PEP is able to create the finite prefix of a net analogously to [14, 6] for Time Petri nets [4, 8]. The finite prefix contains the complete information on the behaviour of nets in a compact way using partial order. The PEP model checker and some other analysis tools use the unfolding as input. On the finite prefix for Time Petri nets, these tools can be used for timed nets. For reachability and deadlock problems, the clp tool [12] and mcsmodels [10] are, besides the original PEP deadlock checker, capable of using finite prefices and thus checking Time Petri nets.

By reachability checking, all properties of interest can be verified, notably:

[2] Denoted by the extra symbol `cs`, which does not affect the net.

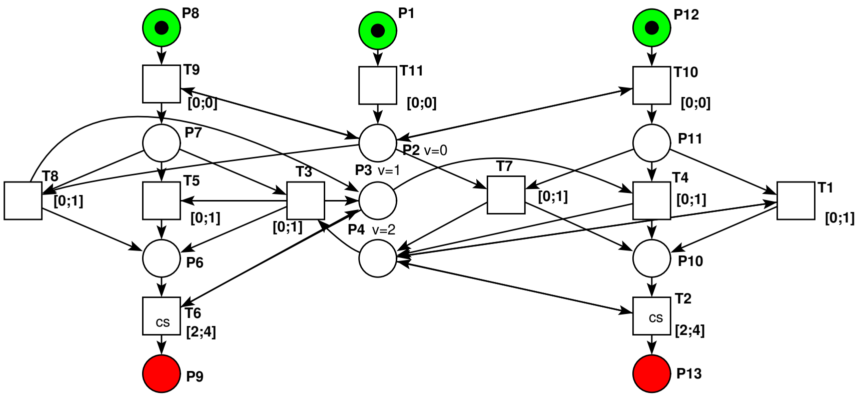

Fig. 2. Fischer's protocol for 2 processes as Time Petri net

- Each process may enter its critical section (P9 or P13 resp.).
- Only one process is in the critical section at a time ($\neg$ P9 $\wedge$ P13).

As checking the second property generates a positive answer, the protocol ensures mutual exclusion. For the first property, the system generates a witnessing firing sequence that might be simulated by the net editor. This holds in general for possible counterexamples and provides an easy way of debugging erroneous code.

Although reachability checking suffices for this example, usually many properties are not expressible that easy and use some kind of temporal logics. To verify them, a set of model checkers is provided by PEP. The PEP model checker uses the CTL fragment defined in [5]. It provides *place assertions* to express properties of the current marking of the net and two temporal operators *"possibly"* for reachability and its dual *"always"*, which may be nested. A second model checker available for Time Petri nets in PEP is the CTL analyser of INA. Both may also check the reachability and mutual exclusion problems stated above, but go far beyond these, and also beyond the scope and space of this paper.

4 Conclusion

We have described a sample session of modelling and verifying a timed system. PEP provides a comprehensive, easy-to-use graphical user interface that supports the entire development process. This includes design, verification, simulation and debugging with a variety of different tools. Although the current implementation offers just a basic approach to timed systems, it alleviates their design by far. Further development will include timed high-level models, e.g. Real-time B(PN)^2, a more complete subset of SDL, or Timed Automata.

Acknowledgements

The author would like to thank Eike Best, Hans Fleischhack and the anonymous referees for their comments on this work.

References

1. Abadi, M., Lamport, L.: An Old-Fashioned Recipe for Real-Time. In Real Time: Theory in Practice. Volume 600 of LNCS. Springer-Verlag (1992) 1–27
2. Best, E., Devillers, R., Hall, J.: The Box Calculus: a New Causal Algebra with Multi-Label Communication. In Proc. of APN'92. Volume 609 of LNCS. Springer-Verlag (1992) 21–69
3. Best, E., Hopkins, R.P.: $B(PN)^2$– a Basic Petri Net Programming Notation. In Proc. of PARLE'93. Volume 694 of LNCS. Springer-Verlag (1993) 379–390
4. Bieber, B., Fleischhack, H.: Model Checking of Time Petri Nets Based on Partial Order Semantics. In Proc. of Concur'99. Volume 1664 of LNCS. Springer-Verlag (1999) 210–225
5. Esparza, J.: Model Checking Using Net Unfoldings. Science of Computer Programming **23**. Elsevier (1994) 151–195
6. Esparza, J., Römer, S., Vogler, W.: An Improvement of McMillan's Unfolding Algorithm. In Proc. of TACAS'96. Volume 1055 of LNCS. Springer-Verlag (1996) 87–106
7. Fleischhack, H., Grahlmann, B.: A Compositional Petri Net Semantics for SDL. In Proc. of ATPN'98. Volume 1420 of LNCS. Springer-Verlag (1998)
8. Fleischhack, H., Stehno, C.: Computing the Finite Prefix of a Time Petri Net. Submitted paper. 2001
9. Grahlmann, G., Moeller, M., Anhalt, U.: A New Interface for the PEP Tool – Parallel Finite Automata. In Proc. of AWPN'95, AIS **22**. FB10 Universität Oldenburg (1995)
10. Heljanko, K.: Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets. In Proc. of TACAS'99. Volume 1579 of LNCS. Springer-Verlag (1999) 240–254
11. Jaeger, J.: Portable Codegenerierung für eine parallele Programmiersprache. Masters thesis. Universität Hildesheim (1997) *in German*
12. Khomenko, V., Koutny, M.: LP Deadlock Checking Using Partial Order Dependencies. In Proc. of CONCUR'2000. Volume 1877 of LNCS. Springer-Verlag (2000) 410–425
13. Koutny, M.: A Compositional Model of Time Petri Nets. In Proc. of Application and Theory of Petri Nets 2000. Volume 1825 of LNCS. Springer-Verlag (2000) 303–322
14. McMillan, K.: Symbolic model checking: An approach to the state explosion problem. Kluwer Academic Publishers (1993)
15. Merlin, P., Faber, D.: Recoverability of Communication Protocols – Implication of a Theoretical Study. IEEE Transactions on Software Communications **24** (1976) 1036-1043
16. Starke, P.: INA – Integrated Net Analyzer Version 2.2. Humboldt-Universität Berlin (1999)