Verification using PEP*

Stephan Melzer, Stefan Römer** and Javier Esparza

Institut für Informatik
Technische Universität München
Arcisstr. 21 D–80290 München
e-mail:{melzers, roemer, esparza}@informatik.tu-muenchen.de

Abstract. **PEP** is a tool for the design, analysis and the verification of parallel programs. Two approaches are presented in this paper being the underlying technique of the verification component of **PEP**.
KEY WORDS: Verification, Net Unfoldings, Linear Programming.

1 Introduction

PEP[1] (**P**rogramming **E**nvironment based on **P**etri Nets) is a tool for the verification of parallel programs which implements two of the main theoretical developments of the Esprit BRA DEMON [2] and its successor, the Esprit WG CALIBAN [4]:

- the design of the Box Calculus, a rich process algebra with a compositional net semantics [3];
- the development of new verification techniques for the Petri net model [8, 9, 14].

Typically, the user starts a **PEP** session by typing a finite-state concurrent program in an imperative, concurrent programming language called **B**asic **P**etri **N**et **P**rogramming **N**otation B(PN)^2 [6]. The language contains control instructions for sequential and (nested) parallel composition, iteration and nondeterministic choice. Different processes can communicate via both handshake and buffered communication channels, as well as shared variables.

PEP automatically translates B(PN)^2 programs into Petri boxes, a particular class of 1-safe, labelled Petri nets. The translation is fully compositional, i.e., in a first step elementary Petri boxes are generated for the atomic actions of the program, and then these Petri boxes are joined using net operators corresponding to the different program constructs. The theoretical framework of this translation is the Box Calculus.

* This work was supported by the Sonderforschungsbereich SFB-342 A3 – SAM.
** Partially this work was done while the author was a member of the University of Hildesheim.
[1] **PEP** is supported by the Deutsche Forschungsgemeinschaft and was mainly developed by the University of Hildesheim [5].

Once the program has been typed, the user can express properties in a simple branching-time logic which extends propositional logic with a possibility operator. Typically, they are used for expressing safety and liveness properties, respectively. The atomic expressions of the two logics are of the form *the variable x has value n* or *the n-th process of the program is at location l*. These properties are automatically translated by **PEP** into properties of the underlying Petri net.

PEP offers the user two different model checking techniques for the verification of properties. They have been developed at the Technical University of Munich and the University of Hildesheim. They are described in sections 2 and 3, respectively. Here we only mention that the two have been developed with the goal of palliating the state explosion problem; in fact, none of them constructs the state space of the system.

2 Verification by Means of Linear Programming

The **PEP** tool offers a model-checking component that is based on a linear upper approximation of the state space of a program. The drawback of this model checker is the fact that the underlying method is only a semidecision method. On the other hand, the approximation is obtained from a structural analysis of the associated Petri box that does avoid the well-known state explosion problem.

Linear upper approximations of the set of reachable states have also been used by Cousot and Halbwachs and others in the field of abstract interpretation [7, 11]. The main difference with our approach is that we derive the linear approximation directly from the structure of the system, in one single step, and not by means of successive approximations, as in [7].

In the Petri net community several structural analysis techniques exist that represent necessary criteria for the reachability test of states [1]. One part of these techniques have an algebraic origin and thereby, can be easily expressed as a system of inequalities. Other techniques need nontrivial appropriate transformations yielding linear algebraic representation [14]. A linear approximation of the state space is obtained by the union of all these systems, denoted by $\mathcal{L}$.

Moreover, we can formulate linear inequalities $\mathcal{L}_\phi$ characterizing states that violate property ϕ. The infeasibility of the system $\mathcal{L}_\phi \cup \mathcal{L}$ implies that ϕ holds for every reachable state of the program. The feasibility is checked by means of a mixed integer programming solver. Properties that can be expressed in such a way belong to the class of safety properties, e.g. deadlock-freedom, mutual exclusion property, *etc.*

Unlike safety properties, liveness properties need the consideration of infinite occurrence sequences of a program. It also exists an upper approximation $\mathcal{L}'$ of those sequences by means of structural analysis. Furthermore this approximation can be restricted (specified) by Horn formulae q_i among variables of the program.

If $\mathcal{L}'$ contains a sequence that models the negation of a liveness property ϕ *and* satisfies all formulae q_i, then ϕ holds for all sequences of the program. This containment test is carried out using constraint programming [10], because its techniques combine Horn logic and linear programming.

In the near future we plan to implement a semidecision method for entire Propositional Linear Time Temporal Logic (PLTL). In this connection the negation of a PLTL-formula ϕ is translated into a Petri net [13], that behaves equivalent to an associated Büchi automaton. If the Petri Box of a program and this Petri net have no common occurrence sequences, then ϕ holds for the program. This check can be solved by the above-mentioned techniques of linear and disjunctive programming respectively.

3 Verification Based on Net Unfoldings

The second model checking technique, unlike the latter one, checks the satisfaction of temporal properties by means of an exact representation of the state space. Reduction methods are used again to avoid the state explosion problem.

One possibility to reduce the state space of the system is to use partial orders (acyclic nets). Several runs of a system can be modelled by net unfoldings, a special partial order structure.

For verification purposes, these unfoldings have the problem of being infinite even for systems with a finite number of states. In the **PEP** environment an improved version [9] of an algorithm introduced by McMillan [12] is used to construct a *finite prefix* of the unfolding that is large enough to represent all the reachable states of the system. Computation of this prefix is terminated by finding so–called *cut–off points* that produce states already contained in the constructed part of the finite prefix.

The unfold method allows a fully automatic verification of safety/liveness properties. A fast model checker [8] has been integrated in the **PEP** tool, which takes the unfolding and a formula as input to verify whether the formula holds for the corresponding program. The formula can be expressed in a simple branching time or linear time logic.

Another interesting problem is the question of detecting deadlocks, i. e. if there exists a reachable state in that no transition of the underlying net is enabled. Deadlock-freedom can be expressed by a temporal logic formula, while there exist more efficient algorithms for this purpose. In the **PEP** tool we have included a component that verifies the deadlock property of a given concurrent program using the finite prefix of its unfolding, based also on a procedure by McMillan [12]. The basic idea behind this algorithm is to find a configuration, a special kind of "path" containing the complete history needed for enabling a certain event (element) of the prefix, which is in conflict with all of the cut–off points. If such a configuration exists, the checked program contains no deadlock.

4 Conclusion

We have described the two verification techniques implemented in the **PEP** tool. The entire tool is embeded in a graphical user interface containing editors for programs, Petri nets, *etc.* The **PEP** tool – already containing the model checker

based on unfoldings – is available via WWW[2] or for further information send an e-mail to `pep_help@informatik.uni-hildesheim.de`.

References

1. Bernd Baumgarten. *Petri-Netze. Grundlagen und Anwendungen.* BI-Wissenschaftsverlag, 1990.
2. Eike Best. Esprit Basic Research Action 3148 DEMON (Design Methods Based on Nets) – Aims, Scope and Achievements. In *Advances in Petri Nets*, volume 609 of *Lecture Notes in Computer Science*, pages 1–20. Springer Verlag, 1992.
3. Eike Best, Raymond Devillers, and Jon G. Hall. The Box Calculus: A new causal Algebra with multi-label Communication. In *Advances in Petri Nets 92*, volume 609 of *Lecture Notes in Computer Science*, pages 21 – 69. Springer-Verlag, 1992.
4. Eike Best, Raymond Devillers, Elisabeth Pelz, Arend Rensink, Manuel Silva, and Enrique Teruel. Caliban – Esprit Basic Research WG 6067. In *Structures in Concurrency Theory*, Workshops in Computing, pages 2 – 31, Berlin, May 1995. Springer Verlag.
5. Eike Best and Hans Fleischhack. PEP: Programming Environment Based on Petri Nets. Hildesheimer Informatik Bericht 14/95, Universtät Hildesheim, May 1995.
6. Eike Best and Richard Pinder Hopkins. B(PN)^2 – a Basic Petri Net Programming Notation. In *Proceedings of PARLE '93*, volume 694 of *Lecture Notes in Computer Science*, pages 379 – 390. Springer-Verlag, 1993.
7. P. Cousot and N.Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages*. ACM-Press, 1978.
8. Javier Esparza. Model checking using net unfoldings. Science of Computer Programming 23, pp. 151 – 195 (1994).
9. Javier Esparza, Stefan Römer, and Walter Vogler. An Improvement of McMillan's Unfolding Algorithm. Sonderforschungsbericht 342/12/95 A, Technische Universtät München, München, August 1995. (to appear in the proceedings of TACAS '96).
10. Thom Frühwirth, Alexander Herold, Volker Küchenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, and Mark Wallace. Constraint Logic Programming – An Informal Introduction. ECRC–93–5, European Computer-Industry Research Centre, München, 1993.
11. N. Halbwachs. About synchronous programming and abstract interpretation. In *SAS '94: Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*, pages 179–192. Springer-Verlag, 1994.
12. K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *4th Workshop on Computer Aided Verification*, pages 164 – 174, Montreal, 1992.
13. Stephan Melzer. Büchi Nets – A Pendant to Büchi Automata. Sonderforschungsbericht, Technische Universität, 1996. (in preparation).
14. Stephan Melzer and Javier Esparza. Checking system properties via integer programming. Sonderforschungsbericht 342/13/95 A, Technische Universität München, August 1995. (to appear in the proceedings of ESOP '96).

[2] `http://www.informatik.uni-hildesheim.de/~pep/HomePage.html`

This article was processed using the LaTeX macro package with LLNCS style