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\(^1\) (Programming Environment based on Petri 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 Basic Petri Net Programming Notation B(PN)\(^2\) [6]. The language contains control instructions for sequential and (nested) parallel composition, iteration and non-deterministic 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 semi-decision 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 \( \phi \). The infeasibility of the system \( \mathcal{L}_\phi \cup \mathcal{L} \) implies that \( \phi \) 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 these 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 \( \phi \) and satisfies all formulae \( q_i \), then \( \phi \) 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 semi-decision method for entire Propositional Linear Time Temporal Logic (PLTL). In this connection the negation of a PLTL-formula \( \phi \) 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 \( \phi \) 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 embedded 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


\(^2\) [http://www.informatik.uni-hildesheim.de/~pep/HomePage.html](http://www.informatik.uni-hildesheim.de/~pep/HomePage.html)
This article was processed using the \LaTeX\ macro package with LLNCS style