A New Interface for the PEP[a] Tool
– Parallel Finite Automata –

Bernd Grahlmann, Matthias Moeller and Ulrich Anhalt

Institut für Informatik
Universität Hildesheim
Marienburger Platz 22
D-31141 Hildesheim
Germany
{bernd, mmoeller, anhalt}@informatik.uni-hildesheim.de

Abstract

This paper introduces a special kind of parallel finite automata (PFA) with B(PN)^2 actions as edge annotations. An algorithm for the automatic translation into a B(PN)^2 program is described. The algorithm is non-trivial because the structure of a PFA is preserved in the generated B(PN)^2 program.

The modelling of the 'Alternating-Bit' protocol using the parallel composition of four finite automata (FA) is considered as an example. The resulting B(PN)^2 program can be used directly as an input for the current version of the **PEP** tool. Thus one further input interface is provided to the user of the tool.

Keywords: Alternating-Bit-Protocol, B(PN)^2, Finite Automata, **PEP**.

1 Introduction

At the present time the **PEP** tool offers four different possibilities to model parallel systems [5, 6]. The most **PEP** specific one is to write a program in the parallel programming language B(PN)^2, that can be automatically translated into a 1-safe Petri net [2, 3]. Furthermore the modelling can be done directly on the net level by editing either a high level net (M-net) or a low level Petri net. Finally it is possible to write a process algebra term.

However, such a variety of different approaches is supported, it turned out that it would be fine, if beyond that PFA could be used for the modelling. This paper presents an algorithm for the automatic translation of a PFA into a B(PN)^2 program. In section 2 a special kind of PFA with B(PN)^2 actions as edge annotations is introduced. The B(PN)^2 *do_od* construct which is fundamental for the work of the compiler is described briefly in section 3. The two passes of the compiler are explained in section 4.

As an example the modelling of the 'Alternating-Bit' protocol using the parallel composition of four FA is done in section 5.1. The resulting B(PN)^2 program that is given in section 5.2 can be used directly as an input for the current version of the **PEP** tool. Thus one further input interface is provided to the user of the tool. Finally a conclusion is given in section 6.

2 Parallel finite automata

In the following B(PN)^2 specific PFA are considered. A PFA is a collection of FA [7] acting in parallel, where one FA corresponds to one process in a program. A FA consists of a start node, a set of local nodes, a set of exit nodes, a set of edges between these nodes and a labelling function that annotates each edge with a B(PN)^2 action. A start node represents the initial state of one process and an exit node represents a state in which the process has terminated. Thus no outgoing edges are accepted for exit nodes. In figure 5.1.1 a PFA that is a collection of four FA is shown.

[a]The **PEP** (Programming Environment based on Petri nets) project is financed by the DFG (Deutsche Forschungsgemeinschaft).

3 B(PN)^2

B(PN)^2 offers the possibility of synchronous communication between parallel processes via channels with capacity 0. A read operation ($c?$) of one process is synchronized with a write operation ($c!$) of another one yielding a transition in the net semantics of the program. Thus there is no need to handle this kind of synchronization in the compiler from a PFA to a B(PN)^2 program.

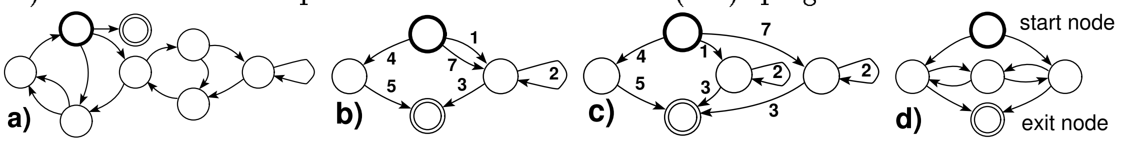
do
  $\Box$ $\langle action\_1_1 \rangle$; ... $\langle action\_1_{m_1} \rangle$; exit
  $\vdots$
  $\Box$ $\langle action\_n_1 \rangle$; ... $\langle action\_n_{m_n} \rangle$; repeat
od

The most interesting part of B(PN)^2 for the compiler is the *do_od* construct. It offers the possibility to combine an arbitrary number of *repeat* and *exit* alternatives to a nondeterministic choice. If *action_p$_1$* is executable the p'th alternative can be chosen. An *exit* exits the loop and an *repeat* starts it once again. Of course *do_od* constructs can be nested.

4 Algorithms underlying the 2-pass compiler

The compiler is a two-pass-compiler. During the first pass each FA of the PFA is considered individually. The structure of the FA, i.e. opening and closing alternatives, loops and cycles, is analysed and stored in the nodes. The second pass prints out the target code using this information.

The compilation assumes two preconditions. First of all every node has to be on a path from a start node to a terminal node. This property is checked at the beginning. Then only those FA are accepted that can be considered as graphs without cycles, which can be supplemented by any number of loops and (also complex) cycles. The essential characteristic is the possibility of a well-structured compilation into *do..od* loops with *repeat* and *exit* alternatives. A violation of this property is recognized during the first pass of the compilation. Thus example a) is accepted, and b) results in the same B(PN)^2 program as c). A large effort is necessary to reject examples like d) that cannot be compiled into a well-structured B(PN)^2 program.



4.1 The First Pass

The main purpose of the first pass is to calculate information that is needed to generate the target code afterwards. For this purpose the FA is scanned repeatedly. The algorithm tries to find paths from the start node to one of the exit nodes. If the FA has the B(PN)^2 specific structure, all nodes and edges are inspected. Loop edges are neglected.

The search along a path (beginning at the start node) is interrupted if a node with at least one incoming edge, that has not been fully inspected (i.e. at least one search that has to arrive via this incoming edge has not arrived) so far, is reached. In this case the algorithm continues with the search along another path. At each moment only one search is active, whereas the interrupted ones are stored in a queue.

If all searches are interrupted and none of them is able to make a step, a cycle starting at one of the currently inspected nodes must be the reason that at least one of the incoming edges has not been inspected. Thus a sequence of searches (one for each of these nodes) for cycles is started. During each of these searches all cycles belonging to the considered node are detected. The sequence of searches is terminated and the global search is continued if at least one cycle has been detected.

A global search ends if an exit node is reached. The whole pass is finished if all searches are terminated.

A fuller understanding of the algorithm is based on the following two details:

1. The following information about the relevant history of the inspected path is associated with each search:
   (a) The starting node of the search.
   (b) The currently inspected node.
   (c) A list of all opened (but not yet closed) alternatives. Each opened alternative is characterised by the corresponding node and the information that the m-th out of n alternatives has been chosen.
   (d) A multiplicity may indicate that more than one searches with different histories will arrive via the same edge.
2. During the run of the algorithm information is generated stepwise for each node with more than one input or output edge, i.e. opening or closing node for alternatives.
   (a) A list of entries, each one containing the history of an incoming search together with the corresponding incoming edge.
   (b) A list of all closed alternatives.
   (c) A flag that indicates whether or not all possible closings have been done.
   (d) A counter for the number of searches that should arrive, because it is possible that more than one search arrives via the same edge.

With this information in mind a more detailed description of some parts of the algorithm is given next:

- If there are more than one incoming edges in node Y and all of them are inspected in full multiplicity, the algorithm tries to close successively as many opened alternatives as possible. Alternatives can be closed if and only if all alternatives that have been opened in node X come together in Y. If a closing can be done, this information is stored. Thus every affected search can delete the information about the corresponding opening in its history later on. If an opening with, say, seven alternatives should be closed but only three of them have come together, the first of the corresponding searches has to increase the multiplicity accordingly to indicate that the other two will arrive via the same edge.

- If all incoming edges are inspected in full multiplicity and all possible closings have been done, the search can be continued. If there are $n > 1$ outgoing edges then a new search with the past history of the current search, supplemented by the newly opened alternative, is stored in the queue for each of the first $n - 1$ outgoing edges. The current search follows the n-th alternative.

- The search for cycles behaves nearly the same as the global search. One of the main differences is that the FA is recognized not to be compilable if the cycle search, starting from node X, arrives at a node that has been visited either from a global search (starting at the start node) or a different cycle search starting at Y.

4.2 The Second Pass

The second pass converts the information provided by the first pass by a recursive run. Each path from a start node to an exit node is tracked and compiled.

- If an edge is followed, the $B(PN)^2$ action in the annotation is output.
- If an opening node is reached, the algorithm is invoked recursively for every *exit* or *repeat* alternative.
- If a closing node for the followed alternative is reached, the (recursively invoked) procedure is exited.

5 Alternating bit protocol

The ‘alternating-bit’ protocol [1, 8] has been designed to ensure reliable transmission of messages even if the communication media looses or duplicates messages or acknowledgments. ‘Send’ tries to send an alternating sequence of ‘0’ and ‘1’ messages. ‘Reply’ acknowledges them with ‘0’ and ‘1’ acknowledgments respectively; ‘Trans’ and ‘Ack’ are the communication media between ‘Send’ and ‘Reply’, and vice versa.

5.1 PFA

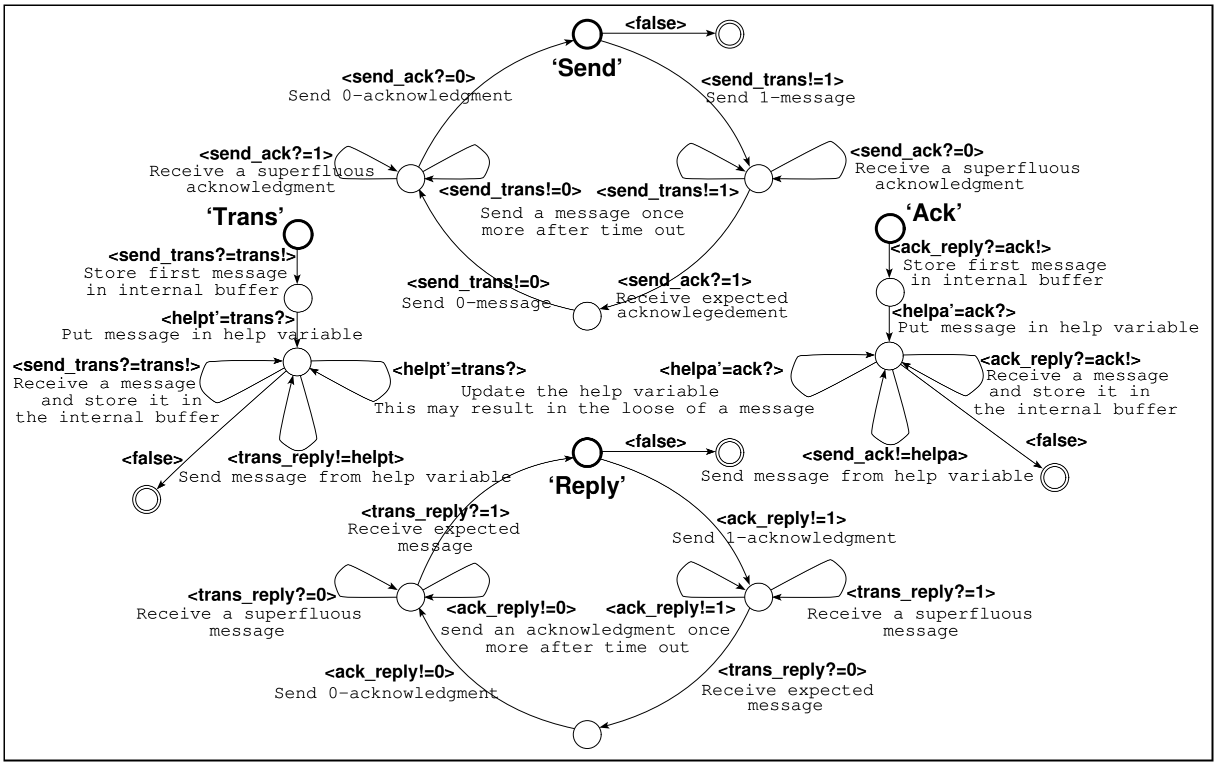


Figure 5.1.1 : Parallel composition of the four FA.
○≡ start node, ○≡ internal node and ◎≡ exit node.
The edges to the exit nodes annotated with $\langle \mathit{false} \rangle$ indicate that the FA do not terminate (remember: every node must be on a path to an exit node).
$\langle ack? = 1 \rangle$ receives 1 on the channel *ack*, $\langle ack! = 1 \rangle$ sends 1.

Figure 5.1.1 shows a model of this protocol as a parallel composition of four FA: 'Send', 'Trans', 'Reply' and 'Ack'. The comments underneath the edge annotations may help to explain the model. 'Trans' and 'Ack' use auxiliary variables to model failure, the loss, or the duplication of messages. They are initialized by the first two actions. A time out can occur at any moment.

5.2 B(PN)^2 semantics

The generated B(PN)^2 program, including variable declarations, is shown in the following two figures. It is obvious that the structure of the PFA is preserved.

```
begin
var ack_reply, trans_reply, send_trans, send_ack: chan 0 of {0..1};
 begin          // Ack
   var  ack:     chan 2 of {0..1};
   var  helpa:   {0..1} init 1;
   ⟨ack_reply? = ack!⟩; ⟨helpa' = ack?⟩;
   do   ⟨helpa' = ack?⟩; repeat
     □  ⟨ack_reply? = ack!⟩; repeat
     □  ⟨send_ack! = helpa⟩; repeat
     □  ⟨false⟩; exit
   od
 end
 ||
 begin          // Trans
   var  trans:   chan 2 of {0..1};
   var  helpt:   {0..1} init 1;
   ⟨send_trans? = trans!⟩; ⟨helpt' = trans?⟩;
   do   ⟨helpt' = trans?⟩; repeat
     □  ⟨send_trans? = trans!⟩; repeat
     □  ⟨trans_reply! = helpt⟩; repeat
     □  ⟨false⟩; exit
   od
 end
 ||
 begin          // Send
   do   ⟨send_trans! = 1⟩;
        do   ⟨send_ack? = 0⟩; repeat
          □  ⟨send_trans! = 1⟩; repeat
          □  ⟨send_ack? = 1⟩; exit
        od; ⟨send_trans! = 0⟩;
        do   ⟨send_ack? = 1⟩; repeat
          □  ⟨send_trans! = 0⟩; repeat
          □  ⟨send_ack? = 0⟩; exit
        od; repeat
     □  ⟨false⟩; exit
   od
 end
 ||
```

Figure 5.2.2 : First part of the generated B(PN)^2 program

```
begin          // Reply
  do    ⟨ack_reply! = 1⟩;
        do    ⟨trans_reply? = 1⟩; repeat
          □   ⟨ack_reply! = 1⟩; repeat
          □   ⟨trans_reply? = 0⟩; exit
        od; ⟨ack_reply! = 0⟩;
        do    ⟨trans_reply? = 0⟩; repeat
          □   ⟨ack_reply! = 0⟩; repeat
          □   ⟨trans_reply? = 1⟩; exit
        od; repeat
     □  ⟨false⟩; exit
  od
 end
end
```

Figure 5.2.3 : Second part of the generated $B(PN)^2$ program

6 Conclusion

An algorithm has been presented that compiles a $B(PN)^2$ specific PFA into a $B(PN)^2$ program. This compiler adds a new input interface to the **PEP** tool. As a consequence the whole functionality of the tool can be used to analyse the PFA, e.g. the integrated model checking algorithm [4] can be used to check various properties.

The presented algorithm covers the whole expressiveness of the *do_od* construct. Every PFA that can be compiled in a useful way is accepted, the other ones are rejected. It has been the main goal to accept as many PFA as possible without producing non-optimal $B(PN)^2$ code.

At the present time it is only possible to use the parallel operator on the highest level by having a parallel composition of FA. This drawback can be overcome elegantly if the PFA model is extended by a new kind of edges having one node in the preset and more than one in the postset or vice versa.

References

[1] K. Bartlett, R. Scantlebury, and W. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5), pages 260 – 261, 1969.

[2] E. Best, H. Fleischhack, W. Fraczak, R. P. Hopkins, H. Klaudel, and E. Pelz. An M-net semantics of $B(PN)^2$. In J. Desel, editor, *Structures in Concurrency Theory*, Workshops in Computing. Springer, 1995.

[3] Eike Best and Richard Pinder Hopkins. $B(PN)^2$ – a Basic Petri Net Programming Notation. In A. Bode, M. Reeve, and G. Wolf, editors, *Proceedings of PARLE '93*, volume 694 of *Lecture Notes in Computer Science*, pages 379 – 390. Springer-Verlag, 1993.

[4] Javier Esparza. *Model Checking Using Net Unfoldings.* Number 23 in Science of Computer Programming, pages 151–195. ELSEVIER, 1994.

[5] Bernd Grahlmann. An Introduction to the Principles, the Functionality and the Use of the PEP System. In Eike Best and Hans Fleischhack, editors, *PEP: Programming Environment Based on Petri Nets*, Hildesheimer Informatik-Berichte 14/95. Universität Hildesheim, 5 / 95.

[6] Bernd Grahlmann. PEP: A Programming Environment based on Petri Nets. Tool Presentations of ATPN'95, 6 / 95.

[7] John E. Hopcraft and Jeffrey D. Ullmann. *Introduction to Automata Theory, and Languages, and Computation.* Addison Wesley, 1994.

[8] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.