Profiting from Spin in PEP*

Bernd Grahlmann[1] and Carola Pohl[2]

[1] Fachbereich Informatik, Universität Oldenburg,
Postfach 2503, D-26111 Oldenburg,
grahlmann@informatik.uni-oldenburg.de,
http://theoretica.informatik.uni-oldenburg.de/~bernd
[2] SerCon, Mainz,
carola.pohl@sercon.de

Abstract. This paper describes how the PEP tool (Programming Environmnet based on Petri nets) profits from an integration of the Spin (Simple PROMELA INterpreter) verification package.
Translation methods from three input formalisms (parallel programs, high-level and low-level Petri nets) into PROMELA (PROtocol MEta LAnguage) are discussed and the Spin based verification is compared with a Petri net based partial order model checker using a number of typical examples.

Keywords: Parallel programs, PEP, Petri nets, PROMELA, SDL, Spin, Verification.

1 Introduction

The PEP tool [3, 5, 14] is a tool for the modelling, simulation, analysis and verification of parallel systems. A number of compilers provide the automatic generation of low-level Petri nets [4] from other input formalisms:

- SDL (Specification Description Language) [8]
- parallel finite automata (PFA) [16],
- parallel programs written in B(PN)^2 [6],
- process algebra terms expressed in the PBC [4], and
- high-level Petri nets [2].

Apart from simulation, also analysis and verification have been based on Petri net theory. Originally, the core of the verification component of PEP was a partial order based model checker [10, 17] which takes the finite prefix of a branching process (an optimised version of the McMillan unfolding [22, 11]) of the low-level Petri net as input. Efficiency was partly gained on the cost of the expressiveness of the supported logic, S_4, a propositional logic on place names, augmented with $\Box$ for 'always' (`AG` in CTL) and $\Diamond$ for 'possibly at some future point' (`EF` in CTL).

* PEP is a joint project between the 'Universität Hildesheim' the 'Carl von Ossietzky Universität Oldenburg' and the 'Humboldt-Universität zu Berlin' which is financed by the DFG (German Research Foundation).

In 1996 we decided to extend the verification component. The main motivations have been:

– to have support for stronger logics, such as CTL (Computational Tree Logic [9]) and LTL (Linear Time temporal Logic [24]), and
– the conviction that there is no verification method which is always superior.

This paper focuses on the integration [25] of the Spin verifier [18, 21] into the PEP tool. In section 2, we describe the translation of low-level Petri nets into PROMELA. Tool oriented aspects of the integration are discussed in section 3. Section 4 shows how parallel systems modelled in three different input formalisms can be verified using the new Spin interface. Expressiveness and efficiency are the major criteria for a comparison with the prefix based model checking approach. Interesting issues of a translation from high-level Petri nets into PROMELA are presented in section 5 and a translation from $B(PN)^2$ programs into PROMELA is analysed in section 6. Finally, we conclude in section 7 and give a list of references.

2 Translating low-level Petri nets into PROMELA

As mentioned in the introduction, low-level Petri nets play a major role within the PEP tool. G. J. Holzmann already presented an approach for the translation of low-level Petri nets into PROMELA programs in [19]. We will now describe similar solutions.

2.1 1-safe Petri nets

The most simple class of low-level Petri nets (1-safe[1] nets without arc weights) are particularly interesting within the PEP framework because $B(PN)^2$ programs and SDL specifications can be automatically compiled into 1-safe Petri nets [12, 13]. For this reason, it is straightforward to consider these nets first.

Fig. 1 shows a small example. A Petri net consists of places (such as P1, P2, P3 and P4) which are marked with tokens (representing resources) and transitions (such as T1, T2, T3 and T4) which resemble actions. Arcs connect places and transitions. They determine the resources which are consumed (e.g., T2 consumes a token from P1 and P2) and produced (e.g., T2 produces a token on P3 and P4) by the occurrence of a transition.

In Fig. 2 a translation of the 1-safe low-level net (from Fig. 1) is shown. It can be seen that:

– one variable of type `bool` is introduced for each place,
– a `proctype Net` is given for the net itself, and
– each transition is modelled by an `atomic` sequence.

In comparison with the solution of G. J. Holzmann we mention four differences:

[1] 1-safe means that each place may contain at most one token.

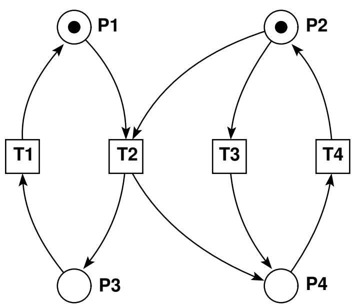

Fig. 1. 1-safe low-level Petri net example.

```
/* 1-safe */
bool P1=1;
bool P2=1;
bool P3=0;
bool P4=0;
bool DEADLOCK=0;

proctype Net()
{
   do
   :: atomic { P3 -> P3=0; P1=1;  }
   :: atomic { P1 && P2 -> P1=0; P2=0; P3=1; P4=1;  }
   :: atomic { P2 -> P2=0; P4=1;  }
   :: atomic { P4 -> P4=0; P2=1;  }
   :: else -> goto dead
   od;
dead: DEADLOCK=1
}

init{run Net()}
```

Fig. 2. PROMELA program for the example of Fig. 1

1. In the case of a 1-safe Petri net the type `bool` is sufficient.
2. The initialisation of the variables for the places prevents undesired verification results.
3. The atomic sequences for the transitions first check that the input places are marked, then set the variables for all input places to `0` and finally set the variables for all output places to `1`.
4. We introduced an auxiliary variable `DEADLOCK` and an additional `else` branch which covers the case that no transition is enabled and thus a deadlock occurred.

Fig. 3 shows the general result of a translation of a 1-safe Petri net with the places P1, ..., Pn and the transitions T1, ..., Tm. We use the following notations:

In(Ti) $\hat{=}$ Set of input places P_{i_1}, ..., P_{i_l} of Ti ($i \in \{1, \cdots, m\}$)
Out(To) $\hat{=}$ Set of output places P_{o_1}, ..., P_{o_t} of To ($o \in \{1, \cdots, m\}$)
Ex(In(Ti)) $\hat{=}$ (P_{i_1} && ... && P_{i_l})
Clear(In(Ti)) $\hat{=}$ P_{i_1}=0; ...; P_{i_l}=0
Set(Out(To)) $\hat{=}$ P_{o_1}=1; ...; P_{o_t}=1

```
bool P1=...;
...
bool Pn=...;
bool DEADLOCK=0;

proctype Net()
{
  do
  :: atomic {  Ex(In(T1)) -> Clear(In(T1)); Set(Out(T1)); }
  ...
  :: atomic {  Ex(In(Tm)) -> Clear(In(Tm)); Set(Out(Tm)); }
  :: else -> goto dead
  od;

dead: DEADLOCK=1
}

init {  run Net() }
```

Fig. 3. PROMELA–pseudocode for a 1-safe low-level Petri net

2.2 Non-safe Petri nets

Fig. 4 shows an example of a low-level Petri net which does not belong to the class of nets considered so far. Already the initial marking shows that, e.g., place P1 may be marked with more than one token. Moreover, some arcs have a weight of more than one.

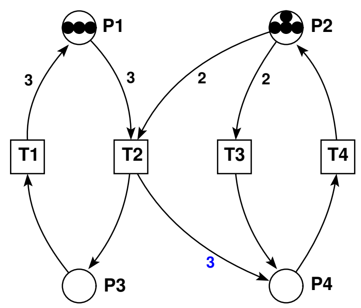

Fig. 4. General low-level Petri net example.

The result of a translation of such a general low-level Petri net (with the places P1, ..., Pn and the transitions T1, ..., Tm) is given in Fig. 5. The arc weights are denoted as follows:

W(P,T) $\hat{=}$ weight of arc from place P to transition T.
W(T,P) $\hat{=}$ weight of arc from transition T to place P.

```
byte P1=...;
...
byte Pn=...;
bool DEADLOCK=0;

proctype Net()
{
   do
   :: atomic { (∀ P_in∈In(T1): (P_in>=W(P_in,T1)) ) ->
                 (∀ P_in∈In(T1): P_in=P_in-W(P_in,T1);)
                 (∀ P_out∈Out(T1): P_out=P_out+W(T1,P_out);) }
   ...
   :: atomic { (∀ P_in∈In(Tm): (P_in>=W(P_in,Tm)) ) ->
                 (∀ P_in∈In(Tm): P_in=P_in-W(P_in,Tm);)
                 (∀ P_out∈Out(Tm): P_out=P_out+W(Tm,P_out);) }
   :: else -> goto dead
   od;

dead: DEADLOCK=1
}

init { run Net() }
```

Fig. 5. PROMELA–pseudocode for a low-level Petri net

A number of changes are necessary:

1. The type `byte` replaces the type `bool`.
2. The atomic sequences for the transitions first check that each input place is (at least) marked with the number of tokens given by the corresponding arc weight, then decrement the variables for all input places by the corresponding arc weight, and finally increment the variables for all output places appropriately.

3 Tool oriented aspects

In the previous section we explained the translation(s) of low-level Petri nets into PROMELA programs which provide the basis for an integration of the Spin verification package into the PEP tool. However, a user-friendly solution requires several more efforts. Typically, the user appreciates the following approach:

– (s)he edits (in a user friendly way) an LTL formula,
– (s)he chooses the verification options,
– (s)he pushes a button,
– (s)he gets the verification result, and
– maybe (s)he simulates the error trail in the Petri net editor (alternatively in the B(PN)^2 or SDL editor).

We discuss the relevant steps in more detail in the subsequent subsections.

3.1 Never claim generation

In principle, PEP not only allows the verification of Petri net formulae, but also the verification of B(PN)^2 or SDL properties by translating them (transparently) into Petri net formulae. Nevertheless, for the purpose of this paper it is sufficient to focus on the steps which are necessary to transform a Petri net formula into a never claim. We (again) distinguish the cases of a 1-safe or non-safe net.

1-safe net

Within an LTL formula for the verification of a 1-safe Petri net, place names, `true` and `false` as well as `DEADLOCK` may be used as predicates. Thus, the formula `[](P1 -> (<>(P2 || DEADLOCK)))` is an example for the net of Fig. 1. In order to be able to use Spin for the never claim generation, we convert the place names (which are hence restricted to begin with a capital letter) into lower case. The result of an invocation of Spin – such as

```
spin -f "[](p1 -> (<>(p2 || deadlock)))"
```

is then added, together with defines like

```
#define p1 P1
#define deadlock DEADLOCK,
```

to the PROMELA program for the Petri net. All these steps are performed transparently for the user.

non-safe net

In this case, the user typically wants to use predicates such as `P3 >= 1` or `P2 == 3` in formulae like

```
[](!((P3 >= 1) && (P2 == 3))).
```

In order to support formulae of this kind, these predicates are extracted and replaced by predicate names `p1`, `p2`, ... which are conform with Spin. Thus, this time Spin may be invoked as follows:

```
spin -f "[](!((p1) && (p2)))"
```

and the result is then added, together with defines like

```
#define p1 (P3 >= 1)
#define p2 (P2 == 3),
```

to the PROMELA program for the Petri net.

3.2 Spin verification options

The Spin verification package supports a modest number of compile-time as well as run-time options. We have chosen to provide the user with an interface to these options which is almost identical to the interface provided by Xspin (the graphical user interface for the Spin verifier). Like this, the user may, for instance, switch from *exhaustive* to *supertrace* mode, decide whether the verification is stopped after a certain number of errors, or restrict the search depth. The C code generation, the compilation of an executable and the run of the verifier are then started in the same way as within Xspin.

3.3 Spin result transformation

Two post calculation steps are necessary to transform the result of the verification run.

Verification result extraction

In a first step, the overall result is determined. Fig. 6 shows the corresponding decision procedure for which the output of the verifier is scanned for various patterns.

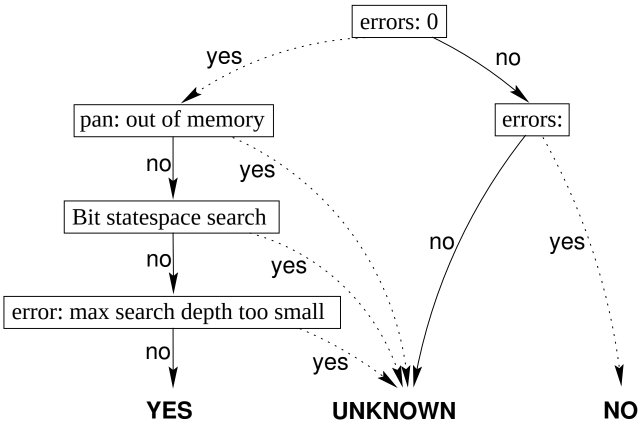

Fig. 6. Verification result extraction scheme

Error trace extraction

In a second step, the error trail (which is produced by the verifier in the case that the answer is no) is analysed. For this purpose the Spin simulator is invoked with the options `-p -t` and the output is redirected into a file. An auxiliary program uses this output together with some additional information (the compiler from Petri nets into PROMELA produces a list which summarises for each transition the line in the PROMELA file and the number of steps in the corresponding atomic sequence) in order to generate a Petri net transition sequence.

_SEQUENCE:
T1,T2,T4
_LOOP:
T1,T3
is a typical result which may then be simulated in the low-level Petri net editor (or optionally in the B(PN)^2 or SDL editor).

4 Examples

In this section we discuss benefits and drawbacks of the described integration of Spin in PEP by considering three different examples.

4.1 Production control example (with Petri nets)

The first example deals with production planning systems as proposed by J. von Steinaecker in [26]. Fig. 7 shows a part of the Petri net based modelling. It focuses on one machine within the manufacturing process. The bottom part displays that the machine may perform three different tasks (Process_A, Process_B and Process_C). Each task requires pre- as well as post-preparation steps (e.g., Pre_A), consumes resources (e.g., from S1) and produces outcome (e.g., on S5). Moreover, additional actions are necessary to change from one process to another (e.g, Change_AB).

A whole system is composed of machines of various types. One of the main questions is whether it is possible to produce a certain amount of products given a certain amount of resources. Furthermore, optimal scheduling plans are requested.

We have chosen this example because it shows one strength of the Spin verifier. In contrast to the prefix based model checker not only 1-safe low-level nets, but also more general bounded nets may be analysed. In the given example, the amount of resources of a certain type may thus be modelled using that number of black tokens.

Of course, the system may also be modelled differently. Fig. 8 shows that high-level Petri nets provide an alternative way to model, e.g., a part of the component shown in Fig 7:

- The places for the resources and products now have subranges of cardinals as type (e.g., $\{0..5\}$) and carry the number of resources as token.
- The adjacent arcs are now inscribed with variables (such as 'S1 or S1').
- The transitions which consumes or produces resources are inscribed with occurrence conditions (like S1'='S1-1 & S2'='S2-1 & S5'='S5+1), which determine how many resources are consumed or produced.

However, this produces considerably more complex low-level Petri nets. For instance, the place S1 is unfolded into six low-level places and the transition Process_A into $5 * 5 * 5 = 125$ low-level transitions.

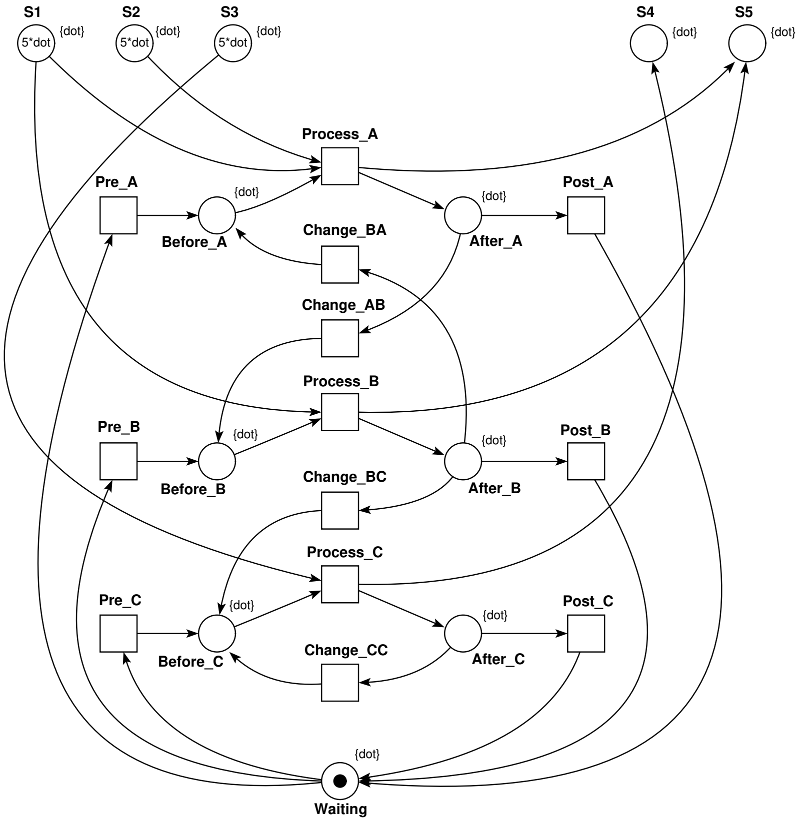

Fig. 7. Production planning example.

As a consequence, verification becomes infeasible with the prefix based model checker because already the calculation of the finite prefix of the branching process is impossible for reasonably sized production systems. As an example, we give the results for a system with three different types of machines:

- The calculation of a reduced prefix (without configuration[2] information) took approximately 80 minutes[3]. This reduced prefix contained almost 52,000 cut-off events. Taking the intermediate results of the first hours of the calculation of the full prefix into account we can estimate a number of more than 1,000,000 configurations. This is (at least for the time being) far beyond the limits of the algorithm.
- We were able to verify a number of reachability problems (i.e., whether or not a certain number of products may be produced with a given number of resources) using the integration of the Spin verifier. In the cases where the

[2] We refer the interested reader to [11] for the details of the prefix calculation.
[3] All tests have been performed on a Pentium II 266 MHz with 128 MB memory.

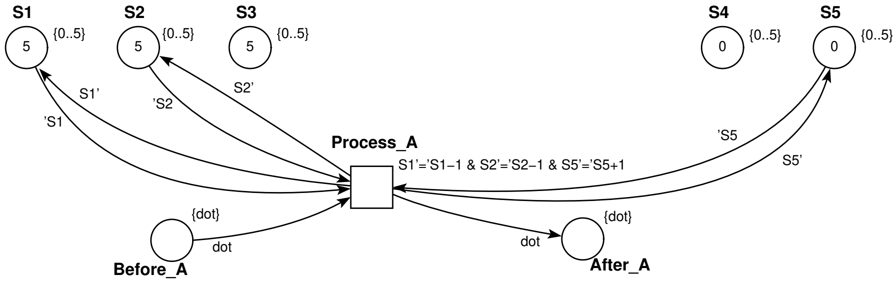

Fig. 8. Production planning example (alternative solution).

products were producable, the verification took between 4 and 10 seconds (including the generation of the verifier). Otherwise, the whole state spaces (up to 3,430,620 states) had to be exploited, which took up to 26 minutes.

4.2 Communication protocol example (with SDL)

As a second example we use an SDL specification in order to show that:

- The integration of Spin in PEP also extends the field of applications which may be verified with Spin. This is particularly interesting because the early work on SDL verification [20] was not continued.
- The expressive power of LTL formulae which is provided by the Spin verifier is an interesting extension of the prefix based model checking approach.

We have chosen a simple ARQ (Automatic Repeat reQuest) communication protocol with alternating acknowledgement (which was taken from [13]). The top-level specification in Fig. 9 shows that the system comprises processes of two different types (up to two instances of the Client process and one instance of the Server process) which exchange signals with parameters 0 and 1 via two signal routes, a and d.

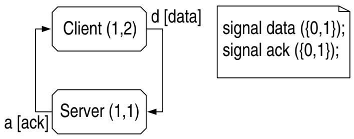

Fig. 9. The ARQ-System.

Without going too much into the details, we mention that each Client instance is responsible for the transmission of exactly one data package. Using the procedure SendPackage this package is send until the correct acknowledgement

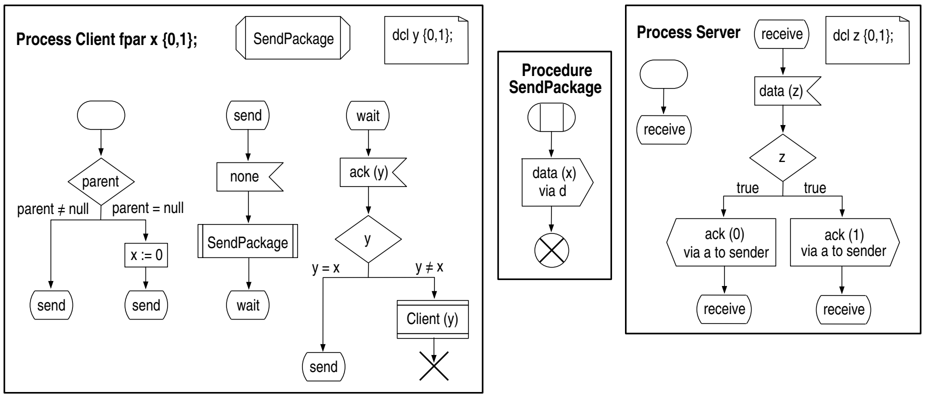

Fig. 10. The *Client* and the *Server* Process.

is received. Then, a new Client instance is created and the instance terminates itself.

From the verification point of view there are two interesting issues:

1. The specification contains a non-trivial deadlock which is related to the non-atomicity of the SDL state transition which first creates (or better tries to create) a new Client instance and then terminates without checking whether the process creation was successful. This deadlock may be found using the prefix based approach within 0.1 second (including the calculation of the prefix) and with the Spin verifier within approximately 0.1 second plus 9.3 seconds for the generation of the verifier.
2. The Server process non-deterministically returns either the correct or a wrong acknowledgement. This means that (without fairness assumptions) it is not ensured that a data package is *eventually* acknowledged correctly. Properties of this kind cannot be expressed in the logic which is supported by the prefix based model checker. Using the integrated Spin verifier we were able to detect this behaviour within approximately 6.9 seconds (including 6.7 seconds for the generation of the verifier). As mentioned above the integration offered the possibility to simulate the corresponding failure trace in the SDL editor.

4.3 Alternating-bit example (with parallel programs)

As a last example we consider a modelling of the 'alternating-bit' protocol [1] in the parallel programming language $B(PN)^2$. We have chosen this example in order to illustrate potential impacts of scaling on the efficiency of the different verification approaches. Furthermore, we use it to prepare the discussion in section 6.

This protocol has been designed to ensure reliable transmission of messages even if the communication media looses or duplicates messages or acknowledgements. Fig. 11 shows that the $B(PN)^2$ program comprises four named blocks, i.e. one for each process:

```
begin
 const CAPACITY 3;
 var send_trans, trans_reply, ack_reply, send_ack: chan 0 of {0..1};
   begin  Send
     do  ⟨send_trans! = 1⟩;                 // action 1  Send message
         do ⟨send_ack? = 0⟩; repeat         // action 2  Receive superfluous ack
         □  ⟨send_trans! = 1⟩; repeat       // action 3  Send message again
         □  ⟨send_ack? = 1⟩; exit           // action 4  Receive correct ack
         od; ⟨send_trans! = 0⟩;             // action 5  Send message
         do ⟨send_ack? = 1⟩; repeat         // action 6  Receive superfluous ack
         □  ⟨send_trans! = 0⟩; repeat       // action 7  Send message again
         □  ⟨send_ack? = 0⟩; exit           // action 8  Receive correct ack
         od; repeat
     od
   end    Send
 || begin  Trans
   var   trans:     chan CAPACITY of {0..1};
   var   helpt:     {0..1} init 1;
     ⟨send_trans? = trans!⟩;                // action 9  Receive first message
     ⟨helpt' = trans?⟩;                     // action 10 Initialise auxiliary variable
     do ⟨helpt' = trans?⟩; repeat           // action 11 Forget message
     □  ⟨send_trans? = trans!⟩; repeat      // action 12 Receive new message
     □  ⟨trans_reply! = helpt⟩; repeat      // action 13 (Re-)transmit message
     od
   end    Trans
 || 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
     od
   end    Reply
 || begin  Ack
   var   ack:       chan CAPACITY 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
     od
   end    Ack
end
```

Fig. 11. Modelling the alternating bit protocol with B(PN)^2

1. Send tries to send an alternating sequence of '0' and '1' messages;
2. Reply acknowledges them with '0' and '1' acknowledgements, respectively;
3. Trans is the communication media from Send to Reply; and
4. Ack is the communication media from Reply to Send.

Instead of explaining the details of the algorithm (the comments in the program should be self-explanatory), we only mention some interesting features of the language B(PN)^2:

– B(PN)^2 supports (nested) parallelism.
– A general **do-od** construct combines usual loop and if-then-else features with non-determinism.
– Channels (as well as stacks) with arbitrary capacity are provided (a full channel blocks).
– Variables can be accessed in three different ways – post primed (value after execution), preprimed (before) and unprimed (before equals after). The type of a variable is always respected.
– Atomic actions (within $\langle\rangle$ brackets) may change an arbitrary number of variables at the same time. Moreover, multiway synchronisations and non-determinism is supported. Each atomic action can also be seen as a condition.

We used prefix based algorithms (a special purpose deadlock-checker [23] and the model checker) as well as the Spin LTL verifier to check some properties of the alternating-bit protocol. Table 1 and Table 2 summarise the results. We briefly mention some interesting observations:

– Low-level Petri nets are not appropriate to model channels. As a consequence, more efforts should be taken to include special objects for channels into low-level nets and to handle them differently within the verification algorithms (cf. [7]).
– Some liveness properties (whether it is always 'possible' to reach a state, or whether a state is always 'eventually' reached) are only expressible in one of the supported logics.
– In the given example, the state spaces as well as the complexity of the prefix tends to grow too quickly. Starting from a capacity of 8 (for the buffers *trans* and *ack*), this has the consequences that, the construction of the prefix (and thus all model checking) becomes too complex, and Spin verifications which have to exploit the whole state space also become infeasible. (Note that the deadlock-freeness test for a capacity of 7 was not possible – even with memory reduction methods – using 256 MB of memory).
– Prefix based methods are superior for the detection of non-reachable states (in this case Spin needs as long as for the deadlock-freeness check), whereas the Spin on-the-fly verification performs dramatically better in cases where an error can be found early.

Capacity	Prefix					Deadlock	Reachability	'possible' Liveness
	C	E	Cutoffs	Configs	Time			
1	1308	471	186	153	0.1	2.3	0.1	10.5
2	3326	1200	511	537	1.0	58.2	0.1	303.7
3	7223	2604	1159	1411	6.4	639.4	0.4	10923.0
4	13485	4851	2185	2886	38.5	11146.0	1.0	
5	22995	8253	3734	5219	193.4		2.4	
6	36682	13133	5954	8693	883.1		5.8	
7	55609	19864	9012	13600	3870.3		12.3	

Table 1. Alternating bit protocol: prefix based analysis

Capacity	Deadlock			Reachability	'eventually' Liveness
	States	Depth	Time		
1	4978	2863	3.7	3.5	3.6
2	12102	8573	4.4	3.6	3.8
3	35914	39721	6.0	3.9	4.0
4	126142	157861	14.4	4.1	4.2
5	491826	756193	42.4	4.3	4.4
6	2022500	3327829	328.4	4.4	4.6
7	8518430	14723059	1200.0	4.6	4.7

Table 2. Alternating bit protocol: Spin based analysis

5 Translating high-level Petri nets into PROMELA

So far, we have described how a relatively simple translation of low-level Petri nets into PROMELA allows the verification of input formalisms (which are supported by PEP) using the Spin verifier. However, it is straightforward to look for translations of more high-level formalisms into PROMELA, in order to improve efficiency. This takes into account that each translation step introduces some overhead. It is natural to consider high-level Petri nets first, because parallel programs as well as SDL specifications are translated via high-level Petri nets into low-level Petri nets.

Within PEP a special class of high-level Petri nets, M-nets (for modular multilabelled nets) [2], is used. We briefly summarise the most relevant characteristics:

- Places have a type which may be a cross-product of sets which contain integers, booleans or the black token.
- Transitions are annotated with an occurrence condition.
- Arcs are inscribed with multisets of variables.

Fig. 12 shows a small net which may serve to give an intuition and to stress some of the features which impose problems for the translation into PROMELA:

– It is difficult to model the various types of places appropriately. Sets like {dot,3,6} and cross-products impose severe problems as far as, e.g, type checking is concerned.
– An occurrence condition may comprise almost arbitrary conditions. Non-determinism is one of the major problems.
– Bindings for the variables, or even tuples, on arcs have to respect the type of output places and are restricted by the available tokens on input places.

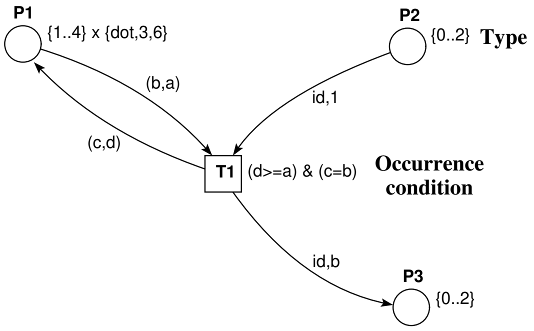

Fig. 12. A small high-level net example.

In Fig. 13 we give a PROMELA pseudocode translation for a general high-level Petri net. As for low-level Petri nets, there is (within a loop) one atomic sequence for each transition. This time the entrance is controlled by a single variable $enabled\_transition_i$.

A key problem is that all possible bindings (of all transitions) have to be checked, i.e. an almost general high-level Petri net simulator must be encoded in PROMELA.

For this purpose we introduce one variable $enabled\_binding_{i_j}$ for each binding of transition i. An inner loop tries all these bindings until either an enabling binding is found or all bindings have been tested without success. A binding is enabling, if the occurrence condition evaluates to **true**, the input places are marked sufficiently and the types of the output places are respected. If no enabling binding for the transition under consideration was found, the transition is marked as not enabled (in this step). Otherwise, the transition is fired and all transitions are marked as potentially enabled for the next step. Finally, all intermediate bindings are reset and all bindings are marked as potentially enabled for the next check.

In principle, it is possible to translate high-level Petri nets this way. However, some tests have shown that, on the one hand an implementation is tedious and on the other hand this general solution is not appropriate to improve efficiency.

```
proctype Net()
{
 Initialisation
 ...
 do
 ::atomic {enabled_transition_1
            -> do
               ::enabled_binding_1_1
                    -> Bind variables;
                       enabled_binding_1_1=0
               :
               ::enabled_binding_1_p
                    -> Bind variables;
                       enabled_binding_1_p=0
               od
               unless {  ( Occurrence condition true &&
                           Check tokens on input places &&
                           Respect types of output places ) ||
                         ( !(enabled_binding_1_1 || ... ||
                              enabled_binding_1_p ) };
               if
               ::!(enabled_binding_1_1 || ... || enabled_binding_1_p )
                    -> enabled_transition_1=0
               ::else
                    -> Fire transition;
                       enabled_transition_1= ... =enabled_transition_m=1
               fi;
               Reset all variable bindings;
               enabled_binding_1_1= ... =enabled_binding_1_p=1;
         }
 :
 ::atomic {enabled_transition_m
            :
         }
 ::!(enabled_transition_1 || ... || enabled_transition_m) -> goto dead
 od;
 dead: deadlock=1
}
```

Fig. 13. PROMELA pseudocode for a general high-level Petri net

6 Translating B(PN)^2 programs into PROMELA

In the previous section we have seen that a translation of high-level Petri nets into PROMELA imposes a number of problems. In this section we will see that a compilation of B(PN)^2 programs is even more difficult. Some of the main problems are related to the distinguishing features of B(PN)^2 (compared with PROMELA) which have already been summarised in subsection 4.3.

Let us first consider an atomic action $\langle$*c1! = x'* $\rangle$. This action is executable if there is a value in the type of the variable x which can be sent to the channel *c1*. A translation into PROMELA must contain a type check and it must cover the sending of all possible values which respect both types (i.e., it must generate random values).

The situation becomes more complicated if channel inputs are engaged. In this case, the first value which is stored in the channel must, e.g., be compared with the type of a variable. This means, that a translation of B(PN)^2 channels into PROMELA channels is at least complicated, because there is no test mechanism which does not remove a value from the channel. Arrays may overcome this problem.

Synchronous communication via channels with capacity zero is another difficult issue. In B(PN)^2 an action $\langle$*c0? = 'x* $\rangle$ may be the guard of an alternative. Such an action may synchronise with an arbitrary number of (different) send actions. Multiway synchronisation is possible as well. This means that a PROMELA program must check at run-time whether a synchronisation is possible. These checks imply that variables which are used within the corresponding actions are changed. In the case that such a check gives the result that the synchronisation was not possible, the original state must be recovered. This includes the values of variables as well as the positions in the control flow.

Moreover, the fact that the verification of the PROMELA code should yield correct model checking results, demands sophisticated introduction of progress labels and fairness assumptions in order to avoid infinite test sequences.

Probably the most severe problem is to give a solution which covers the possibility to change an arbitrary number of variables within one single action. We refer the reader to [25] where a detailed analysis of the problems is given, a restriction of the language B(PN)^2 is proposed, and a PROMELA semantics for the restricted language is defined.

Finally, we mention that B(PN)^2 offers a procedure concept which includes value, value-result and reference parameters and supports also recursion [12]. Covering this in full generality (and full compatibility) is a problem which is at least out of the scope of this paper.

7 Conclusion

This paper briefly described a transparent integration of Spin into PEP which is based on a translation from Petri nets into PROMELA. We used three examples to demonstrate the main benefits of this approach:

– Spin may now be applied to $B(PN)^2$ programs, SDL specifications, parallel finite automata, process algebra terms, high-level and low-level Petri nets.
– PEP gained an LTL model checker.
– As no verification method is always superior, PEP can thus offer more efficient verification for a number of applications.

For a more detailed overview of the PEP system we refer the reader to [15] and the various papers which are available together with the tool at

http://theoretica.informatik.uni-oldenburg.de/∼pep.

Acknowledgement: We would like to thank Michael Kater, Stephan Ptak and Stefan Schwoon for their contributions to the integration of Spin in PEP.

References

1. K. Bartlett, R. Scantlebury, and W. Wilkinson. A Note on Reliable Full-duplex Transmission over Half-duplex links. *Comunications of the ACM*, 12(5):260 – 261, 1969.
2. E. Best, H. Fleischhack, W. Frączak, R. P. Hopkins, H. Klaudel, and E. Pelz. A Class of Composable High Level Petri Nets. In G. De Michelis and M. Diaz, editors, *Proceedings of ATPN'95 (Application and Theory of Petri Nets), Torino*, volume 935 of *Lecture Notes in Computer Science*, pages 103–118. Springer, June 1995.
3. E. Best. Partial Order Verification with PEP. In G. J. Holzmann, D. Peled, and V. Pratt, editors, *Proceedings of POMIV'96 (Partial Order Methods in Verification)*. Am. Math. Soc., 1996.
4. E. Best, R. Devillers, and J. G. Hall. The Box Calculus: a New Causal Algebra with Multi-Label Communication. In G. Rozenberg, editor, *Advances in Petri Nets 92*, volume 609 of *Lecture Notes in Computer Science*, pages 21 – 69. Springer-Verlag, 1992.
5. E. Best and B. Grahlmann. PEP: Documentation and User Guide. Universität Hildesheim., November 1995. Availabe together with the tool via: http://www.informatik.uni-hildesheim.de/∼pep.
6. E. Best and R. P. 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.
7. B. Boigelot and P. Godefroid. Symbolic Verification of Communication Protocols with Infinite State Spaces Using QDDs. In R. Alur and T. A. Henzinger, editors, *Proceedings of CAV'96 (Computer Aided Verification), New Brunswick*, volume 1102 of *Lecture Notes in Computer Science*, pages 1 – 12. Springer-Verlag, 1996.
8. CCITT. *Specification and Description Language, CCITT Z.100, Geneva*. International Consultative Committee on Telegraphy and Telephony, 1992.
9. E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, 1982.
10. J. Esparza. *Model Checking Using Net Unfoldings*, pages 151 – 195. Number 23 in Science of Computer Programming. ELSEVIER, 1994.
11. J. Esparza, S. Römer, and W. Vogler. An Improvement of McMillan's Unfolding Algorithm. In T. Margaria and B. Steffen, editors, *TACAS'96 (Tools and Algorithms for the Construction and Analysis of Systems)*, volume 1055 of *Lecture Notes in Computer Science*, pages 87 – 106. Springer-Verlag, 1996.

12. H. Fleischhack and B. Grahlmann. A Petri Net Semantics for $B(PN)^2$ with Procedures. In *Proceedings of PDSE'97 (Parallel and Distributed Software Engineering), Boston MA*, pages 15 – 27. IEEE Computer Society, May 1997.
13. H. Fleischhack and B. Grahlmann. A Compositional Petri Net Semantics for SDL. In *Proceedings of ATPN'98 (Application and Theory of Petri Nets)*, Lecture Notes in Computer Science. Springer-Verlag, June 1998.
14. B. Grahlmann. The PEP Tool. In O. Grumberg, editor, *Proceedings of CAV'97 (Computer Aided Verification)*, volume 1254 of *Lecture Notes in Computer Science*, pages 440–443. Springer-Verlag, June 1997.
15. B. Grahlmann. Parallel Programs as Petri Nets. Ph.D. thesis. Universität Hildesheim, 1998.
16. B. Grahlmann, M. Moeller, and U. Anhalt. A New Interface for the PEP Tool – Parallel Finite Automata –. In J. Desel, H. Fleischhack, A. Oberweis, and W. Reisig, editors, *Proceedings of AWPN'95 (2. Workshop Algorithmen und Werkzeuge für Petrinetze)*, number 22 in AIS, pages 21–26. FB Informatik Universität Oldenburg, October 1995.
17. B. Graves. Computing Reachability Properties Hidden in Finite Net Unfoldings. In S. Ramesh and G. Sivakumar, editors, *Proceedings of FST&TCS'97 (Foundations of Software Technology and Theoretical Computer Science)*, volume 1346 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
18. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990.
19. G. J. Holzmann. SPIN Verification Examples and Exercises. Bell-Labs., 1996. Availabe via: http://cm.bell-labs.com/cm/cs/what/spin/Man/Exercises.html.
20. G. J. Holzmann and J. Patti. Validating SDL Specifications: An Experiment. In C. Vissers and E. Brinksma, editors, *Proc. 9th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 317 – 326, Twente, Neth., June 1989.
21. G. J. Holzmann and D. Peled. The State of SPIN. In R. Alur and T. A. Henzinger, editors, *Proceedings of CAV'96 (Computer Aided Verification), New Brunswick*, volume 1102 of *Lecture Notes in Computer Science*, pages 385 – 389. Springer-Verlag, June 1996.
22. K. L. McMillan. Automatic Verification Based on Occurence Nets. Technical report, 1992.
23. K. L. McMillan. Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits. In *Proceedings of CAV'92 (4th Workshop on Computer Aided Verification), Montreal – Canada*, pages 164 – 174, Montreal, 1992.
24. A. Pnueli. The temporal logic of programs nets. In *Proceedings of 18*th *IEEE Symposium on Foundations of Computer Science*, pages 46 – 57, 1977.
25. C. Pohl. Integration automatenbasierter Verifikation aus SPIN in PEP. Diplomarbeit, Universität Hildesheim, October 1997.
26. J. von Steinaecker. Petri Net based Modelling, Planning and Control of Logistical Processes under Environmental Goals and Constraints. In G. Morel and F. B. Vernadat, editors, *Proceedings of INCOM'98 (Information Control in Manufacturing) Volume II, Metz, France*, pages 411 – 416, June 1998.