The Reference Component of PEP

Bernd Grahlmann*

ABSTRACT The **PEP** tool is a **P**rogramming **E**nvironment based on **P**etri Nets. Sophisticated programming and verification components are embedded in a user-friendly graphical interface. The basic idea is that the programming component allows the user to design concurrent algorithms in an easy-to-use imperative language, and that the **PEP** system then generates Petri nets from such programs in order to use Petri net theory for simulation and verification purposes.
The main focus of this paper is the reference component which represents the bridge between these two worlds. We integrate references in the formal semantics and present some of the provided features. Among others the simulation of a parallel program can be triggered through the simulation of a Petri net. Program formulae can be transformed automatically into net formulae which can then be an input for the verification component.
PEP has been implemented on Solaris 2.x, SunOS 4.1.x and Linux. Ftp-able versions are available via `www.informatik.uni-hildesheim.de/~pep`.

KEYWORDS B(PN)^2, Model checking, Parallel finite automata, **PEP**, Petri nets, Reference component, Simulation, Temporal logic, Tool.

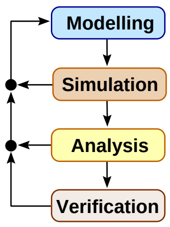


Fig. 1. Development phases.

1 Introduction

The **PEP**[1] tool is a **P**rogramming **E**nvironment based on **P**etri Nets [6]. In order to support the main phases of the development of parallel systems (shown in Fig. 1) it is not sufficient to provide editors for parallel systems, compilers into Petri nets (PN) and simulators as well as analysis and verification algorithms for PN. Only an integrated reference component exploits the full functionality in an adequate way. Users do not have to leave the model they have chosen for the modelling of parallel systems. This paper describes how simulation of a parallel program is triggered through the

*Institut für Informatik, Universität Hildesheim, Marienburger Platz 22, D-31141 Hildesheim, bernd@informatik.uni-hildesheim.de

[1]The PEP project is financed by the DFG (German Research Foundation). This work has been partially supported by the HCM Cooperation Network EXPRESS (Expressiveness of Languages for Concurrency)

simulation of the corresponding PN and how program formulae are transformed automatically into net formulae which are in turn used as an input for the integrated efficient model checker.

This paper is structured as follows. Section 2 describes the **PEP** framework dealing with different types of objects. Parallel finite automata (PFA) and the programming language $B(PN)^2$ (**B**asic **P**etri **N**et **P**rogramming **N**otation) are briefly introduced in sections 3 and 4. The M-net model is presented in more detail in section 5. The most interesting part of the construction of references is presented in section 6 where the M-net semantics for $B(PN)^2$ covering references is given. After presenting an example in section 7, a temporal logic for $B(PN)^2$ is introduced in section 8. The usage of references in the **PEP** tool is depicted in section 9 where new simulation and program verification facilities are explained. Finally, a conclusion and pointers to relevant literature are given.

2 Modelling parallel systems with **PEP**

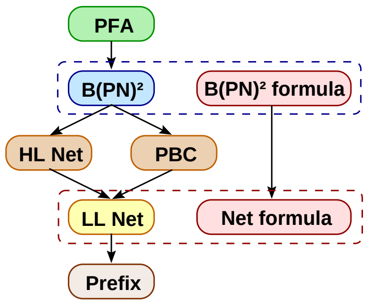


Fig. 2. Objects used by the **PEP** system.

Users can choose between five types of objects in order to model parallel systems (see Fig. 2):

1. Parallel finite automata (PFA) with $B(PN)^2$ actions as arc annotations can be edited and compiled into $B(PN)^2$ programs [11].
2. Parallel algorithms can be expressed in $B(PN)^2$ [7], which is an imperative / predicative programming language.
3. Terms of a process algebra called PBC (**P**etri **B**ox **C**alculus) [2], which is an extension / modification of CCS, can be used. In **PEP**, PBC terms can either be derived automatically from a $B(PN)^2$ program or be designed independently.
4. High-level (HL) PN, called M-nets [5], on which an alternative net semantics of $B(PN)^2$ programs is based [4] can be edited.
5. Arbitrary labelled P/T-nets can be edited. Petri boxes are a special case of low-level (LL) PN, which may arise out of a translation from $B(PN)^2$ programs.

Furthermore, the following objects are used in the **PEP** system:

1. **PEP** allows the definition of a set of temporal logic formulae to allow the user to (model) check a custom designed system property.
2. During verification it may become necessary to calculate the finite prefix of a branching process [8] of an existing LL net. This prefix contains information for model checking [9, 13] a net.

It is up to the user with what kind of object (s)he would like to start the modelling phase. Normally several different objects are created in a modelling cycle. Typically a $B(PN)^2$ program is written, the corresponding M-net and the Petri box are compiled automatically, the prefix is calculated and interesting properties are expressed by formulae. For the purposes of this paper the following five different classes of components are relevant (see Fig. 3):

1. Editors for PFA, $B(PN)^2$ programs, M-nets (HL nets), Petri boxes (LL nets) and formulae.
2. Compilers as follows: PFA $\Rightarrow$ $B(PN)^2$, $B(PN)^2$ $\Rightarrow$ M-net, M-net $\Rightarrow$ Petri box and Petri box $\Rightarrow$ Prefix.
3. Simulators for PFA, $B(PN)^2$ programs, M-nets and Petri boxes.
4. A model checking algorithm [9, 13] for safe PN to determine if a PN satisfies a property given in terms of a temporal logic formula.
5. A reference component, which is a kind of a database server, administers the references between the different objects which are related to one modelling approach. For instance, the compiler $B(PN)^2$ $\Rightarrow$ M-net outputs the relationship between parts of the program (such as actions) and parts of the net (such as transitions) to the reference component. Later on, during the simulation, the M-net simulator may communicate with the reference component in order to request that the $B(PN)^2$ simulator highlights (or executes) an action which corresponds to the currently executed transition.

The interaction between the reference component and the other components is an essential feature of the **PEP** system.

3 PFA

In **PEP** $B(PN)^2$ specific PFA are considered. A PFA is a collection of finite automata (FA) [14] acting in parallel, where one FA corresponds to one process in a program. An FA consists of a start node, a set of local nodes, a set of exit nodes, a set of arcs between these nodes and a labelling function that annotates each arc with a $B(PN)^2$ action. A start node (such as node 1 in Fig. 4) represents the initial state of one process and an exit node (such as node 5) represents a state in which the process has terminated. Thus no outgoing arcs are accepted for exit nodes. In Fig. 4 a PFA consisting of two FA modelling the Peterson algorithm for mutual exclusion of two processes is shown.

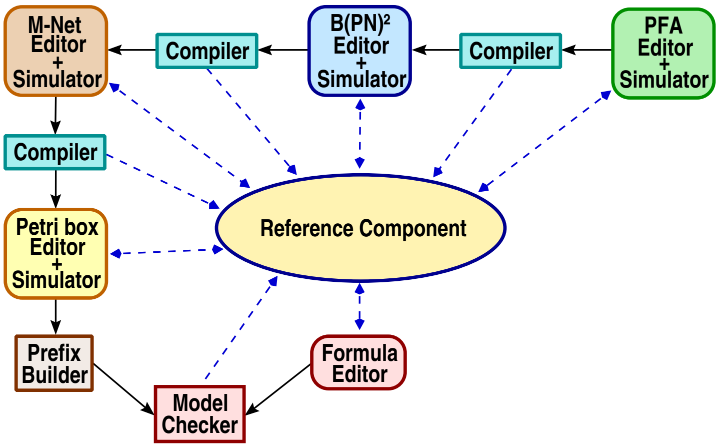


Fig. 3. Interplay between **PEP** system components. The arrows represent input/output (→) or answer/request (⇢) relations between components.

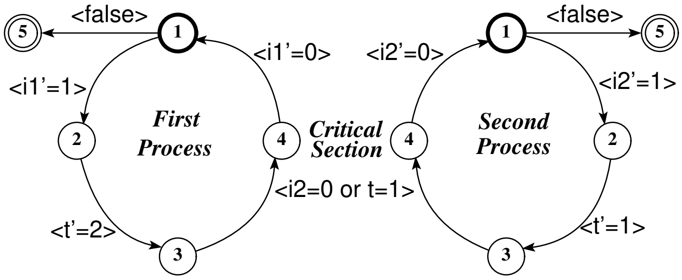


Fig. 4. Peterson algorithm as an PFA.

4 B(PN)^2

B(PN)^2 [7] is an imperative / predicative style parallel programming language whose atomic actions may contain predicates involving the pre- and post-values of variables. Basic command connectives of B(PN)^2 are: sequential composition (;), nondeterministic choice ($\Box$), parallel composition ($\|$), and iteration (**do** ... **od**). Programs are structured into blocks consisting of a declaration part and an instruction part. Processes can share common memory or use channel communication (`c!` denotes writing on a channel and `c?` reading from a channel) or both. The implementation of a procedure concept [10, 15] and abstract data types extend B(PN)^2 to a complete programming language. B(PN)^2 is called **B**asic **P**etri **N**et **P**rogramming **N**otation because it first has been given a compositional semantics in terms of LL PN called Petri boxes [2].

The syntax of B(PN)^2 is depicted in Tab. 1. Not every detail of the language can be explained in this paper. Most features should be selfexplanatory. We may perhaps mention that: k denotes the capacity of the channel or stack; $\langle$expr$\rangle$ denotes an atomic action; and `'v` and `v'` denote pre- and post-values of the variable `v`; `v` implies `'v = 'v` (e.g. $\langle$`x:=x+y`$\rangle$ would be written as $\langle$`x'='x+y`$\rangle$).

prog	::=	block
block	::=	**begin** scope **end**
scope	::=	com | decl; scope
decl	::=	**var** `var-name` : type
type	::=	set | **chan** k **of** set | **stack** k **of** set $\quad (k \in \mathbb{N}_0 \cup \{\infty\})$
com	::=	$\langle$expr$\rangle$ | com $\|$ com | com **;** com |
		do com **enter** alt-set **od** | block
alt-set	::=	com; **repeat** | com; **exit** | alt-set □ alt-set
expr	::=	`v` | `'v` | `v'` | `c!` | `c?` | const | expr op expr | op expr | (expr)
op	::=	`+` | `-` | `*` | `div` | `mod` | `=` | $\neq$ | $<$ | $>$ | $\leq$ | $\geq$ | $\wedge$ | $\vee$ | $\neg$
const	::=	**false** | **true** | z $\quad (z \in \mathbb{N})$

Tab. 1. Syntax of B(PN)^2

5 M-nets

In this section the HL PN model of M-nets (for modular multilabelled nets; cf. [5]) is introduced. We have chosen the M-net model because it allows unfolding (as do most other HL net models) but also composition.

Annotations of places (sets of allowed tokens), arcs (multiset of variables or values or tuples of variables and values), and transitions (occurrence conditions – called value terms) support unfolding into an elementary LL PN. Communication capabilities are denoted by labels of transitions (action terms), while labels of places (called status) denote their interface capabilities. A status can be 'entry', 'exit' or 'internal'. References and data-tags may give the relation to parts of the corresponding B(PN)^2 program.

We exploit the fact that the M-net composition operations – in particular, synchronisation – satisfy various algebraic properties.

5.1 Auxiliary definitions

Let *Val* be a fixed, nonempty and suitably large set of *values*. In our approach it is sufficient to assume that all integers, the Boolean values **true** and **false** and the token $\bullet$ are members of *Val*. Let *Var* be the set of variables (which are interpreted by values of *Val*).

We assume the existence of a fixed but sufficiently large set $\mathbf{A}$ of *action symbols*. The arity $ar(A)$ which is associated with each action symbol $A \in \mathbf{A}$ describes the number of its parameters. The bijection $\bar{\ }: \mathbf{A} \to \mathbf{A}$, called *conjugation*, satisfying $\forall A \in \mathbf{A}$: $\overline{A} \neq A, \overline{\overline{A}} = A$, and $ar(A) = ar(\overline{A})$ groups the elements of $\mathbf{A}$ into pairwise conjugates.

An *action term* is, by definition, a construct $A(\tau_1, \ldots, \tau_{ar(A)})$, where $A \in \mathbf{A}$ and $\tau_j \in Var \cup Val$ for all $1 \leq j \leq ar(A)$. Action terms provide the communication facilities of M-nets. In the definition of the formal semantics we will see that action terms are used to synchronise accesses to variables; and that the resulting nets do not contain action terms.

In addition to the well-known standard definitions given so far, we need some new auxiliary definitions.

We have chosen to base nearly all references (even those of places) on atomic actions and blocks of a program, because these are easier to define than points in the control flow. Therefore, we introduce the function f which maps each atomic action and each block of a B(PN)^2 program to a unique cardinal. A very simple enumeration is sufficient.

$\mathit{Ref}_P = \mathit{Con\text{-}Points} \cup \mathit{Var} \cup \mathit{Val\text{-}Assert}$ is the set containing all possible references which can be related to places. $\mathit{Con\text{-}Points} = \{{}^\circ i, i^\circ | i \in \mathbb{N}\}$ is the set of control points. Intuitively, ${}^\circ f(\mathit{block})$ and $f(\langle \mathit{expr} \rangle)^\circ$ denote 'at begin of' *block* and 'after' $\langle \mathit{expr} \rangle$, respectively. A place of the HL net containing the value of a variable x has a reference 'x'; and $\mathit{Val\text{-}Assert} = \{'x = i\,' | x \in \mathit{Var} \wedge i \in \mathit{Val}\}$ is the set of possible value assertions which can be related (as a reference) to places of the LL net.

$\mathit{Ref}_T = \mathbb{N}$ is the set containing all the possible values of $f(\mathit{block})$ and $f(\langle \mathit{expr} \rangle)$.

The element of the set $\mathit{Data\text{-}Tag} = \{v\}$ is used to mark the places indicating that a variable has a certain value. This data-tag influences for instance the positioning of transitions during synchronisation, and the generation of value assertions during unfolding into an LL net.

5.2 Definition of M-nets

To cover references and data-tags as well, we have to extend the original definition. The main extensions concern the definitions of the $\otimes$ operator and of the basic synchronisation, where the handling of references (but not of data-tags) is introduced.

Definition 5.1 *M-nets*

An *M-net* N is a triple (P, T, ι) such that P is a set of *places*, T is a set of *transitions* with $P \cap T = \emptyset$, and ι is a function with domain $P \cup (P \times T) \cup (T \times P) \cup T$ (called *inscription*) such that:

- For every place $p \in P$, $\iota(p)$ is a tuple $(\lambda_p \mid \alpha_p \mid \varrho_p \mid \psi_p)$, where λ_p is an element of the set $\{e, i, x\}$ called the *(place-)label* or *status*; $\alpha_p \subseteq \mathit{Val}^{\,k}$ (for some $k \in \mathbb{N}$) is a nonempty set called the *place-annotation* or the *type* of p (k is called arity $ar(p)$ of p); $\varrho_p \subseteq \mathit{Ref}_P$ is a set of references, and $\psi_p \subseteq \mathit{Data\text{-}Tag}$ is a set of data-tags.
- For every arc $(p,t) \in (P \times T)$, $\iota(p,t) \in \mathcal{M}_f(\{(a_1, \cdots, a_k) | a_1, \cdots, a_k \in \mathit{Var} \cup \mathit{Val}\})$ with $k = ar(p)$ (idem for arcs $(t,p) \in (T \times P)$), i.e., its inscription is a finite multiset of tuples of variables and values respecting the type of adjacent place p. The meaning of $\iota(p,t) = \emptyset$ is that there is no arc leading from p to t.

- For every transition $t \in T$, $\iota(t)$ is a tuple $(\lambda_t \mid \alpha_t \mid \varrho_t)$, where λ_t is a finite multiset of action terms called its *label*; α_t is a finite multiset of terms called its *annotation* or *value term*; and $\varrho_t \subseteq Ref_T$ is a set of references.

Further, we require that there exists at least one entry and one exit place; that entry places have no incoming arcs and exit places no outgoing arcs; and that all entry and exit places have the type $\{\bullet\}$. ∎ 5.1

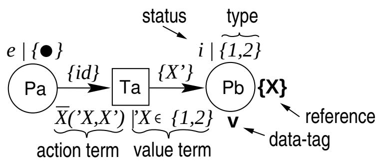


Fig. 5. Simple part of an M-net.

Fig. 5[2] shows part of an M-net with one entry place Pa, one internal place Pb (which holds the value of variable X), one transition Ta and two arcs. The *transition rule* for M-nets is explained informally with the example: If place Pa is marked, transition Ta can occur in two different ways: variable id is bound to $\bullet$ and X' can be bound either to 1 or to 2. Thus, the $\bullet$ is removed from Pa and either 1 or 2 is put on Pb.

5.3 Composition operations

To define the composition operations, we extend the auxiliary net manipulation operators defined in [5]:

- ${}^\bullet(N)$ and $(N)^\bullet$ denote the set of entry and, resp. exit places of N;
- $\otimes\{P_1, \ldots, P_n\}$ multiplies n sets of places. It is essential that the set of references of a created place is the union of the sets of references of the original places.
(e.g., $\otimes\{\{p_1\}, \{p_2, p_3\}\} = \{p_{12}, p_{13}\}$ with $\varrho(p_{12}) = \varrho(p_1) \cup \varrho(p_2)$); and
- $N \oplus \otimes\{P_1, \ldots, P_n\}$ adds $\otimes\{P_1, \ldots, P_n\}$ to N and removes $P_1 \cup \ldots \cup P_n$.

Parallel composition (see left part of Fig. 6) is defined as independent juxtaposition: $N_1 \| N_2 \;=\; 1N_1 \cup 2N_2$.

Sequential composition (see middle part of Fig. 6) merges the exit places of the first net with the entry places of the second. References are handled by the $\otimes$ operator. $N_1; N_2 \;=\; (1N_1 \cup 2N_2) \;\oplus\; \otimes\{(1N_1)^\bullet, {}^\bullet(2N_2)\}$.

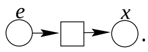
[2]The following figures are simplified: brackets around arc annotations, action terms and references, variables on arcs (like id) which can only be bound to $\bullet$, empty sets, primes around value assertions, and labels of internal places are omitted to improve readability.

Choice (see right part of Fig. 6) merges the entry places of the nets and the exit places of the nets. References are handled by the $\otimes$ operator.

$$N_1 \,\Box\, N_2 = (1N_1 \cup 2N_2) \oplus \otimes\{{}^{\bullet}(1N_1), {}^{\bullet}(2N_2)\} \oplus \otimes\{(1N_1)^{\bullet}, (2N_2)^{\bullet}\}.$$

The iteration construct is $[N_1 * N_2 * N_3]$ (see Fig. 7) which produces the effect of one execution of N_1, followed by zero or more executions of N_2, followed by one execution of N_3. Once more, references are handled by the $\otimes$ operator.

$$\begin{aligned}[N_1 * N_2 * N_3] &= (1N_1 \cup 2N_2 \cup 3N_3 \cup N_{silent}) \\ &\quad \oplus \otimes\{(1N_1)^{\bullet}, {}^{\bullet}(2N_2), N_{silent}{}^{\bullet}, {}^{\bullet}(3N_3)\} \\ &\quad \oplus \otimes\{(2N_2)^{\bullet}, {}^{\bullet}N_{silent}\}\end{aligned}$$

with N_{silent} = .

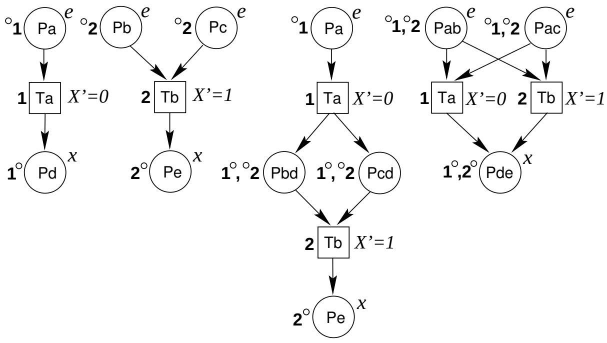


Fig. 6. Example $N_1 \| N_2$, $N_1; N_2$ and $N_1 \,\Box\, N_2$.

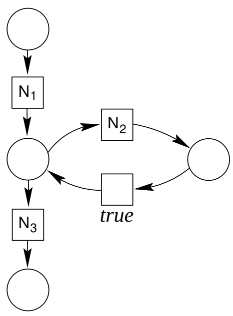


Fig. 7. Iteration schema.

5.4 Synchronisation and restriction

Communication is performed by transition synchronisation. The intuition is that synchronisation of a net w.r.t. an action symbol ($N\mathbf{sy}A$) is performed through a series of basic synchronisations. During a basic synchronisation two corresponding action terms ($A(\cdots)$ and $\overline{A}(\cdots)$) are considered. The communication is performed by a most general unifier which renames the variables in the action terms appropriately. It is important that references and data-tags of places are not effected, whereas the set of references of the resulting transition summarises the sets of references of the involved transitions.

Synchronisation is often followed by restriction. The restriction N **rs** A removes all transitions whose annotations contain action terms $A(\cdots)$ or $\overline{A}(\cdots)$ together with adjacent arcs.

Fig. 8 shows a typical example of synchronisation followed by restriction. This mechanism is used for block structuring. In this example, variable access is depicted. The first subnet shows a place (which may contain a value of a variable X) and a transition for the different access possibilities. The second subnet shows one access to X, decreasing the value by 1.

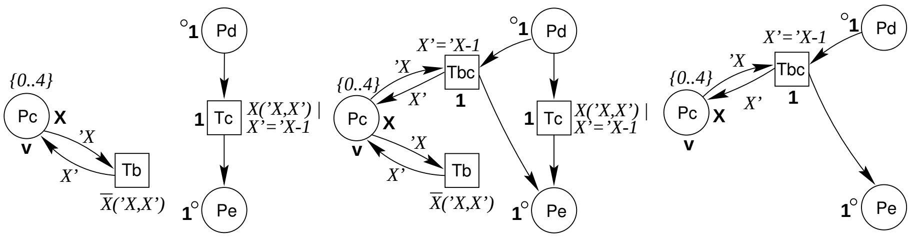


Fig. 8. Example for synchronisation and restriction.

6 Formal semantics

Now, we associate an M-net $\zeta(prog)$ with every program *prog* of the syntax in such a way, that references and data-tags are created automatically as needed. We proceed top-down through the syntax. First, we will consider programs and blocks. The nets for the declarations of variables are then given directly. After presenting the semantics of the different command connectives for parallel, sequence, choice and iteration, we give the semantics of an atomic action. The definition of the semantics is fully compositional.

6.1 Programs and blocks

The **begin-end** program brackets are semantically nearly transparent. The renaming function $\Gamma_{f(\text{block})}$ adds the references $^\circ f(\text{block})$ and $f(\text{block})^\circ$ to the entry and exit places of $\zeta(\text{scope})$, respectively.

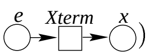
Definition 6.1

$$\zeta(\textbf{begin}\ \text{scope}\ \textbf{end}) = \Gamma_{f(\text{block})}(\zeta(\text{scope})) \qquad \blacksquare\ 6.1$$

A scope may consist of a sequence of variable declarations *decl* followed by a command *com*. The nets for the declarations are juxtaposed with the net for the command (followed by termination actions for variables). The resulting net is first synchronised and then restricted w.r.t. certain action symbols. This ensures that always the correct variable is accessed.

Definition 6.2

$$\zeta(\text{decl};\text{scope}) \quad = \quad (\zeta(\text{decl}) \| \zeta(\text{scope}); \gamma^T(\text{decl}))\,\textbf{sy}\ \delta(\text{decl})\ \textbf{rs}\ \delta(\text{decl})$$

with $\gamma^T(\text{decl}) = ($ 

$)$ and $\delta(\text{decl}) = \{X, Xterm\}$ (for X). $\blacksquare$ 6.2

6.2 Data variables

The semantics of variables is extended by the introduction of the data-tag **v** and a reference for the variable.

Definition 6.3 *Data nets*

$\zeta(\mathbf{var}\ X:\ set) = \mathrm{M}_{data}(X, set)$,
i.e. the parameterised net shown in Fig. 9. ■ 6.3

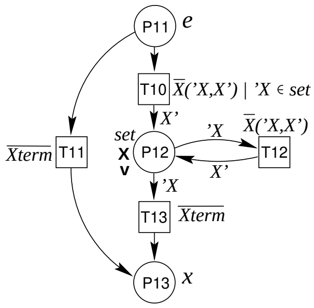


Fig. 9. The data net $\mathrm{M}_{data}(X, set)$.

$T10$ and $T12$ may both synchronise with the variable accesses from within the control flow. Note that $T10$, which provides the initialisation of the variable, is not annotated by a special initialisation action term. This reduces the size of the nets and (what may be more important) the size of the finite prefix of the branching process, because a variable is initialised at the first access and not (perhaps uselessly) at declaration time. $T11$ and $T13$ both synchronise with the corresponding termination transition from within the control flow. We will not consider channels and stacks here.

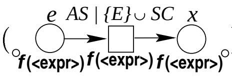
6.3 Atomic actions

The semantics of an atomic action $\langle expr \rangle$ is an M-net $\zeta(\langle expr \rangle)$ with only one transition. The inscription of this transition is constructed recursively from the action terms of the used variables and equations to force the intended equalities at a later synchronisation. In AS the set of action terms is collected; E is used to compose the expression and SC is necessary to allow the usage of unprimed variables assuming that pre- and post-values are equal. The following rules are applied:

$$\begin{array}{rcl}
Ins('x) & = & (\{x('x,x')\}/'x/\phi) \\
Ins(x') & = & (\{x('x,x')\}/x'/\phi) \\
Ins(x) & = & (\{x('x,x')\}/x'/\{('x = x')\}) \\
Ins(const) & = & (\phi/const/\phi) \\
Ins(op\, e_1) & = & (AS_1/op\, E_1/SC_1) \quad \text{with } Ins(e_i) = (AS_i/E_i/SC_i) \\
Ins(e_1\ op\ e_2) & = & (AS_1 \cup AS_2/E_1\ op\ E_2/SC_1 \cup SC_2)
\end{array}$$

E.g. $Ins(x' = y + 1) = (\{x('x,x'), y('y,y')\}/\{x' = y' + 1\}/\{'y = y'\})$.

Definition 6.4 *Atomic action*

$\zeta(\langle expr\rangle) = ($ 

$)$, where $(AS/E/SC) = Ins(expr)$. ■ 6.4

This is the point where (during the compilation) the references for the actions and for the points in the control flow first appear in the semantics.

6.4 Control connectives

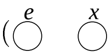
Sequential and parallel composition, iteration and choice are directly translated into the corresponding M-net operations. References are handled correctly due to the correct definition of the $\otimes$ operator.

Definition 6.5 *Parallel and Sequential Composition*

$$\begin{aligned} \zeta(com_1 \| com_2) &= \zeta(com_1) \| \zeta(com_2) \\ \zeta(com_1; com_2) &= \zeta(com_1); \zeta(com_2) \end{aligned}$$ ■ 6.5

Definition 6.6 *Choice and Iteration*

$$\begin{aligned} \zeta(com_1 \,\Box\, com_2) &= \zeta(com_1) \,\Box\, \zeta(com_2) \\ \zeta(\textbf{do } com \textbf{ enter } alt\text{-}set \textbf{ od}) &= [\zeta(com) * R(alt\text{-}set) * E(alt\text{-}set)], \\ \text{and } R(alt\text{-}set_1 \,\Box\, alt\text{-}set_2) &= R(alt\text{-}set_1) \,\Box\, R(alt\text{-}set_2) \\ E(alt\text{-}set_1 \,\Box\, alt\text{-}set_2) &= E(alt\text{-}set_1) \,\Box\, E(alt\text{-}set_2) \\ R(com; \textbf{exit}) &= E(com; \textbf{repeat}) = N_{stop} = (\ \cdots\ ) \\ R(com; \textbf{repeat}) &= E(com; \textbf{exit}) = \zeta(com) \end{aligned}$$



■ 6.6

7 Example

In this section we continue considering the Peterson algorithm already modelled as a PFA. Fig. 10 shows the automatically generated $B(PN)^2$ program. The values of $f(block)$ and $f(\langle expr\rangle)$ are given in brackets.

Fig. 11 was generated with the help of the *Export to PostScript* function of the net editor of the **PEP** tool. It shows the automatically generated M-net semantics with references[3].

Fig. 12 shows the corresponding Petri box. Petri boxes can be considered as a special case of M-nets where, e.g., all places have singleton type $\{\bullet\}$. In comparison with the HL net it is interesting to see the value assertions (like `'i1=0'` for place `P25`) which are easy to generate and very useful for program verification.

[3] Two features of **PEP** are used to enhance readability. Transitions which can never occur and isolated places are removed (automatically), and arcs connecting the variable access transitions within the control flow part to the data nets are hidden. Furthermore, another variable initialisation is chosen and the references (such as $^\circ 1$) which are used internally are made visible.

```
(1) begin
        var i1, i2: {0..1} init 0;
        var t: {1..2} init 1;
(2)     begin                                   begin                          (8)
(3)         do ⟨true⟩ enter                         do ⟨true⟩ enter            (9)
(4)             ⟨i1'=1⟩;                                ⟨i2'=1⟩;               (10)
(5)             ⟨t'=2⟩;                                 ⟨t'=1⟩;                (11)
(6)             ⟨i2=0 or t=1⟩; [CS]   ||              ⟨i1=0 or t=2⟩; [CS]     (12)
(7)             ⟨i1'=0⟩;                                ⟨i2'=0⟩;               (13)
                repeat                                  repeat
            od                                      od
        end                                     end
    end
```

Fig. 10. Peterson algorithm as a $B(PN)^2$ program.

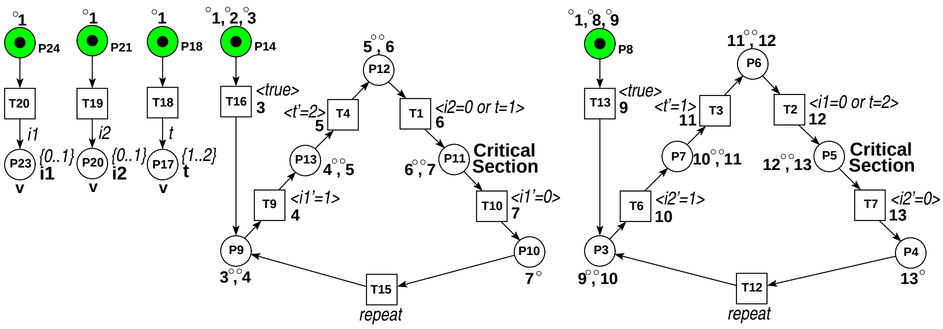


Fig. 11. Peterson algorithm as an HL net.

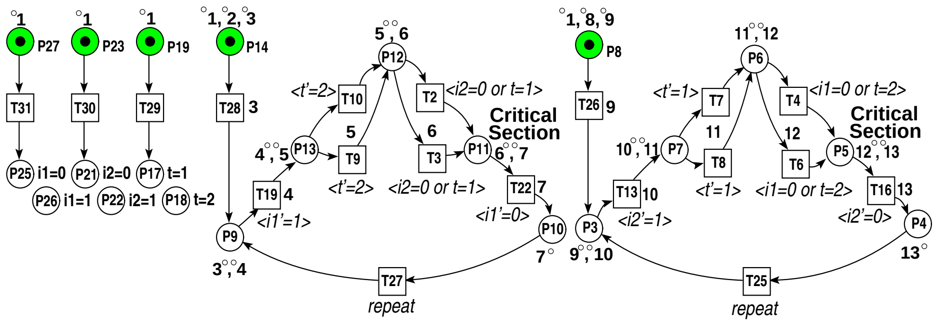


Fig. 12. Peterson algorithm as an LL net.

8 A temporal logic for $B(PN)^2$ programs

The model checker for safe PN integrated in **PEP** has been developed by Esparza [9] and implemented by Graves [13]. This algorithm uses an optimised version of the finite prefix of the branching process of a safe PN [8] and a temporal logic formula as inputs, and then checks whether or not the formula holds for the corresponding net. The original definition of syntax and semantics of these formulae can be extended in a way similar to [16] in order to cover program properties as follows.

Definition 8.1 *Syntax of a branching time logic*

For a safe marked net $N = (P, T, \iota)$ the set of branching time formulae ϕ is defined by the following syntax:

$$\phi \quad ::= \quad \mathbf{true} \mid p \mid c \mid v \mid \neg\phi \mid \phi \wedge \phi \mid \Diamond\phi$$

with ($p \in P, c \subseteq$ *Con-Points*, $v \in$ *Val-Assert*), where $\Diamond\phi$ represents the operator 'there exists a reachable marking such that ϕ'. Other operators such as $\vee$ or $\Box$ can be derived.
E.g., $\phi_1 \vee \phi_2 = \neg(\neg\phi_1 \wedge \neg\phi_2)$ and $\Box\phi = \neg\Diamond\neg\phi$, respectively. ∎ 8.1

The semantics of formulae ϕ is defined in the standard way in terms of (reachable) markings. The only extension is the introduction of a transformation for value assertions and subsets of control points which is essential for case studies like in [12].

Definition 8.2 *Transformations*

A subset c of control points is replaced with the subformula $(p_1 \vee \cdots \vee p_n)$ where $\{p_1, \cdots, p_n\} = \{p_i \in P | c \subseteq \varrho_p(p_i)\}$.

A value assertion v is replaced with the subformula $(p_1 \vee \cdots \vee p_n)$ where $\{p_1, \cdots, p_n\} = \{p_i \in P | v \in \varrho_p(p_i)\}$. ∎ 8.2

Definition 8.3 *Semantics of a branching time logic formula*

A formula $\Diamond\phi$ holds for a marking M, if there is a marking reachable from M for which ϕ holds. A $\Diamond$-free formula can be evaluated directly in a given marking (using the fact that N is safe). A formula ϕ holds for $N = (P, T, \iota)$ if it holds for the initial marking[4] M^0. ∎ 8.3

9 Profiting from references in the **PEP** tool

In this section we exploit how users of the **PEP** tool can profit from references and we explain some parts of the implementation.

The whole **PEP** tool is designed in a very modular way in order to

[4] Within **PEP** the user can choose to evaluate a formula w.r.t. the initial marking (exactly all the entry places are marked) or w.r.t. the current marking (e.g., reached after some simulation steps).

be extendable and regarding the fact that it is developed at universities. Therefore, also the reference component is modularised as far as possible. In the current implementation references are stored in individual files (one for each kind of references, e.g., $\mathrm{B(PN)}^2 \leftrightarrow$ HL net) and most of the functionality is offered by a couple of auxiliary programs which are, for instance, used by the reference component.

It is crucial, that the validity of references is controlled. In the project window of the **PEP** tool the user can see whether or not, e.g., the HL net is connected to the $\mathrm{B(PN)}^2$ program, i.e., whether or not the references between these objects are valid. Editing the program implies that the references become invalid.

9.1 Show references modes

The editors integrated in the **PEP** tool provide different modes to exploit the references.

In the HL net editor, e.g., the user can select a transition (or a place) and depending on the mode (chosen by the user) the corresponding parts of the program (colours are used to distinguish between ${}^\circ\langle \mathit{expr}\rangle$, $\langle \mathit{expr}\rangle$ and $\langle \mathit{expr}\rangle^\circ$) or the transition(s) (or place(s)) of the LL net are highlighted.

The $\mathrm{B(PN)}^2$ editor offers a comfortable possibility to select actions or blocks of a program. After selecting a single atomic action $\langle \mathit{expr}\rangle$ the user can ask for:

1. all transitions whose references contain $f(\langle \mathit{expr}\rangle)$, and
2. all places whose references contain ${}^\circ f(\langle \mathit{expr}\rangle)$ or, resp., $f(\langle \mathit{expr}\rangle)^\circ$.

The same is possible if multiple atomic actions are selected, only that all the corresponding values of $f(\langle \mathit{expr}_k\rangle)$ are considered. In addition, the search can be narrowed to those transitions (or places) whose references match exactly the selected action(s) ($=$ instead of $\subseteq$). The user can choose to which of the other editors the results of the request are forwarded. This feature can, e.g., be used to edit program formulae in the formula editor.

9.2 Simulation

Users who modelled a parallel system by writing a $\mathrm{B(PN)}^2$ program most certainly wants to simulate its behaviour. Perhaps, they do not even want to see the PN. The references constructed by the compiler according to the semantics defined above enable the reference component to offer (among others) this simulation possibility.

The first way is to trigger program simulation by PN simulation. A random or interactive simulation of the HL or LL net can be started and the simulator simply passes the reference (i.e. $f(\langle \mathit{expr}\rangle)$) of each firing transition via the reference component to the program editor/simulator where the corresponding action is highlighted.

Second, an interactive simulation of the program is provided as follows. During each step, the program editor/simulator requests the references of all enabled transitions from the PN simulator. Then it offers an adequate possibility to choose among the activated actions and the possible variable bindings which are then forwarded to the PN simulator in order to fire the corresponding transition.

9.3 Program verification

Formulae can be edited either directly in the formula editor or by use of the program editor and the reference component. In addition, the reference component offers macro expansion features. In the Peterson example, (see Fig. 10 – Fig. 12), e.g., `LIVE(6)` is expanded via $\Box \Diamond$ (`PRESET(T2)` $\vee$ `PRESET(T3)`) to $\Box \Diamond$ ((`P12` $\wedge$ `P21`) $\vee$ (`P12` $\wedge$ `P17`)). Thus, it is possible to verify properties like:

1. 'Does the mutual exclusion property hold?' or 'Is it not possible that both processes are in their critical sections simultaneously?'
   $\neg \Diamond (\{6^\circ, {}^\circ 7\} \wedge \{12^\circ, {}^\circ 13\})$
2. 'Is it always possible that a process enters its critical section?' can be expressed in three different ways:
   (a) $\Box$ (($\Diamond$ $\{6^\circ, {}^\circ 7\}$ $\wedge$ ($\Diamond$ $\{12^\circ, {}^\circ 13\}$))
   (b) $\Box$ (($\Diamond$ ((`'i2=0'` $\vee$ `'t=1'`) $\wedge$ $\{6^\circ, {}^\circ 7\}$)) $\wedge$
   ($\Diamond$ ((`'i1=0'` $\vee$ `'t=2'`) $\wedge$ $\{12^\circ, {}^\circ 13\}$)))
   (c) `LIVE(6)` $\wedge$ `LIVE(12)`

The model checker as well as other analysis algorithms (such as deadlock checkers) returns a sequence of transitions if possible. The tool offers a possibility to visualise this sequence (using an interactive or automatic simulation) in one (or more) of the editors.

The integrated INA [17] tool offers, among others, the possibility to calculate invariants of Petri nets which can then be displayed, for instance, in the corresponding program by use of the reference component.

10 Conclusion

We briefly presented some of the main features of the **PEP** tool. Furthermore, we tried to point out the usefulness of a sophisticated reference component in order to extend PN simulation and PN verification to program simulation and program verification. We believe that a reference component of this style can improve many other tools.

We regret that we could not exploit the other parts of the reference component (like PFA $\leftrightarrow$ B(PN)^2 and HL net $\leftrightarrow$ LL net) in a more detailed way due to restrictions on the length of this paper. For a more detailed overview of the **PEP** system we refer the reader to [6] and the various papers which are available at `http://www.informatik.uni-hildesheim.de/~pep`.

Acknowledgement:
I would like to thank Eike Best, Martin Ackermann, Burkhard Bieber,

Ulf Fildebrandt, Burkhard Graves, Michael Kater, Lutz Pogrell, Robert Riemann, Stefan Römer and Stefan Schwoon for their help and anonymous referees for their comments.

11 References

[1] E. Best. Partial Order Verification with PEP. *Proc. of POMIV'96, Princeton*, 1996.

[2] E. Best, R. Devillers and J. G. Hall. The Box Calculus: a New Causal Algebra with Multi-Label Communication. *Advances in Petri Nets 92*, *LNCS* Vol. 609, 21–69. Springer, 1992.

[3] E. Best and H. Fleischhack, editors. *PEP: Programming Environment Based on Petri Nets.* Hildesheimer Informatik-Berichte 14/95. 1995.

[4] E. Best, H. Fleischhack, W. Frączak, R. P. Hopkins, H. Klaudel, and E. Pelz. An M-Net Semantics of $B(PN)^2$. *Proc. of STRICT*, Workshops in Computing, 85–100. Springer, 1995.

[5] E. Best, H. Fleischhack, W. Frączak, R. P. Hopkins, H. Klaudel, and E. Pelz. A Class of Composable High Level Petri Nets. *Proc. of ATPN'95, Torino*, *LNCS* Vol. 935, 103–118. Springer, 1995.

[6] E. Best and B. Grahlmann. *PEP: Documentation and User Guide.* Universität Hildesheim. ftp.informatik.uni-hildesheim.de /pub/Projekte/PEP/... or http://www.informatik.uni-hildesheim.de/~pep/HomePage.html 1995.

[7] E. Best and R. P. Hopkins. $B(PN)^2$ – a Basic Petri Net Programming Notation. *Proc. of PARLE*, *LNCS* Vol. 694, 379–390. Springer, 1993.

[8] J. Esparza, S. Römer, and W. Vogler. An Improvement of McMillan's Unfolding Algorithm. *Proc. of TACAS'96*, 1996.

[9] J. Esparza. *Model Checking Using Net Unfoldings*, 151–195. Number 23 in Science of Computer Programming. Elsevier, 1994.

[10] H. Fleischhack and B. Grahlmann. *A Petri Net Semantics for B(PN)^2 with Procedures which Allows Verification.* Hildesheimer Informatik-Berichte 21/96. 1996.

[11] B. Grahlmann, M. Moeller, and U. Anhalt. A New Interface for the PEP Tool – Parallel Finite Automata. *Proc. of 2. Workshop Algorithmen und Werkzeuge für Petrinetze*, AIS 22, 21–26. FB Informatik Universität Oldenburg, 1995.

[12] B. Grahlmann. Verifying Telecommunication Protocols with PEP. *Proc. of RELECTRONIC'95, Budapest*, 251–256, 1995.

[13] B. Graves. Erläuterungen zu Esparza's L1-Model-Checker. In [3].

[14] J. E. Hopcraft and J. D. Ullmann. *Introduction to Automata Theory, and Languages, and Computation.* Addison Wesley, 1994.

[15] L. Jenner. A Low-Level Net Semantics for $B(PN)^2$ with Procedures. In [3].

[16] R. Riemann. A Temporal Logic for $B(PN)^2$ Programs. In [3].

[17] P. H. Starke. *INA: Integrated Net Analyzer.* Handbuch, 1992.