Partial Order Verification with PEP*

Eike Best

Institut für Informatik, Universität Hildesheim,
Marienburger Platz 22, D-31141 Hildesheim, Germany
Fax: +49 5121 860475, email: e.best@informatik.uni-hildesheim.de

(August 1996)

Abstract

This paper describes the current status of the verification testbed PEP (Programming Environment based on Petri Nets) from a personal perspective of the author. The paper concentrates on what are perceived as the main highlights and the major shortcomings of PEP.

1 Overview of PEP

PEP [8, 48] is a programming and verification environment which is based on Petri nets, but in which nets play a background role. Primarily, the system accepts two types of input: a program π written in a concurrent programming language and a property ϕ expressed in some temporal logic language. The atoms of ϕ can, for instance, refer to variables and/or to control points of π. Through a sequence of compilation and verification steps, PEP allows ϕ to be checked against π, i.e. to determine whether or not ϕ is true for π (in other words, whether or not π is a model of ϕ). Figure 1 describes the core functional dependencies between PEP's implemented modules.

The user may input a parallel program written in a simple language called $B(PN)^2$ (Basic Petri Net Programming Notation) [9]. A program may be edited and compiled either into a process algebraic expression of the PBC (Petri Box Calculus [3], an extension of CCS [50]) or into a high-level Petri net of the M-net variety [6], and, from either, further into a 1-safe low-level net; both routes yield equivalent low-level nets. In addition, the user may input and edit a temporal logic formula which refers to a program. This formula is compiled into a formula referring to the net associated with the program, if there exists one. It is also possible to edit a net (or a formula referring to it), but then, of

*This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) under grants Be 1267/2-1, Be 1267/2-2, Be 1267/6-1, Fl 207/1-1 and Sta 450/1-1. Cooperation with the Technische Universität München has been supported by project A3 (Spezifikation, Analyse, Modellierung) of the DFG-Sonderforschungsbereich SFB-342 (Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen).

course, the connection with any program (formula) it may have come from is destroyed.

Once the system knows of a 1-safe low-level net (which may have been created either directly or through a program by compilation), the computation of its finite prefix [43] may be initiated. This prefix represents the partial order semantics of the net in concise form. When the finite prefix is constructed, the model-checker is ready to be run. It accepts a (net) formula ϕ and the finite prefix, executes Esparza's model checking algorithm [21] and yields a 'yes' or a 'no', depending on whether or not the formula is true of the net (and hence also whether or not the corresponding program formula – if any – is true of the program – if any).

PEP also has various sideline functionalities in addition to the mainstream functionality just described. For example, $B(PN)^2$ programs can be created automatically by input filters, for instance from PFA (Parallel Finite Automata) [29]. There are output filters as well, for example one for transforming a $B(PN)^2$ program into executable C code [46]. Moreover, PEP includes various algorithms to check specific properties of a net, some of them without needing to compute its prefix. Also, an alternative model checker (which does not need the finite prefix) has been implemented for a special class of nets [4, 65]. These additional functionalities are represented by broken lines in figure 1.

Section 2 describes history and the rationale of PEP, section 3 deals with the programming language and its Petri net semantics, and section 4 describes some of the verification techniques implemented in PEP.

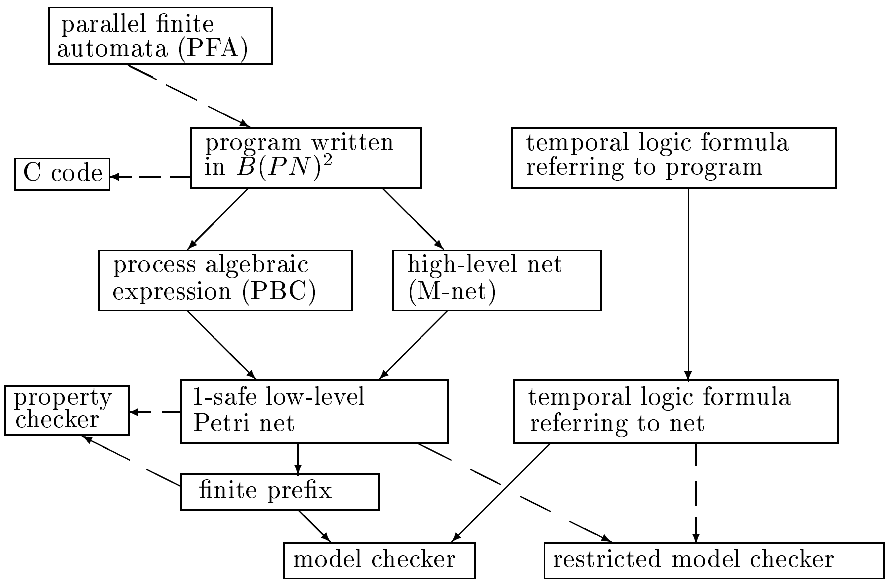

Figure 1: Functionality diagram of the PEP system

2 History and rationale of PEP

The PEP system unites two lines of development: Petri net semantics of concurrent programs and verification algorithms on nets and their partial order semantics.

2.1 Petri net semantics of concurrent programs

For the verification of parallel algorithms expressed in a programming notation, verification techniques such as the Owicki/Gries method [52] are available. For the verification of parallel algorithms expressed by means of Petri nets, other verification techniques such as through S-invariants and traps can be applied [1, 56, 58]. Good programming notations come with an indigenous technique for structuring programs, while Petri nets come with indigenous partial order semantics and analysis methods. Giving a net semantics to a concurrent language may raise the hope that both advantages can be combined, and that verification techniques can be transferred between programming languages and Petri nets.

It may be hoped of such a combination that its compositionality and its usefulness are in proportion. For instance, if a program is made up of variables, sequential compositions and inner blocks, then it is reasonable to expect its associated Petri net to be made up similarly of smaller nets corresponding to the variables and the inner blocks, and combined via a sequential composition defined on nets. This calls for a special kind of algebra on nets, and the box algebra [3] has been developed with that aim in mind.

2.2 Verification algorithms on nets and their partial order semantics

On Petri nets there is a tradition of relating graph-theoretical properties – as well as linear-algebraic properties based on the net's incidence matrix – to behavioural properties. For instance, if a net is covered by an S-invariant [56], then it follows that its state graph (under any initial marking) is finite. Most conditions of this kind are either sufficient or necessary, but not both. It is reasonable to expect that fast graph-theoretical algorithms – or, for that matter, fast linear-algebraic algorithms such as linear programming – can be exploited to check some of these conditions, and then to exclude or to assert certain behavioural properties. Such an approach may be called *static*, because assertions are deduced about the state graph without ever constructing any part of it. Of course, there are limits to this approach, but nevertheless, these limits are far from being fully explored.

More recently, the static approach has been extended to cover not specific properties but a whole class of properties, i.e. a temporal logic. Starting with an observation by Javier Esparza, we showed that a small branching-time temporal logic that can be characterised as 'propositional logic over places, plus the Diamond operator'

can be model-checked by linear algebra – without constructing the state graph – for safe T-systems [4], which are a class of persistent nets, i.e. nets that are essentially without conflicts and choices [39]. It was already clear at the time of writing of [4] it would be difficult to generalise this result to a larger class of nets. Nevertheless, Javier Esparza found a way of model-checking the entire class of safe Petri nets against the same logic which retains a key characteristic property of [4], namely avoiding the construction of the state graph [21]. He showed that instead of constructing the state graph, McMillan's idea [43] of computing a finite prefix of the occurrence net [20, 51] of a net can be exploited.

2.3 Historical remark and acknowledgments

When the PEP project was conceived by Hans Fleischhack and myself in 1993, we were hoping to create not just a testbed for checking the performance of existing Petri-net-based analysis algorithms and for searching for new algorithms, but also a user-friendly environment in which both programs and nets can harmoniously be input, edited, related to each other, simulated, and verified. At that time, all existing Petri net tools were either oriented towards graphical input and had no or very little analysis support, or were oriented exclusively towards analysis without graphical support (the most advanced system of this kind being Peter Starke's INA [62]). None of the systems had the kind of close connection with a concurrent notation that we had envisaged. Thus, we (which initially meant a group consisting of myself and Bernd Grahlmann – who has since then been the chief project researcher and organiser – in Hildesheim and another group led by Hans Fleischhack in Oldenburg) took the risk of starting an implementation effort from scratch, using only know-how from the MOBY project at Oldenburg [23] and input from several students' projects at both sites. The DFG (Deutsche Forschungsgemeinschaft) supports this project with two persons per year over a period of (so far) 1993–1996. In the second stage of the project, 1995–1996, the Humboldt-Universität zu Berlin (by a group led by Peter Starke) has joined the project.

PEP was lucky in getting quite a number of good students interested in the project and contribute to its realisation – names that come to mind are Burkhard Bieber [13], Matthias Damm [16], Burkhard Graves [30], Tobias Himstedt [34], Lars Jenner [37], Michael Kater, Stephan Melzer [46], Stefan Römer [59], Andree Seidel [61] and Thomas Thielke [65], many of whom are still working on and around the project. PEP was also fortunate to have the strong and continued support by Esparza's research group at the Technische Universität München. By these means, as well as by the fortunate circumstance that the EU (European Union), DAAD (Deutscher Akademischer Austauschdienst) and its French counterpart provided funding for related theoretical work (projects DEMON, CALIBAN and POEM), it was possible to develop PEP to the point it has now reached.

It is the work of the persons mentioned in this subsection (and others), more than my own work, that is described in this paper.

3 PEP's inputs and their semantics

PEP primarily accepts two types of input: a program written in the language $B(PN)^2$ [9] and a property referring to a program or to its associated net. $B(PN)^2$ and its Petri net semantics are discussed in sections 3.1 and 3.2. Ways of inputting properties are described in section 3.3.

3.1 PEP's programming language

Ideally, the notation implemented in PEP was meant to serve a similar purpose for parallel programs as Dijkstra's guarded command notation [19] served for sequential nondeterministic programs, namely to represent algorithms in a 'pure' form while having a simple formal semantics. However, at least two additional questions are raised:

- *Should different hardware topologies be supported?* In $B(PN)^2$, the answer is a restricted 'yes', in the sense that both shared memory and message-based topologies are supported. Message buffers may have arbitrary integer size, ranging from 0 for handshake communication to ∞ for unbounded buffers. This may be contrasted with occam [45] which is limited to handshake communication between processes (other communication methods are possible but have to be implemented explicitly).

- *Should special features such as priorities and interrupts be supported?* After convincing ourselves that at least a restricted (i.e. not optimally concurrent) formal semantics of priorities can be given in terms of ordinary Petri nets [10, 32], we have decided, for the time being, not to include priorities in $B(PN)^2$. This may again be contrasted with occam which contains two constructs for expressing priorities between activities.

In addition, it was decided that $B(PN)^2$ should support the following features:

- *Explicit atomic actions.* $B(PN)^2$ allows angular brackets $\langle \ldots \rangle$ to delineate atomic actions. In the translation, every such construct is translated into one, or a set of alternative, single transitions of a Petri net.

- *Pre- and postvalue programming in predicative style.* For instance, an atomic action $\langle x{:=}y \rangle$ would be written as $\langle x'{=}'y \wedge y'{=}'y \rangle$, where $'v$ and v' denote the prevalue and the postvalue, respectively, of v. The idea is that an action touches only such variables mentioned explicitly in it, and any value change making the predicate **true** is acceptable. Note the difference between the above action and $\langle x'{=}'y \rangle$. For the latter, any value change of y would be acceptable in addition to setting the postvalue of x equal to the prevalue of y.

- *Unification of shared memory and channel communication.* To describe channel communication in predicative style, we introduced $c?$ and $c!$ as primitives denoting the value last read on channel c and the value last output to channel c, respectively. They are analogous to the pre- and postvalues of variables.

- *Unification of choices and loops.* $B(PN)^2$ contains a single **do** ... **od** clause both for choices and for loops. The symbol $\Box$ separates alternatives, which can be ended either by the keyword **exit** (indicating exit from the loop) or by the keyword **repeat** (indicating a repetition of the loop).

For instance, figure 2 shows a three-component parallel program which exhibits both shared memory and buffered communication. Note that, due to the channel having capacity 2, both values could be deposited in it without any value being read. If its declaration is changed to **var** c : **chan** 1 **of** $\{3,5\}$, then at most one value could be written before reading, and if it is changed to **var** c : **chan** 0 **of** $\{3,5\}$, then writing and reading are simultaneous. In either case, any of the states $(y,z){=}(3,3)$, $(y,z){=}(3,5)$, $(y,z){=}(5,3)$ or $(y,z){=}(5,5)$ could be a result of the program.

$$
\begin{array}{lll}
\textbf{begin} & \multicolumn{2}{l}{\textbf{var } x : \{3,5\};\ \textbf{var } c : \textbf{chan } 2 \textbf{ of } \{3,5\};} \\
 & & \langle x'{=}3\rangle;\ \langle c!{=}'x \wedge x'{=}'x\rangle \\
 & \| & \langle x'{=}5\rangle;\ \langle c!{=}'x \wedge x'{=}'x\rangle \\
 & \| & \textbf{begin var } y,z : \{3,5\};\ \langle y'{=}c?\rangle;\ \langle z'{=}c?\rangle\ \textbf{end} \\
\textbf{end} & &
\end{array}
$$

Figure 2: A $B(PN)^2$ program with three components and an inner block

$B(PN)^2$ has served a useful purpose of representing algorithms (or nets) linearly. However, it has also turned out to have at least two shortcomings. First, for any large-scale applications, it would be indispensable to include recursion, procedures, and other features. Second, the core language is perhaps slightly too liberal.

As to the first problem, work is in progress to extend $B(PN)^2$ by procedures while still retaining its property of having a compositional net semantics [24, 42]. In a further line of development, object-oriented features are being investigated with respect to their compositional net semantics [41]. These investigations are encouraging, in the sense that all extensions seem to be possible without significant extension of the existing Petri net model on which the semantics of $B(PN)^2$ is based (section 3.2).

Secondly, there seem to be some problems – or at least, debatable issues – with the prevalue/postvalue approach to atomic actions and their Petri net semantics. One of these issues is the (Petri net) semantics of actions such as $a_0 = \langle\textbf{true}\rangle$, $a_1 = \langle x'{=}'x\rangle$ and $a_2 = \langle x'{=}x'\rangle$. At present, a_0 is translated into a single 'silent' transition, a_1 is translated into a choice of transitions which access the variable x but do not change its value, and a_2 is translated into a choice of transitions accessing x and allowing any value change. Operationally, this makes sense: for example, the first action does not interfere with a parallel fourth action accessing x, while the second and third actions do. However, axiomatically, it does not make much sense since the predicate **true** is normally considered equivalent with predicates such as $x'{=}x'$.

Another issue is the syntax of choices and loops. The idea to combine them in a single **do** ... **od** originally arose from translating them into a process algebra which contains only recursion but no iteration. For instance, the program fragment **do** a_1; **exit** $\Box$ a_2; **repeat od** can be translated into the recursive process algebraic expression $X = (a_1 \Box (a_2; X))$ [11]. It turns out that with a well-behaved (in terms of its net semantics) process algebraic iteration construct such as $[E_1 * E_2 * E_3]$, where E_1 is the initialisation, E_2 is the body of the repetition and E_3 is finalisation, the semantics of the general **do** ... **od** is awkward because E_1 must be assumed to be a silent action in general. Other iteration constructs such as $[E_2 * E_3]$ (E_2 being the body and E_3 being finalisation) or more simply $[E_2]$ (denoting E_2^* in terms of regular expressions) are more convenient for giving the semantics of the loop construct, but are less well-behaved in terms of their Petri net semantics.

To avoid these problems, at the present time I favour imposing on $B(PN)^2$ the same restriction that is already built in the guarded command notation and that leads to well-formed [1] nets: that every alternative of a loop construct must begin with a plain action which is not itself another loop. In the present version of $B(PN)^2$, this is implemented by the **enter** clause which separates the initialisation of a **do** ... **od** construct from its body (see figure 3 below for an example).

Another issue is the semantics of multiple communications such as

$$\langle c!{=}5 \wedge d!{=}3 \rangle \| \langle c?{=}x' \wedge c?{=}y' \rangle,$$

where c and d are channels of capacity 0 and x and y are variables. In the present implementation, this parallel command leads to a deadlock, which is due to the underlying semantic ideas stipulating that a single channel should be sufficient (and necessary) for creating a handshake synchronisation out of two separate atomic actions. This approach has been found too restrictive in some contexts and generalisations have been proposed [18, 25].

3.2 The compositional Petri net model underlying $B(PN)^2$

In our approach, two important ideas in giving Petri net semantics to a language such as $B(PN)^2$ are, firstly, that it should be *compositional* and, secondly, that it should be *transparent*. Transparency means that the translation should introduce neither too many auxiliary places and/or transitions nor additional behaviour. Compositionality means that every program object – a variable declaration, an atomic action or a block, ... – should be describable by a stand-alone Petri net, and that the set of all these Petri net ingredients can be combined at the Petri net level by operations which match the syntactic operators used in the program to combine its ingredients (variable declarations, atomic actions, inner blocks, ...).

Robin Milner has already shown in both his books [49, 50] how such a translation can be achieved compositionally at the process algebra level. His approach is, however, lacking in transparency (in the above sense) because of the way of CCS is constructed. For example, sequential composition has to be implemented in a roundabout way, which is not too complicated conceptually, but adds complexity to the resulting expression. As another example, atomic actions such as $\langle x{:=}y \rangle$, where x and y are declared in different blocks, lead to overhead in the translation and hence also in CCS-based analysis of the properties of programs containing such actions.

The box algebra [3] has been devised as a modification and (partial) extension of CCS in order to avoid such overheads. This algebra has been defined together with a direct translation into a class of labelled 1-safe elementary Petri nets called boxes. PEP originally used this translation in order to create a net from a program: first an expression of the algebra is created from the program, and then a box is created from the expression. In practice, however, this approach is of limited usefulness because the resulting nets are usually very large; they are necessarily so large, of course, because all information contained in the program (in particular, variable types) needs to be stored in Petri net form. Already the expressions, which are used as intermediate translation results between programs and nets, tend to become very large in general. (However, they provide a possible interface to tools such as the Edinburgh Concurrency Work Bench [64].) The advantage of this approach is that the full set of Petri net analysis methods – described below in section 4 – is applicable to the result of the translation.

In practice, it turns out that one would wish to translate a program only partially into a net, or into an abbreviated net from which the full net can be derived in a further step if desired. Net theory provides a class of nets for just such a purpose: so-called *high-level nets* ([26, 38] and others). However, in the PEP project high-level nets could not be used directly, because we required all translations to be compositional. Hence prior to using high-level nets, we needed to impose an algebra to make the box algebra operations available for them.

This line of thought gave rise to the model which is now used in PEP: the M-net (modular net) model [6, 7]. M-nets are high-level nets with an additional algebra containing box algebra operations such as choice composition, parallel composition and synchronisation. It is then possible to create an M-net associated with a $B(PN)^2$ program by first constructing little M-nets corresponding to the ingredients (declarations, atomic actions etc.) of the program and then composing these M-nets in the same way as the ingredients of the program are combined. As a rule, the M-net of a program is not significantly larger than the program itself – but, of course, it has a set of inscriptions so as not to lose information. The disadvantage of this approach is that, even though finding structural analysis methods for high-level nets is presently a vigorous area of research (I mention e.g. [60]), there exist very few general methods for analysing a high-level net short of unfolding it, i.e. creating its associated elementary net (which, of course, beats the idea of saving space).

The existing version of PEP does not exploit the compositionality which is built in the semantics. More pessimistically, while it is clear that compositionality is vital for semantics such as Hoare-style axiomatic semantics [35] or weakest precondition semantics [19], it is not yet clear whether compositionality of Petri net semantics can be exploited in any significant way in proofs of programs. The current version of PEP concentrates much more on what I have called transparency, i.e. on minimising the nets that are created, and on applying analysis algorithms to these objects.

3.3 PEP's (current) ways of specifying properties

PEP supports various ways of specifying properties: directly (see section 4.1); by a simple branching-time logic (on which the analysis algorithms described in sections 4.2 and 4.3 are based); and in a linear-time notation (for the semidecision analysis described in section 4.5). The reader will notice that PEP does not (yet) support a truly strong logic, i.e. that some desirable properties may not be expressible, and hence not checkable (in the present version). This is due to a conscious effort of getting as static (and hence, hopefully, as efficient) as possible algorithms for a small (yet not uninteresting) logic first, before extending them at a later stage. It is understood that in a further development of the system, if PEP's indigenous algorithms turn out to be non-generalisable or not easily generalisable, it will be attempted to complement the existing techniques by more traditional state-graph-based algorithms.

The language of the branching-time logic, call it **BL**, refers to a given 1-safe net N with place set $S=\{s_1,\ldots,s_n\}$. An atomic formula is either the constant **true** or a place name s_i. If ϕ, ϕ_1 and ϕ_2 are formulae, then so are $\neg\phi$, $\phi_1\vee\phi_2$ and $\Diamond\phi$. The semantics of this logic refers to pairs (N,M) where N is a net as above and M is a marking of N. By definition, (N,M) always satisfies **true**; (N,M) satisfies s_i iff $M(s_i)>0$; (N,M) satisfies $\neg\phi$ iff it does not satisfy ϕ; (N,M) satisfies $\phi_1\vee\phi_2$ iff it satisfies ϕ_1 or ϕ_2; and (N,M) satisfies $\Diamond\phi$ iff there is a successor marking M' of M such that (N,M') satisfies ϕ. There are derived operators, such as $\wedge=\neg\vee\neg$, $\Box=\neg\Diamond\neg$, etc.

As usual, this simple definition is computationally uninteresting, because using it, the evaluation of a formula ϕ for any given (N,M) involves the (computation and the) traversal of the state graph, possibly many times, depending on the depth of nesting of the diamond operators $\Diamond$. In section 4, more efficient algorithms are described. Examples of properties that can be expressed are:

$\Box\Diamond(\forall s{\in}{}^\bullet t\colon s)$	(liveness of transition t)
$\Diamond\Box s_i$	(token trappable on s_i)
$\Diamond(s_1\wedge\neg s_2\wedge\ldots\wedge\neg s_n)$	(reachability of a marking, in this case $(1,0,\ldots,0)\in\mathbf{N}^{\mid S\mid}$).

Eventuality properties cannot, as a rule, be expressed in **BL**.

A slight change in the syntax of the logic makes formulae refer to programs rather than to nets: given a program, we may allow atomic formulae of the form **true** or $x{=}v$ (where x is a variable and v is a value) or **at** p (where p is a control point). When the program is translated into a corresponding 1-safe net, a program formula may automatically be translated into a corresponding formula referring to that net, because every term of the form $x{=}v$ or **at** p refers to a place of the net. Moreover, the formula is true of the program in its initial state if and only if the corresponding formula is true of the net and its corresponding initial marking. For example, in figure 2, the formula $\Diamond\,(y{=}3 \wedge z{=}3)$ is true, because there exists an execution in which both y and z are set to the value 3.

4 PEP's verification components

PEP attempts to do its verification business as statically as possible, e.g. by running algorithms on the structure of a net (or a program) to deduce properties of the net's (or the program's) behaviour. Five classes of verification techniques can be distinguished in PEP: dedicated analysis, restricted static model-checking, model-checking based on occurrence nets, interfacing to other systems such as INA, and analysis based on linear algebra. Sections 4.1 to 4.5 describe these techniques in turn.

4.1 Dedicated analysis algorithms

In its initial phase, PEP was used as a testbed for students to implement static analysis algorithms. For instance, [17] describes a wealth of theorems giving (often exact) structural conditions for a variety of behavioural properties of certain subclasses of Petri nets. In PEP, nets may first be checked as to whether or not they belong to such a subclass, and if so, one of the structural algorithms can be invoked to test a corresponding property. The test of belonging to a subclass is split into several subtests: 'is the net free-choice?', 'is it a T-system?', 'is it n-bounded' (this test can be neglected if the net comes from a program, since it is then 1-bounded by construction), 'is it live if bounded?' and 'is it deadlock-free?' (using McMillan's algorithm).

The boundedness test involves constructing the state graph, and the corresponding algorithm of PEP is therefore (and because it has not been optimised) rather slow. Nets coming from programs are nearly always non-free-choice, and hence the implemented algorithms for free-choice nets are not useful for such nets. In fact, for these reasons, this line of development of PEP has been all but discontinued, but nevertheless, it may still serve a useful purpose as a testbed for new algorithms. The test for boundedness is faster using the optimised algorithms of INA described below in section 4.4.

4.2 Static model-checking for persistent nets

The essential idea of this model-checker [4] can be described as follows. Let a net N with an initial marking M^0 and a formula ϕ of the temporal logic **BL** be given, such that (N, M^0) is a safe T-system; the problem is to decide whether or not (N, M^0) satisfies ϕ. To check this, consider an innermost subformula of the form $\Diamond(l_1 \wedge \ldots \wedge l_m)$ of ϕ, where each l_i is a literal, i.e. either s_i or $\neg s_i$ for some place s_i. Exploiting the persistence of (N, M^0), it can be shown that this subformula can be equivalently replaced by a formula of the form $\forall t_i{\in}T\colon t_i{\leq}k_i$, where T is the set of transitions of N and the k_i arise as solutions of a linear programming problem which encodes the following question:

> 'What is the maximal number of times that t_i can be executed such that the resulting sequence (there is only one up to equivalent permutations by persistence) does not lose the property of being extendable to a sequence leading to a marking such that all of s_1–s_n are marked/unmarked, depending on whether l_i is $s_i/\neg s_i$?'

Replacing $\Diamond(l_1\wedge\ldots\wedge l_m)$ by $\forall t_i{\in}T\colon t_i{\leq}k_i$ is satisfiability-invariant, i.e. the resulting formula is valid for (N, M^0) if and only if the original formula was. It is now a routine matter to apply this procedure repeatedly until no temporal operator $\Diamond$ (nor $\Box$) are left in the formula, and temporal-operator-free formulae can be evaluated directly on the initial state without computing the state graph. (In order to apply this procedure, the logic has to be extended – temporarily – by atomic formulae of the form $t{\leq}k$, with k being an element of the set $\{-1\}\cup\mathbf{N}\cup\{+\infty\}$; but this is not a problem [4].)

In PEP, this algorithm has been implemented for safe T-systems [65]. Its performance can be startling for people who are used to check other algorithms on very concurrent systems, such as the CCS expression $a_1\|\ldots\|a_n$ which generates 2^n reachable states (or, to mention a less trivial example, Milner's well-known scheduler [50], which is also a T-system). PEP checks formulae on such systems rather quickly.

The model-checking algorithm described in this section has an interesting characteristic property: it shifts complexity away from one of its input parameters (the model) towards the other input parameter (the formula). Since our temporal logic includes the propositional calculus, any model checker is bound to be exponential in the size of the net (note: this is the net, not its state graph!) or in the length of the formula. Interleaving-based model-checking algorithms are exponential in the size of the net (because they generate the state graph) and linear in the length of the formula. The algorithm described in this section is exponential in the length of the formula (because it has to compute disjunctive normal forms repeatedly in order to obtain subformulae of the form $\Diamond(l_1\wedge\ldots\wedge l_m)$), but is provably polynomial in the size of the net. We consider this a desirable property of a model-checking algorithm: the net (and *a fortiori* its state graph) will be very large, in most cases, while the interesting formulae will – in most cases – be of limited size[1] and, in particular, of limited nesting depth.

4.3 Model-checking on finite prefices of occurrence nets

Javier Esparza's model-checking algorithm [21] can be viewed as a generalisation of the algorithm described in section 4.2. The generalisation consists in allowing

[1] Although we will consider an exception in section 5.

any safe Petri net, rather than just persistent ones, as input while retaining essentially the same logic, **BL**. The algorithm itself had to be extended and modified considerably, but Esparza did this in such a way that one of its main properties – viz., shifting algorithmic complexity from the size of the net to the size of the formula – remains as much as possible intact.

Some form of representing behaviour turns out to be necessary, and Esparza has shown that it is in essence sufficient to keep knowledge about the maximal nonsequential processes (i.e. the maximal partial order behaviours [5, 28]) of the input net. (If the net is persistent, then there is only one such process.) A succinct way of representing all processes of the net is by its occurrence net [20, 51], which can loosely be described as a branching structure with processes as 'paths'; the occurrence net of a net is to the set of its processes what the execution tree of a net is to the set of its interleavings.

Unfortunately, the occurrence net of a net is, in general, infinite. Fortunately, there is a way (detected by McMillan [43]) of defining a 'sufficiently large' prefix of the occurrence net – where 'sufficiently large' means that it contains implicitly every reachable marking. That prefix is always finite. Esparza has shown that not only all reachable markings, but also the maximal processes, are recoverable from that finite prefix. Using this finite prefix, model-checking can be done in a similar way as described above, i.e. by replacing innermost subformulae of the form $\Diamond(l_1 \wedge \ldots \wedge l_m)$ by suitable conjunctions not involving $\Diamond$. However, the actual algorithm is significantly more complicated, and it involves the reconstruction of the relevant (maximal) processes from the finite prefix using a continuous 'shift' operator.

Theoretical results about this model-checking algorithm are, (1): that in the special case of safe persistent systems it has polynomial complexity in the size of the prefix, and (2): that for a certain class of safe persistent systems, called safe conflict-free systems, it has polynomial complexity in the size of the net. A corollary of (1) and (2) is that in the special case of safe T-systems, the runtime of this algorithm is provably of the same complexity as that of the previously described algorithm (by orders of magnitude). Moreover, given that there are examples where the finite prefix is an order of magnitude smaller than the state graph, this algorithm performs better than ones based on the latter.

As before, the algorithm performs particularly well for systems with lots of concurrency and little choice, such as $a_1\|\ldots\|a_n$. By contrast, in a typical system without concurrency and with lots of choice, the finite prefix may even be exponential whereas the state graph is only polynomial in size. Consider, for example, the process algebraic term $(a_1{+}b_1);(a_2{+}b_2);\ldots;(a_n{+}b_n)$. The Petri net of this term generates a state graph of size $O(n)$ and a finite prefix of size $O(2^n)$, because after each i'th choice, the rest of the occurrence net gets duplicated. The paper [22], however, describes an improvement of McMillan's unfolding algorithm which allows the calculation of optimised finite prefixes, such that a further result holds, (3): the optimised finite prefix is always of size less or equal to the state graph (in terms of orders of magnitude), and, moreover, the previous results (1) and (2) still hold for the optimised prefix. This optimisation is implemented in the current version of PEP.

```
begin  var h : {1,2} init 1;  var in1, in2 : {false, true} init false;
do    ⟨true⟩   enter   ⟨in1'=true⟩;
                        do ⟨true⟩   enter   ⟨in2=false⟩; exit
                                        □   ⟨in2=true⟩;
                                            do ⟨true⟩   enter   ⟨h=2⟩; ⟨in1'=false⟩;
                                                                ⟨h=1⟩; ⟨in1'=true⟩;
                                                                exit
                                                            □   ⟨h=1⟩; exit
                                            od; repeat
                        od;
                        % cs1 : Critical Section 1
                        ⟨h'=2⟩; ⟨in1'=false⟩; repeat
od
||      do   ... analogous (exchanging 1 and 2) ...            od
end
```

Figure 3: Dekker's algorithm in $B(PN)^2$ notation

After implementing the algorithm in PEP, it was tested on various examples. For instance, at one point of the development, we tested PEP's model-checking algorithm on Dekker's protocol for mutual exclusion (see e.g. [1]). This protocol is reproduced in figure 3 in $B(PN)^2$ notation.[2] At that point in time, we checked the following formulae:

$$\left.\begin{array}{l}
\neg\Diamond(\mathbf{at}\ cs_1 \wedge \mathbf{at}\ cs_2)\\
\Box\,(\mathbf{at}\ cs_1 \;\Rightarrow\; (\Diamond\ \mathbf{at}\ cs_2))\\
\Box\,(\mathbf{at}\ cs_2 \;\Rightarrow\; (\Diamond\ \mathbf{at}\ cs_1))\\
\Diamond\ \mathbf{at}\ cs_1\\
\Diamond\ \mathbf{at}\ cs_2\\
\Box\,\Diamond\,(\mathbf{at}\ cs_1)\\
\Box\,\Diamond\,(\mathbf{at}\ cs_2)\\
\Box\,\Diamond\,(\mathbf{at}\ cs_1 \vee \mathbf{at}\ cs_2),
\end{array}\right\}\quad\text{obtaining, respectively,}\quad\left\{\begin{array}{l}
\mathbf{true}\\ \mathbf{true}\\ \mathbf{true}\\ \mathbf{true}\\ \mathbf{true}\\ \mathbf{false}\\ \mathbf{false}\\ \mathbf{false}.
\end{array}\right.$$

This result is fine for the first five formulae but not for the last three. Burkhard Graves analysed the problem and traced it back to an error in Javier Esparza's paper. It is not possible to describe the full details in this paper, but the essential point is that the finite prefix as defined by McMillan is 'too small' for the 'shift' operator to function in the way it is supposed to function. It is possible to fix this problem by creating a finite prefix which is sufficiently large. This solution is easy to describe and recovers the theoretical results (1) and (2) of Esparza's paper, but it slows down the entire model-checker very significantly. It is only now (July 1996) that Burkhard Graves hopes to have found a way of enlarging the finite prefix in a minimal way while ensuring that the 'shift' operator works as it should and, at the same time, retaining the efficiency of the algorithm. This work will be reported in [31].

[2] We use $\langle h{=}2\rangle$ as an abbreviation of $\langle {}'h{=}2{=}h'\rangle$, and we show this algorithm here explicitly just to give an example for the notation.

4.4 INA interface

There have been recent efforts to combine PEP with Peter Starke's analysis tool INA (Integrated Net Analyser) [62]. Thanks mainly to work by Lutz Pogrell [55], the present version of PEP contains a user-transparent interface between the two tools that were originally developed independently of each other. INA can now be called from the same graphical interface, and nets that are input by PEP can be analysed by INA. In this paper, I refrain from describing the interface, but I mention the analysis capabilities of INA just in order to indicate the added capabilities of the combined tool, PEP/INA. The following is a sample, insignificantly shortened, output of INA, referring to a random net (figure 22 of [2]), reproduced here by courtesy of Peter Starke.

> Start of INA output:
>
> *Current net options are: token type – black (for place/transition nets); time option – intervals; elements – transitions; firing rule – safe; priorities – not to be used; strategy – single transitions.*
>
> *Information on elementary structural properties: the net has no bad reachable states; the net is not statically conflict-free; the net is pure; the net has transitions without pre-places; the net is not coverable by state-machines; the net is not strongly connected; the net is not covered by semipositive P-invariants; the net has transitions without post-place; the net is ordinary, homogeneous, not conservative, not subconservative, not a state machine, not free choice, not extended free choice, not extended simple, not marked, not marked with exactly one token, not a marked graph, connected; the net has a non-blocking multiplicity, no nonempty clean trap, no places without pre-transition, no places without post-transition; the maximal in/out-degree is 4.*
>
> *Computation of the reachability graph. Current analysis options are: no depth restriction; do not print all states; print the dead states; do not print the bad states; no reachability / coverability test. Number of states generated: 642.*
>
> *The net has no dead transitions at the initial marking; the net is bounded; the net is safe; the net has no dead reachable states.*
>
> *Current graph analysis options are: no computation of dynamic conflicts; no computation of distances; no computation of circuits; computation of terminal SC-components; resetability; liveness test.*
>
> *Graph analysis: The initial state is reproducible.*
>
> *Computation of the terminal SC-components. The net is reversible (resetable), covered by semipositive T-invariants, live, live if dead transitions are ignored, live and safe, has no time deadlocks.*
>
> End of INA output.

INA's computation of the reachability graph (state graph) is very fast. Moreover, INA has capabilities for exploiting the stubborn-set method by Valmari [67] and for detecting (and exploiting) state graph symmetries [63]. INA has a small in-built expert system which allows the conclusions of some known theorems to be added to the set of analysis results, provided the premises leading to those conclusions have already been verified for the particular net under consideration.

4.5 Linear-algebraic semidecision analysis

PEP offers a semidecision verification method which is based on a linear upper approximation of the state space. The theory of this method is described in [47] and briefly in [48]. The method extracts from the description of the net and its initial marking, in linear time, a set of linear constraints L that every reachable marking must satisfy. Thus, the solutions of L are a superset of the reachable markings. In order to make use of L for verification, a new set L_P of linear constraints is added to it which specify the markings that do not satisfy a desirable property P. Then, linear programming is used to solve the system $L \cup L_P$; if the system has no solution, every reachable marking satisfies P.

The set of constraints L is actually the union of two subsets L_1 and L_2. L_1 comes from the state equation and has been known for many years. The upper approximation of the state space that can be derived from L_1 is often rough and insufficient to prove many properties. The main contribution of [47] is the definition of L_2, a new set of constraints derived from the traps of the net.

Presently there exist semidecision algorithms for deadlock-freenes (yielding either 'deadlock-free' or 'possibly not deadlock-free, with marking ... being potential deadlock marking' as results) and for the reachability of a marking or a partially specified marking.

Semidecision algorithms, in my opinion, provide good compromises between the inherent algorithmic complexity of full automated verification and the desire to have computer-assistance during validation. Even if such an algorithm yields an indecisive answer, this may still help the user. Another role of automatic verification is in prototyping: typically, a program is verified on a small data domain to gain confidence (or not) for the case of arbitrarily large domains, when automatic verification fails and, if any, manual verification prevails.

5 Performance results

PEP is both a general model-checker and a specific Petri net tool. Hence, its performance can be compared with other model-checkers and with other specific net tools. I report on two such comparisons: one done by Stefan Römer using an article by James C. Corbett [14] and one carried out by Monika Heiner and Peter Deussen as described in [33].

Corbett compares existing systems (SPIN [36], SPIN plus Partial Orders [53], SMV [44] and INCA [15]) on a series of examples, using deadlock detection as a common property to be checked on all examples and all systems. Stefan Römer of the Technische Universität München translated the examples into

PEP input and measured the times for checking the same property (deadlock detection) using PEP's algorithm. It so happens that deadlock-freeness is one of the properties which can be expressed in **BL**, but lead to very large formulae.[3] Because of the importance of deadlock-freeness, therefore, PEP implements a dedicated algorithm (namely, McMillan's) to check this, which uses and exploits the finite (optimised) prefix. Hence this comparison is not really about the general model checking algorithm of PEP but about the dedicated deadlock-detection algorithm.

Table 1 – reproduced here by courtesy of Stefan Römer – gives the preliminary results of the experiment. The 'P(size)' (Problem) column refers to the set of examples given in Corbett's paper. The sets S, T and B, E refer to the places and transitions of the original net and of the finite prefix, respectively.[4] The 'Cuts' column refers to the set of cutoff events (used by the algorithm calculating the finite prefix). The 'F-prefix' and 'Check' columns give the times (in seconds) measured for calculating the finite prefix and for checking the deadlock-freeness property, respectively. The 'C' (Compare) column gives a very crude indication of how PEP's performance relates to the performance of the other systems described in Corbett's paper; $\uparrow$ stands for 'better', $\downarrow$ stands for 'worse' and – stands for 'not applicable' (mainly because the other systems did not give results). The results contained in table 1 have to be read with a pinch of salt, because it was not possible to reproduce exactly the same hardware environment as used by Corbett for his comparison. To compare memory usage, it is necessary to look at the columns 'States' and '$|E|$'. It must be mentioned also that we did not check the examples themselves; Stefan Römer just received files from Corbett which he used as input for PEP.[5] We are presently in the process of repeating the comparisons on a more uniform hardware platform.

The experiment gives a mixed result for PEP: for some examples it performs better than the other model-checkers, for other examples it performs worse. In the light of the theory explained above, PEP is at its best when there is a lot of concurrency but very little choice, and performs comparatively badly in the other extreme, when there is a lot of choice and little concurrency. (In the majority of 'real' cases, there would be a good mix of both concurrency and choice, which PEP, as well as any other automatic model-checker, will have difficulty in coming to grips with.)

The authors of [33] have tested three specifically Petri-net-oriented analysis systems, INA [62], PROD [66] and PEP, on a single common example and a series of properties. The example concerns an industrial production cell with six components: two conveyor belts, a rotatable robot equipped with two extendable arms, an elevating rotatable table, a press and a travelling crane. This case

[3] For a net with about 50 transitions, Bernd Grahlmann has estimated that deadlock-freeness would lead to disjunctive normal forms – which arise necessarily as intermediate stages of the verification – that are about 4 GB long.

[4] Note that the 'transitions' (i.e. $|T|$) column does *not* refer to the transitions of the state graph. Indeed, the number of these transitions has not been counted as they are irrelevant for the algorithms.

[5] This has led to strange effects such as a net with 1047 places and 5633 transitions but only 125 reachable markings.

P(size)	States	$\|S\|$	$\|T\|$	$\|B\|$	$\|E\|$	Cuts	F-prefix	Check	C
CYCL(9)	7423	71	53	172	77	10	0.05	2.18	–
CYCL(12)	74264	95	71	232	104	13	0.13	31.18	–
DAC(12)	14334	84	70	260	146	0	0.12	0.0	↑
DAC(15)	114686	105	88	371	206	0	0.23	0.0	↑
DP(10)	48897	60	40	580	280	90	0.30	0.92	↑
DP(12)	–	72	48	840	408	132	0.62	2.97	↑
DPD(6)	19861	54	54	3786	1892	499	8.92	103.56	↓
DPD(7)	109965	63	63	8630	4314	1129	43.13	1266.08	↓
DPFM(8)	49	87	321	426	209	162	0.08	0.68	–
DPFM(11)	125	1047	5633	2433	1211	1012	1.27	98.30	–
DPH(6)	16897	57	97	14474	7231	3377	85.78	10344.9	–
DPH(7)	79927	66	121	–	–	–	–	–	–
ELEV(3)	7121	327	783	7398	3895	1629	23.75	496.10	↓
ELEV(4)	43440	736	1939	32354	16935	7337	417.32	>13463	↓
FURN(3)	30861	53	99	34505	20770	13837	330.04	>49927	↓
FURN(4)	214757	66	139	–	–	–	–	–	↓
GASN(4)	14847	258	465	15928	7965	2876	115.93	19370.2	↓
GASN(5)	115184	428	841	–	–	–	–	–	↓
GASQ(3)	1705	284	475	2593	1297	490	3.37	102.0	–
GASQ(4)	15431	1428	2705	19864	9933	4060	177.56	35342.2	–
HART(75)	153	377	227	529	302	1	1.13	0.22	↑
HART(100)	203	502	302	704	402	1	2.20	0.32	↑
KEY(4)	44820	164	174	135556	67775	32081	8811.0	–	↓
KEY(5)	–	199	215	–	–	–	–	–	↓
MMGT(3)	7703	122	172	11575	5841	2529	51.56	3166.6	↓
MMGT(4)	66309	158	232	92940	46902	20957	7509.80	–	↓
OVER(4)	4175	71	74	1561	797	240	1.65	7.52	↓
OVER(5)	33460	90	95	7388	3761	1251	30.70	618.39	↓
RING(7)	1700	91	77	813	403	79	0.63	1.20	↑
RING(9)	211528	117	99	1599	795	137	2.20	4.67	↑
RW(9)	523	48	181	9272	4627	4106	5.32	9567.2	↓
RW(12)	4110	63	313	98378	49177	45069	316.84	–	↓
SENT(75)	332	254	105	533	266	40	0.93	1.07	↑
SENT(100)	382	329	130	608	291	40	1.42	1.67	↑
ABP(1)	113	43	95	337	167	56	0.12	0.60	↑
BDS(1)	36097	53	59	12310	6330	3701	44.83	6971.3	↓
DART(1)	–	331	257	–	–	–	–	–	–
FTP(1)	104911	176	529	178077	89042	35247	15645.5	–	↓
FTP(2)	–	229	934	–	–	–	–	–	–
Q(1)	123597	163	194	16090	8402	1173	220.77	1125.12	–
SPD(1)	8690	33	39	5317	3138	1311	15.30	510.7	↓

Table 1: Experimental results by Stefan Römer (cf. [14])

study has recently been used in various (German) projects as a reference example on which various methods, not just Petri nets, can be tested and compared [40].

I will not repeat the experimental results reported in [33], except for mentioning that the speed of checking a property is not unfavourable towards PEP, whenever the property can be expressed in the first place. However, it has to be added that every system compared in [33] has its specific strengths and weaknesses. This is also true for the systems compared in the Corbett experiment described earlier. The authors of [33], therefore, conclude that 'the combination of different tools (even if they provide similar features at first glance) seems to be desirable',[6] a view which I can endorse.

6 Conclusion

PEP is still only a testbed comprising several approaches for the (partial order) verification of Petri nets and concurrent algorithms. In order to create 'a user-friendly environment in which both programs and nets can harmoniously be input, edited, related to each other, simulated, and verified' (quote from section 2), I consider several significant developments to be necessary:

- Adding stronger logics while extending the prefix-based model-checking algorithm conservatively, i.e. preserving its efficiency in the restricted cases.
- Adding other (trace-based or interleaving-based) verification algorithms. Possibly, adding interfaces to SPIN [36] and/or the CWB [64].
- Adding high-level-net analysis methods that do not require unfolding, or that require only partial unfolding into low-level nets.
- Extending the versatility of the input language(s) both by adding procedures to $B(PN)^2$ and by adding new input filters.
- Provision for interactive use and for re-use of intermediate analysis results.
- Better exploitation of compositionality and support for user-defined (as opposed to automatic) verification.
- Modularisation of the system with a view to (partially) providing its source code to the public domain.

I hope that the existing PEP system provides a feasible starting point for such developments.

[6] I changed 'unavoidable' [33] into 'desirable'.

7 Postscript

During POMIV'96 (which was jointly organised by G. Holzmann, D. Peled and V. Pratt), there has been some discussion about 'true' partial order verification.[7] Doron Peled remarked during his talk that what could perhaps be described collectively as 'the set methods' (stubborn sets, sleep sets, persistent sets, ample sets, ... [27, 54, 68]) are essentially based on the exploitation of the (conditional) commutativity of actions, rather than on partial orders, and thus calling them 'partial order methods' may be partially misleading. This point can be seen on the example

$$\langle x{:=}x+1\rangle \parallel \langle x{:=}x+1\rangle,$$

where the two actions commute and are independent in the sense of [27, 54], but do not yield a true partial order when executed, nor are independent in the sense of [12]. Thus, semantic independence – in the sense of giving rise to partial orders – is only one of the circumstances in which the set methods can model-check efficiently, which may partly explain why they would sometimes perform better than PEP on some examples (cf. section 5). By contrast, Esparza's algorithm presented in section 4.3 is based on the representation of behaviour as partial orders and, more particularly, of global states as cuts in such partial orders. The fact that this algorithm performs better than set methods on other examples, is most likely due to the fact that it completely avoids the construction (and hence also the traversal) of the state graph.

On the occasion of comparing the two classes of approaches, another observation could be mentioned. The computation of the finite prefix is often a bottleneck, but it does not have to be recomputed for different properties; once it has been computed for a system, it can be used to check any of the properties expressible in the logic for that system. This is in contrast to most set approaches, where the transition system has to be recomputed each time a different (or sufficiently different) property is to be checked.

Acknowledgments

It is a pleasure to thank Burkhard Bieber, Javier Esparza, Bernd Grahlmann, Hans Fleischhack, Burkhard Graves, Doron Peled and Tom Thielke for comments.

References

[1] E. Best: *Semantics of Sequential and Parallel Programs.* Prentice Hall (1996).

[2] E. Best: A Memory Module Specification using Composable High-level Petri Nets. To appear in *Formal System Specification:* 15 *Contributions for the RPC-Memory Specification.* M. Broy, St. Merz and K. Spies (eds). Springer-Verlag, LNCS (1996).

[7] It is slightly ironic that during the last years, 'true concurrency semantics' has often been used as a synonym of 'partial order semantics'.

[3] E. Best, R. Devillers and J.G. Hall: The Petri Box Calculus: a New Causal Algebra with Multilabel Communication. *Advances in Petri Nets 1992*, G. Rozenberg (ed.). Springer-Verlag, LNCS Vol. 609, 21–69 (1992).

[4] E. Best and J. Esparza: Model Checking of Persistent Petri Nets. Proc. *5th Workshop Computer Science Logic-91.* Springer-Verlag, LNCS Vol. 626, 35-52 (1992).

[5] E. Best and C. Fernández: *Nonsequential Processes: A Petri Net View.* Springer-Verlag, EATCS Monographs on Theoretical Computer Science 13 (1988).

[6] E. Best, H. Fleischhack, W. Frączak, R.P. Hopkins, H. Klaudel and E. Pelz: A Class of Composable High Level Petri Nets. Proc. *Petri Nets'95*, Torino, G. De Michelis, M. Diaz (eds). Springer-Verlag, LNCS Vol. 935, 103–120 (1995).

[7] E. Best, H. Fleischhack, W. Frączak, R.P. Hopkins, H. Klaudel and E. Pelz: An M-net Semantics of $B(PN)^2$. Proc. *STRICT'95*, Berlin, J. Desel (ed.). Springer-Verlag, Workshops in Computing, 85–100 (1995).

[8] E. Best and B. Grahlmann: PEP - more than a Petri Net Tool. Proc. *Tools and Algorithms for the Construction and Analysis of Systems*, 2nd International Workshop, TACAS'96, Passau, March 1996, T. Margaria, B. Steffen (eds). Springer-Verlag, LNCS Vol. 1055, 397–401 (1996).
The system can be downloaded from ftp.informatik.uni-hildesheim.de.

[9] E. Best and R.P. Hopkins: $B(PN)^2$ – a Basic Petri Net Programming Notation. Proc. *PARLE'93*, A. Bode, M. Reeve, G. Wolf (eds). Springer-Verlag, LNCS Vol. 694, 379–390 (1993).

[10] E. Best and M. Koutny: Petri Net Semantics of Priority Systems. *Theoretical Computer Science* 96, 175-215 (1992).

[11] E. Best and M. Koutny: Solving Recursive Net Equations. Proc. *ICALP'95*, Szeged, F. Gecseg, Z. Fülöp (eds). Springer-Verlag, LNCS Vol. 944, 605-623 (1995).

[12] E. Best and Ch. Lengauer: Semantic Independence. *Science of Computer Programming* Vol. 13, 23–50 (1989/90).

[13] B. Bieber: Blockabstraktion in einer Programmiersprache für verteilte Systeme. Diploma Thesis, Universität Hildesheim (1996).

[14] J.C. Corbett: Evaluating Deadlock Detection Methods for Concurrent Software. Technical Report, Dep. of Information and Computer Science, University of Hawaii at Manoa (1995).

[15] J.C. Corbett and G.S. Avrunin: A Practical Method for Bounding the Time between Events in Concurrent Real-time Systems. Proc. *Int. Symp. on Software Testing and Analysis.* New York, 110–116 (1993).

[16] M. Damm: Implementierung der High-Level-Netz-Semantik für $B(PN)^2$. Diploma Thesis, Universität Oldenburg (1995).

[17] J. Desel and J. Esparza: *Free Choice Petri Nets.* Cambridge Tracts in Theoretical Computer Science 40. Cambridge University Press (1995).

[18] R. Devillers: On a more Liberal Synchronisation Operator for the Petri Box Calculus. Technical Report LIT-281, Université Libre de Bruxelles (1993).

[19] E.W. Dijkstra: *A Discipline of Programming.* Prentice Hall (1976).

[20] J. Engelfriet: Branching Processes of Petri Nets. *Acta Informatica* 28, 575–591 (1991).

[21] J. Esparza: Model Checking based on Branching Processes. Habilitation, Hildesheim (1993). Published as: Model Checking Using Net Unfoldings. Proc. *TAPSOFT'93* (1993), M.C. Gaudel, J.P. Jouannaud (eds). Springer-Verlag, LNCS Vol. 668, 613–628 (1993). Full version in *Science of Computer Programming* Vol. 23, 151–195 (1994).

[22] J. Esparza, St. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. Proc. of *Tools and Algorithms for the Construction and Analysis of Systems*, 2nd International Workshop, TACAS'96, Passau, March 1996, T. Margaria, B. Steffen (eds), LNCS Vol. 1055, Springer-Verlag (1996).

[23] H. Fleischhack: Objektnetze in der Büromodellierung – Probleme und Lösungsmöglichkeiten. Internal Report, Universität Oldenburg (1992).

[24] H. Fleischhack and B. Grahlmann: A Petri Net Semantics for $B(PN)^2$ with Procedures which Allows Verification. Hildesheimer Informatikbericht 21/96 (1996).

[25] W. Frączak: *Composition Parallèle et Synchronisation de Systèmes à Événements Discrets.* PhD Thesis, Université Paris-Sud (1996).

[26] H. Genrich: Predicate-transition Nets. In *Petri Nets: Central Models and their Properties*, Advances in Petri Nets 1986, Part I. Springer-Verlag, LNCS Vol.254, 207-247 (1987).

[27] P. Godefroid: On the Costs and Benefits of using Partial-order Methods for the Verification of Concurrent Systems. Proc. *POMIV'96.*

[28] U. Goltz and W. Reisig: The Non-sequential Behaviour of Petri Nets. *Information and Control* Vol. 57/2–3, 125–147 (1983).

[29] B. Grahlmann, M. Moeller and U. Anhalt: A New Interface for the PEP Tool: Parallel Finite Automata. Proc. 2nd Workshop *Algorithmen und Werkzeuge für Petrinetze 1995*, J. Desel, H. Fleischhack, A. Oberweis, and M. Sonnenschein (eds), Report AIS 22, Universität Oldenburg (1995).

[30] B. Graves: Ein Modelchecker für eine Linear-Time-Logik. Diploma Thesis, Universität Hildesheim (1995).

[31] B. Graves: Identification of Specific Processes Contained in McMillan's Finite Unfolding. Work in progress (1996).

[32] M.H.T. Hack: Petri Net Languages. MIT, Laboratory of Computer Science, TR-159 (1976).

[33] M. Heiner and P. Deussen: Petri Net Based Design and Analysis of Reactive Systems. Proc. *WODES'96, Workshop on Discrete Event Systems*, Edinburgh (1996).

[34] T. Himstedt: Entwurf und Implementierung eines High-level-Petrinetzeditors und -simulators. Diploma Thesis, Universität Hildesheim (1995).

[35] C.A.R. (Tony) Hoare: An Axiomatic Basis for Computer Programming. *Communications of the ACM* Vol. 12, 576–580 (1969).

[36] G. Holzmann: *Design and Validation of Computer Protocols*. Prentice Hall (1990).

[37] L. Jenner: Ein Prozedurkonzept für die parallele Hochsprache $B(PN)^2$. Diploma Thesis, Universität Hildesheim (1994).

[38] K. Jensen: *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1.* EATCS Monographs on Theoretical Computer Science (1992).

[39] L.H. Landweber and E.L. Robertson: Properties of Conflict-free and Persistent Petri Nets. *Journal of the ACM* Vol. 25(3), 352–364 (1978).

[40] C. Lewerentz and T. Lindner: Formal Development of Reactive Systems – Case Study: a Production Cell. Springer-Verlag, LNCS Vol. 891 (1995).

[41] J. Lilius: $OOB(PN)^2$: an Object Oriented Petri Net Programming Notation. Proc. *2nd Workshop on Object-Oriented Programming and Models of Concurrency*, Osaka, Japan, 89–103 (1996).

[42] J. Lilius and E. Pelz: An M-net Semantics for $B(PN)^2$ with Procedures. Draft (1995/1996).

[43] K.L. McMillan: Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits. Proc. *4th Workshop on Computer Aided Verification*, 164–174 (1992).

[44] K.L. McMillan: *Symbolic Model Checking*. Kluwer Academic Publishers, Boston (1993).

[45] D. May: Occam. *ACM SIGPLAN Notices* Vol. 18/4, 69–79 (1983).

[46] St. Melzer: Ein C-Codegenerator für $B(PN)^2$. Diploma Thesis, Universität Hildesheim (1995).

[47] St. Melzer and J. Esparza: Checking System Properties via Integer Programming. Proc. *ESOP'96*, Hanne Riis Nielson (ed.). Springer-Verlag, LNCS 1058, 250–264 (1996).

[48] St. Melzer, St. Römer and J. Esparza: Verification using PEP. Proc. *AMAST'96*. Springer-Verlag, LNCS 1101, pp. 591–594. (1996).

[49] A.J.R.G. (Robin) Milner: *A Calculus of Communicating Systems*. Springer-Verlag, LNCS Vol. 92 (1980).

[50] A.J.R.G. (Robin) Milner: *Communication and Concurrency*. Prentice Hall (1989).

[51] M. Nielsen, G.D. Plotkin and G. Winskel: Petri Nets, Event Structures and Domains. *Semantics of Concurrent Computation*, G. Kahn (ed.). Springer-Verlag, LNCS Vol. 70, 266–284 (1979).

[52] S.S. Owicki and D. Gries: An Axiomatic Proof Technique for Parallel Programs. *Acta Informatica* Vol. 6, 319–340 (1976).

[53] Doron Peled: Combining Partial Order Reductions with On-the-fly Model-Checking. *Journal of Formal Methods in Systems Design*, Vol. 8(1), 39–64 (1996).

[54] D. Peled: Partial Order Reduction: Linear and Branching Temporal Logics and Process Algebras. Proc. *POMIV'96*.

[55] L. Pogrell: Inatcl – eine flexible Schnittstelle zu INA. Proc. 2nd Workshop *Algorithmen und Werkzeuge für Petrinetze 1995*, J. Desel, H. Fleischhack, A. Oberweis, and M. Sonnenschein (eds), Report AIS 22, Universität Oldenburg (1995).

[56] W. Reisig: *Petri Nets. An Introduction.* EATCS Monographs on Theoretical Computer Science Vol. 4, Springer-Verlag (1985).

[57] W. Reisig: Petri Nets and Algebraic Specifications. *Theoretical Computer Science* 80, 1-34 (1991).

[58] W. Reisig: Petri Net Models of Distributed Algorithms. Springer-Verlag, LNCS Vol. 1000 (1996).

[59] St. Römer: Implementierung der Branching-Box-Algebra. Diploma Thesis, Universität Hildesheim (1993).

[60] K. Schmidt: *Symbolische Analysealgorithmen für algebraische Petrinetze.* Dissertation (December 1995).

[61] A. Seidel: Entwurf und Implementierung eines Editors für elementare Petriboxen. Diploma Thesis, Universität Oldenburg (1993).

[62] P.H. Starke: *INA – Integrated Net Analyser.* Manual (1992).

[63] P.H. Starke: *Analyse von Petrinetzmodellen.* Verlag B.G. Teubner (1990).

[64] P. Stevens: http://www.dvcs.ed.ac.uk/staff/pxs/index.html

[65] Th. Thielke: Modelchecking als Komponente der Petrinetz-basierten Entwicklungs- und Programmierumgebung PEP. Proc. *Algorithmen und Werkzeuge für Petrinetze*, J. Desel, A. Oberweis, W. Reisig (eds), Berlin (1994).

[66] K. Varpaaniemi, J. Halme, K. Hiekkanen and T. Pyssysalo: PROD reference Manual. Helsinki University of Technology, Digital Systems Laboratory, Series B: Technical Report No.13 (1995).

[67] A. Valmari: Stubborn sets for Reduced State Space Generation. Proc. *10th Int. Conf. on Petri Nets*, Vol. II, Bonn, 1–22 (1989).

[68] A. Valmari: Stubborn Set Methods for Process Algebras. Proc. *POMIV'96.*