Preface

This report contains the contributions to the final workshop of the first period of the project **PEP** (Programming Environment based on Petri nets). **PEP** is a cooperative effort of the Universities of Hildesheim, Oldenburg and Berlin (Humboldt-Universität) supported by the 'Deutsche Forschungsgemeinschaft' in the years 1993 to 1996.

The contributions subsumed in this report describe implementations and generalisations related to the concurrent programming language $B(PN)^2$ and the Petri box calculus.

The first part of the **PEP** workshop took place during the German workshop *Algorithmen und Werkzeuge für Petrinetze* (Humboldt-Universität zu Berlin, October 10-11, 1994). The second was held within the English workshop *Concurrency, Specification and Programming* (Humboldt-Universität zu Berlin, October 12-14, 1994) organised by Jörg Desel and Peter H. Starke respectively. For this reason, the present report contains contributions both in the German and the English language.

The report begins with an introduction to the **PEP** system. The verification component of the **PEP** system is described in sections 2-4. An efficient algorithm for the computation of the *unfolding* of a net is given in section 2. For a restricted class of nets a model checking algorithm is described in section 3, based on linear programming. The model checking algorithm presented in section 4 requires the *unfolding* as an input. Section 5 deals with making explicit the connection between $B(PN)^2$ and its semantics. Section 6 and 7 discuss extensions of the programming language $B(PN)^2$. In section 6 the definition of a temporal logic is proposed, in section 7 the syntax and the semantics of $B(PN)^2$ is extended by a procedure concept. A C-code generator for $B(PN)^2$ is presented in section 8. Finally, section 9 contains a contribution about language preserving reductions.

We thank Robert Riemann (Hildesheim/Paris), Anja Gronewold (Oldenburg/Paris) and Bernd Grahlmann (Hildesheim) for their editorial work in putting together the articles of this report.

E. Best, H. Fleischhack (Hildesheim/Oldenburg, May 1995)

Vorwort

Dieser Bericht enthält die Beiträge zum Abschlußworkshop der ersten Projektphase des Projekts PEP (Programming Environment based on Petri nets). PEP ist ein von der Deutschen Forschungsgemeinschaft von 1993 bis 1996 gefördertes Projekt der Universitäten Hildesheim, Oldenburg und Berlin (Humboldt-Universität).

Die hier aufgenommenen Beiträge dokumentieren die in der ersten Projektphase 1993–1994 vorgenommenen Implementierungen von und Erweiterungen an der Theorie der parallelen Programmiersprache $B(PN)^2$ und dem Petri-Box-Kalkül.

Die erste Hälfte des PEP-Workshops fand im Rahmen des deutschsprachigen Workshops *Algorithmen und Werkzeuge für Petrinetze* (Humboldt-Universität zu Berlin, 10.–11. Oktober 1994) statt, die zweite Hälfte zu Beginn des englischsprachigen Workshops *Concurrency, Specification and Programming* (Humboldt-Universität zu Berlin, 12.–14. Oktober 1994), organisiert von Jörg Desel bzw. Peter H. Starke. Hieraus erklärt sich die Tatsache, daß es in diesem Band sowohl englisch- als auch deutschsprachige Beiträge gibt.

Der Gesamtbericht wird eingeleitet durch eine Einführung in das **PEP**-System. Die Beiträge zwei bis vier beschäftigen sich mit der Verifikationskomponente des **PEP**-Systems. Im zweiten Beitrag wird ein effizienter Algorithmus zur Berechnung der Entfaltung eines Netzes vorgestellt. Der im dritten Abschnitt beschriebene Modelchecking-Algorithmus basiert auf Linearer Programmierung. Der im vierten Beitrag erläuterte Modelchecking-Algorithmus benötigt als Eingabe die Entfaltung aus dem zweiten Beitrag. Der fünfte Beitrag beschreibt eine mögliche Nutzung von Verweisen zwischen den verschiedenen Objekten des **PEP**-Systems. Erweiterungen der verwendeten Programmiersprache werden in den zwei folgenden Abschnitten 6 und 7 beschrieben. Zuerst wird die Definition einer temporalen Logik vorgeschlagen, anschließend wird der Sprachumfang durch ein Prozedurkonzept vervollständigt. Eine praktische Anbindung stellt der in Abschnitt 8 vorgestellte C-Code Generator für $B(PN)^2$ dar. Den Abschluß bildet ein Beitrag über spracherhaltende Reduktionsregeln.

Wir danken Robert Riemann (Hildesheim/Paris), Anja Gronewold (Oldenburg/Paris) und Bernd Grahlmann (Hildesheim) für die Übernahme der Endredaktion bei der Zusammenfassung der Artikel zu diesem Bericht.

E. Best, H. Fleischhack (Hildesheim/Oldenburg, Mai 1995)

Programm des PEP-Workshops

In: Algorithmen und Werkzeuge für Petrinetze, 11.10.94

14.00–14.30	Grahlmann, Bernd (Hildesheim)	**PEP**: Ein Werkzeug zur Modellierung, Simulation, Analyse und Verifikation paralleler Systeme
14.30–15.00	Römer, Stefan (Hildesheim)	Ein effizienter Algorithmus zur Berechnung von Entfaltungen endlicher, 1-sicherer Petrinetze
15.00–15.30		Pause
15.30–16.00	Thielke, Thomas (Hildesheim)	Modelchecking als Komponente der Petrinetz-basierten Entwicklungs- und Programmierumgebung **PEP**
16.00–16.30	Graves, Burkhard (Hildesheim)	Erläuterungen zu Esparza's L1-Model-Checker
16.30–17.00	Damm, Matthias (Oldenburg)	Verweise in **PEP** und mögliche Anwendungen

In: Concurrency, Specification and Programming, 12.10.94

9.00– 9.15	(P. H. Starke)	Opening and Welcome
9.15–12.30	Chairman: E. Best	A Programming Environment based on Petri nets (**PEP**)
9.20–10.00	Grahlmann, Bernd (Hildesheim)	An Introduction to the Principles, the Functionality and the Usage of the **PEP**-System
10.00–10.30	Riemann, Robert (Hildesheim)	A Temporal Logic for $B(PN)^2$ Programs
10.40–11.10		Break
11.10–11.40	Jenner, Lars (Hildesheim)	A Low-Level Net Semantics for $B(PN)^2$ Procedures
11.50–12.20	Melzer, Stephan (Hildesheim)	Conception of a C-Codegenerator for $B(PN)^2$
12.40–14.00		Break
14.00–17.00	Chairman: P. H. Starke	PEP-Session continued
14.00–14.30	Fleischhack, Hans (Oldenburg)	A High-Level Net Semantics for $B(PN)^2$ Procedures
14.40–15.10	Gronewold, Anja (Oldenburg)	Language Preserving Reductions of Safe Petri Nets

Inhalt

1 An Introduction to the Principles, the Functionality and the Use of the PEP System

Bernd Grahlmann
Universität Hildesheim
Institut für Informatik
Marienburger Platz 22
D - 31141 Hildesheim
e-mail: bernd@informatik.uni-hildesheim.de

Abstract

The **P**rogramming **E**nvironment based on **P**etri Nets **PEP** focuses its interest mainly on four kinds of objects well suited to model, to simulate, to analyse and to verify parallel systems:

1. Programs written in a dedicated parallel programming language $B(PN)^2$ (**B**asic **P**etri **N**et **P**rogramming **N**otation)
2. Programs expressed in a process algebra called PBC (**P**etri **B**ox **C**alculus) –
3. Petri boxes, which are a class of 1-safe labelled Petri nets
4. Formulae given in a temporal logic.

The central part of **PEP** is a project management component. Apart from editors for the kinds of objects just described, compilers and analysis algorithms can be invoked.

This talk introduces the design and the functionality of **PEP**. It is augmented by examples of parallel systems being modelled and verified by the tool. Performance results as well as other statistics are presented. Finally an overview concerning future work is given.

1.1 Introduction and Motivation

Nowadays the design of parallel systems is a common task. In offices procedures have to be planned. It might be necessary to insure a certain level of security e. g. some documents should be accesible to only some employees. In factories the usage of machines during the production process has to be optimized – to avoid a deadlock is one of the major side conditions. In a lot of technical appliances different kinds of protocols guarantee correct functionality. In a telephone network for instance protocols are used to establish communication channels. Last but not least there are a lot of people in computer science or even those more silly ones in theoretical computer science who are thinking about new mutual exclusion algorithms, for example. Furthermore they are interested in correctness proofs.

It can therefore be stated that there is a big need for a tool as assistance during the modelling, the simulation, the analysis and the verification of parallel i. e. distributed systems. It can also be ascertained, that there exists a variety of tools supporting the modelling and simulation phase, some of them based on parallel programming languages, some of them on process algebras and others on Petri nets. Each of them has advantages due to these fundamental bases. Furthermore there are some tools offering various analysis algorithms, nearly all of them are lacking a comfortable user interface.

Thus the main goal of the **PEP** project has been to design and implement a tool which subsumes all advantages of the different kinds of existing tools.
First of all the combined application of two of the most accepted theories worldwide – process algebras and Petri nets – on parallel programs allows the choice between three of the best ways to model parallel systems without loosing the possibility to simulate, to analyse and what is most interesting to verify the resulting systems. This means, that the user of the **PEP** Systems has got the power to model parallel systems by writing programs in a high-level parallel programming language called $B(PN)^2$ (**B**asic **P**etri **N**et **P**rogramming **N**otation [9]) and to check almost every property of interest afterwards. Finally it has been straightforward to embed the whole functionality in an easy to use graphical user interface.
After this praising of the **PEP** system a more detailled presentation should justify the laurels given in advance. The brief description of the different kinds of objects handled by the **PEP** prototype (see subsection 1.2) as well as the illustration of the **PEP** system's structure (i. e. the putting together of individual modules) given in subsection 1.3 are the basis for the understanding of the following more specific talks. Subsequently the description of the typical cycle of usage shown in subsection 1.4 and the following illustration by considering a typical example (for a theoretical computer scientist) should mediate an intuition without going into detail. The consideration of a further (somehow more complex) example in subsection 1.6 gives an insight in the efficiency of the verification component. Finally an overview over future work is given.

1.2 Objects used by the PEP system

The users of the **PEP** system can choose between four different kinds of objects when modelling parallel systems:

1. They can write parallel programs in the predicative programming language $B(PN)^2$. These programs are structured in a declaration and an instruction part. Sequential and parallel composition, iteration and non-deterministic choice are possible operations. Different processes can share common memory or use channel communication. The implementation of a procedure concept (see [24]) and abstract data types extends $B(PN)^2$ to a complete programming language where even complex programs can be expressed.
2. Programs in a special process algebra called PBC (**P**etri **B**ox **C**alculus [6]) can either be built automatically or written by hand.
3. The usage of high-level Petri boxes is partely prepared (see [8]) from the theoretical point of view but not fully implemented yet.
4. The net semantics of a $B(PN)^2$ program is defined in terms of Petri boxes [4] which are special, 1-safe, labelled Petri nets.

It depends on the individual user with which kind of object he would like to start the modelling phase. As shown in figure 1.2.1 it is possible to compile a $B(PN)^2$ program in a compositional way into a Petri box. HL-Petri boxes or PBC programs can be intermediate representations.
Furthermore there are two more kinds of object which are important for the **PEP** system.

1. During the verification phase of the usage it is necessary to calculate the finite prefix of a branching process. This "Entfaltung"[1] of a Petri net contains all the necessary information to check (see [17]) various properties of Petri nets.

[1]The german name is used to distinguish it from the unfolding of a high-level net into a low-level net.

2. If the user wants to check whether a modelled system satisfies a special property, he has the possibility to express this property in terms of a temporal logic formula.

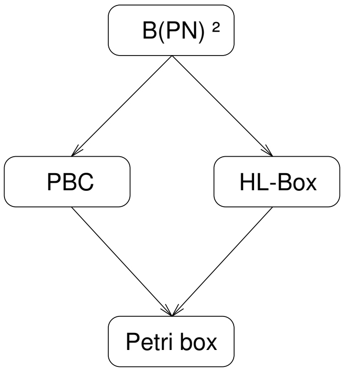


Figure 1.2.1 : *The dependencies between the objects of the* **PEP** *system. The arrows are representing the compiling relation between the different objects.*

During each modelling approach several different objects are created. Normaly a $B(PN)^2$ program is written, the corresponding PBC program as well as the Petri box are compiled automatically, the “Entfaltung“ is calculated and interesting properties are expressed by formulae. It has been straightforward to gather together all these objects (related to an approach) into a project. In addition a project contains the results of the analysis and the verification and some user defined documentation.

1.3 The structure of the PEP prototype

It is now obvious that the **PEP** system has to consist of five different types of components.

1. Editors are needed for $B(PN)^2$ programs, PBC programs, low-level and high-level nets, formulae and project documentation.

2. Among the different compilers there are three

   (a) $B(PN)^2 \Rightarrow$ PBC

   (b) PBC $\Rightarrow$ Low-level Petri box

   (c) Petri box $\Rightarrow$ “Entfaltung“

   that are already integrated in the prototype and two

   (a) $B(PN)^2 \Rightarrow$ High-level Petri box

   (b) High-level Petri box $\Rightarrow$ Low-level Petri box

   that are not fully implemented yet. An additional one which generates C-code taking $B(PN)^2$ programs as input (see [29]) is nearly implemented but not yet integrated.

3. A simulator for low-level Petri boxes enables a first comprehensive intuition of the behaviour of a Petri box. The implementation of the one for high-level Petri boxes is to be finished soon.

4. Some standard algorithms (as free choice, T-system, liveness, reachability and reversibility) are part of the **PEP** prototype. Further ones will follow.

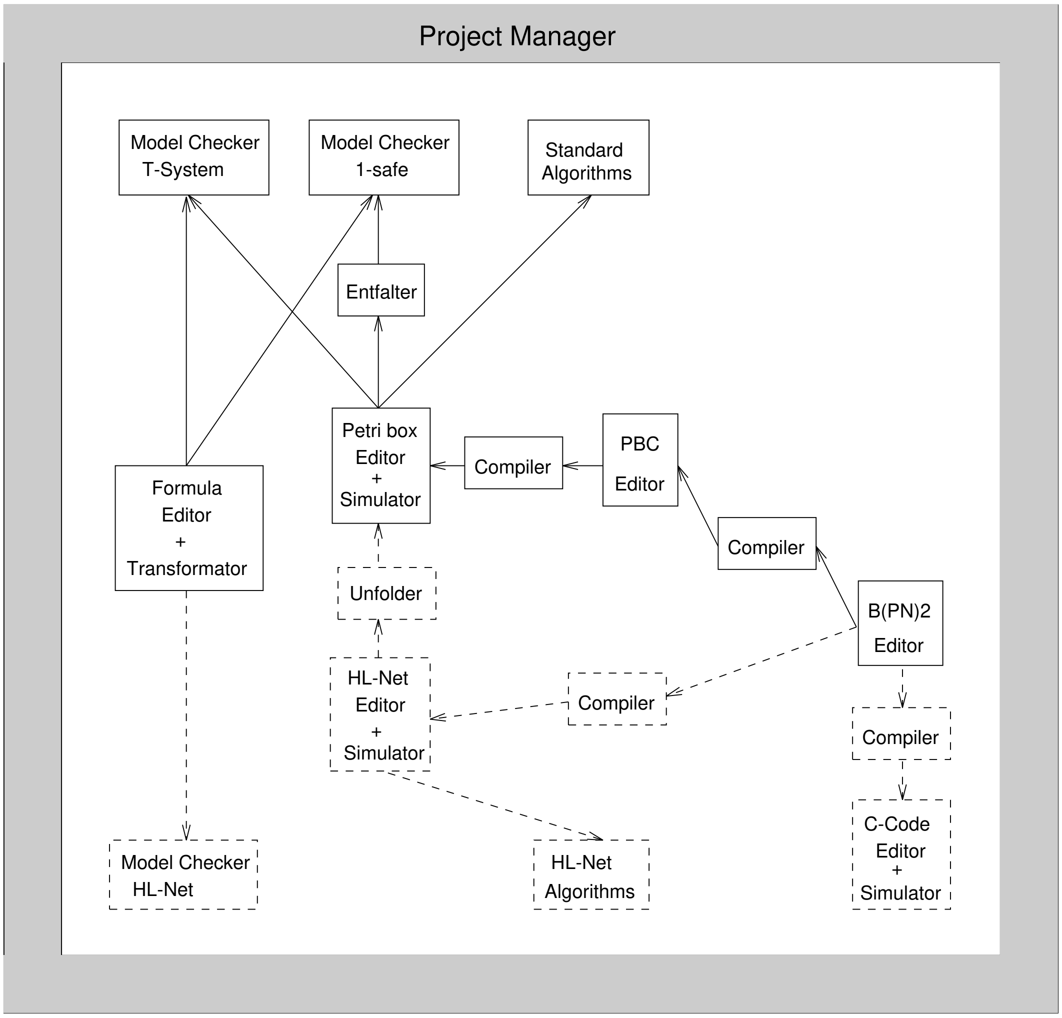


Figure 1.3.2 : *Interplay of the components of the* **PEP** *system. The parts of the system that are not yet implemented are indicated by dashed lines. The arrows are representing the input/output relation between the different components.*

5. The three existing model checking algorithms MC-0 for T-systems [36] and MC-1 and MC-2 [18] for 1-safe Petri nets, which can determine whether a Petri net satisfies a property given in terms of a temporal logic formula, will be completed by a new one for high-level Petri boxes. The transparency and thus the full functionality is reached by a formula transformator, which is able to transform an abstract formula (like "if a program terminates", "if at some point a variable has a certain value" or "if a transition is live") into a formula over the Petri box. This offers a comfortable interface to the model checkers.

Figure 1.3.2 shows the interplay and the interaction between these individual components, which are controlled by the project manager which builds the central part of the **PEP** system.

1.4 Typical cycle of usage

In the last sections the usage of the **PEP** tool has only been described indirectly. Now the main phases during usage are specified and the typical cycle of usage is illustrated.

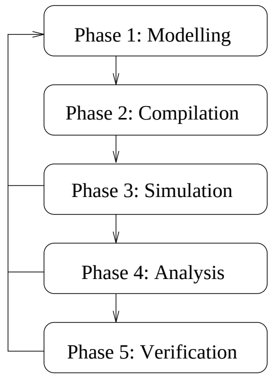


Figure 1.4.3 : **PEP** *cycle of usage*
Of course it is always possible to return directly to the modelling or any other phase, if one recognizes, a mistake.

Figure 1.4.3 shows that the interaction with the **PEP** system can be divided into five parts:

1. Usually the user wants to start by modelling a parallel system. As described he can use parallel prgrams ($B(PN)^2$ programs), terms in a process algebra (PBC programs), high-level or low-level Petri boxes.

2. Simulation, analysis and verification are only possible for Petri boxes. This is the reason for the necessity of the second phase. Depending on the chosen objects different compilers are used to calculate the corresponding Petri box.

3. The simulation offers the possibility to become familiar with the constructed parallel system represented by a Petri box, without spending too much time. Of course it can only mediate a first rough intuition.

4. Some more detailled information can be gained by the application of some standard analysis algorithms. Especially in the case of normal Petri nets it can be of great interest whether a net is live or 1-safe.

5. Finally the major focal point should be the verification phase. In comparison to other tools the verification component enables the checking whether one made a mistake during the modelling phase by verifying various properties. For example it is possible to check whether the mutual exclusion property holds for a written program.

1.5 A mutual exclusion algorithm

After the description of the **PEP** tool on a more abstract level in the first sections it is the aim of this section to show the practise. Without going into detail a rough intuition of the way one can work with the tool will be given and the reader should gain a first impression of what a

$B(PN)^2$ program and the corresponding Petri box look like. More details concerning the $B(PN)^2$ syntax and the related net semantics are given in [9].

```
begin

var b1,b2,c1,c2:[0,1] init 0;
var k:[1,2] init 1;
var in_cs1, in_cs2:[0,1] init 0;

do ⟨b1' = 1⟩;
   do ⟨b2 = 0⟩; exit
   □ ⟨b2 = 1⟩; ⟨c1' = 1⟩;
     do ⟨c2 = 1⟩; exit
     □ ⟨b2 = 0⟩;  exit
     od;
     ⟨k = 1⟩; ⟨c1' = 0⟩; exit
   od;
   ⟨in_cs1' = 1⟩; ⟨in_cs1' = 0⟩;
   ⟨k' = 2⟩; ⟨b1' = 0⟩; repeat
od
||
do ⟨b2' = 1⟩;
   do ⟨b1 = 0⟩; exit
   □ ⟨b1 = 1⟩; ⟨c2' = 1⟩;
     do ⟨c1 = 1⟩;  exit
     □ ⟨b1 = 0⟩; exit
     od;
     ⟨k = 2⟩; ⟨c2' = 0⟩; exit
   od;
   ⟨in_cs2' = 1⟩; ⟨in_cs2' = 0⟩;
   ⟨k' = 1⟩; ⟨b2' = 0⟩; repeat
od

end
```

Figure 1.5.4 : *$B(PN)^2$ program of the mutual exclusion algorithm*

One (at least for a theoretical computer scientist) typical example will be considered. Rolf Walter [38] found a new algorithm to insure mutual exclusion between two processes which is shown in figure 1.5.4 . This algorithm has the advantage of very “small“ atomic actions. Its disadvantage is, though, that its correctness is (independent from the formulation in $B(PN)^2$ or any other language) very difficult to proof. Thus it is a good example for a useful application of the **PEP** tool.

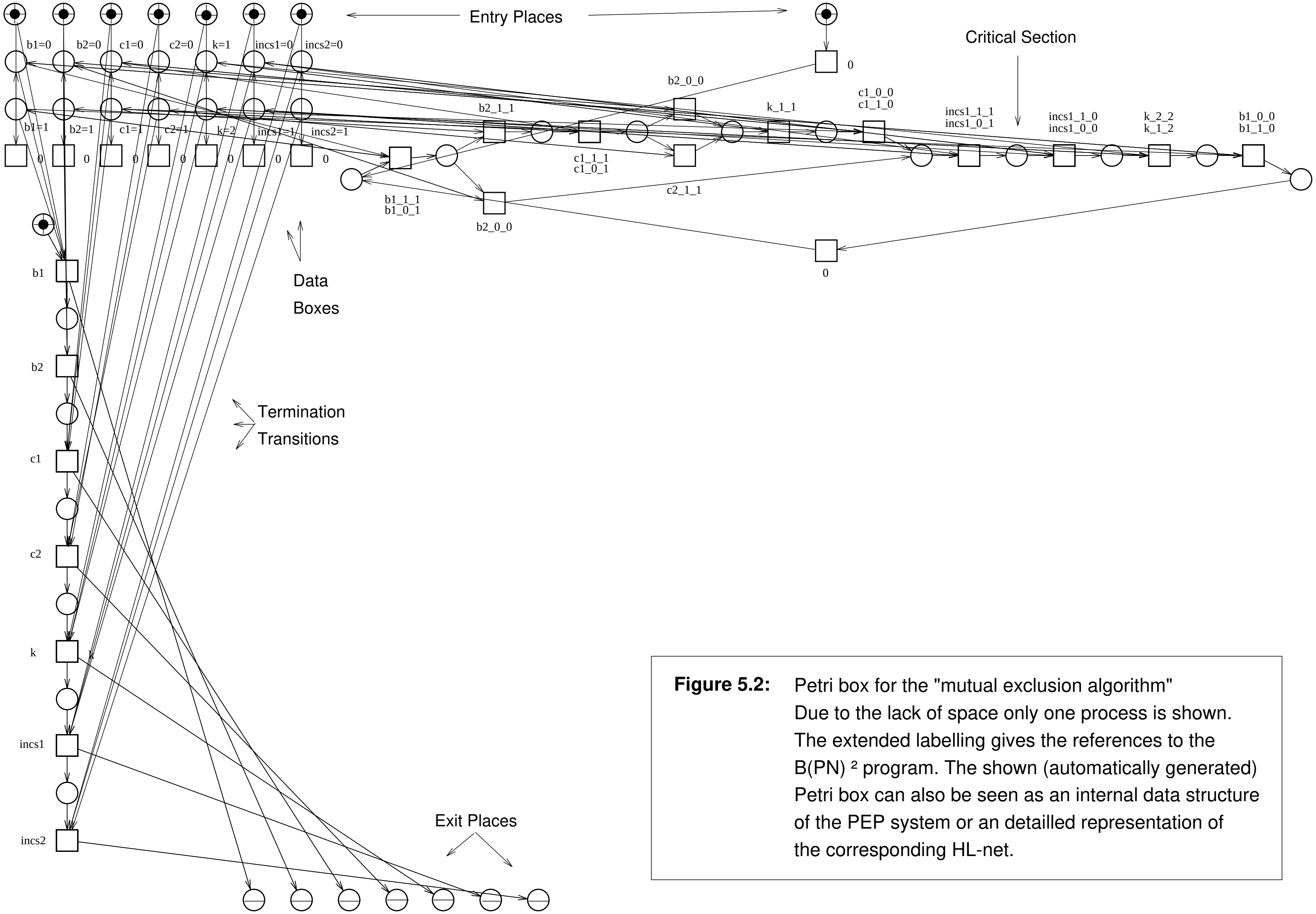


Figure 5.2: Petri box for the "mutual exclusion algorithm"
Due to the lack of space only one process is shown.
The extended labelling gives the references to the
B(PN) ² program. The shown (automatically generated)
Petri box can also be seen as an internal data structure
of the PEP system or an detailled representation of
the corresponding HL-net.

To understand the algorithm it is only necessary to know the semantics of the "do-od"-construct and the difference between pre-, post- or unprimed variables.

The following "do-od"-construct describes a loop with different alternatives which are chosen non-deterministically. During each pass one of the alternatives (with a precondition that is fulfilled) is chosen. If this alternative ends with "exit" the loop is left, if it ends with a "repeat" the loop will be entered once more.

do $\langle x = 0\rangle$; $\langle x' = 1\rangle$; exit
$\square$ $\langle x = 1\rangle$; $\langle x' = 0\rangle$; repeat
od;

A preprimed variable represents the value of the variable before the execution, a postprimed after and an unprimed before and after. An action $\langle 'x = 0\rangle$ can only be executed if the value of the variable "x" equals 0 before the execution. The action $\langle x' = 0\rangle$ sets the value of the variable "x" to 0. Finally an action $\langle x = 0\rangle$ does not change the value of the variable "x", but can only be executed if the value of the variable "x" equals 0 before the execution.

It should now be obvious that the mutual exclusion property in this example means that always at least one of the variables "in_cs_1" or "in_cs_2" has the value 0. In the **PEP** system this can be expressed as a formula in the following manner:

$\square(\text{'}in\_cs1 = 0\text{'} \vee \text{'}in\_cs2 = 0\text{'})$

The references between the $\mathrm{B(PN)}^2$ program and the corresponding net semantics (shown on the next page) that are available in the **PEP** system allow the transformation of this formula into a formula over the Petri box. Similarly the terms $\text{'}in\_cs1 = 0\text{'}$ and $\text{'}in\_cs2 = 0\text{'}$ are automatically replaced by the corresponding place names. Finally the model checker "MC-1" calculates whether the mutual exclusion property is satisfied.

The whole verification, including the compilation from the $\mathrm{B(PN)}^2$ program into the Petri box, the "Entfaltung", the transformation of the formula and the model checking takes less than 1 second. Consequently it can be stated that the **PEP** tool can deal with reliable results a lot faster than human beings can. And it is not even necessary to care about the PBC program or the Petri box at all, because it is sufficient to write a $\mathrm{B(PN)}^2$ program and to express the interesting property as a formula over the $\mathrm{B(PN)}^2$ program.

1.6 The efficiency of the PEP system

Compared to the last section the example considered now yields much bigger nets. This is caused by the usage of procedures and channels. Even though this example should just give an impression of the capacity of the verification component of the **PEP** system.

The solution for the problem of the "Tower of Hanoi" shown in figure 1.6.6 should be briefly explained. The algorithm tries to carry 5 slices from the first tower to one of the other towers. In every step only one slice can be moved and it can only be put on a tower with a smaller topmost slice. In each step a possible move is chosen non-deterministically. The towers are represented by channels, the topmost elements are stored in the variables "$O1$", "$O2$" and "$O3$" and the procedure "M" puts one slice from one tower to another. It is just because of the non-determinism that the verification is not easy.

```
begin

  var t1,t2,t3:   chan 5 of [1,5];
  var 01:         [1,6] init 1;
  var 02:         [1,6] init 6;
  var 03:         [1,6] init 6;
  var h1, h2:     [1,6];

proc M ( ref c1:chan 5 of [1,5], ref 01:[1,6],
         ref c2:chan 5 of [1,5], ref 02:[1,6] )
begin
   ⟨h1′ = c1?⟩; ⟨h2′ = 6⟩;
   do ⟨¬(h1 = 01)⟩; ⟨h2′ = h1⟩; ⟨c1! = h1⟩; ⟨h1′ = c1?⟩; repeat
   □  ⟨h1 = 01⟩ ; ⟨c2! = h1⟩; ⟨02′ = h1⟩; ⟨01′ = h2⟩; exit
   od
end;

⟨t1! = 5⟩; ⟨t1! = 4; ⟨t1! = 3; ⟨t1! = 2⟩; ⟨t1! = 1⟩;

do
   ⟨01 = 6 ∧ 02 = 6 ∨ 03 = 1⟩; exit
 □
   ⟨01 = 6 ∧ 02 = 1 ∧ 03 = 6⟩; exit
 □
   ⟨01 < 02⟩; M (t1, 01, t2, 02); repeat
 □
   ⟨01 < 03⟩; M (t1, 01, t3, 03); repeat
 □
   ⟨02 < 01 ∧ ¬(01 = 6 ∧ 02 = 1 ∧ 03 = 6)⟩; M (t2, 02, t1, 01); repeat
 □
   ⟨02 < 01 ∧ ¬(01 = 6 ∧ 02 = 1 ∧ 03 = 6)⟩; M (t2, 02, t1, 01); repeat
 □
   ⟨03 < 01 ∧ ¬(01 = 6 ∧ 02 = 6 ∧ 03 = 1)⟩; M (t3, 03, t1, 01); repeat
 □
   ⟨03 < 02 ∧ ¬(01 = 6 ∧ 02 = 6 ∧ 03 = 1)⟩; M (t3, 03, t2, 02); repeat
 od

end
```

Figure 1.6.6 : *A $B(PN)^2$ solution for the problem of the "Towers of Hanoi" using iterative procedures*

The property of the program working correctly means it is always possible that all the slices are on the second or the third tower after some more steps. This can be expressed directly as a formula over the $B(PN)^2$ program:

$$\Box\Diamond(('01 = 6' \text{ und } '02 = 6' \text{ und } '03 = 1') \vee ('01 = 6' \text{ und } '02 = 1' \text{ und } '03 = 6'))$$

$'O1 = 6'$ mean, that there is no slice left on the first tower.

n	|P|	|T|	|C|	|E|	$\rightsquigarrow$ Petri box	Entfaltung	Model checking	Gesamt
2	128	450	1 171	458	0.22	1.14	0.04	1.53
3	152	957	4 314	1 695	0.79	13.97	0.38	15.40
4	182	3 106	15 273	6 111	8.80	191.21	3.39	205.57
5	217	25 160	16 224	6 584	767.02	259.09	3.75	1 032.24

Figure 1.6.7 : *Time analysis of the iterative solution of the "Tower of Hanoi"-problem*
"$|P|$" and "$|T|$" are the numbers of places respectively transitions of the created Petri box for a height "n". "$|C|$" and "$|E|$" are the numbers of conditions respectively events in the "Entfaltung". Furthermore the time necessary to compute the Petri box, to calculate the "Entfaltung" to model check the formula and the time for the whole verification is given in seconds.

Table 1.6.7 contains information about the size of the corresponding Petri boxes and the time that is necessary to analyse the B(PN)^2 program. The 1032.24 seconds for the whole verification of the solution for 5 slices mediates an impression of the efficiency of the **PEP** system. A Petri box with more than 25 000 places and more than 16 000 transitions had been checked in less than 5 minutes (on a SUN SPARC10).

This was only possible because of one characteristic of the model checker. The algorithm does not compute the reachability graph but only needs the finite prefix of the branching process as input which is almost always smaller. Like that the "state explosion problem" can be avoided (see [18]).

1.7 Future work

During the work with the **PEP** system a lot of different analyzed examples showed that the tool is capable of handling even big examples. As a realistic one a protocol for telefone networks can be mentioned. On the other hand we also found some even small programs that we cannot verify so far. After all this leads us to the following aims of the future work:

1. A high-level net semantics as well as a model checker for high-level Petri boxes will be developped and implemented, since low-level nets for realistic problems are too big for an useful representation on screen let alone for a simulation.

2. A logic for B(PN)^2 will be developped to improve the possibility of describing properties in terms of the B(PN)^2 program. This is the only possibility to hide the Petri boxes from the user.

3. The language B(PN)^2 will be augmented by the addition of further abstract data types and what is more interesting the full extension of the procedure concept. That way the user will have the full power to model complex parallel systems in an easy way.

As a totally different approach new interfaces to the **PEP** system will be considered. "VHDL", "OCCAM" and finite automata seem to be suitable to extend the interfaces of the tool.

2 Ein effizienter Algorithmus zur Berechnung von Entfaltungen endlicher, 1-sicherer Petrinetze

Stefan Römer
Universität Hildesheim
Institut für Informatik
Marienburger Platz 22
D - 31141 Hildesheim
e-mail: roemer@informatik.uni-hildesheim.de

Abstract

It is reasonable suitable to describe the behaviour of systems (marked Petri nets) by giving their partial order semantics in terms of process nets. Branching processes are defined via condition/event nets, i.e. process nets, which are acyclic and allow forward branching but no backward branching of conditions.

A maximal branching process contains information about all possible runs of a system. Concerning reachability and deadlock properties it is sufficient to construct a finite prefix of this process, called unfolding, which is large enough to implicitly represent all reachable states of the system.

The algorithm for generating unfoldings of 1-safe systems, which is used within the framework of **PEP**, has been developed by K. L. McMillan [28]. We were able to increase efficiency of the implementation by correcting some insufficencies of the original algorithm. Furthermore a number of extensions have been implemented in order to faciliate the subsequent analysis of the unfolded system via model checking.

Some dedicated examples are used to present experimental results as well as to show the limits of the practical applicability of this algorithm.

2.1 Einführung

Zur Beschreibung des Verhaltens paralleler Systeme haben sich in den letzten Jahren zwei Ansätze herauskristallisiert: Interleaving–Semantiken beschreiben das Systemerhalten mit Hilfe von Sprachen; damit verbunden ist das bekannte „State–Explosion"–Problem, das sich aus dem Aufzählen der erreichbaren Zustände ergibt.

Beim zweiten Ansatz beschreiben Halbordnungssemantiken das Verhalten eines Systems mit einer Kausalrelation, die angibt, ob zwei Aktionen unabhängig voneinander (nebenläufig) ablaufen, oder ob die eine Aktion der anderen vorausgehen muß. Auf Ebene der Petri–Netze läßt sich beispielsweise mit Hilfe von Kausalnetzen (ohne Verzweigungen) [3] eine einfache Halbordnungssemantik definieren; damit sind einzelne Durchläufe eines Systems modellierbar.

In [31] werden Bedingungs–Ereignis–Netze eingeführt als Kausalnetze mit Vorwärtsverzweigungen; diese werden in [15] herangezogen zur Definition von Branching–Prozessen von Petri–Netzen. Mit Hilfe der letztgenannten lassen sich mehrere Ausführungen eines Systems in einem einzigen Objekt angeben, einschließlich nichtdeterministischer Alternativen. Die Menge der Branching–Prozesse eines Systems bildet einen vollständigen Verband (bezüglich der Präfixe der Prozesse als zugrundeliegende Halbordnung) [15]. Das größte Element eines solchen Verbandes, der maximale Branching–Prozeß, wird maximale Entfaltung genannt. In ihr sind alle möglichen (auch unverzweigten) Prozesse des Systems enthalten. Das Problem dieser Entfaltungen ist, daß sie selbst für Systeme mit einer endlichen Zahl erreichbarer Markierungen unendlich groß sein können.

McMillan hat in seiner Arbeit [28] für sichere, endliche Systeme einen Algorithmus zur Konstruktion eines endlichen Präfixes der maximalen Entfaltung angegeben, der groß genug ist, daß damit alle vom System erreichbaren Zustände beschrieben sind. Damit ist es möglich, einfache Eigenschaften des zugrundeliegenden Systems zu überprüfen, wie etwa Erreichbarkeit von Markierungen oder das Vorhandensein von Deadlocks. Aufbauend auf diesen Entfaltungen ist ein effizienter Model Checker entwickelt worden, der es gestattet, für die Klasse der 1–sicheren Systeme beliebige Formeln (in einer temporalen Logik ausgedrückt) zu verifizieren [17, 18].

Die Grundidee dieses Entfaltungs–Algorithmus ist die folgende: ausgehend von einer Menge von Bedingungen, die der Startmarkierung des Systems entspricht, wird der endliche Präfix sukzessive („schichtweise") um neue Knoten erweitert. Mit jedem Einfügen eines Ereignisses wird überprüft, ob die sich daraus ergebende Markierung (bzgl. des Systems) nicht schon einmal im vorderen Teil der Entfaltung berechnet worden ist. Ist dies der Fall, so braucht die Entfaltung hinter dem gerade eingefügten Ereignis nicht mehr erweitert zu werden, da alle dort folgenden Ereignisse bereits berechnete Markierungen produzieren würden. Ein Beweis für die Korrektheit des beschriebenen Verfahrens ist in [28] zu finden.

Ziel dieser Arbeit ist es, die eben beschriebene Idee in Form eines Algorithmus zu formalisieren, so wie er im Rahmen des Projektes **PEP** für den Model Checker implementiert worden ist. Daneben werden einige Aspekte betrachtet, die zu Effizienzverbesserungen gegenüber der Originalarbeit von McMillan geführt haben. Für die anschließende Verwendung der Entfaltung durch den Model Checker sind einige Erweiterungen des Basis–Algorithmus vorgenommen worden, die kurz dargestellt werden sollen.

Für die konkrete Implementierung fand die Programmiersprache „C" Verwendung; als Grundlage für die Verwaltung der Netze dienen die in [34] beschriebenen Datenstrukturen.

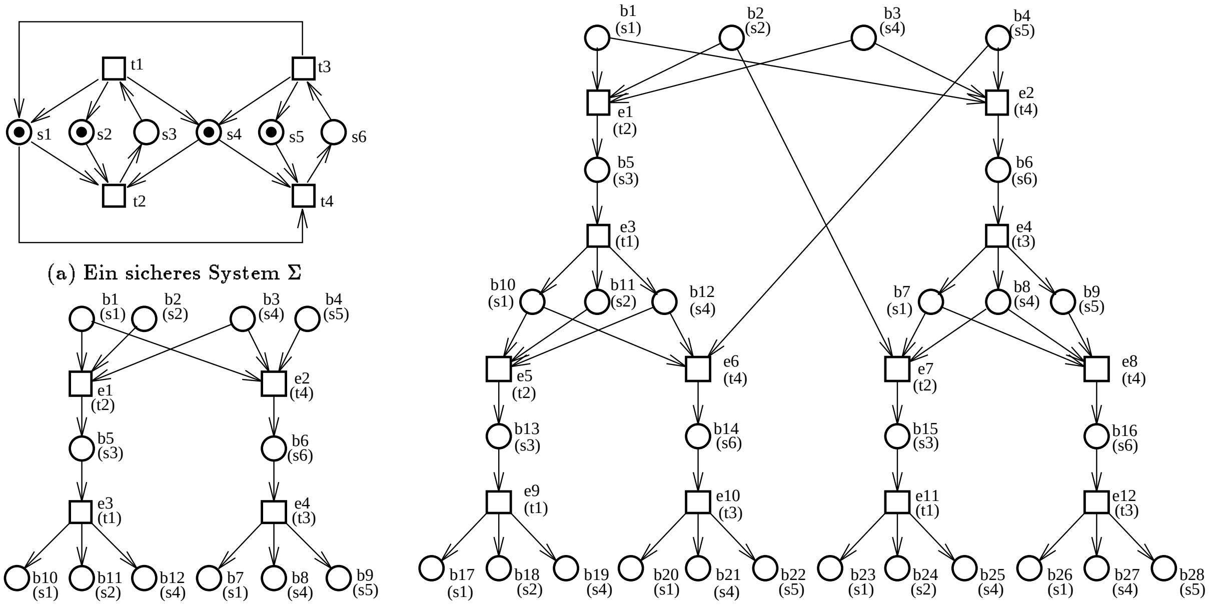
2.2 Grundlegende Begriffe

Bevor im nächsten Abschnitt der eigentliche Algorithmus dargestellt werden kann, ist die Definition einiger wichtiger Begriffe im Zusammenhang mit Branching–Prozessen notwendig.

Für die folgenden Ausführungen sei ein sicheres, endliches System $\Sigma = (S, T, F, M_0)$ vorausgesetzt (Abbildung 2.2.1 zeigt ein Beispiel für ein solches System). Der Branching–Prozess $\beta_m = (N_m, \mathrm{p}_m) = (B_m, E_m, F_m, \mathrm{p}_m)$ beschreibe die maximale Entfaltung dieses Systems, und $\beta_f = (B, E, F, \mathrm{p})$ bezeichne die Entfaltung von Σ, also das endliche Anfangsstück von β_m, das mit Hilfe des unten beschriebenen Algorithmus berechnet werden kann. Die Entfaltung des Beispielsystems sowie ein Ausschnitt aus der maximalen Entfaltung sind in Abbildung 2.2.1 angegeben. Die in den Prozeßnetzen in Klammern gesetzten Stellennamen stellen den Bezug zum Ausgangssystem (Abbildung p) her.

Eine (globale) Konfiguration ist eine Menge von Ereignissen, die einen möglichen teilweisen Durchlauf des zugrundeliegenden Systems repräsentieren. Mit jedem Ereignis e, das zu einer Konfiguration C gehört, sind auch alle Vorgänger von e, d. h.$\downarrow \{e\}$, in C enthalten. In einer Konfiguration können keine zwei Ereignisse in Konflikt stehen. Es ist unmittelbar einsichtig, daß eine Konfiguration endlich ist.

In Abbildung 2.2.1 (b) ist $C_1 = \{e_1, e_3, e_6\}$ ein Beispiel für eine Konfiguration. Die Mengen $C_2 = \{e_3, e_6\}$ und $C_3 = \{e_1, e_2, e_3, e_6\}$ bilden keine Konfigurationen, da das Vorgänger–Ereignis von e_3 fehlt bzw. Ereignisse e_1 und e_2 in Konflikt stehen.



(a) Ein sicheres System Σ

(c) Die Entfaltung β_f von Σ

(b) Ein endlicher Präfix der maximalen Entfaltung β_m von Σ

Abbildung 2.2.1 : *Beispiel für eine Entfaltung*

Die lokale Konfiguration eines einzelnen Ereignisses $e \in E$, $\mathrm{LC}(e)$, ist die Menge der Ereignisse, die Voraussetzung für das Eintreten von e sind, sie besteht also gerade aus den Vorgänger(–Ereignisse)n von e (inklusive e selbst):

$$\mathrm{LocalConfig}(e) = \mathrm{LC}(e) = E \cap \downarrow\{e\}.$$

In der maximalen Entfaltung β_m ist mit jeder Konfiguration C implizit ein B–Schnitt $\mathrm{cut}(C) = (C^\bullet \cup \mathrm{Min}(N_m)) \setminus^\bullet C$ verbunden, der, vermöge der Beschriftungsfunktion p_m auf das System Σ projiziert, einer erreichbaren Markierung entspricht, die im folgenden als Endzustand von C bezeichnet wird.

Der mit der Konfiguration C_1 von oben verbundene Endzustand ist $\{s_2, s_6\}$; der Endzustand von $C_4 = \{e_2, e_4\}$ ist $\{s_1, s_2, s_4, s_5\}$ und entspricht gerade der Startmarkierung M_0.

Das Abbruchkriterium für die Konstruktion der Entfaltung β_f bildet das Erkennen sogenannter Cutoff–Ereignisse. Damit werden solche Ereignisse e im Maximum von β_f bezeichnet, deren lokale Konfiguration mit einem Endzustand im Originalnetz verbunden ist, der von einem bereits in die Entfaltung aufgenommenen Ereignis mit (bezüglich der Größe) kleinerer lokaler Konfiguration erreicht werden kann, d. h.es gibt ein Ereignis e' im bereits konstruierten Teil des Präfixes mit $\mathrm{p}_m(\mathrm{cut}(\mathrm{LC}(e'))) = \mathrm{p}_m(\mathrm{cut}(\mathrm{LC}(e)))$ und $\mathrm{LC}(e') \subset \mathrm{LC}(e)$ (echte Teilmenge). Dabei wird die Startmarkierung von der leeren Konfiguration impliziert. Ist die maximale Entfaltung endlich, so existieren keine Cutoff–Ereignisse; in diesem Fall stimmen Entfaltung β_f und maximale Entfaltung β_m überein.

Die Cutoff-Ereignisse des Netzbeispiels aus Abbildung 2.2.1 sind e_3 und e_4.

2.3 Der Entfaltungs-Algorithmus

Zur besseren Übersichtlichkeit ist der Algorithmus in drei Teile zerlegt worden. Der erste Teil beschreibt das Hauptprogramm mit der Initialisierung der verwendeten globalen Variablen und der Entfaltung, sowie die Hauptschleife.

Der zweite Teilalgorithmus beschreibt die Entscheidungsfindung zum Generieren neu aufzunehmender Ereignisse. Im dritten Teil wird schließlich die Funktion zur Überprüfung der Cutoff-Eigenschaft angegeben.

Algorithmus 2.1 *Entfalten eines endlichen und sicheren Petri-Netzes*

Eingabe: Endliches, sicheres System $\Sigma = (S,\, T,\, F,\, M_0)$ mit Startmarkierung $M_0 \subseteq S$.
Ausgabe: Entfaltung $\beta_f = (B,\, E,\, F',\, p)$.
cutoffs, die Menge der Cutoff-Ereignisse,
configs, die Menge der globalen Konfigurationen von β_f.

```
 1 PROCEDURE unfold():
 2   /* Globale Variablen initialisieren: */
 3   x := 0; y := 0; /* Zähler für neu eingefügte Bedingungen bzw. Ereignisse */
 4   B := ∅; E := ∅; F' := ∅;
 5   cutoffs := ∅; configs := ∅; finalstates := ∅;
 6   queue := λ;
 7   /* Startmarkierung des Netzes in die Entfaltung übertragen: */
 8   FOR ALL s ∈ M_0 DO
 9     x := x + 1; B := B ∪ { b^x }; p(b^x) := s;
10     GenEvents(b^x)
11   ENDFOR;
12   /* Hauptschleife: */
13   WHILE queue ≠ λ DO
14     e := get_first( queue );
15     IF IsCutoffPoint(e) THEN
16       cutoffs := cutoffs ∪ { e }
17     ELSE
18       FOR ALL s ∈ p(e)• DO
19         x := x + 1; B := B ∪ { b^x }; p(b^x) := s; F' := F' ∪ { (e, b^x) };
20         GenEvents( b^x )
21       ENDFOR
22     ENDIF
23   ENDWHILE;
24   /* Bedingungen in den Nachbereichen der Cutoff-Ereignisse generieren: */
25   FOR ALL e ∈ cutoffs DO
26     FOR ALL s ∈ p(e)• DO
27       x := x + 1; B := B ∪ { b^x }; p(b^x) := s; F' := F' ∪ { (e, b^x) }
28     ENDFOR
29   ENDFOR
30 ENDPROC
```

■ 2.1

Die verwendeten Operationen (wie $^\bullet$, p(),...) sind so zu verstehen, daß diese nur auf dem bereits konstruierten Präfix arbeiten (und nicht etwa auf der maximalen Entfaltung β_m).

Nach dem Übertragen der Startmarkierung des Systems in die Entfaltung wird die Hauptschleife, die das Einfügen neuer Ereignisse steuert, solange durchlaufen, wie noch Ereignisse generiert werden können, die eine bislang unbekannte Markierung (Endzustand) in die Entfaltung einbringen können. Für jedes eingefügte Ereignis, welches nicht die Cutoff-Eigenschaft besitzt, werden am Ende der Hauptschleife die entsprechenden Bedingungen in dessen Nachbereich generiert.

Die Schlange *queue* wird innerhalb der Prozedur „GenEvents()“ mit jedem Einfügen eines neuen Ereignisses in die Entfaltung um gerade dieses Ereignis erweitert. Dabei wird für die Schlange die folgende Sortierreihenfolge aufrechtgehalten: die Ereignisse werden nach der Tiefe der jeweils verbundenen lokalen Konfiguration eingeordnet, wobei Elemente mit der geringsten Tiefe als erste aus der Schlange genommen werden (in Zeile 13 des Haupt-Algorithmus 2.1). Mit diesem Vorgehen wird garantiert, daß die Entfaltung sukzessive schichtweise konstruiert wird.

Algorithmus 2.2 ist der zentrale Teil der gesamten Entfaltungsprozedur; aus Effizienzgründen ist dieser Teil gegenüber [28] vollständig geändert worden.

Algorithmus 2.2 *Erzeugen neuer Ereignisse*

Eingabe: System $\Sigma = (S, T, F, M_0)$, Präfix $\beta_i = (B, E, F', p)$ der Entfaltung β_f
Bedingung $b \in B$, zu der Ereignisse generiert werden sollen.

Ausgabe: β_i, erweitert um Ereignisse im Nachbereich von b; die neu eingefügten Ereignisse sind außerdem in *queue* eingeordnet.

```
PROCEDURE GenEvents( b ):
  FOR ALL t ∈ p(b)• DO
    IF |•t| = 1 THEN /* konfliktfrei */
      /* Neues Ereignis mitsamt eingehender Kante einfügen */
      y := y + 1; E := E ∪ { e^y }; p(e^y) := t; F' := F' ∪ { (b, e^y) };
      insert_queue(queue, e^y);
      Configuration(e^y)
    ELSIF |•t| = 2 THEN
      { s } := •t \ {p(b)};
      FOR ALL b' ∈ p^-1(s) : b' co b DO
        y := y + 1; E := E ∪ { e^y }; p(e^y) := t;
        F' := F' ∪ { (b, e^y), (b', e^y) };
        insert_queue(queue, e^y);
        Configuration(e^y)
      ENDFOR
    ELSE /* |•t| ≥ 3 */
      n := |•t| − 1; { s_1, ..., s_n } := •t \ {p(b)};
      /* K enthält die Menge der Kombinationen von Bedingungen, */
      /* die in co-Relation zur aktuell betrachteten Bedingung b stehen. */
      K := { (b_1, ..., b_n) | b_i ∈ p^-1(s_i), b co b_i, 1 ≤ i ≤ n };
      FOR ALL (b_1, ..., b_n) ∈ { (b_1, ..., b_n) ∈ K | b_i co b_j, 1 ≤ i < j ≤ n } DO
        y := y + 1; E := E ∪ { e^y }; p(e^y) := t;
        F' := F' ∪ { (b, e^y), (b_1, e^y), ..., (b_n, e^y) };
        insert_queue(queue, e^y);
        Configuration(e^y)
      ENDFOR
    ENDIF
  ENDFOR
ENDPROC
```

■ 2.2

Die Aufrufe der Prozedur „Configuration()“ werden im nächsten Abschnitt erläutert; für die Konstruktion der Entfaltung haben sie keine Bedeutung.

Die äußere Schleife dieser Funktion betrachtet alle Transitionen t des Systems, die im Nachbereich des Bildes $\mathrm{p}(b)$ der der Prozedur übergebenen Bedingung b liegen; nur Ereignisse, die diesem t im System Σ entsprechen, sind potentielle Kandidaten für eine Erweiterung der Entfaltung. Damit ist der „Suchraum“ gegenüber der von McMillan vorgeschlagenen Prozedur „GenTrans()“ [28] schon erheblich eingeschränkt.

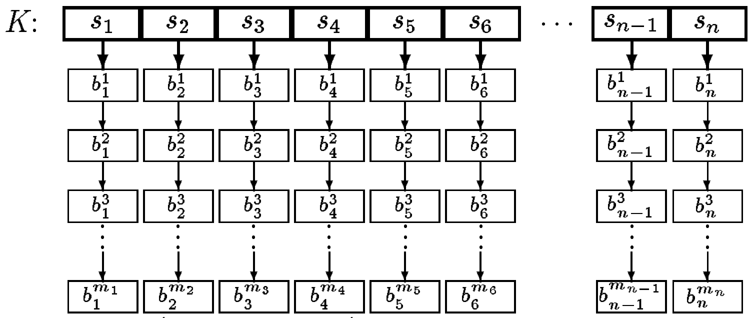
Für den einfachen Fall, daß $|{}^\bullet t| = 1$ gilt, kann unmittelbar ein entsprechendes Ereignis im Nachbereich von b generiert werden, da keinerlei Gefahr für einen Konflikt besteht.

Falls $|{}^\bullet t| = 2$ ist, muß dagegen sichergestellt werden, daß durch das Einfügen eines Ereignisses kein Konflikt produziert wird. Dazu wird jede in Frage kommende Bedingung b' aus der Menge $\mathrm{p}^{-1}(s)$ (im bereits konstruierten Präfix) dahingehend überprüft, ob sie mit b in co-Relation steht. Nur in diesem Fall darf ein gemeinsames Ereignis in den Nachbereich von b und b' aufgenommen werden.

Im allgemeinen Fall, $|{}^\bullet t| \geq 3$, sind die Kombinationen aller in Frage kommenden Bedingungen dahingehend zu überprüfen, daß durch das Einfügen eines gemeinsamen Ereignisses in den Nachbereichen kein Konflikt hergestellt wird. Um die Anzahl dieser möglichen Kombinationen zu senken, werden nur solche Bedingungen betrachtet, die nicht schon mit b in Konflikt stehen. Diese Überprüfung ist jeweils in $O(1)$ durchführbar, wenn zuvor mittels Tiefensuche alle möglichen „Konfliktpartner“ von b in einem Bit-Array markiert werden.

Die Menge K wird dabei keineswegs vollständig berechnet; ihr liegt die in Abbildung 2.3.2 angedeutete Datenstruktur zugrunde. Ein dynamisches Feld verwaltet für jede Stelle s' im Vorbereich von t eine Menge (Liste) der Urbilder $b' \in \mathrm{p}^{-1}(s')$, wobei nur solche Bedingungen in die Listen aufgenommen werden, die in co-Relation zu b stehen. Damit wird ebenfalls nur eine Auswahl an Kandidaten für die anschließende kombinatorische Zusammenstellung von Bedingungsmengen für die Aufnahme neuer Ereignisse übriggelassen. Die Ermittlung der Kombinationen erfolgt direkt auf dieser Datenstruktur, wobei bei glücklicher Plazierung der Bedingungen in den Listen längst nicht alle Möglichkeiten getestet werden müssen.

Der Fall $|^\bullet(t)| = 2$ hätte auch dem allgemeinen Fall untergeordnet werden können; da jedoch bei den meisten Systemen ein Großteil der Transitionen nur ein bzw. zwei Stellen im Vorbereich besitzen, ist es sinnvoll, dafür eine optimierte Variante zur Verfügung zu stellen. Einige Netze, wie beispielsweise das zur Modellierung des n-Buffers, bestehen sogar nur aus Transitionen mit ≤ 2 Stellen im Vorbereich; hier macht sich der Geschwindigkeitsvorteil besonders bemerkbar.



Es sind $b_i^j \in \mathrm{p}^{-1}(s_i)$; alle b_i^j stehen in co-Relation mit der aktuell betrachteten Bedingung b

Abbildung 2.3.2 : *Realisierung der Menge K*

Wie die programminterne Realisation der Funktion p (insbesondere p^{-1}) sowie der Relationen co und # gelöst worden ist, soll an dieser Stelle nicht näher dargestellt werden, da dazu ein tieferer Einblick in die benutzten Datenstrukturen [34] notwendig ist.

Die Einträge der Liste *finalstates* bestehen aus Paaren (f, s), wobei f eine Markierung darstellt und s die Größe der kleinsten lokalen Konfiguration, die diese Markierung als Endzustand impliziert.

Die Berechnung der Endzustände einer lokalen Konfiguration in Zeile 2 erfolgt hierbei auf eine etwas andere Art (ohne den Bezug zu β_m) als in Abschnitt 2.1 beschrieben. Da die Entfaltung sukzessive konstruiert wird, kann ein Ereignis, das auf die Cufoff-Eigenschaft hin überprüft werden soll, noch keine Bedingungen im Nachbereich besitzen. Um den korrekten Endzustand für eine Konfiguration zu ermitteln, muß die Funktion FinalState() folgendermaßen aussehen:

$$\mathrm{FinalState}(e) = \mathrm{p}(e)^\bullet \cup \mathrm{p}(\,(\,\mathrm{LC}(e)^\bullet \cup \mathrm{Min}(N)\,)\,\setminus {}^\bullet\mathrm{LC}(e)\,)\,.$$

Algorithmus 2.3 *Teste Ereignis auf Cutoff-Eigenschaft*

Eingabe: System $\Sigma = (S, T, F, M_0)$, Präfix $\beta_i = (B, E, F', p)$ der Entfaltung β_f,
das zu überprüfende Ereignis e,
globale Liste *finalstates*, in der die bereits ermittelten erreichbaren Markierungen zusammen mit der minimalen Größe einer sie implizierenden Konfiguration abgelegt sind.

Ausgabe: „True“, falls e ein Cutoff-Ereignis ist; andernfalls wird „False“ zurückgegeben und die Markierung in *finalstates* mitsamt Größe der Konfiguration eingefügt.

```
1  bool FUNCTION IsCutoffPoint( e ):
2    LC_size := |LocalConfig(e)|;
3    FS := FinalState(e);
4    IF ∃(f, s) ∈ finalstates : f = FS THEN
5      IF LC_size > s THEN
6        RETURN True
7      ELSE
8        Replace (f, s) in finalstates by (f, LC_size);
9        RETURN False
10     ENDIF
11   ENDIF
12   finalstates := finalstates ∪ { (FS, LC_size) };
13   RETURN False
14 ENDFUNC
```

■ 2.3

Bei der Implementation dieser letztgenannten Funktion wurde auf die in [34] beschriebene Tiefensuche zurückgegriffen; dieses Vorgehen hat sich als effizienter erwiesen als eine Realisierung mittels Listen-/Mengenoperationen. Die Ermittlung der Größe der lokalen Konfiguration (in Zeile 2 von Alg. 2.3) konnte auf diese Weise in die Berechnung des Endzustandes integriert werden.

Im Gegensatz zu [28] werden die Endzustände selbst (als lineare Listen oder Bit-Arrays) im Speicher abgelegt; zum schnelleren Auffinden können die Markierungen wie in [28] beschrieben in einer Hash-Tabelle organisiert werden.

2.4 Erweiterungen für den Model Checker

In diesem Abschnitt sollen kurz einige Erweiterungen beschrieben werden, die zu dem vorher beschriebenen Hauptalgorithmus vorgenommen worden sind, um die erzeugte Entfaltung eines Netzes für den Model Checker [17] als Eingabe weiterverwenden zu können. Die ergänzten Berechnungen hätten auch in den eigentlichen Model Checker integriert werden können; sie sind jedoch in die Berechnung der Entfaltung vorgelagert worden, da es sich um solche Informationen handelt, die unabhängig von der später zu überprüfenden logischen Formel nur ein einziges Mal für eine Entfaltung berechnen werden müssen. Außerdem sind die Berechnungen an dieser Stelle einfacher, da sie direkt mit der Konstruktion der Entfaltung erfolgen können.

Zunächst erwartet der Model Checker die Menge der maximalen globalen Konfigurationen von β_f, hier mit *configs* bezeichnet. Algorithmus 2.4 ermittelt diese Menge sukzessive mit jedem Einfügen eines neuen Ereignisses in die Entfaltung. Ein Aufruf dieser Prozedur erfolgt in Zeile 7 bzw. 14 von Algorithmus 2.2.

Dieser Algorithmus ist zur Zeit noch etwas ineffizient, da für jedes Ereignis alle bereits berechneten Konfigurationen betrachtet werden.

Für den Model Checker ist neben dieser Berechnung zu jedem einzelnen Cutoff-Ereignis e ein sogenanntes korrespondierendes Ereignis e' zu ermitteln. e' korrespondiert mit e, falls es den gleichen Endzustand impliziert; e' kann „leer“ sein, falls der Endzustand von e der Anfangsmarkierung M_0 entspricht. Die korrespondierenden Ereignisse werden in der Initialisierungsphase des Model Checkers zum Aufbau eines Isomorphismus benötigt [17]. Ihre Berechnung läßt sich durch Verwalten einer zweiten Liste parallel zu den Cutoff-Ereignissen leicht lösen und soll hier nicht näher ausgeführt werden.

Algorithmus 2.4 *Berechne globale Konfigurationen*

Eingabe: Präfix $\beta_i = (B, E, F', p)$ der Entfaltung β_f,
configs = Menge der bereits berechneten Konfigurationen,
zuletzt in die Entfaltung eingefügtes Ereignis e.
Ausgabe: *configs* erweitert um diejenigen Konfigurationen, die e enthalten.

```
PROCEDURE Configuration( e ):
  new := ∅;
  pred := •(•e);
  succ := E ∩ ↑(•e);
  IF configs = ∅ THEN
    configs := { {e} }
  ELSE
    FOR ALL C ∈ configs DO
      IF pred ⊆ C THEN
        IF ∃e' ∈ C : e#e' THEN
          C := C \ succ
        ELSE
          configs := configs \ { C }
        ENDIF;
        C := C ∪ { e };
        new := new ∪ { C }
      ENDIF
    ENDFOR;
    configs := configs ∪ new;
  ENDIF
ENDPROC
```

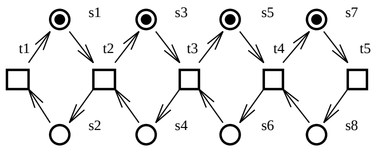
■ 2.4

2.5 Beispiel und Ausblick

Abschließend soll die Laufzeit des Algorithmus (inklusive der Ermittlung der Konfigurationen) anhand des n-Buffer-Beispiels dargestellt werden (Abbildung 2.5.3). In der allgemeinen Form besteht das Netz des n-Buffers aus $2n$ Stellen und $n+1$ Transitionen; die Entfaltung besteht aus $|B| = n^2 + n + 1$ Bedingungen und $|E| = \frac{|B|+1}{2} = \frac{n^2}{2} + \frac{n}{2} + 1$ Ereignissen. Die Entfaltung besitzt genau ein Cutoff-Ereignis (im Beispiel $n = 4$ ist dies $e7$), das stets mit der Startmarkierung korrespondiert, und genau eine maximale Konfiguration, bestehend aus allen Ereignissen. Die Anzahl der erreichbaren Markierungen dieser Klasse von Systemen ist bekannterweise 2^n. Zu beachten bei diesem Beispiel ist, daß in der Entfaltung keine Konflikte existieren.

Die Größe der Entfaltung läßt sich im allgemeinen nur schwer anhand der Größe des zugrundeliegenden Systems abschätzen. Der Zeitbedarf zur Berechnung läßt sich aber bei bekannter Größe der Entfaltung nach oben hin abschätzen, er ist proportional zu $O((\frac{|B|}{i})^i)^2$, wobei $|B|$ die Anzahl der Bedingungen angibt und $i = \max\{|^\bullet t| \mid t \in T\}$ den maximalen Rückwärtsverzweigungsgrad der Transitionen des Systems. Diese *worst case*-Schranke kommt jedoch nur bei Entfaltungen zum Tragen, bei denen sehr viele Transitionen des Systems einen hohen Verzweigungsgrad aufweisen und für die kombinatorische Ermittlung der Bedingungsmengen stets eine große Anzahl von Bedingungen betrachtet werden muß; im Durchschnitt ist der Algorithmus wesentlich schneller, insbesondere bei Netzen, die viele Konflikte aufweisen.

[2] Dieser Faktor ergibt sich aus der Zahl der maximal zu berechnenden Kombinationen von Bedingungen (Menge K in Algorithmus 2.2).



n	Zeit
10	0.02 s
20	0.25 s
30	1.30 s
40	3.62 s
50	9.60 s
60	19.08 s
70	34.93 s
80	59.70 s
90	96.18 s
100	147.90 s
110	217.03 s
120	307.85 s
130	428.13 s
140	577.44 s
150	759.99 s
160	986.13 s
170	1257.61 s
180	1583.32 s
190	1968.07 s
200	2427.31 s
210	2993.50 s
220	3596.59 s

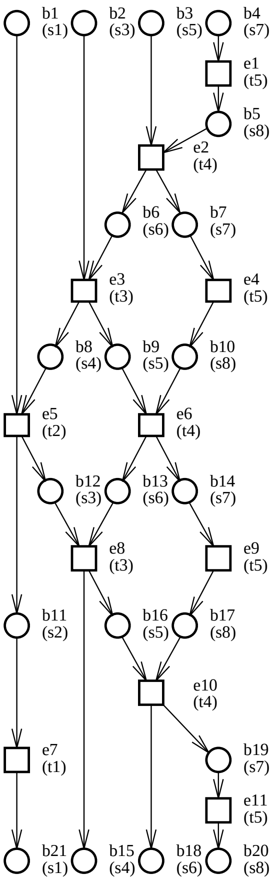


Abbildung 2.5.3 : *Beispiel: n-Buffer; für $n = 4$ System und Entfaltung; Laufzeiten*

Für die Zukunft ist geplant, insbesondere die Ermittlung der maximalen Konfigurationen zu beschleunigen, wobei dieser Teil dann möglichst unabhängig von der eigentlichen Berechnung der Entfaltung gemacht werden soll.

2.6 Anhang: Grundlegende Definitionen

2.6.1 Netze

Ein Petri-Netz ist ein Tripel $N = (S, T, F)$ mit $S \cap T = \emptyset$ und $F \subseteq (S \times T) \cup (T \times S)$. S ist die Menge der Stellen, T die Menge der Transitionen und F die Flußrelation (nur einfache Kantengewichte). In dieser Arbeit wird neben der Flußrelation F häufig auch die charakteristische Funktion F verwendet: $(x, y) \in F \Leftrightarrow F(x, y) = 1$, $(x, y) \notin F \Leftrightarrow F(x, y) = 0$ für $x, y \in S \cup T$.

Der Vorbereich (*pre-set*) von $x \in S \cup T$ ist ${}^\bullet x = \{ y \in S \cup T \mid (y, x) \in F \}$; der Nachbereich (*post-set*) von x ist $x^\bullet = \{ y \in S \cup T \mid (x, y) \in F \}$. Die Verallgemeinerung auf Mengen $X \subseteq S \cup T$ ist definiert als ${}^\bullet X = \bigcup_{x \in X} {}^\bullet x$ bzw. $X^\bullet = \bigcup_{x \in X} x^\bullet$. Ein Netz N ist endlich, falls $|S \cup T| \in \mathbb{N}$ gilt.

Eine Funktion $M : S \to \mathbb{N}$ heißt Markierung. Ein System ist ein Quadrupel $\Sigma = (S, T, F, M_0)$, wobei $N = (S, T, F)$ ein Netz ist und M_0 eine Markierung (Startmarkierung des Systems).

Eine Transition $t \in T$ ist aktiviert unter einer Markierung M gdw. $\forall s \in {}^\bullet t : M(s) \geq 0$ gilt; eine aktivierte Transition kann schalten und somit eine Nachfolgemarkierung M' erzeugen mit $M'(s) = M(s) + F(t, s) - F(s, t)$ für alle $s \in S$ (geschrieben $M\,[\,t > M')$.

Eine Ereignissequenz ist eine Folge $M_0 t_1 M_1 t_2 M_2 \dots t_r M_r$, so daß $\forall i, 1 \leq i \leq r : M_{i-1} [\, t_i > M_i$ gilt; M_r ist dann von M_0 aus erreichbar. $[\, M_0 >$ bezeichnet die Menge aller von M_0 aus erreichbaren Markierungen.

Ein System Σ ist (1-)sicher gdw. $\forall s \in S\ \forall M \in [\, M_0 > : M(s) \leq 1$ gilt. Abkürzend wird bei sicheren Netzen eine Markierung als Menge $M \subseteq S$ angegeben: $s \in M \Leftrightarrow M(s) = 1$.

2.6.2 (Branching-)Prozesse

Sei $N = (B,\ E,\ F')$ ein Netz. N heißt Bedingungs-/Ereignis-Netz (kurz B/E-Netz, *occurrence net*), falls $F' \subseteq (B \times E) \cup (E \times B)$ (nur einfache Kantengewichte), $F'^{+} \cap (F'^{-1})^{+} = \emptyset$ (Azyklizität) und $\forall b \in B : |^{\bullet}b| \leq 1$ (keine Rückwärtsverzweigungen in Bedingungen) gilt. Die Elemente aus B heißen Bedingungen, Elemente aus E sind Ereignisse. Wegen der Azyklizität liegt jedem B/E-Netz eine Halbordnung $(\ X,\ \preceq\ ) = (\ B \cup E,\ F'^{*}\ )$ zugrunde.

Sei $X = B \cup E$. Die Menge der minimalen Elemente eines B/E-Netzes ist definiert als $\mathrm{Min}(N) = \{\, x \in X \mid X \cup {}^{\bullet}x = \emptyset \,\}$. Die Menge der Vorgänger von $Y \subseteq X$ ist definiert als $\downarrow Y = \{\, x \in X \mid \exists y \in Y : x \preceq y \,\}$; entsprechend ist die Menge von Nachfolgern durch $\uparrow Y = \{\, x \in X \mid \exists y \in Y : y \preceq x \,\}$ gegeben.

Zwei Elemente $x, y \in X$ stehen in Konflikt (kurz $x \# y$) gdw. $\exists e_1, e_2 \in E,\ e_1 \neq e_2,\ {}^{\bullet}e_1 \cap {}^{\bullet}e_2 \neq \emptyset : (\ e_1 \preceq x \wedge e_2 \preceq y\ )$; x und y stehen in co-Relation ($x \,\mathrm{co}\, y$), falls $\neg(x \# y \vee x \preceq y \vee y \preceq x)$ gilt. Eine Menge $c \subseteq Y$ heißt co-Menge, falls $\forall x, y \in c : x \,\mathrm{co}\, y$ gilt. Ein Schnitt $c \subseteq Y$ ist eine maximale co-Menge. Ein Schnitt heißt B-Schnitt, falls er zusätzlich $c \subseteq B$ erfüllt.

Sei $\Sigma = (S,\ T,\ F,\ M_0)$ ein System, sei $N = (B,\ E,\ F')$ ein B/E-Netz und sei $p : B \cup E \to S \cup T$ ein Homomorphismus (Beschriftungsfunktion). Das Paar $\beta = (\ N,\ p\ )$ heißt Branching-Prozeß (verzweigender Prozeß, *branching process*) von Σ gdw. die folgenden Prozeß-Axiome erfüllt sind [3], [15]:

(1) $\mathrm{Min}(N)$ ist ein B-Schnitt von N,
(2) N ist $\mathrm{Min}(N)$-diskret [3]
(d. h.jeder Pfad eines Elementes zum Minimum hat endliche Länge),
(3) $p(B) \subseteq S,\ p(E) \subseteq T$ (p ist sortenrichtig),
(4) $\forall e \in E\ \forall s \in S : F(s,\ p(e)) = |p^{-1}(s) \cup {}^{\bullet}e| \ \wedge\ F(p(e),\ s) = |p^{-1}(s) \cup e^{\bullet}|$
(Erhalt lokaler Umgebungen von Transitionen),
(5) $\forall s \in S : M_0(s) = |p^{-1}(s) \cap \mathrm{Min}(N)|$
(Min(N) korrespondiert mit der Startmarkierung M_0) und
(6) $\forall e_1, e_2 \in E : (\ (\ {}^{\bullet}e_1 = {}^{\bullet}e_2 \ \wedge\ p(e_1) = p(e_2)\ ) \Rightarrow e_1 = e_2\ )$.
(Branching-Prozeß-Eigenschaft [15]).

3 Modelchecking als Komponente der Petrinetzbasierten Entwicklungs- und Programmierumgebung PEP

Thomas Thielke
Universität Hildesheim
Institut für Informatik
Marienburger Platz 22
D - 31141 Hildesheim
e-mail: thielke@informatik.uni-hildesheim.de

Abstract

The use of Petri nets for modelling highly parallel systems, especially parallel programs, becomes more and more popular in theoretical as well as practical regards. Verification of system properties often is a severe problem. Due to the state explosion problem most algorithms for analysis fail even for small, non-trivial examples. Model checking - a general approach for analysis - the algorithmic determination of truth or falsehood of a temporal logic formula with respect to a system is an NP-hard problem.

We describe the implementation of an efficient model checker for 1-safe T-systems called MCA as well as its applications and limits when being used in the framework of **PEP**. The basic idea of the algorithm is to avoid the generation of the reachability tree by the suitable manipulation of a temporal logic formula and its evaluation directly on the system rather than the tree. The implementation is ploynomial in the size of the system, but exponential in the size of the formula. The latter is not crucial, because in general meaningful and interesting formulae are small in comparison with the size of the systems in question.

3.1 Einführung und Motivation

Der Einsatz von Petrinetzen für die Modellierung von Systemen mit einem hohen Grad von Parallelität, insbesondere parallelen Programmen, gewinnt in der Theorie, aber auch in der Praxis zunehmend an Bedeutung. Die Verifikation von Eigenschaften des entwickelten Systems stellt oft ein nicht unerhebliches Problem dar. Schon die Betrachtung kleinerer nicht trivialer Beispiele läßt viele Analysealgorithmen an ihre Grenzen stoßen. Oft ist hierfür das bekannte "State-Explosion-Problem" ein Grund. Nicht zuletzt deshalb beschränkt sich in der Praxis die Analysekomponente eines Petrinetzwerkzeuges meist auf die Betrachtung einer Testmenge und eines darauf aufsetzenden Moduls zur statistischen Auswertung.
Ein allgemeinerer Ansatz ist das Modelchecking [11], die algorithmisierte Gültigkeitsprüfung einer temporallogischen Formel bezogen auf das Modell eines Systems. Das Modelchecking erlaubt nicht nur die Verifikation einzelner Eigenschaften, sondern die Betrachtung ganzer Klassen von Eigenschaften. Die hier vorgestellte Implementierung eines Modelchecking-Algorithmus (MCA) von [7, 36] ist polynomiell in der Größe des Systems; benötigt allerdings exponentiellen Aufwand in der Größe der Formel. Letzteres ist jedoch nicht so problematisch, da sinnvolle und interessante Formeln zur Beschreibung von Eigenschaften normalerweise klein sind, verglichen mit dem zu untersuchenden System.
Die Modellierung erfolgt mit sicheren T-Systemen, einer relativ eingeschränkten, aber dennoch

mächtigen Netzklasse. Denn anhand von T-Systemen ist die Netzsemantik, die nichtsequentiellen Prozesse eines Petrinetzes, darstellbar. Basierend hierauf gibt es bereits Erweiterungen [18], die größere Netzklassen anhand nichtsequentieller Prozesse prüfen können.

3.2 Programmieren und Analysieren mit Petrinetzen

Die Petrinetzgestützte Entwicklungs- und Programmierumgebung **PEP** bietet Möglichkeiten zur Modellierung, Simulation, Analyse und zur partiellen Verifikation paralleler Systeme.
Zum Zwecke der Veranschaulichung der Funktionalität von **PEP** wird im folgenden ein Beispiel für die Programmentwicklung dargestellt. Darauf aufbauend werden die Einstiegspunkte für den Entwurf allgemeiner paralleler Systeme skizziert.

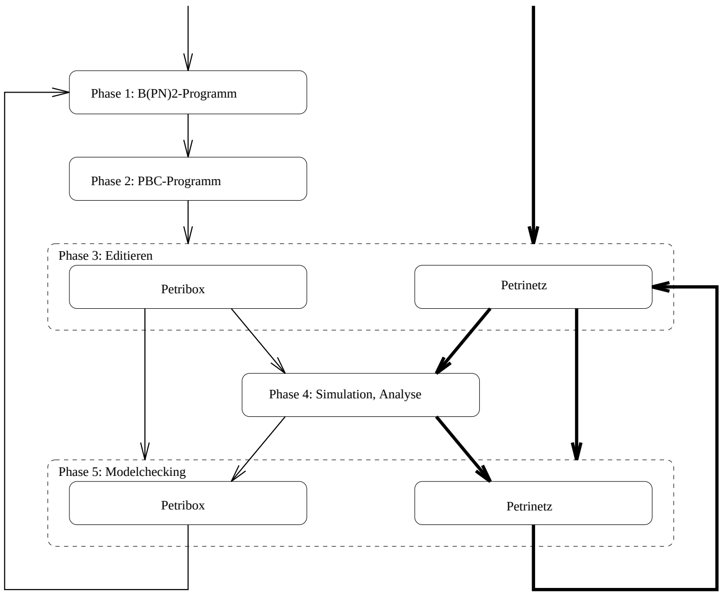


Abbildung 3.2.1 : *Vorgehensmodell für den Entwurf mit* **PEP**

Die Programmentwicklung mit **PEP** gliedert sich in mehrere Phasen, die wiederholt durchlaufen werden können:
Der Benutzer beginnt mit der Eingabe eines parallelen Programms in der Programmiersprache $B(PN)^2$ (Basic Petri Net Programming Notation). Die Syntax von $B(PN)^2$ ist einfach und klar strukturiert. Ein $B(PN)^2$-Programm besteht im wesentlichen aus einem Deklarations- und einem Anweisungsteil. Die im Anweisungsteil erlaubten Operationen umfassen die sequentielle und parallele Komposition, die Iteration und die nichtdeterministische Auswahl.
In der zweiten Phase wird das $B(PN)^2$-Programm automatisch in ein äquivalentes PBC-Programm übersetzt. Gleichzeitig erfolgt eine Überprüfung der syntaktischen Korrektheit des $B(PN)^2$-Programms. PBC-Programme basieren auf der Algebra des Petri Box Calculus [10]; hierbei handelt es sich um eine CCS-ähnliche Prozeßalgebra, die als Grundlage für Phase drei dient.
Für die dritte Phase wird das PBC-Programm auf kompositionelle Art in eine Petribox übersetzt. Petriboxen sind im wesentlichen sichere, beschriftete Petrinetze, die das Verhalten des korrespondierenden $B(PN)^2$-Programms abbilden. Hier kann das Netz mittels eines in **PEP** integrierten Netzeditors weiterbearbeitet werden.

In der vierten Phase kann optional im Netzeditors das Verhalten (des Petrinetzes) graphisch simuliert werden.
Phase fünf enthält das Modelchecking. Hier kann der Benutzer Formeln über dem Netz spezifizieren, die Eigenschaften des Netzes respektive des Programms darstellen. Der Modelchecker überprüft die Gültigkeit einer Formel bezogen auf das markierte Netz. Werden erwünschte Eigenschaften nicht erfüllt, können in Phase eins des Zyklus entsprechende Änderungen am Programm vorgenommen werden.

Für Benutzer, die ihre Systemmodellierung nicht mittels B(PN)^2 durchführen wollen, ist es möglich den Entwurfszyklus direkt in Phase drei mit der Eingabe eines Petrinetzes zu beginnen. Für diesen Fall stellt **PEP** eine Reihe von Analysealgorithmen zur Verfügung. Zum einen können Struktureigenschaften, wie T-System- oder Free-Choice-Eigenschaft, des betrachteten Systems getestet werden; zum anderen sind dynamische Eigenschaften wie Sicherheit, Reversibilität, Lebendigkeit, etc. überprüfbar.
Neben den Anwendungsgebieten Programm- und Systementwurf ist auch gerade hierdurch der Einsatz von **PEP** als Lernsystem für Studenten denkbar; auch ein Einsatz als Frontend und Testbett für weitere Analysealgorithmen ist angedacht.
Als Schwerpunkt dieser Arbeit ist jedoch die Modelchecking-Komponente von **PEP** zu sehen, auf die in den nun folgenden Kapiteln detaillierter eingegangen werden soll. Wobei der hier vorgestellte MCA auf die effiziente Verifikation von Systemen, die direkt editiert wurden (rechter Zyklus der Abbildung 3.2.1), ausgerichtet ist.

3.3 Beschreibung der Logik

Das Modelcheckingproblem (MCP) setzt sich zusammen aus der abstrakten Repräsentation des betrachteten Systems (in dieser Arbeit finden sichere T-Systeme Verwendung) und der Beschreibung zu verifizierender Eigenschaften, die anhand einer geeigneten Logik spezifiziert sind. Bei der hier verwendeten Logik S_4 [23] handelt es sich um eine Aussagenlogik, erweitert um einen Modaloperator $\diamond$, der für die Zwecke des Modelcheckers temporal interpretiert wird. Er bedeutet: "es ist in der Zukunft möglich, daß..". Formal wird die Gültigkeit einer Formel induktiv über den Schaltfolgen des zugrundeliegenden Systems definiert. Die intendierte Semantik der beiden grundlegenden Aussagen kann informell folgendermaßen dargestellt werden:
• Aussagen der Form s werden in Systemen benutzt, die eine Stelle s enthalten. Sie sollen bedeuten, daß nach Ausführung der betrachteten Schaltfolge eine Marke auf s ist.
• Aussagen der Form $t \leq k$ gelten für Systeme, die eine namensgleiche Transition enthalten, und sollen bedeuten, daß in der betrachteten Schaltfolge die Transition t nicht mehr als k-mal schaltet.

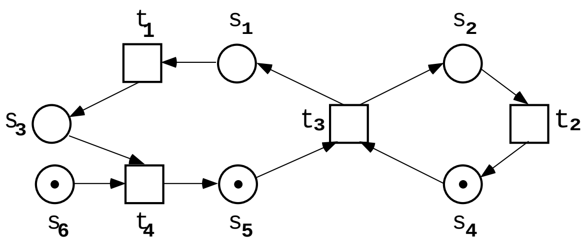


Abbildung 3.3.2 : *Sicheres T-System* Σ

Die Standardform der Formeln der Logik entspricht im wesentlichen der disjunktiven Normalform. Unter Verwendung dieser Logik sind eine Reihe von Sicherheitseigenschaften ausdrückbar; die Formeln beziehen sich auf das T-System in Abbildung 3.3.2.

- *Erreichbarkeit einer Markierung*
  Das System erfüllt $\Diamond(\neg s_1 \wedge s_2 \wedge s_3 \wedge \neg s_4 \wedge \neg s_5 \wedge \neg s_6)$ gdw. die Markierung $M = (0,1,1,0,0,0)$ erreichbar ist.
- *Nebenläufigkeit von Transitionen*
  Das System erfüllt $\Diamond(s_1 \wedge s_2)$ gdw. die Transitionen t_1 und t_2 nebenläufig aktiviert sind.
- *Lebendigkeit von Transitionen*
  Das System erfüllt $\Box\Diamond(s_4 \wedge s_5)$ gdw. die Transition t_3 lebendig ist.

Wie der Leser leicht prüfen kann, sind, bis auf das Beispiel für die Lebendigkeit einer Transition, alle Aussagen für das System Σ aus Abbildung 3.3.2 gültig.

3.4 Implementierung des MCA

Aufgabe des Modelcheckers ist es, die Gültigkeit einer Formel ϕ in einem System Σ zu überprüfen. Die Grundidee des Algorithmus sieht nun vor, die Konstruktion des Transitionssystems zur Prüfung einer Formel zu vermeiden, und stattdessen durch geeignete Manipulation der modallogischen Formel direkt anhand des T-Systems eine Auswertung von ϕ zu erhalten. Dies gelingt durch Reduktion der Berechnung der Modaloperatoren auf die Berechnung gewisser maximaler Schaltvektoren und diese wiederum auf ein lineares Programmierungsproblem. Formeln ohne Modaloperatoren sind direkt anhand der Definitionen in der Anfangsmarkierung auswertbar.

Die aktuelle Version des MCA wurde in C unter Verwendung des SUN C-Compilers auf einer Sun SparcStation 10 unter UNIX SUNOS 5.3 (Solaris 2.3) implementiert und in die XWindow/Motif-Oberfläche von **PEP** integriert. Grundlegend ist das Verständnis der Darstellung der beiden Komponenten des Modelcheckers: das mittels eines T-Systems modellierte System und die modallogische Formel. Die Repräsentation des T-Systems erfolgt anhand einer modifizierten Inzidenzmatrix, welche zusätzlich die Anfangsmarkierung des T-Systems enthält. Die modallogische Formel wird durch einen Strukturbaum in Form eines Binärbaums implementiert.

Der aktuell implementierte MCA-Algorithmus hat folgende Struktur:

Algorithmus 3.1 *Struktur des implementierten MCA*

Eingabe: Ein sicheres T-System Σ und eine Formel ϕ
Ausgabe: Die Formel ϕ ist gültig für das System Σ oder ϕ ist nicht gültig für Σ

begin
  Einlesen der Formel ϕ und umformen in Strukturbaum;
  Einlesen der Netzmatrix des Systems Σ;
  Ersetzen der abgeleiteten Operatoren in ϕ;
  if keine Modaloperatoren **then** Ergebnis anzeigen;
  else
  Menge T_{live} schaltbarer Transitionen bestimmen;
  while kein Ergebnis **do**
    $\phi := \phi$ in Standardform;
    Erzeuge Liste L der Unterformeln in Standardform;
    while L nicht leer $\wedge$ kein Ergebnis **do**
      Simplex ausführen;

Ersetzung von $\Diamond\chi$ durch $\bigwedge_{t\in T} t \leq Last_\chi(t)$ im Strukturbaum;
Auswertung von ϕ anhand des Strukturbaums;
endwhile
endwhile
Ergebnis anzeigen; ■ 3.1
end

Es ist nicht immer notwendig, daß für die Bestimmung des Ergebnisses sämtliche Modaloperatoren entfernt werden, da bekanntlich bei den logischen Operatoren UND und ODER die Kenntnis eines Operanden bereits für die Bewertung des gesamten Ausdrucks ausreichen kann. Basierend auf dieser Heuristik ist es jedoch möglich, das Zeitverhalten des Algorithmus weiter zu verbessern. Die Grundidee der zweiten Heuristik beruht darauf, die Unterformeln in Normalform abhängig vom Abstand ihres Modaloperators zur Wurzel zu entfernen, wobei die Operatoren mit dem kürzesten Abstand zur Wurzel zuerst aufgelöst werden, so daß bereits sehr früh anhand der ersten Heuristik eine vollständige Auswertung vorliegen kann.

3.5 Performanceanalyse des MCA

In diesem Abschnitt soll die Leistungsfähigkeit des Modelcheckers im Bezug auf sein Zeitverhalten für ein ausgewähltes System und typische Formeln untersucht werden.

Zur Erzeugung ausreichend großer Systeme wurde für Testzwecke ein Netzgenerator entworfen, der durch Vervielfältigung einer einzelnen Zelle die Inzidenzmatrix eines größeren T-Systems mit n gleichartigen Zellen generiert. Es können damit regelmäßige lineare und Ringstrukturen erzeugt werden. Dieser Netzgenerator entbehrt jedoch die Funktionalität der graphischen Darstellung der Netze. Für den allgemeinen Entwurf, die Darstellung und das Testen insbesondere nicht regelmäßiger Netzstrukturen werden wir im praktischen Einsatz auf den komfortablen Netzeditor und auf die stetig wachsende Netzbibliothek von **PEP** zurückgreifen.

Im folgenden werden die angeführten theoretischen Ergebnisse anhand der Implementierung diskutiert. Bei dem folgenden System wird dessen Größe bestimmt durch den Faktor der Vervielfältigung der Ausgangsstruktur.

Bei der betrachteten Folge von Systemen handelt es sich um einen parallelen Buffer der Kapazität n und einer Anzahl von k Knoten (#Stellen + #Transitionen). In der Abbildung 3.5.3 ist ein Buffer mit $n = 100$ dargestellt. Der Zustandsgraph hat 2^{100} Knoten.

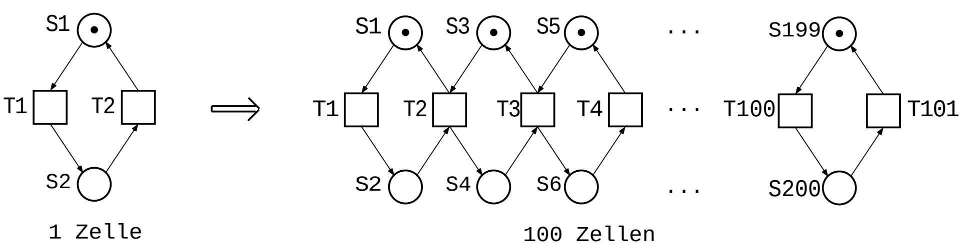


Abbildung 3.5.3 : *Paralleler Buffer mit Kapazität 100*

Die betrachteten Formeln waren:

(i) $\Diamond(S1 \wedge \neg S4 \wedge S42)$ (Erreichbarkeit einer Teilmarkierung)
(ii) $\Box(\neg S5 \vee \neg S6)$ (Gegenseitiger Ausschluß zweier Stellen)
(iii) $\Diamond\Box(S1 \wedge S3)$ (Terminale Teilmarkierung erreichbar)
(iv) $\Box\Diamond(S3 \vee \Box(S1 \wedge S2))$ (Teilmarkierung zyklisch)

Folgende Meßergebnisse wurden anhand des `clock()`-Befehls auf einer Sun SPARCstation 10 bestimmt; die benötigten CPU-Zeiten sind jeweils in Sekunden angegeben:

n	k	(i)	(ii)	(iii)	(iv)
50	151	0.40	0.04	0.70	1.1
100	301	2.7	0.10	5.2	7.6
150	451	8.7	0.25	17	26
200	601	20	0.49	39	59
250	751	39	0.80	76	115
300	901	66	1.2	132	197
350	1051	105	1.7	209	311
400	1201	155	2.3	309	463
450	1351	220	3.1	437	663
500	1501	301	4.0	597	895

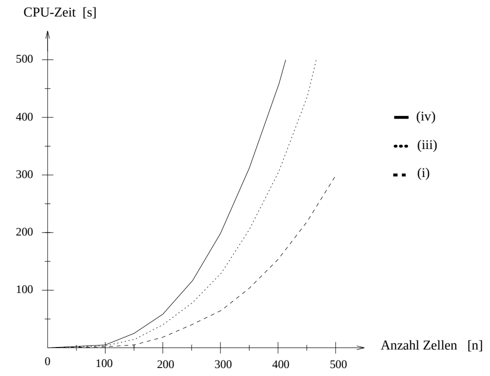


Abbildung 3.5.4 : *Zeitverhalten für den parallelen Buffer*

In [36] werden weitere Ergebnisse für den klassischen Scheduler von Milner [30] präsentiert. Aus diesen Testläufen lassen sich eine Reihe von interessanten Resultaten ableiten. Bei allen Systemen ist zu erkennen, daß die Verdoppelung der Anzahl der Zellen bzw. der Knoten einen Anstieg der Laufzeit um den Faktor sieben bis acht zur Folge hat. Diese Testwerte lassen auf ein ungefähr kubisches, $O(n^{2.8})$, Verhältnis zwischen Anzahl der Zellen und benötigter Laufzeit des Algorithmus schließen.
Bei Formel (i) und (ii) wird einmal der Simplex-Algorithmus aufgerufen, bei Formel (iii) zweimal und bei Formel (iv) dreimal. Bei gleicher Anzahl der Zellen ist im Vergleich zu Formel (i) eine Verdopplung der Laufzeit bei Formel (iii) und eine Verdreifachung bei (iv) zu erkennen. Für Formeln in Standardform deutet dies auf eine lineare Abhängigkeit der Laufzeit von der Tiefe der Formel respektive von der Anzahl aufzulösender $\diamond$-Operatoren hin.
Bei Formel (ii) zeigen sich deutlich bessere Ergebnisse bezüglich der Laufzeit; sogar bei 2000 Zellen werden lediglich 22 Sekunden benötigt. Hier gelangt das Simplex-Verfahren in wenigen Schritten zu einem Resultat, da früh erkannt wird, daß das LGS keine Lösung besitzt. Im allgemeinen kann man dieses Verhalten allerdings nicht voraussetzen, wie auch die übrigen Ergebnisse deutlich zeigen. Weiter ist zu beobachten, daß der verwendete Simplex-Algorithmus [32] im betrachteten Bereich bis zu 1500 Netzknoten die Hauptdeterminante des Zeitbedarfs ist.
Die praktischen Ergebnisse untermauern die theoretische Erkenntnis des polynomiellen Wachstums der Laufzeit in der Größe des Systems. Dies ist ein befriedigendes Ergebnis, insbesondere wenn man die zu erwartetende Laufzeit bei einer direkten Auswertung der Formel betrachtet, was unter Umständen die Bestimmung aller erreichbaren Markierungen des Systems erfordert. Es zeigt sich zudem, daß Werkzeuge wie PROD [37] für diesen Systemtyp nur bis zu einer Größe von 20 Zellen in annehmbarer Zeit Resultate liefern.

3.6 Abschliessende Betrachtungen und Ausblick

Es wurde die Implementierung eines effizienten Modelcheckers vorgestellt, der auch für Systeme mit einer hohen Anzahl erreichbarer Markierungen noch akzeptable Ergebnisse liefert. Die Testergebnisse untermauern die Polynomialität des MCA in der Größe des Systems. Der exponentielle Anteil, bedingt durch die Umformung der Formeln in eine Standardform, war bei den Testläufen unerheblich. Denn zum einen ist die Länge sinnvoller Formeln (verglichen mit der

Größe der zu untersuchenden Systeme) gering, und zum anderen sind viele der zu untersuchenden Formeln bereits in Standardform.
Deutlich wurde die starke Abhängigkeit der Laufzeit des MCA von der Berechnung des linearen Gleichungssystems. Für den praktischen Einsatz empfiehlt sich die Verwendung von für große, schwach besetzte Matrizen geeigneteren Algorithmen zur Lösung des LP-Problem. Es lassen sich damit stark verbesserte Ergebnisse erwarten.
Auch die Beschränkung auf sichere T-Systeme ist bereits überwunden worden. Es bestehen Erweiterungen des Modelcheckers, die auf sicheren Systemen operieren. Das heißt, für allgemeine 1-sichere Systeme gibt [18] einen Algorithmus an, der die Konstruktion des Erreichbarkeitsgraphen vermeidet und statt dessen ein Objekt konstruiert, dessen Größe sehr oft quadratisch oder kubisch in der Größe des Systems ist. Die Zeit-Polynomialität geht aber i. a. verloren.
Für die Zukunft sind eine Reihe von Erweiterungen von **PEP** angedacht bzw. z. T. bereits realisiert. Es sollen an dieser Stelle nur die für das Modelchecking direkt relevanten genannt werden: Die Integration eines High-Level-Netzmodells, die Entwicklung einer adäquaten Logik auf $B(PN)^2$-Programmen, die Integration eines Expertensystems zur Netzanalyse. Ein Schwerpunkt soll der Bereich des High-Level-Modelchecking werden. Hierzu müssen geeignete Teilklassen von High-Level-Netzen identifiziert und entsprechende Algorithmen entwickelt werden.

4 Erläuterungen zu Esparza's L1-Model-Checker

Burkhard Graves
Universität Hildesheim
Institut für Informatik
Marienburger Platz 22
D - 31141 Hildesheim
e-mail: graves@informatik.uni-hildesheim.de

Abstract

Javier Esparza presented in his Habilitation Thesis "A Partial Order Approach to Model Checking" (1993) a new approach dealing with the model checking problem which makes fundamental use of the theory of net unfoldings.

Most of the work done in the field of model checking concentrates on interleaving semantics, which are well suited for an easy design of elegant modelcheckers. One of the main disadvantages of interleaving model checkers is closely related to the state explosion problem. In special cases partial order semantics provides an interesting alternative to deal with this problem, because of the usually smaller partial orders (compared to the set of interleavings) of a system. On the other hand the development of appropriate verification algorithms is much more sophisticated.

In this talk an attempt will be made to explain the fundamental aspects of Esparza's approach. By giving various examples the focus of interest will be to explain the way the model checker implemented in the **PEP** system works.

4.1 Einführung

Unter Model-Checking versteht man den Vorgang des Überprüfens, ob eine Formel ϕ in einem Modell Σ gültig ist ($\Sigma \models \phi$, gelesen: Σ erfüllt ϕ) oder nicht; es handelt sich also um die Lösung eines Entscheidungsproblems.

Wenn man nun Eigenschaften eines nichtsequentiellen Systems als Aussagen in einer temporalen Logik formalisiert, und dem betrachteten System über eine formale Semantik ein mathematisches Objekt zuordnet, auf dem die Logik interpretiert werden kann, dann ist Model-Checking gleichbedeutend mit der Verifikation der Eigenschaften des nichtsequentiellen Systems. Zum Beispiel erfüllt ein paralleles Programm seine Spezifikation genau dann, wenn die Semantik des Programms ein Modell der entsprechend formalisierten Spezifikation ist.

Petrinetze sind eine seit langem bekannte und anerkannte Möglichkeit zur Modellierung nichtsequentieller Systeme. Prinzipiell läßt sich das Verhalten eines markierten Petrinetzes auf zwei verschiedene Arten beschreiben: durch eine Interleaving-Semantik und durch eine Halbordnungs-Semantik. Interleaving-Semantiken besitzen unkomplizierte mathematische Eigenschaften und ermöglichen dadurch einfache und elegante Model-Checker. Ein Nachteil von Interleaving-Semantiken ist jedoch, daß die Größe des zu konstruierenden Zustandraumes schon bei kleinen Systemen, die jedoch viel Parallelität enthalten, explosionsartig anwachsen kann. Mit diesem sogenannten State-Explosion-Problem haben Halbordnungs-Semantiken nicht zu kämpfen, da bei ihnen der Zustandsraum nicht explizit berechnet wird. Vielmehr beschreiben die üblichen

Halbordnungs-Semantiken das Verhalten eines Systems durch die Menge seiner Prozesse oder durch die Angabe seines maximalen Branching-Prozesses.

Aus der Theorie der Halbordnungs-Semantiken von Petrinetzen hat Javier Esparza in seiner Habilitationsschrift [18] einen neuen, sehr effizienten Model-Checking-Algorithmus für 1-sichere Systeme hergeleitet.

Natürlich kann und soll hier nicht auf wenigen Seiten diese großartige Arbeit wiedergegeben werden; auch eine Präsentation des eigentlichen Algorithmus wäre ohne die theoretischen Grundlagen kaum sinnvoll. Statt dessen soll versucht werden, ein intuitives Verständnis der grundlegenden Ideen dieses neuen Model-Checking-Ansatzes zu vermitteln.

Kurz noch zu einigen Vereinbarungen und Abkürzungen, die in den nachfolgenden Kapiteln benutzt werden:

- Wann immer von einem Beispiel die Rede ist, bezieht sich dieses auf Abbildung 4.2.1 auf Seite 37.
- Wann immer J. Esparza zitiert wird, bezieht sich dies auf die Habilitationsschrift [18].
- Nichtsequentielle Systeme werden durch 1-sichere, endliche Petrinetz-Systeme $\Sigma = (S, T, F, M_0)$ modelliert, S bezeichnet die Stellenmenge, T die Transitionenmenge, F die Flußrelation und M_0 die Anfangsmarkierung.
- Die Abkürzung **MC** steht für Model-Checker oder Model-Checking, die Abkürzung **EMC** für Esparza's MC.
- Die Abkürzung **HO** steht für Halbordnung.

4.2 Einige Grundlagen

Die grundlegenden Begriffe aus dem Bereich der Petrinetz-Theorie werden als bekannt vorausgesetzt, bzw. können bei Esparza nachgelesen werden. Andere, nicht ganz so geläufige, wichtige Begriffe sollen im folgenden erläutert werden; allerdings wird dabei soweit wie möglich auf formale Definitionen verzichtet, statt dessen werden viele Begriffe und Vorgänge anhand von Beispielen erläutert.

Es gibt verschiedene HO-Semantiken für Petrinetz-Systeme. Die wohl bekannteste Semantik ordnet einem System die Menge seiner möglichen Prozesse zu. Ein Prozeß ist ein Kausal-Netz zusammen mit einer speziellen Beschriftungsfunktion, die die Verbindung zu dem System herstellt. Ein derartiger Prozeß beschreibt ein mögliches Verhalten des Systems, die Menge aller Prozesse somit das Gesamtverhalten des Systems.

Eine andere (hier benutzte) Semantik ordnet einem System seinen maximalen Branching-Prozeß zu. Ein Branching-Prozeß (beliebig, d.h. nicht unbedingt maximal) ist ein sogenanntes Occurrence-Netz zusammen mit einer speziellen Beschriftungsfunktion, die wiederum die Verbindung zu dem System herstellt. Die Bezeichnung Branching-Prozeß erklärt sich folgendermaßen: während die zuerst erwähnte Semantik durch die Erzeugung von verschiedenen Prozessen Konflikte auflöst, bleiben die Konflikte im Branching-Prozeß erhalten, allerdings verzweigt sich der Branching-Prozeß an jeder Konfliktstelle in die jeweils möglichen Folgeverhalten; der maximale Branching-Prozeß stellt gewissermaßen eine Vereinigung aller möglichen Prozesse dar.

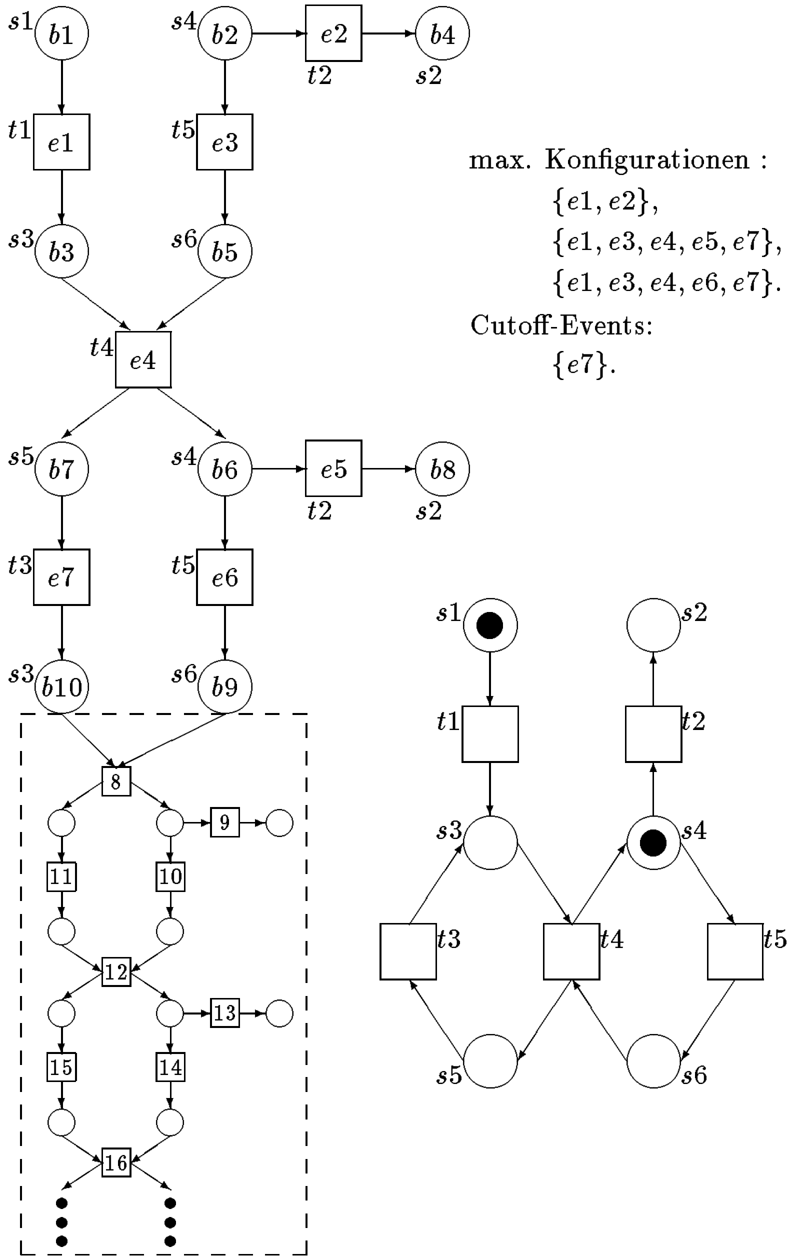


Abbildung 4.2.1 : *Ein System (rechts) und sein Branching-Prozeß*

Tatsächlich bildet die Menge aller Branching-Prozesse bezüglich einer sogenannten Approximierungsrelation einen vollständigen Verband mit dem maximalen Branching-Prozeß als Top- und dem minimalen Branching-Prozeß, welcher nur aus Bedingungen besteht, die der Anfangsmarkierung des Systems entsprechen, als Bottom-Element.

Im Beispiel ist rechts unten ein 1-sicheres System abgebildet, welches ein unendliches Verhalten aufweisen kann. Insofern ist der links davon dargestellte maximale Branching-Prozeß unendlich; jedoch ist nur ein Anfangsstück ausführlich, d.h. vollständig beschriftet dargestellt, der unendlich große Rest befindet sich in dem gestrichelten Kasten (über die Angabe der max. Konfigurationen und des Cutoff-Events möge der Leser im Moment noch hinwegsehen). Anhand dieses Beispiels sollen nun einige wichtige Begriffe erläutert werden, die auch bei der Definition der Logik im nächsten Kapitel eine wichtige Rolle spielen.

Das zugrundeliegende Occurrence-Netz eines Branching-Prozesses ist immer azyklisch und kann insofern als Darstellung einer Halbordnung aufgefasst werden. Im Beispiel bilden $b1$ und $b2$ das Minimum dieser Halbordnung, und die Beschriftung dieser beiden Bedingungen entspricht der Anfangsmarkierung des Systems. Nun stelle man sich vor, auf $b1$ und $b2$ läge jeweils eine Marke; jede mögliche Schaltfolge im Occurrence-Netz entspräche dann bezüglich der Beschriftung der Ereignisse einer entsprechenden Schaltfolge im System, jede erreichbare Markierung im Occurrence-Netz bezüglich der Beschriftung der Bedingungen einer entsprechenden Markierung des Systems. Eine derartige Ereignismenge nennt man Konfiguration (C) und die durch sie definierte Menge von Bedingungen einen Schnitt ($Cut(C)$). $Mark(C)$ sei die Beschriftung von $Cut(C)$. Für jede erreichbare Markierung M des Systems existiert eine endliche Konfiguration C des maximalen Branching-Prozesses, so daß $Mark(C) = M$ ist.

Einige Beispiele: $\{e4\}$ ist keine Konfiguration, da diese Menge nicht kausal abgeschlossen ist, d.h. es fehlen $e1$ und $e3$. $\{e2, e3\}$ ist ebenfalls keine Konfiguration, da die beiden Elemente in Konflikt stehen. $C = \{e1, e3, e4, e5\}$ ist hingegen eine Konfiguration, $Cut(C) = \{b7, b8\}$ und $Mark(C) = \{s2, s5\}$.

4.3 Eine modale Logik für 1-sichere Systeme

Die folgende Grammatik definiert die Syntax der $L1$ genannten branching-time-Logik zu einem gegebenen System Σ:

$$\phi ::= \mathbf{true} \mid s \mid \neg\phi \mid \phi \wedge \phi \mid \diamond\phi \qquad (s \in S).$$

Natürlich lassen sich andere Operatoren aus den gegebenen ableiten, z.B. $\Box\phi = \neg \diamond \neg\phi$.

Die Semantik von $L1$ ergibt sich folgendermaßen: Sei C eine endliche Konfiguration eines beliebigen Branching-Prozesses β von Σ und ϕ eine Formel über Σ.

Die Aussage $(C, \beta) \models \phi$ wird induktiv definiert:

$$\begin{aligned}
(\beta, C) \models s &\quad \text{gdw.} \quad s \in Mark(C) \\
(\beta, C) \models \neg\phi &\quad \text{gdw.} \quad \neg((\beta, C) \models \phi) \\
(\beta, C) \models \phi_1 \wedge \phi_2 &\quad \text{gdw.} \quad ((\beta, C) \models \phi_1) \wedge ((\beta, C) \models \phi_2) \\
(\beta, C) \models \diamond\phi &\quad \text{gdw.} \quad \exists C' \supseteq C : (\beta, C') \models \phi
\end{aligned}$$

Anhand des maximalen Branching-Prozesses β_m von Σ kann nun auch die Gültigkeit einer Formel bezüglich des Systems definiert werden:

$$\Sigma \models \phi \quad \text{gdw.} \quad (\beta_m, \emptyset) \models \phi$$

Im folgenden wird $(\beta_m, C) \models \phi$ zu $C \models \phi$ abgekürzt. $C \models \diamond\phi$ gilt genau dann, wenn C zu einer Konfiguration C' verlängert werden kann, die ϕ erfüllt. $C \models \diamond\phi$ ist also gleichbedeutend mit der Aussage, daß es einen Folgezustand gibt, der ϕ erfüllt. Es wird allerdings nichts darüber ausgesagt, ob dieser Folgezustand überhaupt jemals eintritt. Die Logik $L1$ kann somit Safety-Eigenschaften ausdrücken, wie z.B.

- Erreichbarkeit von Markierungen,
- Nebenläufigkeit von Transitionen und
- Lebendigkeit von Transitionen ($\Sigma \models \Box \diamond s4$ gdw. die Transition $t5$ lebendig ist (was sie jedoch nicht ist)).

Nicht ausdrückbar sind allgemeine Liveness-Eigenschaften, so z.B. die Aussage, daß ein Zustand auf jeden Fall erreicht wird, also nicht nur bloß erreichbar ist. Es soll jedoch ausdrücklich festgehalten werden, daß die Ausdruckskraft von $L1$ mit der von Interleaving-Logiken übereinstimmt, die über erreichbare Markierungen, also den Erreichbarkeitsgraphen interpretiert werden.

Wieder einige Beispiele:

$\{e1, e2\} \models s2$, denn $Mark(\{e1, e2\}) = \{s2, s3\}$.

$\Sigma \models s4$, denn $Mark(\emptyset) = \{s1, s4\}$.

$\{e1, e2\} \models \neg \diamond s5$, denn $s5 \notin Mark(\{e1, e2\})$ und $\{e1, e2\}$ kann auch nicht entsprechend verlängert werden.

$\Sigma \models \diamond s5$, denn $\{e1, e3, e4\} \supseteq \emptyset$ und $\{e1, e3, e4\} \models s5$.

4.4 Der endliche Präfix

Logiken von MC, die auf Interleaving-Semantiken beruhen, basieren auf dem Erreichbarkeitsgraphen, welcher für endliche, 1-sichere Systeme natürlich endlich ist. Insofern kann ein Interleaving-MC die Gültigkeit von Formel auch problemlos auf diesem Graphen überprüfen – wenn auch nicht sehr effizient.

Der maximale Branching-Prozeß derartiger Systeme, auf dem die im vorigen Kapitel definierte Logik basiert, ist jedoch nur dann endlich, wenn das System kein unendliches Verhalten aufweisen kann, d.h. zwangsläufig in einen Deadlock hineinläuft. Insofern ist der maximale Branching-Prozeß zur Verifikation einer Formel ungeeignet.

Statt dessen arbeitet EMC auf einem endlichen Anfangsstück des maximalen Branching-Prozesses. Dieser sogenannte endliche Präfix muß jedoch genügend Informationen enthalten, um entscheiden zu können, ob eine Formel gültig ist oder nicht. Da in der $L1$-Logik die Erreichbarkeit von Markierungen ausdrückbar ist, muß jede erreichbare Markierung M des Systems im endl. Präfix repräsentiert sein, d.h. es muß eine Konfiguration C des endl. Präfix geben, so daß $Mark(C) = M$ ist. Das ein derartiger endl. Präfix überhaupt existiert, ist relativ leicht einzusehen: Die Menge der erreichbaren Markierungen eines endlichen, 1-sicheren Systems ist endlich, und jede erreichbare Markierung wird durch (mind.) einen Schnitt im maximalen Branching-Prozess repräsentiert.

Die sogenannte Entfaltung nach McMillan ermöglicht es, den gewünschten endlichen Präfix zu berechnen; auf die Arbeitsweise dieses Algorithmus soll hier nicht näher eingegangen werden.

Im Beispiel besteht der endliche Präfix des maximalen Branching-Prozesses des Systems aus dem oberen, ausführlich beschrifteten Teil (also ohne den Inhalt der gestrichelten Box). Die maximalen Konfigurationen sind die bezüglich der Teilmengenrelation maximalen Elemente der Menge aller Konfigurationen des endlichen Präfix. Bleibt noch zu erklären, was ein Cutoff-Event ist. Jedem Ereignis $e \in E$ läßt sich eindeutig eine Konfiguration zuordnen und zwar besteht diese aus dem Ereignis selbst und allen Ereignissen, die stattfinden müssen, damit auch das betrachtete Ereignis stattfinden kann.

Zum Beispiel gehört zum Ereignis $e5$ die durch $[e5]$ bezeichnete Konfiguration

$\{e1, e3, e4, e5\}$, hingegen ist $[e2]=\{e2\}$.

Ein Ereignis $e \in E$ ist genau dann ein Cutoff-Event, wenn ein korrespondierendes Ereignis $e' \in E$ existiert, so daß

$$|[e']| < |[e]| \quad \text{und} \quad Mark([e']) = Mark([e]).$$

Im Beispiel ist $e7$ das einzige Cutoff-Event, $e1$ ist das korrespondierende Ereignis.

Konfigurationen des max. Branching-Prozesses, die über den endlichen Präfix hinausgehen, können aus Konfigurationen des endlichen Präfix generiert werden, und zwar durch ein sogenanntes Shiften bezüglich eines seiner Cutoff-Events (anschaulich: bezüglich des entsprechenden Schnittes). Es wird hier keine genaue Definition der $Shift$-Funtion gegeben, wichtig ist das intuitive Verständnis, darum ein Beispiel:

Die Konfiguration $\{e1, e3, e4, e6, e7, e8\}$ gehört nicht zum endlichen Präfix, sie kann jedoch durch einmaliges Shiften der Konfiguration $\{e1, e3, e4\}$ über den $Cut([e7]) = \{b6, b10\}$ wiedergewonnen werden. $e1$ liegt vor dem Schnitt des korrespondierenden Ereignisses von $e7$ ($Cut([e1]) = \{b3, b2\}$) [3] und bleibt somit unverändert. $e3$ wird auf $e6$ abgebildet und $e4$ auf $e8$. Hinzugefügt wird $[e7]$, so daß sich letztendlich die gesuchte Konfiguration ergibt.

In Kurzschreibweise noch ein weiteres Beispiel:

$\{1, 3, 4, 6, 7, 8, 10, 11, 12, 13, 15\}$ wird durch dreimaliges Shiften, beginnend bei der Konfiguration $\{1, 2\}$ gewonnen:

$2 \rightarrow 5, +1, 3, 4, 7;$
$3 \rightarrow 6, 4 \rightarrow 8, 5 \rightarrow 9, 7 \rightarrow 11, +1, 3, 4, 7$
$3 \rightarrow 6, 4 \rightarrow 8, 6 \rightarrow 10, 7 \rightarrow 11, 8 \rightarrow 12, 9 \rightarrow 13, 11 \rightarrow 15, +1, 3, 4, 7.$

Die $Shift$-Funktion hat hat u.a. folgende Eigenschaften:

$$Mark(C) = Mark(Shift(C)) \qquad \text{und} \qquad C \models \phi \quad \text{gdw.} \quad Shift(C) \models \phi.$$

Sei nun $Sat(\phi)$ die Menge aller Konfiguration des maximalen Branching-Prozess, die ϕ erfüllen und $Sat_f(\phi) = Sat(\phi) \cap \Omega$, wobei Ω die Menge aller Konfigurationen des endlichen Präfix bezeichne. Jede Konfiguration in $Sat(\phi)$ kann durch eine Konfiguration aus $Sat_f(\phi)$ durch genügend häufiges Shiften gewonnen werden. Insofern sind diese beiden Mengen äquivalent. Natürlich gilt auch

$$Sat(\phi) = \emptyset \quad \text{gdw.} \quad Sat_f(\phi) = \emptyset.$$

Wenn $Sat(\phi) = \emptyset$ ist, bedeutet dies, daß keine Konfiguration (auch nicht die leere) ϕ erfüllt, und dies ist gleichbedeutend mit $\neg(\Sigma \models \phi)$. Das MC-Problem reduziert sich also darauf, zu überprüfen, ob $Sat_f(\phi)$ leer ist.

Um einen ähnlichen Negativ-Effekt wie das State-Explosion-Problem beim Interleaving-MC zu vermeiden, werden jedoch nicht alle Konfigurationen des endlichen Präfix berechnet, die ϕ erfüllen, sondern nur die maximalen davon; diese Menge wird durch $Last(\phi)$ bezeichnet, d.h. $Last(\phi) = max(Sat_f(\phi))$. Wie diese Berechnung in etwa erfolgt, wird im nächsten Kapitel erläutert.

[3] Die Schnitte $Cut([e7])$ und $Cut([e1])$ eventuell in der Abb. farbig nachtragen

4.5 Der Model-Checker

Jede Formel kann in eine Disjunktion von Formeln in Normalform überführt werden. Dies liefert eine erste Zerlegung des MC-Problems, da der Vorgang des Überprüfens dadurch für jedes Disjunktionsglied getrennt erfolgen kann. Eine Formel ist in Normalform, wenn sie von der folgenden Grammatik generiert werden kann:

$$\begin{array}{lcl} \gamma & ::= & \textbf{true} \mid \textbf{false} \mid s \mid \neg s \mid \gamma \wedge \gamma \\ \phi & ::= & \gamma \mid \phi \wedge \diamond\phi \mid \phi \wedge \neg \diamond \phi \end{array}$$

Der Zeit- und Platzaufwand bei der Umformung einer beliebigen Formel in eine Disjunktion von Formeln in Normalform kann jedoch im ungünstigsten Fall exponentiell von der Länge der zugrunde liegenden Formel abhängen. Wie Esparza gezeigt hat, können aber sowieso keine polynomiell zeitbeschränkten Model-Checking-Algorithmen existieren, es sei denn $P = NP$. Insofern ist dieser Effekt, d.h. die Verschiebung der Komplexität von einem Eingabeparameter (Modell) zum anderen (Formel) erwünscht, da die Netze sehr schnell sehr groß werden können, während die Formalisierungen der meisten interessierenden Eigenschaften vergleichsweise klein bleiben.

Die Gültigkeit von Formeln, die aus γ ableitbar sind, die also keine $\diamond$-Operatoren enthalten, kann einfach durch Untersuchung der Anfangsmarkierung des Systems (bzw. der Beschriftung des Minimums des Occurrence-Netzes) festgestellt werden.

Zum Beispiel gilt $\Sigma \models s1 \wedge \neg s2$, da $M_0(s1) = 1$ und $M_0(s2) = 0$ ist.

Also reduziert sich das MC-Problem auf Formeln der Art $\diamond\phi$, genauer (siehe Normalform):

1. $\diamond\gamma$
2. $\diamond(\phi_1 \wedge \diamond\phi_2)$
3. $\diamond(\phi_1 \wedge \neg \diamond \phi_2)$

Esparza gibt zwei Algorithmen an, um Formeln der ersten Art zu überprüfen. Der erste basiert auf linearer Programmierung, der zweite arbeitet direkt auf dem endlichen Präfix und soll hier kurz anhand eines Beispiels erläutert werden:

Angenommen, es soll überprüft werden, ob $\Sigma \models \diamond(s6 \wedge \neg s3)$ gilt, also ob ein Zustand erreichbar ist, in dem $s6$ markiert und $s3$ unmarkiert ist. Dazu berechnet man ausgehend von den maximalen Konfigurationen des endlichen Präfix alle maximalen Konfigurationen, die $(s6 \wedge \neg s3)$ erfüllen. Existieren derartige Konfigurationen, so gilt $\Sigma \models \diamond(s6 \wedge \neg s3)$. Ausgehend von $C = \{e1, e3, e4, e5, e7\}$, einer maximalen Konfiguration des endlichen Präfix, wird diese Suche einmal beispielhaft beschrieben:

$Mark(C) = \{s2, s3\}$, es ist also weder $s6$ markiert, noch $s3$ unmarkiert. In so einem Fall versucht man sinnvollerweise, zuerst eine Konfiguration zu finden, die einer Markierung entspricht, die $s6$ enthält. Dazu sucht man ein maximales (bezüglich der F^*-Relation) Ereignis in C, daß eine mit $s6$ beschriftete Bedingung im Vorbereich hat. Gibt es ein derartiges Ereignis nicht, so ist C nicht Obermenge einer erfüllenden Konfiguration. In unserem Beispiel gibt es jedoch ein derartiges Ereignis, nämlich $e4$, welches die mit $s6$ beschriftete Bedingung $b5$ im Vorbereich hat. Alle Ereignisse, die dieser Bedingung kausal folgen, werden jetzt aus C entfernt, d.h. $e4$, $e5$ und $e7$, also $C = \{e1, e3\}$. $Mark(C) = \{s3, s6\}$, $s3$ ist also markiert. Um dies zu ändern, entfernt man das Ereignis im Vorbereich der mit $s3$ beschrifteten Bedingung aus C. Hat diese Bedingung gar kein Ereignis im Vorbereich, so ist C wiederum nicht Obermenge einer erfüllenden Konfiguration. In unserem Beispiel kann jedoch $e1$ entfernt werden, wonach C nur noch aus $e3$ besteht. $Mark(C) = \{s1, s6\}$, letztendlich gilt also $\Sigma \models \diamond(s6 \wedge \neg s3)$.

Die Überprüfung von Formeln der zweiten und dritten Art erfolgt streng dekompositionell. Das Konjunktionsglied mit dem temporalen Operator wird abgespalten und der mögliche Lösungsraum für das verbleibende Konjunktionsglied wird in Abhängigkeit von der Erfüllbarkeit (genauer: von den Konfigurationen, die den abgespaltenen Operanden erfüllen) u.U. verkleinert.

Wenn also die Teilformel $\diamond\phi_2$ nicht erfüllbar ist, dann wird der Lösungsraum für das verbleibende Konjunktionsglied in $\diamond(\phi_1 \wedge \diamond\phi_2)$ auf die Dimension 0 verkleinert, denn wenn kein Zustand erreichbar ist, in dem ϕ_2 gilt, dann ist natürlich auch kein Zustand erreichbar, in dem ϕ_1 gilt und der einen Folgezustand hat, der ϕ_2 erfüllt. Der Lösungsraum für das verbleibende Konjunktionsglied in $\diamond(\phi_1 \wedge \neg \diamond \phi_2)$ wird jedoch nicht verkleinert.

Wenn die Teilformel $\diamond\phi_2$ aber erfüllbar ist, dann kann eine Konfiguration, die $\diamond\phi_1$ erfüllt, nur dann auch $\diamond(\phi_1 \wedge \diamond\phi_2)$ erfüllen, wenn sie Teilmenge einer $\diamond\phi_2$ erfüllenden Konfiguration ist. Analog gilt, daß eine Konfiguration, die $\diamond\phi_1$ erfüllt, nur dann auch $\diamond(\phi_1 \wedge \neg \diamond \phi_2)$ erfüllen kann, wenn sie nicht Teilmenge einer $\diamond\phi_2$ erfüllenden Konfiguration ist.

Diese Idee soll wieder anhand eines sehr einfachen Beispiels verdeutlicht werden:

Sei $\phi = \diamond(s6 \wedge \neg \diamond s5)$ und es soll überprüft werden, ob $\Sigma \models \phi$ gilt. Der vorhin beispielhaft erläuterte Algorithmus liefert die folgenden Menge von maximalen Konfigurationen des endlichen Präfix, die $\diamond s5$ erfüllen:

$L_1 = \{\{e1, e3, e4, e6\}, \{e1, e3, e4, e5\}\}$.

Für $\diamond s6$ erhält man die Menge

$L_2 = \{\{e1, e3, e4, e6, e7\}\}$.

$\{e1, e3, e4, e6, e7\}$ ist nicht Teilmenge von einer der Mengen in L_1. Daraus zu schließen, daß $\{e1, e3, e4, e6, e7\}$ eine Konfiguration ist, die ϕ erfüllt und somit auch $\Sigma \models \phi$ gilt, wäre verfrüht und falsch. $\{e1, e3, e4, e6, e7\}$ kann im maximalen Branching-Prozess nämlich zu $\{e1, e3, e4, e6, e7, e8\}$ verlängert werden, und die zu dieser Konfiguration gehörende Markierung enthält sehr wohl $s5$.

Also muß die Menge L_1 noch ein wenig nachbehandelt werden, so daß sie tatsächlich $Last(\diamond s5)$ entspricht. Die Nachbehandlung besteht darin, die Elemente von L_1 bezüglich aller Cutoff-Events (hier also nur $e7$) zu shiften – allerdings begrenzt auf dem endlichen Präfix – und das jeweilige Ergebnis zu L_1 hinzuzufügen. Diese Nachbehandlung wird solange wiederholt, bis L_1 nicht mehr vergrößert werden kann. Dieser Zustand tritt irgendwann ein, denn der endliche Präfix enthält nur endlich viele Konfigurationen. Letztendlich setzt man noch $L_1 := max(L_1)$.

Konkret: Shiften von $\{e1, e3, e4, e6\}$ liefert $\{e1, e3, e4, e6, e7, e8, e10\}$, $e8$ und $e10$ gehören jedoch nicht zum endlichen Präfix, es wird also $\{e1, e3, e4, e6, e7\}$ zu L_1 hinzugefügt. Begrenztes Shiften von $\{e1, e3, e4, e5\}$ oder dem neuen Element bringt keine weiteren neuen Elemente, also kann das Maximum gebildet werden, und man erhält

$L_1 = \{\{e1, e3, e4, e6, e7\}, \{e1, e3, e4, e5\}\} = Last(\diamond s5)$ [4]. Jetzt erkennt man auch, daß der einzige Lösungskandidat für $\diamond s6$ eine Teilmenge eines Elements aus L_1 ist, und somit das System die Formel nicht erfüllt.

[4] $\{e1, e3, e4, e6, e7\}$ ist ein Anfangsstück von unendlich vielen Konfigurationen des maximalen Branching-Prozesses, die $\diamond s5$ erfüllen

4.6 Over and out

Esparza hat das Laufzeitverhalten für verschiedene Netzklassen ausführlich untersucht, mit dem Ergebnis, daß sein MC für bestimmte Netzklassen eine polynomielle Laufzeit erzielt (in Abhängigkeit von der Netzgröße), während alle bis zu diesem Zeitpunkt bekannten MC für die entsprechenden Netzklassen ein exponentielles Laufzeitverhalten aufweisen.

Auf diese Untersuchungen und auf die ebenfalls von Esparza vorgeschlagene Erweiterung der Logik zu einer Linear-Time-Logik wurde hier nicht eingegangen. Überhaupt sollte diese kleine Ausarbeitung nur die grundlegenden Ideen des EMC vermitteln; wer sich ernsthaft mit der Funktionsweise des EMC auseinandersetzen will, sollte zu Esparza's Habilitationsschrift greifen. Wenn dieser Text einen eventuellen Einstieg erleichtern kann, so hat er seinen Zweck erfüllt.

5 Verweise in PEP und mögliche Anwendungen

Matthias Damm
Universität Oldenburg
Fachbereich Informatik
D - 26111 Oldenburg
e-mail: matthias.damm@informatik.uni-oldenburg.de

Abstract

Using the **PEP** system it is possible to translate a program written in the concurrent programming notation $B(PN)^2$ into a box expression of the Petri Box Calculus (PBC). Each box expression can be translated into a Petri box, a special class of safe Petri nets. The token game of the resulting Petri box can be simulated in the Petri box editor.

Each atomic command used in a $B(PN)^2$ program is represented by a single transition, each possible value of a variable by a single place in the resulting Petri box. In many cases (e.g. while learning how to use $B(PN)^2$ or while checking properties of a program by using the built-in Petri net simulator) it can be very useful to know which atomic command corresponds to which transition or which place belongs to which variable's value. Therefore, **PEP**creates a system of references between the $B(PN)^2$ program, the box expression and the Petri box while translating one into the other.

The application of references is very simple: After selecting one or more vertices in the Petri box and calling the "reference" menu item, the corresponding part of the program will be highlighted. The other way round, one simply selects a region of the program text and in return gets the corresponding vertices.

Another possible (but not yet implemented) application is the simulation of a concurrent program based on the simulation of the Petri box. An enabled transition belonging to an atomic command denotes that the program is in a state where this command might be executed. The firing of the transition represents the execution of this command. The declared variables' bindings can be seen in the marking of the places in the data or channel box.

5.1 Einleitung

Petriboxen sind eine spezielle Klasse von Petrinetzen. Durch die Kommunikations-, Eingangs- und Ausgangsschnittstellen können auf kompositionellem Wege aus einfachen Petriboxen komplexere Boxen gebildet werden.

Der Petribox-Kalkül PBC [6] bietet eine Termsprache, die es ermöglicht, Petriboxen durch syntaktische Konstrukte zu generieren. In der PBC-Sprache werden elementare Aktionen mittels Operatoren zu einem Term verknüpft. Solch ein Term wird in eine Petribox übersetzt, indem jede elementare Aktion in die zugehörige Basisbox übersetzt wird und diese gemäß den Kompositionsregeln der Operatoren zu einer komplexeren Box zusammengefügt werden.

Terme des Petribox-Kalküls bilden die Semantik für Programme der parallelen Programmiersprache $B(PN)^2$ (Basic Petri Net Programming Notation [10]). Bei der Übersetzung eines

$B(PN)^2$-Programms in einen Term werden Variablendeklarationen und atomare Anweisungen in passende Konstrukte der Termsprache übertragen.

Das **PEP**-System ermöglicht die Erstellung von $B(PN)^2$-Programmen, die Übersetzung in Terme und die anschliessende Übersetzung in Petriboxen. Diese Netze können mittels eines effizienten Modelcheckers auf bestimmte Eigenschaften hin untersucht werden. Bei den Übersetzungsschritten wird ein System von Verweisen zwischen den drei Ebenen *Programm - Term - Box* aufgebaut. Diese Verweise ermöglichen einerseits das Auffinden der Netzelemente, die aus einem Programmkonstrukt generiert wurden, und umgekehrt das Auffinden des zu einer Stelle oder Transition gehörenden Programmabschnitts.

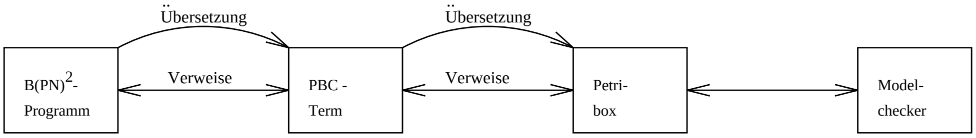


Abbildung 5.1.1 : *Ausschnitt aus dem* ***PEP****-System*

5.2 Anwendungsbeispiele für Verweise

In verschiedenen Situationen hat sich die Anwendung von Verweisen als hilfreich erwiesen.

Da bereits aus $B(PN)^2$-Programmen, die nur aus wenigen Anweisungen bestehen, große, auf den ersten Blick unübersichtliche Petriboxen entstehen können, ist der Zusammenhang zwischen Programm und Netz nicht immer offensichtlich.

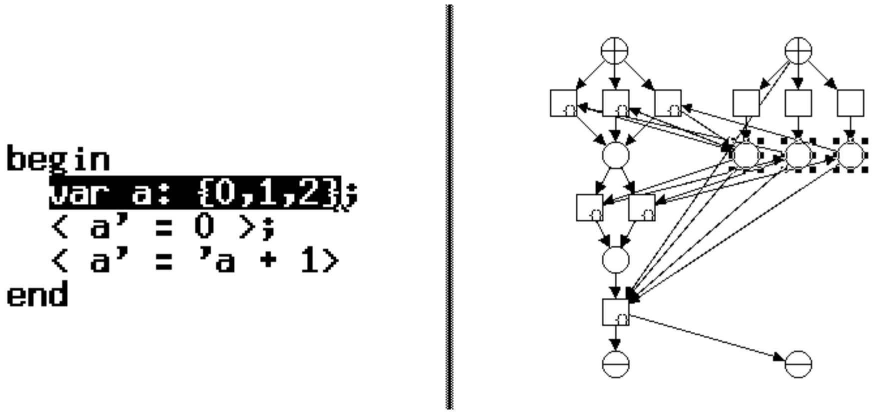


Abbildung 5.2.2 : *Verweis einer Variablendeklaration*

Insbesondere beim Erlernen der Programmiersprache $B(PN)^2$ ist es hilfreich zu wissen, welche Stellen des Netzes aus welchen Deklarationen entstanden sind und welche Transitionen welchen Anweisungen entsprechen. Somit unterstützen Verweise das Verständnis für die Übersetzung und erläutern, warum gerade diese Petribox das Verhalten des Programmes darstellt.

Abbildung 5.2.2 zeigt ein Beispiel für einen Verweis. Im B(PN)^2-Programm wurde die Deklaration der Variablen a selektiert. In der Petribox werden die den Wertebereich repräsentierenden Stellen hervorgehoben dargestellt.

In Verbindung mit dem ebenfalls im **PEP**-System verfügbaren animierten Einzelschrittsimulator für Petrinetze ergeben sich weitere Möglichkeiten.

Da für jeden möglichen Wert des Wertebereichs einer Variablen genau eine Stelle erzeugt wird und durch die Struktur der Box sichergestellt ist, daß aus dieser Stellenmenge bei jeder erreichbaren Markierung höchstens eine Stelle eine Marke trägt, kann während einer Simulation an dieser Stellenmenge abgelesen werden, ob die Variable gerade aktiv ist und wenn ja, welchen Wert sie gerade repräsentiert.

In Abbildung 5.2.3 hat die Variable a den Wert 1, da sich eine Marke auf der entsprechenden Stelle befindet.

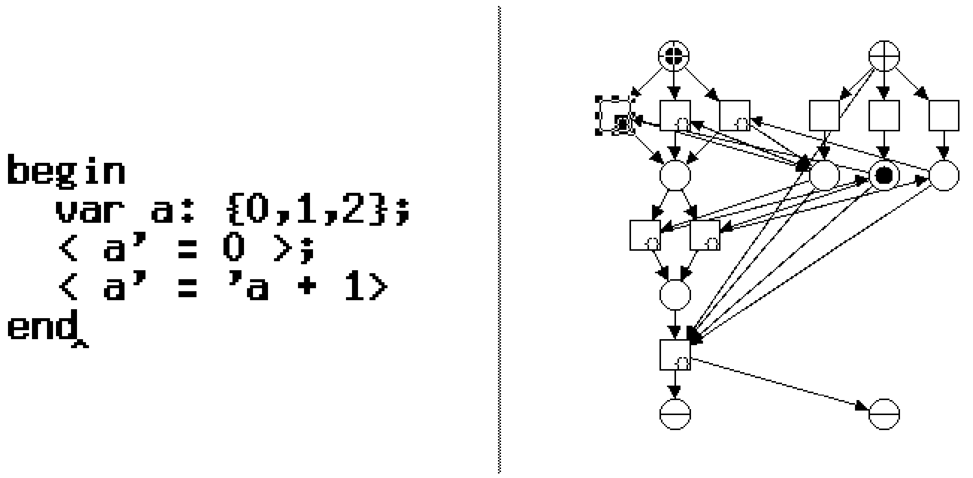


Abbildung 5.2.3 : *Die Variable* a *hat den Wert* 1

Zu jeder atomaren Anweisung des Programms werden eine oder mehrere Transitionen erzeugt. Entstehen mehrere Transitionen, so haben diese eine gemeinsame Vorbereichsstelle. Da bei der Übersetzung nur 1-sichere Petrinetze erzeugt werden, ist also sichergestellt, daß unter jeder erreichbaren Markierung höchstens eine dieser Transitionen schalten kann. Die Aktiviertheit einer dieser Transitionen während einer Simulation zeigt also, daß das Programm in einem Zustand ist, in dem die zugehörige Anweisung des Programms ausgeführt werden kann. Das Schalten dieser Transition repräsentiert die Ausführung der Anweisung.

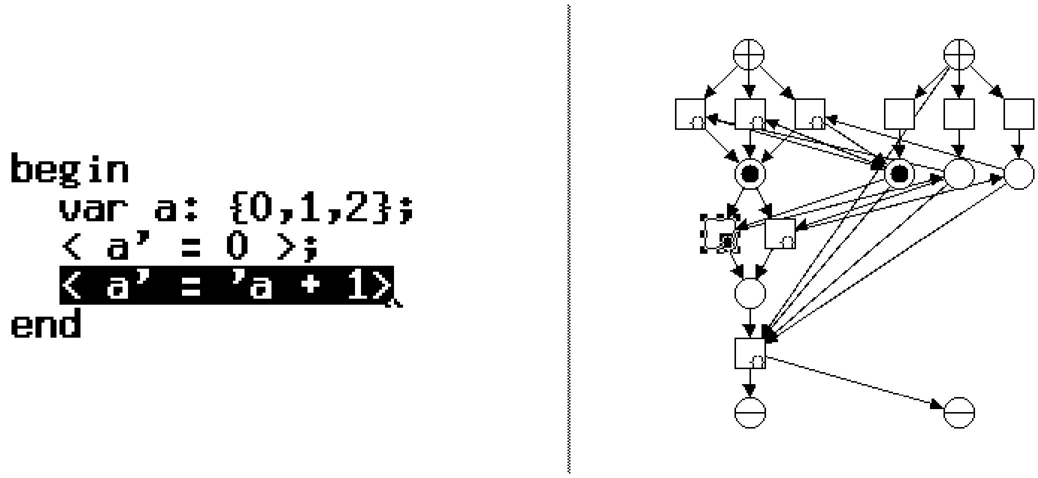


Abbildung 5.2.4 : *Die zur aktivierten Transition gehörende Anweisung*

Eigenschaften des Programms können mit Hilfe des in **PEP** verfügbaren effizienten Modelcheckers für Netze unter Zuhilfenahme der Verweise überprüft werden. Möchte man beispielsweise prüfen lassen, ob zwei Anweisungen des Programms sich gegenseitig ausschliessen, so ermöglichen die Verweise eine Bestimmung der Transitionen, die diesen Anweisungen entsprechen. Der Formelgenerator kann nun für den gegenseitigen Ausschluß dieser Transitionen eine Formel generieren, die vom Modelchecker getestet werden kann. Analog hierzu kann auf einfache Weise getestet werden, ob beispielsweise zwei Variablen gleichzeitig den Wert 0 annehmen. Hierzu werden mittels der Verweise die beiden Stellen bestimmt, die für die beiden Variablen jeweils den Wert 0 repräsentieren. Auch hier kann der Formelgenerator eine Formel für den Modelchecker erzeugen.

5.3 Verweise zwischen Term und Petribox

Im **PEP**-System wird zu einem Term des Petribox-Kalküls zunächst der Syntaxbaum erzeugt, an dessen Blättern die elementaren Aktionen des Terms stehen. An den Knoten des Baumes stehen die verknüpfenden Operatoren. Die Übersetzung in eine Petribox erfolgt Bottom-Up, wobei für jede Aktion zunächst eine Basisbox erzeugt wird, die nur aus je einer Eingangs- und einer Ausgangsstelle sowie einer Transition besteht. Von der Eingangsstelle führt eine Kante zur Transition und von dort eine weitere Kante zur Ausgangsstelle. Da die Ein- und Ausgangsstellen bei der Verknüpfung mit anderen Boxen zu internen Stellen verschmelzen, wird nur ein Verweis zwischen dem Blatt des Syntaxbaums und der entstandenen Transition aufgebaut.

Operatoren an den Knoten des Syntaxbaums beschreiben lediglich die Verknüpfung der Basisboxen. Daher entstehen hier keine neuen Transitionen. Bei der Verknüpfung wird aber eine Multiplikation der Eingangs- bzw. Ausgangsstellen der beteiligten Boxen durchgeführt. Der Wirkungsbereich der Operatoren wird durch die neu entstandenen Stellen repräsentiert. Deshalb werden hier Verweise zwischen dem Operatorknoten und den durch Multiplikation entstandenen Stellen erzeugt.

Die Termsprache des PBC enthält eigene Schlüsselworte zur Erzeugung von Kanal- und Datenboxen. Diese bestehen aus einer Menge von Stellen, die den Zustand des Kanals bzw. der Variablen darstellen, und einer Menge von Transitionen für die Übergänge zwischen den Zuständen. Da diese Transitionen während der Übersetzung mit Transitionen des Kontrollflusses verschmelzen oder entfernt werden, werden hier nur Verweise zwischen den Stellen und dem entsprechenden Blatt des Syntaxbaums aufgebaut.

Technisch werden in **PEP** die Verweise in der internen Darstellung des Syntaxbaums gespeichert. Hierzu enthält jeder Baumknoten eine Menge von Zeigern auf Netzknoten. Wird im Programmtext des PBC-Termes ein Textabschnitt selektiert und hierzu der Verweis auf die Petribox aktiviert, so werden zunächst die zu dieser Textpassage passenden Baumknoten bestimmt und anschliessend die Menge der zugehörigen Netzelemente im Petribox-Editor selektiert dargestellt. Werden umgekehrt im Petribox-Editor eine Stelle oder Transition selektiert und der Verweis auf den PBC-Termtext aktiviert, so werdern zuerst die Baumknoten bestimmt, die Verweise auf das selektierte Netzelement enthalten. Anschliessend wird die zugehörige Textstelle selektiert dargestellt.

5.4 Verweise zwischen B(PN)^2-Programm und Term

Die Erzeugung von Verweisen zwischen einem B(PN)^2-Programm und einem Term ist komplizierter, da es keine direkte Übereinstimmung zwischen Elementen des B(PN)^2-Programms und Knoten des Syntaxbaums des PBC-Terms gibt.

Die atomaren Anweisungen des B(PN)^2-Programms werden in entsprechend benannte Aktionen des PBC-Terms übersetzt. Aus einer Anweisung können mehrere Aktionen entstehen. Dementsprechend werden hier Verweise von der Anweisung zu den entsprechenden Knoten des Syntaxbaums erzeugt.

Die im B(PN)^2-Programmtext auftauchenden Sequenz-, Parallel- und Alternativoperatoren werden direkt in solche des PBC-Terms übersetzt, so daß die Verweise hier eindeutig sind. Ebenso wird mit Variablendeklarationen verfahren, da diese direkt in entsprechende Schlüsselworte übersetzt werden.

Da eine `do-od` -Schleife in ein Iterations-Konstrukt der PBC-Sprache übersetzt wird, wird ein Verweis zwischen dem Textabschnitt, der das `do-od` -Konstrukt enthält und dem Iterationsknoten des Syntaxbaums generiert.

Technisch gesehen werden auch diese Verweise in der internen Darstellung des Syntaxbaums des PBC-Terms gespeichert. Hierfür wird zu jedem Baumknoten Beginn und Ende der zugehörigen Textstelle des B(PN)^2-Programms gespeichert. Soll zu einer Stelle im Programmtext der passende Abschnitt des PBC-Terms gesucht werden, so werden zunächst wieder die passenden Knoten des PBC-Syntaxbaums bestimmt und die anschliessend die zugehörige Textstelle selektiert dargestellt.

Abbildung 5.4.5 verdeutlicht, wie die Verweise im **PEP**-System verwaltet werden. Zuerst wurden die zum selektierten Programmteil `< a' = 'a + 1 >` gehörenden Knoten des Syntaxbaums des PBC-Terms bestimmt (Mitte). Diese wiederum enthalten die Verweise zu den beiden Transitionen, die rechts selektiert dargestellt werden. Diese beiden Transitionen entstanden aus der Übersetzung des selektierten Programmteils.

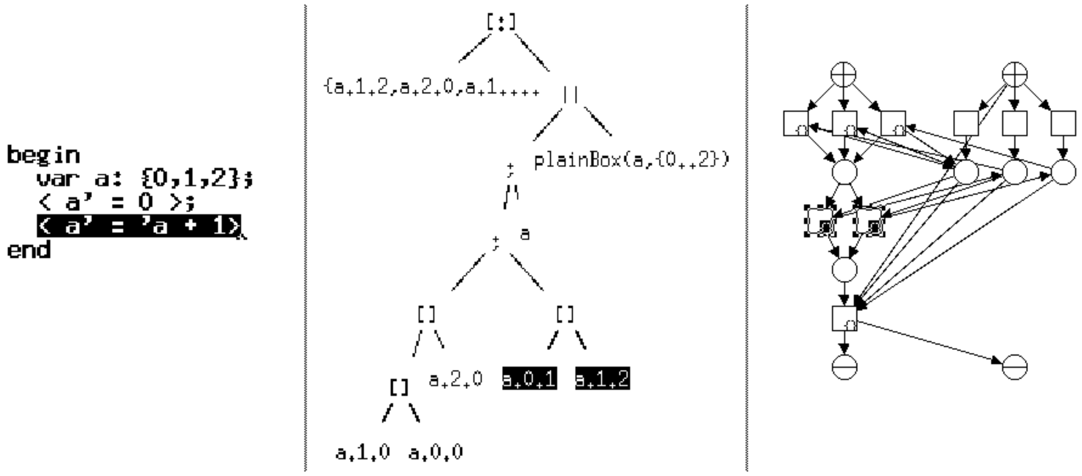


Abbildung 5.4.5 : *Verweis zwischen Programm, Syntaxbaum und Box*

5.5 Ausblick

Mit den oben aufgeführten Anwendungen sind die Einsatzmöglichkeiten der Verweise sicherlich noch nicht erschöpft. Eine weitere interessante, aber noch nicht für das **PEP**-System implementierte Anwendung ist die schrittweise Ausführung des $B(PN)^2$-Programms mittels der bereits existierenden Einzelschrittsimulation der zugehörigen Petribox. Hierzu mußs lediglich für jede aktivierte Transition des Netzes die zugehörige Anweisung im Programmtext-Editor hervorgehoben werden. Dies bedeutet dann, daß sich das Programm in einem Zustand befindet, in dem eine oder mehrere dieser hervorgehoben dargestellten Anweisungen ausgeführt werden können. Zusätzlich kann noch anhand der Markierung der Stellen, die die Wertebereiche darstellen, die jeweilige Belegung der Variablen angezeigt werden.

6 A Temporal Logic for B(PN)^2 Programs

Robert Riemann
Institut für Informatik
Universität Hildesheim
e-mail:riemann@informatik.uni-hildesheim.de

Abstract

An important purpose of a development- and verification environment for parallel programs is to specify and verify properties of a given program. For the programming language B(PN)^2 (Basic Petri Net Programming Notation), the semantics of a program is defined as a Petri net. This Petri net is a representative of an equivalence class, called a Petri Box.

The problem of verifying a formula for B(PN)^2 is solved in the **PEP** system (Programming Environment based on Petri nets) by verifying a corresponding formula on the Petri Box of the program.

A user of the **PEP** system can express a property of a program by means of a formula of a temporal logic defined on Petri nets. The elementary formulae of this logic are place related. This talk describes how one can define a similar logic on B(PN)^2 rather than on nets. The elementary formulae refer to certain points in the control flow of a program, rather than the places of a net. As these points get translated into places, the formulae of a program correspond to places in the net semantics.

The definition of the new logic on B(PN)^2 is given. The semantics of a formula on B(PN)^2 is given as a formula on the Petri Box of the corresponding program. Thus the model checking problem for B(PN)^2 is solved by reducing it to the model checking problem for Petri nets which is integrated in the **PEP** system already.

6.1 Introduction

The model checking problem for B(PN)^2 is reduced to the already solved model checking problem for 1-safe Petri nets by defining the semantics of the temporal logic on B(PN)^2 on the net semantics of the language.

$$\begin{array}{ccc} P & \xrightarrow{\text{program semantics}} & \Sigma \\ \text{B(PN)}^2\text{ program} & & \text{Petri net} \\ \text{with control annotation} & & \text{of } P \\ \rotatebox{-90}{$\models$} & & \rotatebox{-90}{$\models$} \\ \psi & \xrightarrow{\text{logic semantics}} & F \\ \text{formula of temporal} & & \text{formula on Petri net} \\ \text{logic } L_1 & & \end{array}$$

Figure 6.1.1 : *Model checking problem on B(PN)^2*

The basic idea is to extend the syntax and the semantic of B(PN)^2 with elements that on the one hand can be used as elementary formulae of the logic and on the other hand serve as labels of places of the semantic Petri net of a program. The syntactical extension of B(PN)^2 introduces labels for the places in the control flow of a program, while the Data Boxes which are the semantics of declarations are extended by labels for their places which represent the corresponding value of the variable. Once these extensions are introduced to B(PN)^2 the definition of the syntax and semantics of the temporal logic is straightforward.

This article is structured as follows: Section 6.2 defines the syntax of the parallel programming notation B(PN)^2 with control annotations. Its net semantics in terms of Petri Boxes is illustrated in the same section by a small example. In Section 6.3 we define the temporal logic L_1 on B(PN)^2.

6.2 The programing notation B(PN)^2

6.2.1 Syntax of B(PN)^2 with control annotation

In order to address the control points of a B(PN)^2 program the original syntax of B(PN)^2 is extended by names for these points which is called B(PN)^2 with control annotation.

Table 1 describes the syntax of B(PN)^2 with control annotation. Syntactic metavariables are printed in `teletype font` while keywords are printed in **bold**. Other elements of the language are printed *italic*.

`program`	::=	`block`
`block`	::=	$\{\{Bf(block)\}\}$ **begin** `scope` **end** $\{\{Af(block)\}\}$
`scope`	::=	`decl com`
`decl`	::=	ε | **var** var-name: `type`; `decl`
`type`	::=	`set` | **chan** b **of** `set`
`com`	::=	$\{\{bf(\langle$`expr`$\rangle)\}\}$ $\langle$`expr`$\rangle$ $\{\{af(\langle$`expr`$\rangle)\}\}$ |
		`com`; `com` | `com` $\|$ `com` | `block` |
		do `com` **enter**: `alt-set` **od**
`expr`	::=	$'w$ | w' | $c?$ | $c!$ | `const` | `expr` *op* `expr` | *op* `expr`
`const`	::=	**true** | **false** | z $(z \in Z)$
`alt-set`	::=	`com`; **repeat** | `com`; **exit** | `alt-set` $\square$ `alt-set`

Table 1: Basic syntax rules of B(PN)^2 with control annotation

The function f maps each atomic command of a program to a unique name. For the remainder we will assume an enumeration of the atomic commands. The letters b and a in front of the function value of f abbreviate *before* and *after* respectively. As well as the atomic commands f maps blocks of the program to unique names which are here numbers as well. The capitals B and A abbreviate *Before* and *After*. Thus syntactic identical commands and blocks get different names by f.

$\langle$expr$\rangle$ denotes the atomic execution of the expression `expr`. Variables are allowed in two different forms preprimed ($'w$) denoting its pre-value just before the execution of the command but stating nothing about its post-value and postprimed (w') denoting its post-value just after the execution of the command but stating nothing about its pre-value. Communication, over a channel c, is also included in B(PN)^2, with $c!$ performing an output communication, and $c?$

an input communication. Channels with capacity 0 realize synchronous communication while asynchronous communication is performed through channels with capacity greater than 0.

The expressions allow any execution, as long as they describe an assignment of values yielding the value **true** for the expression. Any other executions are disallowed.

The **do** command combines the traditional **if** (in its **exit** construct) with the traditional **while** command (in its **repeat** construct). We define here the syntax of the **do** command by an initialization command which is executed at the beginning of the iteration. With $\langle$**true**$\rangle$ as the initialization command the original B(PN)^2 semantics of the **do** construct as defined in [10] is realized.

We denote the type or domain of a variable x, or a channel c respectively by $VAL(x)$, or $VAL(c)$ respectively.

The sets of sets that are derived by the syntax, given above, are called *placeset*, an element of a placeset is called *controlset* whose elements are *name-elements*. The following Figure 6.2.2 illustrates these notions.

$$\underbrace{\{\underbrace{\{B1, \overbrace{b1}^{name-element}, b2\}}_{controlset}, \{B1, b3\}\}}_{placeset}$$

Figure 6.2.2 : *Explanation of the notion name-element, controlset and placeset*

In order to build correct labels for the places in the semantic net of a program additional rules are necessary. The syntactical rules given so far produce singletons, i.e. each placeset contains a single controlset which itself contains a single name-element. The set multiplication denoted by $\odot$ is used to join controlsets and placesets in order to produce the labels for the control points of a program. The multiplication of two placesets $\mathcal{R}$ and $\mathcal{S}$ produces the set of unions of an element in $\mathcal{R}$ and an element in $\mathcal{S}$.

Definition 6.2.3 : *Multiplication of sets, $\odot$*

Let A and B be two sets of sets.

We define $A \odot B = \{a \cup b \mid a \in A, b \in B\}$. ■ 6.2.3

For example is $\{\{a1\}\} \odot \{\{b2\}\} = \{\{a1, b2\}\}$, the following example motivates the notion 'set multiplication': $\{\{a1, b2\}, \{a3\}\} \odot \{\{b4\}, \{a3\}\} = \{\{a1, b2, b4\}, \{a1, b2, a3\}, \{a3, b4\}, \{a3\}\}$.

From now on we will denote the placeset in front of the command com by $^\bullet(com)$ and the placeset after com by $(com)^\bullet$.

Using set multiplication the following additional syntactic rules of B(PN)^2 with control annotation are defined. We will refer to these rules as *name joining rules*.

Definition 6.2.4 : *Name joining rules*

Sequence rule:

$$\left({}^{\bullet}(com_1) \quad \mathtt{com}_1 \quad (com_1)^{\bullet}\right) \; ; \left({}^{\bullet}(com_2) \quad \mathtt{com}_2 \quad (com_2)^{\bullet}\right)$$
$$::=$$
$${}^{\bullet}(com_1) \quad (\mathtt{com}_1 \quad (com_1)^{\bullet} \odot {}^{\bullet}(com_2) \quad \mathtt{com}_2) \quad (com_2)^{\bullet}$$

Concurrency rule:

$$\left({}^{\bullet}(com_1)\ \mathtt{com}_1(com_1)^{\bullet}\right) \quad \| \quad \left({}^{\bullet}(com_2)\ \mathtt{com}_2(com_2)^{\bullet}\right)$$
$$::=$$
$${}^{\bullet}(com_1) \cup {}^{\bullet}(com_2) \ \ (\ \mathtt{com}_1 \quad \| \quad \mathtt{com}_2) \ \ (com_1)^{\bullet} \cup (com_2)^{\bullet}$$

Exit rule:

$$\left({}^{\bullet}(com)\ \mathtt{com}\ (com)^{\bullet};\ \mathbf{exit}\right) \ ::= \ {}^{\bullet}(com)\ (\mathtt{com};\ \mathbf{exit})\,(com)^{\bullet}$$

Repeat rule:

$$\left({}^{\bullet}(com)\ \mathtt{com}\ (com)^{\bullet};\ \mathbf{repeat}\right) \ ::= \ {}^{\bullet}(com) \odot (com)^{\bullet} \ \ (\ \mathtt{com};\mathbf{repeat}\ ) \ \ \{\emptyset\}$$

Choice rule:

$$\left({}^{\bullet}(alt-set_1)\ \mathtt{alt\text{-}set}_1\ (alt-set_1)^{\bullet} \quad \square \quad {}^{\bullet}(alt-set_2)\ \mathtt{alt\text{-}set}_2\ (alt-set_2)^{\bullet}\right)$$
$$::=$$
$${}^{\bullet}(alt-set_1) \odot {}^{\bullet}(alt-set_2) \ \ (\mathtt{alt\text{-}set}_1 \quad \square \quad \mathtt{alt\text{-}set}_2) \ \ (alt-set_1)^{\bullet} \odot (alt-set_2)^{\bullet}$$

Iteration rule:

$$\left(\mathbf{do}\ {}^{\bullet}(com)\ \mathtt{com}\ (com)^{\bullet};\ \mathbf{enter}\colon {}^{\bullet}(alt-set)\ \mathtt{alt\text{-}set}\ (alt-set)^{\bullet}\mathbf{od}\right)$$
$$::=$$
$${}^{\bullet}(com)\left(\mathbf{do}\ \mathtt{com};\ \mathbf{enter}\colon (com)^{\bullet} \odot {}^{\bullet}(alt-set)\ \mathtt{alt\text{-}set}\ (com)^{\bullet} \odot {}^{\bullet}(alt-set)\ \mathbf{od}\right)(alt-set)^{\bullet}$$

Block rule:

$$\{\{Bf(block)\}\}\ \mathbf{begin}\ \mathtt{decl};\ {}^{\bullet}(com)\ \mathtt{com}\ (com)^{\bullet}\ \mathbf{end}\ \{\{Af(block)\}\}$$
$$::=$$
$$\{\{Bf(block)\}\} \odot {}^{\bullet}(com)\ \mathbf{begin}\ \mathtt{decl};\ \mathtt{com}\ (com)^{\bullet}\ \mathbf{end}\ \{\{Af(block)\}\}$$

■ 6.2.4

The application of the name joining rules and the derivation of a $B(PN)^2$ program with control annotation is demonstrated by the following example. $1, \ldots, 6$ are abbreviations for atomic commands of the form ⟨expr⟩.

```
program  ~1~>  {basic syntax}
               {{B1}}
               begin
                 var x : {0,1};
                 {{b1}} 1 {{a1}};
                 {{b2}} 2 {{a2}} || {{b3}} 3 {{a3}};
                 do {{b4}} 4 {{a4}} enter:
                      {{b5}} 5 {{a5}}; exit
                  □ {{b6}} 6 {{a6}}; repeat
                 od
               end
               {{A1}}

         ~2~>  {exit and repeat rule}
               {{B1}}
               begin
                 var x : {0,1};
                 {{b1}} 1 {{a1}};
                 {{b2}} 2 {{a2}} || {{b3}} 3 {{a3}};
                 do {{b4}} 4 {{a4}} enter:
                      {{b5}} 5; exit {{a5}}
                  □ {{b6, a6}} 6; repeat {∅}
                 od
               end
               {{A1}}
```

$\overset{3}{\leadsto}$ {concurrency rule}
$\{\{B1\}\}$
begin
  var $x : \{0,1\}$;
  $\{\{b1\}\}$ 1 $\{\{a1\}\}$;
  $\{\{b2\},\{b3\}\}$ 2 $\|$ 3 $\{\{a2\},\{a3\}\}$;
  do $\{\{b4\}\}$ 4 $\{\{a4\}\}$ **enter**:
    $\{\{b5\}\}$ 5; **exit** $\{\{a5\}\}$
  $\square$ $\{\{b6, a6\}\}$ 6; **repeat** $\emptyset$
  od
end
$\{\{A1\}\}$

$\overset{4}{\leadsto}$ {choice rule}
$\{\{B1\}\}$
begin
  var $x : \{0,1\}$;
  $\{\{b1\}\}$ 1 $\{\{a1\}\}$;
  $\{\{b2\},\{b3\}\}$ 2 $\|$ 3 $\{\{a2\},\{a3\}\}$;
  do $\{\{b4\}\}$ 4 $\{\{a4\}\}$ **enter**:
  $\{\{b5, b6, a6\}\}$
    5; **exit**
  $\square$ 6; **repeat**
  $\{\{a5\}\}$
  od
end
$\{\{A1\}\}$

$\overset{5}{\leadsto}$ {iteration rule}
$\{\{B1\}\}$
begin
  var $x : \{0,1\}$;
  $\{\{b1\}\}$ 1 $\{\{a1\}\}$;
  $\{\{b2\},\{b3\}\}$ 2 $\|$ 3 $\{\{a2\},\{a3\}\}$;
  $\{\{b4\}\}$
  do 4 **enter**:
  $\{\{a4, b5, b6, a6\}\}$
    5; **exit**
  $\square$ 6; **repeat**
  $\{\{a4, b5, b6, a6\}\}$
  od
  $\{\{a5\}\}$
end
$\{\{A1\}\}$

$\overset{6}{\leadsto}$ {sequence rule}
$\{\{B1\}\}$
begin
  var $x : \{0,1\}$;
  $\{\{b1\}\}$
  1
  $\{\{a1, b2\},\{a1, b3\}\}$
  2 $\|$ 3
  $\{\{a2, b4\},\{a3, b4\}\}$
  do 4 **enter**:
  $\{\{a4, b5, b6, a6\}\}$
    5; **exit**
  $\square$ 6; **repeat**
  $\{\{a4, b5, b6, a6\}\}$
  od
  $\{\{a5\}\}$
end
$\{\{A1\}\}$

$\overset{7}{\leadsto}$ {block rule}
$\{\{B1, b1\}\}$
begin
  var $x : \{0,1\}$;
  1
  $\{\{a1, b2\},\{a1, b3\}\}$
  2 $\|$ 3
  $\{\{a2, b4\},\{a3, b4\}\}$
  do 4 **enter**:
  $\{\{a4, b5, b6, a6\}\}$
    5; **exit**
  $\square$ 6; **repeat**
  $\{\{a4, b5, b6, a6\}\}$
  od
  $\{\{a5\}\}$
end
$\{\{A1\}\}$

6.2.2 Semantics of B(PN)^2

The semantics of B(PN)^2 is defined in terms of Box expressions. Box expressions and their compositional semantics, a class of labelled place/transition nets referred to as Petri Boxes, are comprised in the Petri Box Calculus (PBC) [6].

Instead of the formal definition of the semantics of B(PN)^2 with control annotation we will give here an example that illustrates the translation of a B(PN)^2 program into a Petri Box. For a full definition and further examples the reader is referred to [6, 10, 25, 33] as well as to the next chapter.

The following B(PN)^2 program with control annotation P will serve for the remainder of this paper as an example. It is assumed, that the four atomic actions in P are implicitly numbered from 1 to 2 via the function f. $\oplus$ denotes the addition modulo 1.

$$
\begin{array}{rl}
P \equiv & \{\{B1, b1\}, \{B1, b2\}\} \\
& \textbf{begin var } x : \{0,1\}; \\
& \quad \langle x' = 1 \rangle \parallel \langle x' = {'x} \oplus 1 \rangle \\
& \quad \{\{a1\}, \{a2\}\} \\
& \textbf{end} \\
& \{\{A1\}\}
\end{array}
$$

The program is choosen in order to illustrate for a program using the parallel operator the net semantics of B(PN)^2.

Following the notation of [10] we denote the Petri Box of a program P as $Box(E(P))$. The following Figure 6.2.5 shows the Data Box for the plain variable x, Figure 6.2.6 shows the control-flow of the program P without scoping.

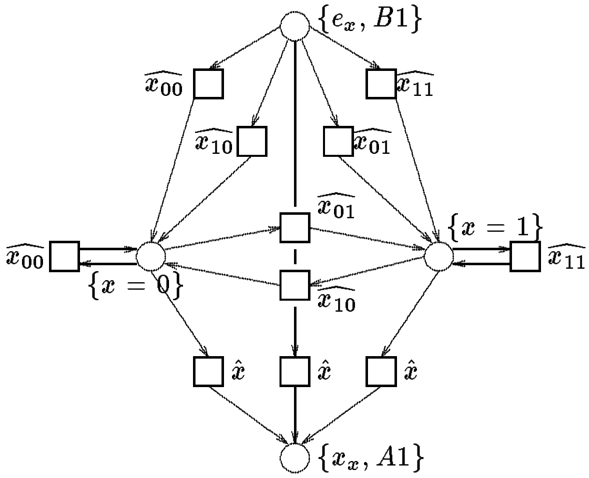


Figure 6.2.5 : $Box\,(\{\{e_x, B1\}\}\, D(x, \{0,1\})\, \{\{x_x, A1\}\})$

Scoping performs the synchronisation between data flow, described by the Data Boxes and the control flow. The scoping operation has no effect on the places of the given subnets but introduces new transitions. The Data Boxes are extended by a labelling of the places with respect to their original definition in [10] that gives the semantics of these places. For instance whenever the place labelled with $x = 1$ is marked, the plain variable x has the value 1. The labels of entry and exit places are extended by the name-elements that indicate the block in which the variable is declared. To get the formal definition of the translation into these Petri Boxes the reader is referred to the above mentioned articles. The Petri Box representing the semantics of the program P, i.e. the Petri Box that results out of the scoping of the two Data Boxes and the control-flow net is shown at the end of the section in Figure 6.3.12.

In this context it is important that a B(PN)^2 program with control annotation can be translated into a minimal represantative of the equivalence class that represents the net semantics of the program. The places of this representative are labelled with the controlsets of the program and the labels introduced for places of the Data Boxes as described above. In the following we will refer to the net semantics $Box(E(P))$ of a B(PN)^2 program with control annotation simply as its Petri Box.

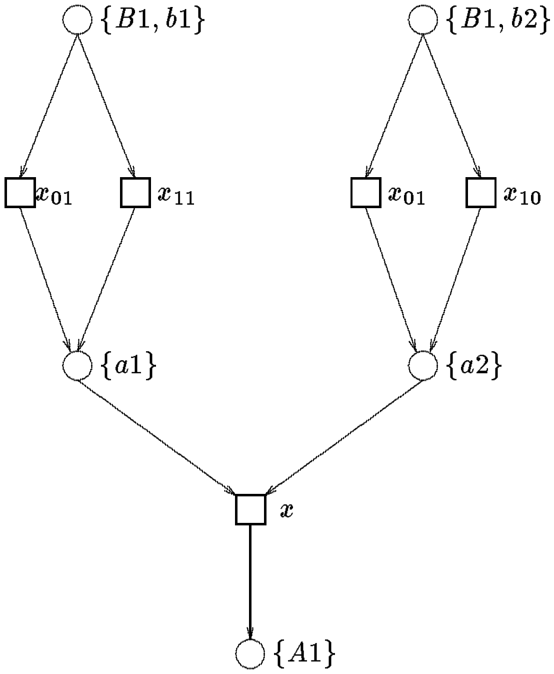


Figure 6.2.6 : *Control-flow net of program P without scoping*

6.3 A temporal logic for B(PN)^2

In this section we define a temporal logic L_1 on B(PN)^2 with control annotation. The logic L_1 extends propositional logic with a possibility operator. The designation L_1 is taken from [18].

The elementary formulae of the B(PN)^2 logic are *control formulae* and *value assertions*. The latter state a certain value of a program variable, the former state that program control has reached a point that is described by a controlset.

6.3.1 The syntax of L_1 formulae

Definition 6.3.7 : *Syntax of value assertion*

Let P be a B(PN)^2 program, $z \in VAR(P)$ a plain or channel variable of P and $i \in VAL(z)$ an element of the domain of z.

A *value assertion* over P is a formula of the form:

$$z = i.$$

■ 6.3.7

The value assertion $c = i$ for the channel variable c means that the next value that will be retrieved from the channel is i. The special symbol $\perp$ is used to state the emptyness of a channel in the value assertion $c = \perp$.

Using value assertions it is possible to express each property of a program concerning the data flow. Furthermore we allow abbreviations for more complex value assertions such as $x \leq 1$ for $x = 0 \vee x = 1$ for a plain variable x.

Definition 6.3.8 : *Control formula*

Let P be a B(PN)^2 program with control annotation.

Every controlset of P is a *control formula* over P. ■ 6.3.8

Definition 6.3.9 : *Syntax of L_1*

Let P be a B(PN)^2 program with control annotation,

The set of formulae of L_1 is defined by the following syntax:

$$\psi ::= \begin{cases} \textbf{true} & & & \text{Truth} \\ V & & & \text{Value assertion} \\ C & & & \text{Control formula} \\ \neg\psi & & & \text{Negation} \\ \psi_1 & \text{und} & \psi_2 & \text{Conjunction} \\ \Diamond\psi & & & \text{Possibly } \psi \end{cases}$$

■ 6.3.9

The propositional logic operators $\vee$ (disjunction), $\Rightarrow$ (implication) and **false** (falsity) are derived in the usual way by means of conjunction and negation. The temporal operator $\Box\psi$ (always) is defined as abbreviation of $\neg\Diamond\neg\psi$.

Intuitively the formula $\Diamond\psi$ means that at least one subsequent state exists which satisfies the formula ψ. Thus $\Box\psi$ means intuitively that there is no subsequent state in which the formula ψ is not valid.

A formula without temporal operator is called *propositional*.

6.3.2 Semantics of L_1 formulae

We first define the function *placelabel*() which takes control formulae and value assertions as its argument and maps them to places of the Petri Box of the program.

Definition 6.3.10 : *placelabel*()

Let P be a B(PN)^2 program with control annotation, $Box(E(P)) = (S, T, F, \lambda)$ the Petri Box of P, $V_1 \equiv x = i$ a value assertion on the plain Variable x of P, $V_2 \equiv c_b = i$ a value assertion on the channel variable c with capacity b and C be a controlset of P.

Define
$placelabel(V_1) = s$ with $s \in S : \lambda(s) = \{V_1\}$,
$placelabel(V_2) = s$ with $s \in S : \lambda(s) = \{V_2\}$ and
$placelabel(C) = s$ with $s \in S : \lambda(s) = C$ respectively. ■ 6.3.10

With the use of the function *placelabel*() we are now able to define the semantics of the temporal logic L_1 for B(PN)^2 programs with control annotation.
TheL_1 formulae are interpreted over a marking of the Petri Box of a program $Box(E(P))$. A marking satisfies an elementary formula if the place that is labelled with the elementary formula is marked.

Definition 6.3.11 : *Semantics of L_1*

Let P be a B(PN)^2 program with control annotation, $Box(E(P)) = (S, T, F, \lambda)$ the Petri Box of P, M_0 the initial marking, M and M' markings of $Box(E(P))$, τ a firing sequence and V a value assertion and C a control formula over P.

$$
\begin{array}{lll}
M \models \mathbf{true} & & \text{always} \\
M \models V & \text{iff} & M(placelabel(V)) = 1 \\
M \models C & \text{iff} & M(placelabel(C)) = 1 \\
M \models \neg\psi & \text{iff} & \text{not } M \models \psi \\
M \models \psi_1 \quad \text{und} \quad \psi_2 & \text{iff} & M \models \psi_1 \text{ and } M \models \psi_2 \\
M \models \Diamond\psi & \text{iff} & \exists \tau, M' : M \xrightarrow{\tau} M' \quad \text{und} \quad M' \models \psi
\end{array}
$$

Finally we define $P \models \psi$ if $M_0 \models \psi$. ■ 6.3.11

Using these definitions we are now able to formulate a L_1 formula which expresses the relational semantics of the program P:

$$\Diamond\,(\{a1\} \quad \text{und} \quad \{a2\} \quad \text{und} \quad (x = 0) \vee (x = 1))$$

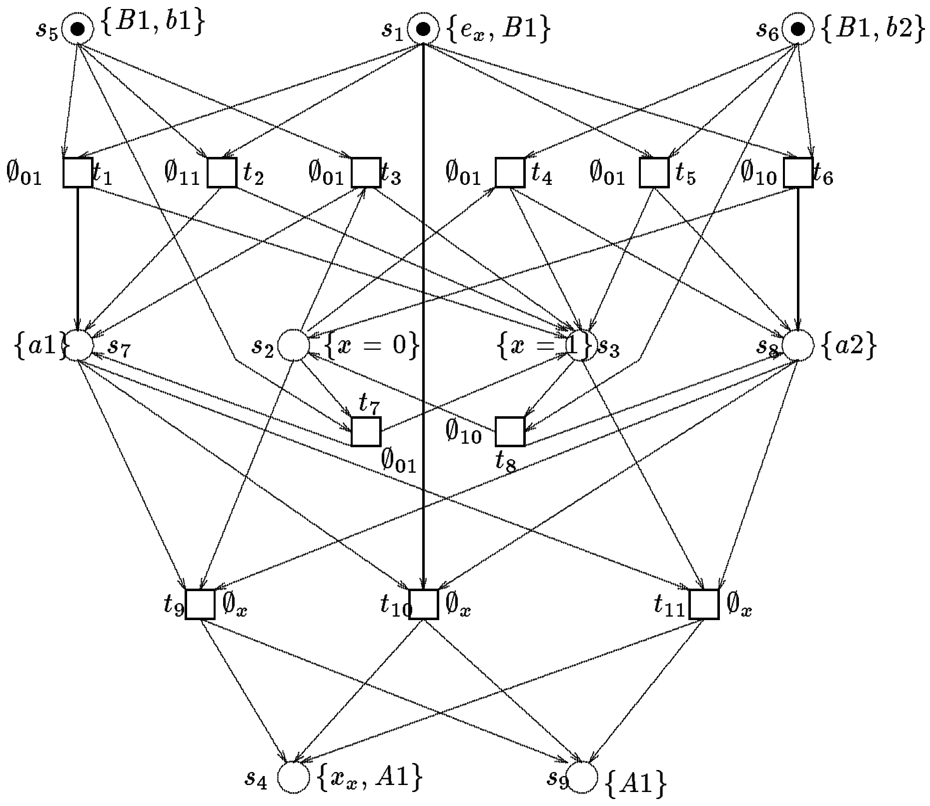


Figure 6.3.12 : *Net semantics of program P with scoping*

7 A Low-Level Net Semantics for B(PN)^2 with Procedures

Lars Jenner
Universität Hildesheim
Institut für Informatik
Marienburger Platz 22
D - 31141 Hildesheim
e-mail: jenner@informatik.uni-hildesheim.de

Abstract

Via the *Petri Box Calculus* PBC, the *Basic Petri Net Programming Notation* B(PN)^2 has been given a low-level net semantics in terms of Petri Boxes ([6],[10]). This paper adresses an extension of B(PN)^2 by procedures within the low-level net framework.

To achieve this, the semantics of the refinement and the recursion operator have been modified to satisfy the axiom of syntactic substitution. The consistency of this modification with the standard PBC is shown in [24].

B(PN)^2 procedures can be declared locally in every block and apply all constructs of B(PN)^2. They can take value and reference parameters, and when accessing global variables, a static binding is established. Parallel execution as well as direct and indirect recursion are possible. Recursion generally leads to an infinite low-level net semantics, and the problem to find a "sufficient" finite approximation is addressed. A high-level net semantics for procedures is discussed in [20].

Main parts of this approach are implemented in the **PEP** system.

7.1 Introduction

B(PN)^2 is a high-level parallel programming language that provides a block structure with global and/or local variables. Communication between concurrent processes can be established synchronously using channels with zero-capacity (handshake), and asynchronously by use of shared data or channels with arbitrary capacity. The basic elements of a B(PN)^2 program are atomic actions. An action is expression oriented and describes a relation between the values or states of the involved variables and channels before (preprimed) and after (postprimed) its execution. Execution of actions can be structured by sequential composition or iteration with nondeterministic choice, but also by parallel composition on every level. This feature distinguishes B(PN)^2 from many other parallel languages and provides a high degree of expressiveness. Conventional if-, for- and while-constructs are modelled by iteration with nondeterministic choice in a uniform manner. Despite its expressiveness, the syntax of B(PN)^2 is fairly short, and given in 7.10.1. Figure 7.1.1 shows a simple program example.

```
begin                   /* block structure */
    var v:[0,1];        /* declaration of variable v, domain {0,1} */
    ⟨ v'=1 ⟩            /* atomic action with assignment of 1 to v */
end                     /* end of block */
```

Figure 7.1.1 : *B(PN)^2 example program*

Every B(PN)^2 program can be mapped to an expression of PBC, which is a process algebra specifically designed to serve as an intermediate step towards a Petri net semantics of B(PN)^2. Atomic actions of B(PN)^2 programs are modelled by a nondeterministic choice from appropriate multi-actions of PBC. Multi-actions are multisets of action names. Action names form the communication interface of actions, and their set is partitioned by a bijection ($\hat{}$) called conjugation. PBC offers a sophisticated synchronisation operator that leads to a synchronous execution of actions which contain conjugate action names. Iteratively, all possible connections are established and a subsequent restriction operator forces synchronisation and closes the communication interface of the synchronised structure. This combination is called scoping and is used to model the block structure of B(PN)^2 programs. Control flow actions are put in parallel and synchronised with their conjugate data flow actions. After restriction, no access to local data is possible from outside the block.

In addition to constructs for sequence, choice, iteration and parallel composition, PBC allows a renaming of multi-actions, which is performed on the contained action names. Furthermore, any expression-variables that must be distinguished from multi-actions can be refined by arbitrary PBC expressions. The μ-operator for recursion is based on successive refinement operations.

As an extension of the PBC syntax, there are parametrized expressions for the data components of B(PN)^2 programs. They can be considered as shorthands for the data flow capabilities, as will become clear later on. The syntax of PBC is given in 7.10.2, the PBC semantics of B(PN)^2 in 7.10.3 and figure 7.1.2 shows the translation of the B(PN)^2 program of figure 7.1.1.

$[\{v, v_{00}, v_{01}, v_{10}, v_{11}\}:$	/* scoping for v, all possible actions */
$\quad \mathbf{V}(v, \{0,1\})$	/* Data Box for v ... */
$\quad \|$	/* ... in parallel to ... */
$\quad ((\{v_{01}\} \; \square \; \{v_{11}\})$	/* ... choice from multi-actions for assignment */
$;\{v\})]$	/* termination of Data Box and end of scoping */

Figure 7.1.2 : *PBC expression for the B(PN)^2 program*

It should be mentioned that action renaming, expression-variables, refinement and recursion are not necessary to define the semantics of B(PN)^2 programs so far. But they will all be used when modelling procedures. Expression-variables, refinement and recursion will be referred to as hierarchical constructs from now on, while the rest are elementary constructs.

By a semantic homomorphism, every PBC expression (or Box expression) is mapped to an equivalence class of Petri nets, called Box. The nets are 1-safe and their transitions are labelled by the multi-actions. Places can be entry, exit or internal places. There are always entry and exit places, no arcs into entry places, no arcs from exit places and all transitions have a nonempty preset and postset of places. The initial marking of a net is one token in every entry place and the final marking is one token in every exit place. The equivalence relation on nets which defines Boxes abstracts from the identities of places and transitions as well as from duplication of nodes, which has no observable effect on the nets behaviour. There is always (up to isomorphism) a unique minimal net representative for every Box, in which no two different nodes duplicate each other. A Box is called finite iff its minimal representative is finite, i.e. the sets of transitions and places are finite.

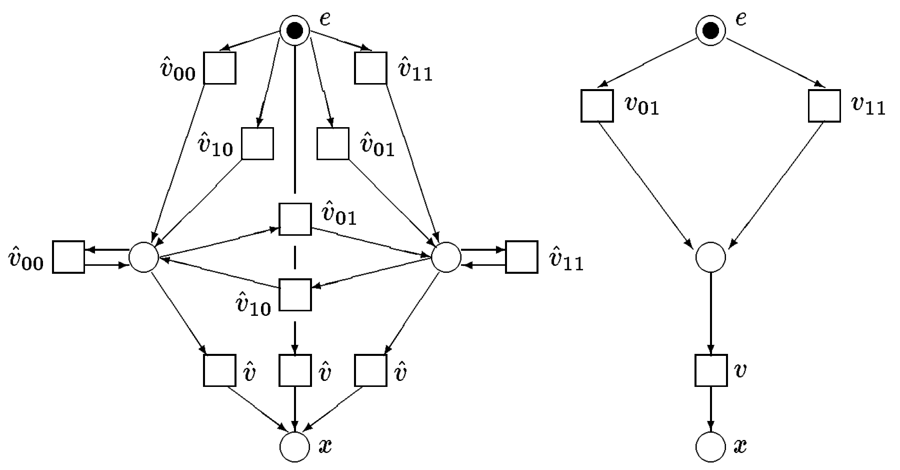


Figure 7.1.3 : *Net with Data Box and control flow without scoping*

The domain of Boxes is equipped with an algebra. Each of its operators corresponds homomorphically to an operator of PBC. In this sense, the Box semantics of PBC is compositional, and it is also denotational since recursion is treated by fixpoint techniques. Various properties of the operators, such as commutativity, associativity, substitutivity and idempotence are derived directly from this algebra and supply the PBC semantics with important flexibility. These properties extend up to the level of B(PN)^2 programs, which are now equipped with a net semantics. A detailed treatment can be found in [6].

The expressions for the data components of a B(PN)^2 program are modelled by special Data Boxes, which are integrated into the Box algebra.

Figure 7.1.3 shows the net semantics of our example *without* scoping, and in figure 7.1.4 the scoping is performed. Here scoping leads to internal actions ($\emptyset$) and the labels of the non-dead transitions are additionally subscripted with their origin.

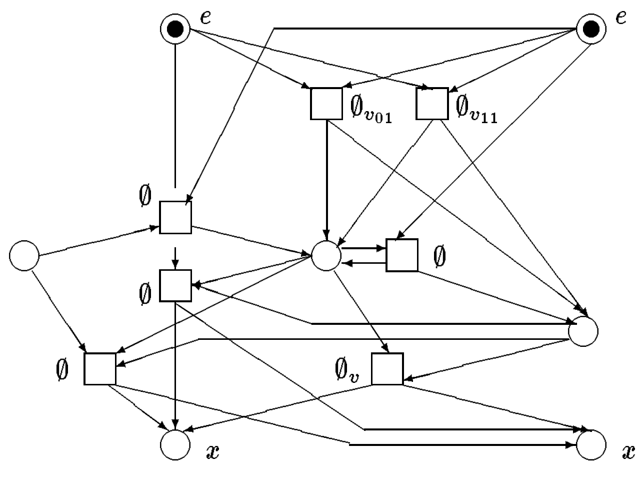


Figure 7.1.4 : *Net with scoping*

Every B(PN)^2 program with finite data has a finite net semantics. On the other hand, recursion in the PBC usually leads to an infinite net (the least upper bound of a chain in a complete partial order), even if the body of recursion has a finite net representation. This fact will influence the usefulness of recursive procedures, because finiteness as well as safeness are conditions for the applicability of the model checker.

Back on the level of $B(PN)^2$, there are still some desirable extensions of the language that would make it more convenient and competitive to other "user-friendly" high-level languages. While from a theoretical point of view, Turing-capability is already ensured by use of iteration and unbounded channels as a tape, if the aim is to model, analyse and verify complex concurrent systems with $B(PN)^2$, one would certainly miss such notions as stacks, arrays, records, functions, procedures, modules, etc. This paper addresses a controlled (i.e. semantically well-founded) extension of $B(PN)^2$ by procedures.

7.2 What is a Procedure ?

Before considering formal and technical details, the question raised above will be given an informal answer. We shall first describe procedures at the level of the $B(PN)^2$ syntax from the view of a programmer.

A procedure has a name and a body, which is simply a block. Inside the body, all constructs of $B(PN)^2$ defined so far and in the following are available. The declaration of a procedure is possible at the beginning of an arbitrary block, like that of variables and channels. From the point of its declaration, the name of the procedure can be used as a command, especially inside the body of the same procedure. Whenever there is an occurrence of the procedure name in the program (a procedure call), control flow branches to the execution of the body. If this call occurs inside the body of the same procedure, then it is a (direct) recursive call. After termination of the body, control flow returns to the point after the call. Each procedure call leads to an individual activation of the procedure.

Inside the body of a procedure, all identifiers that have been declared so far can be used. If an identifier is declared outside the body, it is regarded as global with respect to the body.

The body of a procedure can also dispose of particular local variables that are initialized individually at every procedure call. They are christened value parameters and have to be declared together with the name of the procedure. The event of initialisation is called argument passing at call-by-value.

If there are several activations of a procedure at the same time (by parallel execution or recursion), then there exist distinguished and independent incarnations of the body's local data and value parameters for each activation.

Furthermore, commands in the body can refer to identifiers which are not associated to local Data Boxes. They are called reference parameters, and they also have to be declared together with the name of the procedure. At a procedure call, each reference parameter is assigned an already declared identifier of the same type. These identifiers can be reference parameters themselves. The event of assignment is called argument passing at call-by-reference. Whenever there is an occurrence of a reference parameter inside the body, the correspondig action is performed on the passed argument.

A special construct allows the declaration of several procedures at the same level, what is usually established by a forward-statement. These procedures may call each other, yielding indirect recursion.

scope	::=	... | **proc** pdecl ; scope | **procgrp** proclist **endgrp** ; scope
pdecl	::=	`proc-name` block | `proc-name` (parlist) block
parlist	::=	`par-name` : set | **var** `par-name` : set |
		var `par-name` : **chan** k **of** set | parlist , parlist
proclist	::=	pdecl | proclist , proclist
com	::=	... | `proc-name` | `proc-name` (arglist)
arglist	::=	expr | `var-name` | arglist , arglist

Figure 7.2.5 : *Extension of B(PN)^2 syntax by procedures*

Figure 7.2.5 shows a corresponding extension of B(PN)^2 syntax. The **procgrp**-construct models declaration "at the same level" of all procedures in proclist. If the declaration of a parameter is prefixed by the **var**-keyword, it is a reference parameter, otherwise it is a value parameter; channels cannot be value parameters. Arguments for reference parameters must be plain identifiers (`var-name`), arguments for value parameters can also be value expressions (expr).

With this informal description in mind we are now ready to deal with the following question.

7.3 What is a Net Semantics of Procedures ?

In the framework of B(PN)^2, PBC and Boxes, a fundamental observation is that the execution of a procedure body is usually not an atomic action. In general, the semantics of the body of a procedure is a net with several transitions. For this reason, we cannot extend the definition of procedures to functions, since their call, execution and value return inside an atomic action is impossible. However, the use of reference parameters allows to pass results from a procedure back to the calling environment.

There are at least two different approaches towards gaining a net semantics of B(PN)^2 procedures. The first one is related to the way procedures are treated in code generating compilers. Such compilers produce a unique piece of code for every procedure to which control flow is branched at a call. As there might be several activations of a procedure at the same time, a stack is used to provide memory for the different incarnations of local data and value parameters. In parallel systems one also has to keep track of the distinct branches of control flow, which is done by explicit bookkeeping of call and return addresses. Also, arguments for reference parameters are referred to via memory addresses.

Transferred to our framework, this approach leads to a unique Procedure Box for every procedure declaration. A Procedure Box can be regarded as a generalized Data Box that includes data flow as well as control flow capabilities. Although this approach seems to be very elegant and natural, it has severe drawbacks:

- Even if we use a stack, several activations of a procedure at the same time fill the Procedure Box with indistinguishable tokens from the different activations. Control flow is thereby ambiguous and the important safeness property is lost. A solution could be the serialization of procedure calls, but then we have to give up recursion and loose functionality in parallel execution.
- Each procedure call and return from the body requires an additional transition that restricts the usability of procedures at the entry point of a branch in the iteration construct. To overcome this problem, one would have to allow arcs back to the entry points of the procedure body and thereby influence the properties of the Box algebra.

- While arguments for value parameters could eventually be passed using a channel, it is not clear how reference parameters can be treated. In our framework, there is no notion of addresses and the static structure of nets prohibits any modification at runtime. A solution could be to copy the state of the argument into a local variable or channel of the body and vice versa when returning (call-by-value-result). But this rules out any interference of actions inside and outside the procedure which access the argument. If on the other hand such encapsulation is explicitly desired, it can also be modelled by use of reference parameters and semaphores.

The situation changes if we substitute low-level nets by high-level nets. But in our framework, the problems mentioned above cause us to seek alternatives to the Procedure Box approach.

Related to procedures in programming languages are macros, which are parametrized parts of a program that are not compiled into unique code but are literally inserted at points of their usage, together with their parameters. Recursion of macros is usually not possible, because one would have to deal with infinite code, definitely not a desirable effect. Nevertheless, Box algebra can handle infinite nets. On the other hand, all practically relevant computations take place in a finite state space, and the problem is how to find a finite approximation to the infinite net that is large enough to model all its relevant behaviour. In general this seems to be impossible, but there are solutions for nontrivial cases, as will be discussed later on.

The macro approach leads to unambiguous control flow, safe nets and pleasant parameter passing mechanisms and will therefore be studied further in the remainder of this paper.

The idea of macros is naturally associated to the use of the refinement operator in PBC. A $B(PN)^2$ procedure call should be translated into an expression-variable, and this variable should be refined by the expression corresponding to the body of the procedure. This body expression can be subject to the recursion operator, whenever there is a recursive call. Unambigous control flow is ensured, as every procedure call receives its own body expression. Value parameters are local variables, and reference parameters are referred to via an appropriate renaming function. There are no extra call and return transitions, and by the absence of value parameters, procedures can reasonably be used at the beginning of an iteration branch.

7.4 Refinement or Syntatic Substitution?

Unfortunately, things are not that idyllic. The refinement operator turns out to be compositional but does not satisfy the axiom of syntactic substitution. Figure 7.4.6 indicates that refinement and scoping in general do not commute.

$$[a:\{\hat{a}\} \parallel X][X \leftarrow \{a\}] \qquad \neq \qquad [a:\{\hat{a}\} \parallel \{a\}]$$

Figure 7.4.6 : *Refinement is not equal to syntactic substitution*

This has severe consequences, as will be seen when treating argument passing to value parameters. Hence we need the commutativity of refinement and scoping. Otherwise, the refinement approach looses its functionality and degrades to simple insertion of bodies for procedure calls. Recursion is then ruled out, and parameter passing becomes elaborate and unpleasant, maybe even impossible.

There are two approaches to overcome this problem. The first one is the notion of "pending nets" introduced by R.Devillers in [14]. It has the advantage to preserve compositionality, but it seems to be technically expensive, as it deals with eventually infinite additional labellings and thereby goes beyond the scope of our framework.

The second solution, then, is to pay with compositionality for the commutativity of refinement and scoping. We simply restrict the Box algebra to elementary constructs and perform syntactical reductions on Box expressions with hierarchical operations, before translating them into Boxes. This implies that each occurrence of an expression-variable has to be bound by either refinement or recursion. However, this is guaranteed when dealing with correct B(PN)^2 programs. It can be shown that this modification is consistent with the standard PBC, i.e. leads to the same net semantics, if the refining expressions are subject to renaming. The basic ideas can be found in [24], and 7.10.2 gives the mentioned reduction rules. For our purposes, we can simply regard refinement and recursion as to be syntactically performed on the domain of Box expressions, before any elementary operation is applied.

Although this might seem to be a long-winded description of syntactical substitution, there is a subtile difference concerning recursion. In this case we accept an infinite net semantics but no infinite expressions, as would be generated by successive syntactic substitutions. Recursion should be defined as the least upper bound of a chain of net approximations. The well-definedness of this upper bound has not been guaranteed yet, but this seems to be a technical problem rather than a fundamental one (cf. [5]). From a practical point of view, it is interesting to find a finite approximation that reflects all possible behaviour of the system, excluding possible divergence. However, in [24] it is shown that all approximations are isomorphic and form a chain.

The main reason for choosing this second approach is its adequate and convenient implementability in the present **PEP** system, where we only consider appropriate finite approximations to recursion.

7.5 Static or Dynamic Binding ?

The example given in figure 7.5.7 illustrates an effect that derives from the newly gained commutativity of refinement and scoping: we get a dynamic binding to the inner variable x.

begin	$[\{x, x_{00}, x_{01}, x_{10}, x_{11}\}:$
var x:[0,1];	$\mathbf{V}(x, \{0,1\}) \parallel (($
proc P **begin** ⟨ x'=1 ⟩ **end** ;	$[\{x, x_{00}, x_{01}, x_{10}, x_{11}\}:$
begin	$\mathbf{V}(x, \{0,1\}) \parallel (($
var x:[0,1];	P
P	$); \{x\})]$
end	$); \{x\})]$
end	$[P \leftarrow (\{x_{01}\} \ \square \ \{x_{11}\})]$

Figure 7.5.7 : *Commutative refinement leads to dynamic binding*

If we want to achieve a static binding to the outer variable x, a renaming of all B(PN)^2 identifiers has to be performed. We simply number them in the order of their declaration. This is done by the compiler in a deterministic manner, and each occurrence of an identifier in an action is associated to the last declaration of this name. Now all textually equal identifiers can be distinguished. The result is shown in figure 7.5.8.

$$
\begin{array}{l}
[\{x^1, x^1_{00}, x^1_{01}, x^1_{10}, x^1_{11}\} : \mathbf{V}(x^1, \{0,1\}) \parallel ((\\
\quad [\{x^2, x^2_{00}, x^2_{01}, x^2_{10}, x^2_{11}\} : \mathbf{V}(x^2, \{0,1\}) \parallel ((\\
\qquad P\\
\quad); \{x^2\})]\\
); \{x^1\})]\\
[P \leftarrow (\{x^1_{01}\} \;\square\; \{x^1_{11}\})]
\end{array}
$$

Figure 7.5.8 : *Renaming of identifiers leads to static binding*

This renaming is the price we have to pay for decreased compositionality. But it is worth doing so, as will be seen when treating argument passing to value parameters.

Another observation is that the refinement of the body can now take place directly *after* the block of its declaration. This is possible because of the commutativity and yields more legible translations. In the following, we will also augment readability by choosing textually different identifiers and omitting the enumbering.

7.6 Direct and indirect Recursion

Figure 7.6.9 shows a program with a procedure that may call itself recursively. Direct recursion is obtained by simply prefixing the μ-operator to the body expression, as shown in the translation.

begin	$[\{x, x_{00}, x_{01}, x_{10}, x_{11}\}:$
var x:[0,1];	$\mathbf{V}(x, \{0,1\}) \parallel (($
proc P **begin**	P
do ⟨x'=1⟩; **exit** $\square$ P; **exit od**	$); \{x\})]$
end ;	$[P \leftarrow \mu P.($
P	$[\emptyset * \mathbf{stop} * (\{x_{01}\} \;\square\; \{x_{11}\} \;\square\; P)]$
end	$)]$

Figure 7.6.9 : *Direct recursion via the μ-operator*

Indirect recursion is modelled by the **procgrp**-construct. It is required that all procedures at the same level are declared in a sequence, enclosed by the **procgrp** and **endgrp** keywords. An example is given in figure 7.6.10.

```
begin
   var x:[0,1];
   procgrp
      P begin ⟨ x'=0 ⟩ ; Q end ,
      Q begin ⟨ x'=1 ⟩ ; P end
   endgrp ;
   P || Q
end
```

Figure 7.6.10 : *Using the procgrp-construct to model indirect recursion*

The translation into a Box expression is done inductively. If there is only one declaration in proclist, the usual scheme as shown above is applied. If there are several declarations, the proclist is divided into the head declaration and the tail declarations. The translation of the tail declarations is provided both to the main program and to the body of the head declaration:

$$\begin{aligned} & E(\textbf{procgrp}\ \texttt{P}\ \text{block}\ ,\ \text{proclist}\ \textbf{endgrp}\ \text{scope}) \\ = \ & E(\text{scope})\ E(\text{proclist})\ [P \leftarrow \mu P.(E(\text{block})\ E(\text{proclist}))] \end{aligned}$$

In this way it is ensured that each occurrence of P is either bound by the refinement or by the recursion over the body. The inductive principle extends this property to all declarations in proclist, and the result is independent of the ordering of the declarations. The translation of the example is shown in figure 7.6.11.

$$\begin{array}{l} [\{x, x_{00}, x_{01}, x_{10}, x_{11}\} : \mathbf{V}(x, \{0,1\}) \parallel ((\\ \quad P \parallel Q \\ ); \{x\})] \\ [Q \leftarrow \mu Q.((\{x_{01}\} \ \square\ \{x_{11}\}); P)] \\ [P \leftarrow \mu P.(((\{x_{00}\} \ \square\ \{x_{10}\}); Q)[Q \leftarrow \mu Q.((\{x_{01}\} \ \square\ \{x_{11}\}); P)])] \end{array}$$

Figure 7.6.11 : *Inductive translation of proclist*

7.7 Value and Reference Parameters

Value parameters should be local variables in the body of the procedure. The arguments for value parameters can differ at each procedure call. As there is a scoping for the local variables in the body, no access to these variables is possible after refinement of the call variable by the body of the procedure. This implies that the value parameters cannot be located inside the body, because then there is no way to pass any arguments to them. On the other hand, for each activation of the procedure, a "new" value parameter is needed, so they cannot be located at the level of the procedure declaration itself. We solve this problem by a compromise: value parameters are located at each procedure call. This is shown in figure 7.7.12.

begin	$[\{x, x_{00}, x_{01}, x_{10}, x_{11}\} : \mathbf{V}(x, \{0,1\}) \parallel (($
var `x:[0,1];`	$[\{a, a_{00}, a_{01}, a_{10}, a_{11}\} : \mathbf{V}(a, \{0,1\}) \parallel (($
proc `P (a:[0,1])`	$(\{a_{01}\} \Box \{a_{11}\});$
begin	P
$\langle$ `x'=a` $\rangle$	$); \{a\})]$
end ;	$); \{x\})]$
`P(1)`	$[P \leftarrow \mu P.(\{a_{00}, x_{00}\} \Box \{a_{00}, x_{10}\} \Box$
end	$\{a_{11}, x_{01}\} \Box \{a_{11}, x_{11}\})]$

Figure 7.7.12 : *Data Boxes for value parameters at each procedure call*

By now it should be obvious, why commutativity of refinement and scoping is essential to this approach. Otherwise, there seems to be no practicable way to pass arguments to value parameters. Unfortunately, this treatment does not yet work in the case of a recursive procedure call with value parameters (figure 7.7.13).

```
begin
   proc P (a:[0,1]) begin
      do ⟨ a=0 ⟩ ; exit □ ⟨ a>0 ⟩ ; P(a-1) ; exit od
   end ;
   P(1)
end
```

Figure 7.7.13 : *A procedure with recursive call and value parameter*

1: $[\{a, a_{00}, a_{01}, a_{10}, a_{11}\} : \mathbf{V}(a, \{0,1\}) \parallel (($
2: $\quad(\{a_{01}\} \Box \{a_{11}\});$
3: $\quad P$
4: $); \{a\})]$
5: $[P \leftarrow \mu P.([\emptyset * \mathbf{stop} * (\{a_{00}\} \Box (\{a_{11}\};$
6: $\qquad[\{a, a_{00}, a_{01}, a_{10}, a_{11}\} : \mathbf{V}(a, \{0,1\}) \parallel (($
7: $\qquad\quad(\{a_{11}, a_{00}\} \Box \{a_{11}, a_{10}\});$
8: $\qquad\quad P$
9: $\qquad); \{a\})]$
10: $))])]$

Figure 7.7.14 : *Wrong translation of recursion with value parameter*

In line 7 of the translation in figure 7.7.14, a name clash occurs with the value parameters a of different recursion levels. This problem can be overcome by a second renaming, specific to the value parameters. The solution is shown in figure 7.7.15.

$$
\begin{array}{l}
[\{\tilde{a}, \tilde{a}_{00}, \tilde{a}_{01}, \tilde{a}_{10}, \tilde{a}_{11}\} : \mathbf{V}(\tilde{a}, \{0,1\}) \parallel ((\\
\quad (\{\tilde{a}_{01}\} \,\Box\, \{\tilde{a}_{11}\});\\
\quad P\,[a \to \tilde{a}, a_{00} \to \tilde{a}_{00}, a_{01} \to \tilde{a}_{01}, a_{10} \to \tilde{a}_{10}, a_{11} \to \tilde{a}_{11}]\\
); \{\tilde{a}\})]\\
[P \leftarrow \mu P.([\emptyset * \mathbf{stop} * (\{a_{00}\} \,\Box\, (\{a_{11}\};\\
\qquad [\{\tilde{a}, \tilde{a}_{00}, \tilde{a}_{01}, \tilde{a}_{10}, \tilde{a}_{11}\} : \mathbf{V}(\tilde{a}, \{0,1\}) \parallel ((\\
\qquad\quad (\{a_{11}, \tilde{a}_{00}\} \,\Box\, \{a_{11}, \tilde{a}_{10}\});\\
\qquad\quad P\,[a \to \tilde{a}, a_{00} \to \tilde{a}_{00}, a_{01} \to \tilde{a}_{01}, a_{10} \to \tilde{a}_{10}, a_{11} \to \tilde{a}_{11}]\\
\qquad); \{\tilde{a}\})]\\
\;))])]
\end{array}
$$

Figure 7.7.15 : *Correct translation of recursion with value parameter*

Action names in the body stay plain (a), and action names for the Data Box and argument passing carry a tilde ($\tilde{a}$). This may not be confused with the conjugation ($\hat{a}$) which can be performed on both of them; a and $\tilde{a}$ are different action names with different conjugates. After refinement, the free action names of the body are equipped with the tilde via an appropriate renaming function and can thereby interact with the Data Box. It turns out that action names of different recursion levels are now distinguished by the tilde, and name clashes are avoided.

If there is a call to a procedure with several value parameters, and different parameters receive their value from the same variable, then all argument passing must occur within a single atomic action. This ensures that all value parameters are initialized with the same value, in case there is a possible change of the passed variable in parallel to the procedure call. However, this generally leads to many more transitions than sequential argument passing.

The handling of reference parameters is technically easier. After refinement, they simply have to be renamed to their passed arguments, as shown in figure 7.7.16.

```
begin
    var x:[0,1];
    proc P (var y:[0,1]) begin
        ⟨ y'=1-'y ⟩
    end ;
    P(x)
end
```

$$
\begin{array}{l}
[\{x, x_{00}, x_{01}, x_{10}, x_{11}\} :\\
\quad \mathbf{V}(x, \{0,1\}) \parallel ((\\
\quad P\,[y \to x,\\
\qquad\quad y_{00} \to x_{00}, y_{01} \to x_{01},\\
\qquad\quad y_{10} \to x_{10}, y_{11} \to x_{11}]\\
); \{x\})]\\
[P \leftarrow \mu P.(\{y_{01}\} \,\Box\, \{y_{10}\})]
\end{array}
$$

Figure 7.7.16 : *Treating reference parameters by renaming after refinement*

7.8 The Towers of Hanoi

The interplay of the different ideas shall finally be illustrated by a larger example. The program in figure 7.8.17 solves deterministically and recursively the well-known problem of the "Towers of Hanoi". We will do a symbolic calculation. The variables x,y and z represent the towers (actually their height). The value parameter n takes the number of disks to be moved from tower a via tower b to tower c, which are reference parameters of the Hanoi-procedure H. Initially, x carries all five disks and y and z are empty.

```
begin
   var x:[0,5] init 5;
   var y:[0,5] init 0;
   var z:[0,5] init 0;
   proc H (n:[1,5], var a:[0,5], var b:[0,5], var c:[0,5])
   begin
      do ⟨ n=1 ⟩ ; ⟨ a'='a-1 ⟩ ; ⟨ c'='c+1 ⟩ ; exit
       □ ⟨ n>1 ⟩ ; H(n-1,a,c,b) ; H(1,a,b,c) ; H(n-1,b,a,c) ; exit
      od
    end ;
    H(5,x,y,z)
 end
```

Figure 7.8.17 : *Program for the "Towers of Hanoi"*

$$
\begin{array}{l}
[\{x, x_{00}, \ldots, x_{55}\} : \mathbf{V}(x, \{0,5\}, 5) \parallel ((\\
[\{y, y_{00}, \ldots, y_{55}\} : \mathbf{V}(y, \{0,5\}, 0) \parallel ((\\
[\{z, z_{00}, \ldots, z_{55}\} : \mathbf{V}(z, \{0,5\}, 0) \parallel ((\\
\quad [\{\tilde{n}, \tilde{n}_{11}, \ldots, \tilde{n}_{55}\} : \mathbf{V}(\tilde{n}, \{1,5\}) \parallel ((((\{\tilde{n}_{15}\} \ \Box \ \ldots \ \Box \ \{\tilde{n}_{55}\});\\
\qquad H\,[n \to \tilde{n}, \ldots, a \to x, \ldots, b \to y, \ldots, c \to z, \ldots]\\
\quad); \{\tilde{n}\})]\\
); \{z\})]); \{y\})]); \{x\})]\\
[H \leftarrow \mu H.([\emptyset * \mathbf{stop} * (\\
\quad (\{n_{11}; (\{a_{10}\} \ \Box \ \ldots \ \Box \ \{a_{54}\}); (\{c_{01}\} \ \Box \ \ldots \ \Box \ \{c_{45}\})) \ \Box\\
\quad ((\{n_{22}\} \ \Box \ \ldots \ \Box \ \{n_{55}\}); (\\
\quad [\{\tilde{n}, \tilde{n}_{11}, \ldots, \tilde{n}_{55}\} : \mathbf{V}(\tilde{n}, \{1,5\}) \parallel ((((\{n_{11}, \tilde{n}_{00}\} \ \Box \ \ldots \ \Box \ \{n_{55}, \tilde{n}_{44}\});\\
\qquad H\,[n \to \tilde{n}, \ldots, a \to a, \ldots, b \to c, \ldots, c \to b \ldots]\\
\quad); \{\tilde{n}\})];\\
\quad [\{\tilde{n}, \tilde{n}_{11}, \ldots, \tilde{n}_{55}\} : \mathbf{V}(\tilde{n}, \{1,5\}) \parallel ((((\{\tilde{n}_{11}\} \ \Box \ \ldots \ \Box \ \{\tilde{n}_{51}\});\\
\qquad H\,[n \to \tilde{n}, \ldots, a \to a, \ldots, b \to b, \ldots, c \to c \ldots]\\
\quad); \{\tilde{n}\})];\\
\quad [\{\tilde{n}, \tilde{n}_{11}, \ldots, \tilde{n}_{55}\} : \mathbf{V}(\tilde{n}, \{1,5\}) \parallel ((((\{n_{11}, \tilde{n}_{00}\} \ \Box \ \ldots \ \Box \ \{n_{55}, \tilde{n}_{44}\});\\
\qquad H\,[n \to \tilde{n}, \ldots, a \to b, \ldots, b \to a, \ldots, c \to c \ldots]\\
\quad); \{\tilde{n}\})]))\\
)])]
\end{array}
$$

Figure 7.8.18 : *Translation of the "Hanoi" program*

A slightly abbreviated translation is given in figure 7.8.18 . The recursive calls take place in the lines 12, 15 and 18. Here, new value parameters n are produced and appropriate renaming functions are applied.

7.9 Conclusion

The ideas underlying a low-level net semantics of B(PN)^2 with procedures have been presented. It has turned out that commutativity of refinement and scoping is crucial for argument passing to value parameters. The corresponding modification can be shown to be consistent with the standard PBC, at least for finite approximations to recursion ([24]). The well-definedness of the latter will be subject of future work. It seems to be necessary to define a canonical injection (*embedding*) from one approximation to the next, while by now different embeddings are still possible. Practical applications require a finite representation of the program semantics, and we will now give some thoughts how to achieve this. Informally, an execution of a program can be

Termination	the final marking is reached.
Deadlock	the final marking is not reached but all transitions are dead.
Divergence	the final marking is not reached and there are always live transitions.

Considering only B(PN)^2 programs with finite data, we define the *static state space* of the program to be the cartesian product of the states of all variables and channels, counting only the occurrences of their *declarations* and excluding reference parameter declarations (since they are not associated to additional Data Boxes). It turns out that the *actual* state space with all local data and value parameters depends on the number of different activations of a procedure (by parallel execution or recursion). Nevertheless, we suggest the following

Conjecture:

If there is no divergent execution of a B(PN)^2 program with finite *static state space*, then the cardinality of the *static state space* is an upper bound for the depth of recursion in a finite net approximation, necessary to reflect all possible behaviour of the program.

This idea is motivated by the analogy to the language of a finite automaton, but it is not quite the same, since in our case there is an interdependenc of the recursion depth and the *actual* state space. However, in our framework there is no notion of "pointers", and thereby at each moment, only the *last* activation of a procedure is "visible". Maybe this can still be affected by reference parameters.

On the other hand, this upper bound would be of limited practical use: for the Hanoi program it calculates to $6^3 \cdot 5 = 1080$, while obviously a recursion depth of 5 is sufficient in this case. Certainly, there are also programs that require this large upper bound.

7.10 Appendix

7.10.1 Syntax of B(PN)^2

Figure 7.10.19 shows the syntax of B(PN)^2. Syntactic metavariables are printed in normal font, keywords are printed **bold**, numeric variables are printed *italic* and identifiers in `teletype`.

program	::=	block	
block	::=	**begin** scope **end**	
scope	::=	com | decl; scope	
decl	::=	**var** `var-name` : type | decl; decl	
type	::=	set | set **init** const | **chan** k **of** set	($k \in \mathbb{N}_0 \cup \{\omega\}$)
set	::=	`[`a, b`]`	($a, b \in \mathbb{Z}$, $a \leq b \ \vee\ (a, b) = (\mathbf{false}, \mathbf{true})$)
com	::=	$\langle$ expr $\rangle$ | com || com | com ; com | **do** alt-set **od** | block	
alt-set	::=	com; **repeat** | com; **exit** | alt-set $\Box$ alt-set	
expr	::=	`'v` | `v'` | `v` | `c!` | `c?` | const | expr op expr | op expr | `(`expr`)`	
op	::=	`+` | `-` | `*` | `/` | `%` | `=` | $\neq$ | $<$ | $>$ | $\leq$ | $\geq$ | $\wedge$ | $\vee$ | $\neg$	
const	::=	**false** | **true** | z	($z \in \mathbb{Z}$)

Figure 7.10.19 : *Syntax of B(PN)²*

7.10.2 Syntax and Semantics of PBC

Basic PBC expressions are

b	::=	α |	multi-action
		stop |	deadlock
		$\mathbf{V}(n, w)$ |	declaration of variable n, domain w
		$\mathbf{V}(n, w, i)$ |	declaration of variable n, domain w, initialized i
		$\mathbf{C}(n, k, w)$ |	declaration of channel n, capacity k, domain w

The syntax of PBC expressions E is

E	::=	**b** |	basic expression
		$E[f]$ |	relabelling
		$[a : E]$ |	scoping ($a \in A$)
		$E \parallel E$ |	parallel operator
		$E; E$ |	sequence
		$E \ \Box\ E$ |	choice
		$[E * E * E]$ |	iteration
		X |	expression-variable
		$E[X \leftarrow E]$ |	refinement
		$\mu X.E$ |	recursion

The following axioms serve to reduction of hiearchical constructs into elementary ones.

(1) $\mathbf{b}[X \leftarrow E] \equiv \mathbf{b}$

(2) $(E_1[f])[X \leftarrow E] \equiv (E_1[X \leftarrow E])[f]$

(3) $[a : E_1][X \leftarrow E] \equiv [a : E_1[X \leftarrow E]]$

(4) $(E_1 \parallel E_2)[X \leftarrow E] \equiv (E_1[X \leftarrow E]) \parallel (E_2[X \leftarrow E])$

(5) $(E_1; E_2)[X \leftarrow E] \equiv (E_1[X \leftarrow E]); (E_2[X \leftarrow E])$

(6) $(E_1 \ \Box\ E_2)[X \leftarrow E] \equiv (E_1[X \leftarrow E]) \ \Box\ (E_2[X \leftarrow E])$

(7) $[E_1 * E_2 * E_3][X \leftarrow E] \equiv [E_1[X \leftarrow E] * E_2[X \leftarrow E] * E_3[X \leftarrow E]]$

(8) $X[X \leftarrow E] \equiv E$

(9) $Y[X \leftarrow E] \equiv Y$

(10) $E_1[X \leftarrow E_2][X \leftarrow E_3] \equiv E_1[X \leftarrow (E_2[X \leftarrow E_3])]$

(11) $E_1[X \leftarrow E_2][Y \leftarrow E_3] \equiv E_1[X \leftarrow (E_2[Y \leftarrow E_3])][Y \leftarrow E_3]$

(12) $(\mu X.E_1)[X \leftarrow E] \equiv \mu X.E_1$

(13) $(\mu X.E_1)[Y \leftarrow E] \equiv \mu X.(E_1[Y \leftarrow E])$ (if $X \notin \mathit{free}(E)$)

(14) $(\mu X.E_1) \equiv E_1$ (if $X \notin \mathit{free}(E_1)$)

Application of these axioms yield an equivalent *reduced* expression without any refinement operations. Every recursion-free expression E has then a Box semantics $box(E)$. For every $i \geq 0$, the box_i-semantics of a reduced expression E is defined by

$$\begin{aligned}
box_i(\mathbf{b}) &= box(\mathbf{b})\\
box_i(E[f]) &= box_i(E)[f]\\
box_i([a:E]) &= [a: box_i(E)]\\
box_i(E_1 \parallel E_2) &= box_i(E_1) \parallel box_i(E_2)\\
box_i(E_1; E_2) &= box_i(E_1); box_i(E_2)\\
box_i(E_1 \,\square\, E_2) &= box_i(E_1) \,\square\, box_i(E_2)\\
box_i([E_1 * E_2 * E_3]) &= [box_i(E_1) * box_i(E_2) * box_i(E_3)]\\
box_i(\mu X.E) &= box_i(E[X \leftarrow E]^i[X \leftarrow \mathbf{stop}]), \text{if } rec(E) > 0\\
box_i(\mu X.E) &= box(E[X \leftarrow E]^i[X \leftarrow \mathbf{stop}]), \text{ if } rec(E) = 0
\end{aligned}$$

where $rec(E)$ is the number of recursion constructs in E and $[X \leftarrow E]^i$ denotes the succesive refinement

$$\begin{aligned}
&[X \leftarrow E]^0 = [\,] && \text{(empty refinement)}\\
&[X \leftarrow E]^i = [X \leftarrow E]^{i-1}[X \leftarrow E]
\end{aligned}$$

In [24] it is shown that there exists an injection from $box_i(E)$ to $box_{i+1}(E)$, which could probably be eploited to define recursion as the least upper bound of an appropriate chain.

7.10.3 PBC Semantics of B(PN)^2

The semantic function E is for all B(PN)^2 programs defined by

$$\begin{aligned}
&E(\mathbf{begin}\ \text{scope}\ \mathbf{end}) && = && E(\text{scope})\\
&E(\text{decl; scope}) && = && [S(\text{decl}): D(\text{decl}) \parallel (E(\text{scope}); T(\text{decl}))]\\
&E(\langle\text{expr}\rangle) && = && (\ \square\ \{\alpha | (\alpha, \mathbf{true}) \in F(\text{expr})\})\\
&E(\text{com}_1 \parallel \text{com}_2) && = && (E(\text{com}_1) \parallel E(\text{com}_2))\\
&E(\text{com}_1; \text{com}_2) && = && (E(\text{com}_1)\,;E(\text{com}_2))\\
&E(\mathbf{do}\ \text{alt-set}\ \mathbf{od}) && = && [\emptyset * R(\text{alt-set}) * X(\text{alt-set})]
\end{aligned}$$

The auxiliary function S determines the appropriate scope actions for a declaration, D provides the Data Box and T yields the termination action that empties the Data Box. The auxiliary function F calculates the corresponding multi-actions for B(PN)^2 atomic actions:

$$\begin{aligned}
&F(\texttt{'v}) && = && \{(\{v_{xy}\}, x) | x, y \in val(\texttt{v})\}\\
&F(\texttt{v'}) && = && \{(\{v_{xy}\}, y) | x, y \in val(\texttt{v})\}\\
&F(\texttt{v}) && = && \{(\{v_{xx}\}, x) | x \in val(\texttt{v})\}\\
&F(\texttt{c!}) && = && \{(\{c_{!x}\}, x) | x \in val(\texttt{c})\}\\
&F(\texttt{c?}) && = && \{(\{c_{?x}\}, x) | x \in val(\texttt{c})\}\\
&F(\text{const}) && = && \{(\emptyset, \text{const})\}\\
&F(\text{expr}_1\ \text{op}\ \text{expr}_2) && = && \{(\alpha_1 \cup \alpha_2, x_1\ \text{op}\ x_2) | (\alpha_i, x_i) \in F(\text{expr}_i), i = 1, 2\}\\
&F(\text{op expr}) && = && \{(\alpha, \text{op}\ x) | (\alpha, \text{op}\ x) \in F(\text{expr})\}
\end{aligned}$$

R and X divide the body of iteration into repeat- and exit-branches:

$$\begin{aligned}
&R(\text{com; }\mathbf{repeat}) = X(\text{com; }\mathbf{exit}) && = && E(\text{com})\\
&R(\text{com; }\mathbf{exit}) = X(\text{com; }\mathbf{repeat}) && = && \mathbf{stop}\\
&R(\text{alt-set}_1\ \square\ \text{alt-set}_2) && = && (R(\text{alt-set}_1)\ \square\ R(\text{alt-set}_2))\\
&X(\text{alt-set}_1\ \square\ \text{alt-set}_2) && = && (X(\text{alt-set}_1)\ \square\ X(\text{alt-set}_2))
\end{aligned}$$

A formal extension of the semantic function E for procedure declarations and calls is given in [24].

8 Design and Implementation of a C-Code Generator for B(PN)^2

Stephan Melzer
Institut für Informatik
Universität Hildesheim
e-mail:melzer@informatik.uni-hildesheim.de

Abstract

In this paper the basic ideas underlying the implementation of a Code Generator for B(PN)^2 (**B**asic **P**etri **N**et **P**rogramming **N**otation), i.e. a translator of B(PN)^2 into C is presented. The core syntax of the source language is extended by C-routine calls (i.e user defined procedures or system calls) as atomic commands. The target language is represented in terms of the ANSI C Standard with its operating system routines belonging to the POSIX Standard. The source language is considered to be used on a multiprocessor UNIX[5] platform. The major difficulties of the translation are the following:

- atomic commands (shared-memory-segments with kernel semaphores)
- guarded commands (removal of bounded nondeterminism)
- the parallel operator (distribution to UNIX-processes)
- the synchronisation of the processes through channels with capacity 0.

8.1 Introduction

In the framework of the DFG supported project PEP (**P**rogramming **E**nviroment based on **P**etri Nets) an analysing tool for verifying parallel programs written in B(PN)^2 is developed. The theoretical aspect of B(PN)^2 with its semantics in terms of PBC (**P**etri **B**ox **C**alculus) was one of the aims of the projects CALIBAN (Causal Calculi Based on Nets) and DEMON (Design Methods Based on Nets). Its practical aspects could only be investigated in the net editor of the environment, e.g. simulation of the transition sequences in Petri nets. The extension of the system, i.e. the development of a code generator of B(PN)^2 to ANSI C is intended to provide the possibility of its practical usage. Practical programming languages without the facility of being verified can be translated into B(PN)^2 in order to check their properties. This paper is a starting point of the programming of applications written in B(PN)^2. We have to ensure that the generated code has to have a high degree of portability for its multiple usage on operating system platforms. With respect to the theoretical background of B(PN)^2 we propose to generate a runtime protocol of fired transitions of the executed atomic commands in order to get a triggered input for the existing net editor. The above-mentioned idea provides PEP with a debugging possibility. The essential considerations concerning the implementation of a prototype are represented in the following sections:

Section 2 introduces the syntax of the source language. Section 3 describes the behavioural properties of the code generator. Section 4 and 5 contain the translation of B(PN)^2-elements

[5]UNIX is a registered Trade Mark of UNIX System Laboratories

into C. In this respect several examples are given and some explanatory remarks are included to proceed. Section 6 touches upon the runtime behaviour of the generated code. Section 7 contains some concluding remarks and an outlook. This work is still in progress. That explains the fact that some problems have been left open here.

8.2 The Source Language

In [10] the basis of the general syntax of $B(PN)^2$ has been developed. There the semantics of $B(PN)^2$ is expressed in terms of PBC [6]. On the one hand we extend the syntax by some practical features, on the other hand we restrict the expression in the atomic commands. The loop construct is enlarged by an initial command as it is also proposed in [33]. The semantics of the loop has the following structure[6]:

$$E(\textbf{do}\ \texttt{init}\ \textbf{enter}:\texttt{alt-set}\ \textbf{od}) = [E(\texttt{init}) * R(\texttt{alt}-\texttt{set}) * X(\texttt{alt}-\texttt{set})]$$

The reason for this assumption is mentioned in section 8.5 in the context of the removal of the nondeterminism of the guarded commands. It may be observed, however, that $B(PN)^2$ as yet lacks a procedure concept. That leads us to the more precise design of the code generator for the embedding of such a concept in future. We should also stress the fact that the input/output concept (e.g. usage of channels as streams) is missing in $B(PN)^2$. An introduction of an import interface for C-routines in $B(PN)^2$ can be used as an alternative solution of the above-mentioned problems. In comparison to the preprocessor of C the include-files are specified in the following way:

```
IMPORT "math_io.h"
IMPORT "system.h"
```

We demand a strict seperation of $B(PN)^2$ and C in order to obtain a well defined type checking. That leads us to an explicit declaration of the prototypes of the used C-routines (i.e. access and general data type). E.g.

```
IMPORT "math_io.h"
print_int(const int);
get_int(var int);
...
```

The strict function prototype requires differentiation between variable and constant expressions in its actual parameters of the routine calls, in order to obtain a syntactical difference between $B(PN)^2$ and C (especially for a procedure concept of $B(PN)^2$, which is still under work [24]):

- variable expressions e.g. y', x' (without any composition)
- constant expressions e.g. $'y$, $'x$ or $'x + 2 * \ 'y$

In this context the expression x is neither a variable expression nor a constant expression, because it could be confused with a call by reference of a $B(PN)^2$ procedure. E.g.:

[6]The function E associates to every $B(PN)^2$ program a Box expression, R extracts the reapeat component , X - the exit component

```
...
begin
 var c,x:int init 4711;
 print_int('c + 'x);
 get_int (x')            /* incorrect :  get_int (x)*/    ...
```

The way the atomic commands are introduced in [10] leads us to an inefficient implementation (e.g. expressions such as $\langle (x')^3 - 17(x')^2 + 6x' - 11 = 0 \rangle$ are evidently hard to execute), so that we have to restrict the power of atomic commands in order to get executable expressions. We make the following assumption:

> Variables can only appear on one side of an equation. These variable expressions are treated as assignments. The assignments can be linked by logical AND-operators, if their variables are pairwise different (in order to avoid unbounded nondeterminism, e.g. $\langle x' = 7 \vee x' = 8 \vee \ldots \rangle$).

We must ensure that these assignments can also be combined on top level with constant boolean expressions without any restriction. That leads us to deterministic atomic commands (e.g. $\langle x' = 7 \wedge (c? < 6 \vee {}'x \text{ DIV } 7 = 0) \rangle$).

However, this restriction can be accepted in practical usage, since common programming notations are based on assignments.

8.3 Some Features of the Code Generator

The code generator is a two-pass compiler that produces ANSI C-Code. The latter is compiled by an ANSI C-compiler to executable code. In the first pass the B(PN)^2 program is syntactically and semantically analyzed, the parse tree and symbol table is produced. The second pass generates C-Code along the parse tree (i.e. syntax-driven translation) where the boolean expression of the atomic commands are exported in a separate file. Each pass is implemented in a single module in order to obtain a high degree of portability (e.g. to Par-C or to assembler). Due to the restriction of the target language to ANSI C (ISO/IEC 9899) the compiler of the last pass can be exchanged by an arbitrary ANSI C-compiler. The used operating system routines are taken from the POSIX Standard (**P**ortable **O**perating **S**ystem **I**nterface, IEEE 1003.1 ISO/IEC 9945-1). The shared memory management and the required kernel semaphores belong to the prestandard (POSIX 1003.4 Draft 13) [27]. The obtained flexibility makes it possible for us to use multiprocessor platforms with a multiprocessor supported kernel.

As far as the chief practical application is concerned, there is a need for an error handling that can also be found in common compilers.

A syntactical mistake in a set definition of a variable declaration, for example, is registered as a single error. In case of the wrong set definition the variable is assumed to be an integer, in order to avoid forthcoming related errors (compare with common C-compilers).

The above-mentioned type checking can be separated into two modes, that can be optionally excluded. The first mode checks the type correctness of expressions in atomic commands. If the mistakes are neglected, the semantic of the atomic commands is undefined. By contrast, the paper [10] interprets the above-mentioned mistake as an explicit deadlock. The second mode checks the correctness of C-routine calls. In order to obtain the flexibility (i.e. incompatible parameter types of procedures) we can easily neglect the mistakes. This is possible by means of internal type casting.

8.4 $B(PN)^2$ Data in Shared Memory

Following the principal ideas of $B(PN)^2$ the most preferable IPC (**I**nter **P**rocess **C**ommunication) is the shared memory concept, it is the quickest and the most comfortable way [35] to make data available for many processes. However, the own memory management has to be involved in the generated code, because the shared memory can only be allocated statically. The mutually excluded access is realized by a warning/locking method (deadlock-free) to applicate a semaphore to each variable, in order to have a high degree of concurrency.
A useful side effect of the process creation in UNIX systems is the inheritance of all open IPC descriptors.

8.5 $B(PN)^2$ Operators translated into C

In comparison with control structures (loop, parallel construct and atomic command) which require explicit emulation, being unknown in C, the remaining control structure of $B(PN)^2$ (sequence and block) can be easily translated. To make it more clear we introduce an informal function T for the translation of $B(PN)^2$ constructs into C, which associates a $B(PN)^2$ construct with a pseudo-C-program. Using this abbreviation the translation of sequence and block are the following:

Sequence:	T(com_1; com_2)	=	T(com_1); T(com_2)
Block:	T(decl; com)	=	{memory_allocation; variable_declaration; T(com); }

The identifiers of the used variables are registered in a block-nested symbol table where each variable gets a new unique identifier.
In comparison with C the conception of $B(PN)^2$ makes it possible to declare procedures after each block entry. In case of extended[7] implementation this incompatibility requires unique identifiers.
The remaining structures are discussed in detail in the following subsections.

8.5.1 Atomic Command

Here the reader immediately recognizes the export of the boolean expression of the atomic command and the mutual excluded access to its variables in the shared memory.

```
T(⟨bool-expression⟩) =
do {
   wait (variable_access);
   get_pre_primed_variables (pre_container);
   get_post_primed_variables (post_container);
   if (access_is_ok ∧ bool_expr [8](pre_container, post_container))
      then rewrite_copies (pre_container, post_container);
           signal (variable_access);
```

[7] by a procedure concept

```
8            exit
9        else signal (variable_access);
10           sleep()
11    fi
12 }
13 od
```

In lines 2, 7, 9 the access to the variable segments by the warning/locking protocol is obvious. The lines 3 and 4 show the process of copying the pre-values and the preparing of the post-values for its safe manipulation (chiefly the channels) in case of deactivation of the corresponding transition in the Petri net. In line 5 the activation of the transition is proved; in case of a handshake with another process the boolean function invokes the synchronisation process. In case of an active transition the manipulated variables are updated and the transition label is written in the protocol. In the other case the process is deactivated for a fixed period of time.

8.5.2 Parallel Construct

Identifying parallel components in B(PN)^2 with real processes in UNIX, we have to choose between *heavy weight processes* and *threads*. Threads are more comfortable as far as synchronisation is concerned, but their handling and behaviour is not yet universally standardized (POSIX.4a). The behaviour of heavy weight processes is standardized according to POSIX.1, so that we decide to get more portability instead of getting more suitable identification of the parallel components. We can obviously face the problem if we use the fork command in order to create a child process for each component. However, the sequence of creating is not fair, so that we have to generate a random fork-sequence.

This random choice of the sequence of creating two processes out of the constituents of the parallel operator does not consider into the additionally combined constituents (e.g. $a||b||c$ is executed as $a||(b||c)$ and not as $(a||b)||c$):

```
T(com_1||com_2) =
{
fork ();
if (is_mother_procees)
  then fork ();
      if (is_mother_process)
        then while (¬ (first_ child_terminated ∧
                   second_child_terminated) do
             sleep();
        else T (com_1);
             first_child_terminated := TRUE;
             exit();
      fi
```

[8] imported

```
12     else T(com_2);
13          second_child_terminated := TRUE;
14          exit();
15  fi
16  }
```

In this pseudo-code program the nondeterministic choice is not shown, in order to keep our attention on the main point. In line 6 and 7 the rendezvous with its components is fulfilled by a boolean flag, indicating their termination, instead of an IPC signal (i.e. SIGCHILD).

8.5.3 Loop Construct

The nondeterministic choice of one of the active alternatives is one of the most difficult parts of the translation. There is a solution for the general alternative construct in CSP [19], but the assumption (i.e. synchronisation primitives) does not fit in our environment.

The set of active alternatives refers to the corresponding first atomic commands of each alternative.

If we have a nested block structure, the first atomic command of the deepest block is used; for the loop construct the initial command has to be required, in oder to get a unique first command. If we have a parallel construct at the beginning of an alternative, each first command of all operands has to be checked. The following example shows the set of the first commands (which are underlined):

```
do init_com enter :
com_11 || com_12 || begin
                        com_13||com_14
                     end;repeat
[] com_2; exit
[] begin
     begin
       do com_31 enter:
         com_32;exit
       od
     end;
     com_33
   end;exit
od
```

All first commands of the alternatives have to be proved according to their activation. C-routine calls that are first atomic commands are always active by definition. It should be noticed that pre- and post-values for each active command have to be stored in order to avoid unnecessary calculation of the random choosen alternative. We do not have the intention to go here in further details. See for details [29]

8.5.4 Synchronisation by Channels with Capacity 0

Here the main problem is the merge of distributed atomic commands (in different processes) to one atomic command (i.e. one transition). This example shows the complexity of channel-0's [9]:

$$\underbrace{\langle a! = 0\rangle}_{P1} \,||\, \underbrace{\langle b? = 0\rangle}_{P2} \,||\, \underbrace{\langle a? = b!\rangle}_{P3} \,||\, \underbrace{\langle a? = b!\rangle}_{P4} \,||\, \underbrace{\langle a! = 1\rangle}_{P5} \,||\, \underbrace{\langle b? = 1\rangle}_{P6}$$

In this case the valid synchronisations are:

$$\begin{array}{ll} P1\ P3\ P2 & \text{or} \\ P1\ P4\ P2 & \text{or} \\ P5\ P3\ P6 & \text{or} \\ P5\ P4\ P6, & \end{array}$$

in comparison to an invaild synchronisation:

$$\begin{array}{ll} P1\ P3\ P6 & \text{or} \\ P1\ P4\ P6, & \ldots \end{array}$$

In contrast to the Box Calculus [6] the synchronisation partners have to be found at runtime. Therefore all the interleavings of the participants with channel-0 have to be calculated. They are characterized by their input/output-behaviour.

E.g.:

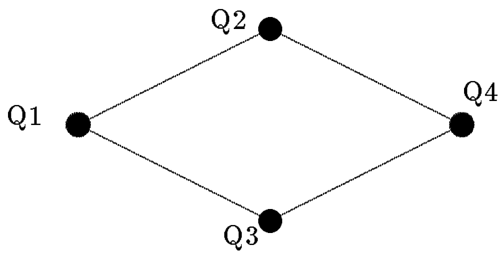
$P1$	OUTPUT: a (value 0)	INPUT: no input
$P5$	OUTPUT: a (value 1)	INPUT: no input
$P3$, $P4$	OUTPUT: b (**unknown**)	INPUT: a
$P2$, $P6$	OUTPUT: no output	INPUT: b

We can easily see that the valid interleaving cannot be described statically [10], but all possible handshakes must be tested according to their common boolean expression.

Fortunately, the static synchronisation can be described as a spanning tree (i.e. it is acyclic) [6]. For example the following program leads to a deadlock (compare [14, 13]):

$$\underbrace{\langle a! = 0 \wedge b! = 1\rangle}_{Q1} \,||\, \underbrace{\langle a? = 0 \wedge c! = a?\rangle}_{Q2} \,||\, \underbrace{\langle b? = 1 \wedge d! = b?\rangle}_{Q3} \,||\, \underbrace{\langle c? = 0 \wedge d? = 1\rangle}_{Q4}$$

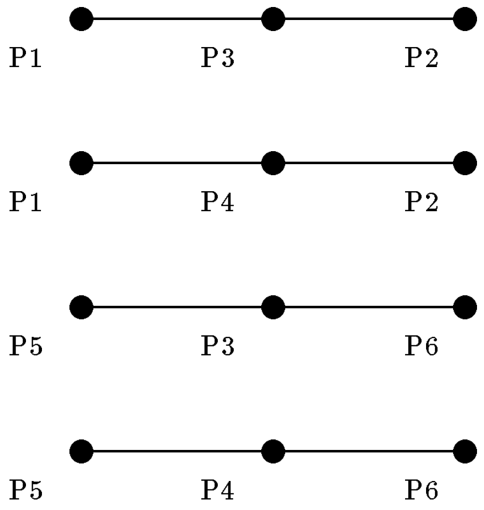
with the spanning graph



[9] abbreviation for channels with capacity 0

[10] in our example: the unknown output of $P3$, $P4$

In constrast our first sample possess the following spanning graphs



These graphs can be expressed by a term $m(P)$, defining all static [11] synchronisations concerning the process P.
E.g.:

$$
\begin{aligned}
m(P3) &= (P1|P5)P3(P2|P6) \\
&= \{P1P3P2, \ldots, P5P3P6\} \\
m(P5) &= P5((P3;(P2|P1))|(P4,(P2|P1)) \\
&= \{P5P3P2, \ldots, P5P4P1\}
\end{aligned}
$$

The acyclic synchronisation graph allows us to use a recursive algorithm in order to calculate $m(P)$. All interleavings of a process P expressed in $m(P)$, are checked related to their common boolean expression.
One valid interleaving is choosen by a random generator to be executed.

8.6 Runtime system

The runtime system concerning the protocolled transition-labels is briefly presented. In contemporary studies of PEP the behaviour of Petri nets can be checked by a model checker or investigated in the net editor. In order to bridge the runtime behaviour of a B(PN)^2 program with its Petri net, so that we get a powerful debugging tool, we have to generate sequences of extended labels of the fired transitions. These sequences can easily be used as X-events that are considered to be input for the net editor. The extended labels are filed in a protocol in the following way :

process xxx : s_0_1 m?0,

i.e. in the process with process-Id xxx the value of s is changed from 0 to 1 and the value 0 is taken from the channel m. This protocol by itself can invoke X-events in case of a modified event-handler of the net editor.

[11] also the invalid one's

8.7 Conclusion

The concepts which are presented in this paper have for the most part been tested in an actual implementation.

The work described in this paper should be taken as an attempt for the design of a code generator. However, B(PN)^2 is still essentially a low-level programming language and requires the implementation of a procedure concept. Furthermore an optimization of the generated code is planed.

9 Language Preserving Reductions of Safe Petri Nets

Anja Gronewold
Fachbereich Informatik
Universität Oldenburg
PF 2503
D - 26111 Oldenburg
e-mail: gronewold@informatik.uni-oldenburg.de

Abstract

Many decision algorithms for Petri nets are known to have exponential complexity, even if restricted to safe nets. In some cases, this high complexity is caused by the computation of the net's reachability graph. But some decision problems for Petri nets can also be solved by analyzing the net language.

A new method for the computation of (a regular expression for) the language of a safe Petri net is presented. This method is based on net reductions as introduced by Berthelot, and it uses the net's trace language to rebuild the possible interleavings which can get lost during the reduction process. A class of nets whose transitions are labelled by regular expressions is defined, and for this class of nets, a set of reduction rules preserving the net language is proposed. Using the rules, a net is reduced as far as possible, resulting either in a single transition (labelled with a regular expression describing the net language) or in a net which is not further reducible. For this (smaller) net, a regular expression can be computed in the usual way.

The set of reduction rules is being implemented as a tool within the **PEP**-System. While the reduction rules are applied to nets belonging to the class of transition-labelled nets (which can be created and edited in the tool's net editor), it is also possible to include Petri Boxes and to apply the rules to these special safe nets that have been created by translation of a $B(PN)^2$- or PBC-program.

9.1 Introduction

One of the benefits of using Petri nets for the specification of concurrent systems is given by the rich variety of analysis methods they offer, as e.g. place and transition invariants, coverability trees or temporal logics for Petri Nets. The essential method for the analysis of nets within the **PEP**-System is model checking by the algorithm of Esparza [17], but also other methods are considered. Many analyzing methods have exponential complexity, which is often caused by the computation of the reachability graph. For example, also for safe nets, the decision of the reachability problem is NP-hard.

But many properties of a system's Petri net model may also be checked by looking at the net language, i.e. the set of all possible transition sequences of the net. The computation of (a regular expression representing) the language of a bounded net classically involves the construction of the reachability graph which is then interpreted as a finite automaton and from which the regular expression is computed. It is well-known that the size of the reachability graph may be exponential in the size of the net - even for safe nets.

We will introduce a new method for the computation of the net language of safe nets. Our approach is based on Berthelot's work on reductions of Petri nets [1], [2]. His method consists of a set of rules which, applied to a net, yields a smaller one while preserving the properties of liveness and boundedness. In this way, a net may be stepwise reduced to a trivial one, for which liveness and boundedness can be decided immediately. The reduction rules we propose will in addition preserve the net language. This is achieved by labelling the transitions of the reduced net with regular expressions over the transitions of the original net. So, if a net is reducible to a single transition, its label can be used to decide certain properties of the start net. Moreover, the net language can be computed by stepping backwards through the nets generated by the reduction process, in each step substituting transition names in the regular expression by the respective labels and applying the independency relation of the net.

This method can be seen as an additional approach to solve decision problems for Petri nets without constructing the reachability graph, as it is also done in the model checking algorithm of Esparza [17] which is based on (an initial segment of) the branching process of a net and in many cases runs in polynomial time even if the reachability graph is exponential in the size of the net. Up to now, we did not find a completeness result concerning the reduction rules because - like already for Berthelot - it is very difficult to characterize the class of nets which are reducible by the rules. Existing results of completeness for some reduction rules cannot easily be adopted, because they either include rules which destroy safety of the net (as the rules of Esparza do [16]), or rules that depend on changing the marking during the reduction process (as it is done in the works of Desel [12] and Kovalyov [26]), which changes the net language, too.

Using the reduction rules, liveness and boundedness of the net is preserved. In addition, all the properties which are determined by the net language are preserved and can be checked by looking at the calculated regular expression representing the language. In the worst case, the analysis can have exponential complexity, but we also have got examples where the reduction is very fast and the size of the reachability graph is exponential in the size of the net.

We are giving here only an informal overview of our work. All the definitions and detailed information can be found in [21]. In the next section, an example for the net reduction and the language computation is given together with the definitions for the net language and three exemplary reduction rules. Section 3 gives an example for a fast reduction of a net where the size of the reachability graph is exponential in the size of the net, and finally in section 4, we discuss some open problems and directions for further work.

9.2 Reduction of safe nets

In this section, we will introduce our method for language preserving reductions of safe nets. This will be done by presenting reduction rules which transform a source net N into a target net N'. N' will consist of less places and/or less transitions than N. During the reduction process, transition sequences of the net may be joined together and represented by one transition whose possible behaviour is then described by a regular expression over transitions.

If the reduced net is a single transition, its label can be used directly to calculate the net language. In the other case, the reachability graph of the reduced (and hence smaller) net has to be computed.

Before defining the net language and some reduction rules, let us consider the following example (cf. figure 9.2.1 - 9.2.6) to show how the computation of the net language works:

Example: This net satisfies the mutual exclusion of the two critical sections t_{21}, t_{22} and t_{31}, t_{32} where the place p plays the role of a semaphor. The language of the net is $L_{N^0} = PrefL((t_1 t_{21} t_{22} t_{31} t_{32} t_4 \cup t_1 t_{31} t_{32} t_{21} t_{22} t_4)^*)$.

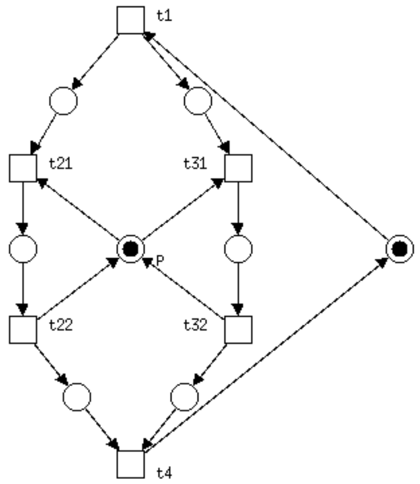


Figure 9.2.1 : $N^0 = (P^0, T^0, F^0, M_0^0)$, $T^0 = \{t_1, t_{21}, t_{22}, t_{31}, t_{32}, t_4\}$, $h_0 = id_{T^0}, I(N^0) = \{(t_1, t_{22}), (t_1, t_{32}), (t_{21}, t_4), (t_{31}, t_4)\}$

Now, look at the following reduction. We observe that each occurrence of t_{21} (resp. t_{31}) is followed by an occurrence of t_{22} (resp. t_{32}). In a first reduction step, we abstract from those single actions and collapse t_{21}, t_{22} (resp. t_{31}, t_{32}) into a single transition u_1 (resp. u_2) which will be labelled by the respective sequence. The resulting net N^1 is shown in figure 9.2.2.

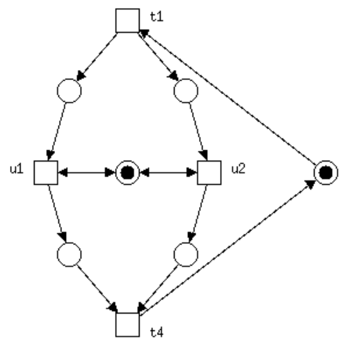


Figure 9.2.2 : $N^1 = (P^1, T^1, F^1, M_0^1), T^1 = \{t_1, t_4, u_1, u_2\}, h_1(u_1) = t_{21}t_{22}, h_1(u_2) = t_{31}t_{32}, h_1(t_i) = t_i, i \in \{1, 4\}, I(N^1) = \emptyset$

The next observation tells us that the semaphore-place does not influence the (string) language of the net N^1. So, in a second reduction step, we take it away, thereby getting the net N^2 (cf. figure 9.2.3).

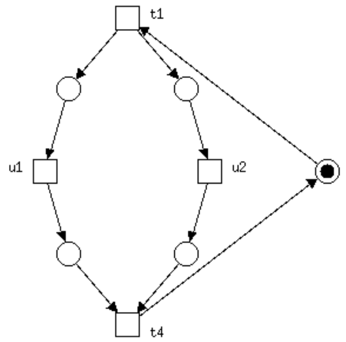


Figure 9.2.3 : $N^2 = (P^2, T^2, F^2, M_0^2), T^2 = T^1, h_2 = id_{T^2}, I(N^2) = \{u_1, u_2\}$

The following reduction steps consist of combining the transitions t_1 and u_1, resulting in a transition u_3 (cf. figure 9.2.4) and combining the transitions u_3 and u_2, resulting in a transition u_4 (cf. bild 9.2.5).

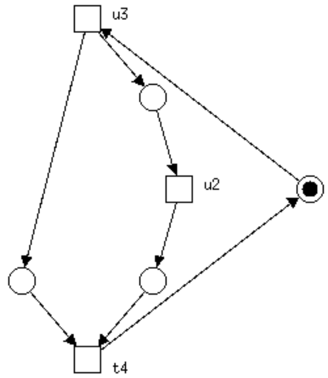


Figure 9.2.4 : $N^3 = (P^3, T^3, F^3, M_0^3), T^3 = \{t_4, u_2, u_3\}, h_3(u_2) = id_{T^3}, h_3(u_3) = t_1 u_1, h_3(t_4) = id_{T^3}, I(N^3) = \emptyset$

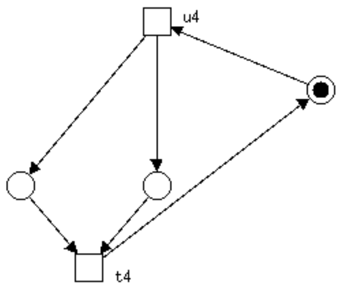


Figure 9.2.5 : $N^4 = (P^4, T^4, F^4, M_0^4), T^4 = \{t_4, u_4\}, h_4(t_4) = id_{T^4}, h_4(u_4) = u_3 u_2, I(N^4) = \emptyset$

Figure 9.2.6 : $N^5 = (P^5, T^5, F^5, M_0^5), T^5 = \{u_5\}, h_5(u_5) = u_4t_4, I(N^5) = \emptyset$

The last reduction step consists of the combination of two steps: in the first step, we contract the transitions u_4 and t_4 to a single one named by u_5 and then – in the second step – we remove the side-condition without affecting the string language of the net (cf. figure 9.2.6).

It is obvious that parallel processes are sequentialized by the reduction algorithm and therefore, we need to consider the independency relation of a net to build the partial commutative closure of the computed regular expressions. (Note that the partial commutative closure is always regular because the language of a safe net is regular.) Unfortunately, it does not suffice to consider the independency relation of the start net, because new independencies can occur during the reduction (cf. figure 9.2.3). These considerations lead to the following definitions:

9.2.1 Definition

(i) Let N be a net and $h : T \to Reg(A)$ a mapping of N's transitions into the regular expressions over the alphabet A. We call $\tilde{N} = (N, h)$ a *labelled net.*

(ii) A *reduction rule* R is a binary relation on the class S of all safe labelled nets N which, for $(\tilde{N}, \tilde{N}') \in R$, satisfies $h' : T' \to Reg(T)$. Let $(\tilde{N}, \tilde{N}') \in R$, then $\tilde{N}$ is called *source net* and $\tilde{N}'$ *target net.* For $(\tilde{N}, \tilde{N}') \in R$, we write $\tilde{N} \to^R \tilde{N}'$ and call $\tilde{N} \to^R \tilde{N}'$ a *reduction step.*

(iii) Let $\Re$ be a finite set of reduction rules. We call a sequence $\alpha \equiv (N_0, h_0) \to^{R_1} (N_1, h_1) \to^{R_2} \cdots \to^{R_{r-1}} (N_{r-1}, h_{r-1}) \to^{R_r} (N_r, h_r)$ of reduction steps, where $0 \le r$ and $R_i \in \Re$ for $1 \le i \le r$, a *derivation of* (N_0, h_0) *of length r in* $\Re$. (N_r, h_r) is called *reduced* in $\Re$, if for no $R \in \Re$, there exists a labelled net (N, h) such that $(N_r, h_r) \to^R (N, h)$.

Next, we define the language of a derivation. A derivation of length 0 is a labelled net. In that case, we have to replace each transition name in the net language by the language of the transition's label, in order to get the language of the derivation. In the inductive step, we take the language of the tail derivation α' and generate the partial commutative closure according to the independency relation of the start net. In the resulting language, we replace each transition name by the language of the transition's label in the start net and build the prefix closure.

9.2.2 Definition

Let $\Re$ be a finite set of reduction rules and α a derivation in $\Re$. The language $L(\alpha)$ of α is inductively defined by:

(1) $\alpha \equiv (N,h):$

$$L(\alpha) = Pref(\bigcup_{w \in L_N} L_h(w)) \quad and$$

(2) $\alpha \equiv (N,h) \rightarrow^R \alpha', where\ R \in \Re\ and\ \alpha'\ is\ a\ derivation\ in\ \Re:$

$$L(\alpha) = Pref(\bigcup_{w \in ST_{I(N)}(L(\alpha'))} L_h(w))$$

Note that the language of the net N is given by the language of the derivation (N, id_T). We will call a reduction rule correct, if the language of the derivation from a source net to the target net corresponds to the language of the source net, i.e. if the rule preserves the language of the source net.

Now, we are able to compute the net language for the above example by applying the definitions:

$$\begin{aligned} L(N^5,h_5) &= Pref(\bigcup_{w \in L_{N^5}} L_{h_5}(w)) \\ &= Pref(\bigcup_{w \in L(u_5)^*} L_{h_5}(w)) \\ &= Pref(\bigcup_{w \in L(u_5)^*} \{h_5(w)\}) \\ &= Pref\ L((u_4t_4)^*) \end{aligned}$$

$I(N^4) = \emptyset \Rightarrow ST_{I(N^4)}L((u_4t_4)^*) = L(u_4t_4)^*.\ Hence$

$$\begin{aligned} L((N^4,h_4) \rightarrow (N^5,h_5)) &= Pref(\bigcup_{w \in ST_{I(N^4)}(L(N^5,h_5))} L_{h_4}(w)) \\ &= Pref(\bigcup_{w \in ST_{I(N^4)}(L(u_4t_4)^*)} L_{h_4}(w)) \\ &= Pref(\bigcup_{w \in Pref\ L(u_4t_4)^*} \{h_4(w)\}) \\ &= Pref\ L((u_3u_2t_4)^*) \end{aligned}$$

$I(N^3) = \emptyset \Rightarrow ST_{I(N^3)}L((u_3u_2t_4)^*) = L(u_3u_2t_4)^*.\ Hence$

$$\begin{aligned} L((N^3,h_3) \rightarrow^* (N^5,h_5)) &= Pref\ L_{h_3}((u_3u_2t_4)^*) \\ &= Pref\ L((t_1u_1u_2t_4)^*) \end{aligned}$$

$I(N^2) = \{(u_1,u_2)\} \Rightarrow ST_{I(N^2)}L((t_1u_1u_2t_4)^*) = L(t_1u_1u_2t_4 \cup t_1u_2u_1t_4)^*.\ Hence$

$$
\begin{aligned}
L((N^2,h_2)\to^*(N^5,h_5)) &= Pref\ L_{h_2}((t_1u_1u_2t_4\cup t_1u_2u_1t_4)^*)\\
&= Pref\ L((t_1u_1u_2t_4\cup t_1u_2u_1t_4)^*)
\end{aligned}
$$

$I(N^1)=\emptyset \Rightarrow ST_{I(N^1)}L((t_1u_1u_2t_4\cup t_1u_2u_1t_4)^*)=L(t_1u_1u_2t_4\cup t_1u_2u_1t_4)^*$. *Hence*

$$
\begin{aligned}
L((N^1,h_1)\to^*(N^5,h_5)) &= Pref\ L_{h_1}((t_1u_1u_2t_4\cup t_1u_2u_1t_4)^*)\\
&= Pref\ L((t_1t_{21}t_{22}t_{31}t_{32}t_4\cup t_1t_{31}t_{32}t_{21}t_{22}t_4)^*)
\end{aligned}
$$

$I(N^0)=\{(t_1,t_{22}),(t_1,t_{32}),(t_{21},t_4),(t_{31},t_4)\}\Rightarrow ST_{I(N^0)}L((t_1t_{21}t_{22}t_{31}t_{32}t_4\cup t_1t_{31}t_{32}t_{21}t_{22}t_4)^*)$
$=L(t_1t_{21}t_{22}t_{31}t_{32}t_4\cup t_1t_{31}t_{32}t_{21}t_{22}t_4)^*$. *Hence*

$$
\begin{aligned}
L((N^0,h_0)\to^*(N^5,h_5)) &= Pref\ L_{h_0}((t_1t_{21}t_{22}t_{31}t_{32}t_4\cup t_1t_{31}t_{32}t_{21}t_{22}t_4)^*)\\
&= Pref\ L((t_1t_{21}t_{22}t_{31}t_{32}t_4\cup t_1t_{31}t_{32}t_{21}t_{22}t_4)^*)
\end{aligned}
$$

9.2.3 The reduction rules

Our set of reduction rules consists of ten rules, from which we will here describe only three examples to give an idea how the reduction algorithm works. The rules are applied to a net as long as possible. The resulting net is either a single transition labelled with a regular expression for the net language, or a net which is not further reducible. For this (smaller) net, a regular expression can be computed in the usual way by constructing the reachability graph. Equally for both cases, the regular expression can be used (without applying the partial commutative closure) to decide properties of the net, and the language of the start net can be computed by replacing the transition names step by step and by building the partial commutative closure.

For a better intuition, we are writing the labels of the transitions into the net.

R1 'Post-Agglomeration' This rule is a pairwise fusion of the pre- and post-transitions of a place p.
Let $\tilde{N}=(N,h)$ and $\tilde{N}'=(N',h')$ two safe labelled nets.
$(\tilde{N},\tilde{N}')\in R1 \Leftrightarrow$

(i) There is a place p whose post-transitions have no other pre-places: ${}^\bullet(p^\bullet)=\{p\}$.
Furthermore, p has the following properties: ${}^\bullet p=\{t_1,\cdots,t_k\}\neq\emptyset; p^\bullet=\{u_1,\cdots,u_n\}$
$\neq\emptyset; p^\bullet\cap{}^\bullet p=\emptyset; M_0(p)=0$.

(ii) For all i,j with $1\le i\le k, 1\le j\le n$, a new transition t_{ij} is introduced in the net, connected in the following way: $T'=(T'\setminus\{t_1,\cdots,t_k,u_1,\cdots,u_n\})\cup\{t_{ij}\}; P'=P\setminus\{p\}$;
$F'=F(t_i\leftarrow t_{ij},u_j\leftarrow t_{ij})|p; M_0'(s)=M_0(s)\ \forall s\in P'$;

$$h'(t) = \begin{cases} t & for\, t \neq t_{ij} \\ t_i u_j & for\, t = t_{ij} \end{cases}$$

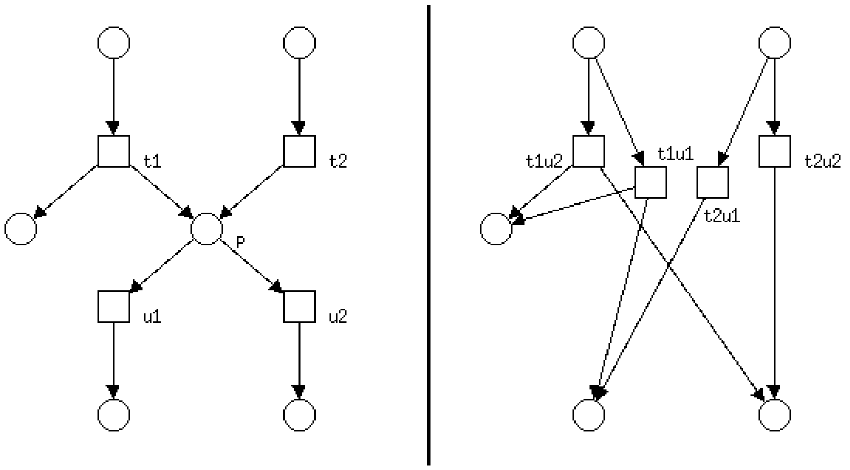


Figure 9.2.7 : *Post-Agglomeration*

R2 'Removal of run-places' Let $\tilde{N} = (N, h)$ and $\tilde{N}' = (N', h')$ two safe labelled nets. $(\tilde{N}, \tilde{N}') \in R2 \Leftrightarrow$

(i) In N, there is a place p such that ${}^\bullet p = p^\bullet$.
(ii) If $M_0(p) = 1$ then remove p:
$T' = T; P' = P \setminus \{p\}; F' = F|p; M_0'(s) = M_0(s) \,\forall\, s \in P'; h'(t) = t \;\forall\, t \in T'$.
If $M_0(p) = 0$ then remove p and all $t' \in {}^\bullet p$:
$T' = T \setminus \{t' | t' \in {}^\bullet p\}; P' = P \setminus \{p\}; F' = F|p, t'; M_0'(s) = M_0(s) \;\forall\, s \in P'$;
$h'(t) = t \;\forall\, t \notin {}^\bullet p$

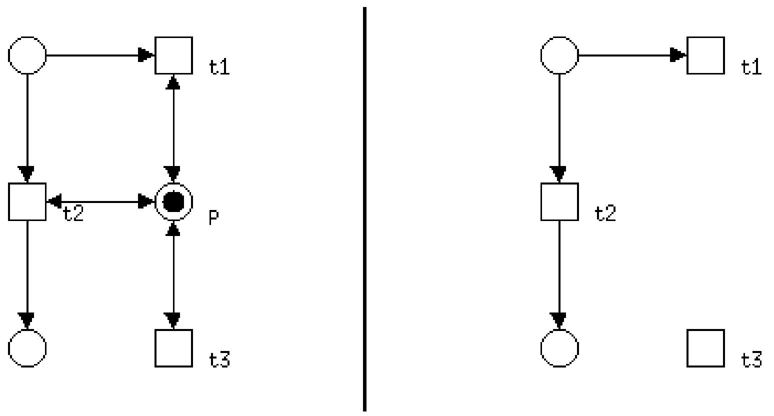


Figure 9.2.8 : *Removal of run-places*

R3 'Removal of parallel transitions' Let $\tilde{N} = (N, h)$ and $\tilde{N}' = (N', h')$ two safe labelled nets.
$(\tilde{N}, \tilde{N}') \in R3 \Leftrightarrow$

(i) In N, there are two parallel transitions t_1 and t_2.
(ii) Let $t_1, t_2 \in T$ two parallel transitions. Then remove t_1:
$T' = T \setminus \{t_1\}; P' = P; F' = F|t_1; M_0'(s) = M_0(s)\ \forall\ s \in P'$;

$$h'(t) = \begin{cases} t & for\ t \neq t_2 \\ t_1 \cup t_2 & for\ t = t_2 \end{cases}$$

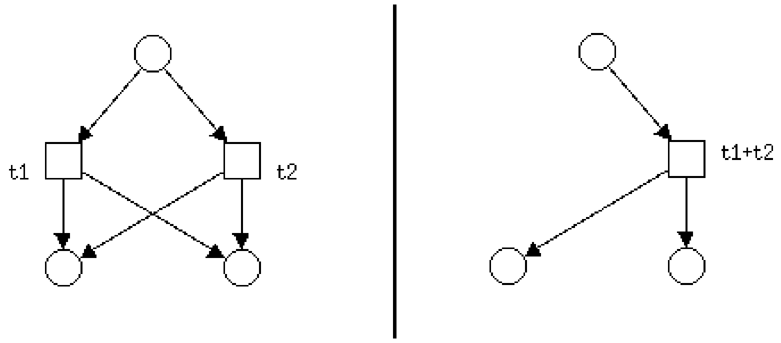


Figure 9.2.9 : *Removal of parallel transitions*

9.3 Fast reduction and analysis

The following net (cf. figure 9.3.10) represents an n-buffer that consists of n unconnected components. Since 2^n different markings are reachable in the net, the size of the reachability graph grows exponentially in the size of the net.

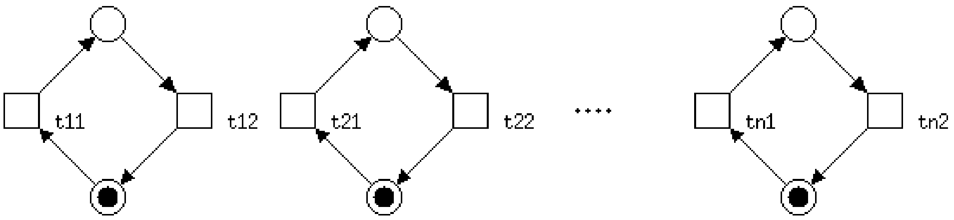


Figure 9.3.10 : *N-buffer*

Now, consider the reduction. After n reduction steps by the rule R1 'Post-Agglomeration', we get a net consisting of n components, each consisting of two vertices (cf. figure 9.3.11).

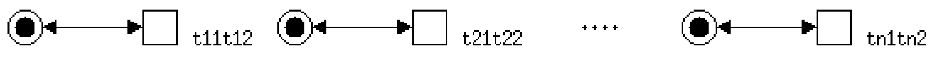


Figure 9.3.11 : *N-buffer after the first n reduction steps*

Once again, we need n reduction steps by the rule R2 'Removal of run-places', which generates n transitions (cf. figure 9.3.12).

Figure 9.3.12 : *N-buffer after the second n reduction steps*

Finally, we use (n-1)-times the rule R3 'Removal of parallel transitions' that builds the union of two isolated transitions. So, we get in 3n-1 reduction steps one transition with the expression $(t_{11}t_{12} \cup t_{21}t_{22} \cup ... \cup t_{n1}t_{n2})^*$ (the expression has got a * because the preset of the resulting transition is empty).

Therefore, the reduction itself is possible in linear time and also the mapping of the transitions to the transitions of the start net is possible in 2n steps. But the possible exchanges of transitions for the partial commutative closure can generate a new regular expression whose size is exponentially larger than the original expression. Nevertheless, it is possible to check some decision problems without applying the partial commutative closure (and without applying the prefix closure). For example, the expression shows us immediately that the net is deadlockfree because the expression is a star-expression which can always be iterated (liveness of a net only depends on the reachable markings and not on the occurrence sequences by which the markings are reached). We presume that our reduction rules always generate a regular expression of the form σ^* if the start net is live (but we did not prove it yet). Besides that, it is also possible to check whithout applying the partial commutative closure whether a place can be marked or unmarked (consider the initial net and check if a transition out of the preset resp. postset of the place occurs in the regular expression).

9.4 Conclusions

We have introduced a new method to compute the net language of safe Petri nets. The computation method is based on language preserving net reductions. In many cases, it only needs time polynomial in the size of the net, although the reachability graph is exponential.

In this sense, our approach contributes to the fast analysis of Petri nets. The main problem, however, lies in the fact that, up to now, we were not able to prove the completeness of our method for a nontrivial class of nets.

Another problem concerning the reduction process can be the complexity of the derivation because in the worst case, the derivation consists of several applications of the rule 'Post-Agglomeration' where the number of transitions in the net can grow because of the cross-product of transitions. In this case, the derivation can grow exponentially in the size of the net. Besides that, the computation of the partial commutative closure and of the prefix closure can be exponential in the size of the generated regular expression.

This shows that future work should concentrate on the analysis of nets, especially on those problems which can be solved by analyzing the regular expression without applying the partial commutative and the prefix closure. Another question is whether there exists a general strategy for the reduction process which minimizes the number of reduction steps.

The set of reduction rules is currently being implemented in the **PEP**-Project. Experimental studies based on this implementation will hopefully contribute to answer these questions.

Literatur

[1] GÉRARD BERTHELOT: *Checking Properties of Nets Using Transformations.* In: G. ROZENBERG (Herausgeber): *Advances in Petri Nets*, Band 222 der Reihe *Lecture Notes in Computer Science*, Seiten 19 – 40. Springer-Verlag, 1986.

[2] GÉRARD BERTHELOT: *Transformations and Decompositions of Nets.* In: *Petri Nets: Central Models and Their Properties*, Band 254 der Reihe *Lecture Notes in Computer Science*, Seiten 359 – 376. Springer-Verlag, 1987.

[3] EIKE BEST und RAYMOND DEVILLERS: *Theoretical Computer Science*, Band 55, Kapitel Sequential and Concurrent Behaviour in Petri Net Theory, Seiten 87 – 136. Elsevier Science Publisher, 1987.

[4] EIKE BEST, RAYMOND DEVILLERS und JAVIER ESPARZA: *General Refinement and Recursion for the Box Calculus.* Nummer26/92 in *Hildesheimer Informatik-Bericht.* Universität Hildesheim, Dezember 1992.

[5] EIKE BEST, RAYMOND DEVILLERS und JAVIER ESPARZA: *General Refinement and Recursion for the Box Calculus.* In: P. ENJALBERT, A. FINKEL und K.W. WAGNER (Herausgeber): *STACS'93*, Band 665 der Reihe *Lecture Notes in Computer Science*, Seiten 130 – 140. Springer-Verlag, 1993.

[6] EIKE BEST, RAYMOND DEVILLERS und JON G. HALL: *The Box Calculus: a New Causal Algebra with Multi-Label Communication.* In: G ROZENBERG (Herausgeber): *Advances in Petri Nets 92*, Band 609 der Reihe *Lecture Notes in Computer Science*, Seiten 21 – 69. Springer-Verlag, 1992.

[7] EIKE BEST und JAVIER ESPARZA: *Model Checking of Persistent Petri Nets.* In: *5th Wokshop Computer Science Logic-91*, Band 626 der Reihe *Lecture Notes in Computer Science*, Seiten 35–52. Springer-Verlag, 1992.

[8] EIKE BEST, HANS FLEISCHHACK, WOJCIECH FRACZAK, RICHARD PINDER HOPKINS, HANNA KLAUDEL und ELISABETH PELZ: *A High Level Petri Net Semantics of B(PN)^2.* Rapport de Recherche 895, LRI, Université Paris-Sud, March 1994. (submitted).

[9] EIKE BEST und RICHARD PINDER HOPKINS: *B(PN)^2 – a Basic Petri Net Programming Notation.* Nummer 27/92 in *Hildesheimer Informatik-Bericht.* Universität Hildesheim, Dezember 1992.

[10] EIKE BEST und RICHARD PINDER HOPKINS: *B(PN)^2 – a Basic Petri Net Programming Notation.* In: A. BODE, M. REEVE und G. WOLF (Herausgeber): *Proceedings of PARLE '93*, Band 694 der Reihe *Lecture Notes in Computer Science*, Seiten 379 – 390. Springer-Verlag, 1993.

[11] E. M. CLARKE, E. A. EMERSON und A. P. SISTLA: *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, Testing and Verification.* ACM Transactions on Programming Languages and Systems, 8(2):244 – 263, 1986.

[12] JÖRG DESEL: *Reduction and Design of Well-Behaved Concurrent Systems.* In: J. C. M. BAETEN und J. W. KLOP (Herausgeber): *CONCUR'90*, Band 458 der Reihe *Lecture Notes in Computer Science*, Seiten 166 – 181. Springer-Verlag, 1991.

[13] RAYMOND DEVILLERS: *On a more Liberal Synchronisation Operator for the Petri Box Calculus.* Technischer BerichtTR-LIT-281, Université Libre de Bruxelles, May 1993.

[14] RAYMOND DEVILLERS: *The Synchronisation Operator Revisited for the Petri Box Calculus.* Technical Report LIT-290, Université Libre de Bruxelles, 1994.

[15] JOST ENGELFRIET: *Branching Processes of Petri Nets.* Acta Informatica, Seiten 575 – 591, 1991.

[16] JAVIER ESPARZA: *Reduction and Synthesis of Live and Bounded Free Choice Petri Nets.* Nummer7/91 in *Hildesheimer Informatik-Bericht.* Universität Hildesheim, Juni 1991.

[17] JAVIER ESPARZA: *Model Checking Using Net Unfoldings.* Hildesheimer Informatik-Bericht. Universität Hildesheim, Oktober 1992.

[18] JAVIER ESPARZA: *A Partial Order Approach to Model Checking.* Habilitationsschrift, Universität Hildesheim, Februar 1993.

[19] H.-C. FENG und R. M. FUJIMOTO: *A Shared Memory Algorithm and Performance Evaluation of the Generalized Alternative Construct in CSP.* In: *Proceedings of the 1988 International Conference on Parallel Processing*, Band II, Seiten 176 – 179. The Pennsylvania State University Press, 1988.

[20] HANS FLEISCHHACK: *A HL-Net Approach for Procedures in B(PN)^2.* In Vorbereitung.

[21] HANS FLEISCHHACK und ANJA GRONEWOLD: *Language Preserving Reductions of Safe Petri Nets.* Forschungsbericht, Carl von Ossietzky Universität Oldenburg, 1994.

[22] HANS FLEISCHHACK, ULRIKE LICHTBLAU, ANJA GRONEWOLD, P. AMTHOR, MATTHIAS DAMM, U. EHRHARDT, R. LUKAS, R. MAJCHSZAK, R. RUST, F. SINNEMANN, V. STEGMANN, M. STICKAN und J. TAPKEN: *Endbericht der Projektgruppe Strukturierte Transformationen auf Netzen (STONE).* Interner Bericht 94/1, Universität Oldenburg, Juli 1994.

[23] G. E. HUGHES und M. J. CRESWELL: *An Introduction to Modal Logic.* Methuen and Co., 1968.

[24] LARS JENNER: *Ein Low-Level Prozedurkonzept für B(PN)^2.* Diplomarbeit, Universität Hildesheim, 1994. In Vorbereitung.

[25] MACIEJ KOUTNY, JAVIER ESPARZA und EIKE BEST: *Operational Semantics for the Petri Box Calculus.* In: B. JONSSON und J. PARROW (Herausgeber): *Proceedings of CONCUR'94*, Band 694 der Reihe *Lecture Notes in Computer Science*, Seiten 210 – 225. Springer-Verlag, 1994.

[26] ANDREI KOVALYOV: *On Complete Reducibility of Some Classes of Petri Nets.* In: *Proceedings of the 11th International Conference on Application and Theory of Petri Nets*, Seiten 352 – 366, Paris, 1990.

[27] D. LEWINE: *POSIX Programmer's Guide.*

[28] K. L. MCMILLAN: *Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits.* School of Computer Science, Carnegie Mellon University, May 1992.

[29] STEPHAN MELZER: *Entwurf und Implementierung eines Codegenerators für B(PN)^2.* Diplomarbeit, Universität Hildesheim, 1994. In Vorbereitung.

[30] ROBIN MILNER: *A Calculus of Communicating Systems.* Springer-Verlag, Berlin, Heidelberg, New York, 1980.

[31] M. NIELSEN, G. PLOTKIN und G. WINSKEL: *Petri Nets, Event Structures and Domains, Part I.* Theoretical Computer Science, 13(1):85 – 108, 1981.

[32] H. W. PRESS, B. P. FLANNERY, S. A. TEUKOLSKY und W. VETTERLING: *Numerical Recipes in C.* Cambridge University Press, 1988.

[33] ROBERT-CHRISTOPH RIEMANN: *Eine temporale Logik für parallele Programme.* Diplomarbeit, Universität Hildesheim, Januar 1994.

[34] STEFAN RÖMER: *Implementation einer kompositionellen Halbordnungssemantik von Petri–Boxen.* Diplomarbeit, Universität Hildesheim, Oktober 1993.

[35] STEVENSON: *Programmierung von UNIX-Netzen.* Prentice-Hall, 1992.

[36] THOMAS THIELKE: *Implementierung eines effizienten Modelchecking-Algorithmus.* In: W. REISIG G. SCHESCHONK (Herausgeber): *Petri-Netze im Einsatz für Entwurf und Entwicklung von Informationssystemen*, Seiten 127–139. Springer-Verlag, 1993.

[37] KIMMO VARPAANIEMI: *PROD User's Guide.* Technischer Bericht Helsinki University of Technology, Digital Systems Laboratory, 1992.

[38] ROLF WALTER: *Persönliche Kommunikation*, 1994.