Programming **E**nvironment based on **P**etri nets

Documentation and User Guide Version 1.4

Professor Dr. Eike Best

Dipl. Inform. Bernd Grahlmann

UNIVERSITÄT HILDESHEIM

Institut für Informatik

— November 1995 —

What is **PEP**?

The **PEP** system (**P**rogramming **E**nvironment based on **P**etri Nets) is a Petri net tool, and much more. **PEP** supports the most important tasks of a good net tool, including high-level and low-level net editing with comfortable simulation facilities. In addition, Petri net editing and simulation is embedded in sophisticated programming and verification components. The programming component allows the user to design concurrent algorithms in an easy to use imperative language, and the **PEP** system then generates Petri nets from such programs. The simulation of a Petri net can even trigger the simulation of the corresponding program.

The **PEP** tool's comprehensive verification components allow the analysis of a large range of properties of parallel systems which can be checked efficiently on either programs or their corresponding nets. This includes user-defined properties specified by temporal logic formulae as well as specific properties for which dedicated algorithms have been implemented.

The **PEP** tool can be considered as an open platform. Further algorithms can be integrated in the user interface very easy. **PEP** has been implemented on Solaris 2.4, Sun OS 4.1.3 and Linux.

How to read the documentation

This document is divided into five parts: Installation, General guidelines, Getting Started, Documentation, and Background.

The 'Installation Notes' in section 1 should be followed if you wish to install **PEP** from scratch on a UNIX platform. They can be skipped if there is already a working system on your machine.

After installation, reading the 'Getting Started' section on page 5 should enable you to get the system running quickly and to play with it for a while. **PEP** is menu-, windows- and mouse-driven and conforms to the standards of an X based application. The general guidelines in section 2 give some further hints.

Sections 4 – 10 comprise the full documentation. You can either read these sections without needing a machine by your side, or you can use them as reference points in case of need.

Section 4 explains the start-up window of the **PEP** tool. The functionality of the project window is described in section 5. The use of the **PEP** file browser is explained in section 6. In sections 7–10, editors for different kinds of objects handled by the tool are documented. The $B(PN)^2$ programming language editor is explained in section 7. The Petri net editors for high level nets (M-nets) and for low level nets (Petri boxes) are described, respectively, in sections 8 and 9. The editor for temporal logic formulae is explained in section 10.

The appendix gives some of the theoretical background of this work. It is recommended to read this part in case you wish to appreciate motivation and rationales behind the **PEP** tool.

Different types of objects used by the **PEP** system as well as the typical design cycle are described briefly in appendix A. The structure of the **PEP** prototype and interdependencies of its different components are shown in appendix B from an implementor's view of **PEP**. The parallel programming language $B(PN)^2$ is defined in section C. Its high-level and low-level net semantics is described in section D. Some examples are shown as well. The temporal logic that can be used to express properties of a $B(PN)^2$ program or a low level net are defined in section E. The different model checking algorithms are explained. Appendix F describes some thoughts on future work.

At the end of this document some pointers to relevant literature and an index are given.

Acknowledgments

The **PEP** system has been developed in 1993 – 1994 as part of a common project of the Universities of Hildesheim and Oldenburg sponsored by the DFG (Deutsche Forschungsgemeinschaft). The project was initiated by Eike Best and Hans Fleischhack, based on theoretical work done by E. Best and others in the EU Basic Research projects DEMON (Design Methods based On Nets) and CALIBAN (CAusal CalcuLI BAsed on Nets). In particular, Javier Esparza developed the theory behind the central model checking algorithm used in **PEP**. Bernd Grahlmann led the Hildesheim group of the **PEP** project. He designed and supervised (often also writing and correcting code) the implementation of **PEP** described in this document, holding overall responsibility for the product. This was carried out in cooperation with the Oldenbourg group, mainly interacting with Matthias Damm, Marc Langnickel, Andree Seidel and the STONE group of students.

The following persons made contributions to the implementation:

Burkhard Bieber	Implemented the main part of the graphical user interface spending an incredible amount of time to make it user-friendly.
Stefan Römer	Took care of all kinds of algorithms and optimized them nearly perfectly during the past years.
Burkhard Graves	Implemented some of the compilers, and the model checking algorithm for safe Petri nets.
Tobias Himstedt	Implemented two Petri net editors and simulators.
Lars Jenner	Developed and implemented the low level procedure semantics.
Martin Ackermann	Implemented the compiler from a $B(PN)^2$ program to an M-net.
Michael Kater	Implemented different text editors and contributed to the Linux port.
Stephan Melzer	Developed and implemented the C code generator.
Thomas Thielke	Implemented a dedicated model checking algorithm for T-systems.
Stephan Merkel	Improved the functionality of some of the compilers.
Carola Pohl	Implemented the on-line help system.

Further contributions were made by Farzad Eghtessadi, Johannes Jäger, Eberhard Meisel, Robert Riemann, Hartmut Trüe and Fatma Uzel. The **PEP** project is being continued in cooperation with the Humboldt-Universität zu Berlin.

Contents

1 Installation of the PEP tool

At the present time we provide **PEP Version 1.4** for Solaris 2.3 or 2.4 (abbreviated as `solaris`), SUN OS 4.1.3 (abbreviated as `sunos4_1_3`) and Linux (abbreviated as `linux`). Installation of the **PEP** tool involves the following steps.

1. Download the file:
   `PEP1.4_`*operating-system*`_dem.tar.gz`
   via ftp from:
   `ftp.informatik.uni-hildesheim.de`
   from the directory:
   `/pub/Projekte/PEP`.
   Note: *operating-system* must be replaced by one of the abbreviations mentioned above.
2. Uncompress it with `gzip -d PEP1.4_`*operating-system*`_dem.tar.gz`, obtaining the file `PEP1.4_`*operating-system*`_dem.tar`.
3. Untar this file with `tar -xvf PEP1.4_`*operating-system*`_dem.tar.gz`; a directory `pep` with several subdirectories is created (make sure no file or directory called `pep` is present before executing this step). Note: If you have `gtar` you may combine steps 2 and 3 by typing `gtar -xvzf PEP1.4_`*operating-system*`_dem.tar.gz`.
4. Set an environment variable `PEPHOME` to the name of the new directory.
   If you use `csh` you can include the following line in your `.cshrc` file:
   `setenv PEPHOME` *pathname*`/pep`
   If you use `bash` you can include the following line in your `.profile` file:
   `export PEPHOME="`*pathname*`/pep"`
   You can set `PEPHOME` temporary by typing one of these commands at the shell prompt.
5. Add the subdirectory *pathname*`/pep/bin` to your path.
   If you use `csh` you can include the following line in your `.cshrc` file:
   `set path = ($path` *pathname*`/pep/bin )`
   If you use `bash` you can include the following line in your `.profile` file:
   `export PATH=$PATH:`*pathname*`/pep/bin`
6. Write an email to `pep_help@informatik.uni-hildesheim.de` (Bernd Grahlmann) and ask for a license file. Your mail should contain:
   (a) Some brief information about your interest in the **PEP** tool and about your institution. The **PEP** tool is distributed free of charge only for non-commercial use.
   (b) The hostid of the computer(s) on which you would like to use the tool. You can get this information by typing `hostid` in your shell. (If you run Solaris `/usr/ucb/hostid` should work.)
7. Insert the file you will receive as *pathname*`/pep/.pep_pswd`.
8. Now you may invoke the peptool from any directory by typing `peptool` at the prompt.

Notes:

1. To use the **PEP** tool you need Tcl (Version 7.4), Tk (Version 4.0) and Tcl-DP (Version 3.2). You can get the corresponding files: `tcl7.4.tar.gz`, `tk4.0.tar.gz` and

`tcl-dp3.2.tar.gz` from many ftp servers.
It is necessary to install the patches: `tcl7.4p1.patch.gz`, `tcl7.4p2.patch.gz` and `tk4.0p1.patch.gz` but not `tk4.0p2.patch.gz`. Tcl, Tk and Tcl-DP can be compiled and installed (under `/usr/local/...`) in the given order easily. On SUN workstations Tk and Tcl-DP must be compiled using Openwindows X11; on a Linux PC you must use X11R6.
You can check the version numbers if you invoke a DP-wish typing `dpwish` at the command line by use of the following 3 commands (the correct output is given in boldface):

`puts $tcl_patchLevel`
7.4p2
`puts $tk_patchLevel`
4.0p1
`puts $dp_version`
3.2

In case of doubt, please ask your system administrator.

2. You can invoke the **PEP** tool and use it in a restricted way even if you have not received a license file.

3. Please communicate any problem you experience to `pep_help@informatik.uni-hildesheim.de`.

2 General guidelines

This section describes some guidelines for the usage of the **PEP** tool. We spent a lot of work to eliminate all bugs and to make the tool user friendly, but nobody and of course no tool is perfect.

- During the work with the **PEP** tool it may happen, that Tcl error boxes pop up (see figure 2.2). Although most of the possible non-intended usages of the tool are caught automatically yielding a **PEP** error box like the one shown in figure 2.1, in most of the cases such a usage is the reason for the occurence of the Tcl error box. In this case you can simply click on the **OK** button and continue with your work. If you discovered a real error or wrong behaviour of the tool, please send an e-mail to `pep_help@informatik.uni-hildesheim.de` containing:

  1. the version of the **PEP** tool in use,
  2. the kind of operating system,
  3. the error message (see figure 2.3) that you will get if you click on the **See Stack Trace** button,
  4. a short description of the history; what you were doing before the error box occured.

Figure 2.1: **PEP** error box.

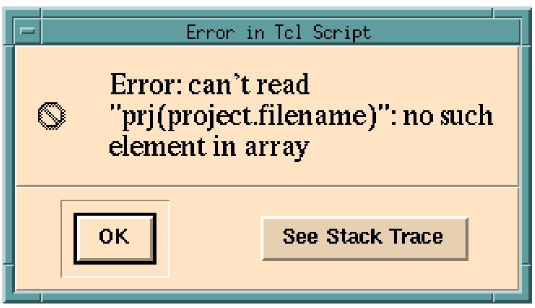

Figure 2.2: Tcl error box.

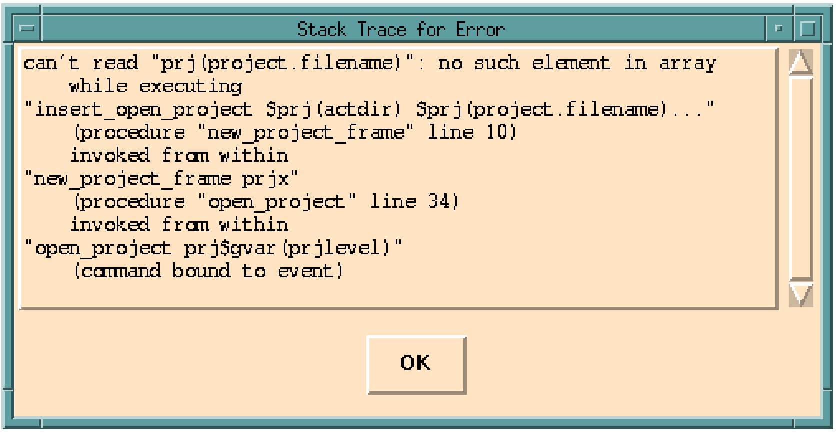

Figure 2.3: See stack trace window.

- In general you can iconify the shell during the work with the **PEP** tool. Some debugging information might be output to the shell if you check the **Display Debug** option in the start-up window. If you are curious you can read them, but it is not necessary for your work with **PEP**.

- In general you do not have to change the default values of the options for the different algorithms and compilers. Most of them are meant for debugging purposes. Some of them may speed up the calculations, but the defaults are nearly perfect for every example. Only some of them (like **No Scoping** for the compiler **B(PN)^2⇒HL Net** or **Take Current Marking Instead of Initial** for the **Prefix Builder**) are interesting for the usual work with the tool. All the figures explaining these options show the default values.

- If you are working with one of the net editors and the editor does not react in the intended way (e.g., does not create a place if you use the **p**-key) it is very likely that the editor is in a different mode. Try to click at any position of the canvas and hit the **Esc** key afterwards.

- If the tool seems to hang it might be the case that there are error boxes or warning boxes or dialog boxes which have not been closed.

3 Getting started

After installing **PEP** and setting the correct paths, the **PEP** tool can be invoked from within any directory by typing `peptool` at the command line. If you are using **PEP** for the first time, we recommend to invoke it from the `Example` directory included in the distribution.

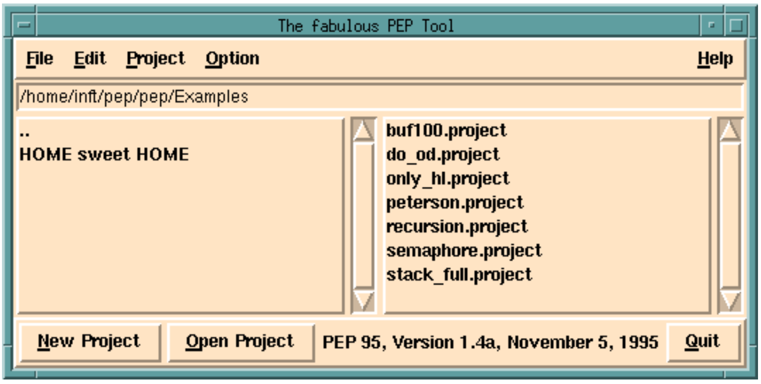

Figure 3.1: The start-up window of the **PEP** system.

The first window that pops up is the start-up window shown in figure 3.1. Its main purpose is to open existing or create new projects. We recommend to open the 'semaphore' project by double-clicking on `semaphore.project` and to follow the steps described below.

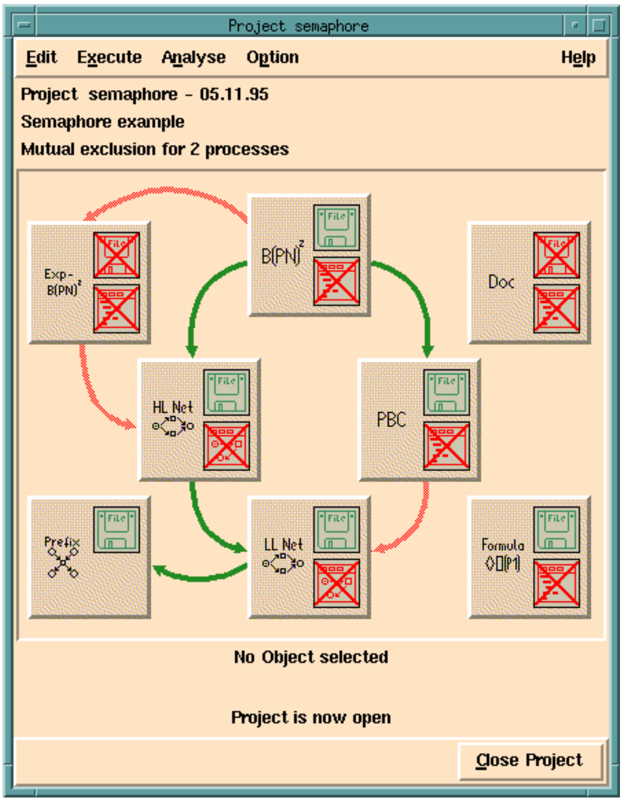

Figure 3.2: Project window belonging to the 'semaphore' project.

- A window belonging to the 'semaphore' project will appear. This window (shown in figure 3.2) is central to the work with a project. It shows the project's objects and their interdependencies. Figure 3.2 displays eight objects: a $B(PN)^2$ program, an expanded $B(PN)^2$ program, a HL (high level) net, a LL (low level) net, a process algebra term (called PBC for **P**etri **B**ox **C**alculus), a temporal logic formula, a finite prefix of the branching process (called Prefix) needed for model checking, and documentation. Objects are saved in separate files in the UNIX file system. Even though the names of these files are not hidden, it is advised not to change the defaults offered by the tool. Possible interdependencies are shown by arrows from one object A to another object B. Arrows can be green, denoting that A currently belongs to B, or red, denoting that currently no dependency exists (for instance, because A has been modified without modifying B accordingly.)

- Open the editor for the B(PN)^2 program by selecting the menu item **Edit▷Open▷ B(PN)^2** or easier by double-clicking on the lower icon within the **B(PN)^2** icon. A text editor window containing an example program for the mutual exclusion of two processes will be displayed.
- Compile the M-net semantics of this program by selecting the menu item **Execute▷ B(PN)^2⇒HL Net**.
- Open the editor for the M-net by selecting the menu item **Edit▷Open▷HL Net** or by double-clicking on the lower icon within the **HL Net** icon. The net editor has an integrated simulator that can be started by selecting the menu items **Action▷Execute** (interactive simulation) or **Action▷Animate** (automatic simulation). If the option **Option▷Highlight Execution** is selected then the corresponding actions in the B(PN)^2 program are highlighted during animation. Furthermore references between actions in the B(PN)^2 program and transitions in the net, and between places in the control flow of the B(PN)^2 program and places in the net, can then be analysed by selecting the menu item **Action▷Show Reference** and clicking on the transitions and places of interest. If necessary the B(PN)^2 editor is opened automatically.
- Compile the Petri box semantics of the program by selecting the menu item **Execute▷ HL Net⇒LL Net**.
- Open the editor for the Petri box by selecting the menu item **Edit▷Open▷LL Net** or easier by double-clicking on the lower icon within the **LL Net** icon. This editor has functionality similar to the high level editor. In particular, there is an analogous simulator.
- There are choices for verifying a property. Either in-built properties (is the net a T-system? is it safe? etc.) can be checked by dedicated algorithms, or a property can be entered and the model checker be invoked to check it. To invoke the dedicated algorithms, select the corresponding menu items of the **Analyse** menu.
- Before invoking the model checker it is necessary to generate the finite prefix of the branching process. To this end, select the menu item **Execute▷LL Net⇒Prefix**.
- Properties can be entered in the formula editor which can be opened by selecting the menu point **Edit▷Open▷Formula** or easier by double-clicking on the lower icon within the **Formula** icon. We recommend to use first the given formula.
- Model checking is started by selecting the menu point **Analyse▷Model Checker (Safe)**. The result is displayed in the bottom of the project window.

Following this general pattern and keeping in mind the standards for window based programs, it will be possible to work with the **PEP** tool in an intuitive way and to explore the system. In case of doubt you can invoke the online help system in the project window. Nevertheless, it is necessary to read the documentation in order to exploit its full functionality.

4 The PEP start-up window

4.1 Description of the start-up window

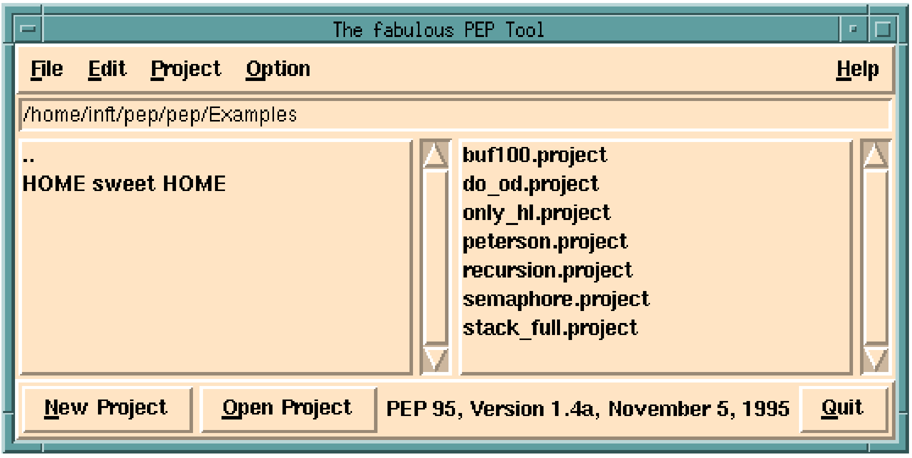

Figure 4.1.1: The start-up window of the **PEP** system.

After invoking **PEP** by typing `peptool` at the command line, the first window that appears is the main window shown in figure 4.1.1. This window is the starting point for the invocation of further windows, each of them belonging to exactly one project. A project comprises all objects belonging to a modelling task. For instance, you may write a $B(PN)^2$ program, then generate the corresponding high and low level nets, (the finite prefix of) the branching process, and finally input temporal logic formulae expressing the properties to be checked. All these objects belong to one project. You may write a different $B(PN)^2$ program and only create some documentation about it, without computing the nets. This will be a different project. You may also use a net editor to create a (HL or LL) net without using a program first. This will be yet another different project.

The start-up window also serves to open existing or to create new projects. Using the left part of the window, the file directory hierarchy can be traversed by double-clicking on the directory names. In the right part of the window, the existing projects in the current directory are displayed. A project can be opened either by a double-click on its name or by a click on the **Open Project** button when the project name is highlighted.

Figure 4.1.2: The new project window.

A new project can be created by clicking on the **New Project** button. The dialog box shown in figure 4.1.2 pops up. A name for the new project can be entered. If a name without the suffix `.project` is entered this suffix is appended automatically, because projects have to end with `.project`.

Clicking the **Quit** button closes all opened projects and quits the **PEP** tool. Be careful, in the current version of **PEP** all unsaved changes (in the currently opened editors) are discarded!

4.2 Menus of the start-up window

The pull-down menus offer the following functionality:

- File
  - ▷ **New Project**: Creates a new project (see figure 4.1.2).
  - ▷ **Open Project**: Opens the selected project.
  - ▷ **Reread Directory**: Rescans the current directory.
  - ▷ **Change Directory**: Switches to the selected directory.
  - ▷ **Make Directory**: Offers the possibility to create a new directory.
  - ▷ **Quit**: Closes all opened projects and quits the **PEP** tool without saving unsaved changes of the currently opened editors.
- Edit
  - ▷ **Copy Project**: Copies a project. The dialog box shown in figure 4.2.3 is displayed. If a name without the suffix `.project` is entered this suffix is appended automatically, because projects have to end with `.project`. An option specifies whether or not all its objects are copied. Be careful, if you do not copy all objects then you end up with two projects using the same files!
  - ▷ **Rename Project**: Renames a project. The dialog box shown in figure 4.2.4 is displayed. If a name without the suffix `.project` is entered this suffix is appended automatically, because projects have to end with `.project`. An option specifies whether or not all its objects are renamed. Be careful, if you do not rename all objects then you end up with a project using files with differing names!
  - ▷ **Delete Project**: Deletes a project. The dialog box shown in figure 4.2.5 is displayed. The name of the project should be entered with or without the postfix `.project`. An option specifies whether or not all its objects are deleted. Be careful, if you do not delete all objects then you end up with owning files that do not belong to any project!
- Help
  - ▷ **About**: Shows the about box for the whole **PEP** tool.

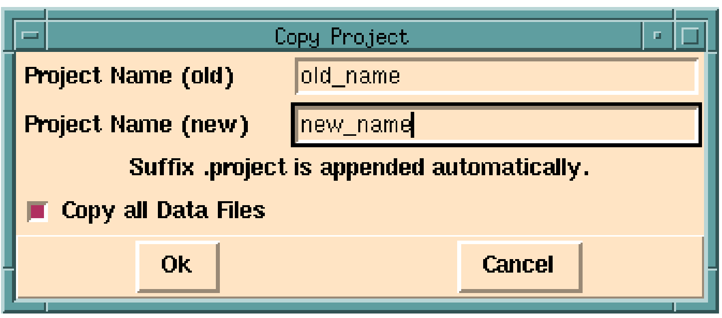

Figure 4.2.3: Copy project window.

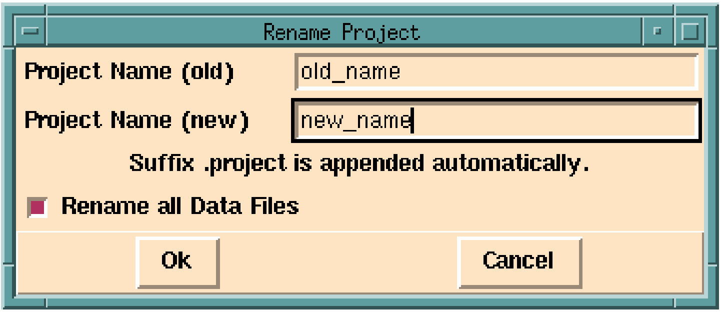

Figure 4.2.4: Rename project window.

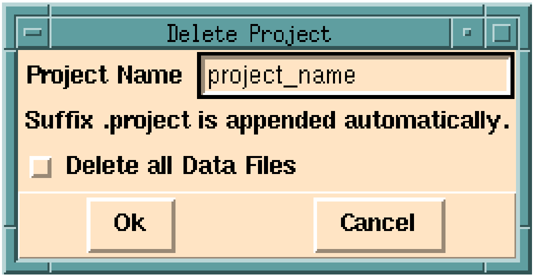

Figure 4.2.5: Delete project window.

- Project
  - ▷ **Project List**: Displays a list of all opened projects (see figure 4.2.6). A double-click on one of them brings its project window to the foreground.

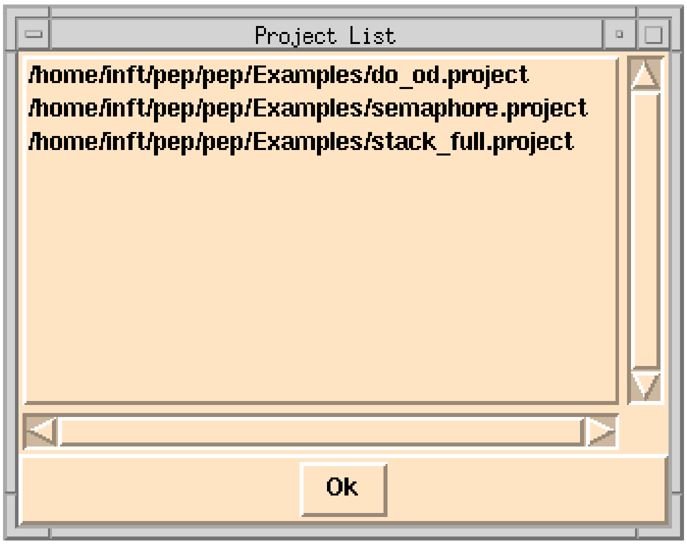

Figure 4.2.6: List of open projects.

- Options — Although these options are available we do not encourage you to use them. They are mainly interesting for the developers of the **PEP** tool.
  - ▷ **Display Debug**: If this option is selected then some debugging information (e.g., all calls to the algorithms invoked by the **PEP** tool, their results, etc.) is written to the shell during the run of the **PEP** tool. Usually you can iconify the shell during your work with the **PEP** tool because it is not needed.
  - ▷ **File Debug**: If this option is selected then the debugging information is continuously written to a file called 'peptool.deb'.
  - ▷ **Show Globals**: Writes Tcl/Tk information to the shell. This information is only interesting for Tcl/Tk programmers.
  - ▷ **Show Open Projects**: Writes a list of all opened projects to the shell.
  - ▷ **Show Com**: Displays some information about communications within the **PEP** tool in a dialog box. This is only interesting if communication between the different parts of the tool does not work correctly.

5 The project window

5.1 Description of the project window

When a project is opened, a project window is displayed on the screen (see figure 5.1.1).

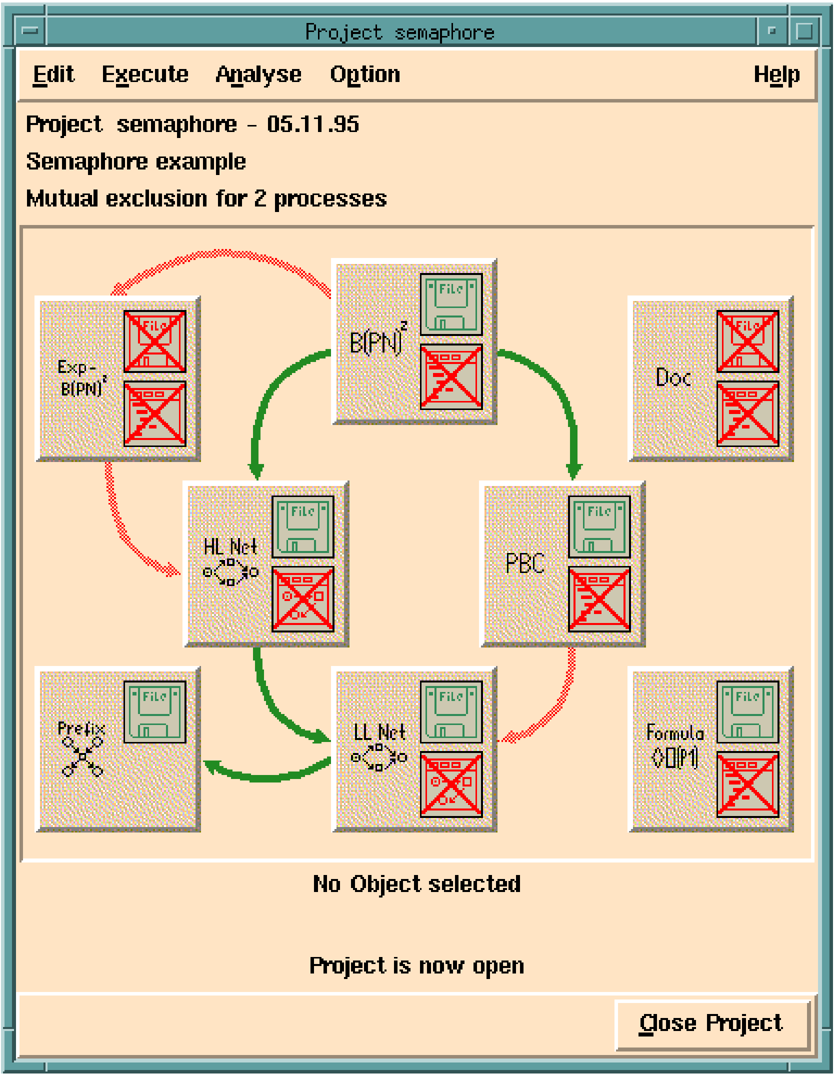

Figure 5.1.1: Project window belonging to the 'semaphore' project.

Since editors can be opened, compilers can be invoked, algorithms can be started, etc., the objects of a project may, at any time, be in different states and various relationships with each other. In addition to the objects themselves and their states, therefore, this window shows the current relationship between the objects of a project. It also allows to specify options for the algorithms and compilers.
The following objects may belong to a project:

- a B(PN)^2 program,
- a B(PN)^2 program with expanded procedures,
- a high level net (M-net),
- a low level net (Petri box),

- a process algebra term (PBC term),
- a finite prefix of the branching process,
- a temporal logic formula,
- a documentation.

The central part of the project window contains one icon for each object. Depending on the state of the project, one of these objects may be associated with another one. Perhaps an M-net has been generated out of the B(PN)^2 program. These relations are presented in a graphical way. All possible relations are displayed by arcs. If object A currently belongs to object B then the arc from A to B is coloured green, otherwise it is coloured red (if a relation is possible but currently not present).

Each object icon contains one or two smaller icons. The top icon tells the user whether or not this object exists. It is coloured green if the object exists and crossed out red if the object does not exist. If this icon is green then there exists a file for the object. If the cursor is placed over the corresponding icon, the location and the file name are displayed under the main part of the window together with further information about the corresponding object. The second icon (if one exists) tells the user whether or not the corresponding editor is currently opened. If the editor is not opened it can be opened by a double-click on this icon, otherwise it is shuffled up to be completely visible.

In the bottom part of the window, the progress of the main actions (invocation of the editors, the algorithms and the compilers, and their results) are reported.

Underneath the pull-down menus, two lines of project information are displayed.

The **Close Project** button can be used to close the project. But be careful, the contents of every opened editor are saved by default before closing! If you want to discard the changes you can use the menu item **Edit**▷**Close**▷**Discard Files**.

5.2 Menus of the project window

Further functionality is offered by the pull-down menus.

- Edit
  - ▷ **Open**: Note: It is easier to open one of the following editors with a double-click on the lower (inner) icon of the corresponding icon.
    - ◦ **B(PN)^2**: Opens the B(PN)^2 editor for the B(PN)^2 program.
    - ◦ **Expanded B(PN)^2**: Opens the B(PN)^2 editor for the expanded B(PN)^2 program, which is the same as the B(PN)^2 program except that procedures have been expanded.
    - ◦ **PBC**: Opens the PBC editor for the process algebra term.
    - ◦ **LL Net**: Opens the editor/simulator for the low level Petri net.
    - ◦ **HL Net**: Opens the editor/simulator for the high level Petri net.
    - ◦ **C Source**: Opens the editor for the generated C code.
      Note: This menu item is disabled in the current version of the **PEP** tool.
    - ◦ **Formula**: Opens the editor for the temporal logic formulae.
    - ◦ **Document**: Opens the editor for the documentation.
  - ▷ **Edit Project Info**: Opens a two-line editor for project information as shown in figure 5.2.2.
  - ▷ **Save Project Info**: Saves the two-line project information, the current information concerning the relations between the different objects, and the analysis results, to the project file.
  - ▷ **Save Entire Project**: Saves the contents of all opened editors.
  - ▷ **Close**:
    - ◦ **Save Files**: Closes the project after saving the contents of all opened editors as well as the two-line project information, the current information concerning the relations between the different objects, and the analysis results, to the project file.
    - ◦ **Discard Files**: Closes the project without saving the contents of all opened editors etc. to the project file.

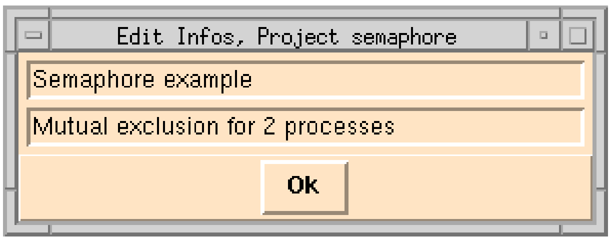

Figure 5.2.2: Edit project information window.

- Execute

  ▷ **B(PN)^2 ⇒ PBC**: Invokes the compiler from a B(PN)^2 program to a process algebra term. The behaviour of the compiler is controlled by the options shown in figure 5.3.8 on page 20 and influenced by the options shown in figure 5.3.7 on page 20.

  ▷ **PBC ⇒ LL Net**: Invokes the compiler from a process algebra term to a low level Petri net (Petri box). The behaviour of the compiler is controlled by the options shown in figure 5.3.9 on page 21.

  ▷ **B(PN)^2 ⇒ LL Net (via PBC)**: Invokes two compilers from a B(PN)^2 program to a process algebra term and then to a low level Petri net (Petri box). This is controlled by the options mentioned in the previous two paragraphs.

  ▷ **B(PN)^2 ⇒ HL Net**: Invokes the compiler from a B(PN)^2 program to a high level net (M-net). The behaviour of the compiler is controlled by the options shown in figure 5.3.10 on page 22 and influenced by the options shown in figure 5.3.7 on page 20.

  ▷ **Expanded B(PN)^2 ⇒ HL Net**: Invokes the compiler from an expanded B(PN)^2 program (without procedures)to a high level net (M-net). The behaviour of the expander is controlled by the options shown in figure 5.3.10 on page 22 and influenced by the options shown in figure 5.3.7 on page 20.

  ▷ **HL Net ⇒ LL Net**: Invokes the unfolder from a high level net (M-net) to a low level net.

  ▷ **B(PN)^2 ⇒ LL Net (via HL Net)**: Invokes two compilers from a B(PN)^2 program to a high level net and then to a low level Petri net (Petri box). If procedures are used in the B(PN)^2 program they are expanded first on the B(PN)^2 level.

  ▷ **B(PN)^2 ⇒ Expanded B(PN)^2**: If procedures are used in a B(PN)^2 program they are expanded on the B(PN)^2 level. The behaviour of the compiler is controlled by the options shown in figure 5.3.15 on page 26 and influenced by the options shown in figure 5.3.7 on page 20.

  ▷ **B(PN)^2 ⇒ C Source**: Invokes the C code generator taking a B(PN)^2 program as input.
  <u>Note</u>: This menu item is disabled in the current version of the **PEP** tool.

- Execute (continued)
  - ▷ **LL Net ⇒ Prefix**: Invokes the prefix builder for the calculation of the finite prefix of the branching process. The behaviour of the prefix builder is controlled by the options shown in figure 5.3.11 on page 23. Be careful, this works correctly only for safe Petri nets, but for reasons of efficiency it is not checked whether or not the net is safe. Every LL net (Petri box) generated out of a $B(PN)^2$ program is automatically safe.
  - ▷ **Minimize Net**: Calculates all transitions of the low level net which are dead under the initial marking and deletes them together with their corresponding arcs. This may reduce the size of the net considerably if the net has been generated out of a $B(PN)^2$ program. Be careful, this works correctly only for safe Petri nets, but for reasons of efficiency it is not checked whether or not the net is safe. Every LL net (Petri box) generated out of a $B(PN)^2$ program is automatically safe.

Note: Compiling a $B(PN)^2$ program into a LL net via a HL net (rather than via a PBC term) creates a number of references from the LL net to the program. Thus, clicking on a transition in the LL net editor will also highlight the atomic action(s) in the $B(PN)^2$ editor to which this transition corresponds if the option **Highlight Execution** is checked. References are explained further in sections 7.2.4, 8.2 and 9.2 (pages 32, 40 and 48, respectively).

- Analyse
  - ▷ **T-System**: Checks whether a low level Petri net is a T-system.
  - ▷ **FC System**: Checks whether a low level net is free choice.
  - ▷ **Safe System**: Checks whether a low level net is safe.
  - ▷ **Bounded System**: Checks whether a low level net is n-bounded. The dialog box shown in figure 5.2.3 is opened. The upper bound n can be entered and the result of the previous test is displayed. The analysis is controlled by the options shown in figure 5.3.12 on page 25.
  - ▷ **Reachability**: Checks whether a marking of a low level net is reachable. The dialog box shown in figure 5.2.4 is opened. The names of the places and the number of tokens on these places can be entered and the results of the previous test are displayed. The analysis is controlled by the options shown in figure 5.3.13 on page 25. The algorithm calculates the reachability graph and works only for bounded (not necessarily safe) Petri nets. If a net is not bounded this is displayed.

- Analyse (continued)

  ▷ **Deadlock-free**
  ▷ **System**: Checks whether a low level Petri net is deadlock-free. The dialog box shown in figure 5.2.5 is opened. This algorithms works on the finite prefix of the branching process, which must have been created previously. Thus it works correctly only for safe systems.

  ▷ **Model Checker**
  ▷ **(T-System)**: Performs very fast model checking for T-systems. This algorithm takes a low level T-system and a formula as input. It uses linear algebra and does not need any precalculations such as the finite prefix. The T-system property is checked automatically, because it would work incorrectly for non-T-systems.

  ▷ **Model Checker**
  ▷ **(Safe)**: Invokes a fast model checking algorithm for safe Petri nets. This algorithm takes the finite prefix of the branching process of a low level Petri net and a formula as input. The behaviour of the model checker is controlled by the options shown in figure 5.3.14 on page 25.

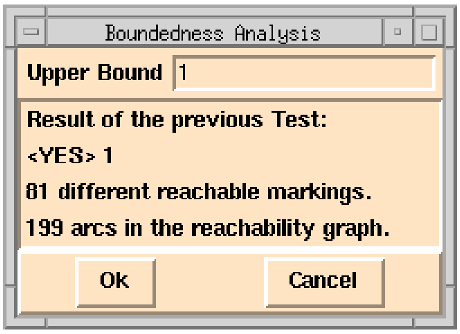

Figure 5.2.3: Boundedness analyser.

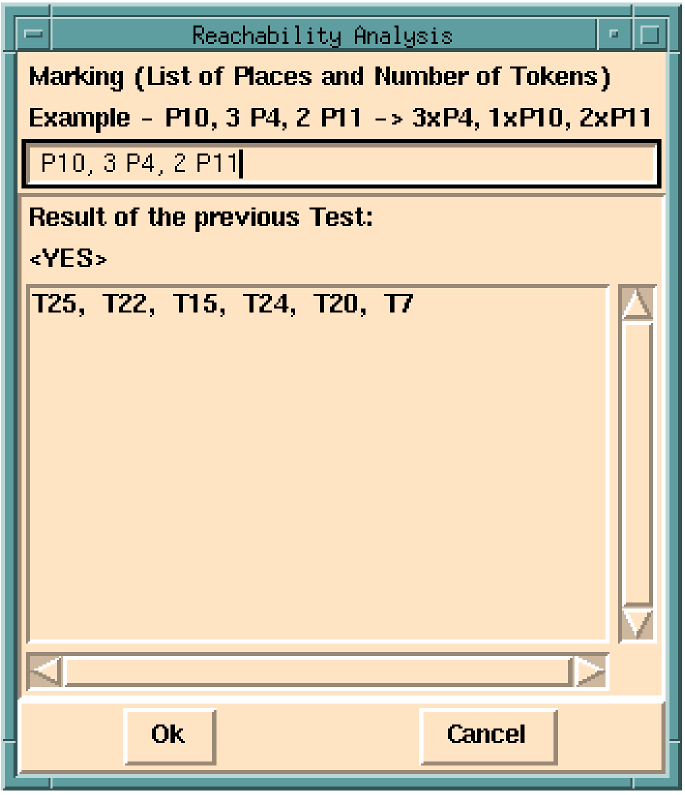

Figure 5.2.4: Reachability analyser.
Checks whether or not a state in which P10 is marked with one token, P4 with 3 tokens and P11 with 2 tokens can be reached.
Note: A **YES** means that the net is bounded. Otherwise the text **The net is not bounded** is displayed. If the marking is reachable the path is displayed in the main part of the window. Otherwise no path is displayed.

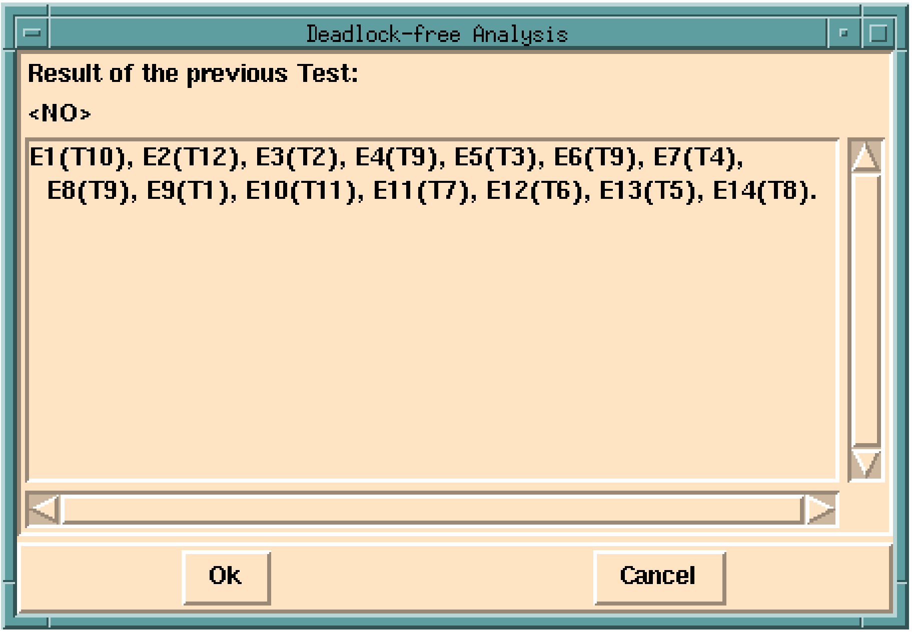

Figure 5.2.5: Analyser to check the deadlock-freeness property.
Checks whether or not a state can be reached in which no transition is enabled and displays the corresponding path if one exists.

- Option
  - ▷ **Converter Options**:
    - ◦ **Syntax Check**: Controls whether or not the syntax of a B(PN)^2 program is checked automatically before a compiler is invoked. The dialog box shown in figure 5.3.7 is displayed.
    - ◦ **B(PN)^2 ⇒ PBC**: Controls the behaviour of the compiler from the B(PN)^2 program to a PBC term. The dialog box shown in figure 5.3.8 is displayed.
      Note: Using this compiler no reference information is generated.
    - ◦ **PBC ⇒ LL Net**: Controls the behaviour of the compiler from the PBC term to the low level Petri net. The dialog box shown in figure 5.3.9 is displayed.
      Note: Using this compiler no reference information is generated.
    - ◦ **B(PN)^2 ⇒ HL Net**: Controls the behaviour of the compiler from the B(PN)^2 program to the high level Petri net (M-net). The dialog box shown in figure 5.3.10 is displayed.
    - ◦ **Prefix Builder**: Controls the behaviour of the prefix builder of the finite prefix of the branching process. The dialog box shown in figure 5.3.11 is displayed.
    - ◦ **Boundedness Analyser**: Controls the behaviour of the boundedness analyser. The dialog box shown in figure 5.3.12 is displayed.
    - ◦ **Reachability Analyser**: Controls the behaviour of the reachability analyser. The dialog box shown in figure 5.3.13 is displayed.
    - ◦ **Model Checker (Safe)**: Controls the behaviour of the model checking algorithm for safe Petri nets. The dialog box shown in figure 5.3.14 is displayed.
    - ◦ **B(PN)^2 Expander**: Controls the behaviour of the B(PN)^2 expander. The dialog box shown in figure 5.3.15 is displayed.
  - ▷ **Debug Output**:
    - ◦ **Show prjx**: Writes information concerning the state of the project to the shell. (E.g. the current relationship between the objects, the information whether or not file exists, or editors are opened, some information about the communication between the different parts of the tool, the sizes and the dates of the files, etc.)

- Help
  - ▷ **Help Viewer**: Invokes the online help system (shown in figure 5.2.6) for the whole **PEP** tool.

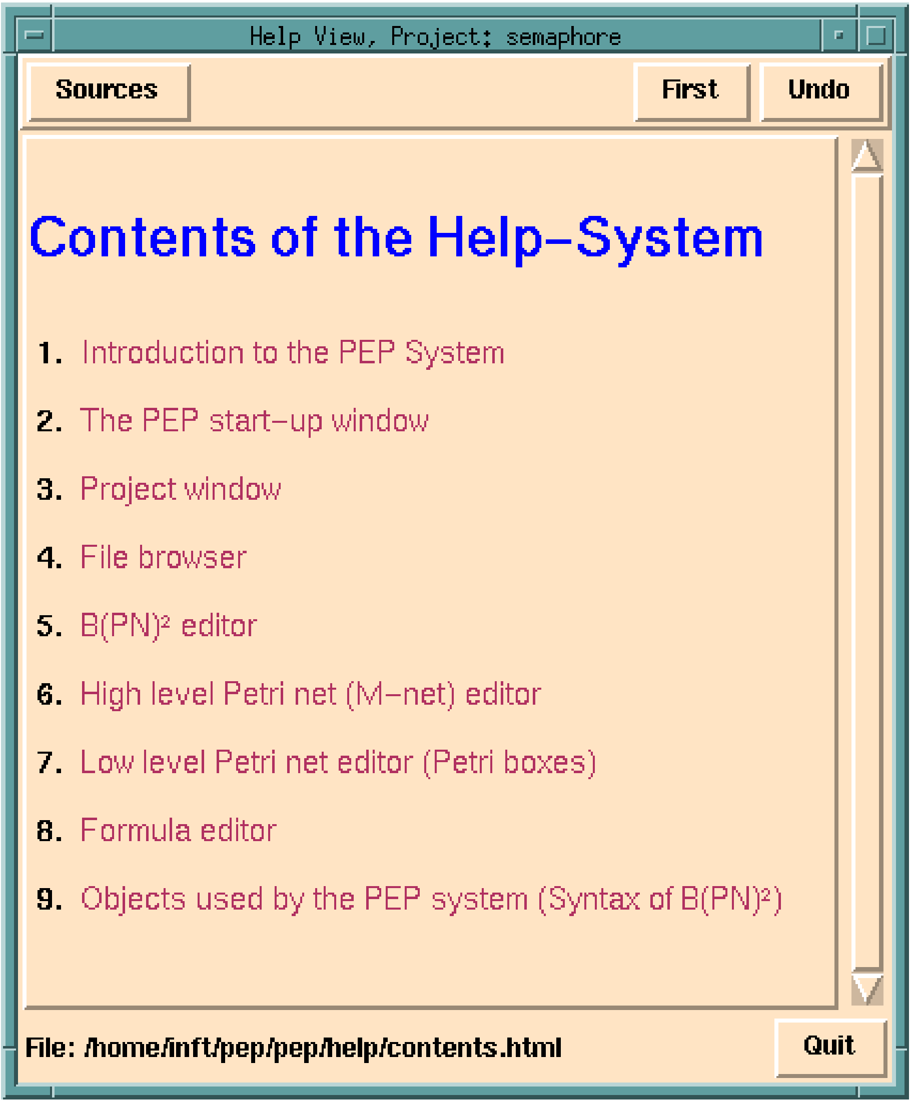

Figure 5.2.6: Online help system of the **PEP** tool.
The help system is based on HTML pages. It can be used in a similar way as netscape. If a text is highlighted in red the corresponding page is opened by a click with the left mouse button. The name of the file corresponding to the current page is displayed in the bottom of the window. You can go back to the previous page (until you reach the first one) using the **Back** icon. You can go back to the first page using the **First** icon. If you would like to load a page by using a browser you can click on the **Source** icon. The help viewer is closed either if you click on the **Quit** button or if the project is closed.

5.3 Explanation of the options

The majority of the options are meant for debugging purposes or for advanced users who are familiar with the algorithms. For normal work it should be sufficient to use the defaults.

5.3.1 Syntax check

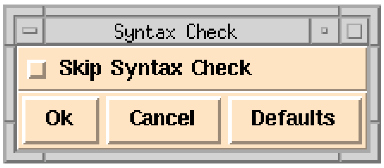

Figure 5.3.7: Options for the syntax check.

- The user can choose to have the syntax of a $B(PN)^2$ program checked before it is compiled into a PBC term or into an M-net. The compiler into M-nets is able to handle some small extensions that are not accepted by the syntax checker or the other compiler. These extensions will be documented in a later release.

5.3.2 $B(PN)^2 \Rightarrow PBC$

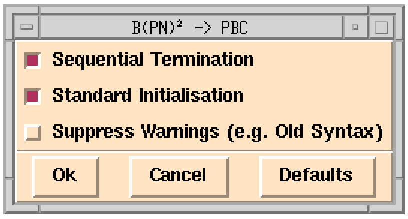

Figure 5.3.8: Options for the compiler $B(PN)^2 \Rightarrow$ PBC.

- Originally there had been an exit transition for every possible combination of variable bindings. Thus a possibly exponential amount of transitions had been created. It is more space efficient to terminate the variables of a block sequentially.
- The user can specify that variables are initialised with the smallest possible value (resp. *false* for Boolean variables).
- If one writes a program using old syntax, warnings can be suppressed. This option is only for compatibility with older versions of the **PEP** tool.

5.3.3 PBC⇒LL Net

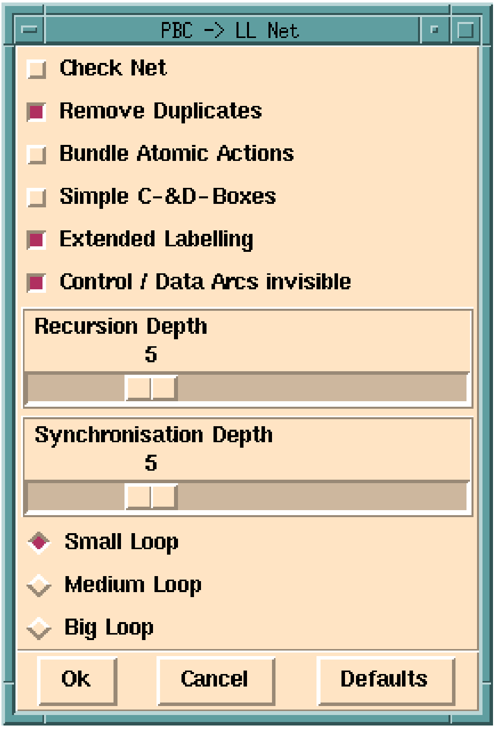

Figure 5.3.9: Options for the compiler PBC ⇒ LL-net.

- The net can be checked for consistency. This is meant only for debugging purposes (e.g. memory allocation is checked).
- Place and transition duplicates (identical pre- and postsets and meanings) can be removed automatically.
- Transitions corresponding to the same action or the same parallel composition of actions can be displayed at the same position on the screen.
- The user can specify that simple boxes are generated for plain and channel variables, otherwise the initialisation transitions are synchronised with transitions of the control flow which yields nets with a smaller prefix.
- Extended labellings can be generated. A transition corresponding to the action ⟨ *a' = 'a - 1* ⟩ will have the meaning *a' = 'a - 1* which can be displayed optionally in the net editors using **Option▷Display Meanings**.
- Arcs between the data boxes of the net and the control flow part of the net can be specified as invisible to enhance readability of the nets constructed from a program. Invisible arcs can be displayed optionally in the net editor using **Option▷Show Invisible Arcs** (see pages 41 and 48).
- The depth of recursion must be restricted to make sure that the LL nets are finite.
  Note: The slider can be moved even when **Recursion Depth** is unchecked, but then it has no effect.
- The maximum number of transitions which can be synchronized by one synchronisation should be restricted to make sure that the LL nets are finite.
  Note: The slider can be moved even when **Synchronisation Depth** is unchecked, but then it has no effect.

- Different loop semantics are implemented in the compiler. All of them behave in an equivalent way. For larger examples the *small* loop should be chosen for efficiency reasons when calculating the finite prefix of the branching process. The *medium* and *large* versions are meant to support future theoretical work.

5.3.4 B(PN)^2⇒M-Net

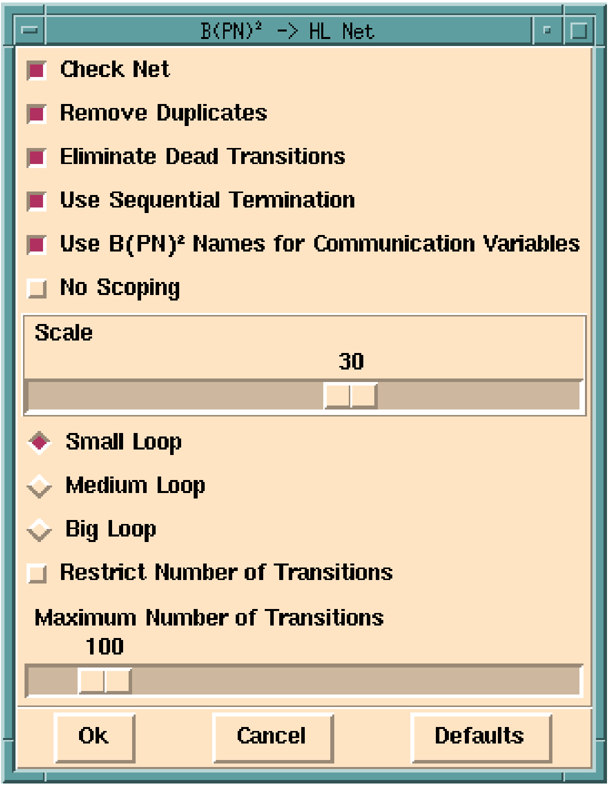

Figure 5.3.10: Options for the compiler B(PN)^2 ⇒ M-net.

- The net can be checked for consistency. This is meant only for debugging purposes.
- Place and transition duplicates can be removed automatically.
- Some transitions can be recognized as initially dead and can be removed automatically.
- Originally there had been an exit transition for every possible combination of variable bindings. Thus a possibly exponential amount of transitions had been created. It is more space efficient to terminate the variables of a block in a sequential way.
- The names of the variables in the B(PN)^2 program can be used for the communication variables in the arc inscriptions. This yields M-nets that are easier to understand, otherwise the communication variables ends with e.g. `@14`.
- If the final scoping (for the outermost block as well as for the inner ones) is omitted then the data boxes and the control flow can be analysed separately. It is not possible to simulate such nets.
- The graphical distances between the elements of the generated net can be changed.

- Different loop semantics are implemented in the compiler. All of them behave in an equivalent way. For larger examples the *small* loop should be chosen for efficiency reasons when calculating the finite prefix of the branching process. The *medium* and *large* versions are meant to support future theoretical work.

- Depending on the power of the machine it might be useful to restrict the maximum number of generated transitions. Otherwise it is possible that one has to wait a while for the end of this calculation or a later one. If the maximum number is reached the compilation is stopped and no net is created. At the present time, no mechanism to stop the execution of an algorithm is implemented. Thus a **kill** command (in the shell) is the only way to stop execution, unless this option is set.
  Note: The slider can be moved even when **Restrict Number of Transitions** is unchecked, but then it has no effect.

Note: Arcs between the data boxes of the net and the control flow part of the net are specified (automatically) as invisible to enhance readability of the nets constructed from a program. Invisible arcs can be displayed optionally in the net editor using **Option▷Show Invisible Arcs** (see pages 41 and 48).

5.3.5 Prefix Builder

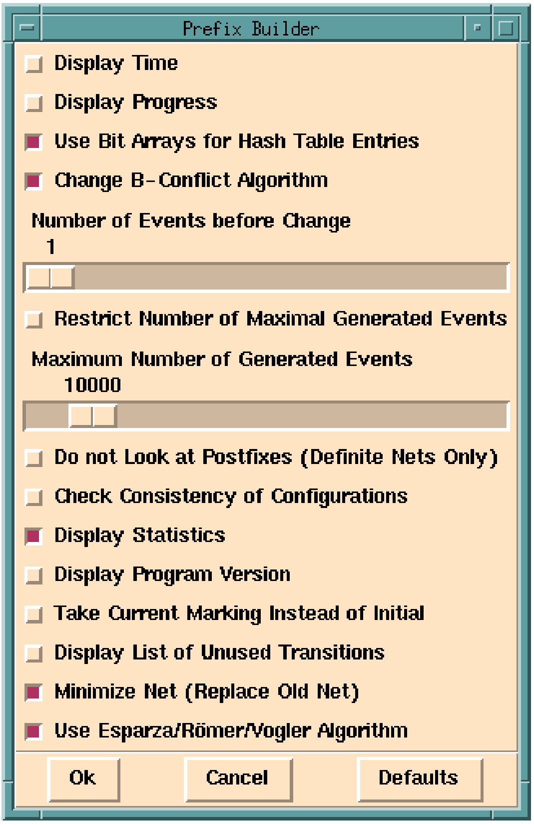

Figure 5.3.11: Options for the prefix builder.

The prefix builder has several options that may increase the speed of the calculations. These options are meant for the developer. Usually the default values are fine. Note that changing critical options may result in incorrect prefixes.

- The time used for the calculation can be written to the shell.
- The progress of the calculation can be written to the shell. This may be useful for large nets.
- Bit arrays can be used internally for hash table entries. This option often increases the speed of the calculation.

- The B-conflict algorithm can be changed. In addition to that it is possible to change it after a certain number of events. This option often increases the speed of the calculation.
  Note: The slider can be moved even when **Number of Events before Change** is unchecked, but then it has no effect.

- Depending on the power of the machine it might be useful to restrict the maximum number of generated events. (You can set the **Display Progress** option to observe it). However there is no direct correspondence between the number of generated events and the duration of the calculation. At the present time, no mechanism to stop the execution of an algorithm is implemented. Thus a **kill** command (in the shell) is the only way to stop execution, unless this option is set.
  Note: The slider can be moved even when **Maximum Number of Generated Events** is unchecked, but then it has no effect.

- For some nets it makes no difference whether postfixes are considered. This option may increase the speed of the calculation. But be careful, it may result in wrong finite prefixes of the branching process! This option is meant only for the developer of the prefix builder.

- The consistency of the calculated configurations can be checked. This option is meant only for debugging purposes.

- The results of the calculations can be written to the shell. Sizes of the original net and the prefix are given. The time used is output.

- The version number of the prefix builder can be written to the shell.

- The current marking can be used for the calculation of the prefix instead of the initial marking. The size of the prefix can be reduced if the net is simulated up to a certain state and this marking is then taken as input for the prefix builder.

- The unused transitions (that is, the initially dead ones that can never occur) can be written to the shell.

- The unused transitions (that is, the initially dead ones that can never occur) can be deleted together with their surrounding arcs. Be careful, the original net is replaced by the reduced net!

- The Esparza/Römer/Vogler algorithm [ERV95] is more efficient than the original one and yields minimal prefixes.

5.3.6 Boundedness Analyser

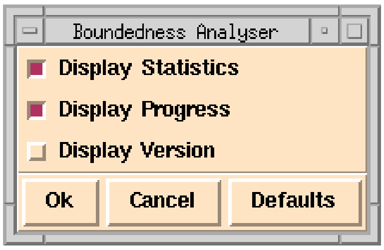

Figure 5.3.12: Options for the boundedness analyser.

- The results of the calculations can be written to the shell. The number of reachable markings and the number of arcs are given.
- Progress information can be written to the shell. This can be useful for large nets.
- The version number of the boundedness analyser can be written to the shell.

5.3.7 Reachability analyser

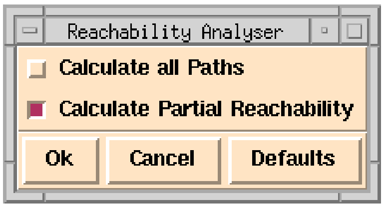

Figure 5.3.13: Options for the reachability analyser.

- Either all paths or a shortest one can be calculated.
- It can be specified that the algorithm should search for a marking of which the given marking is a submarking.

Note: If you check both otions, it is possible that hundreds of paths are calculated and displayed.

5.3.8 Model Checker (Safe)

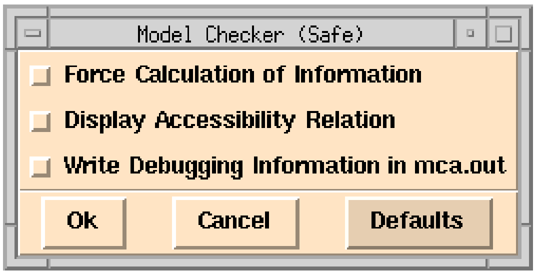

Figure 5.3.14: Options for the model checking algorithm for safe Petri nets.

- The model checker needs certain isomorphisms for the verification. They can be written to a file. It can be chosen whether the model checker decides if it is necessary to calculate them (because they do not yet exist, or because the file containing the prefix is newer than the files containing the isomorphisms) or if they are calculated anyway.

- The accessibility relation can be written to the shell. This option is meant only for debugging purposes.
- Some debugging information can be written to the file mca.out. This option is meant only for debugging purposes.

5.3.9 $B(PN)^2$ expander

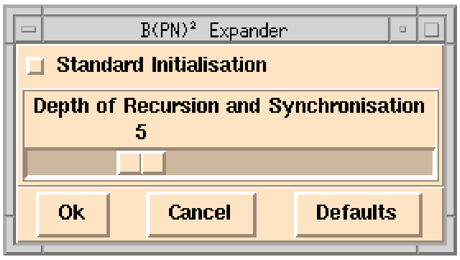

Figure 5.3.15: Options for the $B(PN)^2$ expander.

- The user can specify that variables are initialised with the smallest possible value (resp. *false* for Boolean variables).
- The depth of recursion must be restricted to make sure that the nets are finite.

6 The file browser

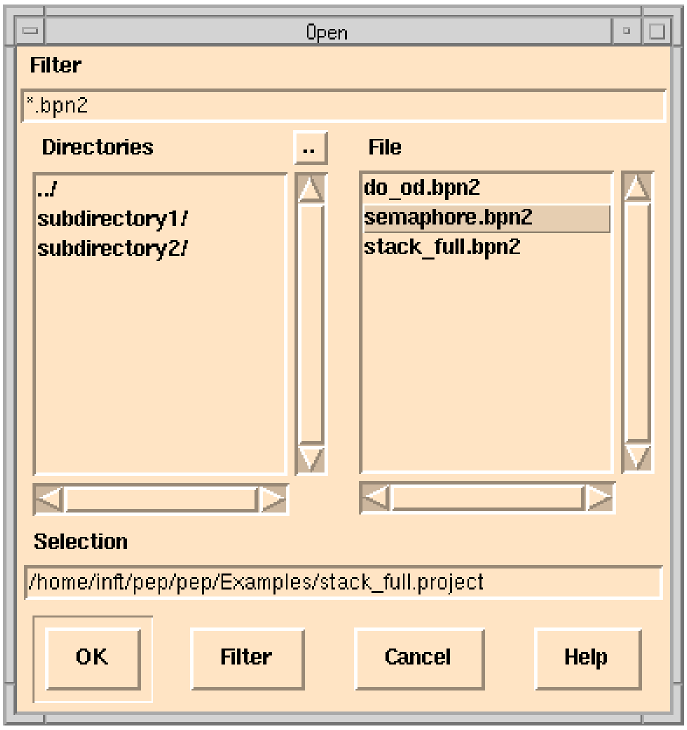

Figure 6.1: File browser.

- Every time a user wants to open a file, a file browser is used for choosing it.
- In the left part of the window the directory hierarchy can be traversed by double-clicking on the directory names.
- In the right part the existing files in the current directory are displayed.
- The list can be restricted by a filter. For example all $B(PN)^2$ programs can be listed by selecting `*.bpn2` as a filter, provided you follow the naming conventions and do not name a $B(PN)^2$ program, say, `bill.txt`.
- The current selection is displayed at the bottom of the window.
- A file can be opened either by a double-click on its name or by a click on the **OK** button when the file is selected.
- The file browser can be closed by a click on the **Cancel** button.
- The **Help** button invokes a help page.

7 The B(PN)^2 editor

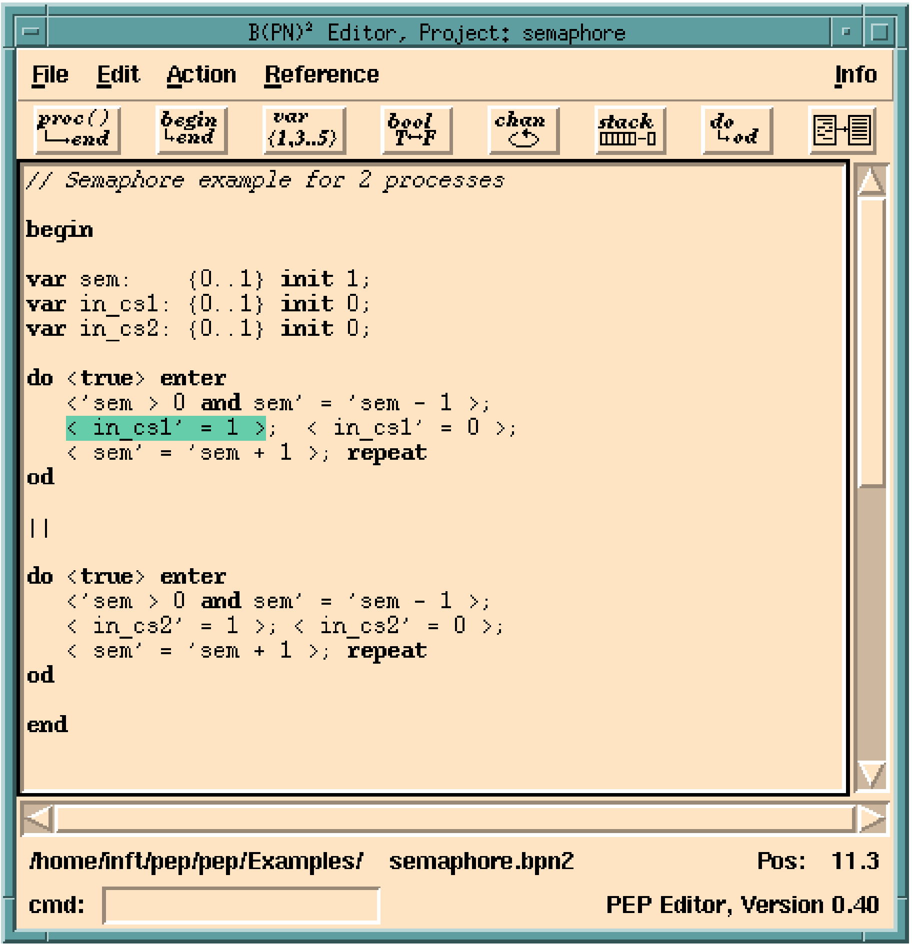

Figure 7.1: B(PN)^2 editor.

7.1 Description of the B(PN)^2 editor window

- The B(PN)^2 editor is a text editor with some B(PN)^2 specific features.
- In the main part of the editor window a B(PN)^2 program can be edited.
- If the keyboard offers this possibility the **cut**, **copy** and **paste** keys can be used.
- The icons underneath the pull-down menu are macros for typing part of a B(PN)^2 program.
  - The **proc()–end** button pastes the declaration of a procedure to the current cursor position.
  - The **begin–end** button pastes a *begin-end* block to the current cursor position.
  - The **var {1,3..5}** button pastes the declaration of a plain variable to the current cursor position.
  - The **bool** button pastes the declaration of a boolean variable to the current cursor position.
  - The **chan** button pastes the declaration of a channel variable to the current cursor position.

   - The **stack** button pastes the declaration of a stack variable to the current cursor position.
   - The **do–od** button pastes a *do-od* construct to the current cursor position.

- The icon underneath the **Help** item invokes an algorithm to display keywords in boldface and comments in italic.

7.2 Menus of the B(PN)^2 editor

- File
  - ▷ **Open**: Opens the file browser to select a B(PN)^2 program.
  - ▷ **Reload**: Reloads the program (e.g. if you made a mistake and want to go back to the previous version).
  - ▷ **Save**: Saves the contents of the B(PN)^2 editor to a file. The name of the file is displayed underneath the main part of the window.
  - ▷ **Save as**: Opens the file browser to save the contents of the B(PN)^2 editor to a file.
  - ▷ **Import File**: Opens the file browser to select the name of a file to be loaded into the B(PN)^2 editor at the current cursor position.
  - ▷ **Export Clipboard**: Opens the file browser to select the name of a file. The contents of the clipboard are saved to this file. You can either use the **copy** button on your keyboard or the menu entry **Edit▷Copy** to copy a selection to the clipboard. Be carefull, the clipboard is not local to the **PEP** tool, it can be changed e.g. if you hit the **copy** key on your keyboard even if you work with a totally different program!
  - ▷ **Quit**: Closes the B(PN)^2 editor. If there are unsaved changes a dialog box (see figure 7.2.2) asks whether or not these changes should be saved before the editor is closed.

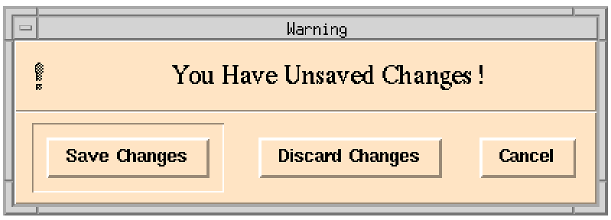

Figure 7.2.2: Unsaved changes window of the B(PN)^2 editor.

- Edit
  - ▷ **Cut**: Cuts the current selection (a part of the text marked using the left mouse button). It is saved to the clipboard and can be used for further **paste** or **export** operations.
  - ▷ **Copy**: Copies the current selection to the clipboard.
  - ▷ **Paste**: Pastes the contents of the clipboard to the current cursor position.

- Edit (continued)

  ▷ **Add to Multi-Selection**: Sometimes it is necessary to select more than one items (e.g. 2 actions corresponding to one transition in the Petri box). After selecting one of them with the left mouse button it can be added to the current multi-selection either by a click on the right mouse button or by this menu entry. A multi-selection of more than one item can be used for the operations in the **Reference** menu explained on page 32, but not for **Copy** or **Paste** operations.

  ▷ **Clear Multi-Selection**: Clears a multi-selection.

  ▷ **Search**: Opens the dialog box (shown in figure 7.2.3) that offers a comfortable searching facility described in the next paragraph.

  ▷ **Replace**: Opens the dialog box (shown in figure 7.2.4) that offers a comfortable search and replace facility described in the next paragraph.

7.2.1 Search

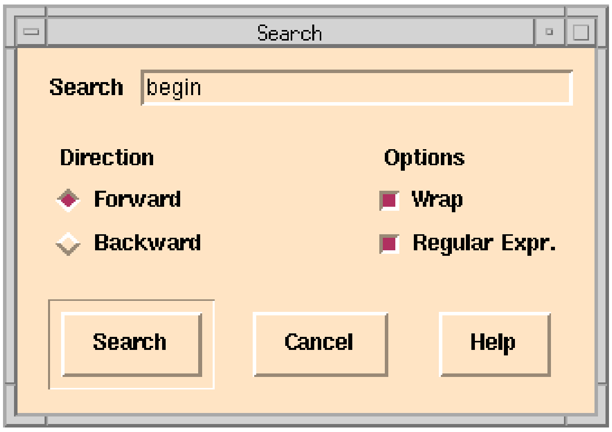

Figure 7.2.3: Search window within the B(PN)^2 editor.

- The user can enter the search term.
- The direction of the search can be specified.
- The behaviour of the search algorithm is controlled by some options:
  - Either the search stops at the end of the text or it is continued at the beginning.
  - The search algorithm accepts regular expressions described on page 31.

7.2.2 Replace

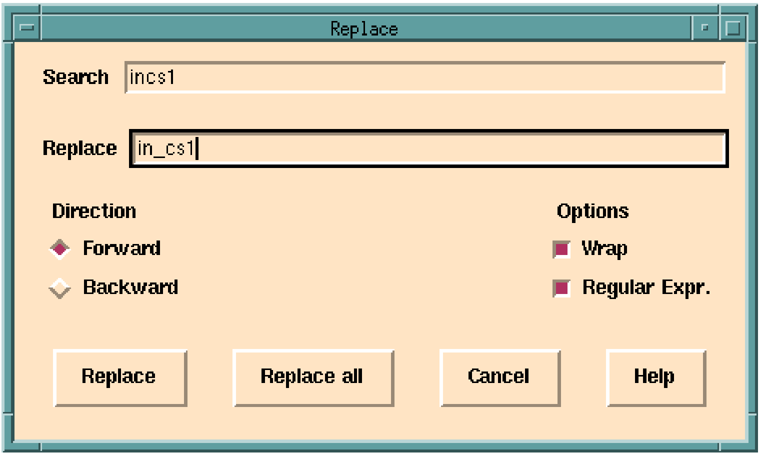

Figure 7.2.4: Replace window within the B(PN)^2 editor.

- The user can enter a term that is to be replaced.
- The user can enter the term that will be inserted instead.
- The direction of the search can be chosen.
- The behaviour of the search algorithm is controlled by some options:
  - Either the search stops at the end of the text or it is continued at the beginning.
  - The search algorithm accepts regular expressions (described below).

7.2.3 Regular expressions

.	Matches any character.
*	Matches zero or more characters. (Repetition operator)
+	Matches one or more characters. (Repetition operator)
?	Matches zero or one character.
( )	Groups a sub-pattern. Subsequent repetition operators or enclosing alternation operators apply to the whole sub-pattern.
|	Alternation operator.
[ ]	Delimits a set of characters. Ranges are specified as x–y. (E.g. [`ae-g`] searches for `a`, `e`, `f` or `g`). If the first character (after the opening braket) in the set is `^`, then there is a match if the remaining characters (whithin the brackets) are not present. (E.g. [`^ae-g`] searches for characters that are not equal to `a`, `e`, `f` or `g`).
^	Represents the beginning of a line. (Only when it is the first character of the search string.)
$	Represents the end of a line. (Only when it is the last character of the search string.)

A powerful way to express a search string is by specifying a regular expression. It has a general pattern specification syntax including the ability to extract substrings from the matching string. A pattern is a sequence of a literal character, a matching character, a repetition clause, an alternation clause, or a sub-pattern grouped by parentheses. The syntax (which is not necessarily the most convenient one, but which is predetermined by Tcl/Tk) is summarized in the previous table: If you want to search a string containing one of the characters in the left column of this table you have to preface it with a backslash.

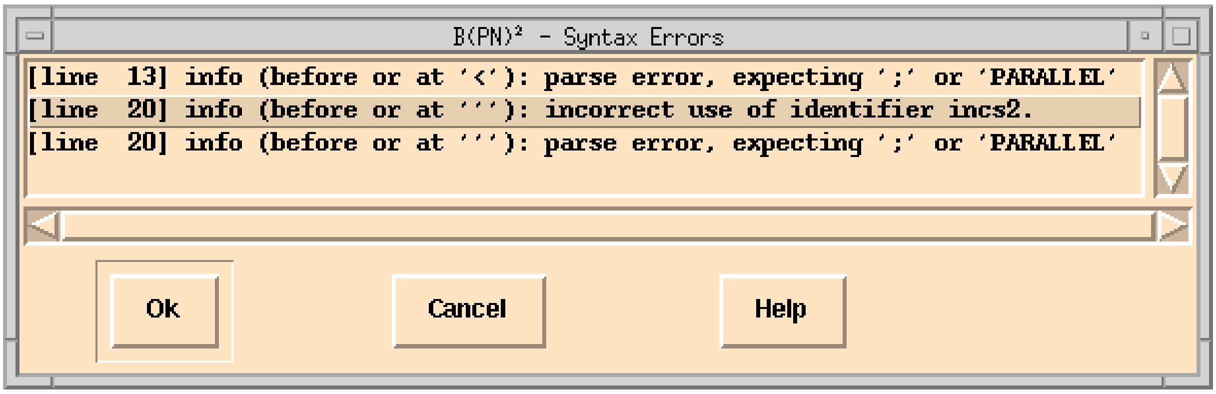

Figure 7.2.5: Syntax check window within the B(PN)^2 editor.

- Action
  - ▷ **Syntax Check**: Invokes an algorithm checking whether or not the B(PN)^2 program is syntactically correct. If you already edited the program after the last saving, a dialog box asks you whether the current version should be saved first or the check should be cancelled. If there is no syntax error a small box displays success. Otherwise the dialog box shown in figure 7.2.5 explains the errors and allows their correction. Each line gives a description of one error. A double-click on one of the lines positions the cursor of the B(PN)^2 editor at the position where the error occurred.

7.2.4 References

The B(PN)^2 editor includes a facility for determining which place in a Petri box corresponds to a given point in the control flow of the associated B(PN)^2 program, or which transition(s) in the Petri box correspond(s) to one or more action(s) in the program.

After selecting, e.g., an action in the B(PN)^2 program, two of the entries in the **Reference** menu can be used to paste the name(s) of the corresponding transition(s) to the formula editor. In this way temporal logic formulae can be made to refer to points of the control flow or actions (using macros like `LIVE`) of the program, even though technically speaking, the temporal logic refers only to places of a net.

If a transition in the Petri box belongs to more than one action in the B(PN)^2 program (i.e., is a synchronisation), then there are two possibilities to refer to this transition. Either you select only one of these actions and use the menu entry **Show Transition in Formula (partial)** or you select all these actions with a multi-selection and use the menu entry **Show Transition in Formula**.

A good way to become familiar with all these kinds of references between a B(PN)^2 program and a net is to use the menu entry **Show Reference** (in combination with the option **Highlight Execution** in one of the net editors (see pages 40 and 48).

- Reference
  - ▷ **Show Place in Formula**: Pastes the name of the place (of the Petri box) corresponding to the current selection (in the $B(PN)^2$ editor) to the formula editor (at the latter's current cursor position), provided there is a reference from the selection to a place of the net.
  - ▷ **Show Transition(s) in Formula (partial)**: Pastes the name(s) of the transition(s) (of the Petri box) corresponding to the current selection (in the $B(PN)^2$ editor) to the formula editor (at the latter's current cursor position), provided there is a reference from the selection to at least one transition of the net. It is sufficient for the transition(s) to belong to more actions than the selected ones.
  - ▷ **Show Transition(s) in Formula**: Pastes the name(s) of the transition(s) (of the Petri box) corresponding to the current selection (in the $B(PN)^2$ editor) to the formula editor (at the latter's current cursor position), provided there is a reference from the selection to at least one transition of the net. It is **not** sufficient for the transition(s) to belong to more actions than the selected ones.
- Info
  - ▷ **Info**: Shows the about box for the text editors integrated in the **PEP** tool.

8 The high level Petri net (M-net) editor

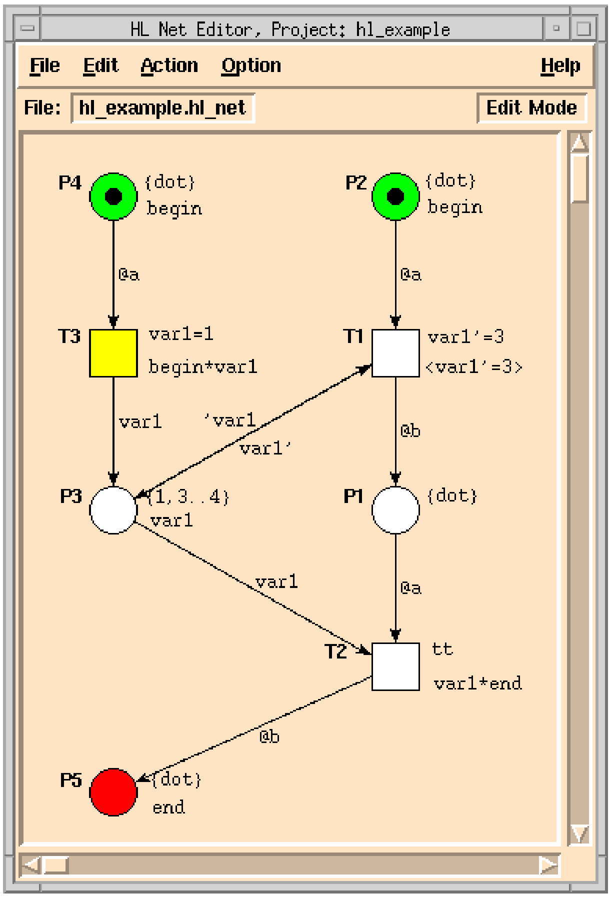

Figure 8.1: High level net editor.

8.1 Description of the high level Petri net editor window

- The high level net editor is a Petri net editor with an integrated simulator offering some M-net specific features.
- The main part of the editor window can be used as a drawing canvas for creating and editing M-nets.
- The functionality of the menu bar is described in the next section.
- The name of the currently edited M-net is displayed underneath the menu bar on the left hand side.
- The current mode of the net editor is displayed underneath the menu bar on the right hand side.
- If the editor is in *Edit Mode* the following can be done:
  - Places can be created by pressing the **p**-key at the desired position of the drawing canvas.
  - Transitions can be created by pressing the **t**-key at the desired position of the drawing canvas.

- Arcs can be created by pressing the **a**-key first and selecting the place (transition) and the transition (place, resp.) with the left mouse button afterwards.
  Note: If you click to quickly, the first node may jump towards the second node.
- Comments can be created by pressing the **c**-key at the desired position of the drawing canvas. The dialog box shown in figure 8.1.5 is opened to edit the text.
- An object (place, transition, arc, comment or annotation) can be selected by a click with the left mouse button.
- A selected object can be moved by holding the left button down while moving the mouse.
- A multi-selection of places and transitions can be extended by a click on another place or transition while the **shift** key is held down.
  * A multi-selection can be deleted by pressing the **del** key.
  * A multi-selection can be moved by holding down the left mouse button (after its last extension) while moving the mouse.
- An area can be selected by the middle mouse button. (If you have a mouse with only two buttons you can probably emulate this by clicking both buttons at the same time).
  * An area can be deleted by pressing the **del** key.
  * An area can be moved by the left mouse button (after clicking somewhere in the area without hitting an object).
- The place dialog window shown in figure 8.1.2 can be opened by a double-click with the left mouse button on a place. Thus the name, the meaning, the type, the initial and the current marking and the status of the place can be changed. The name must be unique w.r.t. the other places of the net; this is checked by the editor.
  Note: The (global) current marking must not necessarily be reachable from the (global) initial marking.
- The transition dialog window shown in figure 8.1.3 can be opened by a double-click with the left mouse button on a transition. Thus the name, the meaning, the action term and the value term of the transition can be changed. The name must be unique w.r.t. the other transitions of the net; this is checked by the editor.
- The arc dialog window shown in figure 8.1.4 can be opened by a double-click with the left mouse button on an arc. Thus the multiset of variables of the arc can be changed. The unique name is generated automatically and cannot be changed.

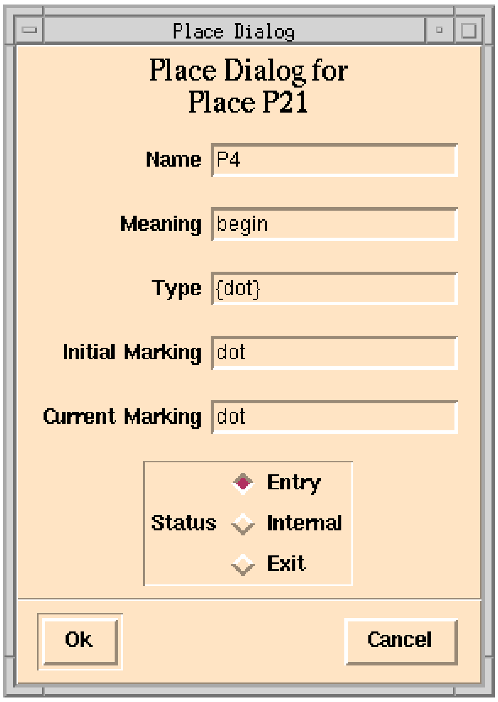

Figure 8.1.2: Place dialog window.
If a net is compiled out of a B(PN)^2 program the meaning gives the reference. A place corresponding to a variable is labelled with the name of this variable, a place corresponding to a point in the control flow can be labelled with `do`, `begin`, `end` or a semicolon.
Types can be given as, e.g., `{dot}`, `{tt,ff}` or `{1,3..5}`, with `tt` representing *true* and `ff` *false* respectively. Cartesian products are allowed, e.g. `{1..4} X {dot,1..2}`.
A marking must be a multi-set over `Type`. E.g., `3*1,4` specifies a marking `1 1 1 4`. 2-tupels can be specified, e.g.
`(1,dot),(2,2),(3,1)`.

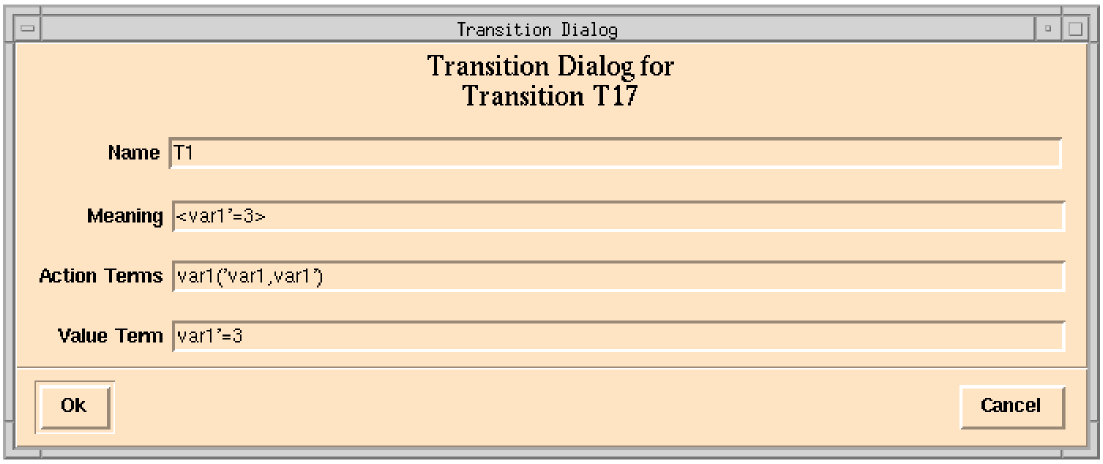

Figure 8.1.3: Transition dialog window.
If a net is compiled out of a B(PN)^2 program the meaning gives the reference. Editing it might destroy its meaning.
Action terms can be given as, e.g.,
`var1('var1,var1')`, `^var1('var1,var1')`, `chan1!(chan1!)` or `^stack1?(stack1?)`
(see D.1 for explanation). You can get some hints if you compile a program into a HL net with the option **no scoping**.
Value terms can be given as, e.g.,
`var1'=ff & var2'=3`.
Sometimes it is necessary to specify (explicitly) the range of a variable by prefixing the value term with, e.g.,
`var1{1,3..5} & var2{tt,ff}:`.
You can also use projections of 2-tupels like in:
`chan1!{1,3..5}: pr0(chan1')=1 & pr1(chan1')=chan1!`
but you might get some errors because the syntax is very restricted. You can get some hints if you compile a program into a HL net.
<u>Note</u>: Within a value term a variable `var1'` is not at all related to a variable `var1` or `'var1`.

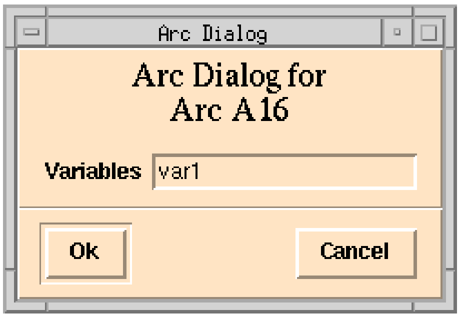

Figure 8.1.4: Arc dialog window.
A multi-set of variables can be specified. E.g. `var1,3*var2` is an abbreviation for `var1,var2,var2,var2`.
Note: The current version of the net editor only handles one single variable per arc correctly, which is sufficient for HL nets generated from a B(PN)^2 program. Otherwise there might be wrong occurences of transitions during the simulation.

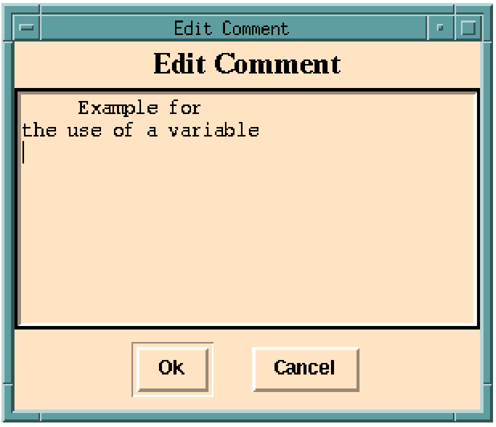

Figure 8.1.5: Comment dialog window.

8.2 Menus of the high level net editor

The pull-down menus offer further functionality.

- File
  - ▷ **Open**: Opens the file browser to select an M-net.
  - ▷ **Reload**: Reloads the M-net.
  - ▷ **Save**: Saves the contents of the net editor to the file whose name is displayed underneath the menu bar.
  - ▷ **Save as**: Opens the file browser to save the contents of the net editor to another file.
  - ▷ **Delete Net**: Clears the editor (i.e., deletes the current net but not the file).
  - ▷ **Export to PostScript**:
    - ○ **All Objects**: A file browser is opened to choose the name of a file to which all objects (places, transitions, arcs, currently displayed kinds of annotations and comments) are exported in PostScript format. The net examples in the appendix have been produced in this way.
    - ○ **Selection**: A file browser is opened to choose the name of a file to which all currently selected objects (places, transitions, arcs, currently displayed kinds of annotations and comments) are exported in PostScript format.
    - ○ **Entire Screen**: A file browser is opened to choose the name of a file to which all objects (places, transitions, arcs, currently displayed kinds of annotations and comments) that are currently visible on the screen are exported in PostScript format.
  - ▷ **Quit**: Closes the net editor. If there are unsaved changes a dialog box asks whether or not these changes should be saved.
    <u>Note</u>: Some changes like moving places or changing a current marking are not recognized. If you would like to keep these changes you should use the menu entry **Save** first.
- Edit
  - ▷ **New Place**: Switches the mode of the net editor to place insertion. The shape of the cursor changes to an internal place with type `{dot}`. The user can create such a new place by clicking at the desired position of the drawing canvas.
  - ▷ **New Transition**: Switches the mode of the net editor to transition insertion. The shape of the cursor changes to a transition. The user can create such a new transition (with empty action term `{}` and value term `true`) by clicking at the desired position of the drawing canvas.

- Edit (continued)
  - ▷ **New Arc**: Switches the mode of the net editor to arc insertion. The user can determine first the start node and then the end node of the arc by clicking at the desired place (transition) and transition (place, resp.).
  - ▷ **Delete Elements**: Deletes the currently selected elements.
  - ▷ **Select All**: Selects all objects (places, transitions and arcs together with their annotations).
    Note: A selection can be de-selected by a click with the middle mouse button at any position of the drawing canvas.
- Action
  - ▷ **Active Transition**: Colours all currently enabled transitions yellow. The mode can be exited by pressing the **Esc** key.
  - ▷ **Execute**: Switches the mode of the net editor so that all currently enabled transitions are coloured yellow. They can be executed by a mouse click. The behaviour depends on the option **Random Bindings**. The mode can be exited by pressing the **Esc** key.
  - ▷ **Reset to Initial Marking**: Resets the marking to the initial marking from which a new simulation of the net can be started.
  - ▷ **Animate**: Switches the mode of the net editor so that an automatic simulation is started. This menu entry is only enabled if the option **Random Bindings** is switched on. During the simulation all currently enabled transitions are coloured yellow. One of them is chosen randomly together with a possible binding for the variables, and is executed. The surrounding arcs are coloured green to show the token flow. The mode can be exited by pressing the **Esc** key.
  - ▷ **Show Reference**: If the M-net belongs to a $B(PN)^2$ program, the correspondence between a place of the net and a point in the control flow of the $B(PN)^2$ program or between a transition in the net and actions in the $B(PN)^2$ program can be demonstrated. The mode of the net editor is switched so that the corresponding text segment is highlighted in aquamarine if the user clicks on a transition or a place. The mode can be exited by pressing the **Esc** key.

- Option
  - ▷ **Display Names**: Chooses whether or not the names of places and transitions are displayed.
  - ▷ **Display Meanings**: Chooses whether or not the meanings of places and transitions are displayed.
  - ▷ **Display Action Terms**: Chooses whether or not the action terms of transitions are displayed.
  - ▷ **Display Value Terms**: Chooses whether or not the value terms of transitions are displayed.
  - ▷ **Display Types**: Chooses whether or not the types of places are displayed.
  - ▷ **Show Invisible Arcs**: Chooses whether or not invisible arcs are displayed. An invisible flag is set by the compiler 'B(PN)$^2\Rightarrow$M-net' for arcs beetwen the data boxes of the net and the control flow part of the net to enhance the readability of the constructed nets (see page 23).
  - ▷ **Gridding**: Chooses whether or not places and transitions have to be aligned along an underlying grid.
  - ▷ **Highlight Execution**: If the M-net belongs to a B(PN)^2 program, the correspondence between a place in the net and a point in the control flow of the B(PN)^2 program or between a transition in the net and actions in the B(PN)^2 program can be demonstrated. The mode of the net editor is switched so that if a transition is executed, the corresponding text segment is highlighted in aquamarine in the B(PN)^2 editor.
  - ▷ **Random Bindings**: Chooses whether a random binding is calculated and taken if a transition is executed, or whether the dialog box shown in figure 8.2.6 is opened to allow the user to test and choose a binding.
  - ▷ **Zoom**: Switches amongst different zoom factors.

Note: The status of a place is always displayed by appropriate colours. Entry places are coloured green, exit places are coloured red and internal places are uncoloured.

- Help
  - ▷ **Info**: Shows an about box.

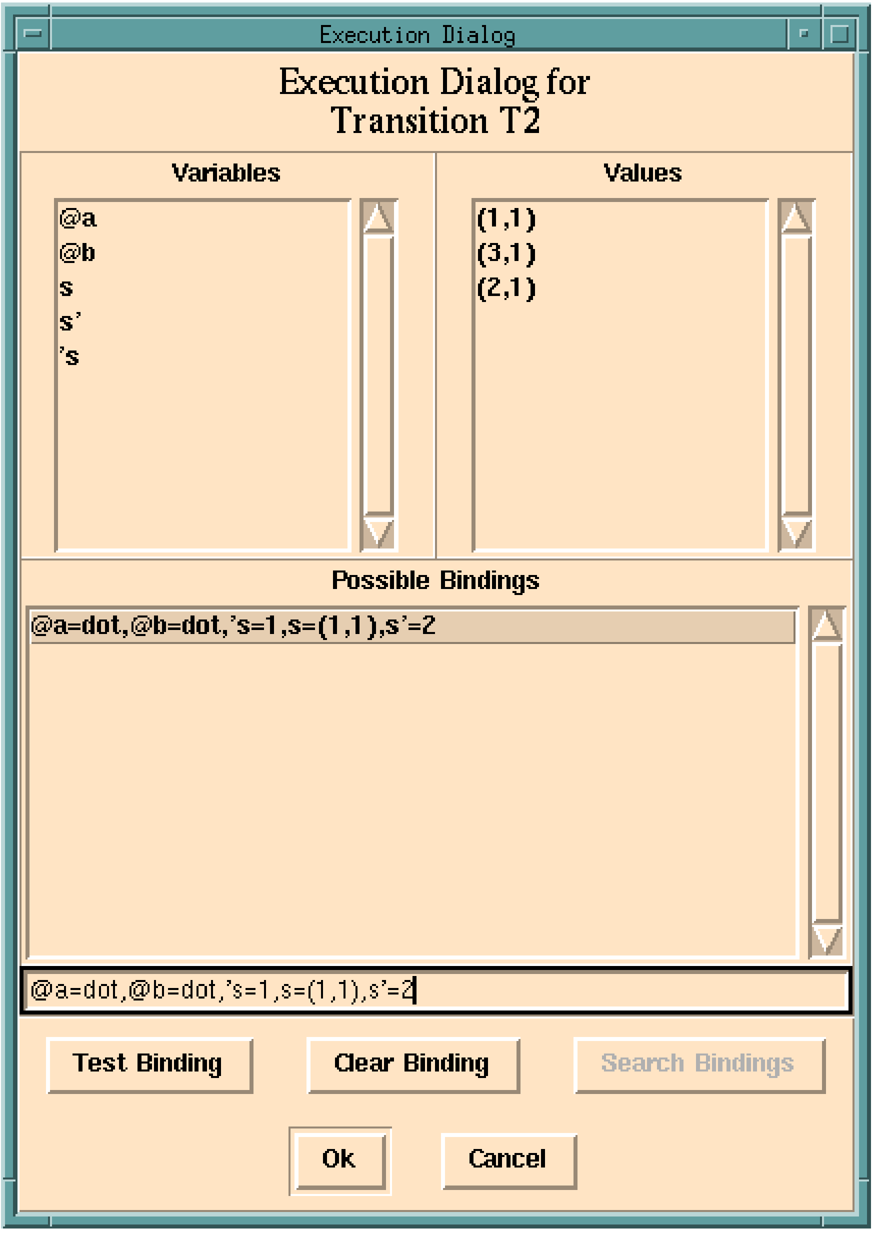

Figure 8.2.6: Execution dialog window.
This execution dialog is opened if the option **Random Bindings** is switched off. If you click on the **Search Bindings** button possible bindings are calculated and the part of the window displaying these bindings is opened. You can select one of these bindings by clicking on it. If a binding is displayed in the bottom of the window it can be tested using the **Test Bindings** button or cleared using the **Clear Bindings** button. The transition is executed with the currently chosen binding if you click on the **OK** button. You can also create a binding by clicking first on a variable name and then on a value.

9 The low level Petri net (Petri box) editor

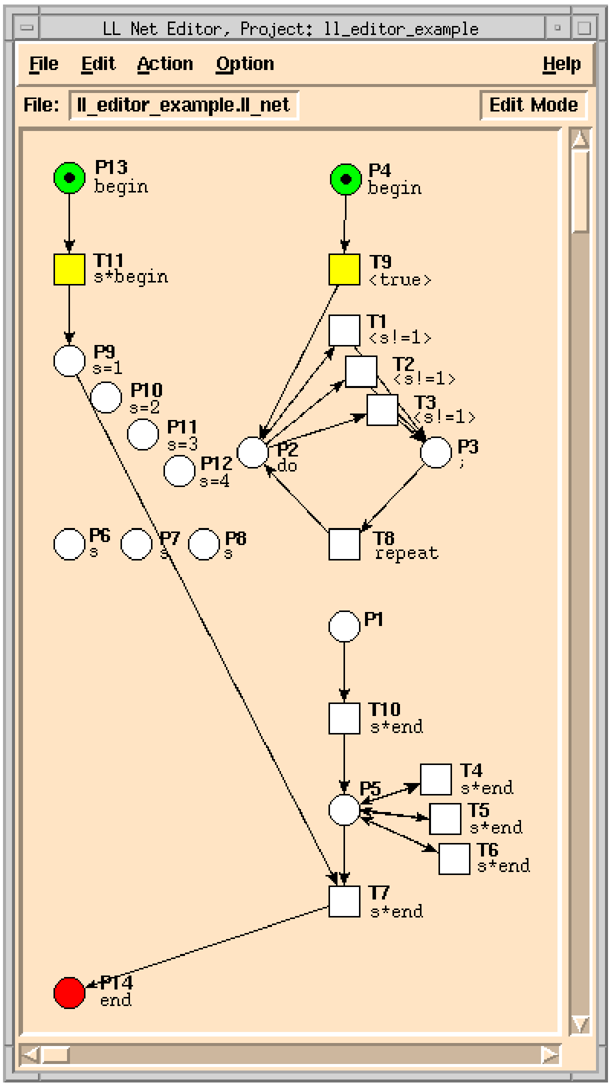

Figure 9.1: Low level net editor.

LL nets are special HL nets, esentially by having all places type `{dot}` and all transition value terms `true`. The windows and dialogs of the LL net editor correspondingly simplified (restricted) versions of those of the HL net editor.

9.1 Description of the low level Petri net editor window

- The low level net editor is a Petri net editor with an integrated simulator offering some Petri box specific features.
- The main part of the editor window can be used as a drawing canvas for creating and editing low level Petri nets.
- The functionality of the menu bar is described in the next section.
- The name of the currently edited LL net is displayed underneath the menu bar on the left hand side.
- The current mode of the net editor is displayed underneath the menu bar on the right hand side.

- If the editor is in *Edit Mode* the following can be done:
  - Places can be created by pressing the **p**-key at the desired position of the drawing canvas.
  - Transitions can be created by pressing the **t**-key at the desired position of the drawing canvas.
  - Arcs can be created by pressing the **a**-key first and selecting the place (transition) and the transition (place, resp.) with the left mouse button afterwards.
    Note: If you click to quickly, the first node may jump towards the second node.
  - Comments can be created by pressing the **c**-key at the desired position of the drawing canvas. The dialog box shown in figure 9.1.2 is opened to edit the text.
  - An object (place, transition, arc, comment or annotation – like name or meaning) can be selected by a click with the left mouse button.
  - A selected object can be moved by holding the left button down while moving the mouse.
  - A multi-selection of places and transitions can be extended by a click on another place or transition while the **shift** key is held down.
    - A multi-selection can be deleted by pressing the **del** key.
    - A multi-selection can be moved by holding down the left mouse button (after its last extension) while moving the mouse.
  - An area can be selected by enframing it using the middle mouse button. (If you have a mouse with only two buttons you can probably emulate this by clicking both buttons at the same time).
    - An area can be deleted by pressing the **del** key.
    - An area can be moved by the left mouse button (after clicking somewhere in the area without hitting an object).
  - The place dialog window shown in figure 9.1.3 can be opened by a double-click with the left mouse button on a place. Thus the name, the meaning, the initial and the current marking and the status of the place can be changed. The name must be unique w.r.t. the other places of the net; this is checked by the editor.
    Note: The (global) current marking must not necessarily be reachable from the (global) initial marking.
  - The transition dialog window shown in figure 9.1.4 can be opened by a double-click with the left mouse button on a transition. Thus the name and the meaning of the transition can be changed. The name must be unique w.r.t. the other transitions of the net; this is checked by the editor.
  - The arc dialog window shown in figure 9.1.5 can be opened by a double-click with the left mouse button on a arc. Thus the weight of the arc can be changed. The weight of the arc (in terms of LL nets) is the cardinality of the variable multiset of this arc (in terms of HL nets). The unique name is generated automatically and cannot be changed.

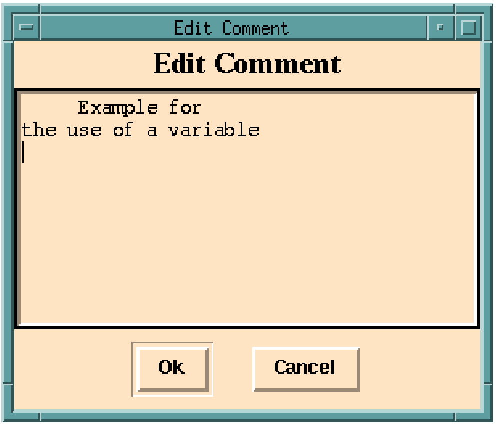

Figure 9.1.2: Comment dialog window.

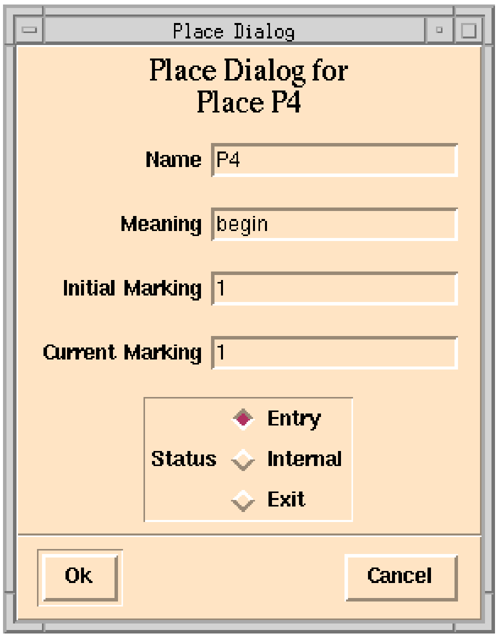

Figure 9.1.3: Place dialog window.

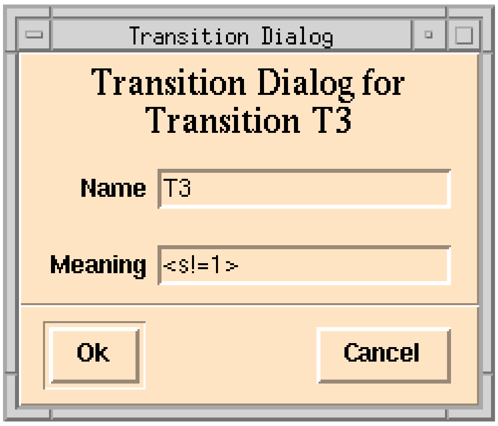

Figure 9.1.4: Transition dialog window.

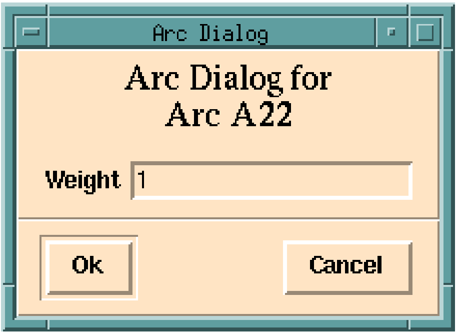

Figure 9.1.5: Arc dialog window.

9.2 Menus of the low level net editor

The pull-down menus offer further functionality.

- File
  - ▷ **Open**: Opens the file browser to select a low level Petri net.
  - ▷ **Reload**: Reloads the low level Petri net.
  - ▷ **Save**: Saves the contents of the net editor to the file whose name is displayed underneath the menu bar.
  - ▷ **Save as**: Opens the file browser to save the contents of the net editor to a file.
  - ▷ **Delete Net**: Clears the editor (i.e., deletes the current net but not the file).
  - ▷ **Export to PostScript**:
    - ∘ **All Objects**: A file browser is opened to choose the name of a file to which all objects (places, transitions, arcs, currently displayed kinds of annotations and comments) are exported in PostScript format. The net examples in the appendix have been produced in this way.
    - ∘ **Selection**: A file browser is opened to choose the name of a file to which all currently selected objects (places, transitions, arcs, currently displayed kinds of annotations and comments) are exported in PostScript format.
    - ∘ **Entire Screen**: A file browser is opened to choose the name of a file to which all objects (places, transitions, arcs, currently displayed kinds of annotations and comments) that are currently visible on the screen are exported in PostScript format.
  - ▷ **Quit**: Closes the net editor. If there are unsaved changes a dialog box asks whether or not these changes should be saved.
    Note: Some changes like moving places or changing a current marking are not recognized. If you would like to keep these changes you should use the menu entry **Save** first.

- Edit
  - ▷ **New Place**: Switches the mode of the net editor to place insertion. The shape of the cursor changes to an internal place without a marking. The user can create such a new places by clicking at the desired position of the drawing canvas.
  - ▷ **New Transition**: Switches the mode of the net editor to transition insertion. The shape of the cursor changes to a transition. The user can create such a new transition by clicking at the desired position of the drawing canvas.
  - ▷ **New Arc**: Switches the mode of the net editor to arc insertion. The user can determine the start node and afterwards the end node of the arc by clicking at the desired place (transition) and transition (place, resp.).
  - ▷ **Delete Elements**: Deletes the currently selected elements.
  - ▷ **Select All**: Selects all objects (places, transitions and their arcs together with their annotations) of the net.
    Note: A selection can be de-selected by a click with the middle mouse button at any position of the drawing canvas.
  - ▷ **Create Comment**: Switches the mode of the net editor to comment insertion. The user can create a comment by clicking at the desired position. The dialog box shown in figure 9.1.2 is opened to edit the text.
- Action
  - ▷ **Active Transition**: Colours all currently enabled transitions yellow. The mode can be exited by pressing the **Esc** key.
  - ▷ **Execute**: Switches the mode of the net editor so that all currently enabled transitions are coloured yellow. The user can execute them by a mouse click. The mode can be exited by pressing the **Esc** key.
  - ▷ **Reset to Initial: Marking**: Resets the marking to the initial marking from which a new simulation of the net can be started.
  - ▷ **Animate**: Switches the mode of the net editor so that an automatic simulation is started. During the simulation all currently enabled transitions are coloured yellow. One of them is chosen randomly and is executed. The surrounding arcs are coloured green to show the token flow. The mode can be exited by pressing the **Esc** key.

- Action (continued)
  - ▷ **Show Reference**: If the low level net is a Petri box that has been compiled out of the B(PN)^2 program via an M-net the correspondence between a place of the net and a point in the control flow of the B(PN)^2 program or between a transition in the net and actions in the B(PN)^2 program can be demonstrated. The mode of the net editor is switched so that the corresponding text segment is highlighted in aquamarine if the user clicks on a transition or a place. The mode can be left by pressing the **Esc** key.
- Option
  - ▷ **Display Names**: Chooses whether or not the names of places and transitions are displayed.
  - ▷ **Display Meanings**: Chooses whether or not the meanings of places and transitions are displayed.
  - ▷ **Gridding**: Chooses whether or not places and transitions have to be aligned along an underlying grid.
  - ▷ **Show Invisible Arcs**: Chooses whether or not invisible arcs are displayed. An invisible flag is (can be) set by the compiler 'B(PN)^2⇒M-net' ('PBC⇒LL Net', resp.) for arcs beetwen the data boxes of the net and the control flow part of the net to enhance the readability of the constructed nets (see pages 21 and 23).
  - ▷ **Highlight Exexution**: If the low level net is a Petri box that has been compiled out of the B(PN)^2 program via an M-net, the correspondence between a place in the net and a point in the control flow of the B(PN)^2 program or between a transition in the net and actions in the B(PN)^2 program can be demonstrated. The mode of the net editor is switched so that if a transition is executed, the corresponding text segment is highlighted in aquamarine in the B(PN)^2 editor.
  - ▷ **Zoom**: Switches amongst different zoom factors.

Note: The status of a place is always displayed by appropriate colours. Entry places are coloured green, exit places are coloured red and internal places are uncoloured.

- Help
  - ▷ **Info**: Shows an about box.

10 The formula editor

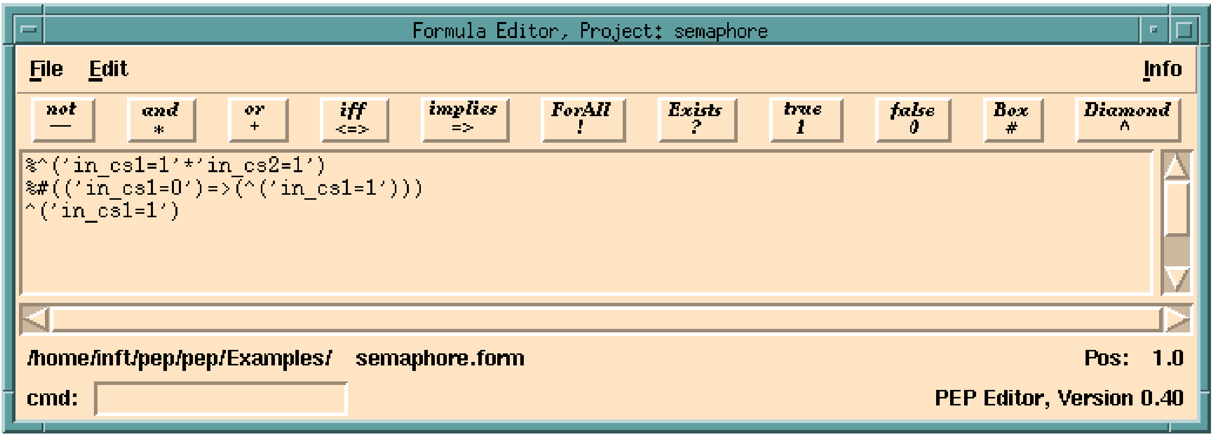

Figure 10.1: Formula editor.

10.1 Description of the formula editor window

- The formula editor is a text editor with some temporal logic formula specific features.
- In the main part of the editor window formulae can be edited.
- If the keyboard offers this possibility the **cut**, **copy** and **paste** keys can be used.
- The icons underneath the pull-down menu (except the far right one) allow to type parts of a formula by clicking on them. Thus the user need not be familiar with the abbreviations of the operators. The following table lists, for each button, the mathematical symbol together with the characters that are inserted if you click on the button.

Button	Inserted characters	Mathematical symbol
not	–	$\neg$
and	*	$\wedge$
or	+	$\vee$
iff	<=>	$\Leftrightarrow$
implies	=>	$\Rightarrow$
Forall	!	$\forall$
Exists	?	$\exists$
True	1	**true**
False	0	**false**
Box	#	$\Box$ – **always**
Diamond	^	$\Diamond$ – **possibly**

- Formulae involve place names of a net (e.g., `# Place-name`). Furthermore variable bindings can be used (e.g., `# 'Variable-name = Value'`). They are automatically transformed into place names. Brackets can be used to structure formulae (i.e. to overrule priorities of the operators). You can keep different formulae within one formula file if you prefix the currently unused formulae with an % (see figure 10.1).

Note: Using characters as in the middle column of the above table is a compromise because formulae must also be interpretable as ASCII text.

10.2 Menus of the formula editor

- File
  - ▷ **Open**: Opens the file browser to select a formula file.
  - ▷ **Reload**: Reloads the formula file.
  - ▷ **Save**: Saves the contents of the formula editor to the file whose name is displayed underneath the main part of the window.
  - ▷ **Save as**: Opens the file browser to save the contents of the formula editor to another file.
  - ▷ **Import File**: Opens the file browser to select the name of a file to be loaded in the formula editor at the current cursor position.
  - ▷ **Export Clipboard**: Opens the file browser to select the name of another file. The contents of the clipboard are saved to this file. You can either use the **copy** button on your keyboard or the menu entry **Edit**▷**Copy** to copy the selection to the clipboard.
  - ▷ **Quit**: Closes the formula editor. If there are unsaved changes a dialog box asks whether or not these changes should be saved.
- Edit
  - ▷ **Cut**: Cuts the current selection. It is saved to the clipboard and can be used for further **paste** operations.
  - ▷ **Copy**: Copies the current selection to the clipboard.
  - ▷ **Paste**: Pastes the contents of the clipboard to the current cursor position.
  - ▷ **Search**: Opens a dialog box as shown in figure 7.2.3 on page 30.
  - ▷ **Replace**: Opens a dialog box as shown in figure 7.2.4 on page 31.
- Info
  - ▷ **Info**: Shows the about box for the text editors integrated in the **PEP** tool.

A A user's view of PEP

This section describes various objects used by **PEP** and their relationships as perceived by the system's user. Section A.1 gives a brief overview of **PEP**'s objects; a more detailed description can be found in sections C, D and E, respectively. Section A.2 decribes a typical cycle of usage of **PEP**. Section B depicts the structure of the **PEP** tool from the implementor's point of view. Section E gives some details about the integrated model checking algorithms. Finally an overview over future work is given in section F.

A.1 Objects of PEP

Users can choose primarily between the following types of objects in order to model parallel systems. It is up to him (or her) which object (s)he would like to use initially.

1. Parallel algorithms can be expressed in B(PN)^2 [BH93], which is an imperative / predicative programming / specification language. The atomic actions of this language may contain predicates involving prevalues and postvalues of variables (similar to the sequential languages proposed in [Heh85] or in Z [Spi92]. Basic command connectives of B(PN)^2 are sequential composition, nondeterministic choice, iteration and parallel composition. Communication may be done by shared variables or by channels or shared stacks.

   B(PN)^2 is extended to a complete programming language by a procedure concept [Jen94, Jen95, LP95] and abstract data types [KP94, KP95]. Procedures have been implemented in a limited way in the current **PEP** version. The present version of **PEP** allows channels and stacks as abstract data types.

   The name B(PN)^2 stands for **B**asic **P**etri **N**et **P**rogramming **N**otation. It refers to the fact that B(PN)^2 has a compositional semantics in terms of low level (LL) Petri nets called boxes [BDH92]. Boxes are a class of safe place and transition labelled nets. From a different angle, B(PN)^2 can be viewed as a macro language for a class of structured LL nets.

2. Terms of a process algebra called PBC (**P**etri **B**ox **C**alculus) may be used. The PBC [BDH92, BK95, Dev94] is partly an extension and partly a modification of CCS [Mil89] allowing multisynchronisation. PBC terms may also arise as intermediate objects in a translation from B(PN)^2 to nets. Thus in **PEP** a PBC term can either be derived automatically from a B(PN)^2 program or be designed separately.

3. Labelled LL (low level) Petri nets may be designed. More precisely, the class of nets **PEP** deals with on the low level is the class of (marked) labelled place / transition nets [Rei85]. Boxes are a subclass; they may arise out of a translation from B(PN)^2 programs into LL nets. For instance, boxes are always guaranteed to be safe under their standard initial marking.

4. Labelled HL (high level) nets may be designed. More precisely, the class of high level nets **PEP** deals with is essentially the class of coloured nets [Jen92] with additional place and transition labels needed for compositionality and for the connection with B(PN)^2 programs. M-nets [BFF$^+$95a] are a special subclass; they may arise out of an alternative net semantics of B(PN)^2 programs [BFF$^+$95b].

All four classes of objects mentioned in this list have stand-alone editors in **PEP**. They are also shown in the core of Figure A.1.1. A B(PN)^2 program can thus be compiled into a Petri box in two different ways: using a PBC term as an intermediate step or using an HL net as an intermediate step. The two translations produce equivalent LL nets. The system remembers the fact that, for example, a given PBC term comes from a given B(PN)^2 program. However, if the term is edited then this link is lost. The same is true for links created by other translations.

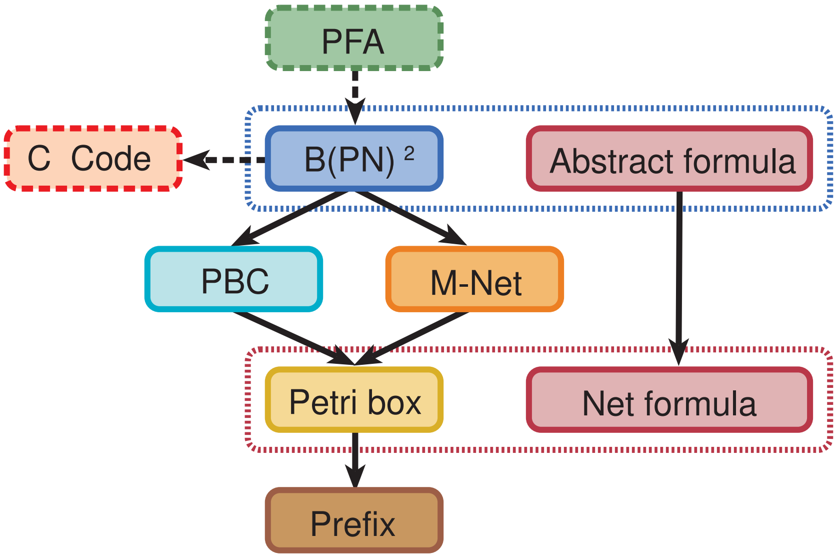

Figure A.1.1: Possible dependencies between the objects of the **PEP** system.

The process of creating and editing the primary objects is normally embedded in a design cycle during which, in addition, further objects may be created and used:

5. The user may (model) check a system property of his (her) own design by defining a set of temporal logic formulae in the modal logic S_4, that is, propositional logic on place names augmented with $\Box$ for 'always' and $\Diamond$ for 'possibly at some future point'. A formula may refer abstractly (indirectly) to a B(PN)^2 program and describe one of its properties. For actual model checking they are translated into a formula referring to the corresponding net, and actual model checking is done (possibly in the background) on the net and the net formula - see also Figure A.1.1 where this relationship is shown schematically.

6. Model checking in **PEP** [Esp93, Gra95b] has two inputs: a net and a formula. It is based on the principle of tilting the inevitable computational complexity of the algorithms as much as possible towards the size of the formula rather than the size of the net (which normally has an exponential state space). Thus, **PEP**'s model checkers are comparably efficient in the size of the net but comparably inefficient in the size of the formula. In particular, none of the model checkers needs to construct the state space even partially. Instead, it may become necessary to calculate the finite prefix [McM92, ERV95] of a branching process of an existing LL net (see also the bottom of Figure A.1.1).

7. Instead of writing a B(PN)^2 program directly, **PEP** also has input filters in order to translate programs or specifications written in other languages into B(PN)^2. Figure A.1.1 shows one such filter called **PFA**. It accepts a Parallel Finite Automaton as input and yields a B(PN)^2 program as output [GMA95]. This option is, however, not integrated in the present version of **PEP**.

8. C code may be generated from a $B(PN)^2$ program (see Figure A.1.1). This option is, however, disabled in the present version of **PEP**.

During a design cycle, typically, a $B(PN)^2$ program is written, the corresponding high-level net and the low-level Petri box are compiled automatically, the prefix is calculated and interesting properties are expressed by formulae and are checked. Naturally, **PEP** allows to add documentation at various points of the design cycle.

Section A.2 explains the design cycle in more detail. All objects relating to a single design cycle are collected into a machine object called project. In addition, a project may contain analysis and verification results or runtime documentation. Projects are the basic entities handled by the **PEP** system even though a project may correspond to several different files on the underlying operating system (with a few exceptions there is a file for each object defined above).

A.2 A typical design cycle using PEP

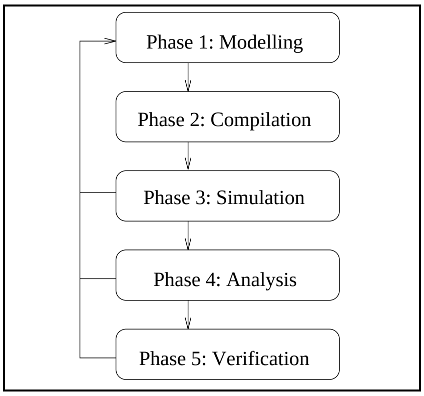

Figure A.2.2: Typical **PEP** cycle of usage.

Figure A.2.2 illustrates five main phases during typical usage of the **PEP** tool.

1. Usually the user wishes to realise a model of a parallel system. As described above, (s)he can use parallel programs ($B(PN)^2$ programs), terms in a process algebra (PBC programs), or high-level or low-level Petri nets.
2. Depending on the initial object and depending on the route of compilation (see Figure A.1.1), other objects can be created by a compilation process (or several compilation processes).
3. A simulation module allows the user to become familiar with the parallel system under consideration. This can give valuable hints but no proof of correctness. Simulation can be done directly in terms of HL and LL nets and indirectly in terms of $B(PN)^2$ programs, exploiting references between (either HL or LL) nets and programs.
4. Some specific information can be gained by the application of standard analysis algorithms. Particularly in the case of ordinary Petri nets, for instance, it can be of great interest whether a net is live or safe.
5. A focal point is the verification phase. Checking whether one made a mistake during the modelling phase may be aided by verifying various properties. For example, it is possible to check whether the mutual exclusion property holds for a program without

consideration of the corresponding Petri box. LL nets do not have to be on-screen for verification to be possible. Thus, typically, the user sees the program (and perhaps the HL net) while verification is performed on the corresponding LL net (and associated prefix) stored on disk.

B An implementor's view of PEP

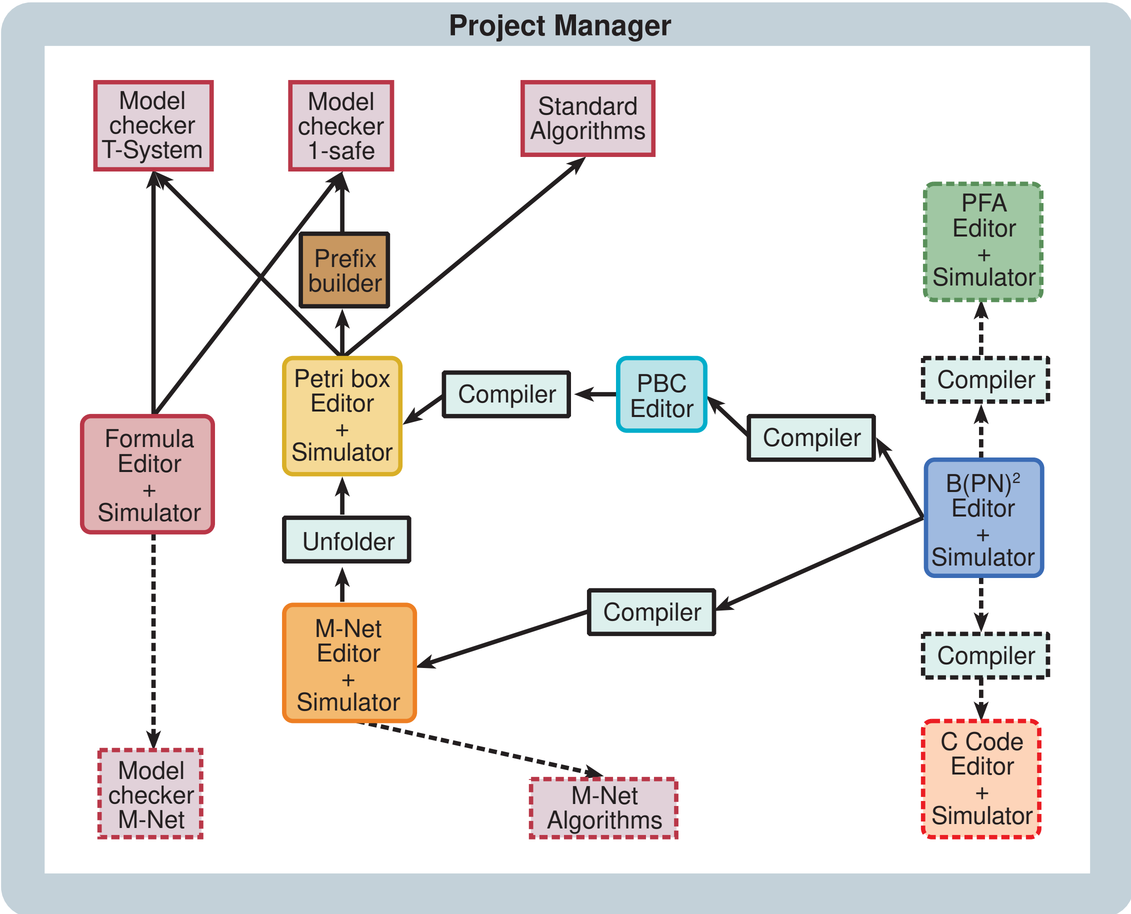

Figure B.1: Interplay between **PEP** system components. Not yet integrated parts of the system are indicated by dashed lines and boxes. The arrows represent input/output relations between components.

Corresponding to the five phases described in section A.2, **PEP** as viewed by its implementors consists of the following classes of components:

1. Editors for $B(PN)^2$ programs, PBC terms, low-level nets, high-level nets, formulae and project documentation.
2. Compilers as follows:
   (a) $B(PN)^2$ program $\Rightarrow$ PBC term.
   (b) PBC term $\Rightarrow$ box (LL net).
   (c) LL net $\Rightarrow$ prefix.
   (d) $B(PN)^2$ program $\Rightarrow$ M-net (HL net).

(e) HL net ⇒ LL net.

Two additional compilers - which are not yet fully integrated in the tool - generate C code from B(PN)^2 programs [Mel95a, Mel95b] or B(PN)^{2}programs from parallel finite automata [GMA95].

3. Simulators for HL and LL boxes [Him95]. Simulation of a B(PN)^2 program can be triggered by simulation of a corresponding net based on references between either actions of the B(PN)^2 program and transitions of the corresponding nets or points in the control flow of the different processes and places of the nets.
4. Standard algorithms (checking free choice / T-system properties, deadlockfreeness, liveness, reachability and reversibility). The algorithms for liveness and reversibility are not integrated in the current version of the **PEP** tool, because they are using the commercial tool CPLEX.
5. Three model checking algorithms, MC-0 for T-systems [BE92, Thi93] and MC-1 and MC-2 [Esp93, Gra95b] for safe LL nets, which can determine whether a Petri net satisfies a property given in terms of a temporal logic formula. The transparency and full functionality of the verification component is obtained by a formula transformer capable of transforming an abstract formula (such as 'if a program terminates', 'whether at some point a variable has a certain value', 'if some processes are at the same time at certain points in the control flow' or 'if an action is live') into a formula referring to the corresponding Petri box. This offers a comfortable interface to the model checkers.

The components are controlled by a project manager representing the central part of **PEP**. Figure B.1 shows the interplay and the interaction between components as realised by the project manager.

C B(PN)^2 programs and PBC terms

This section is structured as follows. Section C.1 describes B(PN)^2 without procedures. Section C.2 describes procedures. Section C.3 briefly explains the PBC.

C.1 Syntax of B(PN)^2

Figure C.1.1 shows the syntax of B(PN)^2 as implemented in the present version of **PEP**. A few non-standard items of this syntax are explained below.

In declarations, k denotes the capacity of a channel or a stack, respectively. Handshake is implemented by k=0. In commands, brackets $\langle\ldots\rangle$ delineate atomic actions. The command 'com' before the **enter** clause of a **do** command denotes initialisation; **repeat** clauses are repeated and **exit** clauses lead to termination of a **do** command. The symbol $\Box$ indicates nondeterministic choice.

In expressions, the letter `v` refers to a plain variable, the letter `c` refers to a channel and the letter `s` refers to a stack. `'v` and `v'` stand for the prevalue and the postvalue of `v`, respectively, and `v` stands for the prevalue of `v`, assuming implicitly `'v=v'`. `c!` and `c?` denote the value output (input, respectively) on channel `c`. For example, `c!='v` is **true** iff the prevalue of variable `v` and the value output on channel `c` are equal; thus, for instance, occam's command c!5 [May83] can be modelled in B(PN)^2 by `c`!=5, provided c has been declared with capacity 0. The expression `c!!` yields **true** iff `c` is full and `c??` yields **true** iff `c` is empty. Similarly, `s!` and `s?` denote the value output (input, respectively) on stack `s`. The expression `s!!` yields **true** iff `s` is full and `s??` yields **true** iff `s` is empty. The expressions `!!` and `??` are only defined for channels or stacks with capacity $k > 0$. Stacks are analogous to channels except that they incorporate a FIFO rather than a LIFO strategy. Note that both data types can be shared between parallel components.

A comment can be placed at the end of any line. It must begin with the symbol "//". Keywords are insensitive to upper / lower case. The keywords `parallel`, `choice` can alternatively be used instead of the corresponding operators $\|$ and $\square$. Names of variables must be lower case if you want to compile via a PBC term.

program	::=	block
block	::=	**begin** scope **end**
scope	::=	com $\mid$ decl; scope
decl	::=	**var** varlist : type $\mid$ **const** `const-name` = constexpr $\mid$ decl; decl
varlist	::=	`var-name` $\mid$ varlist, varlist
type	::=	set $\mid$ set **init** const $\mid$ **chan** k **of** set $\mid$ **stack** k **of** set $\quad (k \in \mathbb{N})$
set	::=	{ elems }
elems	::=	elems,elems $\mid$ const $\mid$ $a..b \quad (a, b \in \mathbb{N},\ a<b \lor (a,b) = (\mathbf{false}, \mathbf{true}))$
com	::=	$\langle$ expr $\rangle$ $\mid$ com $\|$ com $\mid$ com ; com $\mid$ **do** alt-set **od** $\mid$ **do** com **enter** alt-set **od** $\mid$ block
alt-set	::=	com; **repeat** $\mid$ com; **exit** $\mid$ alt-set $\square$ alt-set
constexpr	::=	const op const $\mid$ op const $\mid$ (constexpr)
expr	::=	`'v` $\mid$ `v'` $\mid$ `v` $\mid$ `c!` $\mid$ `c?` $\mid$ `c!!` $\mid$ `c??` $\mid$ `s!` $\mid$ `s?` $\mid$ `s!!` $\mid$ `s??` $\mid$ const $\mid$ expr op expr $\mid$ op expr $\mid$ (expr)
op	::=	`+` $\mid$ `-` $\mid$ `*` $\mid$ **div** $\mid$ **mod** $\mid$ `=` $\mid$ $\neq$ $\mid$ `<` $\mid$ `>` $\mid$ $\leq$ $\mid$ $\geq$ $\mid$ $\land$ $\mid$ $\lor$ $\mid$ $\neg$
const	::=	`false` $\mid$ `true` $\mid$ $z \quad (z \in \mathbb{N})$

Figure C.1.1: Syntax of B(PN)^2

Table C.1.2 defines the ASCII representation of special symbols:

Symbol	`ASCII`
$\langle$	`<`
$\rangle$	`>`
$\square$	`[]`
div	`div` or `/`
mod	`mod` or `%`
$\neq$	`#`
$\leq$	`<=`
$\geq$	`>=`
$\land$	`and` or `&`
$\lor$	`or` or `|`
$\neg$	`not` or `!`

Table C.1.2: ASCII representation of special symbols

C.2 Extension of $B(PN)^2$ syntax by procedures

Figure C.2.3 shows an extension of the $B(PN)^2$ syntax by procedures as implemented in the current version of **PEP**. Keywords **ref** and **const** indicate reference parameter and constant parameters, respectively; unqualified parameters are by value. Channels and stacks must be defined by reference. Arguments for reference parameters must be plain identifiers (`var-name`), arguments for value or constant parameters can also be value expressions (expr). The keywords **const**, **proc** and **ref** can be typed either in upper or in lower case. Names of procedures must be upper case if you want to compile via a PBC term. If you want to use constant parameters you have to check the **Skip Syntax Check** option, and then you can invoke first the **$B(PN)^2$ expander** and afterwards the **Expanded $B(PN)^2$→ HL Net** compiler.

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

Figure C.2.3: Extension of $B(PN)^2$ syntax by procedures.

C.3 PBC terms

PBC terms are implemented in **PEP** by means of a syntax referring to [BDH92]. The exact syntax will be given in a future version of this documentation. **PEP** users who nevertheless wish to experiment with the PBC should use a sample $B(PN)^2$ program, invoke the compiler into a PBC term and check / edit the result using the PBC editor.

D Nets

The structure of this section is as follows. Section D.1 and D.2 describe HL (high level) and LL (low level) nets, respectively, as implemented in **PEP**. In Section D.3 the net semantics of $B(PN)^2$ programs is explained. The semantics is illustrated on examples in Section D.4.

D.1 High level nets and M-nets

PEP accepts high level nets consisting, as usual, of places, transitions and arcs. Every place, transition and arc has a unique name. For arcs, this name is internal and can be read but not changed. Names of places and transitions can be changed by the user as long as the result is still unique. Furthermore, places, transitions and arcs may be equipped with inscriptions as follows.

- Places may carry a label, a type, a meaning and a reference. Labels and types come from M-net theory [BFF+95a]. Meanings and references links a place to the $B(PN)^2$ program it arises from - if it does. A meaning is essentially a comment that can be changed by the user (and thus perhaps lose its meaning), while a reference is an internal link yielding the program control point or variable name associated with a place. A reference is implemented as a reference to a part of the program text, e.g., line 5 from column 4 to 16. We recommend to hide most of the other inscriptions, because meanings are the easiest way to understand the relationship between a program and a net.

A place label[1] can be one of three kinds: entry or e (represented in **PEP** by green colour), exit or x (represented by red colour) and internal (black colour). Labels govern the algebraic properties of M-nets; for instance, the sequence of two M-nets is formed by gluing the exit places of the first to the entry places of the second[2]. Place labels also govern standard markings as described below.

A place type can be either $\{\bullet\}$ (meaning the place is of ordinary place / transition net kind), or of the same kind as the type of a variable declaration (and, in fact, serving the same purpose). More precisely, the present version of **PEP** supports the following types: $\{\text{dot}\} = \{\bullet\}$; subsets of the set $\{\textbf{false}, \textbf{true}\}$; and finite subsets of the set $\mathbb{N}$ of non-negative integers. Furthermore 2-tupels like $\{1..4\}$ x $\{1..4, dot\}$ are supported with some restrictions. Place types are essentially the 'colour sets' known from HL net theory [Jen92]. They govern the unfolding of a HL net into a LL net.

Entry and exit places have default type $\{\text{dot}\}$, internal places may have other types. Place labels indicate the standard initial marking: all entry places have a black token and all other places have no token, as well as the standard final marking: all exit places have a black token and all other places have no token.

A place meaning could, for example, be 'begin' (even though the place label is not e but internal). This could be the case if this place is the start point of an internally nested block of a program.

Furthermore a place can have a meaning and a reference to the B(PN)^2 program. For instance, there is a direct reference between a place in the control flow of a B(PN)^2 program and a place in the corresponding M-net.

- Transitions may carry a label, a condition, a meaning and a reference. Again, labels and conditions come from M-net theory while meanings are essentially comments and references link the transition to a program action.

  A transition label is a multiset of parametrised action names (or action terms). During compositional construction of the net associated to a B(PN)^2 program, this multiset usually contains the free (in the sense of not yet declared) variables of the atomic action to which the transition corresponds. When the construction is completed and the program is well-formed (all variables are declared) then the labels of all transitions are empty. For example, a nonempty action term can be of the form `x('x,x')` where `x` is some B(PN)^2 variable. This represents the fact that the prevalue of `x` is `'x` and the postvalue of `x` is `x'` and that they are not yet bound to specific values. In general, action terms govern synchronisation of transitions. For instance, a term $\overline{\texttt{x}}(3,4)$ occurring in the net corresponding to the translation of the declaration of `x` could be synchronised

[1] The word 'status' is used instead of 'label' in the HL editor

[2] The operation of sequential composition and other operations are presently built into the internal parts of the system; they are not visible to the user. It is conceivable that at some later point in time the system is extended to allow the user to draw several nets and form their compositions.

with `x('x,x')` by binding `'x` to `3` and `'x` to `4`; overlining $\overline{\mathtt{x}}$ indicates conjugation similarly as in CCS [Mil89].

A transition condition (for which **PEP** also uses the words *value term*) is a Boolean expression which must be made true for the transition to occur. When a transition arises from an atomic $B(PN)^2$ action a then this condition is simply the expression inside a. More generally, a value term is typically a conjunction of terms referring to the variables used in the arc annotations, obeying the same syntax as $B(PN)^2$ expressions. For example (continuing the above example), if a transition t with value term `'x`>0 has an input arc a annotated by {`'x`} then t can only occur if `'x` can be bound to a number greater 0, i.e., if the place on the other end of a (has corresponding type and) carries a corresponding token. (Note that what in terms of $B(PN)^2$ denotes pre- and postvalues is taken as variables in terms of the associated HL net.) Transition conditions govern unfolding as in standard HL net theory.

Transition meanings could, for instance (and somehow redundantly), denote the corresponding atomic actions. Once more a reference is implemented as a reference to a part of the program text, e.g., line 5 from column 4 to 16 and meanings are meant to support the understanding of the relationship between a program and a net.

- Arcs may carry multisets of variables as their inscriptions (in the above example, a carries a singleton). These variables can be used in value terms of neighbouring transitions. This specifies the possible bindings and thus the circumstances of transition occurrences, as in standard HL net theory. The same mechanism underlies the unfolding of a HL net into a low level net. Such variables can also occur as parameters in the action terms of a neighbouring transition. They are then replaced by the respective values in every occurrence of the transition.

Markings M of an HL net are functions from places to multisets such that for every place s, $M(s)$ is a finite multiset over the type of s.

D.2 Low level nets and boxes

LL nets are special cases of HL nets in the following sense. All place types are $\{\bullet\}$. Thus markings, being multisets over $\{\bullet\}$, can simply be interpreted as functions from the set of places into $\mathbb{N}$, as usual in place / transition net theory [Rei85]. All transition conditions are **true** and all action terms are multisets of unparametrised actions. In fact, in **PEP** it is assumed that all transition labels are simply the empty multiset, since this is always the result of translating a well-formed $B(PN)^2$ program. As a consequence of all places having singleton type, only the sizes of the multisets of variables associated with arcs matter; thus a multiset of, say, size 3 on arc a needs 3 tokens on any input place of a for the output transition of a to occur, and thus, in standard terminology, this arc has weight 3. Thus, LL nets are arc-weighted place / transition nets for which places are divided into the three classes described above (and both places and transitions may have meanings and references).

D.3 Net semantics of $B(PN)^2$

The translation of a $B(PN)^2$ block - and hence a $B(PN)^2$ program - into a net is achieved by the following method. The translation is described inside-out, starting with basic program ingredients. Every atomic action $\langle \ldots \rangle$ is translated into a single transition (on the M-net level) or a set of alternative transitions (on the box level). The labels of these transitions mention program variables occurring freely in the corresponding atomic action. By this label, such transitions are later synchronised with corresponding transitions coming from the declaration of the variables. Every program connective is translated into operations on M-nets or boxes, respectively, according to its semantics. By doing this for the entire block body, a net describing its control flow is created.

Every declaration is translated into a special net according to its meaning (see below for examples) which is put in parallel with the net of the block body. At block level, a synchronisation followed by a restriction on all local variables is performed (synchronisation followed immediately by restriction is called scoping). This effects synchronisation of atomic action transitions with data transitions. If an atomic action contains partly local and partly global variables then synchronisation is performed only partly for its local variables. There are also block exit actions effecting the de-initialisation of all variables upon exit of the block. This is important in case a block is nested inside a loop because between iterations, values of inner variables must be erased. By doing the above for all local variables, a net combining data flow and control flow description of the block is created.

The translation described above is performed innermost-first for all blocks of the program. The outermost block contains no global variables; thus all transitions that are eventually created carry the $\emptyset$ label characterising them as internal, i.e., not waiting to be synchronised any further. In **PEP** only the end result of this translation is available to the user for further manipulation.

D.4 Examples

The first example shows how the control flow of a block body is translated into (HL and LL) nets. Figure D.4.1 shows a program. Figure D.4.2 and Figure D.4.3 show an M-net and a box, respectively, corresponding to the **do** construct in the block body. Note that the latter is an unfolding of the former in the sense of high level nets.

```
begin
var a: {1..4} init 1;
do ⟨a' = 2⟩ enter
     ⟨a > 1⟩; ⟨a' = 'a − 1⟩; exit
  ▯ ⟨a < 4⟩; ⟨a' = 'a + 1⟩; repeat
od
end
```

Figure D.4.1: A $B(PN)^2$ program.

The second example shows how a declaration is translated into (HL and LL) nets. Figure D.4.4 shows a program. Figure D.4.5 shows the M-net corresponding to variable *name*. Note that the internal (non-entry and non-exit) place has the same type as the variable (three tokens 1, 3 and 4). Figure D.4.6 shows the LL box corresponding to variable *name*. Note that it has three internal places corresponding to the values of *name*, and note also that it is an unfolding of the M-net of Figure D.4.5. Figure D.4.7 shows the M-net which corresponds to the entire block shown in Figure D.4.4. It arises by scoping[3] the data M-net (shown in Figure D.4.5) and the control M-net of the block body (not shown) by variable *name*.

[3]See section D.3 for the notion of scoping.

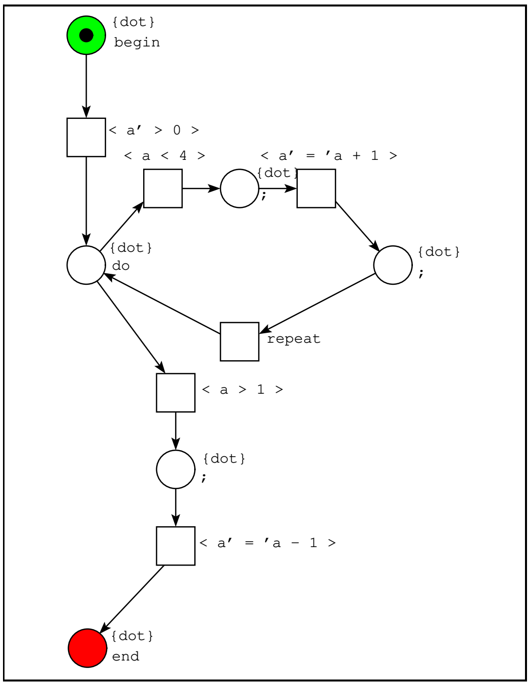

Figure D.4.2: M-net corresponding to the block body of Figure D.4.1 The places are labelled with a type and a meaning. Transitions are shown with their meaning. Arc inscriptions are omitted because there are only black tokens.

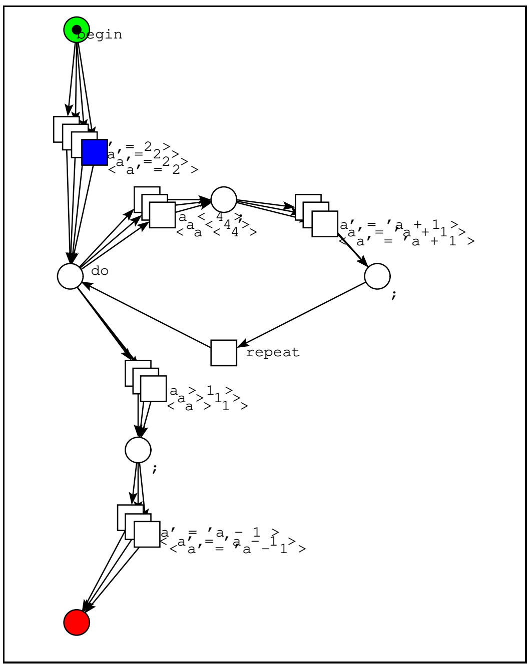

Figure D.4.3: Petri box corresponding to the block body of Figure D.4.1 Places and transitions are only labelled with their meaning

Similarly, Figure D.4.8 shows the LL box which corresponds to the block shown in Figure D.4.4. It arises by scoping the data box (shown in Figure D.4.6) and the control box of the block body (not shown) by variable *name*. Note that again, Figure D.4.8 is an unfolding of Figure D.4.7. Thus, scoping is a congruence with respect to unfolding. This is true of all M-net operations [BFF+95a].

begin
var *name*: {1,3..4} **init** 1;
$\langle name' = 3 \rangle$
end

Figure D.4.4: Second B(PN)^2 example

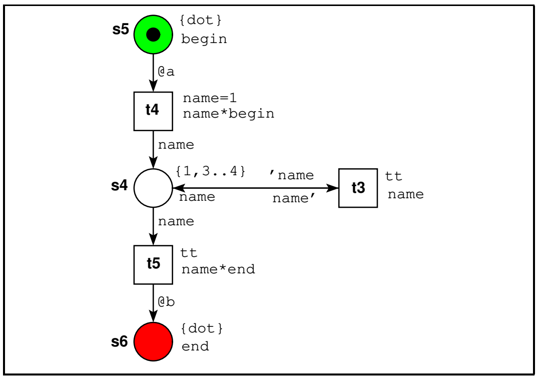

Figure D.4.5: M-net data box for Figure D.4.4 Place and transition names are shown. The valueterm of a transition is positioned above the meaning. Arc inscription are also shown. Note that `@a` and `@b` are generated (by the compiler **B(PN)^2→HL Net**) as arc inscriptions indicating that it is an output (input resp.) arc of a place with type {`dot`}.

The next example shows the use of channels. Channels are data structures implementing a first-in-first-out strategy. The capacity k of a channel indicates how many values can be written before any has to be read; $k = 0$ means handshake synchronisation. In the programs, channels c are addressed by input commands $c?$ and output commands $c!$. Thus, for example, if a program is to be changed by increasing the channel's capacities, only declarations need to be modified.

Figure D.4.9 shows a program in which a channel of capacity 1 is declared. Figure D.4.10 shows the M-net corresponding to variable *chan1*. Note that the type of the middle place is a Cartesian product of channel capacity and channel type. Figure D.4.11 shows the control flow M-net of the program and Figure D.4.12 shows the M-net of the whole block, which is a scoping of the previous two nets with respect to variable *chan1*.

Stack data structures are treated analogously, except that they implement a last-in-first-out strategy. The last example shows a program with a stack of capacity 3 (Figure D.4.13), the two M-nets describing its data and control (Figure D.4.14 and D.4.15, respectively) and the scoped M-net describing the entire block (Figure D.4.16).

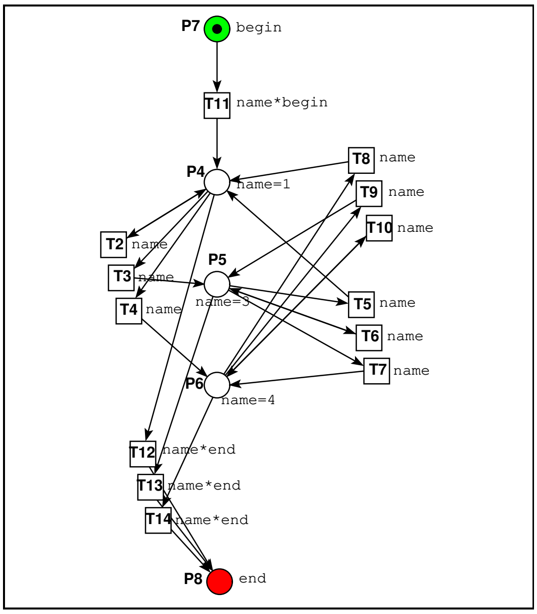

Figure D.4.6: LL data box for Figure D.4.4

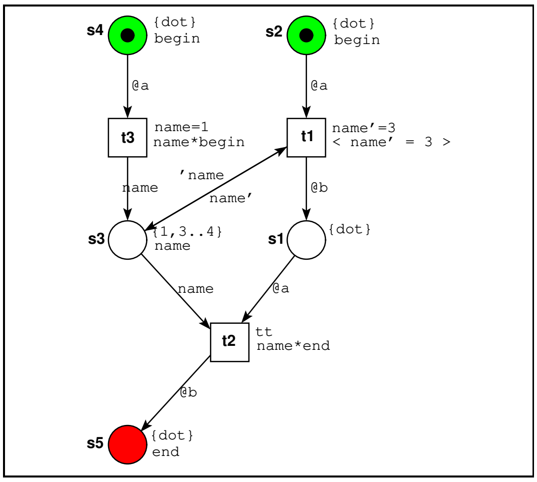

Figure D.4.7: M-net of Figure D.4.4

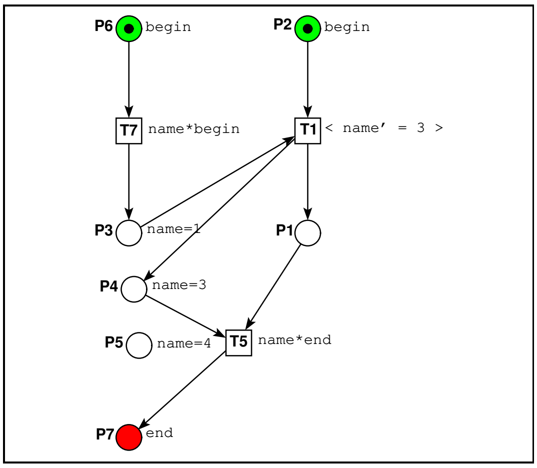

Figure D.4.8: LL box of Figure D.4.4

begin
var *chan1*: **chan** 1 **of** $\{3..5\}$ **init** 1;
$\langle chan1! = 3\rangle \parallel \langle chan1? = 3\rangle$
end

Figure D.4.9: Third B(PN)^2 example

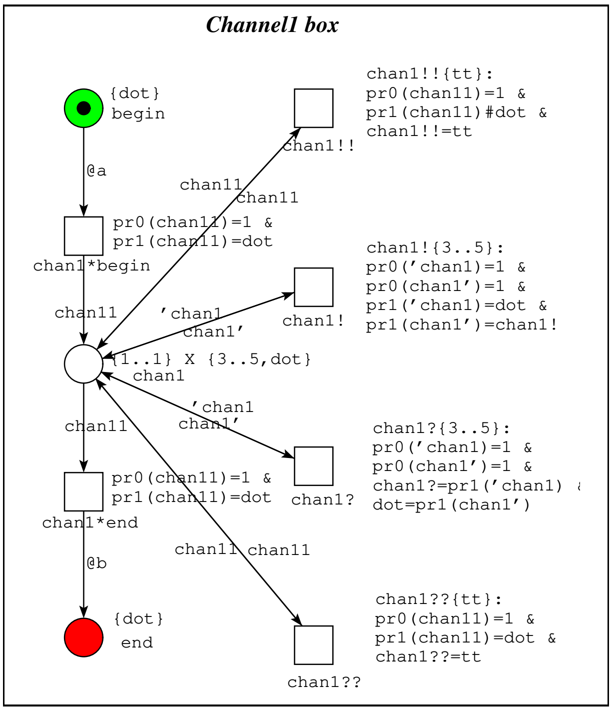

Figure D.4.10: M-net channel box for Figure D.4.9. A cartesian product is represented as `{1..1} X {3..5,dot}`. E.g. the first component is referenced with `pr0('chan1)`. The range of a variable can be restricted by prefixing the valueterm, e.g., with `chan1!{3..5}`.

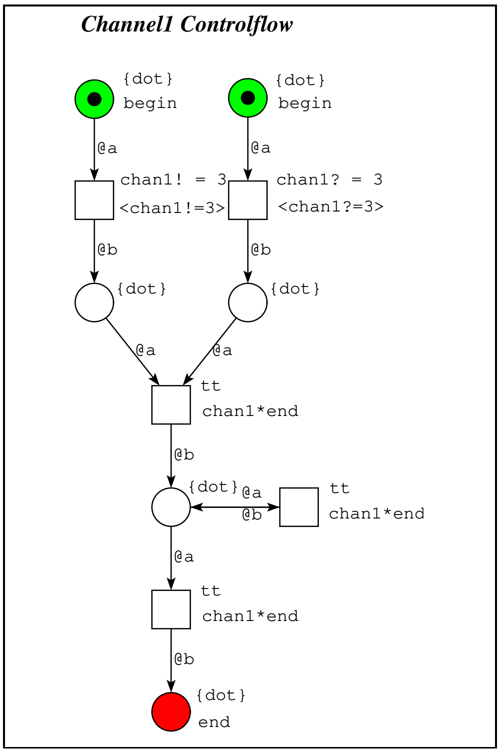

Figure D.4.11: M-net control box for Figure D.4.9

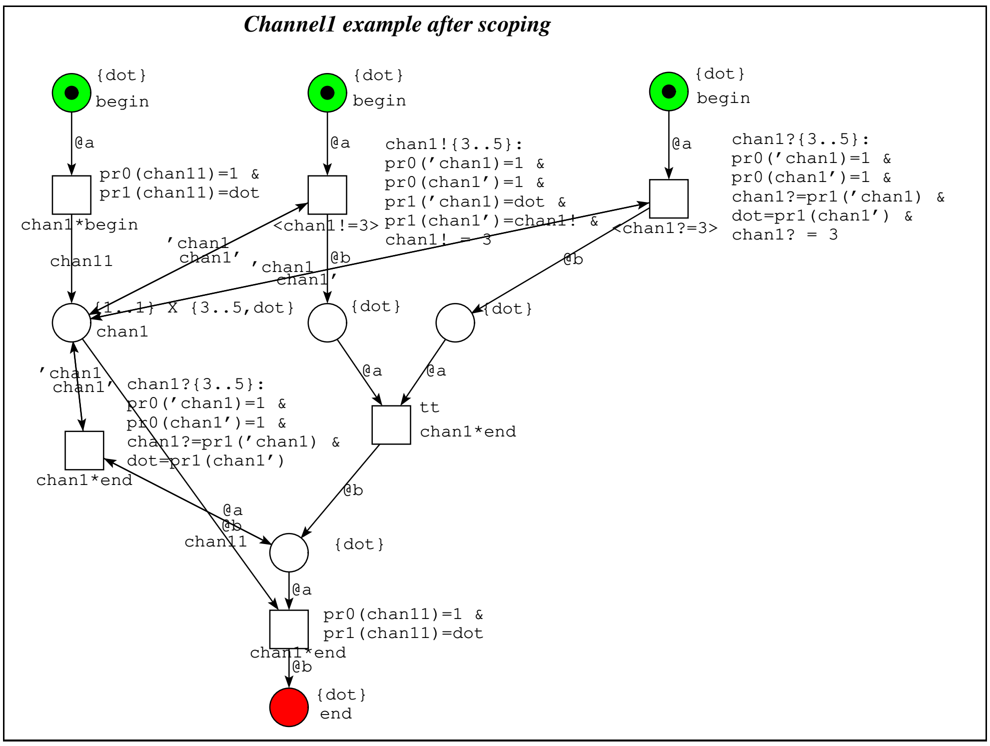

Figure D.4.12: M-net for Figure D.4.9

begin
var *stack3*: **stack** 3 **of** $\{3..5\}$ **init** 1;
$\langle stack3! = 3\rangle \parallel \langle stack3? = 3\rangle$
end

Figure D.4.13: Third B(PN)^2 example

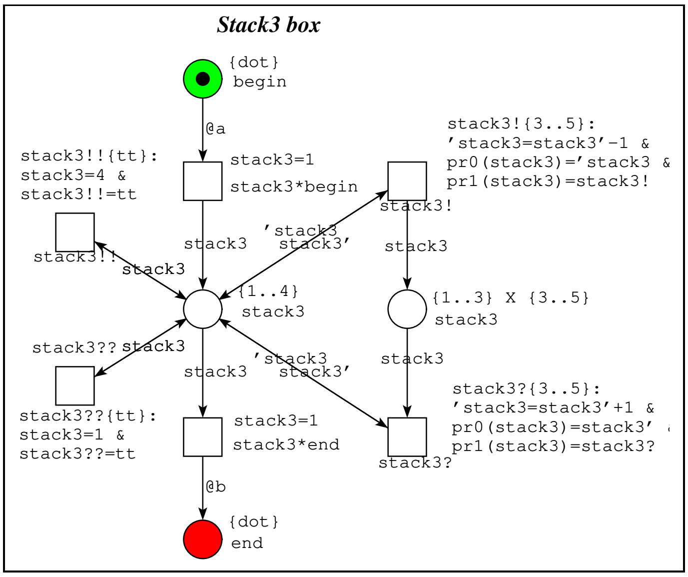

Figure D.4.14: M-net stack box for Figure D.4.13

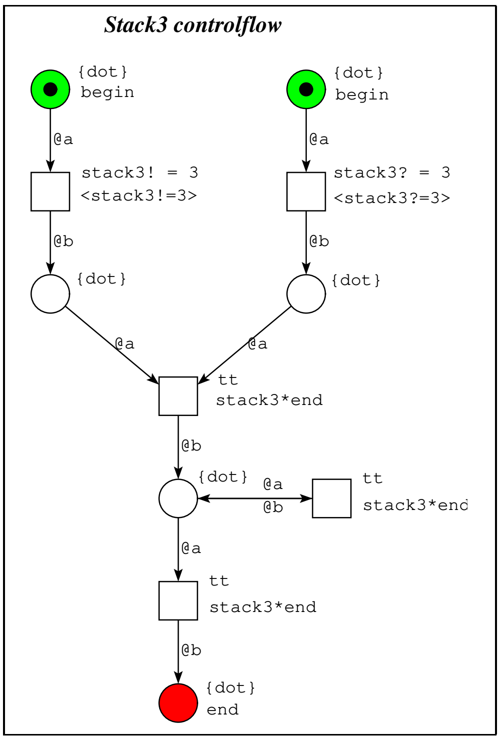

Figure D.4.15: M-net control box for Figure D.4.13

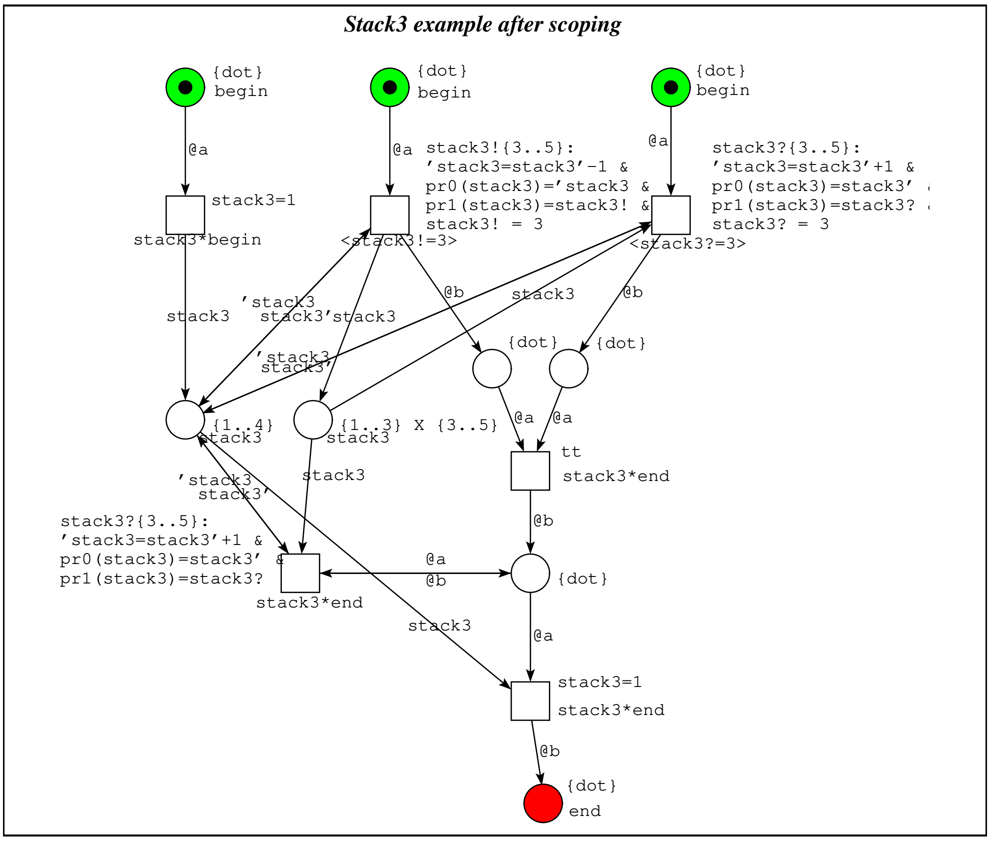

Figure D.4.16: M-net for Figure D.4.13

E Verification algorithms

PEP incorporates dedicated verification algorithms for specific properties and three different model checking algorithms MC-0, MC-1 and MC-2 for generic properties. Algorithm MC-1 model checks properties described in a branching time logic. This algorithm is based on the finite prefix of a safe net; it does not need to construct the graph of reachable markings. Algorithm MC-0 checks the same set of properties but is optimised for safe T-systems. In fact, it uses a completely different approach based on linear programming and needs neither the prefix nor the marking graph. Algorithm MC-2 is an extension of MC-1. It works for the same class of nets but for a slightly extended (linear time) logic which allows to specify more liveness properties than the branching time logic[4].

In this section we first describe, in this order, MC-1 (Section E.1), MC-2 (Section E.2) and MC-0 (Section E.3). Section E.4 describes how formulae are represented in **PEP**.

E.1 A branching time logic model checker MC-1

The model checker for safe Petri nets integrated in **PEP** has been developed by Esparza [Esp94] and implemented by Graves [Gra95a]. This algorithm uses the finite prefix of the branching process of a safe Petri net as defined by MacMillan [McM92] and a temporal logic formula as inputs, and then checks whether or not the formula holds for the corresponding net.

Syntax

For a safe marked net $\Sigma = (P, T, F, M^0)$ the set of branching time formulae ϕ is defined by the following syntax:

$$\phi \quad ::= \quad \mathbf{true} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \Diamond\phi \qquad\qquad (p \in P),$$

where $\Diamond\phi$ represents the operator 'there exists a reachable marking such that ϕ'. Other operators such as $\vee$ or $\Box$ can be derived e.g. by $\phi_1 \vee \phi_2 = \neg(\neg\phi_1 \wedge \neg\phi_2)$ and by $\Box\phi = \neg\Diamond\neg\phi$, respectively.

Semantics

The semantics of formulae ϕ is defined in the standard way in terms of (reachable) markings. A formula $\Diamond\phi$ holds for a marking M if there is a marking reachable from M for which ϕ holds. A $\Diamond$-free formula can be evaluated directly in a given marking (using the fact that Σ is safe). A formula ϕ holds for $\Sigma = (P, T, F, M^0)$ if it holds for M^0.

In this logic various safety properties can be expressed, such as, for instance:

Reachability of a marking

The system in Figure E.1.1 satisfies

$\Diamond(\neg P1 \wedge P2 \wedge \neg P3 \wedge P4 \wedge \neg P5 \wedge \neg P6)$

iff the marking $\{P2, P4\}$ is reachable.

Concurrency of transitions

The system in Figure E.1.1 satisfies

$\Diamond(P4 \wedge P5)$

iff a marking is reachable which enables $T3$ and $T5$ concurrently.

[4]**NOTE**: The current version of **PEP** includes a preliminary version of this algorithm. There exist theoretical problems which are not yet completely solved.

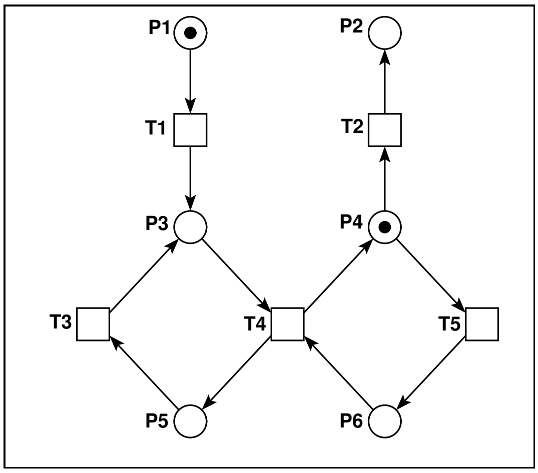

Figure E.1.1: A safe system

Liveness of transitions

The system in Figure E.1.1 satisfies

$\Box\Diamond P4$

iff transition $T5$ can be enabled from every reachable marking.

Mutual exclusion

The system in Figure E.1.2 satisfies

$\Box(\neg cs1 \vee \neg cs2)$

iff places $cs1$ and $cs2$ cannot be marked at the same time.

Absence of deadlocks

The system in Figure E.1.2 satisfies

$\Box(\neg cs1 \Rightarrow \Diamond cs1) \wedge \Box(cs1 \Rightarrow \Diamond\neg cs1)$

iff the left agent can always enter and exit its critical section.

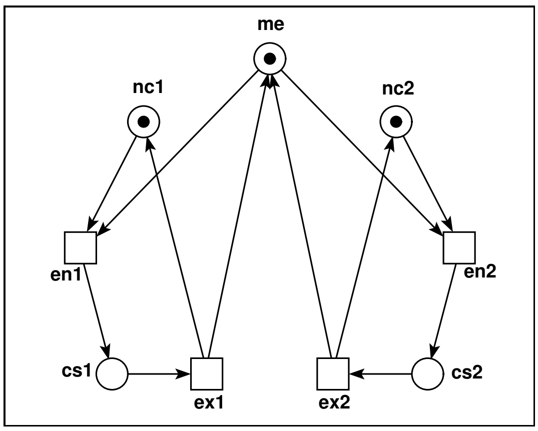

Figure E.1.2: A mutual exclusion example

E.2 A linear time logic model checker MC-2

The linear time logic [Esp93] defined in this section extends the previously defined one. By this logic, more general liveness properties can be expressed. For a fragment of the logic, also defined below, an efficient model checker has been implemented in **PEP** [Gra95a].

Syntax of the linear time extension

Two classes of formulae are distinguished: process formulae and system formulae. As before let $\Sigma = (P, T, F, M^0)$ be a safe marked net.

Syntax of process formulae

The set of linear time process formulae ϕ is defined by

$$\phi \quad ::= \quad \mathbf{true} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \Diamond\phi \qquad\qquad (p \in P),$$

which is the same (with the same meaning) as the previous syntax; but such formulae will only be applied to the processes of Σ.

Syntax of system formulae

The set of linear time system formulae Φ is defined by:

$$\Phi \quad ::= \quad \forall\phi \mid \exists\phi,$$

where ϕ is a process formula over Σ.

Semantics of system formulae

Let ϕ be a system formula over Σ.

$$\begin{array}{lll} \Sigma \models \forall\phi & \text{iff} & \text{for all maximal processes } \pi \text{ of } \Sigma\text{: } \pi \models \phi. \\ \Sigma \models \exists\phi & \text{iff} & \text{there is a maximal process } \pi \text{ of } \Sigma\text{: } \pi \models \phi. \end{array}$$

Note: When the branching time modality $\Diamond$ is applied to a maximal process then 'there is some future marking' turns into 'eventually' (for processes, but not for interleavings).

In the linear time logic various liveness properties can be expressed, such as, for instance:

- **p will be marked eventually:**
  This property can be expressed by the following formula:
  $\forall\Diamond p$ (or equivalently: $\neg\exists\neg\Diamond p$).

- **p_1 yields p_2:**
  This means that if p_1 is marked then eventually p_2 will be marked. This property can be expressed by the following formula:
  $\forall\Box(p_1 \Rightarrow \Diamond p_2)$ (or equivalently: $\neg\exists\Diamond(p1 \wedge \neg\Diamond p2)$).

- **p is infinitely often marked and unmarked in every process:**
  This property can be expressed by
  p yields $\neg p \wedge \neg p$ yields p.

Note: It is not expressible in this logic that two places $p1$ and $p2$ will eventually be marked concurrently.

A fragment of the linear time logic

For a marked net $\Sigma = (P, T, F, M^0)$ the set of linear time process formulae Φ is defined by the following syntax:

$$\begin{array}{lll} \gamma & ::= & \mathbf{true} \mid \mathbf{false} \mid p \\ \phi & ::= & \gamma \wedge \Diamond\gamma \mid \gamma \wedge \neg\Diamond\gamma \\ \Phi & ::= & \exists\Diamond\phi \mid \exists\neg\Diamond\phi, \end{array}$$

where p is a place of Σ.
This fragment can be efficiently checked by **PEP** using non-state-graph based methods (i.e., the finite prefix of the branching process).

E.3 The T-system model checker MC-0

As far as the user is concerned, the T-system model checker serves the same purpose as the other one, provided the net to be checked is a T-system (also called a marked graph [Rei85]). It may yield the result (even) more quickly. Nevertheless, the theoretical differences between MC-0 and MC-1 are not very large since both are provably polynomial in the size of the T-system (in this restricted case). The **PEP** tool checks automatically whether or not the underlying net is actually a T-system before invoking MC-0. The MC-0 algorithm is based on integer linear programming [BE92, Thi93].

E.4 Representation of formulae in PEP

In the **PEP** formula editor, formulae can be typed in terms of the ASCII representation shown in table E.4.3.

Symbol	`ASCII`
$\neg$	`-`
$\wedge$	`*`
$\vee$	`+`
$\Leftrightarrow$	`<=>`
$\Rightarrow$	`=>`
$\forall$	`!`
$\exists$	`?`
true	`1`
false	`0`
$\Diamond$	`^`
$\Box$	`#`

Table E.4.3: ASCII representation of temporal logic formulae

F Future work

During work with the **PEP** system, many different case studies have demonstrated that the tool is capable of handling comprehensive examples. As realistic examples, a protocol for telephone networks [Gra95] and a pilot behaviour model [RO95] can be mentioned. The experiences lead us to the following aims for future work:

1. A model-checker for high-level Petri boxes should be developed and implemented, in such a way that the unfolding into low-level Petri boxes – often yielding too big nets for realistic problems – can be bypassed if desired. Obviously, this aim calls for intensive research. Various research groups at different places are already engaged in such work.
2. The language $B(PN)^2$ should be augmented by provision of abstract data types and by fully extending the procedure concept. In this way the user will have the full power to model complex parallel systems easily.

3. The capabilities of the INA system [Sta92] (in particular, its state graph checker and its theorem data base) will be combined with the other analysis components of the **PEP** tool.

In a different line of development, new interface add-ons to the **PEP** system will be implemented. 'SDL', 'VHDL', 'OCCAM' and finite automata (see [GMA95]) are presently considered to be suitable interface languages which, when added to the tool, may greatly enhance its appeal to users of those languages.

References

[BDH92] Eike Best, Raymond Devillers, and Jon G. Hall. The Box Calculus: a New Causal Algebra with Multi-Label Communication. In G. Rozenberg, editor, *Advances in Petri Nets 92*, volume 609 of *Lecture Notes in Computer Science*, pages 21 – 69. Springer-Verlag, 1992.

[BE92] Eike Best and Javier Esparza. Model Checking of Persistent Petri Nets. In *5th Workshop Computer Science Logic-91*, volume 626 of *Lecture Notes in Computer Science*, pages 35–52. Springer-Verlag, 1992.

[BF95] Eike Best and Hans Fleischhack, editors. *PEP: Programming Environment Based on Petri Nets.* Number 14/95 in Hildesheimer Informatik-Berichte. May 95.

[BFF+95a] E. Best, H. Fleischhack, W. Frączak, R. P. Hopkins, H. Klaudel, and E. Pelz. A Class of Composable High Level Petri Nets. In G. De Michelis and M. Diaz, editors, *Proceedings of Application and Theory of Petri Nets 1995, Torino*, volume 935 of *Lecture Notes in Computer Science*, pages 103–118. Springer, June 1995.

[BFF+95b] E. Best, H. Fleischhack, W. Frączak, R. P. Hopkins, H. Klaudel, and E. Pelz. An M-Net Semantics of $B(PN)^2$. In J. Desel, editor, *Structures in Concurrency Theory*, Workshops in Computing. Springer, 1995.

[BH93] Eike Best and Richard Pinder Hopkins. $B(PN)^2$ – a Basic Petri Net Programming Notation. In A. Bode, M. Reeve, and G. Wolf, editors, *Proceedings of PARLE '93*, volume 694 of *Lecture Notes in Computer Science*, pages 379 – 390. Springer-Verlag, 1993.

[BK95] Eike Best and Maciej Koutny. A Refined View of the Box Calculus. In G. De Michelis and M. Diaz, editors, *Proceedings of 16th Conference on Application and Theory of Petri Nets 1995, Torino*, volume 935 of *Lecture Notes in Computer Science*, pages 103–118. Springer, June 1995.

[Dev94] Raymond Devillers. S-invariant Analysis of General Recursive Petri Boxes. Technical Report LIT-294, Université Libre de Bruxelles, 1994. Acta Informatica (1995).

[ERV95] Javier Esparza, Stefan Römer, and Walter Vogler. An Improvement of McMillan's Unfolding Algorithm. Technical Report TUM-19525, SFB-Bericht Nr. 342/12/95 A, Technische Universität München, August 1995.

[Esp93] Javier Esparza. *A Partial Order Approach to Model Checking.* Habilitationsschrift, Universität Hildesheim, February 1993. An abridged version has appeared as [Esp94].

[Esp94] Javier Esparza. *Model Checking Using Net Unfoldings*, pages 151–195. Number 23 in Science of Computer Programming. ELSEVIER, 1994.

[GMA95] Bernd Grahlmann, Matthias Moeller, and Ulrich Anhalt. A New Interface for the PEP Tool — Parallel Finite Automata —. In J. Desel, H. Fleischhack, A. Oberweis, and W. Reisig, editors, *2. Workshop Algorithmen und Werkzeuge für Petrinetze*, number AIS 22, pages 21–26. FB Informatik Universität Oldenburg, October 1995.

[Gra95] Bernd Grahlmann. Verifying Telecommunication Protocols with PEP. In *Proceedings of RELECTRONIC'95*, pages 251–256, Budapest Hungary, October 95. Scientific Society for Telecommunications.

[Gra95a] Burkhard Graves. Ein Modelchecker für eine Linear-Time-Logik. Diplomarbeit, Universität Hildesheim, January 1995.

[Gra95b] Burkhard Graves. Erläuterungen zu Esparza's L1-Model-Checker. In [BF95], pages 35–43. Universität Hildesheim, May 1995.

[Heh85] E.R. Hehner. *Predicative Programming.* Prentice Hall, 1985.

[Him95] Tobias Himstedt. Entwurf und Implementierung eines Highlevel-Petrinetz Editors und Simulators. Diplomarbeit, Universität Hildesheim, January 1995.

[Jen92] Kurt Jensen. *Coloured Petri Nets. Basic Concepts.* EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1992.

[Jen94] Lars Jenner. Ein Prozedurkonzept für die parallele Hochsprache $B(PN)^2$. Diplomarbeit, Universität Hildesheim, October 1994.

[Jen95] Lars Jenner. A Low-Level Net Semantics for $\mathrm{B(PN)}^2$ with Procedures. In [BF95], pages 61–76. Universität Hildesheim, May 1995.

[KP94] Hanna Klaudel and Elisabeth Pelz. Handling Abstract Data Types in the Petri box Calculus. Technical Report TR 94-05, Univ. Paris XII Val de Marne, 1994.

[KP95] Hanna Klaudel and Elisabeth Pelz. Communication as Unification in the Petri box Calculus. In *Proceedings of FCT'95*, number 965: in LNCS, pages 303–312, Dresden, 1995. Springer-Verlag.

[LP95] Johan Lilius and Elisabeth Pelz. An M-Net Semantics for $\mathrm{B(PN)}^2$ with Procedures. Draft, 1995.

[May83] David May. *OCCAM*, volume 18 / 84) of *SIGPLAN Notices*, pages 69–79. 1983.

[McM92] K. L. McMillan. Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits. In *4th Workshop on Computer Aided Verification*, pages 164 – 174, Montreal, 1992.

[Mel95a] Stephan Melzer. Codegenerierung für eine parallele Programmiersprache. Diplomarbeit, Universität Hildesheim, February 1995.

[Mel95b] Stephan Melzer. Design and Implementation of a C-Code Generator for $\mathrm{B(PN)}^2$. In [BF95], pages 77–85. Universität Hildesheim, May 1995.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Rei85] Wolfgang Reisig. *Petri Nets. An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science.* Springer Verlag, 1985.

[RO95] W. Ruckdeschel and R. Onken. Petri Net Modelling, Analysis and Realtime-Simulation of Pilot Behaviour. In *Application and Theory of Petri Nets*, June 95.

[Spi92] J. M. Spivey. *The Z Notation. A Reference Manual.* Prentice Hall, 1992.

[Sta92] Peter H. Starke. INA: Integrated Net Analyzer. Handbuch, 1992.

[Thi93] Thomas Thielke. Implementierung eines effizienten Modelchecking-Algorithmus. In G. Scheschonk and W. Reisig, editors, *Petri-Netze im Einsatz für Entwurf und Entwicklung von Informationssystemen*, pages 127–139. Springer-Verlag, 1993.

Index