Differences between PROD Reference Manual and the latest version of PROD

Windows-PROD

There is no supported MS-DOS version of PROD any more. Instead, PROD can now be run in Windows (3.1, 3.11, '95, NT). The Windows version is more reliable than the past MS-DOS versions. In principle, any analysis task that can be performed successfully using the UNIX version can be performed successfully using the Windows version.
If the batch program `prod` and interruption possibilities are not counted, the programs of the Windows version of PROD are used just like the identically named programs in the UNIX version of PROD. What comes to interruption, it suffices to say that the Windows version of PROD has no supported way to interrupt reachability graph generation or a graph query.
One possible way to do analysis is as follows.

1. Start an MS-DOS emulator such as MS-DOS Prompt and go to the directory that contains the net description file of the net you are going to analyze. Let's assume that the name of the net description file is `mynet.net`.

2. In the MS-DOS emulator, execute the command "`prod mynet.init`" and then the command "`tmp.bat`".

3. Start the File Manager (`winfile`) in Windows and go to the directory that contains the net description. The name of the directory should remain highlighted in the displayed directory tree. (See Figure 1.)

4. Run "`prpp` *options-if-wanted* `mynet`" by using the Run option of the File menu of the File Manager. If you want, you can copy text from the Stdin/Stdout/Stderr window of the `prpp` program to some text editor window in the usual "copy and paste" style. After you've seen that the program has finished its execution and you do not need the output of the program any more, you can close the windows of the program with a single action e.g. by choosing the Exit option from the File menu of the window named by the program. These instructions apply to all those programs of PROD that are run by using the above mentioned Run option. (See Figure 2.)

5. In the MS-DOS emulator, execute the command "`prod mynet.compile`" and then the command "`tmp.bat`".

6. Run "`mynet` *options-if-wanted*" by using the above mentioned Run option. (See Figure 3.)

7. If you are interested in the strongly connected components of the generated reachability graph, run "`strong` *options-if-wanted* `mynet`" by using the above mentioned Run option.

8. Run "`probe` *options-if-wanted* `mynet`" by using the above mentioned Run option. In the Stdin/Stdout/Stderr window, you can type anything and see what `probe` responds. Typing "`quit`" is a recommendable way to make `probe` finish its execution. (See Figure 4.)

9. If you want to do comparative analysis, go back to step 6.

10. In the MS-DOS emulator, execute the command "`prod mynet.clean`" and then the commands "`tmp.bat`" and "`exit`".

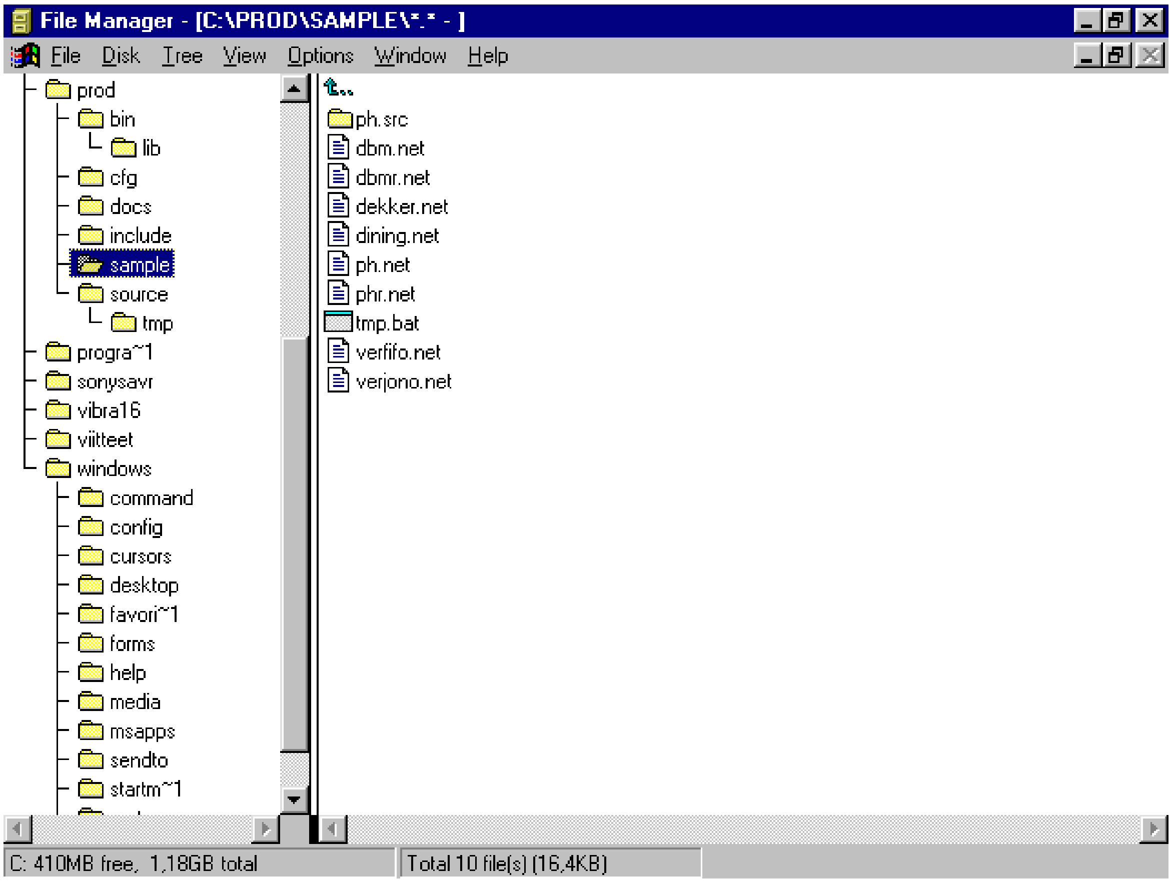

Figure 1: An example screen for step 3.

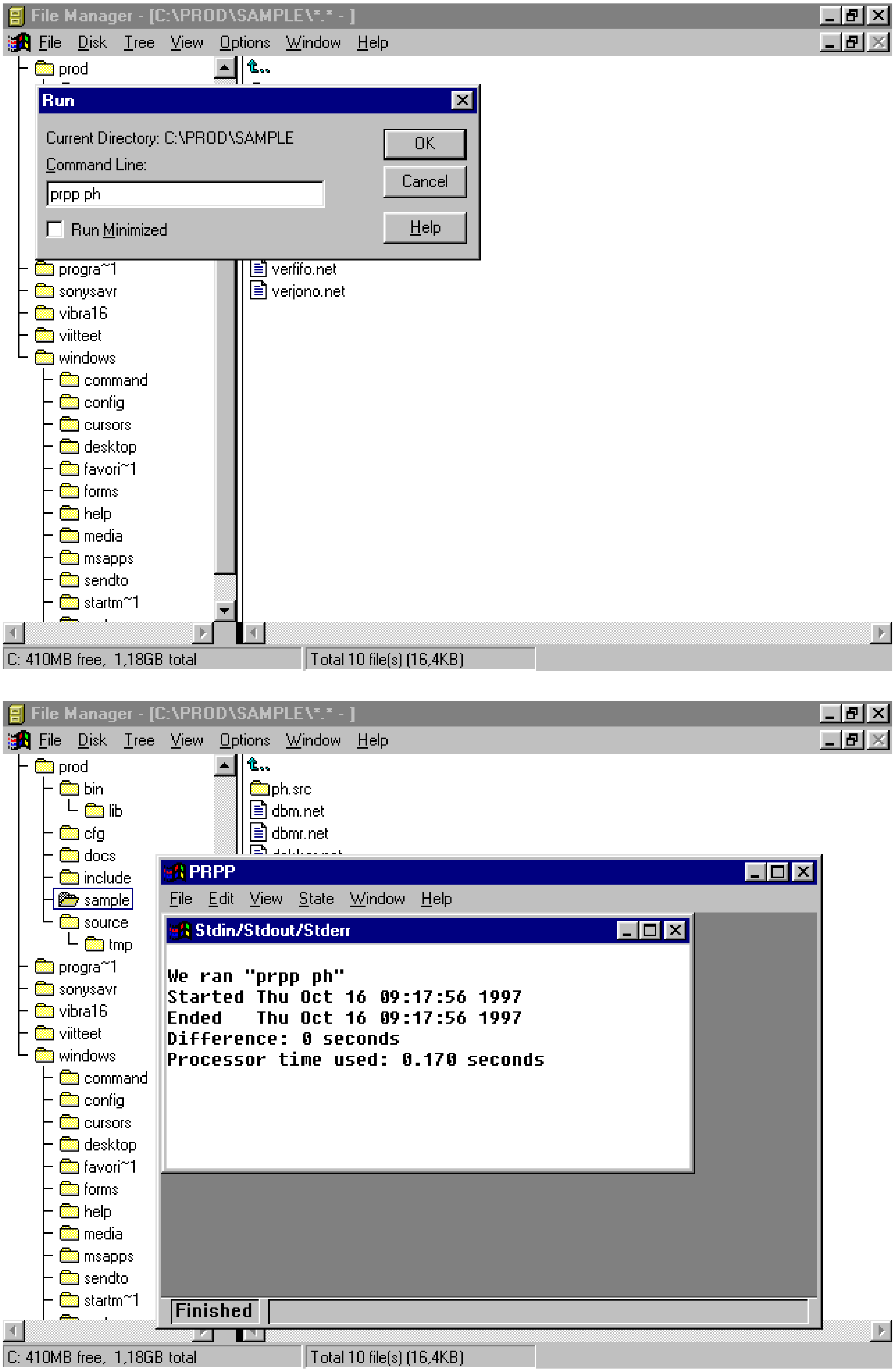

Figure 2: Example screens for step 4.

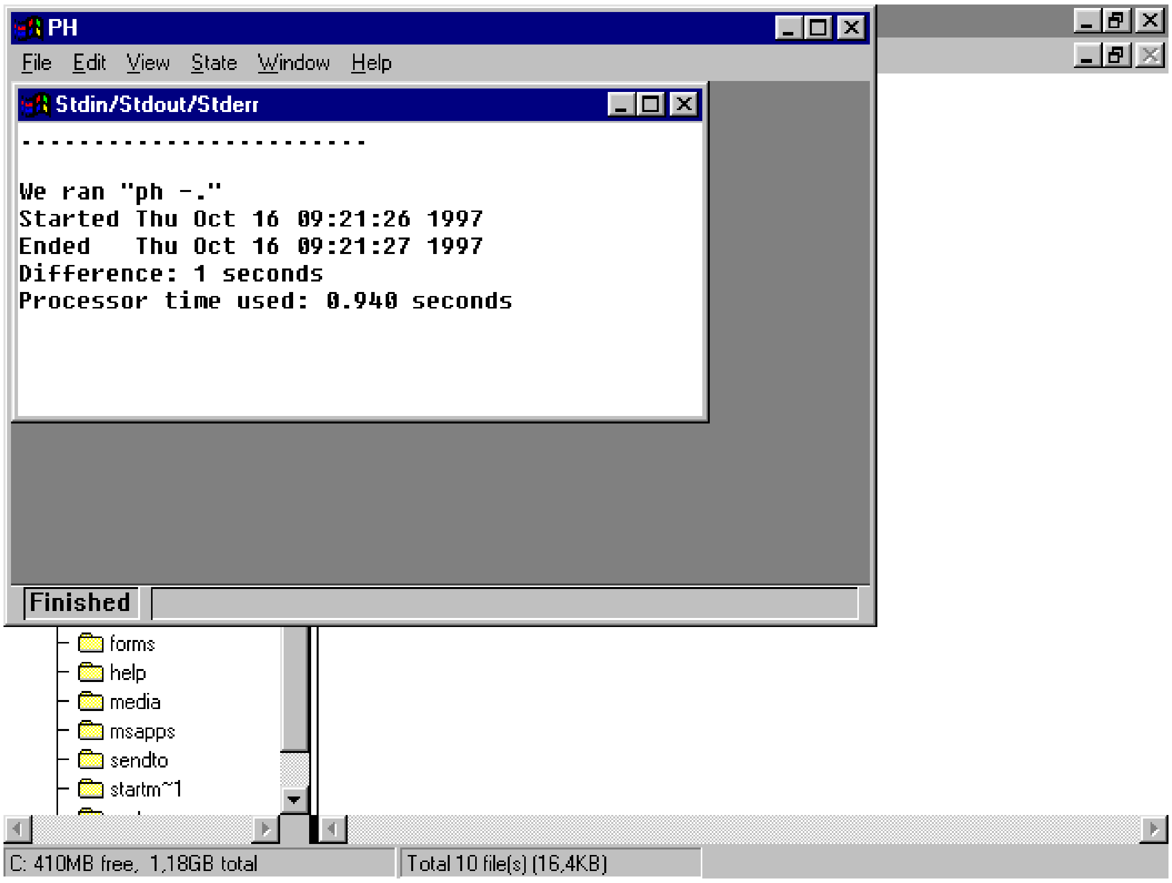

Figure 3: An example screen for step 6.

```
PROBE - [Stdin/Stdout/Stderr]
File  Edit  View  State  Window  Help
0#statistics
Number of nodes: 242
Number of arrows: 805
Number of terminal nodes: 1
Number of nodes that have been completely processed: 242
Number of strongly connected components: 2
Number of nontrivial terminal strongly connected components: 0
0#sets
Strongly connected components: %%0..%%1
Special sets:
  %0: ** terminal nodes **
0#query volatile verbose bspan(true) %0
PATH
Node 0, belongs to strongly connected component %%1
  thinking: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 1
Node 1, belongs to strongly connected component %%1
  thinking: <.2.> + <.3.> + <.4.> + <.5.>
  forks: <.2.> + <.3.> + <.4.> + <.5.>
  withLeft: <.1.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 2
Node 6, belongs to strongly connected component %%1
  thinking: <.3.> + <.4.> + <.5.>
  forks: <.3.> + <.4.> + <.5.>
  withLeft: <.1.> + <.2.>
Running  Input pending in Stdin/Stdout/Stderr
```

```
PROBE - [Stdin/Stdout/Stderr]
File  Edit  View  State  Window  Help
  forks: <.2.> + <.3.> + <.4.> + <.5.>
  withLeft: <.1.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 2
Node 6, belongs to strongly connected component %%1
  thinking: <.3.> + <.4.> + <.5.>
  forks: <.3.> + <.4.> + <.5.>
  withLeft: <.1.> + <.2.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 3
Node 21, belongs to strongly connected component %%1
  thinking: <.4.> + <.5.>
  forks: <.4.> + <.5.>
  withLeft: <.1.> + <.2.> + <.3.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 4
Node 51, belongs to strongly connected component %%1
  thinking: <.5.>
  forks: <.5.>
  withLeft: <.1.> + <.2.> + <.3.> + <.4.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 5
Node 91, belongs to strongly connected component %%0
  withLeft: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
------------------------------------------------
1 paths
------------------------------------------------
0#
Running  Input pending in Stdin/Stdout/Stderr
```

Figure 4: Example screens for step 8.

Many C files from one net description

Since some C compilers are not able to compile a large source file, `prpp` now produces one or more source files from a single net description. If the name of the net description file is `mynet.net`, the source files are put into a directory `mynet.src`. The `prod` program creates the directory for `prpp` if the directory does not already exist. In the net description, any stuff that is not inside a directive should be such that the stuff can be put into a header file that can be included in several source files. The user of `prod` can have his/her own source files, header files and object files in `mynet.src`, with the consequence that all source files become compiled and all object files become linked into the reachability graph generator program `mynet`. Each file produced by `prpp` into `mynet.src` has a name that starts with the underline chracter "_". Therefore, if the user puts his/her own files into `mynet.src`, none of their names should start with the underline character.

Extending the scope of synonyms

Synonyms are as described in Section 3.6. Let `mynet.src` be as above. One of the files produced by `prpp` and located in `mynet.src` has the name `_enum.h`. Whenever *name* is a synonym for *value* in the net description, `_enum.h` has a corresponding macro definition
`#define` *name value*
If a user's own C source file, say `own.c`, in `mynet.src`, starts with
`#include "_enum.h"`
then `own.c` can use any of the names in `_enum.h`, and the meaning of the names is the same as in the net description.

A Cartesian product

An addition to the "*marking ::=*" list in Section 3.9.2.

- *marking* `*` *marking*

  This means the associatively interpreted Cartesian product of the leftmost and the rightmost marking.

  For example, the value of "(5<..>+<.0.>+<.1.>) * (3<..>+<.0.>+<.2.>)" is

  $$15\langle\rangle + 8\langle 0\rangle + 3\langle 1\rangle + 5\langle 2\rangle + \langle 0,0\rangle + \langle 0,2\rangle + \langle 1,0\rangle + \langle 1,2\rangle,$$

  and the value of "(2<.0.>+3<.0,1.>) * (4<.2.>+5<.1,2.>)" is

  $$8\langle 0,2\rangle + 22\langle 0,1,2\rangle + 15\langle 0,1,1,2\rangle.$$

  A set of integer numbers can be expressed using unary tuples, and the Cartesian product operator can then be used for constructing tuples of greater arity. This is particularly useful in "mk" and "let" (introduced below) when the set of values of a field or variable is not an interval.

Direct limits for input variable combinations

Sometimes place limits are too coarse in determining the possible combinations of input variables of a transition. Now it is possible to specify an upper bound to the set of possible combinations directly. The specification is put inside a transition description. It is of the form "let *single-tuple-expression* `<=` *marking*;" where *single-tuple-expression* is like in Section 3.9.3

and *marking* like in Section 3.9.2 except that the fields of *single-tuple-expression* can now be plain variable names only whereas *marking* is not allowed to contain references to places. Moreover, *single-tuple-expression* must contain each input variable name of the transition exactly once and is not allowed to contain anything else.

The truth value of the "<=" operation is determined just like the corresponding binary value on page 15 if we assume that 1 represents "true" and 0 represents "false". In other words, we require that every combination of input variable values must be in the multiset presented by *marking*. "let" affects enabledness in the same way as "gate" but is more economical.

To keep things clear, "let" is forbidden in a net that contains place limits. (See the description of the "-R" option of `prpp` later in this appendix.)

Let's consider the net in Section 4.6. We can first drop the place limits as follows.

```
#place p mk(<.2..n.>)
#place q
```

Then we can replace the only gate expression of the net by a "let" specification.

```
#trans u
    let <.x.> <= <.4..20.> + <.23..n.>;
    in  { p: <.x.>; q: <.x - 1.>; }
    out { p: <.x - 1.>; q: <.x.>; }
#endtr
```

An advanced example: the transition at the beginning of page 12 can be transformed into the following equivalent form.

```
#trans t2_3
    let <.x, r, m, r2.> <= <.1..N, 1..L.> * (<.E, 1..L.> + <.S, 1.>);
    in  { A: <.r.> + (m == E)<.r2.>; W: <.x, m.>; }
    out { U: (m == E)<.x, m, r, r2.> + (m != E)<.x, m, r, r.>; }
#endtr
```

A firing sequence on an arrow

Page 29, line 8:

"An *arrow* corresponds to a fired transition instance."

An updated version:

"An *arrow* corresponds to a fired sequence of transition instances such that the first element of the sequence can be any transition instance, whereas all other elements of the sequence, if any, are invisible transition instances."

Page 39, lines 8 and 9:

"The following holds for those immediate successor arrows y of x that correspond to transition instances in the value of *firing_class*."

An updated version:

"The following holds for each immediate successor arrow y of x that corresponds to a sequence that has at least one transition instance in the value of *firing_class*".

`prod` shell script

The `UNIX` version of the batch program `prod` (sections 6 and 7.1) is now a plain shell script. The script is called in the same way as the old program, except that there is no option that would start with "-". You can copy the script for yourself and then modify it. The `UNIX` version has no `prodfile` any more.

A new option for prpp

Page 45, lines 13 and 14:

```
  prpp [-AaCL] [-u integer] [-D macro=value]...
       [-U macro]... [-I directory]... [-t] file
```

An updated version:

```
  prpp [-AaCLR] [-u integer] [-D macro=value]...
       [-U macro]... [-I directory]... [-t] file
```

An addition to the description of options in Section 7.2:

- **-R**

  Place limits ("lo" and "hi") are not allowed. Instead, possible combinations of input variables of transitions are defined by means of "let". If "-R" is not used, place limits are supported and "let" is forbidden.

New options for reachability graph generators

Page 46, line 15:

```
mynet.exe [-.BCcDdEefiSsuVvxyz] [-b n] [-g n] [-m n] [-r n]
```

An updated version:

```
mynet [-.BCcDdEefiLPqSsuVvXxYyZz] [-b n] [-g n] [-m n] [-r n]
```

Additions to the description of options in Section 7.3:

- **-L**

  The stubborn set method is used, and an exhaustive cardinality minimization w.r.t. enabled transition instances is carried out. The actual minimization task is performed by the `Smodels` library (see `http://saturn.tcs.hut.fi/pub/smodels/`). The logic program for `Smodels` is revised at each encountered nonterminal marking. An error message is displayed if the generator program has been created without appropriate inclusion of the `Smodels` library. Otherwise "-L" overrides each of "-s", "-d" and "-Z".

- **-P**

  A `PEP` (see `http://theoretica.Informatik.Uni-Oldenburg.DE/~pep/`) low-level net description file is produced. If the PROD net description file is `mynet.net`, `mynet.ll_net` is the corresponding `PEP` net description file. (In case of `Windows-PROD`, the `PEP` net description file is `mynet.pep` instead of `mynet.ll_net`.) "-P" is ignored if the `PROD` net description file has any `#prec`, `#tester` or `#verify` declaration or if the reachability graph is generated without unfolding. In case that only the `PEP` net description file is of interest, a good combination of options is "-u -P -g 1".

  Some algorithms in `PEP` assume that the given low-level net is 1-safe. However, since the input syntax of `PEP` supports place-transition nets of any kind, `PROD` does not care whether or not the low-level net is 1-safe.

- **-q**

  This option saves storage space by eliminating some "intermediate" markings. However, no information about the firing of transitions is lost. "-q" preserves all CFFD properties and is correct and complete in on-the-fly verification. Moreover, "-q" does not affect the abilities of the stubborn set method, the sleep set method nor the symmetry method. If "-q" is not used, then each arrow in the reachability graph corresponds to a single transition instance.

- **-X**

  This option is a variant of "-L". The difference is that "-X" uses a single logic program which is parameterized by a so called "compute-rule". Such a logic program typically has much more rules than any of the marking-dependent logic programs that the option "-L" would construct. Consequently, "-X" tends to be slower than "-L" even though one might expect the opposite. (It should be easy for `Smodels` to compute a useful lower bound to the set of ordinary rules that are directly or indirectly deactivated by the "compute-rule". The rules found in this way could then be imagined to be nonexistent as long as the "compute-rule" is not changed.) An error message is displayed if the generator program has been created without appropriate inclusion of the `Smodels` library. Otherwise "-X" overrides each of "-s", "-d", "-Z" and "-L".

- **-Y**

  If an LTL formula is being verified and either "-d" or "-L" is used, a refined version of the LTL-preserving stubborn set method is chosen. This version is not CFFD-preserving. It is more goal-oriented. The structure of the formula is utilized. "-Y" is ignored and a warning message is displayed when "-Y" is meaningless or not supported. Otherwise "-Y" overrides both "-C" and "-Z".

- **-Z**

  If the deletion algorithm of the stubborn set method (the "-d" option) is used, then the algorithm is run possibly several times at a marking, in order to find a stubborn set that has the least number (with respect to the definition of stubbornness) of enabled transition instances. (Note that there would be no easy way to do this with the incremental algorithm.) Success is guaranteed if the least number is one, or if at most five transition instances are enabled at the marking.

A new CTL model checker in `probe`

A new CTL model checker has been included in `probe`. You need to generate the full reachability graph of the system to obtain correct results from the model checking (i.e. the stubborn set method should not be used, and not any other reductive method either). It is used as follows when using the ph.net sample net as an example:

```
%: cp $PRODPATH/sample/ph.net .
%: prod ph.init
%: ./ph -.
.......................
%: probe ph
0#check ef(withLeft >= <.3.>)
Check CTL formula.
Evaluating atomic subformulas.
Evaluating CTL subformulas.
Result: true
Path:
0 [> 1 [> 6 [> 21
0#look 21
Node 21
  thinking: <.4.> + <.5.>
  forks: <.4.> + <.5.>
  withLeft: <.1.> + <.2.> + <.3.>
------------------------------------------------
```

```
0#check au(true, withRight >= <.4.>)
Check CTL formula.
Evaluating atomic subformulas.
Evaluating CTL subformulas.
Result: false
Path:
0 [> 1 [> 6 [> 21 [> 51 [> 91 [> 91
0#quit
```

The supported CTL formulas are: `ax(f)`, `af(f)`, `ag(f)`, `au(f1, f2)`, `ex(f)`, `ef(f)`, `eg(f)`, and `eu(f1, f2)`. Note the counterexamples and witnesses facility of the model checker for easier debugging of specifications. A counterexample path is produced if a universally quantified formula is *false*, and a witness path is produced if a existentially quantified formula is `true`.

You might notice that checking the first CTL formula is somewhat slower than subsequent formulas. This is to be expected because the rechability relation is processed into memory before the first formula is checked. For more information on the CTL model checker implementation see [1].

Syntax of CTL formulas

An addition to Section 5.3, Ordinary commands, page 32.

check *ctl_formula*

This command checks the CTL formula *ctl_formula*. (See the description of *ctl_formula* in Section 5.4.4 .)

An addition to section 5.4.4 Formulas, page 38.
ctl_formula ::=

- **(** *ctl_formula* **)**

  This has the value of *ctl_formula*.

- *expression*

  The result of this is a boolean truth value B. Let x be the evaluation node. If the value of *expression* in x is 0, then B is false. Otherwise B is true.

- **false**

  The value of this is the boolean value false.

- **true**

  The value of this is the boolean value true.

- **not** *ctl_formula*

  The result of this is a boolean truth value B. Let x be the evaluation node. If the value of *ctl_formula* in x is true, then B is false. Otherwise B is true.

- **eu (** *ctl_formula* , *ctl_formula* **)**

  The result of this is a boolean truth value B. Let M be the Kripke structure corresponding to the reachability graph of the system, x be the evaluation node, f_1 the leftmost *ctl_formula*, and f_2 the rightmost *ctl_formula*. B is true iff for some fullpath $(s_0, s_1, \ldots)$,

such that $s_0 = x$,
$\exists i \, [i \geq 0 \wedge M, s_i \models f_2 \wedge \forall j \, [0 \leq j < i \Longrightarrow M, s_j \models f_1]]$.

- **au** (*ctl_formula* , *ctl_formula*)

  The result of this is a boolean truth value B. Let M be the Kripke structure corresponding to the reachability graph of the system, x be the evaluation node, f_1 the leftmost *ctl_formula*, and f_2 the rightmost *ctl_formula*. B is true iff for all fullpaths $(s_0, s_1, \ldots)$, such that $s_0 = x$,
  $\exists i \, [i \geq 0 \wedge M, s_i \models f_2 \wedge \forall j \, [0 \leq j < i \Longrightarrow M, s_j \models f_1]]$.

- **ef** (*ctl_formula*)

  This is a shorthand for the *ctl_formula:* `eu (true, ctl_formula)`.

- **af** (*ctl_formula*)

  This is a shorthand for the *ctl_formula:* `au (true, ctl_formula)`.

- **eg** (*ctl_formula*)

  This is a shorthand for the *ctl_formula:* `not (af (not (ctl_formula)))`.

- **ag** (*ctl_formula*)

  This is a shorthand for the *ctl_formula:* `not (ef (not (ctl_formula)))`.

- **ex** (*ctl_formula*)

  The result of this is a boolean truth value B. Let M be the Kripke structure corresponding to the reachability graph of the system, R the reachability relation between system states, x the evaluation node, and f_1 the subformula *ctl_formula*. B is true iff iff $M, t \models f_1$ for some state t such that $(x, t) \in R$.

- **ax** (*ctl_formula*)

  The result of this is a boolean truth value B. Let M be the Kripke structure corresponding to the reachability graph of the system, R the reachability relation between system states, x the evaluation node, and f_1 the subformula *ctl_formula*. B is true iff iff $M, t \models f_1$ for all states t such that $(x, t) \in R$.

- *ctl_formula* **and** *ctl_formula*

  The result of this is a boolean truth value B. Let x be the evaluation node. If the value of the leftmost *ctl_formula* in x is true, then B is the value of the rightmost *ctl_formula* in x. Otherwise B is false.

- *ctl_formula* **or** *ctl_formula*

  The result of this is a boolean truth value B. Let x be the evaluation node. If the value of the leftmost *ctl_formula* in x is true, then B is true. Otherwise B is the value of the rightmost *ctl_formula* in x.

Command line editing option in `probe`

The Unix version of `probe` can now be compiled with GNU-readline library for command line history and editing. Edit the Makefile to enable this feature which is off by default. Many Unix systems have this library installed (look for `/usr/lib/libreadline.a`), but you can also install the required readline-library yourself from most GNU mirror sites.

Redirection of `probe` output

This enables you to for example use more as a pager inside `probe`. It also enables to make more complex filters for `probe` output (WIN32 and Unix versions only). Syntax: Run a filter program with stdin and out redirected:
`probe-command |>! filter-program`
For example to use more as a pager for probe output:
`0#showends all succs |>! more`
You can also redirect `probe` output into a file:
`probe-command |> new-file`
`0#showends all succs |> foo`
or append to an existing file:
`probe-command |>> existing-file`
`0#look |>> foo`

Copyright changed to GNU Copyleft

The `PROD` tool has been placed under `GNU Copyleft` as of version 3.2. See the `COPYING` file for details on the licencing conditions. If you need to use some of the code with even less restrictive licencing, please contact us.

Bibliography

References

[1] Heljanko, K.: *Model Checking the Branching Time Temporal Logic CTL*. Research Report A45, Digital Systems Laboratory, Helsinki University of Technology, Espoo, 1997. Available as: `http://saturn.hut.fi/pub/reports/A45.ps.Z`.