HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **B:** Technical Reports
No. **13**; August 1995

ISSN 0783–540X
ISBN 951–22–2707–X

PROD REFERENCE MANUAL

Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen,
and Tino Pyssysalo

Digital Systems Laboratory
Department of Computer Science
Helsinki University of Technology
Otaniemi, FINLAND

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
FIN–02150 ESPOO, FINLAND

HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **B:** Technical Reports
No. **13**; August 1995

ISSN 0783–540X
ISBN 951–22–2707–X

PROD Reference Manual

Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen, and Tino Pyssysalo

Abstract: PROD is a Pr/T-net reachability analysis tool that supports on-the-fly verification of linear time temporal properties with the aid of the stubborn set method. Branching time temporal properties can be verified, too.

Keywords: Pr/T-nets, on-the-fly verification, stubborn sets

Printing: TKK Monistamo; Otaniemi 1995

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
FIN–02150 ESPOO, FINLAND

Phone: $\frac{90}{+358\text{–}0}$ 4511
Telex: 125 161 htkk fi
Telefax: +358–0–465 077
E-mail: lab@saturn.hut.fi

Contents

1 Introduction 1

2 Getting started 2

3 The net description language 6

3.1 Syntax conventions 6

3.2 Places 7

3.3 Transitions 8

3.4 Precedences 12

3.5 Macros 13

3.6 Synonyms 13

3.7 Conditional compilation 14

3.8 File inclusion 14

3.9 Integer and marking expressions 14

4 On-the-fly verification 18

4.1 Verification with a tester 18

4.2 Detection of deadlocks 20

4.3 Detecting unexpected multiplicities 20

4.4 The dining philosophers revisited 20

4.5 Linear time temporal properties 22

4.6 A buffer example 24

4.7 Dekker's algorithm 25

5 The query program probe 29

5.1 Command line conventions 29

5.2 Literal commands 30

5.3 Ordinary commands 32

5.4 Expressions, formulas and such 35

5.5 probe in on-the-fly verification 40

5.6 `probe` in non-on-the-fly verification 40

5.7 Connection to CTL 41

6 The batch program `prod` 41

6.1 The syntax of a command file 42

6.2 An example 43

6.3 `prod` in `MS-DOS` 44

7 The subprograms of `PROD` 44

7.1 prod 44

7.2 prpp 45

7.3 The reachability graph generator 46

7.4 strong 50

7.5 probe 51

7.6 araprod 52

Acknowledgements 54

How to get `PROD` 54

References 54

1 Introduction

Reachability analysis, also known as *exhaustive simulation* or *state space generation*, is a powerful formal method for detecting errors in such concurrent and distributed systems that have a finite state space. It suffers from the so called *state space explosion problem*, however: the state space of the system can be far too large with respect to the time and other resources needed to inspect all states in the space. Fortunately, errors can be detected in a variety of cases without inspecting all reachable states of the system. Techniques alleviating the state space explosion problem have been designed. Among such techniques, Valmari's *stubborn set method* [16, 17, 18, 19, 20] is one of the most promising.

On-the-fly verification of a property means that the property is verified during state space generation, in contrary to the traditional approach where properties are verified after state space generation. As soon as it is known whether the property holds, the generation of the state space can be stopped. Since an erroneous system can have a much larger state space than the intended correct system, it is important to find errors as soon as possible. On the other hand, even in the case that all states become generated, the overhead caused by on-the-fly verification, compared to non-on-the-fly verification, is often negligible.

Pr/T-nets [6, 10] are a widely used model for concurrent and distributed systems. `PROD`, a Pr/T-net reachability analysis tool, has been developed by a group of researchers of Digital Systems Laboratory and students of Helsinki University of Technology. On-the-fly verification of *linear time temporal properties* [5, 9] with the aid of the stubborn set method has been implemented in `PROD`. *Branching time temporal properties* [2, 5, 15] can be verified, too, though only after state space generation and without the stubborn set method.

`PROD`'s net description language is the C preprocessor language extended with net description directives. A net description is compiled into an executable reachability graph generator program. `PROD` consists of a net description language preprocessor `prpp`, a reachability graph generator program, a program called `strong` computing the strongly connected components of the graph, a graph query program `probe`, and a batch program `prod`. (`PROD` is the whole tool, `prod` a part of it.)

In addition, there is an interconnection between `PROD`and the `ARA` tool [21]. `ARA` is a LOTOS reachability analysis tool developed in the Technical Research Centre of Finland. The interconnection between `PROD` and `ARA` consists of a program called `araprod` and a set of options for the reachability graph generator program of a net.

The rest of this manual has been organized as follows. An introductory example of the usage of `PROD` is given in Section 2. The net description language is presented in Section 3, except the on-the-fly verification directives. Section 4 is devoted to on-the-fly verification. The reachability graph inspection program, `probe`, and the batch program, `prod`, are presented in Sections 5 and 6, respectively. Finally, Section 7 explains how the subprograms of `PROD` can be run.

2 Getting started

This section gives an introductory example of the usage of PROD. We shall model the classical problem of dining philosophers which has been introduced by Edsger W. Dijkstra. The quoted description of the problem is from [3].

"We now turn to the problem of the Five Dining Philosophers. The life of a philosopher consists of an alternation of thinking and eating:

```
cycle begin think;
            eat
      end
```

Five philosophers, numbered from 0 through 4 are living in a house where the table is laid for them, each philosopher having his own place at the table:

[[A figure has been omitted.]]

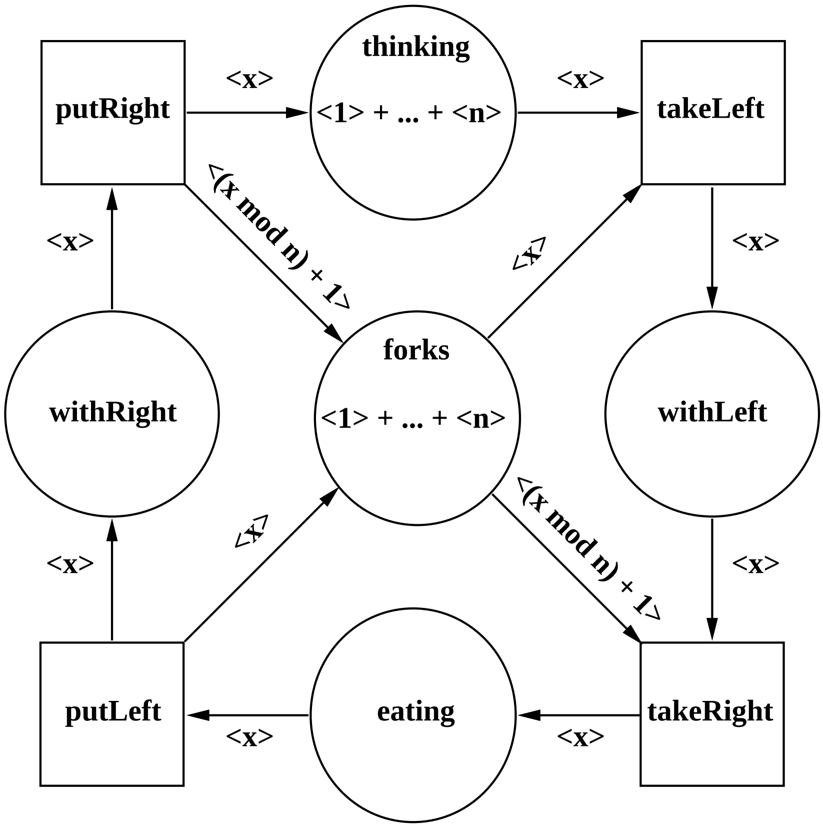
Their only problem — besides those of philosophy — is that the dish served is a very difficult kind of spaghetti, that has to be eaten with two forks. There are two forks next to each plate, so that presents no difficulty; as a consequence, however, no two neighbours may be eating simultaneously.

A very naive solution associates with each fork a binary semaphore with the initial value =1 (indicating that the fork is free) and, naming in each philosopher these semaphores in a local terminology, we could think the following solution for the philosopher's life adequate

```
cycle begin think;
            P(left-hand fork); P(right-hand fork);
            eat;
            V(left-hand fork); V(right-hand fork);
      end
```

[[The quotation ends.]]"

The following figure presents a Pr/T-net model of the problem of dining philosophers. The generalization of the problem of five philosophers into a problem of n philosophers is obvious. We have chosen to number the philosophers from 1 through n, instead of 0 through $n-1$. Below the figure, we have the same net written in PROD's description language. (You do not have to rewrite the net because it is included in the PROD package. See page 54 for more information.) The number of philosophers, n, is fixed by the net preprocessor program prpp. The default value for n is five. One can specify any other value by giving prpp an option of the form '-Dn=value'.



```
#ifndef n
#define n 5
#endif
#define LEFT(x)  (x)
#define RIGHT(x) (1 + ((x) % n))
#place thinking  lo(<.1.>) hi(<.n.>) mk(<.1..n.>)
#place forks     mk(<.1..n.>)
#place withLeft  lo(<.1.>) hi(<.n.>)
#place eating    lo(<.1.>) hi(<.n.>)
#place withRight lo(<.1.>) hi(<.n.>)
#trans takeLeft
   in  { thinking: <.ph.>; forks: <.LEFT(ph).>; }
   out { withLeft: <.ph.>; }
#endtr
#trans takeRight
   in  { forks: <.RIGHT(ph).>; withLeft: <.ph.>; }
   out { eating: <.ph.>; }
#endtr
#trans putLeft
   in  { eating: <.ph.>; }
   out { withRight: <.ph.>; forks: <.LEFT(ph).>; }
#endtr
#trans putRight
   in  { withRight: <.ph.>; }
   out { thinking: <.ph.>; forks: <.RIGHT(ph).>; }
#endtr
```

By using macros, we can avoid the effort of repeating a complicated expression. RIGHT(x) is the right-hand and, for uniformity, LEFT(x) the left-hand fork of philosopher x.

A non-empty initial marking of a place is denoted by ‘mk’. ‘<.’ is the left and ‘.>’ the right angle bracket. ‘<.1..n.>’ means ‘$\sum_{i=1}^{n}$<.i.>’. Without the operator ‘..’, we would have to write a sum such as ‘<.1.>+<.2.>+ <.3.>+<.4.>+<.5.>’ explicitly.

The ‘hi(<.n.>)’ definition denies the corresponding place all tuples with a value more than n. The meaning of ‘lo’ is analogous. In this particular net, ‘lo’ and ‘hi’ seem somewhat unnecessary because there would not ever be any of the denied tuples anyway. The reason for these restrictions on this net is to make the net easily unfoldable into a P/T-net. The place ‘forks’ needs neither ‘lo’ nor ‘hi’ because the restrictions on the other places are sufficient to determine the ‘somewhere enabled’ transition instances. A transition instance is ‘somewhere enabled’ if it is enabled at some, not necessarily reachable, marking that respects the ‘lo’ and ‘hi’ restrictions.

We now consider the generation and inspection of a reachability graph. Let `ph.net` be a file containing the above net description. The first thing to do is to create the reachability graph generator program. We create it as follows. The number of philosophers is 5, i.e. the default value.

```
kva@mimas.hut.fi 1: prod ph.init
```

We then generate the full reachability graph.

```
kva@mimas.hut.fi 2: ph
```

For completeness, we compute the strongly connected components of the graph.

```
kva@mimas.hut.fi 3: strong ph
```

We can now inspect the graph. We first observe that the graph has one terminal node and two strongly connected components. From this we conclude that all nodes except the terminal node are reachable from each other.

```
kva@mimas.hut.fi 4: probe ph
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
```

We then see that the terminal node corresponds to a terminal marking where each philosopher holds his left-hand fork. We also see that one of the shortest paths to the marking is such that the philosophers take their left-hand forks in the order in which they are numbered.

```
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.>
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#quit
```

The generation of the full reachability graph is wasteful if we are interested in terminal markings only. It is much better to use the stubborn set method. For any net, the stubborn set method finds all reachable terminal markings. In the following, we generate a reduced reachability graph by using the stubborn set method. The reduced graph is considerably smaller than the full reachability graph. It can be shown [16] that in the case of n philosophers, the full reachability graph has $3^n - 1$ nodes and $n(2 \cdot 3^{n-1} - 1)$ arrows whereas the stubborn set method easily constructs a reduced graph of $3n^2 - 3n + 2$ nodes and $4n^2 - 3n$ arrows only.

```
kva@mimas.hut.fi 5: ph -s

kva@mimas.hut.fi 6: probe ph
0#statistics
Number of nodes: 62
Number of arrows: 85
Number of terminal nodes: 1
Number of nodes that have been completely processed: 62
Strongly connected components have not been computed
0#quit
```

The detection of terminal markings is the most simple application of the stubborn set method. We shall see some more advanced applications in Section 4. PROD has also two other reductive reachability graph generation methods, the *sleep set method* and the *symmetry method*, that can be used in certain cases. The description of graph generation options in Section 7.3 explains the applicability and usage of the different reachability graph generation methods of PROD.

When you have finished your analysis, you can do as follows, in order to remove the files created during the analysis.

```
kva@mimas.hut.fi 7: prod ph.clean
```

3 The net description language

As said in Section 1, PROD's net description language is the C preprocessor language extended with net description directives. A net description is compiled into an executable reachability graph generator program. The first part of the compilation is performed by prpp, the net description language preprocessor of PROD. Given a net description, prpp produces a source file for a C compiler. In addition, prpp produces some other files needed in the generation and inspection of a reachability graph.

This section presents the net description language. However, the directives related to on-the-fly verification are not considered until in Section 4.

3.1 Syntax conventions

- **Bolded** parts of statements are written into net description file as such.
- *italic* parts of syntax are replaced by parameters given in statement description part.
- Square brackets, [ ], mean that items inside the square brackets are optional.
- Three consecutive dots, ..., stand for repetition.

3.1.1 Identifiers

Identifiers begin with a letter, an underscore ("_") or a dollar ("$"), and may continue with a number of letters, digits, underscores or dollars. Identifiers are case-sensitive.

3.1.2 Line continuation

The end of a *logical line* is the ending delimiter for some directives in the net description language. A logical line consists of one or more physical lines in such a way that each of the physical lines, except the last one, ends with a backslash. The following is an example of how we can make a declaration in a single logical line without having to make it in a single physical line.

```
#place cci_slave_empty lo(<.0, 0.>) \
    hi(<.MAX_CCI_SLAVE, MAX_CCI_SLAVE_QUEUE.>) \
    mk(<.0..MAX_CCI_SLAVE, 0..MAX_CCI_SLAVE_QUEUE.>)
```

3.2 Places

Pr/T-net places are declared with the following syntax.

#place *place-name* [**lo(***lower-limit***)**] [**hi(***upper-limit***)**] [**mk(***initial-marking***)**]

- *Place-name* is a unique identifier for a place.
- *Lower-limit*, *upper-limit* and *initial-marking* have the same syntax as *marking* in Section 3.9.2. If *lower-limit*, *upper-limit* or *initial-marking* contains the name of some already defined place, the name refers to the initial marking of the place. (So, a name of a place in a limit does not refer to any limit of the place. If you want to parameterize a limit expression to be used in several places, you should use macros.)

If l_j is the jth field in a tuple of arity n in the value of *lower-limit*, the place cannot contain tuples of arity n where the jth field is less than l_j.

For example, "lo(<.3.>+<.2,4.>)" has the effect that the field of a unary tuple cannot be less than 3, the first field of a binary tuple cannot be less than 2, and the second field of a binary tuple cannot be less than 4. (Note that "lo(<.2.>+<.3.>+<.1,4.>+<.2,3.>)" has exactly the same effect as "lo(<.3.>+<.2,4.>)".)

Correspondingly, if h_j is the jth field in a tuple of arity n in the value of *upper-limit*, the place cannot contain tuples of arity n where the jth field is greater than h_j.

For example, "hi(<.6.>+<.4,8.>)" has the effect that the field of a unary tuple cannot be greater than 6, the first field of a binary tuple cannot be greater than 4, and the second field of a binary tuple cannot be greater than 8. (Note that "hi(<.6.>+<.7.>+<.4,9.>+<.5,8.>)" has exactly the same effect as "hi(<.6.>+<.4,8.>)".)

If there is no mk, the initial marking of the place is empty.

A place declaration must fit in one logical line.

In the below example, every possible binary tuple of `cci_slave_empty` occurs exactly once in the initial marking.

```
#place cci_slave_empty lo(<.0, 0.>) \
    hi(<.MAX_CCI_SLAVE, MAX_CCI_SLAVE_QUEUE.>) \
    mk(<.0..MAX_CCI_SLAVE, 0..MAX_CCI_SLAVE_QUEUE.>)
```

If the macros `MAX_CCI_SLAVE` and `MAX_CCI_SLAVE_QUEUE` have values 1 and 2, respectively, then the initial marking could be rewritten in the following long form:

```
    mk(<.0,0.>+<.0,1.>+<.0,2.>+<.1,0.>+<.1,1.>+<.1,2.>)
```

3.3 Transitions

Transitions are declared with the following syntax. Note that all parts, except the name of the transition, are optional.

#**trans** *transition-name* [**fd(***list* **)**]
[**in** {*place-name : input-expression; ...*}]
[**out** {*place-name : output-expression; ...*}]
[**gate** *gate-expression*;]
[**comp** *C-block*]
#**endtr**

- *Transition-name* is an identifier for transition name. Each transition name must be unique.
- *List* is a list of variables. No variable can occur more than once in the list. Variables not occurring in any *input-expression* or *ouput-expression* of the transition are not allowed.
- *Place-name* is an identifier for an existing place.
- *Input-expression* is a multituple expression of the form described in Section 3.9.3. A variable is an *input variable* of a transition if and only if it occurs in some input expression of the transition. Input variables are of type `unsigned long`. An input tuple is *strong* if and only if it has a non-zero constant multiplier or no multiplier. An input variable is *strong with respect to an input tuple* if and only if the tuple has a field that contains the name and nothing but the name of the variable. If an input variable x is not strong with respect to any strong input tuple, x must be strong with respect to all those input tuples that contain x, and x must neither occur in any strong input tuple nor in any multiplier of any input tuple. (This condition is sufficient for unifiability.)

- *Output-expression* is a multituple expression of the form described in Section 3.9.3. A variable is an *output variable* of a transition if and only if it is not an input variable of the transition and occurs in some output expression of the transition. Output variables are of type `unsigned long`. The above variable restrictions do not apply to output expressions.
- *Gate-expression* is an integer-valued expression of the C language and can contain variables, function calls and (square bracket style) array references. However, output variables are not allowed.
- *C-block* is a block written in the C language, beginning and ending with a brace.

We strongly suggest that you do not define unnecessary variables. A variable can be considered unnecessary if its value is constant or can uniquely be expressed by means of other variables. Unnecessary input variables may cause difficulties if you want to use the stubborn set method or any method that requires an unfolding of the net. (See Section 7.3.)

The **fd** variables are meaningful only if the net has at least one "`#prec`" declaration. We shall return to these variables in Section 3.4.

The `comp` block is executed once for each input variable value combination corresponding to the markings of the input places, satisfying the gates and respecting the output place limits. (An input variable value combination satisfies a gate if and only if the value of the expression of the gate is not 0 when evaluated with that combination.) The block has four special macros: the nullary macros `Accept()`, `Visible()` and `Invisible()`, and the unary macro `Precedence(class)`. These macros can be called in the same way as functions that have no return value. The `comp` block can be omitted if there is no output variable. Then the effect is as if the following had been given: `comp { Accept(); }`.

An instance of the transition is enabled if and only if the instance is accepted by `Accept()`. The output variables, if such exist, should be set before used, thus at least before the first `Accept()`. The `Accept()` macro accepts an instance if and only if the instance respects the output place limits.

A place limit violation that prevents a transition instance from being enabled is no error. The reason is that the efficiency of unfolding is considered more important than the detection of non-catastrophic questionable features in the net description. (From one point of view, the C language as such is questionable because you can do almost anything in C.) The following example motivates this choice.

```
#trans t
    in  { p: <.x.>; ... }
    gate ...;
    out { q: <.x.>; ... }
    comp { ... }
#endtr
```

Let's assume that the above `t` transition is to be unfolded. Clearly, the limits of `p` give us some range for `x`. However, if we get a smaller range by utilizing the limits of q, we may get a significant reduction in the time needed for unfolding. (Note that x is not necessarily the only input variable of t.) To put it bluntly, unfolding is the only reason for place limits in the net description language. Place limits were chosen to the language instead of explicit transition variable limits because the latter would lead to unnecessarily long net descriptions.

A transition instance can be made visible or invisible by calling the macros Visible() and Invisible(). (These two macros are meaningful only in non-on-the-fly verification. Otherwise they are not available. See sections 4.1 and 4.5.) Depending on how the reachability graph generation program is called, either all transition instances are visible by default or all transition instances are invisible by default. Visibility information is essential when option "-A" or "-C" is given to the reachability graph generator. The visibility or invisibility should be chosen before those Accept() calls that concern the intended transition instances.

For example, all instances of the below i transition are invisible. To get a meaning to this transition, you can imagine that a process x makes an internal move from a state to another. With i, process 0 can move from 3 to 2 and from 7 to 6. Respectively, process 2 can move from 2 to 1, from 3 to 2, from 3 to 4, from 4 to 1, etc.

```
#trans i
    in  { p: <.x, y.>; }
    out { p: <.x, z.>; }
    gate (x == 0) || (x == 2);
    comp { Invisible();
        if (x != 0) goto g0;
        if (y == 3) { z = 2; Accept(); goto end; }
        if (y == 7) { z = 6; Accept(); }
        goto end;
    g0: ;
        if (y == 2) { z = 1; Accept(); goto end; }
        if (y == 3) { z = 2; Accept(); z = 4; Accept(); goto end; }
        if (y == 4) { z = 1; Accept(); goto end; }
        if (y == 5) { z = 4; Accept(); z = 6; Accept(); goto end; }
        if (y == 6) { z = 1; Accept(); goto end; }
        if (y == 7) { z = 6; Accept(); z = 8; Accept(); goto end; }
        if (y == 8) { z = 1; Accept(); goto end; }
        if (y == 9) { z = 2; Accept(); z = 8; Accept(); }
    end: ; }
#endtr
```

Transition instances are attached to precedence classes by calling the Precedence macro. (This macro is meaningful only if there is at least one "#prec" declaration. Otherwise it is not available. See Section 3.4.) Precedence takes a single argument which is some precedence class. The default class is 0. The precedence class should be chosen before those Accept() calls that concern the intended transition instances.

We now demonstrate the usage of non-constant tuple multipliers in arc expressions. We start from a net that does not have such multipliers. The below net, practically identical to a net given in [6], is a model of a simple resource management scheme. However, we are here more interested in its syntax than its semantics.

```
#define N 3
#define L 2
#enum S, E
#place A lo(<.1.>) hi(<.L.>) mk(<.1..L.>)
#place C lo(<.1.>) hi(<.L.>)
#place I lo(<.1.>) hi(<.N.>) mk(<.1..N.>)
#place R lo(<.1, E.>) hi(<.N, S.>)
#place U lo(<.1, E, 1, 1.>) hi(<.N, S, L, L.>)
#place W lo(<.1, E.>) hi(<.N, S.>)
#trans t1
    in  { I: <.x.>; }
    out { W: <.x, m.>; }
    comp(m) { Accept(E); Accept(S); }
#endtr
#trans t2
    in  { A: <.r1.> + <.r2.>; W: <.x, E.>; }
    out { U: <.x, E, r1, r2.>; }
#endtr
#trans t3
    in  { A: <.r.>; W: <.x, S.>; }
    out { U: <.x, S, r, r.>; }
#endtr
#trans t4
    in  { U: <.x, E, r1, r2.>; }
    out { C: <.r1.> + <.r2.>; R: <.x, E.>;}
#endtr
#trans t5
    in  { U: <.x, S, r, r.>; }
    out { C: <.r.>; R: <.x, S.>; }
#endtr
#trans t6
    in  { R: <.x, m.>; }
    out { I: <.x.>; }
#endtr
#trans t7
    in  { C: <.r.>; }
    out { A: <.r.>; }
#endtr
```

Using the *merging* technique presented in [6], the above transitions `t4` and `t5` can be replaced by a single transition, `t4_5`, as follows:

```
#trans t4_5
    in  { U: <.x, m, r, r2.>; }
    out { C: <.r.> + (m == E)<.r2.>; R: <.x, m.>;}
#endtr
```

Moreover, by taking a full advantage of the merging technique, the transitions `t2` and `t3` can be replaced by a single transition, `t2_3`, as follows:

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

The only purpose of the above gate is to eliminate redundant transition instances. Without the gate, any value greater than or equal to 1 and less than or equal to `L` would have been valid for `r2` in the case `m != E`. (The limits of the `A` place determine the bounds of `r2`.)

3.4 Precedences

If the net description contains "`#prec`" declarations then the *static priority method* [11, 19, 22] is applied in the generation of the reachability graph.

Let transition instances x and y be enabled at a marking M of the net. (The case where y is x is included in this consideration.) Let a be the precedence class of x and b the precedence class of y. The instance y is not fired at M if a has a priority over b, the length of the **fd** list of y is the same as the length of the **fd** list of x, and for each i, the variable in the ith position in the **fd** list of y has the same value as the variable in the ith position in the fd list of x.

By default, all transition instances are in the precedence class 0. No class has any priority over the class 0. The class 0 has no priority over any class. **Fd** lists are empty by default.

Unfortunately, the static priority method does not automatically preserve any property of the net. Therefore, an error message is given if the method is attempted to be used in on-the-fly verification. (On-the-fly verification is described in Section 4.) On the other hand, as can be seen from Section 7.3, the generator of the reachability graph refuses to use reductive reachability graph generation methods (other than the static priority method) if there is any precedence declaration.

Precedences between precedence classes are declared with the following syntax. Note that precedence classes, except 0, have symbolic names, and it is impossible to try to affect the status of the 0 class.

#prec *precedence-class precedence-operator precedence-class*

- *Precedence-class* is the symbolic name of the precedence class. Symbolic names have numerical values. The values are of type unsigned int and start from 1. The order of class numbers has absolutely nothing to do with the precedence relation.

- *Precedence-operator* is one of <, >, <<, >>.

A precedence declaration must fit in one logical line.

The following list explains the meaning of these operators.

- a < b
  The class b has a priority over the class a.
- a > b
  The class a has a priority over the class b.
- a << b
  If v is b or b < v or b < . . . < v then v has a priority over a.
- a >> b
  If v is b or v < b or v < . . . < b then a has a priority over v.

Below we have an example of a precedence declaration.

```
#prec A >> B > C > D
```

The class A has a priority over B, C and D. The class B has a priority over C but not over D. The class C has a priority over D.

3.5 Macros

Macros can be defined in the following two ways, and also by using "-D" options in the command line of `prpp`. (See Section 7.2.)

#define *identifier symbol-string*
#define *identifier(identifier,...,identifier) symbol-string*

A macro definition must fit in one logical line.

Macros become expanded in the same way as in the C preprocessor language, except that some parts in the net description are expected to be free of any occurrences of macros. However, integer-valued expressions can always be hidden behind macros.

Undefinitions can be given in the following way, and also by using "-U" options in the command line of `prpp`. (See Section 7.2.)

#undef *identifier*

`prpp` writes definitions and undefinitions of macros in order of appearance into a definition file. (Command line definitions and undefinitions appear first of all.) By loading the definition file user may utilize these definitions when inspecting a reachability graph of the net with `probe`. If the name of the net description file is `mynet.net` then the name of the definition file is `mynet.def`.

3.6 Synonyms

A *synonym* resembles a macro without parentheses. However, the value of a synonym is determined automatically and is always some number of type `unsigned long`. In addition, when `probe` displays a tuple field or a transition variable value corresponding to a synonym, it displays the name of the synonym.

Synonyms are defined by lines of the following form.

#**enum** *identifier [, identifier ...]*

The values of the synonyms are determined as follows: The value of the first defined synonym is the greatest element of `unsigned long`. If the value of the kth defined synonym is x, then the value of the $(k+1)$th defined synonym is $x-1$.

This numbering is good for the following reasons.

- The values of synonyms do not form a gap in the middle of the set of the numbers of type `unsigned long`. This is useful in the hopefully rare cases where `probe` displays the name of a synonym while you expect it to display a number.
- You can insert other declarations between synonym declarations without affecting the values of the synonyms.
- You can implement an increasing numbering by reversing the list that you have in mind.

3.7 Conditional compilation

The following directives can be used for conditional compilation: "`#ifdef`", "`#ifndef`" "`#if`", "`#else`", and "`#endif`". They are just like in the C preprocessor language, except that the argument of "`#if`" is an *expression* of the form described in Section 3.9.1. If such expression contains the name of some already defined place, the name refers to the initial marking of the place.

3.8 File inclusion

Files can be included by the "`#include`" directive just like in the C preprocessor language. Directories to be searched can be specified by using "-I" options in the command line of `prpp`. (See Section 7.2.)

3.9 Integer and marking expressions

This section presents the syntax and semantics of those integer and marking expressions that were used in sections 3.2 and 3.3.

3.9.1 Integer expressions

simple_expression ::=

- **card** (*marking*)

  This means the cardinality, i.e. the number of tuples, of the marking. Every occurrence of a tuple is counted. (The syntax and semantics of *marking* are presented in Section 3.9.2.)

For example, the value of **card**(3<..>+2<.0.>+<.4,5.>) is 6.

- (*expression*)

  This has the value of *expression*. (The syntax and semantics of *expression* are presented later in this section.)

- **field**[*expression*]

  This is meaningful only as a subexpression of the rightmost operand of ":". (See Section 3.9.2.) Let M be the value of the leftmost operand of the innermost such ":". If the value of *expression* is i, then **field**[*expression*] means the $(i+1)$th field of some tuple of M. (The leftmost field is the first field.) An error message is displayed if the arity of some tuple in M is less than $i+1$.

- ***digits***

  This means the number obtained by considering *digits*, a string of decimal digits, a decimal number of type `unsigned long`.

expression ::=

- *simple_expression*

- *marking* == *marking*

  The value of this is 1 if the markings are equal. Otherwise the value is 0. (The syntax and semantics of *marking* are presented in Section 3.9.2.)

- *marking* != *marking*

  The value of this is 1 if the markings are not equal. Otherwise the value is 0.

- *marking* <= *marking*

  The value of this is 1 if every tuple of the leftmost marking occurs at least equally many times in the rightmost marking. Otherwise the value is 0.

  For example, "<.1.> <= <.0.> + <.1.>" and "<..> + 2<.0.> <= <..> + 3<.0.>" have the value 1, whereas "<.0.> <= <.1.>" and "<..> + 2<.1.> <= 4<.1.>" have the value 0.

- *marking* >= *marking*

  The value of "A >= B" is equal to the value of "B <= A".

- *marking* < *marking*

  The value of "A < B" is equal to the value of "(A <= B) && (A != B)". Note that the value of "A < B" is not necessarily equal to the value of "!(A >= B)".

  For example, both "<.0.> < <.1.>" and "<.0.> >= <.1.>" have the value 0.

- *marking* > *marking*

  The value of "A > B" is equal to the value of "B < A".

- *op expression*

  Here *op* is "-", "!" or "~". The value is determined just like in the C language.

- *expression op expression*

  Here *op* is "`*`", "`/`", "`%`", "`+`", "`-`", "`>>`", "`<<`", "`<`", "`<=`", "`>`", "`>=`", "`==`", "`!=`", "`&`", "`|`", "`^`", "`&&`", or "`||`". The value is determined just like in the C language.

- *expression* `?` *expression* `:` *expression*

  The value of this determined just like in the C language.

3.9.2 Marking expressions

range ::=

- *expression*

  The value of this is $\{x\}$ where x is the value of the expression.

- *expression* `..` *expression*

  The value of this is $\{z \in \texttt{unsigned long} \mid x \leq z \leq y\}$ where x is the value of the leftmost expression and y is the value of the rightmost expression.

  For example, the value of "`0..2`" is $\{0, 1, 2\}$, whereas the value of "`1..0`" is the empty set.

- `(` *range* `)`

  This has the value of *range*.

rangelist ::=

- *range*

  The value of this is a list of one element that is the value of *range*.

- *rangelist* `,` *range*

  The value of this is the list obtained by inserting the only element of the value of *range* to the tail of the value of *rangelist*.

  For example, the value of "`0..2, 3, 4..5`" is the list "$\{0, 1, 2\}, \{3\}, \{4, 5\}$".

simple_marking ::=

- *place*

  This means the marking of *place*.

- **empty**

  This means the empty marking.

- `<.` *rangelist* `.>`

  This means the marking $\Sigma_{x_1 \in A_1} \ldots \Sigma_{x_n \in A_n} \langle x_1, \ldots, x_n \rangle$ where n is the length and A_k the kth element of the value of *rangelist.* Note that the marking is empty if some A_k is empty.

  For example, `<.1..4.>.` means $\Sigma_{i=1}^{4} \langle i \rangle$, `<.1..2, 2..4.>` means $\Sigma_{i=1}^{2} \Sigma_{j=2}^{4} \langle i, j \rangle$, and `<.1..2, 5..4.>` means the empty marking.

- `<. .>`

  This means the marking $\langle\rangle$.

- (*marking*)

  This has the value of *marking.*

marking ::=

- *simple_marking*
- *marking* & *marking*

  This means the intersection of the leftmost and the rightmost marking. The number of occurrences of a tuple in the result is the minimum of m and n where m is the number of occurrences of the tuple in the leftmost marking and n is the number of occurrences of the tuple in the rightmost marking.

  For example, the value of "(5<..>+<.0.>+<.1.>) & (3<..>+<.0.>+<.2.>)" is $3\langle\rangle + \langle 0 \rangle$.

- *marking* + *marking*

  This means the sum of the leftmost and the rightmost marking. The number of occurrences of a tuple in the result is $m+n$ where m is the number of occurrences of the tuple in the leftmost marking and n is the number of occurrences of the tuple in the rightmost marking.

  For example, the value of "(5<..>+<.0.>+<.1.>) + (3<..>+<.0.>+<.2.>)" is $8\langle\rangle + 2\langle 0 \rangle + \langle 1 \rangle + \langle 2 \rangle$.

- *marking* - *marking*

  The number of occurrences of a tuple in the result is $m - n$ where m is the number of occurrences of the tuple in the leftmost marking and n is the number of occurrences of the tuple in the rightmost marking. However, if m is less than n then the tuple does not occur in the result at all.

  For example, the value of "(5<..>+<.0.>+<.1.>) - (3<..>+<.0.>+<.2.>)" is $2\langle\rangle + \langle 1 \rangle$.

- *simple_marking* : *simple_expression*

  The number of occurrences of a tuple in the result is 0 if the value of *simple_expression* is 0. Otherwise the tuple occurs as many times in the result as in the value of *simple_marking.*

  For example, (<.3,2.>+<.4,5.>+<.5,3.>+<.2,4,3.>):(**field**[0] < **field**[1]) means $\langle 4, 5 \rangle + \langle 2, 4, 3 \rangle$.

- *simple_expression simple_marking*

  The number of occurrences of a tuple in the result is m times n where m is the value of *simple_expression* and n is the number of occurrences of the tuple in the value of *simple_marking*.

  For example, the value of "3(<..>+2<.0.>)" is $3\langle\rangle + 6\langle 0\rangle$.

3.9.3 Multituple expressions

In Section 3.3, we used the notion of a *multituple expression*. It is somewhat different from the notion of a marking expression that was defined in Section 3.9.2.

A multituple expression is of the form

[n_1] *single-tuple-expression* [+ [[n_2] *single-tuple-expression* ...]]

where *single-tuple-expression* is of the form

<.[a_1 , a_2, ...].>

As before, the arity of a tuple can be anything, and the type of a tuple multiplier or field is `unsigned long`. However, now a tuple multiplier or a field is an expression of the C language and can contain variables, function calls and (square bracket style) array references.

For example, "`<.a[x].> + <.x,y.> + f(x,y)<..>`" is a valid multituple expression if the elements of the `a` array and the return values of the `f` function are integer numbers. (They do not have to be of the `unsigned long` type because a cast to `unsigned long` is done anyway.)

4 On-the-fly verification

As said in Section 1, *on-the-fly verification of a property* means that the property is verified during state space generation, in contrary to the traditional approach where properties are verified after state space generation. As soon as it is known whether the property holds, the generation of the state space can be stopped.

In `PROD`, there are essentially two approaches to on-the-fly verification: a tester approach, described in sections 4.1, 4.2, 4.3 and 4.4, and a formula approach, described in sections 4.5, 4.6 and 4.7. (You do not have to rewrite the nets considered in sections 4.4, 4.6 and 4.7 because these nets are included in the `PROD` package. See page 54 for more information.)

4.1 Verification with a tester

The tester approach in `PROD` is based on the approach presented by Valmari [20]. A *tester* in a Pr/T-net is a unique place together with the arcs connected to the place. At any reachable marking, the tester place contains exactly one tuple, and such tuple is unary. The value inside the tuple is the *state* of the tester. An *action* is a transition instance, i.e. a transition with a single combination of values of the variables of the transition. An action is *visible* if and only if the action is connected to the tester place, i.e. there is at least one arc between the tester place and the transition of the action in such a way that the expression on the arc has a nonempty value. (In an arc expression of PROD, a tuple can be multiplied by a non-constant expression.)

We can now associate special meanings with the states of the tester and proceed as in [20]. The algorithm there is directly applicable to the generation of the reachability graph of the net since a reachable marking of the net can be imagined to be a pair of the state of the tester and the state of the actual system. If stubborn sets are wanted, they are computed by a Petri net oriented algorithm which satisfies the conditions mentioned in [20]. The stubborn set method is chosen at the reachability graph generation phase by an option exactly in the same way as in the case of non-on-the-fly verification.

The tester is declared as follows:

#tester *place-name* [**reject(***multiset***)**] [**deadlock(***multiset***)**] \
[**livelock(***multiset***)**] [**infinite(***multiset***)**]

Here we have the name of the tester place, the set of *reject states,* the set of *deadlock monitor states,* the set of *livelock monitor states,* and the set of *infinite path monitor states.* A state of the tester can be in one, in more than one, or in none of these sets. *multiset* can be considered a marking, and the syntax of *multiset* is the same as the syntax of *marking*, given in Section 3.9.2. For any number a, a is a reject state (deadlock monitor state, livelock monitor state, infinite monitor state, respectively) if and only if <.a.> occurs at least once in the value of the *multiset* of **reject** (**deadlock**, **livelock**, **infinite**, respectively).

A tester declaration must fit in one logical line.

The generation of the reachability is automatically stopped and a report is displayed whenever

- a reject state is encountered,
- a deadlock monitor state is encountered and no transition is enabled at the corresponding marking,
- a livelock monitor state is encountered and a loop of invisible actions through the corresponding marking has been found, or
- an infinite path monitor state is encountered and such loop through the corresponding marking has been found that the action immediately following the marking in the loop is visible.

Note that the above list gives the full semantics of these states. If a deadlock means an undesirable reachable marking where no transition is enabled, then deadlock monitor states specify what is undesirable. Correspondingly, if a livelock means an undesirable reachable non-progress loop, then livelock monitor states specify what is undesirable.

(As usual, an execution of an action makes progress if and only if the action is visible.) Infinite path monitor states specify what kind of progress loops are undesirable. Reject states specify markings that are undesirable as such.

4.2 Detection of deadlocks

Let's assume that you have a net with no tester and no `#verify`. If you want to know whether the net has some reachable marking where no transition is enabled, all you need to do is to add the following two lines to the end of the net and then proceed as if your purpose was to generate a reachability graph.

```
#place tester lo(<.0.>) hi(<.0.>) mk(<.0.>)
#tester tester deadlock(<.0.>)
```

If there is some reachable marking where no transition is enabled, then you will get a message about one such marking, and the generation of the reachability graph is stopped automatically.

4.3 Detecting unexpected multiplicities

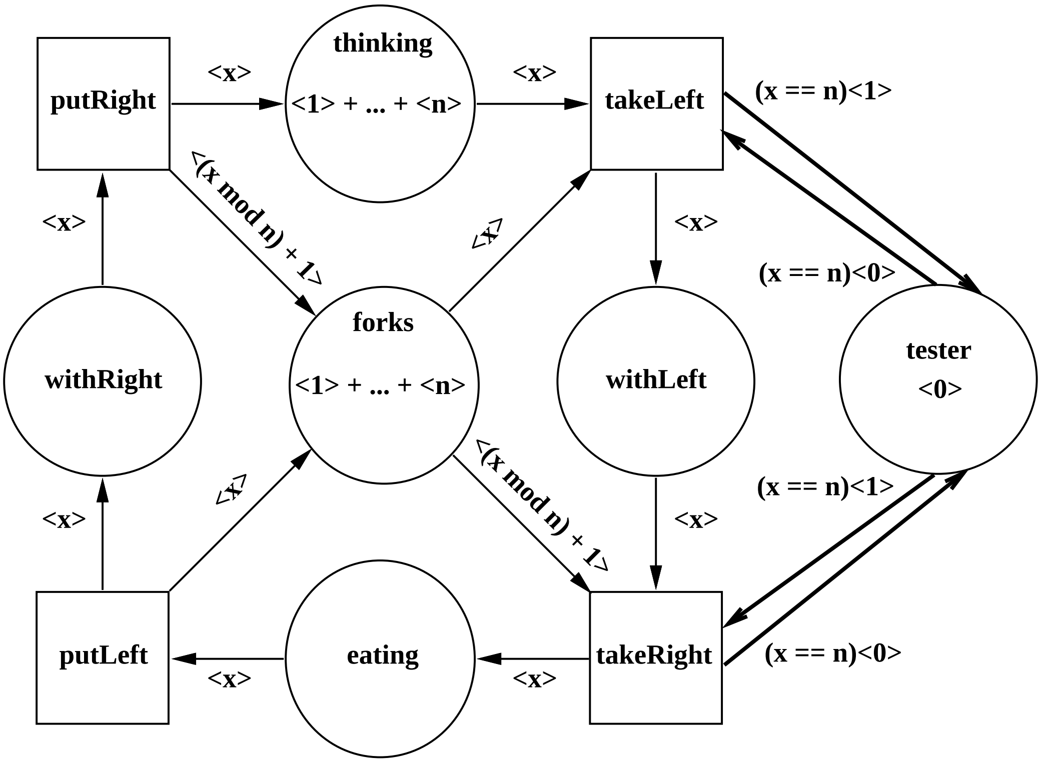
Let's again assume that you have a net with no tester and no `#verify`. Let's further assume that you want to know if some unary tuple occurs more than once in a place `p` in some reachable marking. As in the previous section, you can add a tester to the net. In this case, you can add the following to the end of the net description:

```
#place tester lo(<.0.>) hi(<.1.>) mk(<.0.>)
#tester tester reject(<.1.>)
#trans bad
    in { p: 2<.x.>; tester: <.0.>; }  out { p: 2<.x.>; tester: <.1.>; }
#endtr
```

If some unary tuple occurs more than once in `p` in some reachable marking, then you will get a message about one such marking, and the generation of the reachability graph is stopped automatically.

4.4 The dining philosophers revisited

We now return to the dining philosopher example of Section 2. We demonstrate the usage of a livelock monitor state in the case where there are 1994 philosophers. The stubborn set method is used in the example, otherwise we could not manage with so many philosophers. Our goal is to check if there exists a reachable loop where the nth philosopher holds his left-hand fork, that is whether the system can move is such a way that philosopher n never gets his right-hand fork and thus starves. The only visible actions are those instances of `takeLeft` and `takeRight` where x is equal to n. As one might guess, the reachability graph generator finds a loop of the specified kind. The `probe` part of the example is included for completeness. We suggest that the reader returns to that part of the example after reading the description of `probe`.



```
kva@mimas.hut.fi 1: cat dining.net
#define n 1994
#define LEFT(x)  (x)
#define RIGHT(x) (1 + ((x) % n))
#place thinking  lo(<.1.>) hi(<.n.>) mk(<.1..n.>)
#place forks     mk(<.1..n.>)
#place withLeft  lo(<.1.>) hi(<.n.>)
#place eating    lo(<.1.>) hi(<.n.>)
#place withRight lo(<.1.>) hi(<.n.>)
#place tester    lo(<.0.>) hi(<.1.>) mk(<.0.>)
#tester tester   livelock(<.1.>)
#trans takeRight
   in  { forks: <.RIGHT(x).>; withLeft: <.x.>;
         tester: (x == n)<.1.>; }
   out { eating: <.x.>; tester: (x == n)<.0.>; }
#endtr
#trans takeLeft
   in  { thinking: <.x.>; forks: <.LEFT(x).>;
         tester: (x == n)<.0.>; }
   out { withLeft: <.x.>; tester: (x == n)<.1.>; }
#endtr
#trans putLeft
   in  { eating: <.x.>; }
   out { withRight: <.x.>; forks: <.LEFT(x).>; }
#endtr
```

```
#trans putRight
   in  { withRight: <.x.>; }
   out { thinking: <.x.>; forks: <.RIGHT(x).>; }
#endtr
kva@mimas.hut.fi 2: prod dining.init
kva@mimas.hut.fi 3: dining -s

Livelock reached
Loop 1994 [1> 1996 [0> 1997 [0> 1999 [0> 1994
For more information, start "probe dining" and look at the set %1
kva@mimas.hut.fi 4: probe dining
0#goto 1994
1994#calc withLeft
<.1994.>
1994#succ arrow 1
Arrow 1: transition takeLeft, precedence class 0
  x = 1
to node 1996
------------------------------------------------
1994#next 1
1996#succ arrow 0
Arrow 0: transition takeRight, precedence class 0
  x = 1
to node 1997
------------------------------------------------
1996#next 0
1997#succ arrow 0
Arrow 0: transition putLeft, precedence class 0
  x = 1
to node 1999
------------------------------------------------
1997#next 0
1999#succ arrow 0
Arrow 0: transition putRight, precedence class 0
  x = 1
to node 1994
------------------------------------------------
1999#quit
```

4.5 Linear time temporal properties

Linear time temporal properties [5, 9] can be verified on-the-fly by including a line of the form “`#verify` formula;” in the net. Then `prpp` automatically constructs a corresponding tester, not requiring help from the user. A formula is of the form “expression”, “not A”, “A and B”, “A or B”, “A implies B”, “eventually A”, “henceforth A”, “A until B”, or “A unless B” where “A” and “B” are formulas and “expression” is of the form described in Section 3.9.1.

prpp constructs a tester with infinite path monitor states. The name of the tester place is t_e_s_t_e_r. prpp adds a transition called i_n_i_t_i_a_l_i_z_e to the net, and then this transition is the only enabled transition at the initial marking. prpp eliminates terminating firing sequences by adding a transition called d_u_m_m_y to the net. This transition is enabled if and only if no other transition is enabled.

In the context where no terminating firing sequence exists, verifying a formula means showing that each infinite firing sequence starting from the initial marking satisfies the formula. The rules of satisfaction are as follows. (They correspond to rules given in [5].) An infinite firing sequence satisfies a formula if and only if the infinite sequence of markings corresponding to the firing sequence satisfies the formula. Let then u be an infinite sequence of markings.

- The sequence u satisfies "expression" if and only if the value of the expression is not 0 at the first marking of u.
- The sequence u satisfies "not A" if and only if u does not satisfy "A".
- The sequence u satisfies "A and B" if and only if u satisfies "A" and "B".
- The sequence u satisfies "A or B" if and only if u satisfies "A" or "B".
- The sequence u satisfies "A implies B" if and only if u satisfies "B" or does not satisfy "A".
- The sequence u satisfies "eventually A" if and only if some infinite suffix of u satisfies "A".
- The sequence u satisfies "henceforth A" if and only if every infinite suffix of u satisfies "A".
- The sequence u satisfies "A until B" if and only if there exist a finite (possibly empty) sequence v of markings and an infinite sequence w of markings in such a way that u is vw, w satisfies "B", and for every suffix v' of v, $v'w$ satisfies "A".
- The sequence u satisfies "A unless B" if and only if u satisfies "henceforth A" or "A until B".

The generation of the reachability graph is automatically stopped and a report is displayed whenever an infinite firing sequence starting from the initial marking and not satisfying the formula is found.

If a subformula contains no temporal operator ("eventually", "henceforth", "until" or "unless"), then, for efficiency reasons, it is often good to use "!" instead of "not", "&&" instead of "and", "||" instead of "or", and "<=" instead of "implies" in the subformula.

The stubborn set method can be used in the verification of a linear time temporal formula. However, PROD must then compute an upper estimate of the set of transition instances that can directly change the truth values of the non-temporal subformulas. (A formula is temporal if and only if it contains some "eventually", "henceforth", "until" or "unless".) The estimate computed by PROD consists exactly of those transition instances that are connected to the places mentioned in the formula. Consequently, you should try to minimize the number of such transition instances. Unlike in Section 4.1, here we use the simple rule that a transition instance is connected to a place if and only if there is at least one arc between the place and the high level transition in question. If every transition instance is in connection with one or more places mentioned in the formula, then the stubborn set method does not alleviate the state space explosion at all.

We need the notion of visibility in Section 7. The definition of visibility given in Section 4.1 is not applicable to the verification of a linear time temporal formula. The following definition is applicable: a transition instance is visible if and only if it is connected to some place mentioned in the formula.

4.6 A buffer example

The following example, inspired by [14], is a model of a FIFO buffer. The size of the buffer is n. The place p_1 has the tuple $\langle\rangle$ if the first position of the buffer is empty. Otherwise p_1 has no tuple. For each i between 2 and n, the place p has the tuple $\langle i\rangle$ if and only if the ith position of the buffer is empty. The place q_2 has the tuple $\langle\rangle$ if the second position of the buffer is occupied. Otherwise q_2 has no tuple. The place q_21 has the tuple $\langle\rangle$ if the 21st position of the buffer is occupied. Otherwise q_21 has no tuple. For each i less than 2 or between 4 and 20 or greater than 21, the place q has the tuple $\langle i\rangle$ if and only if the ith position of the buffer is occupied.

```
#ifndef n
#define n 25
#endif
#place p_1 mk(<..>)
#place p lo(<.2.>) hi(<.n.>) mk(<.2..n.>)
#place q lo(<.1.>) hi(<.n.>)
#place q_2
#place q_21
#trans t1
    in  { p_1: <..>; }
    out { q: <.1.>; }
#endtr
#trans t2
    in  { p: <.2.>; q: <.1.>; }
    out { p_1: <..>; q_2: <..>; }
#endtr
#trans t3
    in  { p: <.3.>; q_2: <..>; }
    out { p: <.2.>; q: <.3.>; }
#endtr
#trans t21
    in  { p: <.21.>; q: <.20.>; }
    out { p: <.20.>; q_21: <..>; }
#endtr
```

```
#trans t22
    in  { p: <.22.>; q_21: <..>; }
    out { p: <.21.>; q: <.22.>; }
#endtr
#trans u
    in  { p: <.x.>; q: <.x - 1.>; }
    gate ((x >= 4) && (x <= 20)) || (x >= 23);
    out { p: <.x - 1.>; q: <.x.>; }
#endtr
#trans v
    in  { q: <.n.>; }
    out { p: <.n.>; }
#endtr
#verify eventually((p_1 != empty) && (q_2 == empty) && (q_21 != empty));
```

The argument of the above `#verify` is a linear time temporal formula stating that every infinite behaviour of the buffer goes through some configuration where the first and the second position are empty and the 21st position is occupied. From the following we see that the formula does not hold.

```
kva@mimas.hut.fi 4: prod verfifo.init
kva@mimas.hut.fi 5: verfifo

Illegal infinite path found
Loop 641 [0> 615 [0> 617 [0> [[ output removed ]] [0> 640 [0> 641
For more information, start "probe verfifo" and look at the set %1
```

The above succeeds without the stubborn set method, but if n is greater than 25, it is best to use the stubborn set method. Then n can be, say, 3000. (Run `prpp` without the "-u" option if you want n to be 3000.)

The above net may seem unnecessarily complicated because we have tried to minimize the number of transition instances connected to the places mentioned in the formula. We have done this because of the stubborn set method. (See Section 4.5.)

4.7 Dekker's algorithm

We now show that Dekker's solution to the critical section problem in the case of two processes [3] is free of starvation if neither of the processes is left without processor time forever. The quoted description of Dekker's solution is from [27].

“There are two processes P_1 and P_2, two boolean-valued variables b_1 and b_2 whose initial values are `false`, and a variable k which may take the values 1 and 2 and whose initial value is arbitrary. The ith process ($i = 1, 2$) may be described as follows (where j is the index of the other process):

```
while true do
begin
    <noncritical section>;
    $b_i$ := true;
    while $b_j$ do
        if $k = j$ then begin
                            $b_i$ := false;
                            while $k = j$ do skip;
                            $b_i$ := true
                        end;
    <critical section>;
    $k := j$;
    $b_i$ := false
end;
```

Note that P_i indicates its wish to execute its critical section by setting b_i to `true`; note also the rôle of the variable k. [[The quotation ends.]]”

The below net, `dekker.net`, corresponds to the above algorithm. You can see the correspondence by comparing the names of the places and transitions to the above algorithm. We have chosen the initial value of the variable k of the above algorithm to be 1, but the case where the initial value is 2 is symmetric. To be able to show freeness of starvation, we have included the `scheduler` place in the net. Due to the `choose_i` transition, all scheduling alternatives whatsoever are taken into account. We thus use the same approach as [9].

```
#place noncritical lo(<.1.>) hi(<.2.>) mk(<.1..2.>)
#place wait_set_b_i lo(<.1.>) hi(<.2.>)
#place wait_read_b_j lo(<.1.>) hi(<.2.>)
#place wait_read_k lo(<.1.>) hi(<.2.>)
#place wait_clear_b_i lo(<.1.>) hi(<.2.>)
#place wait_read_k_again lo(<.1.>) hi(<.2.>)
#place wait_set_b_i_again lo(<.1.>) hi(<.2.>)
#place wait_enter lo(<.1.>) hi(<.2.>)
#place critical lo(<.1.>) hi(<.2.>)
#place wait_set_k lo(<.1.>) hi(<.2.>)
#place wait_clear_b_i_again lo(<.1.>) hi(<.2.>)
#place b lo(<.1, 0.>) hi(<.2, 1.>) mk(<.1..2, 0.>)
#place k lo(<.1.>) hi(<.2.>) mk(<.1.>)
#place scheduler lo(<.0.>) hi(<.2.>) mk(<.0.>)
#verify (henceforth(eventually(scheduler == <.1.>) and
                    eventually(scheduler == <.2.>))) implies
        henceforth(((wait_set_b_i >= <.1.>) implies
                    (eventually (wait_set_k >= <.1.>))) and
                   ((wait_set_b_i >= <.2.>) implies
                    (eventually (wait_set_k >= <.2.>))));
#trans choose_i
    in  { scheduler: <.0.>; }
    out { scheduler: <.i.>; }
    comp { i = 1; Accept(); i = 2; Accept(); }
#endtr
#trans do_noncritical
    in  { noncritical: <.i.>; scheduler: <.i.>; }
    out { noncritical: <.i.>; scheduler: <.0.>; }
#endtr
#trans desire
    in  { noncritical: <.i.>; scheduler: <.i.>; }
    out { wait_set_b_i: <.i.>; scheduler: <.0.>; }
#endtr
#trans set_b_i
    in  { wait_set_b_i: <.i.>; b: <.i, 0.>; scheduler: <.i.>; }
    out { wait_read_b_j: <.i.>; b: <.i, 1.>; scheduler: <.0.>; }
#endtr
#trans read_b_j_true
    in  { wait_read_b_j: <.i.>; b: <.3 - i, 1.>; scheduler: <.i.>; }
    out { wait_read_k: <.i.>; b: <.3 - i, 1.>; scheduler: <.0.>; }
#endtr
#trans read_b_j_false
    in  { wait_read_b_j: <.i.>; b: <.3 - i, 0.>; scheduler: <.i.>; }
    out { wait_enter: <.i.>; b: <.3 - i, 0.>; scheduler: <.0.>; }
#endtr
#trans read_k_eq_j
    in  { wait_read_k: <.i.>; k: <.3 - i.>; scheduler: <.i.>; }
    out { wait_clear_b_i: <.i.>; k: <.3 - i.>; scheduler: <.0.>; }
#endtr
#trans read_k_ne_j
    in  { wait_read_k: <.i.>; k: <.i.>; scheduler: <.i.>; }
    out { wait_read_b_j: <.i.>; k: <.i.>; scheduler: <.0.>; }
#endtr
#trans clear_b_i
    in  { wait_clear_b_i: <.i.>; b: <.i, 1.>; scheduler: <.i.>; }
    out { wait_read_k_again: <.i.>; b: <.i, 0.>; scheduler: <.0.>; }
#endtr
#trans read_k_again_eq_j
    in  { wait_read_k_again: <.i.>; k: <.3 - i.>; scheduler: <.i.>; }
    out { wait_read_k_again: <.i.>; k: <.3 - i.>; scheduler: <.0.>; }
#endtr
#trans read_k_again_ne_j
    in  { wait_read_k_again: <.i.>; k: <.i.>; scheduler: <.i.>; }
    out { wait_set_b_i_again: <.i.>; k: <.i.>; scheduler: <.0.>; }
#endtr
#trans set_b_i_again
    in  { wait_set_b_i_again: <.i.>; b: <.i, 0.>; scheduler: <.i.>; }
    out { wait_read_b_j: <.i.>; b: <.i, 1.>; scheduler: <.0.>; }
```

```
#endtr
#trans enter
    in  { wait_enter: <.i.>; scheduler: <.i.>; }
    out { critical: <.i.>; scheduler: <.0.>; }
#endtr
#trans exit
    in  { critical: <.i.>; scheduler: <.i.>; }
    out { wait_set_k: <.i.>; scheduler: <.0.>; }
#endtr
#trans set_k
    in  { wait_set_k: <.i.>; k: <.z.>; scheduler: <.i.>; }
    out { wait_clear_b_i_again: <.i.>; k: <.3 - i.>; scheduler: <.0.>; }
#endtr
#trans clear_b_i_again
    in  { wait_clear_b_i_again: <.i.>; b: <.i, 1.>; scheduler: <.i.>; }
    out { noncritical: <.i.>; b: <.i, 0.>; scheduler: <.0.>; }
#endtr
```

The argument of the above #verify is a linear time temporal formula stating that if the scheduler chooses both processes infinitely many times in an infinite execution, then any request for entering a critical section will be fulfilled in the execution, i.e. the process which made the request will enter and exit its critical section. The formula holds which can be seen from the fact that the reachability graph becomes generated without any "illegal infinite path" message.

```
kva@mimas.hut.fi 1: prod dekker.init
kva@mimas.hut.fi 2: dekker

kva+mimas.hut.fi 3:
```

It is not very restrictive to require that the scheduler should choose both processes infinitely many times in an infinite execution. If a process does not want to do anything else than do_noncritical, it does not have to. If we remove the #verify# declaration and then add the two "deadlock detection lines" of Section 4.2 to the end of the net, we see that at each reachable marking of the new net, modified.net, at least one transition is enabled. (Otherwise the reachability graph generator would complain.)

```
kva@mimas.hut.fi 3: prod modified.init
kva@mimas.hut.fi 4: modified

kva+mimas.hut.fi 5:
```

From this it follows that for any scheduling, whenever a process is chosen, the process can actually do something. Knowing that the formula mentioned above holds, we can now conclude that there is no possibility for starvation if neither of the processes is left without processor time forever.

An interested reader may compare these results to the results presented in [4, 27]. The non-liveness results in [4, 27] are based on somewhat weak scheduling assumptions such as the following: whenever a process is chosen, the process can actually do something. (As can be seen from the above, any scheduling satisfies this assumption as far as Dekker's algorithm is concerned.)

5 The query program `probe`

This section is devoted to the inspection of a reachability graph. If we have a reachability graph, we can inspect it with the `probe` program.

Let's first explain some terms. A *node* in the reachability graph corresponds to a reachable marking of the net. An *arrow* corresponds to a fired transition instance. The set of *immediate successor arrows* of a node depends on the graph generation method. By default, the set of immediate successor arrows corresponds to all enabled transition instances.

A node is *terminal* if no transition instance is enabled at the corresponding marking or if a reject state occurs in the marking. (See Section 4.1.) A node has been *processed completely* if and only if the graph generator program has checked the node and created all of its immediate successor arrows. Note that the lack of immediate successor arrows does not necessarily mean that the node would be terminal. At some marking, some graph generation method can intentionally leave all enabled transition instances without being fired.

A graph is *strongly connected* if and only if for each two nodes, there is a path from one to the other. (Since 'for each two nodes' is symmetrical, there is a path in the opposite direction, too.) A *strongly connected component* of a graph is such a strongly connected subgraph of the graph that is not a subgraph of any other strongly connected subgraph of the graph. It follows that each node of a graph is in one and only one strongly connected component of the graph. Strongly connected component B of a graph is an *immediate successor* of strongly connected component A of the graph if and only if there is a real arrow from A to B in the graph and A is different from B. A strongly connected component of a graph is *terminal* if and only if it has no immediate successor. A terminal strongly connected component of a graph is *trivial* if and only it is a terminal node or the whole graph.

5.1 Command line conventions

5.1.1 Command prompt

`probe` is an interactive program though it is possible to run it in a batch style. `probe` displays the number of the current node of the graph in a command prompt. When the program is started it displays the prompt

```
0#
```

which states that the current node of the graph is node number 0, which represents the initial marking of the net.

5.1.2 Line continuation

A command must fit in one *logical line*. A logical line consists of one or more physical lines in such a way that each of the physical lines, except the last one, ends with a backslash. We have thus exactly the same line continuation convention as in the net description language. (See Section 3.1.2.)

5.1.3 Comments in a command line

Comments can be included in a command line simply by quoting them between "`/*`" and "`*/`" as in the C language. Comments can be nested and enclose multiple lines, even without any backslash.

5.1.4 Multiple commands in a command line

Multiple commands can be typed on a single command line by separating them with "`#`". `probe` displays the results of commands in the same order as they were given in the command line.

However this does not affect the use of literal commands. (See section 5.2.) For example:

```
        define looksucc look#succ
```

defines macro `looksucc`, which expands to command sequence **look # succ**.

5.1.5 Interruption

You can interrupt the execution of a command by pressing `CTRL-C`. Unfortunately, in `MS-DOS` the keyboard interruption may be impossible if there is no screen output.

5.1.6 Verbosity

The user can define the verbosity level of commands. Verbosity options are *mute*, *verbose* and *superverbose*. The number of verbosity levels is four because the default verbosity level is distinct. The default is more than *mute* but less than *verbose*.

5.2 Literal commands

`probe` literal commands form a special set of commands in the sense that they do not go through macro expansion, they cannot be used in macro definitions, they must appear at the beginning of the line, and they always continue to the end of the whole command line.

define *name body*

define *name* (*arguments*) *body*

Defines the macro (or constant) named *name*. The body of the macro is given in *body* and possible arguments in *arguments*.

The body of the macro can be anything. The name of the macro cannot be a reserved word in `probe`.

After a macro has been defined, it can be used as a part of any other command in `probe` simply by typing its name to command line.

defs *name**

This displays the definitions of the named macros. If abbreviations are supplied, `defs` displays all possible completions. If no name is supplied, `defs` displays the definitions of all defined macros.

undef *name**

Undefines named macros.

undefall

Undefines all defined macros.

help *subject*

This is an on-line help on the subject. If an abbreviation is supplied, all possible completions are listed. (If the abbreviation is complete but not unique, a full description of the complete subject is displayed.) The command **help** with no argument displays all possible subjects.

load *name*

This loads a command file *name* and executes the commands in it.

If file *name* does not exist and `PRODPATH` environment variable has been defined, `probe` looks for `$PRODPATH/`*name* (`%PRODPATH%\`*name* in `MS-DOS`). *name* can contain directory parts (and a drive identifier in `MS-DOS`).

5.3 Ordinary commands

The following is the description of commands currently known to `probe`.

prev

We backtrack to the previous node. The previous node is the node from which we came to the current node by **next** command.

sgsucc *scomp*

This displays information about the successor components of the strongly connected component *scomp*. (See the description of *scomp* in Section 5.4.2.)

sgpred *scomp*

This displays information about the predecessor components of the strongly connected component *scomp*. (See the description of *scomp* in Section 5.4.2.)

calc *expression*

This evaluates and displays the value of the *expression*. (See the description of *expression* in Section 5.4.1.)

calc *marking*

This evaluates and displays the value of the *marking*. (See the description of *marking* in Section 5.4.1.)

clear *set*

This deletes the reviewable *set*. (See the description of *set* in Section 5.4.2.)

clearall

This deletes all non-special reviewable sets. (There may be some special reviewable sets that have been built automatically in the beginning of the `probe`session.)

termcomps

This displays information about the nontrivial terminal strongly connected components in the reachability graph. (A terminal node or a strongly connected reachability graph is a trivial terminal strongly connected component.)

The *preds* option causes `probe` to display information about the predecessor components for each component.

allcomps

This displays information about all strongly connected components in the reachability graph.

The *preds* option causes `probe` to display information about the predecessor components for each component. Similary the *succs* option causes `probe` to display information about the successor components for each component.

build *set_operation*

This computes the value of *set_operation* and displays the result. If *volatile* option is not supplied, a reviewable set is built from the value. (See the description of *set_operation* in Section 5.4.2.)

goto *expression*

This sets the current node equal to the node pointed by the *expression.*

look *expression*

This displays the marking at the node pointed by the *expression.*

look

This displays the marking at the current node.

pred *expression*

This displays the information of predecessor arrows of the node pointed by *expression.*

pred

This displays the information of predecessor arrows of the current node or the node pointed by expression.

query *formula*

This displays the value of *formula*. If the *volatile* option is not supplied, a reviewable set is built from the value. (See the description of *formula* in Section 5.4.4.) The search can be interrupted by pressing `CTRL-C`. Then only such paths are displayed or put into the reviable set that were found before the search was interrupted. Unfortunately, in `MS-DOS` the keyboard interruption may be impossible if there is no screen output.

query node *formula*

This is similar to the plain **query** command except that all nodes in the graph are visited, and for each node, the value of *formula* is displayed and, if a reviewable set is built, the paths in the value are put into the reviable set.

quit

Quits `probe` program. Returns control back to calling programs.

showends *set_operation*

This displays information about the end nodes of the paths in the value of *set_operation*. (See the description of *set_operation* in Section 5.4.2.)

The **showends** command can be used with options *preds* and *succs*. The *preds* option causes `probe` to display information about the predecessor nodes and arrows for each node. Similary the *succs* option causes `probe` to display information about the successor nodes and arrows for each node. Options can be combined and they can be typed before or after the actual command.

showends all

This is like the above **showends** except that information is displayed about all nodes in the graph.

shrink *set*

This shrinks *set* so that only the path end nodes remain.

statistics

This displays the statistical information on the reachability graph. The statistical information contains the number of nodes, arrows, terminal nodes, handled nodes, strongly connected components and nontrivial terminal strongly connected components.

next *expression*

This moves from the current node to the target node of arrow pointed by *expression*. The target node becomes the new current node.

succ *expression*

This displays the information about the successor arrows of the node pointed by *expression*.

succ

This displays the information about the successor arrows of the current node.

succ *node* arrow *number*

This displays the information about the defined arrow from the defined node. The node is defined by *node* and the arrow by *number*. Both *node* and *number* must be valid expressions.

succ arrow *expression*

This displays the information about the arrow from the current node. The number of arrow is defined by *expression*.

sets

This displays a list of the reviewable sets. For each reviewable set, the number of the set is listed together with the command which produced the set.

5.4 Expressions, formulas and such

5.4.1 Expressions

simple_expression ::=

- ***variable***

  This has the value of *variable* in the transition instance in question. Note that any identifier occuring in the argument of a transition in the argument of **fire** is assumed to be the name of some variable whereas any identifier anywhere else is assumed to be either the name of some transition or place. (See the description of *firing_class* in Section 5.4.3.)

- **preclass**

  The value of this is the precedence class of the transition instance in question. To avoid meaningless expressions, **preclass** can only occur in the argument of **fire**.

- **faithful**

  If the transition instance in question produces the successor marking, faithful is 1. Otherwise faithful is 0 and the instance produces a marking equivalent but not equal to the successor marking. If symmetry method has not been used, each transition instance produces the successor marking. To avoid meaningless expressions, **faithful** can only occur in the argument of **fire**.

- .

  The value of this is the number of the current node in the reachability graph. Note that unlike the evaluation node, the current node is constant during the search.

- **card** (*marking*)

  This is just like in Section 3.9.1, and so are the following three expressions.

- (*expression*)

- **field** [*expression*]

- ***integer***

expression ::= [[Consult from Section 3.9.1.]]

marking ::= [[Consult from Section 3.9.2.]]

5.4.2 Sets

set ::=

- % *simple_expression*

  The value of this is the set of paths in %*n* where *n* is the value of *simple_expression* at the marking corresponding to the evaluation node. The set %*n* is the reviewable set which has the number *n*.

- (*set*)

  This has the value of *set*.

scomp ::=

- %% *simple_expression*

  The value of this is the set of paths in %%n where n is the value of *simple_expression* at the marking corresponding to the evaluation node. Each path in %%n has one node and nothing else. The nodes in %%$(n+1)$th strongly connected component of the reachability graph.

- (*scomp*)

  This has the value of *scomp*.

set_operation ::=

- (*set_operation*)
- *set*
- *scomp*
- *set_operation* | *set_operation*

  The value of this is the union of the values of the set operations.

- *set_operation* + *set_operation*

  The value of this is the union of the values of the set operations.

- *set_operation* & *set_operation*

  The value of this is the intersection of the values of the set operations.

- *set_operation* - *set_operation*

  The value of this is the set of those paths that are in the value of the leftmost *set_operation* but not in the value of the rightmost *set_operation*.

5.4.3 Firing classes

firing_class ::=

- *transition* (*expression*)

  The value of this is the set of those instances of *transition* for which the value of *expression* is not 0. Note that any identifier occuring in the argument of **fire** is assumed to be either the name of some transition or variable whereas any identifier anywhere else is assumed to be the name of some place. (See the description of *simple_expression* in Section 5.4.1.)

- (*firing_class*)

  This has the value of *firing_class*.

- ! *firing_class*

  The value of this is the set of those transition instances that are not in the value of *firing_class*.

- *firing_class* || *firing_class*

  The value of this is the union of the values of the firing classes.

- *firing_class* && *firing_class*

  The value of this is the intersection of the values of the firing classes.

5.4.4 Formulas

atomic_formula ::=

- @ *expression*

  The value of this formula is a set S of paths determined as follows. Let x be the evaluation node and M the marking corresponding to x. If the number of x is equal to the value of *expression* at M, then S is $\{x\}$. Otherwise S is empty.

- *set_operation*

  The value of this formula is a set S of paths determined as follows. Let x be the evaluation node. If x is the end node of some path in the value of *set_operation*, then S is $\{x\}$. Otherwise S is empty.

- *expression*

  The value of this formula is a set S of paths determined as follows. Let x be the evaluation node and M the marking corresponding to x. If the value of *expression* at M is not 0, then S is $\{x\}$. Otherwise S is empty.

formula ::=

- (*formula*)

  This has the value of *formula*.

- *atomic_formula*

- **false**

  The value of this is the empty set.

- **true**

  The value of this is $\{x\}$ where x is the evaluation node.

- **not** *formula*

  The value of this is a set S of paths determined as follows. Let x be the evaluation node. If the value of *formula* in x is empty, then S is $\{x\}$. Otherwise S is empty.

- **step** *formula*

  The value of this is is the minimal set S of paths determined as follows. Let x be the evaluation node. The following holds for all immediate successor arrows y of x. Let x' be the target node of y. Let S' be the value of *formula* in x'. For each path p in S', xyp is in S.

- **fire** (*firing_class*) *formula*

  The value of this is is the minimal set S of paths determined as follows. Let x be the evaluation node. The following holds for those immediate successor arrows y of x that correspond to transition instances in the value of *firing_class*. Let x' be the target node of y. Let S' be the value of *formula* in x'. For each path p in S', xyp is in S.

- **bpath** (*formula,formula*) *formula*

  The value of this is is the minimal set S of paths determined as follows. Let x be the evaluation node. Let ϕ be the leftmost, η the middle and ψ the rightmost formula.

  The following holds for each finite loopless path p starting from x. Let x' be the end node of p. Let q be the prefix of p for which qx' is p. Let S' be the value of ψ in x'. If the value of ϕ is nonempty in every node of q, then for each path p' in S', qp' is in S.

  The following holds for each finite loop-ended path r starting from x and having only one loop. If the value of ϕ is nonempty in every node of r and the value of η is nonempty in the node starting the loop, then r is in S.

- **dpath** (*formula,formula*) *formula*

  As far as the result of the evaluation is concerned, **dpath** has exactly the same effect as **bpath**. The difference between **dpath** and **bpath** is that in the case of **dpath**, the search is performed in a “depth first” order, whereas in the case of **bpath**, the search is performed in a “breadth first” order. Note that in a “breadth first” search, short paths are found before long paths. On the other hand, a “breadth first” search often consumes much more space than a corresponding “depth first” search.

- **bspan** (*formula*) *formula*

  This is similar to **bpath** with a false middle formula except that alternative paths are ignored and, consequently, the search is faster. Note that in a “breadth first” search, the first path found to a node is one of the shortest paths to the node.

- **dspan** (*formula,formula*) *formula*

  This is similar to **dpath** except that alternative paths are ignored and, consequently, the search is faster.

- *formula* **and** *formula*

  The value of this is a set S of paths determined as follows. Let x be the evaluation node. If the value of the leftmost formula in x is not empty, then S is the value of the rightmost formula in x. Otherwise S is empty.

- *formula* **or** *formula*

  The value of this is the union of the values of the formulas in the evaluation node.

5.5 `probe` in on-the-fly verification

If on-the-fly verification has found an error and asks you to look at the set `%1`, the following two commands suffice. We avoided them in Section 4.4 just to avoid unnecessary long output.

```
build volatile verbose %1
query volatile verbose bspan(true) %1
```

The former shows the error which is always a single path. The latter shows one of the shortest paths to the error.

5.6 `probe` in non-on-the-fly verification

In non-on-the-fly verification, it may be useful to ask the following before doing anything else.

```
defs
statistics
sets
```

Let's repeat what was said in sections 5.2, 5.3 and 5.3. The `defs` command without arguments displays the definitions of all defined macros. The `statistics` command displays the statistical information on the reachability graph. The statistical information contains the number of nodes, arrows, terminal nodes, handled nodes, strongly connected components and nontrivial terminal strongly connected components. The `sets` command displays a list of the reviewable sets. For each reviewable set, the number of the set is listed together with the command which produced the set.

The following is an example where a property is verified by searching for counterexamples. If `p` and `q` are places, you may ask

```
query volatile node \
not not dspan(((p + q) : (field[0] > 5)) != empty, true) false
```

Then you see all those nodes that start some loops where the following holds for each node x in the loops: the marking corresponding to x is such that the leftmost field of some tuple in `p` or `q` is greater than 5. The property you are verifying is that there is no such loop.

Let's assume that the property does not hold and one of the nodes mentioned in the output of the above command has the number 99. To see some loops of the above kind, you can use the following commands.

```
goto 99
query dspan(((p + q) : (field[0] > 5)) != empty, true) false
```

The output is non-verbose, but you are informed that a reviewable set %n has been built. Let's assume that n is 2. Then the following gives more information about the loops.

```
build volatile verbose %2
```

As you can see from the above example, non-on-the-fly verification is somewhat complicated. We strongly suggest that you try on-the-fly verification before trying non-on-the-fly verification.

5.7 Connection to CTL

The following definitions implement the operators of CTL [2, 5, 15]. We use the **not** operator to avoid unnecessary searches. The value of a CTL formula is either empty or a set containing the evaluation node and nothing else. It is straightforward to show that a CTL formula holds in the sense defined in [2, 5, 15] if and only if the value of the formula is nonempty.

```
#define Not(f) (not (f))
#define And(f1, f2) (not not ((f1) and (f2)))
#define Or(f1, f2) (not ((not (f1)) and not (f2)))
#define IfThen(f1, f2) (not ((f1) and not (f2)))
#define NextOnSomeBranch(f) (not not step (f))
#define NextOnAllBranches(f) (not step not (f))
#define UntilOnSomeBranch(f1, f2) (not not dspan(f1, false) (f2))
#define UntilOnAllBranches(f1, f2) (not dspan(not (f2), true) \
((not ((step true) and (f1))) and not (f2)))
#define EventuallyOnSomeBranch(f) (not not dspan(true, false) (f))
#define EventuallyOnAllBranches(f) (not dspan(not (f), true) \
((not step true) and not (f)))
#define HenceforthOnSomeBranch(f) (not not dspan(not (f), true) \
((not step true) and not not (f)))
#define HenceforthOnAllBranches(f) (not dspan(true, false) (not (f)))
```

You do not have to rewrite these macros because they are already in the `ctl.prb` file. You can use "`load ctl.prb`" to get these macros defined if they have not been defined yet.

6 The batch program `prod`

The generation of the reachability graph for a Pr/T-net with `PROD` system consists of several phases. The purpose of the `prod`batch program is to avoid repetition of frequently needed long commands. Commands are collected in a special command file. The default command file, used in the examples of this manual, has the name `prodfile`. You can copy this file for yourself and then modify it.

6.1 The syntax of a command file

A basic block in a command file of prod is of the following form.

```
$*.action:
    command
    command
    ...
    ...
;
```

If the *action* part of the command matches the suffix of the target given in the command line (see -t *target* from above) then the command section is performed. The ":" delimits the end of the action suffix. The command section is terminated by ;.

Line continuation

A command must fit in one *logical line*. A logical line consists of one or more physical lines in such a way that each of the physical lines, except the last one, ends with a backslash. We have thus exactly the same line continuation convention as in the net description language. (See Section 3.1.2.)

Conditional commands

You can specify commands to be executed conditionally as follows.

```
$IF (argument operator argument)
    [ commands ]
$ELSE
    [ commands ]
$FI
```

The $IF part of the command is performed if the conditional expression given $IF statement is evaluated true, otherwise the $ELSE part of the command is performed. The above operator is either "!=" or "==". The $IF..$ELSE..$FI constructs can be nested.

Macros

Macros are defined in the following way.

```
$(MACRO_NAME) = macro_value
```

A later occurrence of $(MACRO_NAME) is then replaced by macro_value.

Predefined macros $*, $@

prod provides two predefined macros, $* and $@. $* represents the individual base name of a particular action. (A base name denotes the prefix before the ".".) $@ represents the complete action name. (An action name denotes the "prefix.suffix" combination).

Silent commands and status ignoration

By default, prod shows each command before execution. However, commands preceded by "@" are not shown.

By default, prod exits if any executed command exits with a non-zero status. However, prod ignores a non-zero status of any command preceded by "-".

"@" and "-" can be used together in any order.

Comments

Comments in a command file start with a pound, "#", and last to the end of the logical line. However, two consecutive pounds, "##", do not start a comment but are interpreted as one literal pound.

6.2 An example

The examples in this manual can be reproduced with the aid of the following command file.

```
$(CURDIR)=./#
$(DELETE)=/bin/rm -f#
$(LINK)=cc -g -o#
$(O)=.o#
$(PROD)=/home/kva/public/prod/#
$(PRBIN)=$(PROD)bin/#
$(PRINCL)=$(PROD)include#
$(PRLIB)=$(PRBIN)lib/#
$(COMPILE)=cc -c -DOWNALLOC -DUNIX -g -I$(PRINCL)#
$(X)=#

$*.init:
  @$(PRBIN)prpp -DUNIX -I$(PRINCL) -L -u2000 $*.net
  @$(COMPILE) $*.c
  @$(DELETE) $*$(X)
  @$(LINK) $*$(X) $*$(O) $(PRLIB)*$(O)
;
$*.probe:
  @$(PRBIN)probe -. -lctl.prb -l$*.def $*.gph
;
$*.clean:
  @$(DELETE) $*.adr
  @$(DELETE) $*.dra
  @$(DELETE) $*.aws
  @$(DELETE) $*.wsa
  @$(DELETE) $*.c
  @$(DELETE) $*.dat
  @$(DELETE) $*.def
  @$(DELETE) $*.err
  @$(DELETE) $*$(X)
  @$(DELETE) $*.fld
  @$(DELETE) $*.gph
  @$(DELETE) $*.phg
  @$(DELETE) $*.hsh
  @$(DELETE) $*.shh
  @$(DELETE) $*.pre
  @$(DELETE) $*.stk
  @$(DELETE) $*.tks
  @$(DELETE) $*$(O)
```

6.3 prod in MS-DOS

The above description implicitly assumed that prod was used in UNIX. In MS-DOS, prod does not execute the commands of the command file. Instead, it outputs tmp.bat, an MS-DOS batch file which is to be executed after the execution of prod. tmp.bat is called without parameters. The execution of tmp.bat after prod in MS-DOS has the same effect as the execution of prod in UNIX.

7 The subprograms of PROD

This section explains how the subprograms of PROD can be run.

7.1 prod

As said in Section 6, the purpose of the prod batch program is to avoid repetition of frequently needed long command. The command line of prod is of the following form.

```
  prod [-m macro=value]... [-p file] [[-t] target]...
```

The command line options are described below.

- **-m** *macro=value*

  $(*macro*) is an abbreviation of *value* in the command file of prod.

- **-p** *file*

  file is used instead of `prodfile`, the default command file.

  If *file* does not exist and `PRODPATH` environment variable has been defined, `prod` looks for `$PRODPATH/`*file* (`%PRODPATH%\`*file* in `MS-DOS`). *file* can contain directory parts (and a drive identifier in `MS-DOS`).

- **-t** *target*

  If *target* has the suffix ".suffix" and the command file contains a block having the name "$*.suffix", the commands in the block are executed. "-t" can be omitted if the first character in *target* is not "-".

7.2 prpp

`prpp` reads the net description file and produces a corresponding C file. The command line of `prpp` is of the following form.

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

The command line options are described below.

- **-A**

  Each comment is replaced by a single space character in the resulting C file. By default each comment is replaced by the empty string.

- **-a**

  If this option is used, `prpp` does not allow tuples of different arity in a place, not even in different markings.

- **-C**

  Comments are preserved as such.

- **-L**

  "`#line`" directives are written into appropriate positions in the resulting C file. The purpose is to make error messages point to the net description file when possible.

- **-u** *integer*

  The difference between the highest and the lowest value of a transition input variable cannot be more than *integer*. `prpp` decides the transition input variable ranges from the place tuple limits. This option is needed if the net is to be unfolded later.

- **-D** *macro=value*

  macro is an abbreviation of *value* in the net description file. The "*=value*" can be omitted. A plain "-D *macro*" is equivalent to "-D *macro=1*".

- **-U** *macro*

  If *macro* has been defined earlier on the command line, it becomes undefined. This makes sense, for example, when the definition has been hidden behind a macro in the command file of `prod`.

- **-I** *directory*

  directory becomes the first directory in the list of directories to be checked when an `#include` line has been encountered in the net description file.

- `-t` *file*

  file contains the net description. *file* can be given without the ".net" suffix. "-t" can be omitted if the first character in *file* is not "-".

7.3 The reachability graph generator

If the net description file is `mynet.net`, `mynet` is the executable reachability graph generator program produced by `prpp`, C compiler, and linker. The command line of the graph generator program is of the following form.

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

The command line options are described below. When we describe some reachability graph generation method, we may use the expression "correct and complete in on-the-fly verification". By "correct" we mean that any error reported by the method is an actual error of the type specified in the net description. By "complete" we mean that if there are one or more errors of the type specified in the net description, then exactly one such error is reported. We do not require more reports because in on-the-fly verification, the generation of the reachability graph is stopped whenever an error is found.

- `-.`

  For every tenth node that has been processed completely, a dot is displayed on the screen. However, in the case of the sleep set method (the "-S" option), the number of poppings from the search stack is counted instead. The "-." option is especially useful in `MS-DOS` where the keyboard interruption may be impossible without screen output.

- **-A** As a side effect, an LTS (labelled transition system) file is produced for the ARA tool [21]. The LTS is equal to the reachability graph with the exception that the nodes of the LTS contain no information. The name of an arc in the LTS is derived from the corresponding transition instance. However, if the instance is invisible, the name is simply "i". (See sections 3.3, 4.1 and 4.5 to know what is meant by a visible transition instance.) ARA can visualize an LTS or check whether two LTS's are CFFD-equivalent [21]. If the name of the net is `mynet.net`, the name of the LTS file is `mynet.lts`. Option "-A" is ignored if the "-c" option is given.

- **-B**

  During reachability graph generation, the set of computed states is kept in a BDD (binary decision diagram) [1]. The current implementation merely increases the time and space consumed during generation, but an efficient implementation may be available in the future.

- **-C**

  A *CFFD-preserving version of the stubborn set method* [19, 24] is used, and the reachability graph is generated in a "depth first" order. CFFD-preservation uses information about the visibility of transition instances. See sections 3.3, 4.1 and 4.5 to know what is meant by a visible transition instance. The CFFD-preserving stubborn set method preserves all CFFD properties and is correct and complete in on-the-fly verification. However, the graph generator complains and refuses to use the stubborn set method if the net description contains some `#prec` declaration. The net is unfolded before graph generation. "-C" has also the effect of "-s", so "-C" without "-d" chooses the incremental algorithm, and "-C" with "-d" chooses the deletion algorithm. In the verification of a linear time temporal formula, the stubborn set method is automatically CFFD-preserving, so "-s" and "-C" have then no effective difference.

- **-c**

  The generation of the reachability graph is continued. (See Section 7.3.1.)

- **-D**

  The reachability graph is generated in a "depth first" order. By default, the order is "breadth first".

- **-d**

  The *stubborn set method* is applied using the *deletion algorithm* [17, 18, 23]. The stubborn set method preserves all reachable terminal markings and is correct and complete in on-the-fly verification. However, the graph generator complains and refuses to use the stubborn set method if the net description contains some `#prec` declaration. The net is unfolded before graph generation. "-d" overrides "-s".

- **-e**

  The *symmetry method* (the *equivalent marking method*) [12, 13, 24] is used. The symmetry method preserves all reachable terminal markings in such a way that for each reachable terminal marking M, some symmetry maps M to some marking occurring in the generated reachability graph. Moreover, any marking occurring in the generated reachability graph is actually reachable from the initial marking. However, the graph generator complains and refuses to use the symmetry method if the net description contains some `#tester`, `#verify` or `#prec` declaration, or if the sleep set method ("-S") or the CFFD-preserving stubborn set method ("-C") is to be used. The graph generator uses every *place symmetry*, i.e. every such place mapping that corresponds to some symmetry. Unfortunately, the computation of the place symmetries may take long.

- **-E**

  This option is like “-e” except that only such symmetries are accepted where the initial marking is symmetric. A marking is symmetric if and only if no symmetry maps the marking to any other marking. From the symmetricity of the initial marking it follows that if a marking M is reachable from the initial marking and if some symmetry maps M to a marking M', then M' is also reachable from the initial marking. “-E” overrides “-e”.

- **-f**

  This option is like “-D” except that arrows are linked to each other in a different way, thus making the query program `probe` display the successor arrows of a node in a decreasing order.

- **-i**

  All transition instances are invisible by default. (This option is meaningful only in non-on-the-fly verification. Otherwise it is ignored. See sections 3.3, 4.1 and 4.5 to know what is meant by a visible transition instance.) Otherwise they would be visible by default. Visibility information is essential when option “-A” or “-C” is given.

- **-S**

  The *sleep set method* [7, 8, 23, 25, 26, 28] is used. The sleep set method alone as well as the combination of the sleep set method and the stubborn set method preserve all reachable terminal markings and are correct and complete in on-the-fly verification. However, the graph generator complains and refuses to use the sleep set method if the net description contains some `#verify` or `#prec` declaration, or if there is some livelock monitor state or infinite path monitor state, or if the CFFD-preserving stubborn set method is to be used. The net is unfolded before graph generation. “-S” has also the effect of “-f”.

- **-s**

  The *stubborn set method* is applied using the *incremental algorithm* (“the algorithm using strongly connected components”) [16, 18]. The stubborn set method preserves all reachable terminal markings and is correct and complete in on-the-fly verification. However, the graph generator complains and refuses to use the stubborn set method if the net description contains some `#prec` declaration. The net is unfolded before graph generation.

- **-u**

  The net is unfolded before graph generation. By default the generation proceeds on the Pr/T-net level by deciding the values of the transition variables according to the inputs of the transitions.

- **-V**

  If the symmetry method is used, then each place symmetry, except the identity mapping, is displayed. “-V” overrides “-v”.

- **-v**

  If the symmetry method is used, then each generator of the group of place symmetries, except the identity mapping, is displayed.

- **-x**

  This option has an effect only together with the incremental algorithm of the stubborn set method. The incremental algorithm chooses the disabling low level place of a disabled transition instance randomly. By default the first one is selected.

- **-y**

  This option has an effect only together with the incremental algorithm of the stubborn set method. The incremental algorithm starts from the first enabled transition instance and chooses the first stubborn set found. By default all enabled transition instances are visited and a stubborn set having the least number of enabled transition instances is chosen. (The collection of sets from which the winner is chosen may be a minor subset of the collection of all stubborn sets.)

- **-z**

  This option has an effect only together with the stubborn set method. If the deletion algorithm is used, the deletion algorithm chooses the candidate to be deleted randomly, instead of always trying to delete the first possible candidate. If the incremental algorithm is used, the incremental algorithm works as in case of "-y" except that the algorithm starts from a randomly selected enabled transition instance, instead of the first enabled transition instance. However, "-z" is ignored if the CFFD-preserving incremental algorithm is used. "-z" overrides "-y".

- **-b** *n*

  The size of the hash table used in graph generation is the greatest prime number less than or equal to 2^n. The default value of n is 10. The hash table is segmented in such a way that n can be greater than 16 even in `MS-DOS`.

- **-g** *n*

  When n nodes become completely processed, the generation of the reachability graph is stopped as soon as possible. (In the case of the "-c" option, the nodes that have earlier been processed completely are not counted.) This option has a different meaning if the sleep set method (the "-S" option) is used. Then n refers to the number of poppings from the search stack.

- **-m** *n*

  The reachability graph is kept in memory. Otherwise the graph files would be manipulated directly. If the net decription file is `mynet.net`, then bytes are reserved for `mynet.gph` (n bytes), `mynet.aws` (n bytes) and `mynet.adr` ($1 + (\text{unsigned})((n-1)/k)$ bytes where k is an internal constant). In the case of the sleep set method, bytes are reserved for `mynet.stk`, too (n bytes). If a linear time temporal formula is verified, then bytes are reserved for also `mynet.phg` (n bytes), `mynet.wsa` (n bytes) and `mynet.dra` (as many bytes as for `mynet.adr`). If the number of bytes reserved for a file turns out to be insufficient, direct manipulation of the file is started.

- **-r** *n*

  n is used as a seed number for the random number generator. The default value of *n* is 1. However, if "-c" has been given, the default value is determined in such a way that the preceding interruption in graph generation does not affect the pseudo-random sequence of the random number generator.

7.3.1 Interrupting the generation

The generation of a reachability graph can be interrupted. If the generation takes place in the foreground or if you are in `MS-DOS`, all you need to do is to press the control key and the C key simultaneously. (If the generation takes place in the background, you can execute "`kill -2 pid`" where `pid` is the number of the generator process.) You can later continue the generation by calling the generator program with the "-c" option.

In `UNIX`, you have an alternative way to interrupt the generation. If the generation takes place in the foreground, you can press the control key and the Z key simultaneously. (If the generation takes place in the background, you can try "`kill -18 pid`". The success of this command is not guaranteed, but the authors of this manual do not know anything better.) If you then want to continue the generation, you can use the `fg` (or `bg`) command of the C shell to continue the run of the generation job. (Alternatively, you can try "`kill -19 pid`", but as above, the success is not guaranteed.)

There may be an observable delay between your interrupt action and the actual interrupt of the generation. There is no guaranteed upper bound for the delay.

7.4 strong

`strong` computes the strongly connected components of the reachability graph. The command line of `strong` is of the following form.

```
strong [-G] [-t] graph
```

The command line options are described below.

- **-G**

  The reachability graph is kept in memory throughout the computation of the strongly connected components. Otherwise the graph files would be manipulated directly.

- **-t** *graph*

  If the net description file is `mynet.net`, *graph* should be either `mynet.gph` or `mynet`. "-t" can be omitted if the first character in *graph* is not "-".

7.5 probe

`probe` is a program for inspecting the reachability graph. The command line of `probe` is of the following form.

```
probe [-deGiMq.] [-h file] [-l file]... [-w n] [-t] graph
```

The command line options are described below.

- **-d**

  Error messages are directed to the standard error output flow, instead of the standard ordinary output flow.

- **-e**

  Everything read from the standard input flow is "redisplayed" on the screen. This makes sense when the standard input flow is actually a file.

  For example command

  ```
  probe -e mynet.gph < commandfile > logfile
  ```

  produces a complete log corresponding to the commands in `commandfile`.

- **-G**

  The reachability graph is kept in memory throughout the `probe` session. Otherwise the graph files would be manipulated directly. "-G" overrides "-M".

- **-i**

  Everything read from a command file is displayed on the screen.

- **-M**

  Strongly connected components are kept in memory. By default a component is read from a file when needed, with the exception that getting the number of the strongly connected component containing a given node does not require such reading.

- **-q**

  No prompt is displayed.

- **-.**

  In the context of "query mute", a dot is displayed for every tenth accepted path. This option is especially useful in `MS-DOS` where the keyboard interruption may be impossible without screen output.

- **-h** *file*

  file is the help file, instead of `probhelp`, the default help file.

  If *file* does not exist and `PRODPATH` environment variable has been defined, `probe` looks for `$PRODPATH/`*file* (`%PRODPATH%\`*file* in `MS-DOS`). *file* can contain directory parts (and a drive identifier in `MS-DOS`).

- -l *file*

  The commands in *file* are executed at the beginning of the run of `probe`.

  If *file* does not exist and `PRODPATH` environment variable has been defined, `probe` looks for `$PRODPATH/`*file* (`%PRODPATH%\`*file* in `MS-DOS`). *name* can contain directory parts (and a drive identifier in `MS-DOS`).

- -w *n*

  When markings, paths on the default verbosity level, or strongly connected components are displayed, the maximum length of a line is *n*. The default value of *n* is 79.

- -t *graph*

  If the net description file is `mynet.net`, *graph* should be either `mynet.gph` or `mynet`. "-t" can be omitted if the first character in *graph* is not "-".

7.6 araprod

`araprod` computes the parallel composition [19] of given labelled transition systems (LTS's) and hides given actions. The composition is computed by constructing a Petri net, generating a reachability graph, and interpreting the graph as an LTS. On-the-fly verification of the kind presented in Section 4.1 is also possible, and then the net in question is the net corresponding to the parallel composition of the LTS's. The construction of the net is such that every state of the LTS presenting the tester becomes a state of the tester in the sense defined in Section 4.1. The command line of `araprod` is of the following form.

```
araprod [-.BCcDdEefSsxyz] [-b n] [-g n] [-h hidefile]...
        [-M monitorfile] [-m n] -o outputfile [-r n]
        [-T testerfile] [[-t] ltsfile]...
```

The command line options are described below.

- -h *hidefile*

  The actions mentioned in the file *hidefile* are hidden. (However, actions are not hidden if on-the-fly verification has been chosen by options "-M" and "-T".) The format of the file is the following: any string of the form `[A-Za-z_$][A-Za-z0-9_$]*` is considered an action name except when inside a comment. A comment begins with two consecutive minus characters, `--`, and continues to the end of the line. Otherwise line breaks are treated like space characters and tabulators.

- -o *outputfile*

  The computed LTS is put into the file *outputfile*. The format of ARA [21] is used. The suffix ".lts" is added to the name *outputfile* iff *outputfile* does not already have that suffix. An ordinary PROD net desription is put into a file having the same name as *outputfile* but with the suffix ".net". On the other hand, `araprod` also produces the reachability graph, so there is no need to compile the net description. Instead, one can directly inspect the graph by `probe` or compute the strongly connected components of the graph by `strong`.

- **-M** *monitorfile*

  If options "-M" and "-T" are issued, `araprod` performs on-the-fly verification. The file *monitorfile* lists the monitor states of the tester. There are four sets of monitor states: reject states, deadlock monitor states, livelock monitor states, and infinite path monitor states. The meaning of these sets was described in Section 4.1. The format of the file *monitorfile* is

  ```
  [ REJECTS
    [ integer [ .. integer] ]
    ...
    END_REJECTS ]
  [ DEADLOCKS
    [ integer [ .. integer] ]
    ...
    END_DEADLOCKS ]
  [ LIVELOCKS
    [ integer [ .. integer] ]
    ...
    END_LIVELOCKS ]
  [ INFINITES
    [ integer [ .. integer] ]
    ...
    END_INFINITES ]
  ```

  Line breaks are included in the format. A block enclosed by "[" and "]" is optional. "..." means that there can be arbitrarily many lines of the kind of the previous line. A comment begins with two consecutive minus characters, `--`, and continues to the end of the line. ".." is a range operator like in `prpp`. "*a* .. *b*" means that all values greater than or equal to *a* and less than or equal to *b* are included.

- **-T** *testerfile*

  If options "-M" and "-T" are issued, `araprod` performs on-the-fly verification. The file *testerfile* contains the LTS of the tester. The format of ARA [21] is required. No suffix in the name *testerfile* can be omitted since an LTS file can have any name. If it is not wanted to compose the tester LTS with itself in the parallel composition, the name of the tester LTS should not be repeated in the command line of `araprod`. (Every occurrence of the name means a separate component.)

- **-t** *ltsfile*

  The file *ltsfile* contains one LTS. The format of ARA [21] is required. No suffix in the name *ltsfile* can be omitted since an LTS file can have any name. "-t" can be omitted if the first character in the name *ltsfile* is not "-". If it is not wanted to compose an LTS with itself in the parallel composition, the name of the LTS should not be repeated in the command line of `araprod`. (Every occurrence of the name means a separate component.)

The other options are exactly like in Section 7.3. Graph generation can be interrupted and continued as described in Section 7.3, except that `araprod` is in the place of `mynet`.

Acknowledgements

PROD has been developed by Johannes Helander, Tomi Janhunen, Ismo Kangas, Kari Nurmela, Kenneth Oksanen, Olavi Pesonen, Marko Rauhamaa, James Reilly, Heikki Suonsivu, Kimmo Valkealahti, Kimmo Varpaaniemi, and Pauli Väisänen. PROD has been documented by the developers as well as Peter Grönberg, Jaakko Halme, Kari Hiekkanen, Tino Pyssysalo, and Mikko Tiusanen.

The development of the tool has been supported by Helsinki University of Technology, the Technology Development Centre of Finland, the Academy of Finland, the Emil Aaltonen Foundation, Nokia Telecommunications Oy, Telecom Finland Oy, and Commit Oy.

How to get `PROD`

`PROD` is free of charge and can be obtained using `ftp`. The `ftp` site is `saturn.hut.fi` (Internet address 192.26.133.104). The files needed are in `pub/prod/`. The `README` file gives detailed instructions on how to install `PROD` on a `UNIX` or on an `MS-DOS` environment and where to find example nets, especially those considered in this manual.

References

[1] Bryant, R.E.: *Graph-Based Algorithms for Boolean Function Manipulation.* IEEE Transactions on Computers C-35 (1986) 8, pp. 677–691.

[2] Clarke, E.M., Emerson, E.A., and Sistla, A.P.: *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications.* ACM Transactions on Programming Languages and Systems 8 (1986) 2, pp. 244–263.

[3] Dijkstra, E.W.: *Hierarchical Ordering of Sequential Processes.* In Hoare, C.A.R, and Perrott, R.H. (Eds.), Operating Systems Techniques, Proceedings of a Seminar held at Queen's University, Belfast 1971, APIC Studies in Data Processing No. 9, Academic Press, London 1972, pp. 72–93.

[4] Eloranta, J.: *Minimal Transition Systems with Respect to Divergence Preserving Behavioural Experiences.* Doctoral thesis, University of Helsinki, Department of Computer Science, Report A-1994-1, Helsinki 1994, 162 p.

[5] Emerson, E.A.: *Temporal and Modal Logic.* In van Leeuwen, J. (Ed.), Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics, Elsevier, Amsterdam 1990, pp. 995–1072.

[6] Genrich, H.J.: *Predicate/Transition Nets.* In Brauer, W., Reisig, W., and Rozenberg, G. (Eds.), Petri Nets: Central Models and Their Properties — Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, September 1986. Lecture Notes in Computer Science 254, Springer-Verlag, Berlin 1987, pp. 207–247.

[7] Godefroid, P.: *Partial-Order Methods for the Verification of Concurrent Systems — An Approach to the State-Explosion Problem.* Doctoral thesis, University of Liège, November 1994, 134 p.

[8] Godefroid, P. and Wolper, P.: *Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties.* Formal Methods in System Design 2 (1993) 2, pp. 149–164.

[9] Gribomont, P., and Wolper, P.: *Temporal Logic.* In Thayse, A. (Ed.), From Modal Logic to Deductive Databases – Introducing a Logic Based Approach to Artificial Intelligence, John Wiley & Sons 1989, pp. 165–233.

[10] Jensen, K., and Rozenberg, G. (Eds.): *High-level Petri Nets. Theory and Application.* Springer-Verlag, Berlin 1991, 724 p.

[11] Rauhamaa, M.: *A Comparative Study of Methods for Efficient Reachability Analysis.* Helsinki University of Technology, Digital Systems Laboratory Report A 14, Espoo 1990, 61 p.

[12] Schmidt, K.: *Symmetries of Petri Nets.* Petri Net Newsletter 43 (1993), pp. 9–25.

[13] Starke, P.H.: *Reachability Analysis of Petri Nets Using Symmetries.* Systems Analysis – Modelling – Simulation 8 (1991) 4/5, pp. 293–303.

[14] Thielke, Th.: *Modelchecking als Komponente der Petrinetzbasierten Entwicklungs- und Programmierumgebung PEP.* Desel. J., Oberweis, A., and Reisig, W. (Eds.), Proceedings of the "Algorithmen und Werkzeuge für Petrinetze" workshop, Berlin, October 1994. Universität Karlsruhe (TH), Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Bericht 309, Karlsruhe 1994, pp. 74–79.

[15] Tuominen, H.: *Logic in Petri Net Analysis.* Helsinki University of Technology, Digital Systems Laboratory Report A 5, Espoo 1988, 53 p.

[16] Valmari, A.: *Error Detection by Reduced Reachability Graph Generation.* Proceedings of the 9th European Workshop on Application and Theory of Petri Nets, Venice, June 1988, pp. 95–112.

[17] Valmari, A.: *Heuristics for Lazy State Space Generation Speeds up Analysis of Concurrent Systems.* Mäkelä, M., Linnainmaa, S., and Ukkonen, E. (Eds.), Proceedings of the Finnish Artificial Intelligence Symposium (Suomen tekoälytutkimuksen päivät), Vol. 2, Helsinki 1988, pp. 640–650.

[18] Valmari, A.: *State Space Generation: Efficiency and Practicality.* Doctoral thesis, Tampere University of Technology Publications 55, Tampere 1988, 170 p.

[19] Valmari, A.: *Alleviating State Explosion during Verification of Behavioural Equivalence.* University of Helsinki, Department of Computer Science, Report A-1992-4, Helsinki 1992, 57 p.

[20] Valmari, A.: *On-the-Fly Verification with Stubborn Sets.* Courcoubetis, C. (Ed.), Proceedings of the 5th International Conference on Computer-Aided Verification, Elounda, Greece, June/July 1993. Lecture Notes in Computer Science 697, Springer-Verlag, Berlin 1993, pp. 397–408.

[21] Valmari, A., Kemppainen, J., Clegg, M., and Levanto, M.: *Putting Advanced Reachability Analysis Techniques Together: the "ARA" Tool.* Proceedings of Formal Methods Europe '93. Lecture Notes in Computer Science 670, Springer-Verlag, Berlin 1993, pp. 597–616.

[22] Valmari, A., and Tiusanen, M.: *A Graph Model for Efficient Reachability Analysis of Description Languages.* Proceedings of the 8th European Workshop on Application and Theory of Petri Nets, Zaragoza, June 1987, pp. 349–366.

[23] Varpaaniemi, K.: *Efficient Detection of Deadlocks in Petri Nets.* Helsinki University of Technology, Digital Systems Laboratory Report A 26, Espoo, October 1993, 56 p.

[24] Varpaaniemi, K.: *On Computing Symmetries and Stubborn Sets.* Helsinki University of Technology, Digital Systems Laboratory Report B 12, Espoo, April 1994, 16 p.

[25] Varpaaniemi, K.: *On Combining the Stubborn Set Method with the Sleep Set Method.* Valette, R. (Ed.), Proceedings of the 15th International Conference on Application and Theory of Petri Nets, Zaragoza, June 1994. Lecture Notes in Computer Science 815, Springer-Verlag, Berlin 1994, pp. 548–567.

[26] Varpaaniemi, K.: *The Sleep Set Method Revisited.* Czaja, L., Burkhard, H.-D., and Starke, P.H. (Eds.), Proceedings of the 3rd International Workshop on Concurrency, Specification, and Programming, Berlin, October 1994. Humboldt Universität zu Berlin, Institut für Informatik, Informatik-Bericht 36, Berlin 1994, 10 p.

[27] Walker, D.J.: *Automated Analysis of Mutual Exclusion Algorithms using CCS.* Formal Aspects of Computing 1 (1989) 3, pp. 273–292.

[28] Wolper, P., and Godefroid, P.: *Partial-Order Methods for Temporal Verification.* Best, E. (Ed.), Proceedings of the 4th International Conference on Concurrency Theory, Hildesheim, August 1993. Lecture Notes in Computer Science 715, Springer-Verlag, Berlin 1993, pp. 233–246.