HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **B:** Technical Reports
No. **11**; June 1993

ISSN 0783–540X
ISBN 951–22–1588–8

PROD—A Pr/T-NET REACHABILITY ANALYSIS TOOL

PETER GRÖNBERG, MIKKO TIUSANEN, AND KIMMO VARPAANIEMI

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
SF–02150 ESPOO, FINLAND

HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **B:** Technical Reports ISSN 0783–540X
No. **11**; June 1993 ISBN 951–22–1588–8

PROD—A Pr/T-Net Reachability Analysis Tool

PETER GRÖNBERG, MIKKO TIUSANEN, AND KIMMO VARPAANIEMI

Abstract: PROD is a reachability analysis tool for Pr/T-nets. The stubborn set method for reduced state space generation has been implemented in it. PROD also has a rich query language for inspecting the generated state space. PROD is available and free of charge. A UNIX or an MS-DOS version of PROD can be obtained using ftp. A reference manual of PROD is available on request. The corresponding LaTeX files are included in the package that contains the software. This report briefly introduces PROD and demonstrates its usage. Special attention is paid to the query language. The principles of the stubborn set method are also presented.

Keywords: Petri nets, Pr/T-nets, reachability analysis, stubborn set method.

Printing: TKK Monistamo; Otaniemi 1993

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

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

Contents

1 Introduction

Petri nets [6, 1] are very suitable in the modelling of concurrent and distributed systems. The problem of Petri nets until the 80's was their very large size when systems of the real world were modelled. The problem was solved by developing high level Petri nets which made it possible to write compact models in a natural way. One class of high level Petri nets, Pr/T-nets [4], has been long studied in Digital Systems Laboratory.

Reachability analysis is a powerful method to analyze Petri nets. Using it we can easily check whether certain properties hold or not. Unfortunately, reachability analysis suffers from *state space explosion.* For example, the number of states of a finite state system may grow exponentially with respect to a parameter. If we are primarily interested in such properties as the existence of terminal states, we don't necessarily have to generate the complete state space. The *stubborn set method* [9, 13] is one of the most promising methods to find terminal states by inspecting a relatively small number of states only.

It is difficult to interpret a large state space without some automatic tool. For proper understanding of the behaviour of the net, it should be possible to ask the tool to show state and event sequences satisfying given property. A plain 'yes' or 'no' answer to an existence question is not necessarily very informative.

`PROD` is a reachability analysis tool for Pr/T-nets. The stubborn set method has been implemented in it. `PROD` also has a rich query language for inspecting the generated state space.

`PROD` has been developed by a group of researchers of Digital Systems Laboratory and students of Helsinki University of Technology. The shortcomings in `PRENA` [5] led to the decision to develop `PROD`. `PRENA` is an older reachability analysis tool for Pr/T-nets that has been developed in Digital Systems Laboratory, but `PROD` is strictly different from `PRENA`.

`PROD` is available and free of charge. A `UNIX` or an `MS-DOS` version of `PROD` can be obtained using `ftp`. A reference manual of `PROD` is available on request. The corresponding LaTeX files are included in the package that contains the software.

Section 2 is an introduction to `PROD` and gives instructions on the usage of `PROD`. It also tells how to get `PROD`. The modelling of systems by means of Petri nets is described together with an example in Section 3. Section 4 gives an example of using `PROD` to inspect a state space and then presents the stubborn set method. Appendix A presents the syntax and semantics of the formulae in the query language.

2 Getting started

This section introduces PROD and gives instructions on how to get started in using PROD. Subsection 2.1 briefly describes what PROD is and how it is used. The availability and installation of PROD are considered in Subsection 2.2.

2.1 Structure and usage

PROD consists of a net description language preprocessor prpp, a 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. The term 'strongly connected component' will be explained in Section 4.)

PROD's net description language is the C preprocessor language extended with net description directives. Net description is compiled into an executable reachability graph generator program. PROD's graph query language resembles CTL (Computation Tree Logic) [2].

Figure 1 shows the overall structure of PROD. The arrows around prod present the role of prod in executing the other programs. The other arrows present the order of execution. The white-headed arrows present alternative orders. The computation of strongly connected components is not necessary and can be done after inspecting the graph. There is also a possibility to interrupt graph generation, inspect the graph, and continue the generation.

prod takes a file called 'prodfile' as its input. 'prodfile' tells which programs should be executed, and how and when. A user can use the default 'prodfile' or write another. Of course, it is also possible to call the programs prpp, probe, etc. directly, instead of using prod.

PROD is very independent of machine and operating system. Up till now it has been installed on a number of UNIX machines and an MS-DOS PC.

The environment variable 'PATH' should contain the complete name of the directory where the programs prpp, strong, probe, and prod are. An environment variable called 'PRODPATH' should be equal to the complete name of the directory where the default 'prodfile' is.

The most straightforward way to use PROD is to write a net description into a file, say, 'mynet.net' and run 'prod mynet.net'. If we assume the current default 'prodfile', the different phases of computation are printed on the screen until the graph query program gives a prompt and waits for input. The user then types investigation commands such as 'statistics' and finally closes the session by typing 'quit'.

2.2 Getting PROD

PROD is free of charge and can be obtained using ftp. Both a UNIX and an MS-DOS version are available. The ftp site is 'saturn.hut.fi' (Internet address 192.26.133.104).

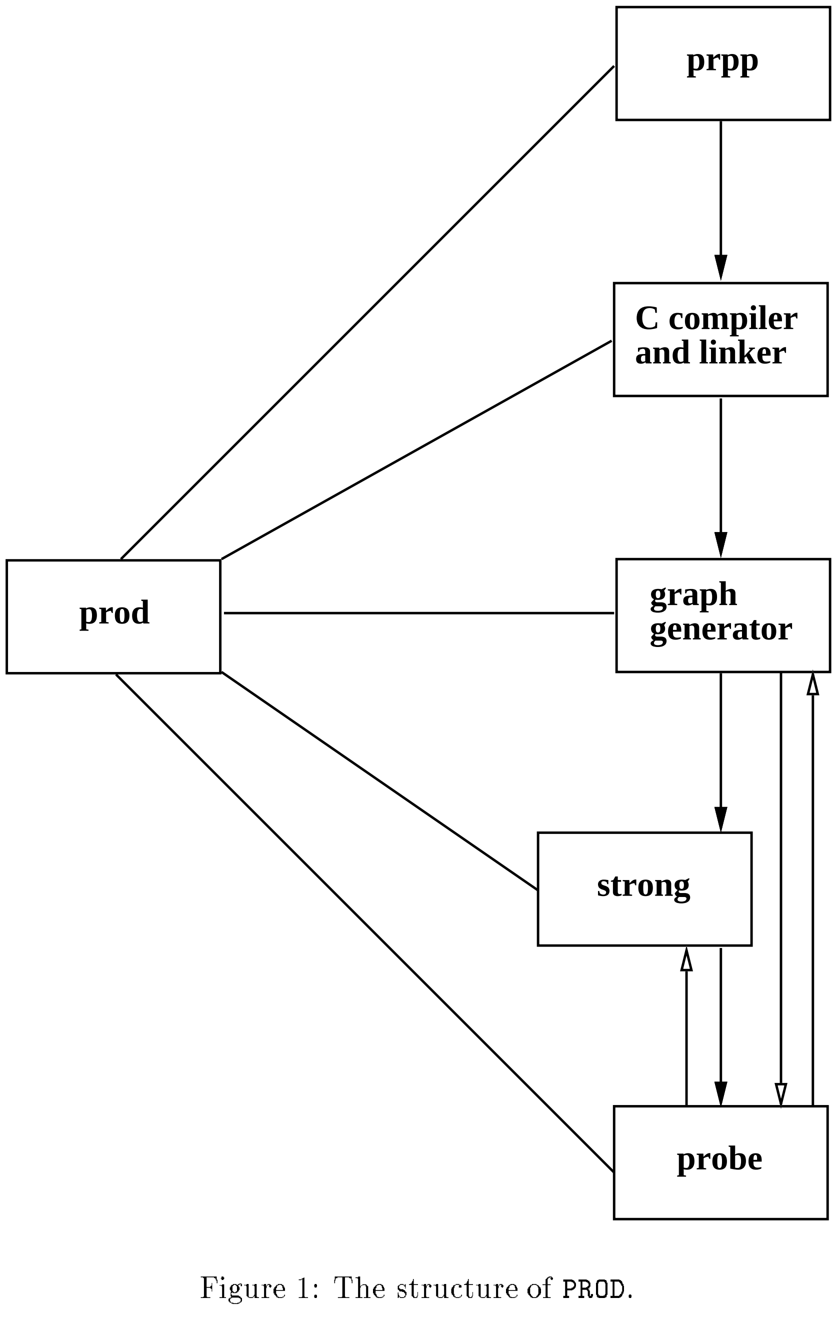

Figure 1: The structure of PROD.

'anonymous' can be used as a login name, and E-mail address for password. The `UNIX` version is in 'pub/prod/prod' and the `MS-DOS` version in 'pub/prod/pcprod'. Both contain an archive file and two instruction files, 'README' and 'INSTALL'. Binary mode must be used in file transfer. 'README' gives general information about `PROD`. 'INSTALL' tells how to open the archive file and install `PROD`.

The LaTeX files of the reference manual of `PROD` are in the archive file, together with the software.

3 Modelling of systems

This section considers the modelling of systems by using Petri nets. Subsection 3.1 is a very informal introduction to Petri nets. The classical problem of dining philosophers is modelled in Subsection 3.2. The problem has been widely used in computer science to popularize concurrency control problems.

3.1 Petri nets

Petri nets were invented by Carl Adam Petri and published in his PhD Thesis in 1962 [6]. In a later paper [7] he discusses a possible motivation, which we shall base the following introduction on.

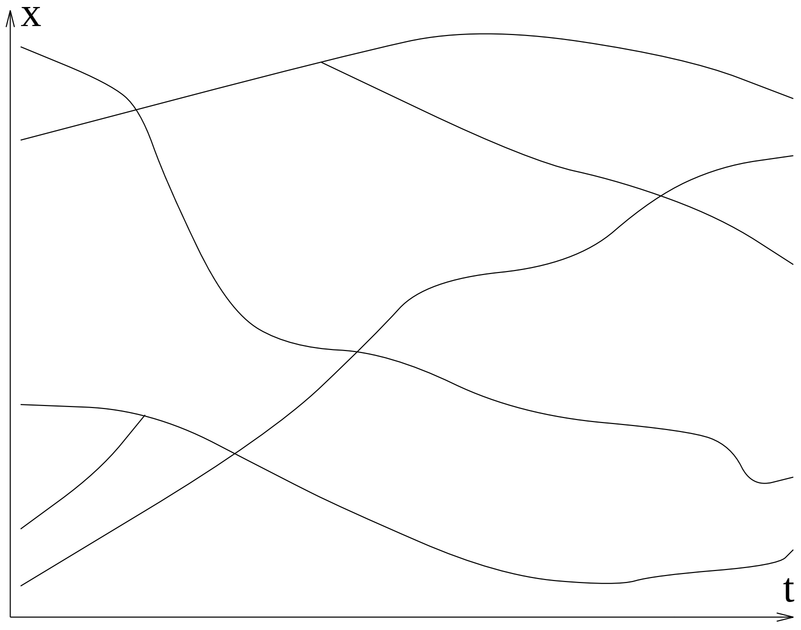

Figure 2: A collection of world lines in 1-dimensional space.

Consider a collection of world lines in a space-time continuum, say, a history of some of particles moving around and colliding with each other (for an example in 1-dimensional space plus time, see Figure 2). The obvious interactions among the particles are represented by the world lines of two or more of them crossing at a point.

Now, group the points on these world lines so that a crossing point forms a group, equivalence class, on its own, and any line segment between some two crossings (or either end of a world line) without any other crossing point intervening forms another kind of group. We shall call any group of points of the first kind an *event* and any of the second kind a *condition*. Let us represent the events by boxes or bars, and conditions by circles. Obviously, an event is next to the conditions that have a line segment end or start at that particular crossing point. The passage of time gives a direction to this relation of being next to another group, turning it to a follower or *flow* relation. In fact, we have constructed a bipartite, directed, acyclic graph

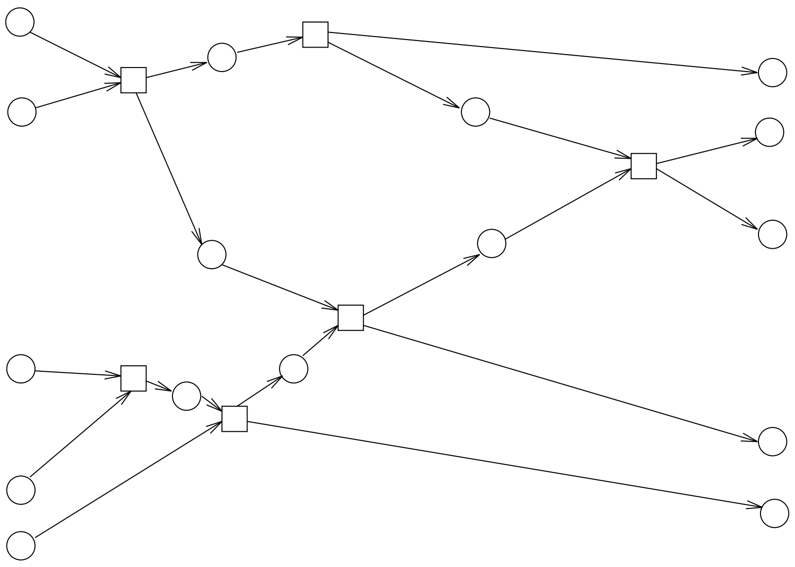

Figure 3: A history of conditions and events.

of groups and the flow relation with an obvious connection to the world lines, see Figure 3. Moreover, there is at most one event immediately before a condition, and at most one event immediately after a condition.

Since this is a history of the particles, all the choices as to which interactions have taken place have already been fixed. The history is, however, *concurrent*: there are events that are not necessarily one before the other, but are simply unrelated to a large enough degree to occur concurrently.

Now, assume the history were the result of a particle system evolution, a *physical* process, perhaps repetitive, perhaps containing choices, perhaps containing concurrent events. Then it would be appealing to describe the system using the same kinds of primitives, say *transitions* and *places*, similarly interconnected. However, the flow relation need no longer be acyclic (to allow for repetitions), nor need the places have at most one predecessor (to allow for forgetting exactly which interaction produced the particle) or at most one follower (to allow for choices among possible interactions). Note that it is unusual to have the behavior (here: history) and the system be described using the same notation. Yet, the notations are not identical, since the history has more properties than the description of the process. Also, the system has a *state*, the distribution of the particles, that is usually called a *marking*. The system would then produce the history much as the tire of a car leaves tracks in dirt, by "unfolding", with the current distribution of particles corresponding to the point of contact between the tire and dirt. There are some problems, however, that must be addressed to make the model well-defined.

Consider the particle that caused a particular line segment to be recorded in the set of world lines. This has typically many attributes, such as color, mass, or momentum. For the model of the system to be well-defined, there must be enough similarity among the attributes of the particles that reside in a place in order for the transitions to be meaningful. Also, the transitions will uphold some "laws of nature", such as conservation of mass-energy and momentum in the case of particles. These can be expressed as a relation over the attributes of the particles involved, sometimes called a *firing condition.*

The evolution of the system of particles can be depicted by the particles containing attributes (call them *tokens*) being transformed and transported from place to place through the occurrence, or *firing*, of transitions. For a transition to be able to occur (to be *enabled*), some particular kinds of tokens must be present, and the firing condition (laws of nature) must permit the transition to occur. We call a state *terminal* if and only if no transition is enabled at it. When a transition occurs, it will remove all the tokens it required from the corresponding places, and deposit some, perhaps entirely different tokens to some places, as dictated by the flow relation. The model obtained in this way is the essential core of any of the so called *high-level* Petri net models (for many of these, see [1]). The class of interest here is that of Predicate/Transition nets (or Pr/T-nets). These use tuples to represent tokens with attributes, and first-order predicate calculus formulae to express the firing conditions. A transition can have many *instances*, in the sense that it can be enabled for many combinations of values for the *variables* appearing on the tuples attached to the arcs and in the firing conditions. For a formal definition of these we refer the reader to [4].

Most of the time, the particles involved are specialized enough to make any two of them distinguishable: the tokens have an *identity.* Sometimes it is useful, however, to ignore the identity of tokens. The solution employed in Petri nets is then to simply *count* the tokens similar enough in a place. The logical conclusion of this abstraction is to distinguish no two tokens, so that all tokens are, in a sense "black dots". Obviously, if tokens are not distinguished, there is little point in stating any firing conditions for transitions, since these operate on the attributes of tokens, of which there are none. The result is a model generally called Petri Nets, sometimes also Place/Transition systems (or P/T-nets). These can be seen as a special case of Pr/T-nets after all the tokens have been *projected* to become the empty tuple $\langle\rangle$ as described in [4], and all the firing conditions removed.

It is amusing to note that the so called high-level Petri nets are on a distinctly lower level of abstraction in the above sense than plain Petri nets. The reason for the name is that it is possible to give a translation (sometimes called *unfolding*, though this does not involve the relation between system and its history) of a high-level net to a plain Petri net that preserves the *reachability relation* between corresponding markings. A marking is *immediately reachable* from another if it can be obtained from the latter by firing a transition (transition instance in a Pr/T-net); it is *reachable* from the latter if there is a path from the latter to it. The two nets will then be 1-to-1 simulations of each other. Unfolding employs potentially many transitions (places) to represent any transition (place) of the high-level net. It is in this sense that the plain Petri nets are on a lower level.

Since their conception, Petri nets and its variants have been used to model and analyze the behavior of a large variety of systems, such as computer, social, or economic ones. We hold that the appeal of the model stems from the basis in physics, since particles and their interactions are intuitively well understood ideas and easily identified almost anywhere. A token can model a person in a particular state of mind, a copy of document in an office, or—as we shall see—the location of the control and the values of local variables of a process.

3.2 Dining philosophers

We shall now 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.]]”

Figure 4 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 $n-1$.

Figure 5 presents the same net written in `PROD`'s description language. 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’.

Macros are useful because they spare us the effort of repeating a complicated or less intuitive 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.

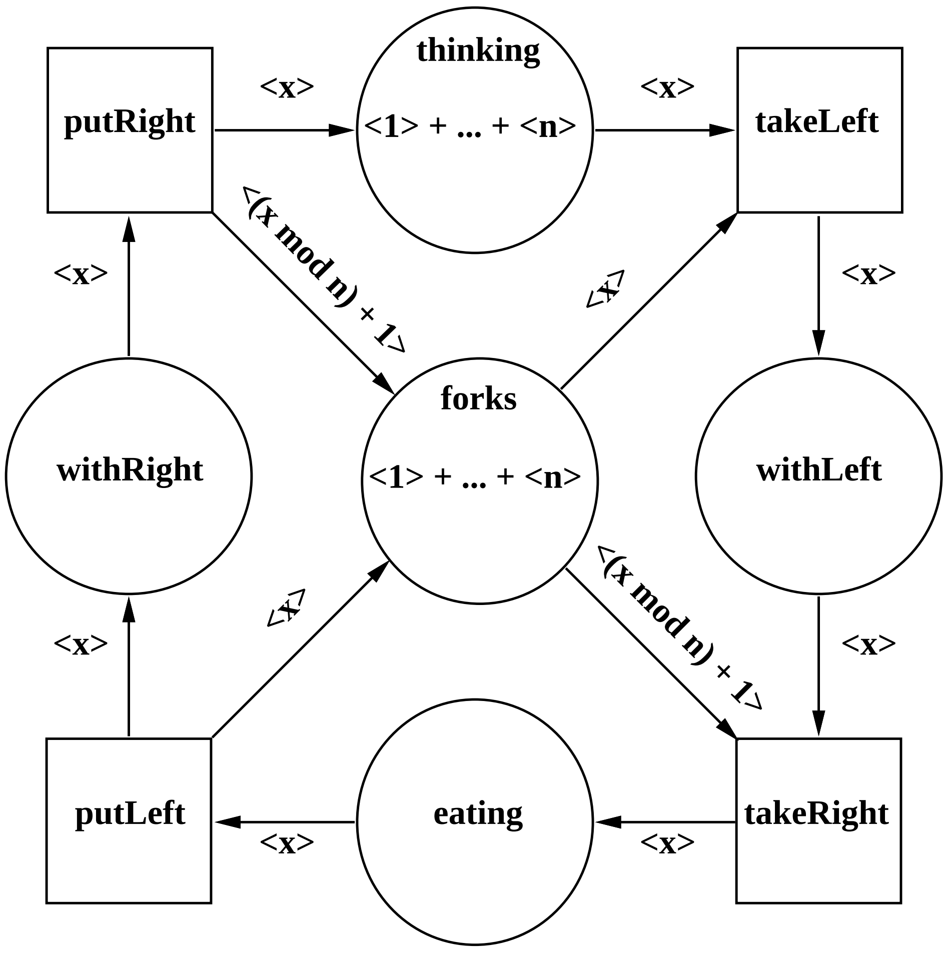

Figure 4: Dining philosophers.

```
#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
```

Figure 5: The philosopher system presented in PROD's description language.

'hi(`<.`n`.>`)' denies the corresponding place all tuples with 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.

There should be nothing difficult in the transition description part of this net. Variable ph is local to each transition and has the same predefined integer type as the values in the tuples in the places.

4 Reachability analysis

This section begins with an investigation of a complete state space and ends with a description of a state space generation method. The order is intentional. Subsection 4.1 assumes ‘ordinary’ state space generation, while Subsection 4.2 describes the stubborn set method that tries to generate a small but suitable incomplete state space.

4.1 Investigating system behaviour

In this subsection, the complete state space of the net in Figure 5 is investigated. The number of philosophers, n, is five. This example can be reproduced by writing the net description into ‘ph.net’, putting the investigation commands into ‘ph.btc’, running ‘`prod` ph.net’, typing ‘quit’, and running
‘`probe` ph.gph -. -lph.def -w70 -e < ph.btc > ph.log’.
The reproduced example is then in ‘ph.log’.

Some terms need to be explained. A *reachability graph* presents the state space of the system. A *node* in the reachability graph corresponds to a reachable marking of the net. It is possible to include *fact transitions* in a net description. Fact transitions present undesirable situations, so an enabled instance of a fact transition is undesirable. The net in Figure 5 has no fact transition. A *real arrow* corresponds to a fired instance of a non-fact transition. A *fact arrow* corresponds to an instance of a fact transition. A real arrow has a target node, but a fact arrow has no target node. In other words, only non-fact transitions are fired as far as `PROD` is concerned. 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. In the case of the stubborn set method which will be presented in Subsection 4.2, the set of immediate successor arrows corresponds to all enabled transition instances in a computed stubborn set as well as all enabled instances of the fact transitions. The fact transitions are imagined to be non-existent when a stubborn set is computed. The example in this subsection has been produced without using the stubborn set method.

A node is *terminal* if and only if it has no real immediate successor arrow. A node has been *completely processed* if and only if the graph generator program has checked the node and created all of its immediate successor arrows.

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.

The command line prompt of probe is a pound preceded by the number of the current node. The initial node has the number 0. '\' at the end of a line means that a command continues at the next line.

We first ask what macro definitions we already have. We shall use n in the sequel.

```
0#defs
#define LEFT(x) (x)
#define RIGHT(x) (1 + ((x) % n))
#define UNIX 1
#define n 5
```

'statistics' tells us that there is one terminal node and two strongly connected components. Since a terminal node as such is a strongly connected component, we conclude that all the other nodes are in the other component.

```
0#statistics
Number of nodes: 242
Number of (real) arrows: 805
Number of terminal nodes: 1
Number of fact arrow source nodes: 0
Number of fact arrows: 0
Number of nodes that have been completely processed: 242
Number of strongly connected components: 2
Number of nontrivial terminal strongly connected components: 0
```

All sets in probe consist of paths. Each strongly connected component is represented by one and only one '$$-set'. Such set consists of only the nodes in the component. When it is known that each path in a set is just a node, 'showends' is a convenient way to show the set. We observe that the system can get into a terminal state in which each philosopher holds his left-hand fork.

```
0#sets
Strongly connected components: $$0..$$1
Special sets:
  $0: ** terminal nodes **
  $1: ** fact arrow source nodes **
0#showends $0
Node 91, belongs to strongly connected component $$0
  withLeft: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
-----------------------------------------------
1 path end nodes
-----------------------------------------------
```

We evaluate some expressions with respect to the current node. The two new macros will be used many times in the sequel.

```
0#look
Node 0, belongs to strongly connected component $$1
  thinking: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
------------------------------------------------
0#calc thinking
<.1.> + <.2.> + <.3.> + <.4.> + <.5.>
0#calc card(thinking)
5
0#define submarking(marking, formula) ((marking):(formula))
0#define firstFieldInTuple (field[0])
0#calc submarking(thinking, firstFieldInTuple == 1)
<.1.>
0#calc submarking(thinking, firstFieldInTuple < 1)
empty
0#calc submarking(thinking, firstFieldInTuple > 1)
<.2.> + <.3.> + <.4.> + <.5.>
0#calc card(submarking(thinking, firstFieldInTuple > 1))
4
```

The current node can be changed by firing a transition instance, by backtracking, or by just jumping to another node. Each transition instance is in some *precedence class*. As far as our example is concerned, it is sufficient to know that transition instances have no priority to each other, and all transition instances are in the default precedence class 0.

```
0#succ verbose
Arrow 0: transition takeLeft, precedence class 0
  ph = 1
Node 1, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.4.> + <.5.>
  forks: <.2.> + <.3.> + <.4.> + <.5.>
  withLeft: <.1.>
------------------------------------------------
Arrow 1: transition takeLeft, precedence class 0
  ph = 2
Node 2, belongs to strongly connected component $$1
  thinking: <.1.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.3.> + <.4.> + <.5.>
  withLeft: <.2.>
------------------------------------------------
Arrow 2: transition takeLeft, precedence class 0
  ph = 3
Node 3, belongs to strongly connected component $$1
  thinking: <.1.> + <.2.> + <.4.> + <.5.>
  forks: <.1.> + <.2.> + <.4.> + <.5.>
  withLeft: <.3.>
------------------------------------------------
Arrow 3: transition takeLeft, precedence class 0
  ph = 4
Node 4, belongs to strongly connected component $$1
  thinking: <.1.> + <.2.> + <.3.> + <.5.>
  forks: <.1.> + <.2.> + <.3.> + <.5.>
  withLeft: <.4.>
-------------------------------------------------
Arrow 4: transition takeLeft, precedence class 0
  ph = 5
Node 5, belongs to strongly connected component $$1
  thinking: <.1.> + <.2.> + <.3.> + <.4.>
  forks: <.1.> + <.2.> + <.3.> + <.4.>
  withLeft: <.5.>
-------------------------------------------------
Node 0 has 5 successor arrows
-------------------------------------------------
0#next 0
1#look
Node 1, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.4.> + <.5.>
  forks: <.2.> + <.3.> + <.4.> + <.5.>
  withLeft: <.1.>
-------------------------------------------------
1#succ
Arrow 0: transition takeLeft, precedence class 0
  ph = 2
to node 6
-------------------------------------------------
Arrow 1: transition takeLeft, precedence class 0
  ph = 3
to node 7
-------------------------------------------------
Arrow 2: transition takeLeft, precedence class 0
  ph = 4
to node 8
-------------------------------------------------
Arrow 3: transition takeLeft, precedence class 0
  ph = 5
to node 9
-------------------------------------------------
Arrow 4: transition takeRight, precedence class 0
  ph = 1
to node 10
-------------------------------------------------
Node 1 has 5 successor arrows
-------------------------------------------------
1#next 2
8#look
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
-------------------------------------------------
8#prev
```

```
1#prev
0#goto 91
91#look
Node 91, belongs to strongly connected component $$0
  withLeft: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
------------------------------------------------
91#goto 8
```

We start demonstrating ‘query’ and ‘query node’ by considering atomic argument formulae. ‘volatile’ means ‘do not build a set’. ‘verbose’ and ‘mute’ are verbosity levels. ‘verbose’ is above and ‘mute’ below the default verbosity level.

```
8#define qvo query volatile
8#define qvov qvo verbose
8#define qvom qvo mute
8#qvov eating == empty
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
------------------------------------------------
1 paths
------------------------------------------------
8#qvov thinking == empty
0 paths
------------------------------------------------
8#qvov node thinking == empty
PATH
Node 91, belongs to strongly connected component $$0
  withLeft: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
------------------------------------------------
1 paths
------------------------------------------------
8#qvov node (card(thinking) == n - 2) && (card(eating) == 2)
PATH
Node 64, belongs to strongly connected component $$1
  thinking: <.2.> + <.4.> + <.5.>
  forks: <.5.>
  eating: <.1.> + <.3.>
------------------------------------------------
PATH
Node 68, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.3.>
  eating: <.1.> + <.4.>
------------------------------------------------
PATH
Node 79, belongs to strongly connected component $$1
  thinking: <.1.> + <.3.> + <.5.>
  forks: <.1.>
  eating: <.2.> + <.4.>
------------------------------------------------
PATH
Node 82, belongs to strongly connected component $$1
  thinking: <.1.> + <.3.> + <.4.>
  forks: <.4.>
  eating: <.2.> + <.5.>
------------------------------------------------
PATH
Node 87, belongs to strongly connected component $$1
  thinking: <.1.> + <.2.> + <.4.>
  forks: <.2.>
  eating: <.3.> + <.5.>
------------------------------------------------
5 paths
------------------------------------------------
```

'step' and 'fire' are 'next state' operators. 'fire' has an argument which describes the transition instance.

```
8#qvov step (submarking(withLeft, firstFieldInTuple <= 2) != empty)
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 2
Node 22, belongs to strongly connected component $$1
  thinking: <.3.> + <.5.>
  forks: <.3.> + <.5.>
  withLeft: <.1.> + <.2.> + <.4.>
------------------------------------------------
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
Arrow 1: transition takeLeft, precedence class 0
  ph = 3
Node 25, belongs to strongly connected component $$1
  thinking: <.2.> + <.5.>
  forks: <.2.> + <.5.>
  withLeft: <.1.> + <.3.> + <.4.>
------------------------------------------------
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
Arrow 2: transition takeLeft, precedence class 0
  ph = 5
Node 29, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.>
  forks: <.2.> + <.3.>
  withLeft: <.1.> + <.4.> + <.5.>
-------------------------------------------------
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
Arrow 4: transition takeRight, precedence class 0
  ph = 4
Node 31, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.>
  withLeft: <.1.>
  eating: <.4.>
-------------------------------------------------
4 paths
-------------------------------------------------
8#define trueExpression 1
8#qvov fire(takeRight(trueExpression)) \
       (submarking(withLeft, firstFieldInTuple <= 2) != empty)
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
Arrow 4: transition takeRight, precedence class 0
  ph = 4
Node 31, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.>
  withLeft: <.1.>
  eating: <.4.>
-------------------------------------------------
1 paths
-------------------------------------------------
```

The negation operator ‘not’ is introduced. It is a very useful operator, as we shall see in the sequel.

```
8#qvov fire(takeLeft(ph < 2)) true
0 paths
-------------------------------------------------
8#qvov not (fire(takeLeft(ph < 2)) true)
PATH
```

```
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
------------------------------------------------
1 paths
------------------------------------------------
8#qvov fire(takeLeft(ph > 2)) true
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
Arrow 1: transition takeLeft, precedence class 0
  ph = 3
Node 25, belongs to strongly connected component $$1
  thinking: <.2.> + <.5.>
  forks: <.2.> + <.5.>
  withLeft: <.1.> + <.3.> + <.4.>
------------------------------------------------
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
Arrow 2: transition takeLeft, precedence class 0
  ph = 5
Node 29, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.>
  forks: <.2.> + <.3.>
  withLeft: <.1.> + <.4.> + <.5.>
------------------------------------------------
2 paths
------------------------------------------------
8#qvov not (fire(takeLeft(ph > 2)) true)
0 paths
------------------------------------------------
8#qvov not (not (fire(takeLeft(ph > 2)) true))
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
------------------------------------------------
1 paths
------------------------------------------------
```

The complement of a set of transition instances is introduced.

```
8#qvov fire(!takeLeft(ph > 2)) true
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 2
Node 22, belongs to strongly connected component $$1
  thinking: <.3.> + <.5.>
  forks: <.3.> + <.5.>
  withLeft: <.1.> + <.2.> + <.4.>
------------------------------------------------
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
Arrow 3: transition takeRight, precedence class 0
  ph = 1
Node 30, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.3.> + <.5.>
  withLeft: <.4.>
  eating: <.1.>
------------------------------------------------
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
Arrow 4: transition takeRight, precedence class 0
  ph = 4
Node 31, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.>
  withLeft: <.1.>
  eating: <.4.>
------------------------------------------------
3 paths
------------------------------------------------
```

'and' operator tests the first argument and, conditionally, evaluates the second argument.

```
8#qvov (fire(!takeLeft(ph > 2)) true) \
       and (fire(takeLeft(ph > 2)) true)
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
Arrow 1: transition takeLeft, precedence class 0
  ph = 3
Node 25, belongs to strongly connected component $$1
  thinking: <.2.> + <.5.>
  forks: <.2.> + <.5.>
  withLeft: <.1.> + <.3.> + <.4.>
------------------------------------------------
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
Arrow 2: transition takeLeft, precedence class 0
  ph = 5
Node 29, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.>
  forks: <.2.> + <.3.>
  withLeft: <.1.> + <.4.> + <.5.>
------------------------------------------------
2 paths
------------------------------------------------
8#qvov (fire(takeLeft(ph < 2)) true) \
       and (fire(takeLeft(ph > 2)) true)
0 paths
------------------------------------------------
```

If, as in the net in our example, transition instances are distinguishable with respect to their effect on the places, 'fire' can be simulated by combining 'step', a postcondition, and a precondition.

```
8#qvov node (submarking(thinking, firstFieldInTuple <= 2) != empty) \
            and (step (thinking == empty))
PATH
Node 59, belongs to strongly connected component $$1
  thinking: <.2.>
  forks: <.2.>
  withLeft: <.1.> + <.3.> + <.4.> + <.5.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 2
Node 91, belongs to strongly connected component $$0
  withLeft: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
------------------------------------------------
PATH
Node 72, belongs to strongly connected component $$1
  thinking: <.1.>
  forks: <.1.>
  withLeft: <.2.> + <.3.> + <.4.> + <.5.>
Arrow 0: transition takeLeft, precedence class 0
```

```
  ph = 1
Node 91, belongs to strongly connected component $$0
  withLeft: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
------------------------------------------------
2 paths
------------------------------------------------
8#qvov node fire(takeLeft(ph <= 2)) (thinking == empty)
PATH
Node 59, belongs to strongly connected component $$1
  thinking: <.2.>
  forks: <.2.>
  withLeft: <.1.> + <.3.> + <.4.> + <.5.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 2
Node 91, belongs to strongly connected component $$0
  withLeft: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
------------------------------------------------
PATH
Node 72, belongs to strongly connected component $$1
  thinking: <.1.>
  forks: <.1.>
  withLeft: <.2.> + <.3.> + <.4.> + <.5.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 1
Node 91, belongs to strongly connected component $$0
  withLeft: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
------------------------------------------------
2 paths
------------------------------------------------
```

Negation saves computation time and space when only the set of start nodes or the existence of paths is of interest.

```
8#qvov node not (not (fire(takeLeft(ph <= 2)) (thinking == empty)))
PATH
Node 59, belongs to strongly connected component $$1
  thinking: <.2.>
  forks: <.2.>
  withLeft: <.1.> + <.3.> + <.4.> + <.5.>
------------------------------------------------
PATH
Node 72, belongs to strongly connected component $$1
  thinking: <.1.>
  forks: <.1.>
  withLeft: <.2.> + <.3.> + <.4.> + <.5.>
------------------------------------------------
2 paths
------------------------------------------------
```

Terminal nodes can be found in the following way, instead of looking at `$0`. However, looking at $0 is preferable because 'query node' makes `probe` visit all nodes.

```
8#qvov node not (step true)
PATH
Node 91, belongs to strongly connected component $$0
  withLeft: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
-------------------------------------------------
1 paths
-------------------------------------------------
```

Result storing is demonstrated in the context of a twofold 'fire' formula.

```
8#query mute fire(takeLeft(ph > 2)) \
             (fire(takeRight(trueExpression)) true)
3 paths
Built set $2
8#build volatile verbose $2
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
Arrow 1: transition takeLeft, precedence class 0
  ph = 3
Node 25, belongs to strongly connected component $$1
  thinking: <.2.> + <.5.>
  forks: <.2.> + <.5.>
  withLeft: <.1.> + <.3.> + <.4.>
Arrow 2: transition takeRight, precedence class 0
  ph = 1
Node 60, belongs to strongly connected component $$1
  thinking: <.2.> + <.5.>
  forks: <.5.>
  withLeft: <.3.> + <.4.>
  eating: <.1.>
-------------------------------------------------
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
Arrow 1: transition takeLeft, precedence class 0
  ph = 3
Node 25, belongs to strongly connected component $$1
  thinking: <.2.> + <.5.>
  forks: <.2.> + <.5.>
  withLeft: <.1.> + <.3.> + <.4.>
Arrow 3: transition takeRight, precedence class 0
  ph = 4
Node 61, belongs to strongly connected component $$1
```

```
  thinking: <.2.> + <.5.>
  forks: <.2.>
  withLeft: <.1.> + <.3.>
  eating: <.4.>
-------------------------------------------------
PATH
Node 8, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.5.>
  forks: <.2.> + <.3.> + <.5.>
  withLeft: <.1.> + <.4.>
Arrow 2: transition takeLeft, precedence class 0
  ph = 5
Node 29, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.>
  forks: <.2.> + <.3.>
  withLeft: <.1.> + <.4.> + <.5.>
Arrow 2: transition takeRight, precedence class 0
  ph = 1
Node 67, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.>
  forks: <.3.>
  withLeft: <.4.> + <.5.>
  eating: <.1.>
-------------------------------------------------
3 paths
-------------------------------------------------
```

A set formula is an atomic formula which holds only at the end nodes of the paths in the set. 'showends' shows the end nodes of the paths in the set.

```
8#qvov $2
0 paths
-------------------------------------------------
8#goto 61
61#qvov $2
PATH
Node 61, belongs to strongly connected component $$1
  thinking: <.2.> + <.5.>
  forks: <.2.>
  withLeft: <.1.> + <.3.>
  eating: <.4.>
-------------------------------------------------
1 paths
-------------------------------------------------
61#showends $2
Node 60, belongs to strongly connected component $$1
  thinking: <.2.> + <.5.>
  forks: <.5.>
  withLeft: <.3.> + <.4.>
  eating: <.1.>
```

```
------------------------------------------------
Node 61, belongs to strongly connected component $$1
  thinking: <.2.> + <.5.>
  forks: <.2.>
  withLeft: <.1.> + <.3.>
  eating: <.4.>
------------------------------------------------
Node 67, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.>
  forks: <.3.>
  withLeft: <.4.> + <.5.>
  eating: <.1.>
------------------------------------------------
3 path end nodes
------------------------------------------------
```

We now ask for a path of the minimum length from the initial node to the terminal node. If there were more than one terminal node, the query would return for each terminal node one of the shortest paths to that node.

```
61#goto 0
0#qvov 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
-------------------------------------------------
```

There are very many paths from the initial node to the terminal node, even if we restrict ourselves to those paths where no philosopher eats. Such paths cannot contain any loop.

```
0#qvom dpath(eating == empty, false) $0
............
120 paths
```

'b' in 'bspan' comes from 'breadth-first search'. 'dspan' is some kind of a 'depth-first' analogy to 'bspan'. In the following, 'bspan' returns a short path to a node. 'dspan' also returns exactly one path, without any loop, to the same node, but the path is quite long. However, it should be emphasized that breadth-first search is sometimes much more space consuming than depth-first search.

```
0#qvo bspan(true) \
      (submarking(thinking, firstFieldInTuple == 1) != empty) \
      && (card(withLeft) == n - 1)
   0  [1>    2  [1>   11  [1>   34  [1>   72
1 paths
-------------------------------------------------
0#qvo dspan(true, false) \
      (submarking(thinking, firstFieldInTuple == 1) != empty) \
      && (card(withLeft) == n - 1)
   0  [0>    1  [0>    6  [0>   21  [0>   51  [1>   92  [0>  132  [0>
 167  [0>  192  [0>  212  [0>  227  [0>  232  [0>  237  [0>  223  [1>
 164  [0>  178  [1>  201  [0>  219  [1>  150  [0>  176  [0>  200  [1>
 128  [0>  144  [1>  177  [2>   64  [0>  105  [0>  145  [0>  179  [0>
 202  [0>  220  [0>  165  [0>  186  [0>  208  [0>  224  [1>  162  [0>
 188  [0>  209  [0>  225  [0>  166  [0>  190  [0>  205  [0>  222  [0>
 231  [0>  236  [1>  210  [1>  126  [0>  159  [0>  189  [2>   79  [0>
  96  [0>  136  [0>  171  [0>  196  [0>  216  [0>  161  [0>  172  [0>
 197  [1>  112  [0>  143  [2>   27  [0>   60  [0>  101  [0>  141  [0>
 175  [0>  199  [0>  218  [0>  230  [0>  235  [0>  240  [1>  221  [2>
 151  [0>  181  [1>  203  [1>  130  [0>  155  [1>   36  [0>   53  [0>
  93  [0>  133  [0>  168  [0>  193  [0>  213  [0>  228  [0>  233  [0>
 238  [2>  217  [0>  229  [0>  234  [2>  204  [2>  131  [0>  160  [0>
 183  [1>   73  [1>  115  [1>  153  [1>  184  [1>  206  [2>  124  [0>
 137  [1>   24  [0>   55  [0>   95  [0>  135  [0>  170  [0>  195  [0>
 215  [1>  158  [1>   48  [0>   77  [0>  114  [0>  152  [1>   34  [1>
  72
1 paths
-------------------------------------------------
```

Unlike ‘bspan’, ‘dspan’ detects loops. ‘dspan(formula, true) false’ returns paths that satisfy the formula and end in a single loop. ‘dpath(formula, true) false’ returns all such paths. ‘dspan’ ignores alternative routes but is guaranteed to find all those loops that ‘dpath’ finds. Below, we search for loops where philosopher number 1 is thinking or holding his left-hand fork, and philosophers with number greater than 2 are thinking. ‘query’ lists paths in the order they are found. In a set, paths are ordered with respect to node and arrow numbers so that the number of the end node is the most significant, the number of the preceding arrow the next significant, etc.

```
0#define testFormula \
         ((card(submarking(thinking, firstFieldInTuple > 2)) \
           == n - 2) \
          && (submarking(eating, firstFieldInTuple == 1) \
              == empty))
0#query dspan(testFormula, true) false
   0  [0>    1  [0>    6  [3>   24  [2>   58  [3>    1,        0  [1>
   2  [4>   14  [3>   42  [3>    0
2 paths
Built set $3
------------------------------------------------
0#query dpath(testFormula, true) false
   0  [0>    1  [0>    6  [3>   24  [2>   58  [3>    1,        0  [1>
   2  [0>    6  [3>   24  [2>   58  [3>    1  [0>    6,        0  [1>
   2  [4>   14  [0>   24  [2>   58  [3>    1  [0>    6  [3>   24,
   0  [1>    2  [4>   14  [3>   42  [0>   58  [3>    1  [0>    6  [3>
  24  [2>   58,        0  [1>    2  [4>   14  [3>   42  [3>    0
5 paths
Built set $4
------------------------------------------------
0#build volatile verbose $3
PATH
Node 0, belongs to strongly connected component $$1
  thinking: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
Arrow 1: transition takeLeft, precedence class 0
  ph = 2
Node 2, belongs to strongly connected component $$1
  thinking: <.1.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.3.> + <.4.> + <.5.>
  withLeft: <.2.>
Arrow 4: transition takeRight, precedence class 0
  ph = 2
Node 14, belongs to strongly connected component $$1
  thinking: <.1.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.4.> + <.5.>
  eating: <.2.>
Arrow 3: transition putLeft, precedence class 0
  ph = 2
Node 42, belongs to strongly connected component $$1
```

```
  thinking: <.1.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.2.> + <.4.> + <.5.>
  withRight: <.2.>
Arrow 3: transition putRight, precedence class 0
  ph = 2
Node 0, belongs to strongly connected component $$1
  thinking: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
-------------------------------------------------
PATH
Node 0, belongs to strongly connected component $$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 3: transition takeRight, precedence class 0
  ph = 2
Node 24, belongs to strongly connected component $$1
  thinking: <.3.> + <.4.> + <.5.>
  forks: <.4.> + <.5.>
  withLeft: <.1.>
  eating: <.2.>
Arrow 2: transition putLeft, precedence class 0
  ph = 2
Node 58, belongs to strongly connected component $$1
  thinking: <.3.> + <.4.> + <.5.>
  forks: <.2.> + <.4.> + <.5.>
  withLeft: <.1.>
  withRight: <.2.>
Arrow 3: transition putRight, precedence class 0
  ph = 2
Node 1, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.4.> + <.5.>
  forks: <.2.> + <.3.> + <.4.> + <.5.>
  withLeft: <.1.>
-------------------------------------------------
2 paths
-------------------------------------------------
0#build volatile verbose $4 - $3
```

```
PATH
Node 0, belongs to strongly connected component $$1
  thinking: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
Arrow 1: transition takeLeft, precedence class 0
  ph = 2
Node 2, belongs to strongly connected component $$1
  thinking: <.1.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.3.> + <.4.> + <.5.>
  withLeft: <.2.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 1
Node 6, belongs to strongly connected component $$1
  thinking: <.3.> + <.4.> + <.5.>
  forks: <.3.> + <.4.> + <.5.>
  withLeft: <.1.> + <.2.>
Arrow 3: transition takeRight, precedence class 0
  ph = 2
Node 24, belongs to strongly connected component $$1
  thinking: <.3.> + <.4.> + <.5.>
  forks: <.4.> + <.5.>
  withLeft: <.1.>
  eating: <.2.>
Arrow 2: transition putLeft, precedence class 0
  ph = 2
Node 58, belongs to strongly connected component $$1
  thinking: <.3.> + <.4.> + <.5.>
  forks: <.2.> + <.4.> + <.5.>
  withLeft: <.1.>
  withRight: <.2.>
Arrow 3: transition putRight, precedence class 0
  ph = 2
Node 1, belongs to strongly connected component $$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 $$1
  thinking: <.3.> + <.4.> + <.5.>
  forks: <.3.> + <.4.> + <.5.>
  withLeft: <.1.> + <.2.>
------------------------------------------------
PATH
Node 0, belongs to strongly connected component $$1
  thinking: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
Arrow 1: transition takeLeft, precedence class 0
  ph = 2
```

```
Node 2, belongs to strongly connected component $$1
  thinking: <.1.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.3.> + <.4.> + <.5.>
  withLeft: <.2.>
Arrow 4: transition takeRight, precedence class 0
  ph = 2
Node 14, belongs to strongly connected component $$1
  thinking: <.1.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.4.> + <.5.>
  eating: <.2.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 1
Node 24, belongs to strongly connected component $$1
  thinking: <.3.> + <.4.> + <.5.>
  forks: <.4.> + <.5.>
  withLeft: <.1.>
  eating: <.2.>
Arrow 2: transition putLeft, precedence class 0
  ph = 2
Node 58, belongs to strongly connected component $$1
  thinking: <.3.> + <.4.> + <.5.>
  forks: <.2.> + <.4.> + <.5.>
  withLeft: <.1.>
  withRight: <.2.>
Arrow 3: transition putRight, precedence class 0
  ph = 2
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 3: transition takeRight, precedence class 0
  ph = 2
Node 24, belongs to strongly connected component $$1
  thinking: <.3.> + <.4.> + <.5.>
  forks: <.4.> + <.5.>
  withLeft: <.1.>
  eating: <.2.>
------------------------------------------------
PATH
Node 0, belongs to strongly connected component $$1
  thinking: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
Arrow 1: transition takeLeft, precedence class 0
```

```
  ph = 2
Node 2, belongs to strongly connected component $$1
  thinking: <.1.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.3.> + <.4.> + <.5.>
  withLeft: <.2.>
Arrow 4: transition takeRight, precedence class 0
  ph = 2
Node 14, belongs to strongly connected component $$1
  thinking: <.1.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.4.> + <.5.>
  eating: <.2.>
Arrow 3: transition putLeft, precedence class 0
  ph = 2
Node 42, belongs to strongly connected component $$1
  thinking: <.1.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.2.> + <.4.> + <.5.>
  withRight: <.2.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 1
Node 58, belongs to strongly connected component $$1
  thinking: <.3.> + <.4.> + <.5.>
  forks: <.2.> + <.4.> + <.5.>
  withLeft: <.1.>
  withRight: <.2.>
Arrow 3: transition putRight, precedence class 0
  ph = 2
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 3: transition takeRight, precedence class 0
  ph = 2
Node 24, belongs to strongly connected component $$1
  thinking: <.3.> + <.4.> + <.5.>
  forks: <.4.> + <.5.>
  withLeft: <.1.>
  eating: <.2.>
Arrow 2: transition putLeft, precedence class 0
  ph = 2
Node 58, belongs to strongly connected component $$1
  thinking: <.3.> + <.4.> + <.5.>
  forks: <.2.> + <.4.> + <.5.>
  withLeft: <.1.>
  withRight: <.2.>
------------------------------------------------
3 paths
------------------------------------------------
```

The next query, given at the initial node, is a general way to ask whether the graph has any loop. There is, of course, a loop if the number of strongly connected components is less than the number of nodes in the graph. A problem arises when there are as many strongly connected components as nodes.

```
0#qvov not (not (dspan(true, true) false))
PATH
Node 0, belongs to strongly connected component $$1
  thinking: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.2.> + <.3.> + <.4.> + <.5.>
------------------------------------------------
1 paths
------------------------------------------------
```

Suppose that we want to see a short loop-ended path in which philosopher number 1 is holding his right-hand fork. We proceed silently, by giving more and more specific formulae and using negation to save computation time and space, until the number of start nodes is sufficiently small. We then jump to the start node with the lowest number and ask for actual paths.

```
0#qvom node not (not (dspan(submarking(withRight, \
                                     firstFieldInTuple == 1) \
                            != empty, \
                            true) false))
.....
53 paths
0#qvom node not (not (dspan((submarking(withRight, \
                                      firstFieldInTuple == 1) \
                             != empty) \
                            && (card(thinking) >= n - 2), \
                            true) false))
.
11 paths
0#qvo node not (not ( \
        dspan((submarking(withRight, firstFieldInTuple == 1) \
               != empty) \
              && (card(submarking(thinking, \
                                  firstFieldInTuple != 3)) \
                  == n - 2), \
              true) false))
  33,       65,      108,      148
4 paths
------------------------------------------------
0#goto 33
33#qvov dspan((submarking(withRight, firstFieldInTuple == 1) \
              != empty) \
             && (card(submarking(thinking, \
                                 firstFieldInTuple != 3)) \
                 == n - 2), \
             true) false
PATH
Node 33, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.3.> + <.4.> + <.5.>
  withRight: <.1.>
Arrow 0: transition takeLeft, precedence class 0
  ph = 3
Node 65, belongs to strongly connected component $$1
  thinking: <.2.> + <.4.> + <.5.>
  forks: <.1.> + <.4.> + <.5.>
  withLeft: <.3.>
  withRight: <.1.>
Arrow 2: transition takeRight, precedence class 0
  ph = 3
Node 108, belongs to strongly connected component $$1
  thinking: <.2.> + <.4.> + <.5.>
  forks: <.1.> + <.5.>
  eating: <.3.>
  withRight: <.1.>
Arrow 1: transition putLeft, precedence class 0
  ph = 3
Node 148, belongs to strongly connected component $$1
  thinking: <.2.> + <.4.> + <.5.>
  forks: <.1.> + <.3.> + <.5.>
  withRight: <.1.> + <.3.>
Arrow 2: transition putRight, precedence class 0
  ph = 3
Node 33, belongs to strongly connected component $$1
  thinking: <.2.> + <.3.> + <.4.> + <.5.>
  forks: <.1.> + <.3.> + <.4.> + <.5.>
  withRight: <.1.>
------------------------------------------------
1 paths
------------------------------------------------
```

We quit `probe`.

```
33#quit
```

4.2 Stubborn set method

The stubborn set method is a method due to Valmari [9, 10, 11, 12] that attempts to exploit the *commutativity* of transitions in order to reduce the number of states of the state space actually inspected during the generation of the state space, while still finding all terminal states and detecting the existence of infinite firing sequences of a P/T-net. Transitions may be *ignored*: there may be transitions that occur in the complete state space but not in the reduced state space. If the stubborn sets used in the state space generation are guaranteed to be sufficiently 'strong', the ignoring phenomenon can be eliminated by using an algorithm presented by Valmari [13]. More recently, Valmari has also presented an algorithm that verifies any pregiven stuttering-invariant linear time temporal logic formula [14], and then an on-the-fly verification algorithm [15], both using stubborn sets. PROD uses the very weak definition of stubbornness given in [9]. Such definition is better than any stronger definition if the goal is to find the terminal states by inspecting as few states as possible. The above three advanced algorithms have not been implemented in PROD.

In PROD, the reachability graph generator program unfolds a Pr/T-net into a P/T-net and applies the stubborn set method to the P/T-net. Applying the stubborn set method directly on the Pr/T-net level would not give us more freedom in net description because we should still go through such transition instances that are not enabled at the current marking but could possibly be enabled at some reachable marking. (This has been said in other words in [16].) `probe` folds the state space information back into the Pr/T-net level. `probe` sees only the generated part of the state space, so an answer to a query just tells about the reduced reachability graph.

The aim of the stubborn set method is to reduce the *branching factor* (number of successors) at each actually generated marking. The idea is the following. Consider a marking, say M, whether reachable or not, that has at least one enabled transition. Now, try to construct a partition of the the set of transitions T into two sets, say T_s and $\overline{T_s}$, so that the following diagram commutes for some $t \in T_s$ and any $\sigma \in (\overline{T_s})^*$ such that $M\ [t\rangle\ \wedge M\ [\sigma\rangle$:

$$\begin{tikzcd}
M \arrow[r, "\sigma"] \arrow[d, "t"'] & M' \arrow[d, "t"] \\
M''' \arrow[r, "\sigma"'] & M''
\end{tikzcd}$$

If one furthermore can guarantee that firing any $\sigma \in \overline{T_s}^*$ cannot enable any transition in T_s, there is no need to generate any of the internal markings on the path $M\ [\sigma\rangle\ M'\ [t\rangle\ M'''$: any terminal state that can be found by generating such a path, can be found by firing some enabled transition $t \in T_s$, if such exists.

A glimpse of the relevance of the stubborn set method can be obtained by considering the state space produced by $k > 1$ strictly sequential, $n > 0$-transition, entirely independent (unconnected) Petri net components. The state space of these is a k-dimensional hypercube, with $n + 1$ states possible in each dimension, giving a total of $(n + 1)^k$ states. The stubborn set method will only consider one path through this hypercube, a total of $nk + 1$ states, a reduction to polynomial size. Adding a transition that is enabled in an internal state of the hypercube, the method will force that state to be generated and the firing of the added transition to be investicated.

Valmari has given several algorithms to compute the set of enabled transitions of a stubborn set. We shall describe the so called *incremental algorithm* [9, 11]. Given a method to pick a place (the *scapegoat*) that disables a non-enabled transition, any marking M determines a binary relation $R_M(t, t')$: "if t is in a stubborn set, so must be t', immediately due to the 'safe part' of the definition". From now on, we shall assume that such a method to pick the scapegoat exists and is efficiently computable. The algorithm will work for any well-defined method, but the reduction in the number of states varies. We omit the definition of stubbornness because the main idea of the algorithm is completely independent of the actual definition of stubbornness. It is sufficient to know that the 'safe part' of the definition guarantees that the above diagram commutes for all enabled transitions in the stubborn set. On the other hand, the definition used by `PROD` [9] is extremely difficult to understand without simplifying assumptions about the P/T-net.

Starting from some enabled transition t and computing the set of enabled transitions in the first-found strongly connected component of the graph (T, R_M) that really has an enabled transition using Tarjan's algorithm [8], we have the set of enabled transitions of a stubborn set. Such strongly connected component will be found, since t was enabled, though $transition$ need not be in the stubborn set found.

The correctness of the algorithm relies on the fact that Tarjan's component search algorithm will find a component only after all the components that can be reached from it can be found. If the algorithm is stopped as soon as it finds a component with at least one enabled transition, the component *together with* all the components that can be reached from it forms a stubborn set. But, we are only interested in the enabled transitions in the stubborn set, and the other components reachable from the found one do *not* contain enabled transitions since these were found earlier, so the others can be ignored. The time complexity of the incremental algorithm is $O(\mu\nu|T|)$, where T is the set of transitions, μ is the maximum number of input places of a transition, and ν is the maximum of the maximum number of input transitions of a place and the maximum number of output transitions of a place [9].

Without change in complexity, the incremental algorithm can be optimized to find such stubborn set that contains the least number of enabled transitions, where 'least' holds only with respect to the graph (T, R_M). All what is needed is to complete the depth-first search and application of Tarjan's algorithm so that all enabled and only enabled transitions are checked in the outermost loop of the search.

The complete state space of the dining philosopher system in Figure 4 has $3^n - 1$ states and $n(2 \cdot 3^{n-1} - 1)$ state transitions. The incremental algorithm always produces a reduced state space of only $3n^2 - 3n + 2$ states and $4n^2 - 3n$ state transitions [9]. If $n = 5$ this means a reduction from 242 nodes and 805 real arrows in the reachability graph of Subsection 4.1 to 62 nodes and 85 real arrows. If we assume the files mentioned at the beginning of Subsection 4.1, the reduced reachability graph can be produced by compiling 'ph.net' into a reachability graph generator program 'ph.exe' and running 'ph.exe -. -s ph.gph'.

The stubborn sets produced by the incremental algorithm may contain unnecessarily many enabled transitions. To solve this problem, Valmari has developed the so called *deletion algorithm.* The deletion algorithm finds a stubborn set which is minimal in the sense that no proper subset of its enabled transitions can be the set of all enabled transitions of any stubborn set. The stubborn set is found in time $O(\mu\rho|T|^2)$, where T and ν are as above, and ρ is the maximum number of adjacent transitions of a place [10, 11] . The deletion algorithm utilizes the definition of stubbornness completely, unlike the incremental algorithm. No one has presented any algorithm that would find a stubborn set having a minimum number of enabled transitions in polynomial time with respect to the number of places and transitions. Such set is not necessarily the best choice [11] but it is difficult to define a better simple goal.

In PROD, both the incremental algorithm and the deletion algorithm have been implemented. The definition of stubbornness for P/T-nets given in [9] is used for both, though all the definitions Valmari has used when presenting the deletion algorithm are strictly different from that definition. This is possible because the idea behind the deletion algorithm is common to all definitions of stubbornness. Two versions of the incremental algorithm were mentioned above, the 'original' and the 'optimized' version. Both of them have been implemented in PROD.

Acknowledgements

The overall design of PROD is mainly due to Marko Rauhamaa. Following persons have been developing PROD: 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. In addition to the developers, following persons have been documenting PROD: Peter Grönberg, Jaakko Halme, and Kari Hiekkanen.

References

[1] *Petri Nets: Central Models and Their Properties.* Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, September 1986, Brauer W, Reisig W, Rozenberg G (eds), Lecture Notes in Computer Science 254, Springer-Verlag 1987, 480 p.

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

[3] Dijkstra EW: *Hierarchical Ordering of Sequential Processes.* In Hoare CAR, Perrott RH (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, 72–93.

[4] Genrich HJ: *Predicate/Transition Nets.* In [1], 207–247.

[5] Hjort P, Lindqvist M, Muhonen T, Ruohtula E, Valkonen K: PRENA — Predicate/Transition Analyzer. Course material of Advanced Course of Petri Nets, Bad Honnef 1986, 5 p.

[6] Petri CA: *Kommunikation mit Automaten.* Schriften des IIM Nr. 2 (1962), Institut für Instrumentelle Mathematik, Bonn, Germany. English translation: *Communication with Automata.* Technical Report RADC–TR–65–377, Vol. 1, Suppl. 1, Griffith Air Force Base, New York 1966.

[7] Petri CA: *State-Transition Structures in Physics and in Computation.* International Journal of Theoretical Physics 21 (1982) 12, 979–992.

[8] Tarjan RE: *Depth-First Search and Linear Graph Algorithms.* SIAM Journal of Computing 1(1972) 2, 146–160.

[9] Valmari A: *Error Detection by Reduced Reachability graph generation.* Proceedings of the Ninth European Workshop on Application and Theory of Petri Nets, Venice 1988, 95–112.

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

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

[12] Valmari A: *Eliminating Redundant Interleavings during Concurrent Program Verification.* Proceedings of Parallel Architectures and Languages Europe '89 Vol. 2, Lecture Notes in Computer Science 366, Springer-Verlag, Berlin 1989, 89–103.

[13] Valmari A: *Stubborn Sets for Reduced State Space Generation.* Rozenberg G (ed), Advances in Petri Nets 1990, Lecture Notes in Computer Science 483, Springer-Verlag, Berlin 1991, 491–515.

[14] Valmari A: *Stubborn Attack on State Explosion.* Formal Methods in System Design 1 (1992), 297–322.

[15] Valmari A: *On-The-Fly Verification with Stubborn Sets.* Accepted to Computer-Aided Verification 93, June 28–July 1, 1993, Heraklion, Greece, 12 p.

[16] Varpaaniemi K, Rauhamaa M: *The Stubborn Set Method in Practice.* Jensen K (ed), Proceedings of the 13th International Conference on Application and Theory of Petri Nets, Sheffield 1992. Lecture Notes in Computer Science 616, Springer-Verlag, Berlin 1992, 389–393.

A Formulae in the query language

This appendix presents the most important part of the query language: the formulae. A formula is an argument of the 'query' command in `probe`. The part of the query language consisting of formulae will be called PQL. The syntax and semantics of PQL is presented. We do not get inside the atomic formulae in this presentation, and not inside the so called *firing classes* either. An atomic formula is typically a C-like expression about the markings of the places in the net. There is an extended version of this appendix showing that each formula in CTL (Computation Tree Logic) [2] can be transformed into a semantically equivalent formula in PQL. The extended version is available on request.

The *alphabet* of PQL is

$$\Sigma = \Psi \ \cup \ \Theta \ \cup \ \{true, false, not, and, or, node, step,$$

$$bpath, dpath, bspan, dspan, (,),\ ,\ \},$$

where Ψ is the set of atomic formulae and Θ is the set of firing classes.

Definition A.1 *The set of PQL-*formulae *is the least set $\Phi \subset \Sigma^*$ such that if ϕ, ϕ_1, ϕ_2 and $\phi_3 \in \Phi$ are formulae and $\theta \in \Theta$ is a firing class, then*

1. $\Psi \subset \Phi$,
2. $\mathtt{true} \in \Phi$,
3. $\mathtt{false} \in \Phi$,
4. $\mathtt{not}\ \phi \in \Phi$,
5. $(\phi_1\ \mathtt{and}\ \phi_2) \in \Phi$,
6. $(\phi_1\ \mathtt{or}\ \phi_2) \in \Phi$,
7. $step\ \phi \in \Phi$,
8. $bpath(\phi_1, \phi_2)\phi_3 \in \Phi$,
9. $dpath(\phi_1, \phi_2)\phi_3 \in \Phi$,
10. $bspan(\phi_1)\phi_2 \in \Phi$,
11. $dspan(\phi_1, \phi_2)\phi_3 \in \Phi$,
12. $fire(\theta)\phi \in \Phi$.

Definition A.2 *A* labelled directed graph *is a triple $\langle W, L, R\rangle$, where*

- *W is a finite set of vertices (net markings, worlds of a model),*
- *L is a finite set of labelled edges (transition instances), and*
- *$R \subseteq \{wlw' \mid w, w' \in W \text{ and } l \in L\}$ is a set of three-element strings defining the labeled directed edges (transition instances) of the graph.*

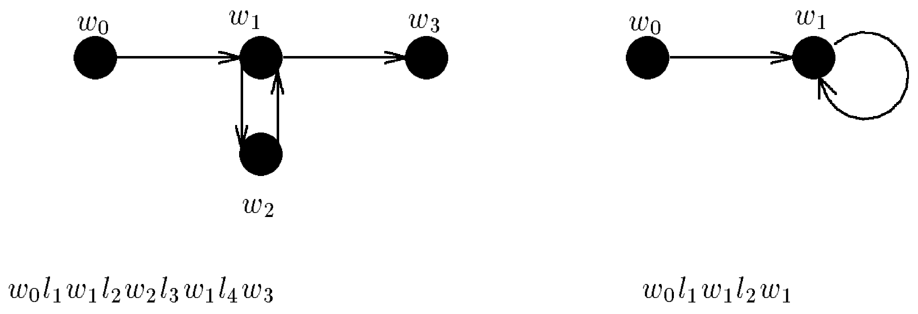

Figure 6: A cycle and an end cycle.

Definition A.3 *The set R^* of* paths *of a given graph $\mathcal{G} = \langle W, L, R\rangle$ and the projection $end : R^* \to W$ are defined as follows:*

- *for all $p = wlw' \in R$, $p \in R^*$ and $end(wlw') = w'$,*
- *for all $w \in W$, $w \in R^*$ (an empty path of zero length), and $end(w) = w$,*
- *if $w\lambda$ and $w'\lambda' \in R^*$ and $end(w\lambda) = w'$, then $w\lambda\lambda' \in R^*$, and $end(w\lambda\lambda') = end(w'\lambda')$,*
- *there are no other paths in R^*.*

Let R and R^* be as defined above. Then we say that $\langle W, L, R^*\rangle$ is a labelled graph, namely the graph $\langle W, L, R\rangle$.

A *cycle* is a path where the starting and end vertices coincide. If for a path $w_0\lambda = w_0 l_1 w_1 \ldots l_n w_n$ there are $i, j \in \{0, \ldots, n\}$ such that $i \neq j$ and $w_i = w_j$ we say that the path has a cycle. Specially, a path has *an end cycle*, if either i or j is n, that is, there is a world w' on the path such that $w_n = w'$.

In a graph $\mathcal{G}$, a path (that belongs to R^*) will be called *maximal*, if it is a path without cycles and it ends to a vertex with no successors in the graph or if it is a path with only one cycle and this cycle is an end cycle.

In this appendix, paths always have a finite length; the sets of vertices and labels are finite and the cycles can be thought as finite paths where the start and end vertices coincide.

Definition A.4 *Let $\mathcal{G} = \langle W, L, R\rangle$ and $\mathcal{G}' = \langle W', L', R'\rangle$ be labelled directed graphs.*

$\mathcal{G}'$ spans $\mathcal{G}$ iff

- *$W' = W$, and*

- *$R' = R_T \subseteq R$ in such a way that $\mathcal{G}'$ is a spanning tree of $\mathcal{G}$.*

$\mathcal{G}'$ spans $\mathcal{G}$ with cycles iff

- *$W' = W$, and*
- *$R' = R_T \bigcup R_C \subseteq R$ in such a way that $\langle W', L', R_T \rangle$ is a spanning tree of $\mathcal{G}$ and R_C consists exactly of all the labelled edges needed to define all cycles in $\mathcal{G}$.*

For paths we define the partial concatenation operator '$\circ$':

Definition A.5 *Let $p = w\lambda$, $p' = w'\lambda' \in R^*$ for some $\mathcal{G}$.*

- *$p \circ p' = w\lambda\lambda'$, if $end(p) = w'$ (that is, $p \circ p' \in R^*$), and*
- *undefined otherwise.*

If P is a set of paths, we can write $p \circ P$ for $\{p \circ p'' \mid p'' \in P\}$.

Definition A.6 *A formula $\phi \in \Phi_{PQL}$ is* true *in a world $w \in W$ for a model $\mathcal{M}$, with an interpretation $\mathcal{I}$ of formulae (defined later), $\mathcal{M} \models_w^{PQL} \phi$ iff $\mathcal{I}(w, \phi) \neq \emptyset$, otherwise it is* false *and we write $\mathcal{M} \not\models_w^{PQL} \phi$.*

For paths $p = w_0 l_1 w_1 \ldots l_n w_n$, $\mathcal{M} \models_p^{PQL} \phi$ means, that for all $0 \leq i < n$, $\mathcal{M} \models_{w_i}^{PQL} \phi$.

For a graph $\mathcal{G} = \langle W, L, R \rangle$, we define:

Definition A.7 *$R^*_{span}[w, \phi] \subseteq R^*$ is a set of paths*

$$\{p' = w\lambda' \in R^* \mid \mathcal{M} \models_{p'}^{PQL} \phi\}$$

that as a labelled directed graph $\langle W, L, R^*_{span}[w, \phi] \rangle$ spans $\langle W, L, \{p = w\lambda \in R^* \mid \mathcal{M} \models_p^{PQL} \phi\} \rangle$.

$R^*_{cspan}[w, \phi] \subseteq R^*$ is a set of paths

$$\{p'' = w\lambda'' \in R^* \mid \mathcal{M} \models_{p''}^{PQL} \phi\}$$

that as a labelled directed graph $\langle W, L, R^*_{cspan}[w, \phi] \rangle$ spans $\langle W, L, \{p = w\lambda \in R^* \mid \mathcal{M} \models_p^{PQL} \phi\} \rangle$ with cycles.

In this appendix, we regard atomic formulae to be known to be either true or false at a vertex and each firing class to define a set of edges ($\theta \in \Theta = 2^L$).

Definition A.8 *A model for the query language PQL is a triple $\mathcal{M} = \langle \mathcal{G}, \mathcal{V}, \mathcal{I} \rangle$,*

- *$\mathcal{G}$ is a labelled directed graph,*

- $\mathcal{V} : \Psi \rightarrow 2^W$ *is a local valuation of atomic formulae,*
- $\mathcal{I} : W \times \Phi \rightarrow 2^{R^*}$, ϕ, ϕ_1, ϕ_2 *and* $\phi_3 \in \Phi$, $\psi \in \Psi$, *is the following* interpretation *of PQL:*

  1. $\mathcal{I}(w, \psi) = \begin{cases} w & \text{if } w \in \mathcal{V}(\psi) \\ \emptyset & \text{otherwise} \end{cases}$,
  2. $\mathcal{I}(w, false) = \emptyset$,
  3. $\mathcal{I}(w, true) = w$,
  4. $\mathcal{I}(w, not\ \phi) = \begin{cases} \emptyset & \text{if } \mathcal{M} \models_w^{PQL} \phi \\ w & \text{otherwise} \end{cases}$,
  5. $\mathcal{I}(w, (\phi_1\ or\ \phi_2)) = \mathcal{I}(w, \phi_1) \cup \mathcal{I}(w, \phi_2)$,
  6. $\mathcal{I}(w, (\phi_1\ and\ \phi_2)) = \begin{cases} \mathcal{I}(w, \phi_2) & \text{if } \mathcal{M} \models_w^{PQL} \phi_1 \\ \emptyset & \text{otherwise} \end{cases}$,
  7. $\mathcal{I}(w, step\ \phi) = \{wlw' \circ P \mid wlw' \in R \text{ and } P = \mathcal{I}(w', \phi)\}$,
  8. $\mathcal{I}(w, bpath(\phi_1, \phi_2)\phi_3) =$
     $= \{p \circ P \mid p = w\lambda \in R^* \text{ has no cycles and } \mathcal{M} \models_p^{PQL} \phi_1, \text{ and } P = \mathcal{I}(end(p), \phi_3)\}$
     $\cup \{p \mid p = w\lambda \in R^* \text{ has only an end cycle and } \mathcal{M} \models_p^{PQL} \phi_1, \text{ and } \mathcal{M} \models_{end(p)}^{PQL} \phi_2\}$,
  9. $\mathcal{I}(w, dpath(\phi_1, \phi_2)\phi_3) = \mathcal{I}(w, bpath(\phi_1, \phi_2)\phi_3)$,
  10. $\mathcal{I}(w, dspan(\phi_1, \phi_2)\phi_3) =$
      $= \{p' \circ P \mid p' = w\lambda \in R^*_{cspan}[w, \phi_1] \text{ has no cycles and } \mathcal{M} \models_{p'}^{PQL} \phi_1, \text{ and } P = \mathcal{I}(end(p'), \phi_3)\}$
      $\cup \{p' \mid p' = w\lambda \in R^*_{cspan}[w, \phi_1] \text{ has only an end cycle and } \mathcal{M} \models_{p'}^{PQL} \phi_1, \text{ and } \mathcal{M} \models_{end(p')}^{PQL} \phi_2\}$,
  11. $\mathcal{I}(w, bspan(\phi_1)\phi_2) = \{p' \circ P \mid p' = w\lambda \in R^*_{span}[w, \phi_1], \mathcal{M} \models_{p'}^{PQL} \phi_1 \text{ and } P = \mathcal{I}(w, \phi_2)\}$,
  12. $\mathcal{I}(w, fire(\theta)\phi) = \{wlw' \circ P \mid wlw' \in R\ \&\&\ l \in \theta\ \&\&\ P = \mathcal{I}(w', \phi)\}$

      where $\theta \in 2^L$ *is a firing class.*

Note that:

- not necessarily $\mathcal{I}(w, not\ not\ \phi) = \mathcal{I}(w, \phi)$,
- not necessarily $\mathcal{I}(w, (\phi_1\ and\ \phi_2)) = \mathcal{I}(w, (\phi_2\ and\ \phi_1))$,
- $\mathcal{I}(w, dspan(\phi_1, \phi_2)\phi_3) \subseteq \mathcal{I}(w, dpath(\phi_1, \phi_2)\phi_3)$,
- $\mathcal{I}(w, bspan(\phi_1)\phi_2) \subseteq \mathcal{I}(w, bpath(\phi_1, false)\phi_2)$.

bpath and *dpath* differ in how `probe` implements them; the search is done either breadth-first or depth-first.

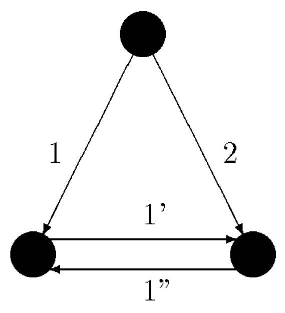

Figure 7: An undetectable cycle for *bspan*.

Spanning operators are much like their 'path' counterparts but here we do not demand all possible answers; every time a vertex is visited, it is marked and no additional edges to that vertex will be used as a part of a path (the search spans a subgraph).

This is true as is for *bspan* where, when constructing p', the part of the path satisfying the first set condition (see def. A.8), every edge leading to an already visited vertex will be rejected. Even if the vertex was visited on the path we are currently on. This means that p' won't contain cycles. The reason why we do not allow cycles while constructing p' for *bspan* is because there are cycles that wouldn't be detected even if we allowed additional edges to a vertex already visited on the current path.

Figure 7 gives an example of this. Even if we were trying to detect cycles by making difference between vertices visited on the current path and vertices visited otherwise, the cycle of this graph would be undetected. Traversing the graph in breadth-first manner starting at the parent vertex, the children would be visited first using labelled edges 1 and 2. Traversing the edge 1' would lead to a vertex visited on the second path and so that additional edge to that vertex would be rejected. Traversing the edge 1" would lead to a vertex visited by the first path so this edge would also be rejected and no cycles would be found. Even if some cycles could be found, it was decided that no cycles might be better than 'some cycles'; this because otherwise the user could make a query about some kind of a cycle and then the query could state that no such cycles were found even if there were some.

The *dspan* is a little different. Allowed to use a new edge when going to a vertex visited earlier by the current path, it will find at least one path with a cycle for every cycle found by *dpath* (or *bpath*).

We close this appendix by stating two quite obvious but important things about span queries relatively to the corresponding path queries:

$$\mathcal{M} \models_w^{PQL} dspan(\phi_1, \phi_2)\phi_3 \Leftrightarrow \mathcal{M} \models_w^{PQL} dpath(\phi_1, \phi_2)\phi_3,$$

and

$$\mathcal{M} \models_w^{PQL} bspan(\phi_1)\phi_3 \Leftrightarrow \mathcal{M} \models_w^{PQL} bpath(\phi_1, false)\phi_3.$$