A Guided Tour of P-NUT (Release 2.2)

by

Rami R. Razouk
Information and Computer Science Dept.
University of California, Irvine

Abstract

P-NUT is a suite of tools for constructing and analyzing Petri Net models. The tools have been developed at UCI to aid researchers in applying Petri Nets to the design of concurrent hardware/software. The tools support state-space analysis, simulation, performance evaluation and verification. While the tools are useful in their current state, the P-NUT system is just beginning to achieve its overall objective of aiding in the design of complex distributed real-time systems. This report provides a guided tour of the tools for researchers who are interested in exploring P-NUT's capabilities.

Technical Report #86-25
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

January 1987

A Guided Tour of P-NUT (Release 2.2)

Rami R. Razouk
Information and Computer Science Dept.
University of California, Irvine

The purpose of this report is to introduce the reader to the P-NUT [1] suite of tools. The reader is assumed to understand Petri Nets. For an introduction to Petri Nets the reader is referred to [PetJ 81] and [Age79]. For further information about Petri net extensions the reader is referred to [Dia82]. Additional references are provided throughout this document.

1 Introduction and Disclaimers

Before beginning this guided tour it is important to understand some of the history and objectives of the P-NUT system in order to accurately evaluate its merits and deficiencies.

The P-NUT system is a set of tools developed by the Distributed Systems Project in the Information and Computer Science Department of the University of California at Irvine (UCI). The development of the tools has been funded in part the the National Science Foundation and by a series of MICRO grants funded jointly by the State of California and Hughes Aircraft. All the software in release 2.2 is available free of charge. With one exception, release 2.2 of P-NUT includes ALL the source code as well as some demonstration files which will be used throughout this document. The exception is the source code for a tool (TRACER-TOOL) which is a recently developed graphics program for debugging simulation models. This tool is very experimental and executes only on SUN 3's. Because of the experimental nature of this tool (and known bugs) it is only distributed to those sites which can execute the binaries directly.

The tools have been built to assists researchers (faculty and graduate students of UCI) in applying various Petri net based analysis methods which were developed at UCI and elsewhere. The overall objective of P-NUT is to support the design

[1]This work was supported in part by a MICRO grant co-sponsored by Hughes Aircraft and the University of California, and by a grant from the National Science Foundation (DCR 84-06756)

of complex distributed systems. A heavy emphasis is placed on models which support concurrency. Time-dependent behavior is also a major concern. The current tools are prototypes which are intended to show the feasibility of applying analysis methods to real problems, and are intended to provide guidance for future tool development. All software was developed by students and faculty of the Information and Computer Science Department. Although great care was taken in developing the software, no guarantees are made or implied about the correctness of the software or its suitability for use outside the University. Although members of the research group are eager to find out about (and correct) possible problems which may exist in the software, the software is NOT supported.

The development of P-NUT began in the summer of 1984. The system is therefore in its infancy. Only a few of the overall goals of the P-NUT system have been met. Development is continuing at a frantic pace. Therefore, this release of P-NUT (2.2) is an early release and will become obsolete by the summer of 1987. Release 3.0 will be a significantly enhanced version of the system which will rapidly displace the older release. This release is intended for researchers who are interested in investigating the use of the tools in some research area. The tools are not ready for use in large development efforts. Release 2.2 of P-NUT is only suitable for systems running some version of 4.2bsd UNIX [2] (e.g. SUN's version 3.0, ULTRIX). Significant changes must be made to this software in order to make it run on other UNIX systems (e.g. EUNICE on VMS).

Now that we have dispensed with the preliminaries, Welcome to P-NUT!!!

2 Installing P-NUT

Release 2.2 of P-NUT is available in UNIX **tar** format on 1/2 inch tape or 1/4 inch cartridge tape (for SUN computers). The tape includes binaries for SUN 3's, but the source can be used to *make* binaries for any 4.2bsd compatible system. In order to install the software a directory should be created to house all the source and binary files. It is preferred that a new user account be set up with the name "pnut". The user should go to the pnut home directory and type:

```
tar xvf /dev/rmt8
```

where rmt8 is the device name of the tape drive. The tape has been prepared so that it will create several subdirectories within the pnut directory:

[2] UNIX is a Trademark/Service Mark of the Bell System.

bin The bin directory contains all executable binaries and shell scripts. In order for a user to use pnut, the pnut/bin directory will have to be added to his/her search path ($PATH).

src The src directory contains all source code and the main Makefile for P-NUT. It is further subdivided into a set of directories, one for each tool in the system. ALL the source code for release 2.2 is written in C.

doc The doc directory is intended to contain some tool user manuals and some documentation. Documentation of release 2.2 is VERY scarce. The main documentation is in the form of comments in the source code.

man The man directory contains online manual information. This information is organized much like UNIX's man facility (In fact the pnut manual macros were taken from UNIX's man macros). The pnut man directory is organized as a set of numbered subdirectories, one per manual section. Release 2.2 contains only one manual section.

demo The demo directory contains several demonstration files. The important files for the purpose of this guided tour are name **dining.tpp** and **mp.net**. These files are discussed further in the following sections.

help The help directory contains help files used by the tracertool program. This directory will only be needed on SUN 3 systems which will run the tracertool program.

The entire P-NUT system requires roughly 4500 blocks of free disk space (2000 for source, 2500 for binaries). Once the tape has been read in, the binaries must be created. If the system on which the code will execute is a SUN 3, the binaries will execute directly. No recompilation is necessary. If, on the other hand, the code is to execute on other hardware supporting 4.2 UNIX, then the binaries will have to be created. This can be done by using the **make** facilities of UNIX. A Makefile can be found in pnut/src which can make and install all the p-nut tools. To make and install the system all that must be done is to type:

```
make "BIN= binary-directory"
```

where binary-directory should be replaced by the full path name of the directory where the binaries should reside. The default bin directory is "/g/ds/pnut/bin" (specific to UCI's machines).

In order to see the effect of the make command, it is also possible to type

```
make "BIN=binary-directory" test
```

The result is an echoing of all the commands which will be executed (the commands will not be executed).

Once the tools have been compiled and installed it is possible to remove source files created by lex and yacc by typing:

```
make veryclean
```

The cleaning command saves disk space and is strongly recommended for users who are not planning on modifying the code in any way.

Building the system may require several hours. If difficulties are encountered the user should contact Rami Razouk in the ICS Department (714-856-6354 or 714-856-7403).

Once the system has been built some minor changes to some shell scripts are necessary before using P-NUT. The bin directory contains a c-shell script named **pnutman**. This script contains the path name of the manual directory. This path name should be edited to reflect the correct path. The commands needed to run troff must also be tailored to the new environment (some sites use dtroff in place of troff). The bin directory also contains c-shell script name **printstat**. This script also invokes troff and should also be tailored to the new environment. Both **printstat** and **pnutman** use **tseetool**, a troff pre-viewer available for the SUN's. If **tseetool** is not available, the shell script should be modified to use nroff and more. The final script which may require modification is the **tracertool** script. This file is only used on SUN 3 systems running the **tracertool** program. The script invokes the tracer program and passes it the path name of the help files. This path name should be changed to reflect the new home directory for P-NUT. P-NUT is now ready to run.

3 A word about the P-NUT 'environment'

P-NUT has been built as a set of small tool fragments (see figure 1), each dedicated to performing some limited task (efficiently). Using P-NUT involves putting together these tool fragments in interesting ways. Tool fragments are 'put together' by passing the output of one tool fragment as input to another. UNIX's pipes can be used for that purpose to avoid storing intermediate results in files. Whenever intermediate results are to be used repeatedly, they should be stored as files on disk in order to save processing time. P-NUT tools have been built in the (best?) UNIX tradition. They read their inputs for 'standard input' and produce output to 'standard output'. Unless the user specifically changes standard input and

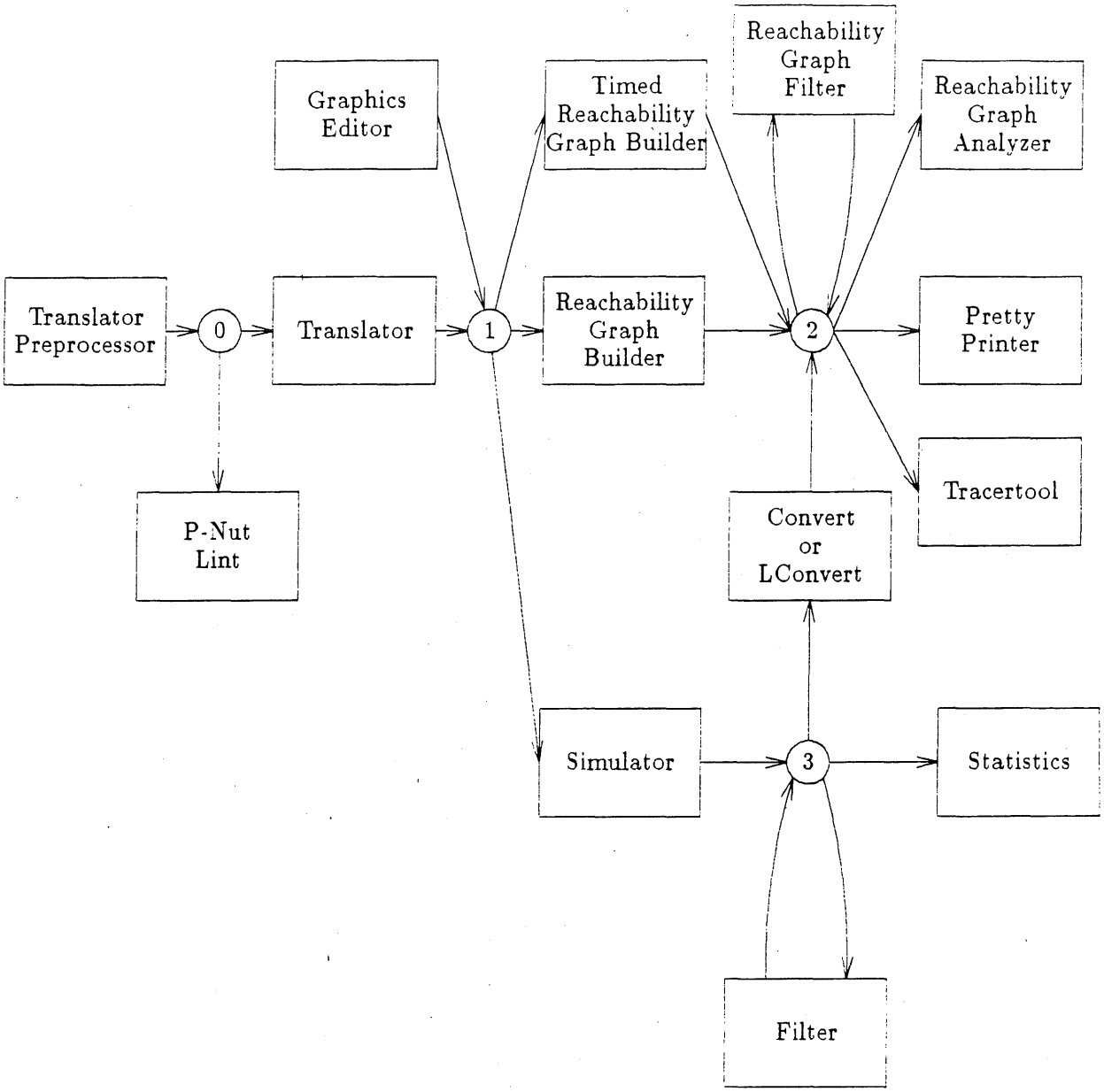

Figure 1: The P-NUT suite of tools

standard output, the tools will expect input from the terminal and will produce their results (sometimes ugly intermediate results) to the terminal.

The tools interact in fairly simple ways. There are really only three interesting types of objects that can be created and manipulated in P-NUT (depicted as

circles in 1):

1. Petri Nets
2. Reachability Graphs
3. Execution Traces

Petri nets (circle 0 in figure 1) are accepted by the system in textual form and transformed into an internal representation (circle 1 in figure 1). It is this internal representation that drives all remaining tools.

Reachability graphs (circle 2 in figure 1) are representations of all of (or part of) the state-space of a Petri Net. These are graphs whose nodes represent states (there are many representations depending on what you are interested in) and whose edges represent state transitions. Depending on the technique used to produce the graph, edges may contain more or less information. For example, timed reachability graphs contain timing information while untimed graphs do not. Graphs produced by some tools have special properties. For example, graphs produces by **Lconvert** (linear convert) are linear sequence of states, i.e. every state has exactly one successor except for some terminal state.

It is important to note that a reachability graph can only be interpreted (and understood) if the net from which it is constructed is known. The binding of a reachability graph to a net is permanent and is handled rather crudely in P-NUT: the reachability graph **includes** the complete specification of the net. Clearly, storage gains could be achieved if P-NUT was interfaced with some sophisticated database management system which could maintain this binding information.

Execution traces (circle 3 in figure 1) are representations of portions of the state space of a net. An execution trace is one long path through that state space, where the same state may be visited many many times. A trace is represented by some initial state followed by a set of state deltas. It is a space-efficient way of representing a linear sequence of states (when compared to a reachability graph). As is the case with graphs, execution traces are specific to nets. The binding of an execution trace (or a series of traces) to a net is also permanent and is handled crudely by P-NUT. Each execution trace **includes** the complete specification of the net. A benefit of this crude mechanism is that the user need not worry about the binding.

All intermediate forms are in ASCII format and can be examined easily. While this is a particularly useful feature for P-NUT developers, it is not intended for P-NUT users. The only other objects created (but not manipulated) by the tools are user output (e.g. performance statistics reports).

The remaining sections of this report give a guided tour of the system. The files included in the *pnut/demo* directory will be used throughout the report. To use these files the reader should copy them into a private directory.

4 Creating A Petri Net

Two tools can be used to create a Petri Net. The main tool is the 'translator' (**transl**) which transforms the textual form of a Petri Net into its internal form. A textual description of a Petri Net consists of a listing of all the transitions in the net. For each transition, the user can specify:

1. Data-dependent preconditions *(optional)*
2. Name *(optional)*
3. Input places (control preconditions)
4. Timing information *(optional)*
5. Output places (control postconditions) *(optional)*
6. Actions (data transformations) *(optional)*

Data-dependent preconditions are boolean expressions on data variables. These conditions are not supported by any tool (other than the translator) in release 2.2.

A transition can be given a name in order to simplify analysis. The name must appear in a pair of ':'.

Input places consist of a comma-separated list of places which must hold tokens before the transition can fire. If multiple tokens are needed, the number of tokens is specified (in parentheses) after the name of the place. Inhibitor arcs (arcs which inhibit the firing of a transition if tokens are present in a place) can be described by specifying (in parentheses following a particular place name) that zero tokens are required in a place. For example fork(0) specifies that the place named fork must have zero tokens on it in order for the transition to be enabled. In the model supported by P-NUT if a transition's input places have enough tokens to allow the transition to fire multiple times, the transition is considered to be enabled many times concurrently. As a consequence, it is not possible to have a transition whose only inputs are inhibitor inputs. In such a case the transition would be considered enabled and infinite number of times.

Timing information may include enabling times (delays before a transition can begin to fire), firing times (duration of firing) and relative firing frequencies (the

```
:t1: p1 -> p2(2), p3
:t2: p2 -> p4
:t3: p3 -> p5
:t4: p4(2), p5 -> p1
<p1>              /* initial state */
```

Figure 2: Simple Petri Net

frequency with which this transition fires relative to other conflicting transitions). This timing information is consistent with the model of time described in [RP84]. Currently, only constants can be used to specify these values. The simulator allows for these constants to be used as means of exponential distributions. The most significant change in release 3.0 is expected to be support arbitrary timing expressions.

Output places consist of a comma-separated list of places which will gain tokens when the transition finishes firing. If multiple tokens are added to a place, the number of tokens (an integer constant) is specified (in parentheses) after the name of the place. It is possible for a transition to have no output places.

Actions are program segments which cause data variables to change. Currently an interpreted language developed for the reachability graph analyzer [MR85] (see section 6) is used. No tool, other than the translator supports these actions (release 2.2 supports "uninterpreted" models). Release 3.0 will support these actions for simulation purposes.

Figure 2 shows a small example of a Petri Net. The reader can create and store the net by invoking the translator without redirecting input. This is accomplished by typing:

```
transl > net.out
```

Since the translator will expect input from the terminal, the net can be typed in directly (without mistakes!). When the net is completed (by typing a <CTRL>-D for an end-of-file) the translator will produce the net in the file *net.out*.

The translator is not intended to be used interactively and does not tolerate errors. The normal way of using the translator is to create a file (using an editor) which holds the textual description of the net. In order to facilitate debugging of nets a new tool (**pnl**) has been added to release 2.2. **Pnl** (P-NUT Lint, patterned after the lint program in UNIX) scans the net for syntactic errors as well as patterns which often indicate semantic errors. For example, **pnl** will warn the

```
/*
 * Dining philosophers problem with three philosophers.
 * Set n equal to the desired number of philosophers.
 */

for n=3 {

array philosopher_thinking(n), philosopher_1_fork(n), philosopher_eating(n)
array fork_free(n)

for i=0 to n-1 {
philosopher_thinking[i], fork_free[i] -> philosopher_1_fork[i]
philosopher_thinking[i], fork_free[(i+1) % n] -> philosopher_1_fork[i]
philosopher_1_fork[i], fork_free[i] -> philosopher_eating[i]
philosopher_1_fork[i], fork_free[(i+1) % n] -> philosopher_eating[i]
philosopher_eating[i], fork_free[i](0),
        fork_free[(i+1) % n](0) -> philosopher_thinking[i], fork_free[i],
                                   fork_free[(i+1) % n]

<philosopher_thinking[i], fork_free[i]>
}
}
```

Figure 3: TPP input for dining philosophers

user if a place is used at the output of a transition but not at the input of another. While such a pattern does not always signal an error it is often caused by simple typographical errors. If no errors or warnings are found **pnl** simply reports how many places and transitions were found in the net. Note that **pnl** only produces a short report as its output. It does not duplicate the actions of **transl**.

Another tool which aids in creating Petri Nets is the translator pre-processor (**tpp**). As its name implies it is a pre-processor which supports more compact representations of nets whose structure is regular. It allows the user to specify a net for one component and then to replicate that component (using looping constructs) in order to create the complete net. The function of the processor is to "unroll" the loops and to create a net which can be processed by **transl**. File *dining.tpp* in the *pnut/demo* directory contains the dining philosopher net in format suitable for **tpp**. Figure 3 shows that form.

By changing the assignment to n it is possible to create nets for different numbers of philosophers. A 5-philosopher net can be created by changing the file to assign 5 to the variable n and then typing:

```
tpp dining.tpp | more
```

The output is the textual equivalent which can be processed by **transl**. It should be clear what **tpp** does. To check the validity of the tpp output, it can be sent to **pnl** by typing:

```
tpp dining.tpp | pnl
```

Note that **pnl** cannot process the input to **tpp** directly. The output of **tpp** is normally piped directly to **transl**. The intermediate form of the 5-philosopher net can be generated by typing:

```
tpp dining.tpp | transl > dining5.pn
```

File *dining5.pn* now contains the 5-philosopher net.

Tpp and **transl** support Timed Petri Nets. Timed nets include fixed enabling and firing times, and firing frequencies. Figure 4 shows an example of a simple shared-bus multiprocessor system. This model can be found in file *mp.net* in the *pnut/demo* directory. The model of a single processor is given. The model is expanded into multiple processors using **tpp**. In this model, the processor fetches instructions from local memory (consuming some cycles), decodes the instructions, and then executes the instructions by fetching operands (if any), executing the instruction and writing results (if necessary) back to memory. Interaction between processors occurs when they contend for access to the bus (places *bus_free* and *bus_busy*) to read some operands and to write some results. The reader should examine this model closely before going on. The parenthesized numbers in the model describe enabling times, firing times, and firing frequencies. All firing times are zero in this model because processing and propagation delays are modeled with enabling times. Probabilities (such as the probability of getting a particular type of instruction) are provided as the third number. Sometimes these probabilities are given as real numbers which add up to one (as probabilities should) or as integers from which the actual probabilities are calculated (e.g. firing frequencies of 1 and 1 yield probabilities of 0.5 and 0.5). The calculation of probabilities in Timed Petri Nets is a complex issue, especially if transitions are allowed to be enabled multiple times concurrently. For further insight into this problem the reader is referred to [WPS86]. Note the use of inhibitor arcs associated with place

```
for n = 1 {
array fetch(n), ready(n), read(n), execute(n), write(n)
array nowrite(n), noread(n), local_read(n), remote_read(n)
array bus_read(n), done(n), local_write(n), remote_write(n)
array bus_write(n)

for i = 0 to n-1 {
fetch[i] -> (2, 0, 1) ready[i]  /* fetch instruction */
ready[i] -> (0, 0, 0.1) read[i], execute[i], write[i]
ready[i] -> (0, 0, 0.4) read[i], execute[i]
ready[i] -> (0, 0, 0.2) noread[i], execute[i], write[i]
ready[i] -> (0, 0, 0.3) noread[i], execute[i]
read[i] -> (0, 0, 0.6) local_read[i]
read[i] -> (0, 0, 0.4) remote_read[i]
local_read[i] -> (2, 0, 1) noread[i]
remote_read[i],bus_free -> (0, 0, 1) bus_read[i], bus_busy
bus_read[i], bus_busy -> (3, 0, 1) bus_free, noread[i]
noread[i], execute[i] -> (3, 0, 1) done[i]
done[i], write[i](0) -> (0, 0, 1) fetch[i]
done[i], write[i] -> (0, 0, 6) local_write[i]
done[i], write[i] -> (0, 0, 4) remote_write[i]
local_write[i] -> (2, 0, 1) fetch[i]
remote_write[i], bus_free -> (0, 0, 1) bus_write[i], bus_busy
bus_write[i], bus_busy -> (3, 0, 1) bus_free, fetch[i]
<fetch[i]>
}
}
<bus_free>
```

Figure 4: Petri Net Model of Multiprocessor

write. In this model the presence of in token in that place indicates that a write to memory is needed. The absence of the token signals that no write is necessary.

By altering the assignment to n it is possible to generate nets for different numbers of processors. A 1-processor system can be created by assigning 1 to n and then typing:

```
tpp mp.net | transl > mp1.pn
```

The last tool which is useful in creating Petri Nets is a graphics editor which is currently under development. This editor is not included in release 2.2. It is mentioned here in order to answer the frequent question regarding graphics input capabilities in P-NUT. Release 3.0 of P-NUT should include a graphics editor.

5 Building and Printing Reachability Graphs

A reachability graph is a finite representation of the complete state space of the system being modeled. A reachability graph consists of nodes which represent distinct states which can be reached, and arcs which represent potential state transitions. Since the state-space of a concurrent system is usually quite large, it is very difficult (if not impossible) to construct the complete state-space. Rather, a finite representation of the complete state-space is constructed by focusing attention on parts of the system which are of interest. In Petri Net, this focusing of attention can be easily done by omiting portions of the model of the system. For example, by ignoring timing information, it is possible to construct a reachability graph which retains only control-flow information (distribution of tokens on the net). In this type of graph, the "state" of the system is completely described by the distribution of tokens on places. Arcs in such a graph can be labeled by the Petri Net transitions which cause the system to go from the source state to the destination state. It is this type of graph that is discussed at length in [Pet81].

Other types of reachability graphs can be constructed. In [RP84] a "Timed" reachability graph is defined where each state is characterized by the token distribution and by timing information about the various transitions in the net. The arcs in these graphs are labeled with sets of transitions which begin or end firing simultaneously, and with a timing delay describing the amount of time it takes to go from one state to another. Whenever a state has multiple successors, each successor has a probability (actually associated with the arc).

Yet another type of reachability graph is one which can be constructed from a simulation trace. It is possible to summarize simulation results by constructing a graph which represents all the states which were actually reached during a

simulation. Arcs in such "partial" reachability graphs can be labeled with timing delays and probabilities (actual frequencies with which arcs were traversed). P-NUT provides separate tools for creating each type of graph.

5.1 Un-timed reachability graphs

Un-timed reachability graphs can be built using the **Reachability Graph Builder (RGB)**. **Rgb** can build only finite graphs (infinite graphs will be dealt with in release 3). **Rgb** has been designed to be efficient in space and time by taking advantage of the modeler's knowledge of the model [RH85]. If the model is known to be bounded at less than 127 (no place ever holds more than 127 tokens), the program can save a good deal of memory. If the model is known to be safe (bounded at 1), the program can save both time and space. This last version of **rgb** has successfully built graphs of 20,000 states in less than seven minutes of cpu time on a VAX 11/750. [3]

In order to experiment with rgb, the reader should try to construct the reachability graph for the 5 dining philosophers. Since this net is safe, its reachability graph can be built by typing

```
rgb -s dining5.pn > dining5.rg
```

This graph is built relatively quickly (considering its size). To see the results of the analysis, it is possible to print the graph using the **Reachability Graph Printer (rgp)**. Try typing:

```
rgp dining5.rg | more
```

By piping the result of **rgp** through the **more** program, the graph can be viewed slowly. **rgp** displays the graph as a tree, and numbers all the states. A detailed description of each state is listed at the end of the display. In this case the graph is large and difficult to scan (by a human). Analysis of reachability graphs is discussed in section 6. Various parameters of **rgp** can be used to display portions of the graph. By typing

```
rgp -b -s10 -d2 dining5.rg | more
```

rgp will display the portion of the graph at distance 2 backward from state 10.

[3] VAX is a Trademark of Digital Equipment Corporation.

5.2 Timed reachability graphs

Timed reachability graphs can be built using the **Timed Reachability Graph Builder (TRGB)**. In release 2.2, **Trgb** has been generalized to deal with any type of timed Petri Net so long as the enabling times and firing times are fixed (GSPN's and DSPN's are not supported [MBC84][MC86]). In many cases timed reachability graphs are very large. **Trgb** does not detect potentially infinite graphs. It simply continues to execute until it runs out of memory (or disk space). This tool is primarily useful for systems with small state spaces.

An example of a timed reachability graph can be obtained by analyzing a 1-processor model of the multiprocessor system described earlier. To accomplish this, the *mp.net* file should be edited to ensure that a 1-processor model is created (by assigning 1 to n). Once the editing is completed a 1-processor model can be built by typing:

```
tpp mp.net | transl > mp1.pn
```

This model can then be analyzed using **trgb** by typing:

```
trgb < mp1.pn > mp1.trg
```

The graph is now stored in file *mp1.trg* and can be displayed by typing:

```
rgp mp1.trg | more
```

Note that a timed reachability graph contains different information than an untimed one. Arcs are labeled with probabilities and time delays (zero probabilities and zero delays are omitted). The description of a state includes information about which transitions were enabled (Remaining Enabling Time: RET) and which transitions were firing (Remaining Firing Time: RFT). Note also that the probabilities appearing in the graph are consistent with those specified in the model. A 2-processor model can also be analyzed in this way, but its graph is considerably larger.

5.3 Partial Reachability Graphs

The third technique for generating reachability graphs is to transform the output of the simulator (an execution trace) into a graph. Such a graph is NOT a representation of the complete state-space. It only represent the small portion of the state-space which was visited during a particular simulation experiment.

The generation of execution traces will be discussed later. For now it is sufficient to say that two conversion program (**convert** and **lconvert**) can be used to transform an execution trace into a graph. First, **lconvert** is intended to simply change the form of the execution trace. All the information is retained. Rather than representing an execution as a state followed by a sequence of state deltas, the execution is represented as a collection of states connected by edges. Graphs produced by **lconvert** have the peculiar characteristic that each state has exactly one successor (except for the final state) and that each state has exactly one predecessor (except for the initial state). The name of the tool (**lconvert**) stands for Linear Convert. The main reason to convert the form of an execution trace is to use the reachability graph analysis tools on a trace.

The second conversion program (**convert**) attempts to reconstruct the state-space of the system being modeled in the form of a general graph (not linear). To accomplish this goal **convert** recognizes states when they are revisited. When a state is first visited during a simulation run, the state is added as a node in the graph. When that state is revisited, an arc is added to the original node representing that state. In this release of P-NUT the **convert** program merges duplicate arcs between nodes. If two arcs are found between a pair of states they are combined into one. The delay associated with the arc is the (weighted) average of the two delays. The weight associated with an arc is the number of times it was traversed during an execution of the model. These weights are used in the delay calculations as well as in assigning probabilities to arcs. The graph produced by **convert** is generally orders of magnitude smaller than the equivalent execution trace. Unfortunately, valuable information is lost in this conversion process.

- The individual delays encountered in going from one state to another are lost (only their average is known). Future releases may permit additional data to be remembered such as variance (or possibly the representation of a distribution).

- The particular events which caused a state transition are also lost. In the process of merging arcs it is not possible to preserve the information about the individual events (they can't be averaged).

- Information about sequences of states are lost. By converting into a graph some sequences will appear to be possible in the graph that in fact never appeared in the execution trace. This is best illustrated by an example. If the following trace is generated (where the S's are states):
  $S_1 S_2 S_3 S_1 S_4 S_5$ ...
  the resulting graph will show a path from S_3 to S_2 (since S_3 can reach S_1 and S_1 can reach S_2).

An example of converting an execution trace will be discussed later.

5.4 Filtered Reachability Graphs

Release 2.2 of P-NUT includes a tool which is capable of producing one reachability graph from another. The **Reachability Graph Filter (rgf)** program allows the user to construct a subgraph by defining the characteristics of the states which are to be retained. These characteristics are specified in the form of predicates in the **RGA** language described below. All states satisfying a predicate are retained and all arcs between retained states are retained in the resulting subgraph. **Rgf** therefore allows the user to focus attention on small sets of states which may be of interest. Note that it is possible to create a set of disconnected subgraphs using **rgf**.

6 Analyzing Reachability Graphs

Once reachability graphs have been constructed they can be analyzed interactively using the **Reachability Graph Analyzer (RGA)**. This tool allows a designer to debug models and even to prove correctness. The input language is first order predicate calculus with the addition of branching-time temporal logic operators. In some sense the function of the tools is to accept a high level specification and to verify that the behavior of the model (as represented by the reachability graph) is consistent with that specification. Another way to view the tool is as an interactive mechanism for "asking questions" about the model. In the discussion below we will be using the dining philosopher net as an example. To analyze the reachability graph for 3 philosophers (the graph constructed earlier) type:

```
rga dining5.rg
```

The tools is "expression oriented" in that it views anything typed in at the terminal as an expression which needs to be evaluated. For example typing the expression

```
2+3
```

will cause **rga** to respond with **5**. Expressions can be constructed using arithmetic, boolean, and set operators with existential and universal quantifiers. Simple data types such as integers, reals, sets and sequences are supported. Assignment statements and simple control structures (if-the-else) are also supported. In order

to answer questions about a graph **rga** has an understanding of some predefined sets. It knows of the set of states in the graph (S) and the set of arcs in the graph (A). The details of the capabilities of the tool are given in the **rga** user manual [Mor84].

The simplest question to ask **rga** is the number of states and arcs. The cardinality of these sets can be obtained by typing:

```
card(S)
card(A)
```

In order to find out is there are any states which have no successors (deadlocked states) it is sufficient to ask if there are any states whose number of successors is 0.

```
exists s in S [ nsucc(s) = 0 ]
```

Of course, this model of dining philosophers has a deadlock. To capture the set of deadlocked states it is possible to create the set and assign it to a variable.

```
deadlocks := { s in S | nsucc(s) = 0 }
```

The assignments also produces the value being assigned. Therefore, a side-effect of assigning the set to the variable deadlocks is that the set is also displayed at the terminal. Ending a line with a ';' causes the value of the expression to be discarded. The reader should note that there is only one deadlocked state. To display the marking in that state the **setop** function can be used. **setop** applies an arbitrary function (in this case the **showstate** function) to a set. Therefore the set of deadlocked states can be displayed by typing

```
setop(showstate, deadlocks)
```

Rga also supports the definition of functions. It is possible, for example, to define a function which, given a state, returns the number of philosophers which are eating in that state. Such a function can be defined as follows:

```
eating(s) [n] ::= n := 0; \
          forall p in philosopher_eating [ n := n+p(s); true];\
          n
```

This function has one formal parameter (s) and one local variable (n). The variable n keeps a running total of philosophers eating. The forall statement in this case is used as a looping constructs which traverses the set of places called *philosopher_eating*. This set of places was defined in the net as an *array*. An array

of places and transitions can be accessed as an ordered set in **rga**. In the loop, the number of tokens in the place is added to n. Every philosopher which is eating will have a token in the corresponding *philosopher_eating[i]* place. The *true* expression ensures that all the elements of the set are traversed (forall stops at the first false value). Finally the value of the function is the value of the last expression: *n*.

To test this function it is possible to ask for the number of philosophers eating in the initial state:

```
eating(#0)
```

Once this function is defined it is possible to ask if it is true that the number of philosophers eating in any state is always less than 2 (for 5 philosophers). This can be determined by typing:

```
forall s in S [ eating(s) <= 2 ]
```

The answer should be *true*. Temporal logic operators can be used to test the *fairness* of the model by asking if it is true that every philosopher eventually eats. This can be asked as follows:

```
forall p in philosopher_eating [ inev(#0, p(C) = 1, true) ]
```

This expression returns true if for every place indicating a philosopher eating, a state will inevitably be reached from the initial state in which that place has a token. Of course, in this model this is not the case. Exiting **rga** is accomplished by typing <**CTRL-D**> (Control-d).

Rga is the most powerful and innovative tool in P-NUT. The reader is encouraged to experiment further with it and to scrutinize the manual to get an idea of **rga**'s power.

7 Simulating a Petri Net

The P-NUT simulator is a simple simulation engine which "pushes" tokens around a Timed Petri Net. The input to the simulator is a Petri Net and a few simulation commands. The output of the simulator is one or more execution traces. The execution traces are not intended for the user and must be processed by other tools before they are ready for human eyes. However, because the simulator outputs directly to standard output, the user will see the execution traces unless output is redirected. In the examples below several ways of redirecting output are shown. The example to be used will be a 2-processor multiprocessor example. The 2-processor model can be created by editing the *mp.net* file and changing the assignment to *n* (to 2). The net can then be created by typing:

```
tpp mp.net | transl > mp2.pn
```

The simulator is invoked by typing:

```
simulator
```

Once in the simulator, output should be redirected to a file. This is accomplished by typing:

```
> mp2.trace
```

This command will cause output to be stored in file *mp2.trace*. Next, a net should be read into the simulator. This can be accomplished by typing:

```
< mp2.pn
```

The simulator now expects simulation commands. Typically a breakpoint is set and the simulation is started.

```
stop when clock 300
run
```

The simulator stops when the simulation time reaches 300. Another run can be initiated (resulting in a second execution trace).

```
stop when clock 600
run
```

The simulator can be terminated by typing:

```
exit
```

The simulation results can now be examined using the **stat** and **printstat** tools. **Stat** analyzes the trace and produces output which is compatible with **tbl** and **troff**. **Printstat** simply makes sure that these tools are invoked correctly. To generate the performance report for the simulation above, type:

```
stat < mp2.trace | printstat -q
```

The output is a report indicating (among other things) the average number of tokens in every place. This data can be used to derive utilization measures. For example the average number of tokens in place *bus_busy* indicates the utilization of the shared bus. Since two traces were produced, the output of **stat** includes confidence intervals (using batch means). The -q option of printstat generates a

"quick" version of the report to the terminal using **nroff**. The -t option of printstat generates output to a Laser Printer using **troff**.

If a shorter report is desired it is possible to **filter** the simulation and obtain a report only about places and transitions of interest. The output of the **filter** program is a valid execution trace and could be stored in a file for later processing. The command below pipes that trace directly to the **stat** program in order to obtain the performance statistics report. The command below request that only data relevant to places bus_busy and bus_free (the places which model the two states of the bus) be retained.

```
filter bus_busy bus_free < mp2.trace | stat | prinstat -q
```

The resulting report should be consistent with the report obtained from the complete trace.

It is possible to avoid storing the complete simulation trace by filtering the execution traces directly from the simulator. This is accomplished from inside the **simulator**.

```
simulator
| "tee mp2.trace | filter bus_free bus_busy | stat > mp2.stat"
< mp2.pn
stop when clock 500
run
exit
```

This sequence of commands will cause the output of the **simulator** to be stored in file *mp2.trace*, filtered and then piped directly to the **stat** program. When the simulator exits, file *mp2.stat* should contain the performance statistics. The reader should produce a report by typing:

```
printstat -q mp2.stat
```

8 Analyzing Simulation Results

A big drawback to most existing simulation tools is the inability to determine if the observed behavior of the model is consistent with higher level specifications of the system. This drawback is primarily the result of the fact that simulation models and tools are typically intended to provide performance data. Debugging simulation models is very difficult. P-NUT addresses this concern by allowing the

user to convert an execution trace produced by the simulator into a partial reachability graph which can be analyzed using **rga**. A user can therefore determine if the behavior of the model (as observed during a particular simulation experiment) is consistent with their expectations.

As was discussed earlier, there are two ways of converting a simulation trace into a reachability graph. The first way recognizes that a particular state is usually visited many times during a single simulation experiment. All identical states can then be grouped together, thereby generating a graph where each state is unique. As each state is revisited during a simulation, additional arcs are added between states. The resulting graph (usually) has few states and a lot of arcs. The number of arcs can be reduced by simply grouping arcs between each pair of states. Information about individual arcs is lost in this process (e.g. the events which caused the state transition) although some information can be retained (e.g. the average time required for each state transition). This approach to analyzing a simulation trace is attractive in that it produces a small compact graph which can be analyzed efficiently. The drawback is that a good deal of information is lost in the process. In addition to the information lost in grouping arcs, information is also lost about sequences of states. The resulting graph contains state sequences which did not occur in the real simulation. This fact must be remembered in analyzing the resulting graph.

The second way of viewing a simulation trace as a reachability graph is to retain all information and to simply convert the form of the trace to be compatible with tools which expect reachability graphs. In this approach no information is gained or lost. The form is simply changed. The resulting graph is not a graph at all. It is a linear sequence of states. Identical states (markings of the net) can appear many times in a sequence. Changing the form of the trace does buys us the ability to analyze the trace using RGA. Results of such analyses are quite accurate since all information about a simulation is retained. The major drawback is that space limitations preclude the analysis of very long traces. This problem can be eleviated somewhat by filtering traces before converting them into reachability graph form.

Two tools have been added to P-NUT to support these two conversion processes. **Convert** is a tool which supports the first conversion process. It groups together identical states and groups arcs between states by simply averaging their timing delays and discarding all other information. To excercise this tool type and print the result:

```
convert < mp2.trace > mp2.simrg
rgp mp2.simrg | more
```

The resulting reachability graph is in file *mp2.simrg* and can be analyzed by

typing:

```
rga mp2.simrg
```

The number of distinct states visited can be obtained by typing:

```
card(S)
```

To test if it is true that the shared bus is either busy or free (to find errors in the model) the user can type:

```
forall s in S [ bus_busy(s) | bus_free(s) ]
```

To determine if a particular path in the model was traversed during a particular simulation experiment it is possible to ask if particular states were ever reached. For example, typing the following commands will determine if there was ever a case where a processor was ready to access the bus and couldn't because the bus was busy. The first command constructs the set of places which indicate a processor which is ready to use the bus. The second command detemines if a state exists where the bus is busy and one of the selected places holds a token.

```
ready_to_use_bus := union(remote_read, remote_write)
exists s in S [bus_busy(s)&exists p in ready_to_use_bus[p(s)>0]]
```

An additional "feature" of the **convert** program is that states in which the system spends no time are ignored. Zero-time state transitions can hide intermediate states which may be of interest. For example, a situation where a bus is released by one processor and then instantly grabbed by another appears in the graph as a single transition where the bus never becomes free. As a result, if the user asks if it is the case that every allocation of the bus inevitably leads to a release of the bus, the answer will be no. This problem can be demonstrated by asking the question:

```
forall s in {s' in S | bus_busy(s')} [ inev(s, bus_free(C), true) ]
```

Future releases of P-NUT will allow the user the option to preserve information about zero-time transitions.

The second view of reachability graphs is supported by the **lconvert** (Linear Convert) program. It produces a graph which can also be analyzed by RGA. To build and print such a graph type:

```
lconvert < mp2.trace > mp2.lrg
rgp mp2.lrg | more
```

The reader may want to experiment with using **rga** on the output of **lconvert**. The output of **lconvert** can also be used to drive the **tracertool** program which displays simulation results graphically.

9 On-line manuals

P-NUT has a simple on-line help facility. To obtain information about a particular tool the user can type:

```
pnutman <tool-name>
```

The user can also obtain a complete user manual by typing

```
pnutman "*"
```

Hardcopy versions of the manual can be obtained by typing

```
pnutman -t <tool-name>
```

In order for this command to work, it is necessary to edit the C-shell script located in *pnut/bin/pnutman*. This shell script contains the appropriate troff commands to produce hardcopy output. The troff commands are site-specific and should be tailored (at UCI the troff command is **itroff**). Appendix A of this report contains a complete user manual.

10 Conclusion

This document provides a brief and incomplete tour of the set of tools available in P-NUT. Release 2.2 of P-NUT is an early release which supports limited types of Petri-Net models. Future releases will allow more flexible models of time and will also support "interpreted" nets. Users of P-NUT are encouraged to provide the P-NUT group with feedback. The group is eager to fix any problems although, as stated earlier, no support can be guaranteed in a University environment.

References

[Age79] T. Agerwala. Putting Petri nets to work. *IEEE Computer*, 85–94, December 1979.

[Dia82] M. Diaz. Modelling and analysis of communication and cooperation protocols using Petri net based models. *Computer Networks*, 6:419–441, December 1982.

[MBC84] M.A. Marsan, G. Balbo, and G. Conte. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2(2):93–122, 1984.

[MC86] M.A. Marsan and G. Chiola. On Petri nets with deterministic and exponential transition firing times. In *Proccedings of the Seventh European Workshop on Application and Theroy of Petrn Nets*, Oxford, England, June 1986.

[Mor84] E.T. Morgan. *RGA Users Manual.* Technical Report 243, Information and Computer Science Dept., University of California, Irvine, December 1984.

[MR85] E. Timothy Morgan and Rami R. Razouk. Computer-aided analysis of concurrent systems. In *Proccedings of the 5th International Workshop on Protocol Specification Verification and Testing*, Toulouse, France, June 1985.

[Pet81] J. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, N.J., 1981.

[RH85] R.R. Razouk and D.S. Hirschberg. *Tools for Efficient Analysis of Concurrent Software Systems.* Technical Report 85-15, Information and Computer Science Dept., University of California, Irvine, June 1985.

[RP84] R.R. Razouk and C. Phelps. Performance analysis using timed Petri nets. In Y. Yemini and S. Yemini, editors, *Protocol Specification, Testing, and Verification IV*, North-Holland Pub. Co., 1984.

[WPS86] D.L. Woo, C.V. Phelps, and R.D. Sidwell. Timed Petri net probability semantics. In *Proccedings of the Seventh European Workshop on Application and Theroy of Petrn Nets*, pages 131–149, Oxford, England, June 1986.

NAME

convert – convert an execution trace (filtered or unfiltered) into a reachability graph (a probabilistic finite state machine).

SYNOPSIS

convert [file]

DESCRIPTION

Convert transforms an execution trace (currently produced by the simulator) into a probabilistic finite state machine and outputs it to standard output in a form consistent with standard PNUT reachability graphs. The resulting graph can be printed using **rgp** and examined using **rga.**

If the input execution trace is unfiltered the resulting reachability graph will be a partial reachability graph of the original net. If the input execution trace has been filtered (using **filter**) the resulting reachability graph is NOT a graph for a valid Petri Net. Only activity in filtered places and transitions is retained.

In constructing the reachability graph, times associated with arcs are calculated by averaging the delays which occur in the execution trace. No additional information such as probability distribution or standard deviation is currently retained.

Since execution traces (unlike reachability graphs) do not include information about future actions, it is not possible for the reachability graph produced by **convert** to contain remaining firing time and remaining enabling time information. The resulting reachability graph does accurately describe the number of firings of a transition in a particular state.

SEE ALSO

filter, simulator, rgp, rga

NAME

filter – filter execution trace and produce a smaller execution trace which includes only activities related to specified places and transitions.

SYNOPSIS

filter [name ...]

DESCRIPTION

Filter transforms an execution trace into a shorter execution trace which represents only the activities related to user-specified places and transitions. The places and transitions whose activities are retained are specified at the command line as a list of names. Place and transition names can be mixed and specified in any order. Incorrect names are ignored. Names of place and transition arrays must be listed one by one (in this version).

Input to filter is expected from standard input. Input redirection must be used if execution traces are stored in a file.

The output of filter is an execution trace for a PARTIAL Petri Net. When the results are analyzed (after using **convert**) the user should be aware of the fact that the analysis has been applied to a partial net.

SEE ALSO

convert, simulator, stat

NAME

pnc – Petri Net Compiler

SYNOPSIS

pnc [*flags*] [*Petri_net* [*Ada_program*]]

DESCRIPTION

Pnc reads a Petri net (in the same form as read by *transl(1)*). If no *Petri_net* is given on the command line, *stdin* is read. If no output file is specified, the Ada program is written to *stdout*.

The following flags are recognized:

– **u** *package*
Use the specified *package*.

– **d** Generate code which detects deadlocks and halts. Normally the generated programs are expected to execute indefinitely.

All output is written to *stderr*.

SEE ALSO

tpp(1), rgb(1), trgb(1), transl(1), pnl(1)

KNOWN BUGS

Pnc does not handle inhibitor arcs correctly.

NAME

pnl – Petri Net Lint

SYNOPSIS

pnl [*Petri_net*]

DESCRIPTION

Pnl reads a Petri net (in the same form as read by *transl(1)*). If no *Petri_net* is given on the command line, *stdin* is read. No flags are recognized.

Pnl checks the input for syntax errors by parsing it as *transl(1)* does. If there are no syntax errors, *pnl* will perform some additional checks and issue warnings if possible errors are found. Currently, these checks include (1) The use of a place on the left of some transition, but not on the right of any transition or in the initialization, and (2) The use of a place on the right of some transition and/or in the initialization, but not on the left of any transition.

Finally, a summary is printed giving the number of transitions and places in the net, and the number of warnings issued by *pnl*. All output is written to *stderr*.

SEE ALSO

tpp(1), rgb(1), trgb(1), transl(1)

NAME

printstat – print performance statistics generated by **stat.**

SYNOPSIS

printstat [**-qntc**] [*file*]

DESCRIPTION

PRINTSTAT accepts the output of **stat** and formats it using **tbl and nroff/troff.** It allows the user to control the formatting through the use of some options.

– q option produces a quick printout to the CRT (doesn't use ms macros). The

– n option produces nroff output (this is the default). The

– t option produces troff output and sends it to the printer. The

– c option produces troff output and pipes it through tseetool.

The *file* specified should contain output generated by **stat.** If no file is given, **PRINTSTAT** reads from stdin.

KNOWN BUGS

WARNING: Some versions of **tbl** on the SUN's have a bug which will cause the statistics to be improperly formatted. To get around the bug it is necessary to manually edit the output of **stat** to specify that number columns (n in tbl) are left justified (l in tbl).

SEE ALSO

filter(1), simulator(1), stat(1)

NAME

rga – interactively analyze a reachability graph

SYNOPSIS

rga [– s] [*files...*]

DESCRIPTION

RGA reads a Petri net and its reachability graph. It then allows the user to do computer-assisted interactive analysis of the graph.

The – s flag indicates that **RGA** should print dots as it reads the graph, for processing large graphs. The *files* specified should contain first the Petri net whose reachability graph is being analyzed, followed by the reachability graph itself, followed by **RGA** commands. Normally, the Petri net and its reachability graph will be contained in the same input file, produced by *rgb*, *trgb*, or *convert*. The file /dev/tty is automatically appended to any list of files, including the empty list. If no files are given, **RGA** first reads from stdin.

When reading from a terminal, **RGA** prompts for input with a > character. Input at that point may include either **RGA** expressions or commands, as described in *The RGA Users Manual*.

SEE ALSO

transl(1), tpp(1), rgb(1), trgb(1), convert(1)

NAME

rgb – build a reachability graph

SYNOPSIS

rgb – [**sb**[**127**]] [*file*]

DESCRIPTION

RGB reads a Petri net and builds its reachability graph. It allows the user to specify some simple information about the Petri net which can be used to build the graph efficiently.

The – **s** flag indicates that the net is known to be safe. The – **b** flag indicates that the net is bounded, but that the bound is not known or is higher that 127. The – **b127** flag indicates that the net is bounded at 127 or less. The reachability graphs of such nets require less main storage during building. The *file* specified should contain the Petri net whose reachability graph is being built. Normally, the Petri net is produced by *transl*. If no file is given, **RGB** reads from stdin.

SEE ALSO

transl(1), rga(1), rgp(1)

NAME

rgf – filter reachability graphs

SYNOPSIS

rgf [*files...*] [**–e***boolean expression*] [*states...*]

DESCRIPTION

RGF reads a Petri net and its reachability graph. It echos the Petri net on its standard output. A partial reachability graph containing selected states of the input reachability graph and the arcs which connect them is then output.

The states may be selected in either of two ways. First, they may be specified on the command line by number. Any argument which begins with a digit is assumed to be a state number specification. All states thus listed on the command line will be included in the output reachability graph.

The second method of specifying states is by giving a *boolean expression.* All states for which this expression evaluates to **true** will also be included in the output. Only the arcs in the original reachability graph which begin and end on a state in the partial reachability graph will be included in the output.

RGF will read the Petri net and its reachability graph from its standard input if no files are specified on the command line.

SEE ALSO

transl(1), tpp(1), rgb(1), trgb(1), convert(1), rga(1)

NAME

rgp – print a reachability graph

SYNOPSIS

rgp [**options**] [*file*]

DESCRIPTION

RGP reads a Petri net and its reachability graph and prints a tree representation of the reachability graph. It allows the user to control the display through the use of some options.

-b flag causes the graph to be printed backwards (traverse predecessor links starting from the specified state).

-s< NUMBER>
flag causes the printing to start at state numbered < NUMBER> .

-d< NUMBER>
flag causes only states which are at distance < NUMBER> (or less) from the starting state to be printed.

-w< NUMBER>
flag indicates that the display should be tailored to a maximum line width of < NUMBER> . The *file* specified should contain the Petri net and its reachability graph (as produced by convert, rgb or trgb). If no file is given, **RGP** reads from stdin.

SEE ALSO

convert(1), rgb(1), trgb(1)

NAME

simulator – simulates execution of a (timed) petri net

SYNOPSIS

simulator [-aceprtvx file_names]

DESCRIPTION

Files given on the command line are evaluated from left to right with 'stdin' always being last. Switches may appear anywhere and are always set before any file is read. Input files may contain either simulator commands or a Petri Net definition. Switches and file I/O may also be set from within the simulator.

There are two input levels. The first is '(stdin)' and the other is '(simulator)'. At '(stdin)' the simulator has not yet received a Petri Net definition. Most simulator commands are executable from both levels.

The simulator's output is meant for use with statistical packages for interpretation. A typical session might be: set environment (for discussion of this please see manual), redirect output, input a net from a file, set a halting point, run, examine statistics. Since the input interface allows commands from a file, a script file may be used to run the simulator in background, redirecting the simulator command I/O to a log file at invocation (by redirecting stdout), and redirecting simulator/statistical output (from within the simulator) to a file with the simulator pipe (|) command.

Switches:

- **-a** (not yet implemented -- will allow evaluation of actions)
- **-c** TRUE outputs the clock value, FALSE outputs the change in clock value
- **-e** TRUE echos the Petri Net canonical form as it is input, FALSE supresses it
- **-p** TRUE uses the probabilities (frequencies) assigned to transitions when choosing a set of transition firings, FALSE uses a frequency of 1 for all transitions
- **-r** (not yet implemented -- will allow evaluation of predicates)
- **-t** (not yet implemented -- will allow execution of non-timed Petri Nets)
- **-v** TRUE causes multiply enabled transitions to compete with each other for firing, thereby increasing a multiply enabled transition's probability for firing, FALSE allows only one enabling of a transition to compete for firing in the probability calculations
- **-x** TRUE causes enabling times and firing times for all transitions to be exponentially distributed with the specified enabling and firing times as the mean. FALSE (the default) causes enabling times and firing times for all transitions to be constant. The simulator also provides an internal mechanism (the exp command) to make times of individual transitions exponentially distributed.

Switches toggle via the 'set -[cepvx]' command (for more info, '?set').

Help is available by typing help or ? at either prompt.

SEE ALSO

convert(1), filter(1), stat(1)

DIAGNOSTICS

Self explanatory.

BUGS

The simulator does not work correctly when all the input arcs of a transition are inhibitor arcs. There must be at least one non-inhibitor arc going into every transition.

NAME

stat – calculate performance statistics from execution trace

SYNOPSIS

stat [*file*]

DESCRIPTION

STAT reads a Petri net and a set of execution traces and produces a report on various performance related questions. If multiple execution traces are given to the stat program it will provide confidence interval measures for the various performance statistics (confidence level is fixed at 90%). Currently, execution traces are produced by the **simulator** or the **filter** programs. The output of the **simulator** (and **filter**) can be piped directly to **stat.**

The *file* specified should contain the Petri net and one or more execution traces. The report includes: 1) maximum number of concurrent tokens in a place, 2) minimum number of concurrent tokens in a place, 3) average number of concurrent tokens in a place, 4) maximum number of concurrent firings of a transition, 5) minimum number of concurrent firings of a transition, 6) average number of concurrent firings of a transition, 7) number of times a transition began and finished firing,

SEE ALSO

simulator(1), filter(1)

NAME

tpp – perform macro expansion on a Petri net specification

SYNOPSIS

tpp [**–n**] [Petri_net [Expanded_net]]

DESCRIPTION

Tpp reads a Petri net and expands certain macro constructs, writing an Expanded net suitable for input to *transl(1)*. Other than the special macro and looping constructs recognized by *tpp*, the input should be in the form expected by *transl*, and the input will be copied to the output file.

If no Petri_net is given on the command line, stdin is read. Similarly, stdout is written unless an Expanded_net name is specified.

Tpp's input is a superset of that of the *transl* program. It is divided into two parts, function declarations followed by a section which will generate the array, transition, and initial state descriptions for *transl*. The function declarations section is identical to that in *transl*.

Tpp recognizes a special command **for** which allows identifiers to be set and/or looped through a range of expression values. For each iteration of the loop, array, transition, and initial marking descriptions are generated, evaluating expressions as integers. These expressions can occur where parentheses are allowed in *transl's* input and within square brackets immediately following a place name.

The **for** statement takes one of three forms:

for *id= expr* {
for *id= expr* to *expr* {
for *id= expr* to *expr* by *expr*

After a line of one of these forms, one or more array, transition, or initial marking lines may be given in arbitrary order, followed by a line containing a single "}" character. When *tpp* generates its output, it produces all array declarations first, then all transitions, and finally all initial marking declarations, so that its output is legal input for *transl*.

Immediately following a place identifier, and preceeding any token count in parentheses, an expression may be given surrounded by square brackets. On each iteration of the enclosing loops, the expression is evaluated. These expressions, the ones enclosed in parentheses, and those used in the **for** statements themselves, can include any of the arithmetic operations allowed in the **RGA** language. They should evaluate to an integer value. Multiple place names separated by expressions in square brackets may be specified, thus allowing multiple subscripts.

The **–n** flag indicates that unnamed transitions should be numbered in the output with names of the form $*n*. The numbered listing will facilitate graph analysis with **RGA.** *Tpp* can also function as a Petri net pretty-printer, since it will accept any input acceptable by *transl*, and it will always produce its output in a standard format.

SEE ALSO

transl(1)

NAME

transl – translate a Petri net to canonical form

SYNOPSIS

transl [**–e***default_enable_time*] [**–f***default_firing_time*] [**–p***default_probability*] [*Petri_net* [*Translated_net*]]

DESCRIPTION

Transl reads a Petri net and translates it into an easily machine-processable canonical form which may be read by other tools, including *rgb* and *trgb*.

If no *Petri_net* is given on the command line, stdin is read. Similarly, stdout is written unless a *Translated_net* name is specified. The **–e**, **–f**, and **–p** flags may be used to override the default enabling and firing times and transition firing probabilities. They are immediately followed by the floating point time. The defaults are otherwise 0.0 and 1.0, respectively.

The input consists of four sections. The first declares functions, the second declares arrays of places and/or transitions, the third declares the transitions which make up the Petri net, and the fourth defines the marking of the initial state of the net. Arrays of places are declared using the **array** keyword, which is followed by one or more place array names and sizes separated by commas. The sizes are given in parentheses. Thus

array philosophers_thinking(3)

declares an array of places called **philosophers_thinking0**, **philosophers_thinking1**, and **philosophers_thinking2**. Arrays of transition are similarly declared using the **tarray** keyword.

If any RGA-type functions are to be defined, they are declared following the array declarations using the same syntax as in the RGA interpreter. Function definitions are preceded and succeeded by %% (double percent signs).

The transitions section consists of a one line description of each transition which contains the input and output places as well as optional timing information, actions, and preconditions, and possibly a transition name. It has the following format:

[*precondition*] :*name*: *input_places* -> *timing output_places*

where only the -> and at least one input place are required. The *input_places* and *output_places* are the names of the input and output places for the transition, separated by commas. Transition and place names consist of upper or lower case letters, digits, or underlines, and begin with a letter. Multiple arcs are specified by placing the number of arcs in parentheses after the place name. Similarly, inhibitor arcs are specified by giving the number of arcs as zero. The actions and preconditions are boolean expressions in the RGA language. In the *actions*, local variables may be specified within square brackets; e.g., {[i] i:= a+ 1}.

Timed Petri Nets are described by including at least one transition with the *timing* option. This option has the following format: (*enable_time*, *firing_time*, *probability*) where each of the items and trailing commas are optional. Each of these values may be specified to be arbitrary expressions; usually they will be integer or floating point numbers.

These lines which describe the transitions are followed by one or more lines specifying the initial marking which have the following format: < *place*, *place*, ...> where each of the listed places will initially contain a token. To put more than one token in a place, put the number of tokens it is to contain in parentheses after the place name: listing the place twice will not work.

Comments may be included by enclosing them between /* and */. Comments are not restricted to one line, but may be as long as needed. Lines may be broken after any comma if needed for readability, and spaces are ignored except to separate identifiers.

Transitions are not required to be named or have output places. If timing information is supplied for any transition in the net, then default values apply to all transitions for which it is otherwise not specified. The default enable time is 0, and the default firing time is 1. The default probability depends upon the probabilities specified for the other transitions in the same conflict

set; it will be the value which puts the sum of the probabilities as close as possible to 100.

SEE ALSO

tpp(1), rgb(1), trgb(1)

NAME

trgb – builds the timed reachability graph for a timed Petri net

SYNOPSIS

trgb [**– v**] [*Petri_Net* [*Reachability_Graph*]]

DESCRIPTION

Trgb reads a timed Petri net and builds its timed reachability graph. The input is read from the standard input if not specified on the command line, and it is assumed to be a canonical-form timed Petri net description. Output is to standard output if no output file is specified, and it consists of a timed reachability graph in canonical form.

The **– v** option causes states to be printed on stderr symbolically as they are discovered.

Typing the interrupt character causes the current state of the graph to be dumped (creating a partial reachability graph). This feature may be used in conjunction with the **– v** option when debugging a Petri net whose reachability graph is infinite.

BUGS

Trgb does not work correctly if ALL of the input arcs of a transition are inhibitor arcs.

AUTHOR

Tim Morgan

SEE ALSO

rgb, rga, simulator, rgp, convert