Information relating to the Perl files intended to generate a LHPN file from SPICE output

Kevin Jones - 11 August 2006

1 Introduction

In order to provide a Spice front-end to the ATACS verification tool, the state space must first be discretized in order to provide a linear model. The data is first sorted into bins. The size of these bins are optimized to reduce rate discrepancies between the data points within each bin. Each variable is divided into a certain number of divisions. The combination of these divisions forms the bins. (see Fig. 1 for an example of the division lines and data points). The encoding of these data, according to the bins in which they are placed, are then transformed into places on a labeled hybrid Petri net (LHPN). Each place name is a concatenation of the bins in which each variable lay. Ergo, each place name consists of one digit for each variable in the LHPN. Transitions between these places are controlled by rates and enabling conditions, each variable corresponding to a value from the original circuit. Transitions are named using a concatenation of the places between which the transition exists. Additionally, each place is given a Boolean encoding corresponding to a binary representation of the place name. Each transition also then corresponds to an assignment of Boolean variables to match the places between which the transition moves. See Fig. 2 for an example of a simple LHPN. To avoid deadlocks (see section 5.6 for details) and to simulate spans of rates, extra places and transitions are added, identical except for the use of a different rate.

As an added ability, paramsweep.pl generates multiple Spice files with varying parameters. This uses syntax that will run in Ngspice, but with added commands specific to this tool. See the section on paramsweep.pl for details as to the syntax of these added commands.

2 Simple Usage Information (see Secs. 4, 5 for details on the usage of Ngspice)

To generate multiple Spice runs, use paramsweep.pl

Usage: ./paramsweep.pl <netlist>.cir

(see paramsweep.pl section for details on special netlist syntax)

Spice output in ASCII format (using Ngsconvert) -> convert.pl (or bigconvert.pl for multiple Spice files)

Usage: ./convert.pl <input filename> <filename>.dat OR

./bigconvert.pl <filename>.dat <input filenames>

THEN <filename>.dat -> goodbinsort.pl

Usage: ./goodbinsort <filename>.dat <number of bins>

(if different number of bins for different variables, just type multiple numbers at the end of the command line, but no given variable should have greater than nine bins)

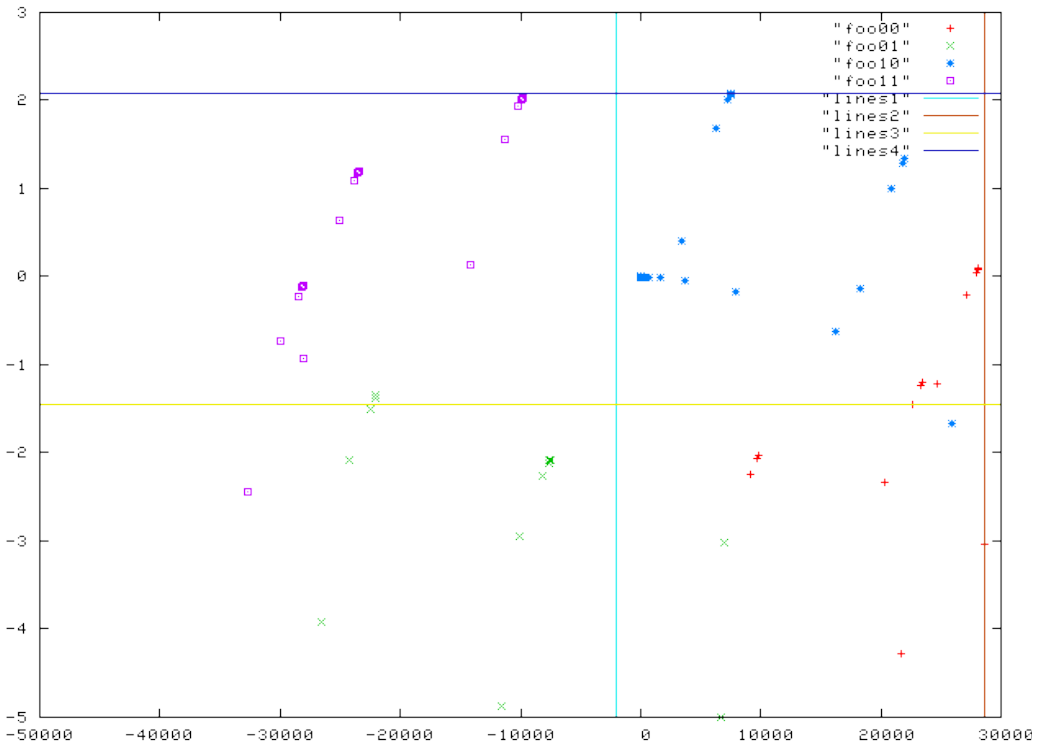
THEN <filename>.sort -> graphIt.pl

Usage: ./graphIt.pl -esdb <filename>.sort

(alternative flag options available, type ./graphIt.pl -help for more information; the given command generates the LHPN described in Sec. 3)

THEN <filename>.g -> postproc.pl
Usage: ./postproc.pl (-t) <filename>.g

(a)



(b)

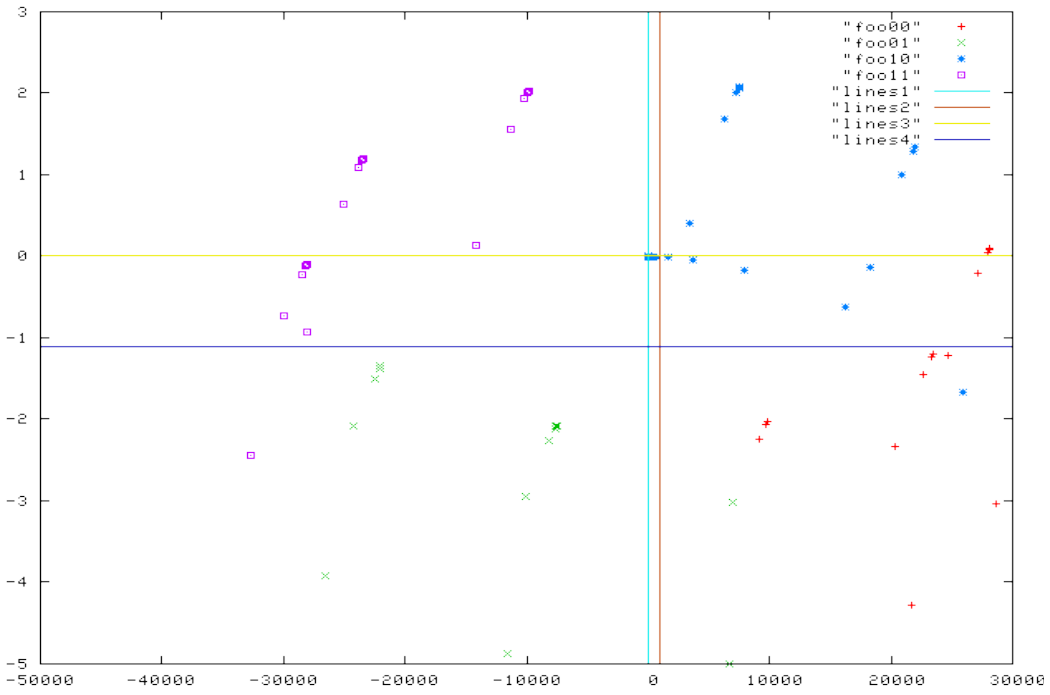


Fig.1 Sorting of Data into bins

The top figure (a) shows the data divided into equally sized bins. The bottom figure (b) shows the same data divided into bins that have been improved according to the algorithm mentioned. Each color of dot represents a different rate encoding.

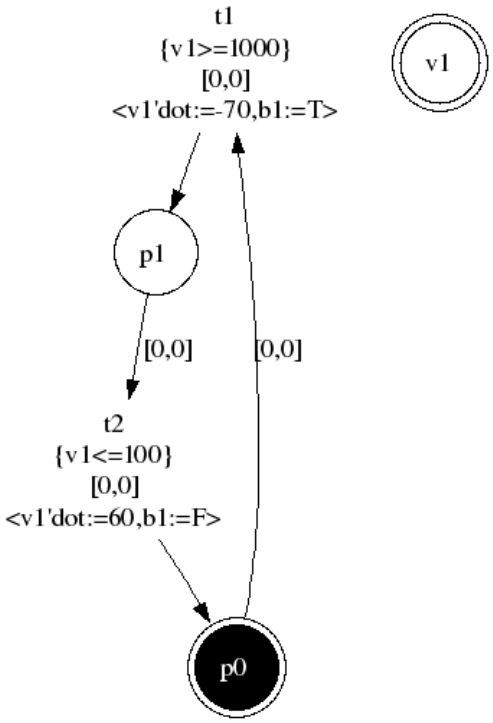


Fig. 2 A Simple LHPN

3 Expanded Usage Information (see Secs. 4, 5 for details on the usage of Ngspice)

The examples generated were created using Ngspice rework-15, though rework-17 also fits the output format. Other versions of Ngspice are currently untested. The output needs to be in ASCII format. If it is not, simply run the binary rawfile through Ngsconvert, which will convert the binary file into ASCII format (see Ngspice documentation on how to use Sconvert).

The ASCII output file can then be run through convert.pl, which will turn the data into a table format, each column being a variable (the first being time) and each row being a time point. For multiple simulation runs, run them through bigconvert.pl.

This script concatenates the multiple simulation runs into a single table of the same format as convert. The output from both of these can be run through the rest of the programs in the same way, and they should both end with the extension .dat.

Transient data from any simulator can be run through the remaining tools provided the format is correct. The data should be in tabular format using floating point variables in scientific notation. Each column should be delimited by whitespace (anything but a newline will work). Each row should be delimited by a newline. The data should be in tabular format, each variable having its own column and each time point having its own row. The first column should be time. Fig. 3 shows an example of the table format in which the .dat files are organized. The first column is

time, the second is a voltage, and the third column is a current, and the table shows the first seven time points of the transient simulation. If desired, a heading may be added to the data table. Note, however, that the first value in the time column must be zero.

0.000000e+00	0.000000e+00	0.000000e+00
3.000000e-08	1.129617e-01	-1.694425e-08
3.168051e-08	1.196439e-01	-1.794956e-08
3.504152e-08	1.337171e-01	-2.007843e-08
4.176354e-08	1.661159e-01	-2.511714e-08
5.520758e-08	2.479173e-01	-3.903284e-08
8.209567e-08	4.794892e-01	-8.792927e-08

Fig. 3 Data table format

Once the data have been converted into a table format, they are then ready to be sorted into bins, using goodbinsort.pl. This program both generates the bins and encodes the data as a concatenation of the bin numbers. The output of goodbinsort.pl, which is the original data table with the encoding appended onto each line (the file will end with a .sort extension).

This file can then be used to generate a LHPN using graphIt.pl. Once the LHPN has been generated, postproc.pl provides additional transitions and places to help avoid deadlocks (see section on postproc.pl for more details). It also allows for a simulated nondeterminism in a state space analysis in which spans of rates are not allowed. Generally, postproc.pl allows for the simulation of behaviors that occur when spans of rates are allowed, but it removes the actual occurrence of spans of rates.

Fig. 4 shows a flow chart illustrating how the files and programs interact. Each program is shown, followed by the file type generated by that program, and in turn followed by the programs in which those files may be used.

4 Simple Ngspice Usage

To run a simulation from the command line using simulation instructions that are included in the netlist, use

./ngspice -b -r <output rawfile> <input netlist>.

The -b flag tells Ngspice to run in batch mode, meaning that the program is not opened in the terminal. The -r flag notes that the following string is the file to which the output data will be printed. Other flags and options are available. Consult Ngspice documentation for details. Otherwise, Ngspice can be run in a terminal, using commands found in the Ngspice manual (included in the download of the source code). Note: if Ngspice is opened from a terminal,

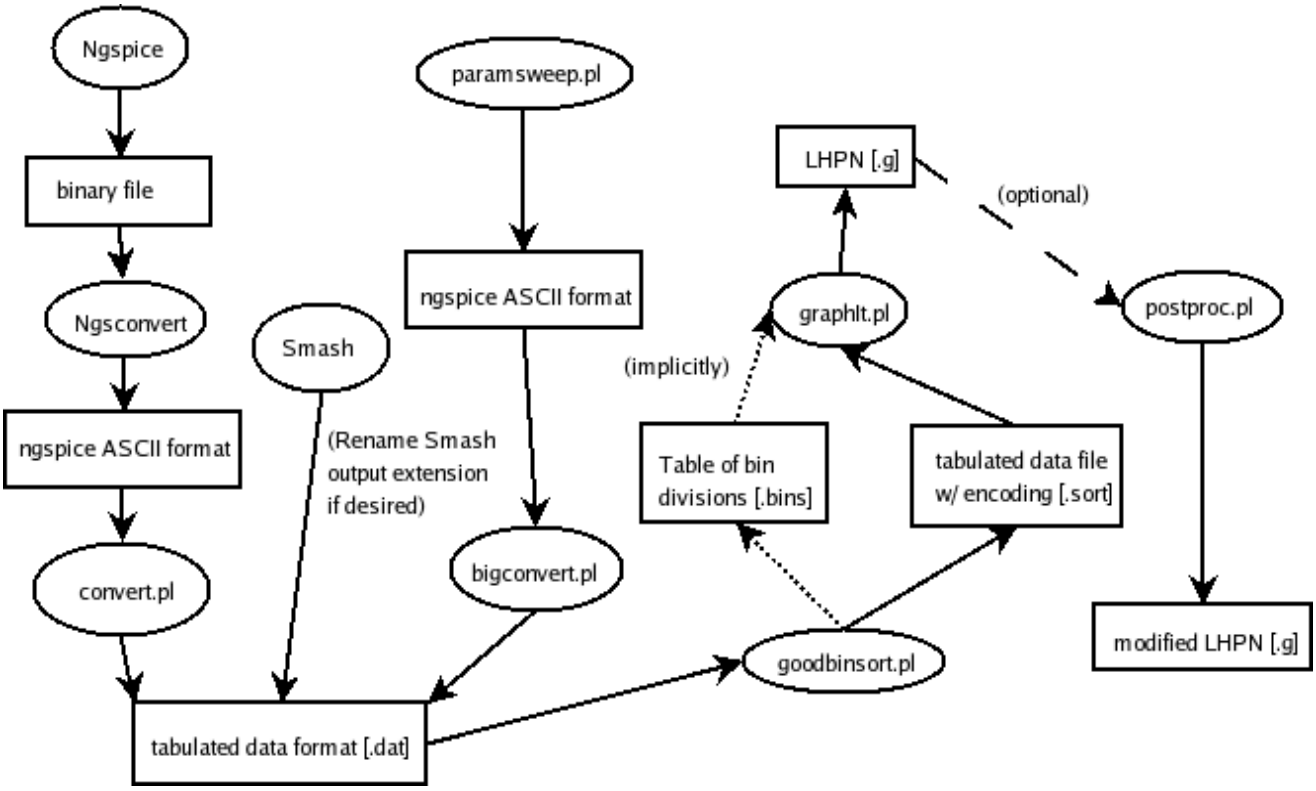


Fig. 4 Flow chart of scripts and files explained in this document
Note: goodbinsort.pl creates both the tabulated data file and a file that holds a table of the dividing lines between bins. The tabulated data file must be explicitly named as the input file for graphIt.pl, while the table of bin divisions will automatically be opened by graphIt.pl without being prompted to do so.

simulation instructions included in the netlist are irrelevant, as simulation instructions must be explicitly given.

To convert the Ngspice binary rawfile to the ASCII format that can be entered into convert.pl or bigconvert.pl, use Ngsconvert. The easy way to make the conversion is by running the program from the command line using

./ngsconvert b <rawfile> a <ASCII output file>.

The -b flag states that the original file is in the Ngspice binary format, and the -a flag states that the output file of Sconvert is to be in ASCII format. This program can also be opened from a terminal, and prompts the user with explicit and intuitive instructions when opened.

5 Ngspice Details

Ngspice is an open source circuit simulator based on previous Spice editions. A download of the simulator, as well as information concerning it, is available at http://ngspice.sourceforge.net/. The netlist syntax is similar to Spice3f5, though with notable differences. Because exhaustive documentation is not available for Ngspice, the developers refer users to Spice3f4 documentation

as a workable replacement. A link to the recommended documentation is available in the documentation section of the aforementioned website.

6 Smash Information

A free download of SMASH Seduction can be downloaded from Dolphin Integration (http://dolphin-integration.com). While this download has some limitations, it is useful for small simulations. More powerful versions of Smash can be purchased from Dolphin Integration. Smash usage is more complex than Ngspice, but is well documented in the information provided by Dolphin Integration. To create a tabulated ASCII format consistent with goodbinsort.pl, add a .OPTION DAT line to the pattern file (see SMASH User Manual 2.6.1.2 p.29). The output file created with this option (having a .ref.tran.dat suffix) may be directly entered into goodbinsort.pl. To run simulations, Smash requires that each netlist be placed in a file with a .nsx extension and a declaration of the language used on the first line. Additionally, each netlist file must be connected to a pattern file in the same directory with a .pat extension. The documentation for Smash does not provide detailed information on the syntax of pattern files. It may be beneficial for one learning Smash to simply copy a previously existing pattern file and alter it to fit the associated netlist.

7 Program Semantics

7.1 paramsweep.pl

This program generates multiple Spice simulation runs with different parameter values. These parameter values are generated using specifications given within the Spice deck. These specifications are preceded by an asterisk (*) so as to be commented out of the Spice code, thus allowing for normal use of the netlist. There are three types of parametric values that can be generated: uniformly distributed random values, Gaussian distributed random values, and uniformly distributed values. Each of these parameters makes use of the .define ability of Ngspice. The .define option is used as it normally is, with the parameter name and default value in its usual place. Following the usual .define parameters, further commands are given following the asterisk. Each option for generating different parameter values has its own syntax, which is explained in more detail later on.

7.2 Syntax for netlist:

7.2.1 Number of iterations

Write on one specific line, for *n* iterations

*<*n*> iterations

7.2.2 Uniformly distributed values

Include the additional specifications within a .define line, as with normal and sweep

.define <parameter name> <default value> *uniform(<min>,<max>)

This option generates a random value uniformly distributed between <min> and <max>. It generates a new value for each simulation run.

7.2.3 Normally distributed values

Include the additional specifications within a .define line, as with uniform and sweep

.define <parameter name> <default value> *normal(<mean>,<std dev>)

This option generates a normally distributed random values with the given mean and standard deviation. It generates a new value for each simulation run.

7.2.4 Parameter sweep

Include the additional specifications within a .define line, as with normal and uniform

.define <parameter name> <default value> *sweep(<level>,<min>,<max>,<step>)

Sweeps through values from <min> to <max>, increasing by <step> for each iteration. Nesting levels are used to determine how multiple parameters interact. For multiple parameters, if the first parameter has a nesting level greater than the second parameter, then the first parameter will remain at its first value while the second parameter sweeps through its values. The first parameter then switches to its second value while the second parameter sweeps through its values again. This continues until either the total iteration count is reached or until the top nesting level has swept entirely through its values. If two variables are in the same nesting level, then the increment together through their respective values. If all nesting levels have swept through all their values, then they begin again to sweep through their values, continuing until the iteration count has been reached. In contrast, if the sweep will not be completed in the number of iterations specified, then the user is warned of this and the stated number of iterations is completed. In this case, there will be sweep values that will not be used.

7.3 convert.pl

This program converts ASCII-style Ngspice output into a data table. The first column is the value of time for each data point, and each additional column is a voltage or current from the simulated circuit. This data table format is then used throughout the rest of the programs.

7.4 bigconvert.pl

This program works like convert.pl, but accepts multiple Spice output files. It concatenates

the data tables into one long data table, with a zero time point marking the beginning of each new simulation run.

7.5 goodbinsort.pl

This program both creates the bins into which the data will be sorted and sorts the data into those bins. The bins are optimized so that the rates of each variable within each bin are either positive or negative as much as possible. This minimizes problems caused by bins containing both positive and negative rates, in which the variable goes either up or down ad infinitum while the enabling rate leading to the next transition is in the opposite direction. The program postproc.pl also helps to remedy this problem, as is discussed in detail later. Additionally, this program supports the use of a different number of bins per each variable (Note, this program does not support more than 9 bins for any given variable because, as a practical matter, each variable is only assigned one digit). For example, a data table with two variables (time excluded) could be transformed from a state space with four bins (two bins per each variable) to one with six bins (one variable with two bins, the other with three). In the sorting, each data point is defined by which bin in which it lies for each variable. The dividing lines for each variable intersect to form bins of n-dimensions, where n is the number of variables in the data table. Graphically, this can be seen in two dimensions in Fig. 1.

Initially, each variable is divided equally into a user-specified number of bins. Each data point is then sorted into those bins and given what I will refer to as a variable encoding. Additionally, each data point is given an encoding according to whether each variable is increasing or decreasing at that point in time. This data encoding is thus a string of bits, each bit representing one of the variables. I will refer to this as a rate encoding. A score is then calculated by finding the percentage of data points exist in bins in which they share a rate encoding with the most prevalent rate encoding. At this point, the bins are ready to be adjusted in an attempt to optimize the bins according to the aforementioned algorithm.

In an attempt to optimize the size of the bins, several locations are considered, equally spaced across each bin. Each dividing line is then placed at each of these locations, and the location that yields the best score is chosen. In practical terms, this is done by moving each line to the next location, testing to see if the score has been improved, then keeping it there if the score is better. This is done for each dividing line, then repeated until each location has been considered. Thus, the best of all these positions will be the one that is kept.

Once the best position has been found, then the input data table is printed to the output file.

The value encoding is appended to the end of each line of data. The table with appended encodings is saved in a file with a .sort extension. The dividing lines between each bin are also saved in a file for use as enabling conditions. In this file, each column is a variable, and the divisions between each bin is a row in each column, with the top value being the minimum value. This file has a .bins extension.

7.6 graphIt.pl

This program generates the LHPN that represents a discretization of the simulation run. Each bin created by goodbinsort.pl is turned into a place on the LHPN, and transitions are created following the simulation run, according to how the values change through time. Ideally, this LHPN would follow the transient run in a discretized form. Initial conditions are added according to the state of the system when time equals zero.

Flags: (Note: use -esdb to create a file compatible with postproc.pl)

7.6.1 -e (Enabling conditions)

Each time a transition takes place, it is equivalent to one or more variables crossing the threshold into an adjacent bin. The enabling conditions for the transitions reflect this state of events. The enabling conditions require a variable to move above or below the threshold into that bin for the transition to take place. The enabling condition uses the place names to determine whether the variable needs to cross above or below a certain threshold, then uses <filename>.bin to determine the value that the variable needs to cross. (Note: This flag should only be used in conjunction with either -r or -s)

7.6.2 -r (Average rates)

There are a number of data points connected to any given place in the LHPN. Each of these data points has a given rate at which each variable is traveling at that point in time. When this flag is selected, the program averages the rates within each place. When the Petri net transitions into that place, each variable is assigned the average rate of that variable for the place into which the net is transitioning. This selection can lead to bugs if a place contains both positive and negative rates. For this reason, it is recommended that the -s flag and postproc.pl be used if spans of rates cannot be. (Note: if this flag or the -s flag are selected, then all other necessary lines are automatically added)

7.6.3 -s (Spanned rates)

This option works very similarly to the average rates flag discussed above. The primary difference is that rather than calculating the average rate for each place, the minimum and

maximum values are found, and are both inserted into the LHPN. This allows for a rate to be nondeterministically chosen from a span of rates.

7.6.4 -d (Delay assignments)

Originally, this option was created to be replaced by rates and enabling conditions. Rather than enabling the transition after a variable had crossed a certain threshold, the transition was designed not to fire until the time had passed at which the variable would have crossed the enabling condition. Since the rates and enabling conditions have been introduced, that has become obsolete, though it can still be done. The only use for this option currently is for postproc.pl, in which some transitions are given nondeterministic delays.

7.6.5 -b (Boolean encoding)

This option encodes each place as a string of Boolean values. Each place name is the binary representation of each of the variables' bins. Each transition is therefore connected to a series of Boolean assignments, which change the Boolean variables from the encoding for the previous place to the following place.

7.7 postproc.pl

This program is meant to replace rate spans with multiple transitions that use different rates to simulate nondeterministic behavior. The input file ought to be the output from graphIt.pl using the -esdb flags. For each transition with a span of rates, this program sets the minimum rate initially. It then creates a new place into which the LHPN may nondeterministically transition, which has the maximum rate (see Fig. 5). In this way, the variable values in the approximation are equivalent to the values in a run with actual nondeterministic rates (see Fig. 6).

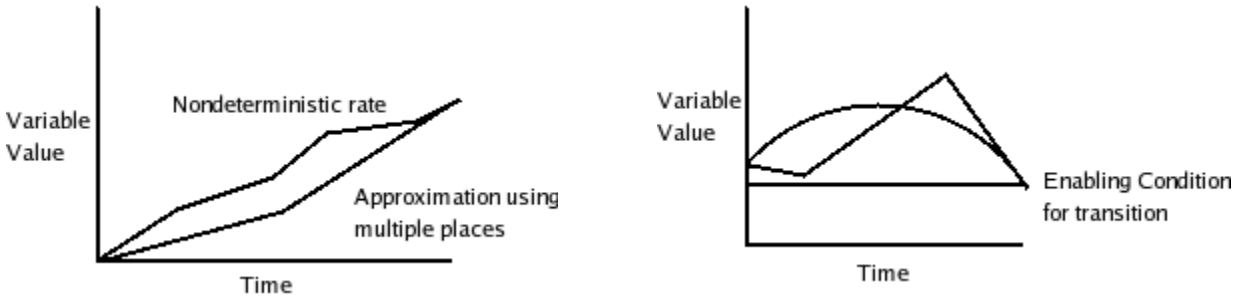


Fig .5

Using multiple places with different rates, nondeterministic rates can be approximated with a usable degree of accuracy. Notice that when the rate spans zero, allowing the graph to transition back to the minimum rate allows for proper behavior without erroneously crossing the enabling condition line.

For rates that span zero, a transition is added that returns to the original place so that the variable can increase, then decrease. This prevents the variable from dropping below an enabling condition too early.

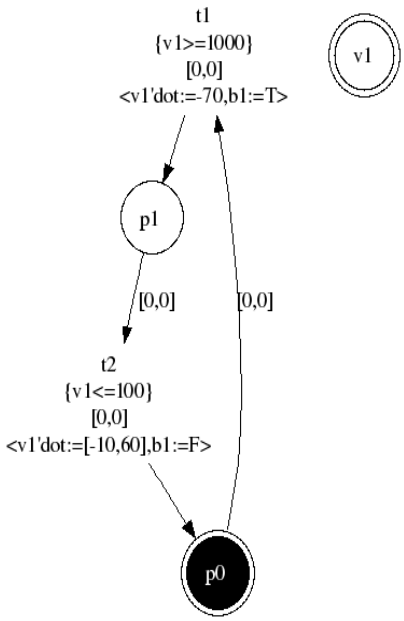
If an average rate is used rather than multiple rates that span zero, then the graph could conceivably either increase or decrease infinitely with an enabling condition in the opposite direction. Having a positive and a negative rate helps to mitigate this problem, thus helping to prevent deadlocks.

Each transition from the original place (which uses the minimum rate) is copied so that the graph can transition from the copied place (which uses the maximum rate) as if it were the original place. The place and the copied place effectively act as one place, but allowing for the approximation of spans of rates.

To separate the Boolean encodings of a place and its copy, an additional Boolean variable, called cp, is added. This variable is set to zero when the graph is in its original place and one when the graph is in the copied place.

7.7.1 -t (Adds additional transitions)

This flag adds a transition directly to the copied place created by this program. In essence, this allows the graph to move directly into the higher rate, skipping over the original place entirely. In Fig. 6, this option would create one additional transition linking p1 directly to cp0, rather than moving from p1 to p0, then on to cp0.



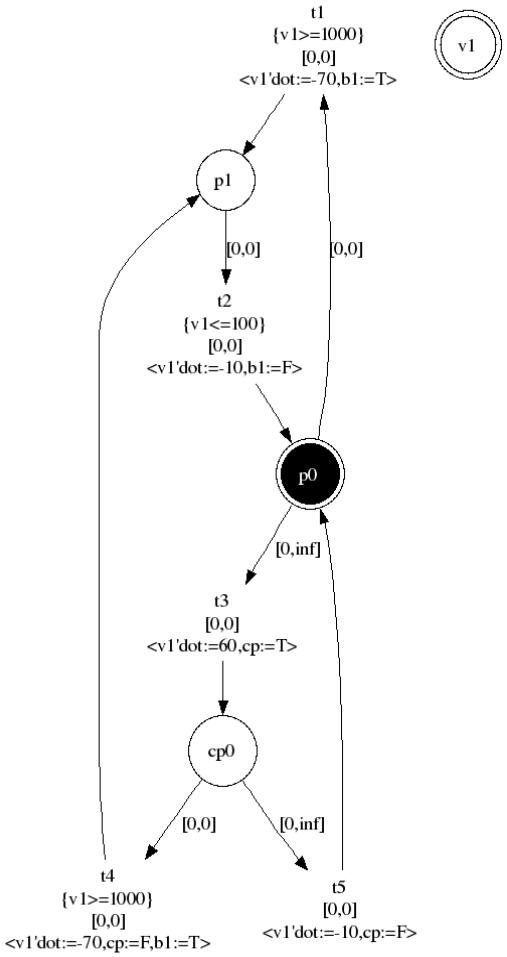


Fig. 6 LHPN with and without spanned rates

Because the rate in p0 spans zero, cp0 can either transition to p1 or back to p0, as explained in the text. Notice that p0 and cp0 are equivalent, except that cp0 has a rate of 60, while p0 has a rate of -10