

**Imperial College of Science,
Technology and Medicine
(University of London)
Department of Computing**

PIPE2: Platform Independent

Petri Net Editor

by

Nadeem Akharware MEng. MIEE

**Submitted in partial fulfilment
of the requirements for the MSc.
Degree in Computing Science of the
University of London and for the
Diploma of Imperial College of
Science, Technology and Medicine.**

September 2005

Abstract

Petri nets are a modelling formalism for describing concurrency and synchronization in distributed systems. PIPE2 is the “Platform Independent Petri Net Editor”, a Java based, open source, graphical tool for drawing and analysing Petri nets. The applications GSPN analysis module was greatly improved to handle hundreds of thousand of tangible states rather than just 1000. State space exploration algorithms and memory efficient schemes to enable this improvement are discussed along with numerical analysis methods for solving the steady state distribution. In addition to improving the GSPN analysis module, some user interface enhancements and bug fixes were made and are also outlined.

Acknowledgements

I would like to thank the following people:

- My supervisor Dr. William J. Knottenbelt for suggesting the project and advising me on it for the past 4 months.
- My family for putting up with me taking over the computer at home for the past year of the MSc.
- My fiancé Nazli for motivating me with a wedding date at the end of the project and for helping me take my mind off the project.

Contents

Abstract	iii
Acknowledgements	iv
1. Introduction	1
2. Background Theory	3
2.1. Petri Nets	3
2.2. Generalised Stochastic Petri Nets (GSPN's)	6
2.3. Markov Theory	7
2.3.1. Random Variables	7
2.3.2. Stochastic Processes	7
2.3.3. Markov Processes	8
2.3.4. Continuous-Time Markov Chains	8
2.4. An Example	10
3. State Space Exploration	13
3.1. A basic algorithm	13
3.2. Memory and Time Efficient Strategies	14
3.2.1. A Linked List	14
3.2.2. A Hash Table with Full State Information	15
3.2.3. A Probabilistic Dynamic Technique	16
3.2.4. The primary and secondary hashing functions h_1 and h_2	18
3.2.5. Vanishing State Elimination	19
4. Steady State Solution	24
4.1. Direct Methods	24
4.2. Iterative Methods	24
4.2.1. Jacobi's Method	25
4.2.2. Gauss-Siedel	25
4.3. Efficient Memory Utilisation and Computation	25
5. Results	28
5.1. State Space Exploration	28
5.1.1. A simple GSPN	28
5.1.2. A GSPN with Vanishing States	29
5.1.3. The Courier Protocol Software GSPN	31
5.2. Steady State Solution	34
5.2.1. A Simple GSPN	34
5.2.2. A GSPN with Vanishing States	35
5.2.3. The Courier Protocol Software GSPN	36
6. Other Work	38
6.1. Bug Fixing	38
6.1.1. File Save Bug	38
6.1.2. Edit Arc Weight Bug	38
6.1.3. Invariant Analysis Bug	38
6.1.4. HTML File Save Formatting	40
6.1.5. Code Optimisation	41
6.2. User Interface Enhancements	41
6.3. Product Marketing	45
7. Conclusion & Future Work	46
7.1. Summary	46
7.2. Future Work	47
Appendix A – User Guide	48
Installation	48
Analysing a GSPN	48
Bibliography	52

1. Introduction

PIPE, the Platform Independent Petri Net Editor, began its life as an MSc. Group project in 2003. The aim was to create an application for editing and analysing Petri nets which are a formal system for describing concurrency and synchronization in distributed systems. However, there were already tools available to do this. A list of Petri net software applications can be found at the Petri nets world web-site: <http://www.informatik.uni-hamburg.de/TGI/PetriNets/> In fact, Imperial College has already developed a highly capable Petri net analysis tool called DNAmaca which is capable of running on a parallel processor system and analyse extremely large Petri net models.

Existing Petri net tool functionality is limited to that which their creators provide at the time of writing [BC+03]. The key difference with PIPE was to be ease of use, platform independence and easy extensibility. In this respect, the initial project was partially successful in that it was platform independent having been written in Java, open source, all code being available at www.sourceforge.net, and also extensible through the ability to easily add new analysis modules. Ease of use however was lacking to a large extent.

The following year, a new MSc. group project team took on the task of improving the user interface and bug fixing the analysis modules. They had five areas to target for development [BC+04]:

- Review of existing code and documentation including optimisation and bug fixing,
- Extension of capabilities to support Generalised Stochastic Petri Nets including representational and editing tools and relevant analysis modules
- Interfacing with DNAmaca for Petri net analysis
- Improved editing tools – improved interface and functionality for *creating* nets
- GUI – improved interface and functionality for *working* with nets.

They were broadly successful and along the way a new official branch of the project was created at 'sourceforge.net' called PIPE2. However there were still some shortcomings with the tool. The Petri net editing interface was still not as easy to use as it could be and the analysis capability was so limited that PIPE2 could not really be used for any serious analysis of generalised stochastic Petri net's.

In 2005, the task of enhancing the GUI and analysis capabilities of PIPE2 was offered as an individual MSc. project. There were three areas that needed work:

- Fix a number of moderately serious bugs in the user interface
 - Saving files did not always work

- The invariant analysis module did not give the correct analysis results and did not use the correct labels for the results
- Changing arc weights resulted in an error which was nonsensical
- Other as yet unknown bugs that may be found along the way
- Improve the arc editing tool. The options for editing an arc once it was drawn were limited and consisted of being able to delete an arc or change its weight.
- Improve the GSPN analysis capability. The existing capability was so severely limited that it was useless and all analysis of any real models had to be handed off to DNAmaca.

The rest of this report describes all the work carried out on the tasks listed above. The report first covers background theory on Petri nets including Generalised Stochastic Petri Nets (GSPN's), and also covers Markov theory so that it is possible to understand how GSPN's can be analysed. The next two chapters then cover the background theory of how GSPN analysis can be implemented in software and also how it was actually implemented in PIPE2. Chapter 5 then covers testing and results followed by a further chapter covering all the bug fixing and user interface enhancement.

2. Background Theory

2.1. Petri Nets

Petri nets are a formal method for describing the concurrency and synchronization present in distributed systems or processes. They were first described by Carl Adam Petri in 1962 as a method for determining the correctness of concurrent systems. Many variations of the original Petri nets now exist which have extended their usefulness and they have been used for modelling communications protocols, parallel programs and distributed databases amongst other things [Kno99].

The simplest type of Petri nets, sometimes referred to as 'ordinary' Petri nets [BK02], are Place-Transition nets. These consist of 4 elements:

- **Places**, drawn as circles. These represent conditions or objects for example, program variables. Places contain zero or more,
- **Tokens**, drawn as black dots. These represent the specific value of the condition or object for example, the value of the program variable. The pattern/arrangement of tokens across all the places in a place-transition net is called the marking.
- **Transitions**, drawn as rectangles. Transitions model activities which result in a change in the values of conditions and objects and hence a change in the marking of the place-transition net.
- **Arcs**, drawn as arrows between places and transitions. Arcs show the relationships between specific places and transitions and thus indicate which objects and conditions are changed by which activity.

Place-transition nets are bipartite graphs meaning a place can only be connected to a transition and vice-versa. Places cannot be connected to places and transitions cannot be connected to transitions.

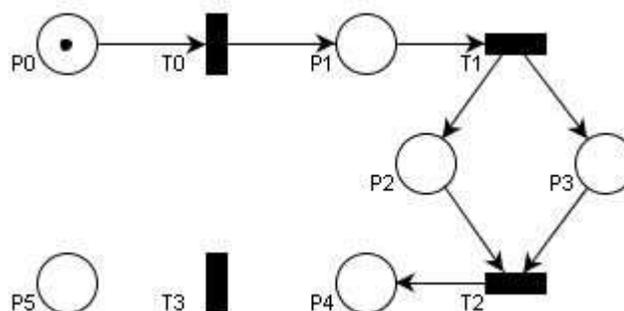


Figure 2-1 A Place-Transition Net

Figure 2-1 shows a simple place-transition net with six places, four transitions and eight arcs. One of the places (P0) contains a token. We say P0 is an input place for transition T0 and accordingly T0 is an output transition of P0. P1 is an output place of T0 and accordingly T0 is an input transition for P1 and so on. Places and transitions can have more than one input or output element as can be seen with T1, T2, P2 and P3. Places and transitions can also be unconnected such as P5 and T3. As expected, such elements do not have any influence on the rest of the net and can be neglected during analysis.

T0 is said to be 'enabled' as its input place has a token on it. The number of tokens required on input places for a transition to be enabled can be shown by adding a weight to the arc joining them. If all the input places for a transition have the required number of tokens, the transition can 'fire'. This destroys the number of tokens specified by the arc weights from the input places and creates new tokens on the output places. Note however that an enabled transition does not have to fire. The weight on arcs from transitions to their output places identifies the number of tokens created on an output place when the transition fires. By convention, arc weights of 1 are not shown explicitly.

A place-transition net can be formally described as follows [BK02]:

A place-transition net is a 5-tuple $PN = (P, T, I^-, I^+, M_0)$ where

- $P = \{p_1, \dots, p_n\}$ is a finite and non-empty set of places,
- $T = \{t_1, \dots, t_n\}$ is a finite and non-empty set of transitions,
- $P \cap T = \emptyset$,
- $I^-, I^+ : P \times T \rightarrow \mathbb{N}_0^1$ are the backward and forward incidence functions respectively,
- $M_0 : P \rightarrow \mathbb{N}_0$ is the initial marking.

The incidence functions specify the connections between places and transitions. If $I^-(p,t) > 0$, an arc leads from *place* p to *transition* t . As such, I^- is called the backward incidence function of transition t . I^- maps into the natural numbers and is the weight of the arc from p to t , as such it is the number of tokens destroyed on place p when transition t fires. Similarly, if $I^+(p,t) > 0$, an arc leads from *transition* t to *place* p . It is called the forward incidence function and is the number of tokens created on place p when t fires.

With this formal definition in place we can now say the following:

- A marking of a place-transition net is a function $M : P \rightarrow \mathbb{N}_0$, where $M(p)$ is the number of token on place p .
- A transition $t \in T$ is enabled at M iff² $M(p) \geq I^-(p,t), \forall p \in P$.
- A transition $t \in T$ enabled at marking M , may fire resulting in a new marking M' where $M'(p) = M(p) - I^-(p,t) + I^+(p,t), \forall p \in P$.

¹ \mathbb{N} denotes the set of positive integers and \mathbb{N}_0 additionally includes 0.

² iff means "if and only if".

- We say M' is *directly* reachable from M , written as $M \rightarrow M'$. Let \rightarrow^* be the reflexive and transitive closure of \rightarrow . A marking M' is *reachable* from M iff $M \rightarrow^* M'$
- The reachability set of a place-transition net PN is $R(PN) := \{M | M_0 \rightarrow^* M\}$
- We say a place-transition net is bounded iff $\forall p \in P : \exists k \in \mathbb{N}_0 : \forall M \in R(PN) : M(p) \leq k$

The most common way to analyse a place-transition net is to analyse its reachability set as all properties are defined from this set.

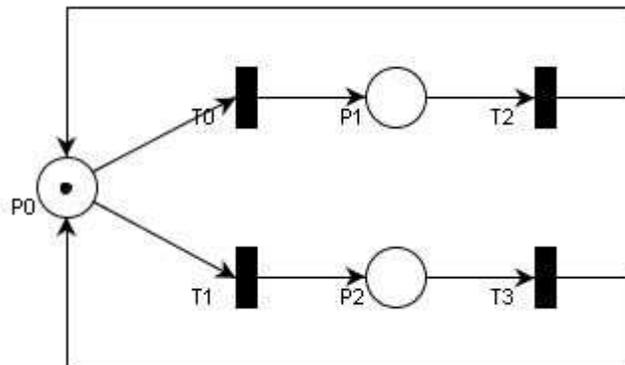


Figure 2-2 A place transition net

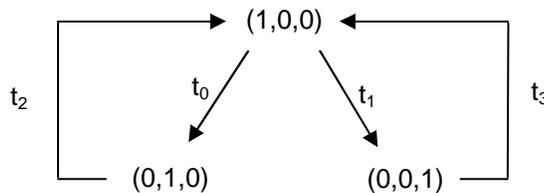


Figure 2-3 The corresponding reachability graph

Figure 2-2 and Figure 2-3 show a place-transition net and its corresponding reachability graph. The nodes in the graph are markings of the place-transition net. Two nodes M and M' are connected with a directed arc if $M \rightarrow M'$ and the arc is labelled with the transition enabled in M which fires to lead to M' . The graph is obtained by starting with the initial marking of the place-transition net as a root node and then adding directly reachable markings as leaves. The process is then repeated with the leaves and so on. If we reach a marking which has been previously explored, an arc is drawn back to that marking and it obviously does not need to be explored further. In the case of unbounded nets, the reachability set is infinite. Special notation is used for these sets but will not be covered here.

2.2. Generalised Stochastic Petri Nets (GSPN's)

Place-transition nets can be used to test a system for certain desired 'correctness' properties such as boundedness, liveness and freedom from deadlock. However, place-transition nets do not include time in the model making it impossible to analyse performance of a system. As such place-transition nets have been extended to include several types of time-augmented Petri nets. Generalised Stochastic Petri Nets fall into the category of time-augmented Petri nets and are a flexible and widely used representation. GSPN's have two types of transitions, *immediate* transitions and *timed* transitions. Immediate transitions are shown in Petri nets as filled rectangles. Timed transitions are shown as empty rectangles. Once enabled, immediate transitions fire in zero time. Timed transitions fire after a random exponentially distributed time. More than one transition can be enabled in any particular marking M . If the set of enabled transitions $EN_T(M)$ contains any immediate transitions, then only the immediate transitions are considered enabled and no timed transition can fire. In other words, in a GSPN firing of immediate transitions has priority over timed transitions.

Using and extending the same notation and definitions from the previous section we can describe GSPN's as follows:

A GSPN is a 4-tuple $GSPN = (PN, T_1, T_2, W)$ where

- $PN = (P, T, I^-, I^+, M_0)$ is the underlying place-transition net
- $T_1 \subseteq T$ is the set of timed transitions, $T_1 \neq \emptyset$
- $T_2 \subset T$ is the set of immediate transitions, $T_1 \cap T_2 = \emptyset$, $T = T_1 \cup T_2$
- $W = (w_1, \dots, w_{|T|})$ is an array whose entry $w_i \in \mathbb{R}^{+3}$ is a (possibly marking dependent) rate of a negative exponential distribution specifying the firing delay when transition t_i is a timed transition or is a (possibly marking dependent) firing weight when transition t_i is an immediate transition.

Each distinct marking in the reachability set of a GSPN is a state of the system. As such the terms marking and state are interchangeable when discussing GSPN's. The reachability set can also be referred to as the state space of the Petri net. The state space of a GSPN actually contains two types of markings. As immediate transitions fire in zero time, markings in which immediate transitions are enabled have a sojourn time of zero. These markings are called vanishing states as they will never be seen by an observer who randomly examines the stochastic process of a GSPN. States in which only timed transitions are enabled will be seen by an observer and as such are referred to as tangible states. Vanishing states have no effect on the performance statistics derived for a GSPN due to their zero sojourn time and so they are usually eliminated during or after

³ \mathbb{R}^+ is the set of positive real numbers

state space generation [Kno96]. Throughout the rest of this report, it will be assumed that the GSPN has a finite reachability set.

In order to analyse a GSPN, we utilise a branch of mathematics called Markov theory.

2.3. Markov Theory

2.3.1. Random Variables

A variable is random and denoted as χ if we cannot tell for certain what its value will be. The random variable χ is said to be discrete if the set of values it can be is countable (but not necessarily finite). As we do not know for certain what the value of χ is, we say it will have a value x with a probability of $p_\chi(x)$,

$$p_\chi(x) = P[\chi=x]$$

Eq. 2-1

where x is a real number and $0 \leq p_\chi(x) \leq 1$ for all values of x . The cumulative distribution function of a random variable χ is:

$$F_\chi(a) = \sum_{x \leq a} p_\chi(x)$$

Eq. 2-2

If the set of values χ can be is not countable, then it is a continuous random variable. Now we talk about χ being less than or equal to a certain value 'a'. The cumulative distribution function changes from summation to integration and

$$F_\chi(a) = P[\chi \leq x]$$

Eq. 2-3

2.3.2. Stochastic Processes

A stochastic process is a family of random variables $\{\chi(t)\}$ indexed by the time parameter t . If the time index is a countable set, the process is a discrete time process otherwise it is a continuous time process. The values or states that members of $\{\chi(t)\}$ can take are the state space of the process. If the state space is discrete, the process is called a chain.

As explained in the previous subsections, the behaviour of a system can often be characterised by determining all the states in a system and by describing how the system changes from one state to another. In general, systems such as these can be described as a stochastic process.

2.3.3. Markov Processes

In 1907 a Russian mathematician called A. A. Markov, described a class of stochastic processes where [BK02]:

$$P[\chi(t) = x \mid \chi(t_n) = x_n, \chi(t_{n-1}) = x_{n-1}, \dots, \chi(t_0) = x_0] = P[\chi(t) = x \mid \chi(t_n) = x_n] \text{ where } t > t_n > t_{n-1} > \dots > t_0$$

Eq. 2-4

In other words, the future of the process from time t_n onwards is determined only by the present state. This condition is known as the Markov property. A Markov process is a stochastic process for which the Markov property holds. An implication of the Markov property is that the distribution of the sojourn time in any state cannot depend on the time spent in that state,

$$P[\chi \geq y + s \mid \chi \geq s] = P[\chi \geq y]$$

Eq. 2-5

This condition places restrictions on the distribution of time spent in a state [Kno96].

A Markov process is said to be *homogeneous* or *stationary* if it is invariant to shifts in time i.e.

$$P[\chi(t+s) = x \mid \chi(t_n+s) = x_n] = P[\chi(t) = x \mid \chi(t_n) = x_n]$$

Eq. 2-6

2.3.4. Continuous-Time Markov Chains

A continuous-time Markov chain (CTMC) is a Markov process with a discrete state space and a state that may change at any time. A stochastic process $\{\chi(t)\}$ forms a CTMC if [Kno96]:

For all integers n and for any sequence $t_0, t_1, \dots, t_n, t_{n+1}$ with $t_0 < t_1 < \dots < t_n < t_{n+1}$ we have,

$$P[\chi(t_{n+1}) = x_{n+1} \mid \chi(t_0) = x_0, \chi(t_1) = x_1, \dots, \chi(t_n) = x_n] = P[\chi(t_{n+1}) = x_{n+1} \mid \chi(t_n) = x_n]$$

Eq. 2-7

The condition which restricts the distribution of sojourn times (Eq. 2-5) is only met by the exponential distribution for CTMC's. A homogeneous CTMC can be represented by a set of states and an infinitesimal generator matrix Q where Q_{ij} for $i \neq j$ is the exponential transition rate between state x_i and state x_j . The parameter of the exponential sojourn time in state x_i is $-Q_{ii}$ where

$$Q_{ii} = -\sum_{j \neq i} Q_{ij}$$

Eq. 2-8

i.e. it is the negative sum of all the elements on that row of the matrix. As such the sum of all the elements on a row of matrix Q is 0,

$$\sum_j Q_{ij} = 0 \quad \forall i$$

Eq. 2-9

The most important step in a Markov chain analysis is to determine how much time is spent in each state x_j .

$$\pi_j^{(m)} = P\{\chi_m = x_j\}$$

Eq. 2-10

is the probability of finding the Markov chain in state x_j at time step m . If we have a CTMC whose state probability distribution $\pi_j^{(m)}$ does not change when $m \rightarrow \infty$, the distribution is said to be stationary or said to have reached a steady state π_j . This can only be the case if, for a CTMC with infinitesimal generator matrix Q ,

$$\pi Q = 0$$

Eq. 2-11

In a finite, homogeneous, irreducible CTMC, the limiting probabilities $\{\pi_j\}$ always exist and are independent of the initial probability distribution. Also, the set $\{\pi_j\}$ is a stationary probability distribution which can be uniquely determined by solving the set of equations [Kno96]:

$$-q_{jj}\pi_j + \sum_{i \neq j} q_{ij}\pi_i = 0$$

Eq. 2-12

and

$$\sum_i \pi_i = 1$$

Eq. 2-13

These equations are sometimes called the global balance equations and they can be rewritten in vector form as:

$$\pi Q = 0$$

Eq. 2-14

where $\pi = (\pi_1, \pi_2, \dots)$ is the steady state probability vector, hence meeting the condition of Eq. 2-11.

If we now look back to GSPN's and their reachability graphs, it can be shown that the sojourn time in a particular marking or state is independent of the time already spent in that state. In fact, sojourn times are exponentially distributed. Also, the probability of changing state is independent of the sojourn time. These two facts imply that a GSPN describes a Markov process and analysis of the GSPN can be carried out through analysis of the underlying Markov process. Each state in the reachability graph can be regarded as a state of the corresponding Markov chain. The firing rate λ_i of each transition t_i labelling each arc in the reachability graph marks the arcs in the corresponding Markov chain.

2.4. An Example

Consider the GSPN of Figure 2-4. This is a simple GSPN for which the reachability graph can be easily determined and is shown in Figure 2-5. There are no vanishing states in this example which is no surprise as there are no immediate transitions.

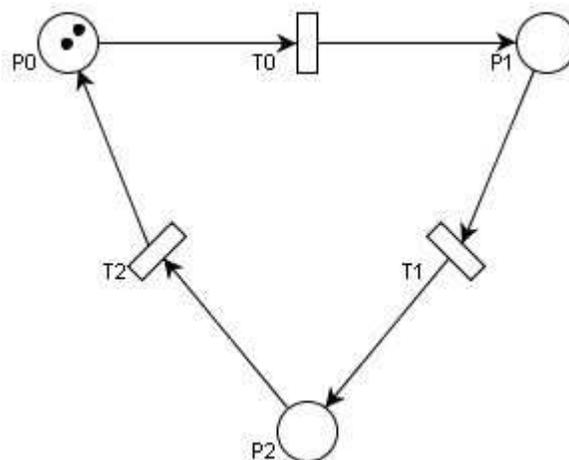


Figure 2-4 A simple GSPN

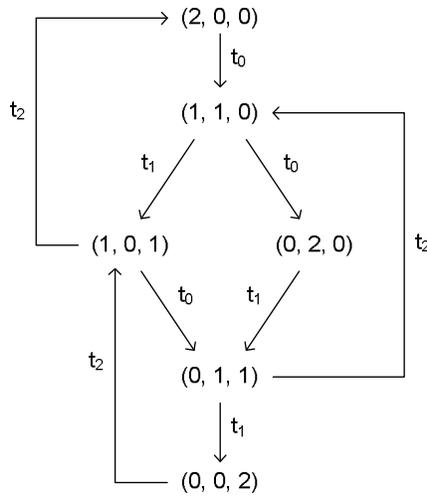


Figure 2-5 The corresponding reachability graph

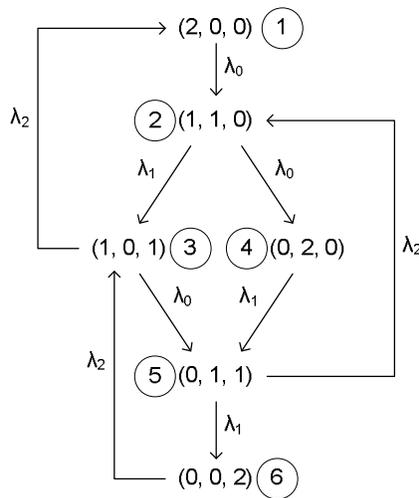


Figure 2-6 The corresponding Markov chain

Using the information in the reachability graph we can create the corresponding Markov chain, Figure 2-6, and immediately write down the infinitesimal generator matrix Q as:

$$Q = \begin{pmatrix} -\lambda_0 & \lambda_0 & 0 & 0 & 0 & 0 \\ 0 & -(\lambda_0 + \lambda_1) & \lambda_1 & \lambda_0 & 0 & 0 \\ \lambda_2 & 0 & -(\lambda_0 + \lambda_2) & 0 & \lambda_0 & 0 \\ 0 & 0 & 0 & -\lambda_1 & \lambda_1 & 0 \\ 0 & \lambda_2 & 0 & 0 & -(\lambda_1 + \lambda_2) & \lambda_1 \\ 0 & 0 & \lambda_2 & 0 & 0 & -\lambda_2 \end{pmatrix}$$

To understand how the matrix has been composed we first number the states in the Markov chain as indicated in Figure 2-6. Now each row index can be thought of as the state an arc in the Markov chain is *from* and each column is the state the arc is *to*. The element at the row-column intersection is then the rate of the transition leading to that state change. Diagonal elements in the matrix are the negative sum of the off-diagonals on that row i.e. each row will sum to zero. If a state has an arc to itself, that rate can be ignored as clearly it will have no influence at steady state.

Now we have Q, we need to solve the equation

$$\pi Q = 0$$

Eq. 2-15

for the steady state distribution π . The equation can be re-written as

$$Q^T \pi^T = 0$$

Eq. 2-16

This is now in the form $Ax = b$, and there are many standard methods for solving equations in this form including direct methods such as Gaussian elimination.

For our example, if we set the rates for all three timed transitions to 1.0 giving $\lambda_0 = \lambda_1 = \lambda_2 = 1.0$ we get the following steady state solution:

$$\pi = \left(\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6} \right)$$

Now with a steady state solution in hand it is possible to continue and calculate parameters such as the average number of tokens on a place, the token probability distribution and transition throughput. This will not be shown here.

3. State Space Exploration

3.1. A basic algorithm

The previous chapter covers all the material necessary to understand what is involved in analysing a Generalised Stochastic Petri Net. PIPE is intended to be a tool which can automate the analysis process of Petri nets and in this chapter and the next we analyse how this automation can be carried out.

The first step in analysing any GSPN is to determine its reachability set or state space. At first glance this may seem relatively simple. If we start in marking M_i and keep a stack of states waiting to be explored (S_w) and a set of states that have been explored (S_e), the following algorithm lends itself to the task:

```
push (Mi, Sw)
Se = {Mi}
while(Sw not empty)
    Mc = pop(Sw)
    determine all successors of marking Mc
    for each successor Ms
        if Ms  $\notin$  Se
            Push(Ms, Sw)
            Se = Se  $\cup$  {Ms}
        end if
    for ends
while ends
```

Figure 3-1 A simple state space exploration algorithm

In fact, this is almost exactly the algorithm used by the original GSPN analysis module in PIPE. This algorithm is very simple to program and leads to a depth first search (DFS) of the state space of the GSPN under consideration. As the algorithm proceeds, information on the transitions between states can be recorded at the same time as determining the successors to a state enabling the creation of the infinitesimal generator matrix.

The stack of states waiting to be explored at any one moment in time is not especially large and is limited by the depth of the reachability graph. As such it is not normally an issue in terms of memory storage requirements. The set of explored states however is another matter. It needs to hold enough information to be able to determine whether the current state under scrutiny has been explored before or is new and we have to be able to search through this information quickly. Eventually, it will hold information on every single state in the state space of which there could be hundreds of thousands if not millions. An efficient layout and management of the set of explored states is needed.

3.2. Memory and Time Efficient Strategies

3.2.1. A Linked List

The set of explored states is crucial in determining the entire state space. Any scheme needs to record enough information on each state to be able to identify it with a high level of confidence. The simplest solution is a linked list where each node in the list gives a full description of the state.

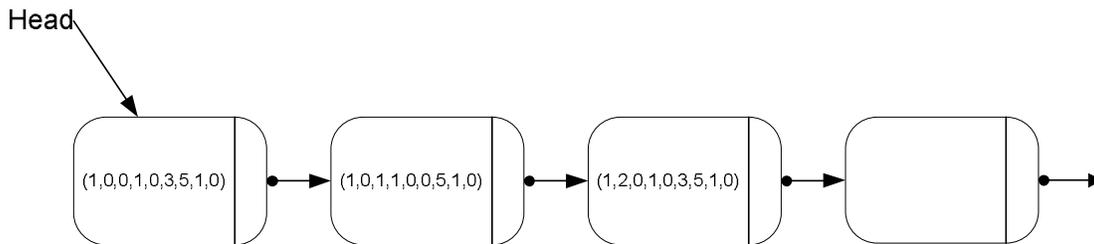


Figure 3-2 The set of explored states as a linked list

This is very easy to program and as each node in the list describes a state completely, you can be completely confident of correctly identifying whether a state has been previously explored. As the number of explored states increases however, the time it takes to search the list increases. Adding new elements to the list can be kept to a constant time by just adding to the head of the list but on average, searching the list will require $N/2$ comparisons for a successful search (a search hit), i.e. the state has been found in the set of explored states, and N comparisons for an unsuccessful search (a search miss) where N is the number of nodes or explored states in the list. Even if the list were to be ordered in some way, insertion, search hits and search misses would all take $N/2$ comparisons on average [Sed98]. As the number of explored states could reach many hundreds of thousands, the search time would rapidly become unacceptable and be the most time consuming part of any GSPN analysis.

A further source of difficulty arises not from the linked list itself but from the fact that each node carries a full description of a state. Whilst this guarantees correct identification of states, the memory requirements become very significant. The state descriptor would most likely be an array where the number of elements is determined by the number of places in the Petri net. Each array element would be an integer which in the case of the Java Virtual Machine is fixed to be 4 bytes in size. This means that each array or state descriptor could easily be 100 bytes. Even if we ignore the memory overhead for implementing a linked list, 100,000 states would call for almost 10Mbytes of memory and in general, n explored states would require dxn bytes where d is the size of the state descriptor in

bytes. Whilst most modern computer systems now include hundreds of megabytes of RAM, this is still a significant amount of memory to be consuming.

3.2.2. A Hash Table with Full State Information

Dealing with issues of search time first, hash tables eventually come to mind as they provide a good time-space trade-off [Sed98]. Using the state as a key into the hash table provides very fast access. If we were not limited by memory, every state could map to its own row in a hash table. However, we must make a compromise and allow more than one state to map to a row in the hash table. Each hash row then needs a scheme to cope with a collision on that row, i.e. the fact that more than one state maps to it. If each row is made to be the head of a linked list, we now have r smaller lists to search where r is the number of rows in the hash table. This is the familiar hash table with separate chaining.

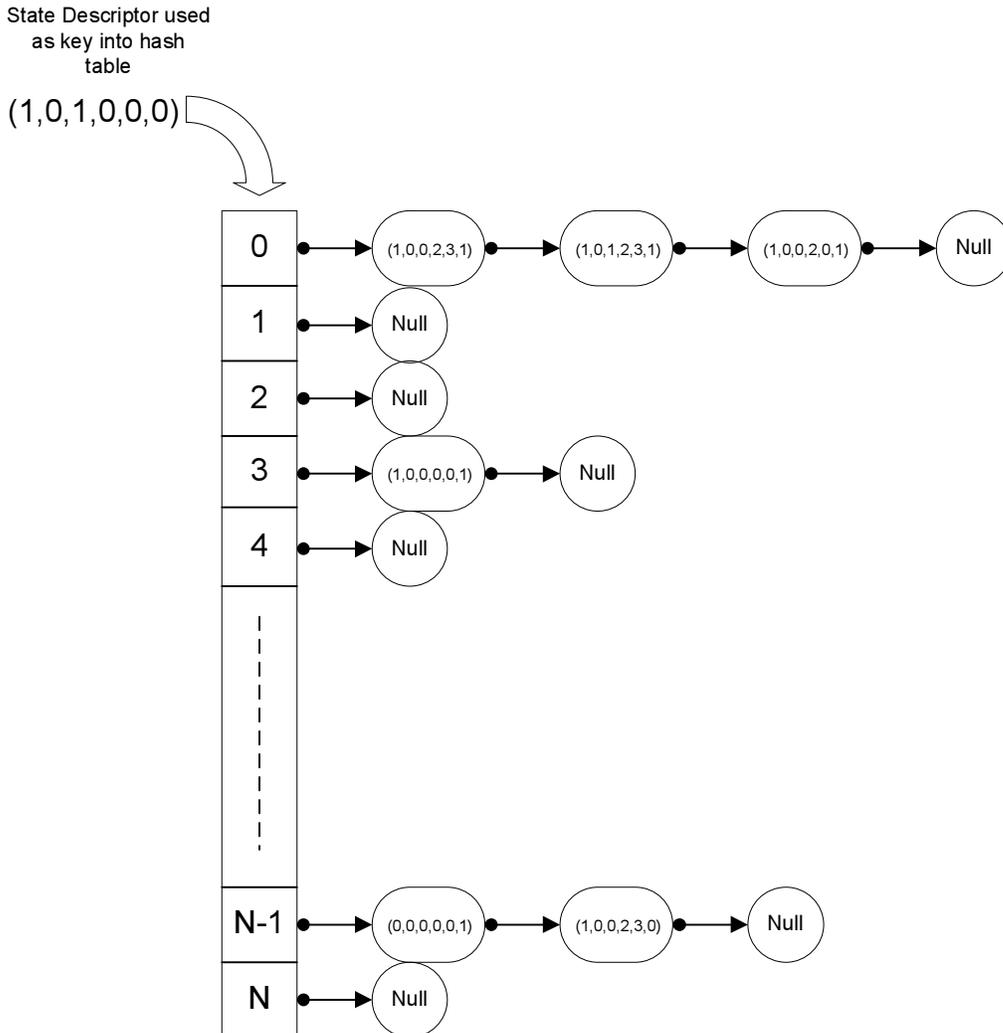


Figure 3-3 A hash table with separate chaining using the state descriptor as a key indexing into the table

Now search times become on average n/r comparisons for a search miss and $n/2r$ for a search hit where n is the number of states stored.

Coming back to memory requirements, we see that it has become even more of an issue. Instead of one linked list, we now have the memory overhead of r linked lists plus that of the hash table itself. Assuming each hash table row requires h bytes, our memory consumption has increased from dn to $dn+rh$ bytes!

3.2.3. A Probabilistic Dynamic Technique

Instead of storing a full state descriptor, memory requirements could be reduced in the hash table by storing a more compact description of a state. There are a number of probabilistic techniques briefly described in [Kno96]. The first, Holzmann's bit state hashing, represents the table of explored states as a bit vector. A hash function maps states to positions in this bit vector. Once a state has been explored, the corresponding bit is set to 1 otherwise it is zero. The problem with this technique is that there is no scheme for collision resolution. If more than one state hashes to the same bit position, only one of them will be recorded, the others will be erroneously discarded. To minimise the probability of omission the ratio of states to hash table entries must be very low. This makes the bit vector impractically large.

The second technique, Leroy and Wolper's hash compaction, is to hash a state to a compressed k bit key which is then used to map into a smaller hash table. The details of this method are limited so it's not clear how the compressed key is used to map into the smaller hash table. The third technique covered is by Stern and Dill and improves the omission probability of Leroy and Wolper's method by independently calculating the compressed value of a state and its hash value.

The final technique described in [Kno96] is the method now implemented in PIPE2. It combines the probabilistic techniques briefly described above with the dynamic storage allocation technique used in the exhaustive state space exploration of section 3.2.2. Each row of the hash table still contains a linked list but now each element in the list contains a compressed version of the state rather than a full description of the state. Two independent hashing functions are required. The first hash function h_1 is used to determine which hash table row the state belongs to. The second hash function h_2 is used to create a compressed state descriptor value. The steps for deciding whether a state s has already been explored now become,

- Calculate the primary hash key $h_1(s)$ to determine the row of the hash table,

- Calculate the secondary hash key $h_2(s)$ to get the compressed state descriptor and compare it to the state descriptors already in the table at that row,
- If the compressed state descriptor is already present, the state is considered to have been explored and nothing further needs to be done with it, otherwise add the compressed state descriptor to the hash table row, determine all the states successors and push them on to the stack of states to be explored.

Full State Descriptor used as key into hash table through primary hashing function h_1

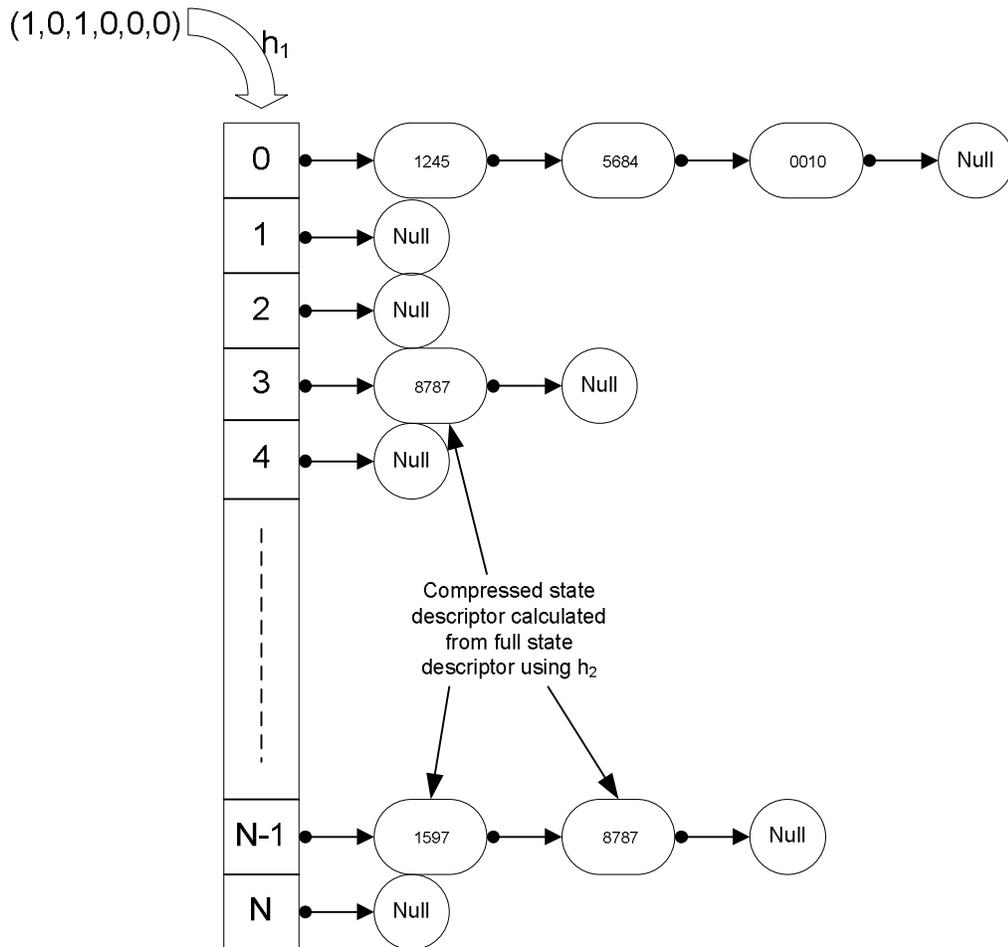


Figure 3-4 A hash table using compressed state descriptors

Now the probability of omission has been reduced as a state is deemed to have been explored only if $h_1(s)$ and $h_2(s)$ equal that for a state already in the hash table. It is still possible for two different states to be mistakenly identified as the same but this will be the case with all probabilistic methods and is the price that has to be paid if less memory is to be used. Memory requirements are now lower than if a full state descriptor was used as the hash function h_2 compresses a

state to a single integer value. Comparing memory requirements to the case where 100 bytes were needed for a full state descriptor, we now use 4 bytes and this will be the case now matter how large the full state descriptor becomes!

We still need to consider how many rows we wish to have in our hash table. To assist in distributing keys randomly, it is preferable to use a prime number for the number of rows in the table. Dr. Knottenbelt in [Kno96] calculates the optimum number of rows for storing 2million states using a 32bit key for an omission probability of 0.01 as being 46566 rows. The 4812th prime is 46567⁴ and this is used in PIPE2 as the number of rows in the hash table.

3.2.4. The primary and secondary hashing functions h_1 and h_2

Most books on algorithms such as [Knu98] and [Sed98] cover hashing functions in depth. Fundamentally, hashing functions transform keys, in our case the full state descriptor, into an address in a table. They are normally simple arithmetic processes but as is always the case with hashing functions, they must be carefully chosen to distribute keys randomly and there is no single perfect hashing function for all uses. There are two hashing functions used for the explored states table in PIPE2. The primary hashing function, h_1 , maps a state to a row in the hash table. The secondary hash function, h_2 , calculates a value from the full state descriptor which acts as a compressed representation of the state. Both the primary and secondary hashing functions use a process of summation to generate the hash keys. The primary function converts a state s to a key as follows:

$$s = \{s_1, s_2, \dots, s_n\}$$

$$\text{total}_1 = \sum_{i=1}^n s_i$$

$$\text{total}_2 = \text{total}_1 + \sum_{i=1}^{n-1} s_i$$

...

$$\text{total}_n = \sum_{i=1}^{n-1} \text{total}_i + s_1$$

$$h_1(s) = \sum_{i=1}^n \text{total}_i$$

This algorithm is very easy to program as a simple 'for' loop as in Figure 3-5. The secondary hashing function uses a similar algorithm but generates a different hash key. Each summation total_i in the primary hash function adds a smaller and smaller subset of the state descriptor s to the previous total. The

⁴ Obtained from The Primes Database <http://primes.utm.edu/nthprime/>

secondary hash function does the same but instead of removing the tail of the list from the summation, it removes the head. This means the primary and secondary hashing functions give different weights to each element in the state descriptor. A small change in element s_n would give a small change in the primary hash function but a large change in the secondary hash function.

As PIPE2 is implemented in Java, we need to consider the high possibility of overflow in the integer storing the running total as the hashing algorithm progresses. This would manifest itself as the integer becoming negative as Java has no unsigned integer data type. As the integer returned by the primary hashing function will be used as an index into a hash table, it must be explicitly wrapped round to zero again. This is not required for the secondary hashing function as it is just used to identify a state and not index into a table.

```
public int h1(){
    int total = 0;
    for(int offset = 0; offset < state.length; offset++){
        total = (2*total);
        for(int index = 0; index < (state.length - offset); index++){
            total += state[index];
        }
    }
    // As the above for loop results in massive integers
    // check we haven't overflowed and gone negative.
    // If we have then wrap round to zero.
    if(total < 0)
        total = Integer.MAX_VALUE+total;

    return total;
}
```

Figure 3-5 A java method for calculating the primary hash key

A further step when using the primary hash key is to limit it to the range of the number of rows in the hash table. This is carried out with simple modulo arithmetic by the method calling the hashing function and not the hashing function itself.

3.2.5. Vanishing State Elimination

Further space savings can be made during state space exploration. The basic algorithm described in section 3.1 will keep track of both tangible states and vanishing states. However, once the state space has been explored the vanishing states are of little or no use in any analysis. A better algorithm would eliminate vanishing states 'on the fly' and only tangible states would be recorded. The transition rates of the vanishing states between two tangible states would be combined into a single 'effective transition rate' as in Figure 3-6. An additional

benefit of vanishing state elimination comes in the form of reduced effort when solving the steady state of the underlying Markov chain. PIPE2 uses a modified version of the algorithm described in [Kno96] to perform a breadth first search of the state space with vanishing state elimination, Figure 3-7. It uses a hash table, S_e , as described previously for the set of explored states, a *queue* Q_t of tangible states waiting to be explored and a *stack* S_v of vanishing states waiting to be explored and then eliminated. Each object in the queue or stack is in the form of a record $\langle s, r \rangle$ where s is the state and r is the associated rate of entry into that state.

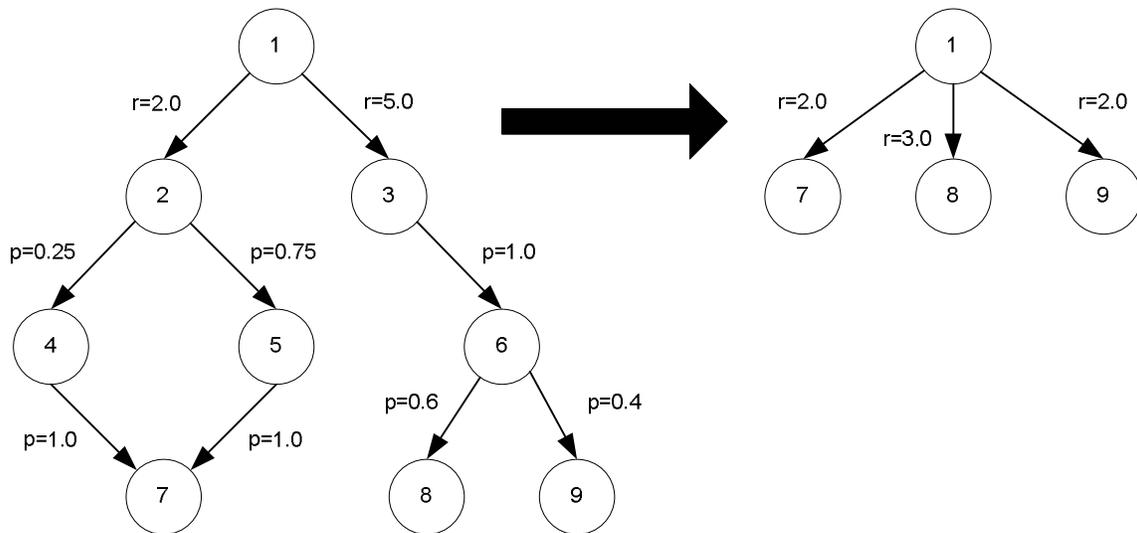


Figure 3-6 An example of vanishing state elimination

The algorithm begins by determining the initial tangible states of the system and placing them in a queue, given an initial state s_i which could be vanishing. The algorithm then proceeds to its second phase where tangible states are dequeued in turn. As each tangible state is dequeued, its successors are determined by firing all enabled transitions in turn. Any tangible successors are added to the queue of tangible states to be explored later, vanishing states however are added to the stack of vanishing states and immediately explored in depth first search fashion. Any vanishing successors to a vanishing state are also pushed on to the stack and explored immediately until a tangible successor is found which can then be added to the queue. Vanishing states continue to be explored until the stack is empty meaning all tangible successors have been found. Rates of transition between tangible states are determined as the algorithm progresses by using the information on the vanishing state stack together with two methods ' $prob(s, s')$ ' and ' $rate(s, s')$ ' to propagate transition probabilities through successive vanishing states. The method $prob(s, s')$ calculates the probability of a transition from a vanishing state s to another state s' , which could be tangible or vanishing. The method $rate(s, s')$ calculates the rate of a transition from a tangible state s to another state s' which could be tangible or vanishing.

```

/* Phase I: Initialise Qt with initial tangible states */
if (si is tangible)
    enqueue(Qt, si)
    Se = {si}
else
    push(Sv, <si, 1.0>)
    while(Sv not empty)
        pop(Sv, <v, p>)
        determine all successors to v
        for each successor v'
            if(v' is tangible)
                if(v' ∉ Se)
                    enqueue(Qt, v')
                    Se = Se ∪ {v'}
                end if
            else
                p' = p*prob(v, v')
                if(p' > ε)
                    push(Sv, <v', p'>)
                end if
            end if else
        end for
    end while
end if else
/* Phase II: State space exploration with vanishing state elimination */
while(Qt not empty)
    dequeue(Qt, s)
    determine all successors to s
    for each successor s'
        if (s' is tangible)
            transition(s, s', rate(s, s'))
            if (s' ∉ Se)
                enqueue(Qt, s')
                Se = Se ∪ {s'}
            end if
        else
            push(Sv, <s', rate(s, s')>)
            while(Sv not empty)
                pop(Sv, <v, p>)
                determine all successors to v
                for each successor v'
                    p' = p*prob(v, v')
                    if(v' is tangible)
                        if(v' ∉ Se)
                            enqueue(Qt, v')
                            Se = Se ∪ {s'}
                        end if
                    transition(s, v', p')
                    else
                        if(p' > ε)
                            push(Sv, <v', p'>)
                        end if
                    end if else
                end for
            end while
        end if else
    end for
    write s and its tangible successors to disk for later use
end while

```

Figure 3-7 The state space exploration algorithm used in PIPE2

At appropriate points, the algorithm records information for creating the reachability graph through the use of a method called $\text{transition}(s, s', r)$. This method records the fact that there is an arc in the reachability graph from tangible state s , to tangible state s' with effective transition rate r . The method uses a list which temporarily records all the arcs in the graph from state s to its tangible successors. Once all the tangible successors have been found and the effective rates of transition to them determined, the list can be written to a file and started again for the next tangible state to be explored. In this way, it does not require any substantial amount of memory.

Whilst eliminating vanishing states, two special cases need to be considered. One is the case of cycles of vanishing states as illustrated in Figure 3-8. As we do not keep vanishing states in our explored states table, it is not possible to tell that a certain vanishing state has been seen before. One possibility suggested in [Kno96] is to keep a local stack of explored vanishing states and then apply traditional matrix arithmetic methods to reduce the infinitesimal generator matrix locally. Another simpler method is to deal with cycles by simply dropping vanishing states whose propagated effective entry rate falls below a certain threshold value, ϵ . This is the method implemented in the algorithm of Figure 3-7.

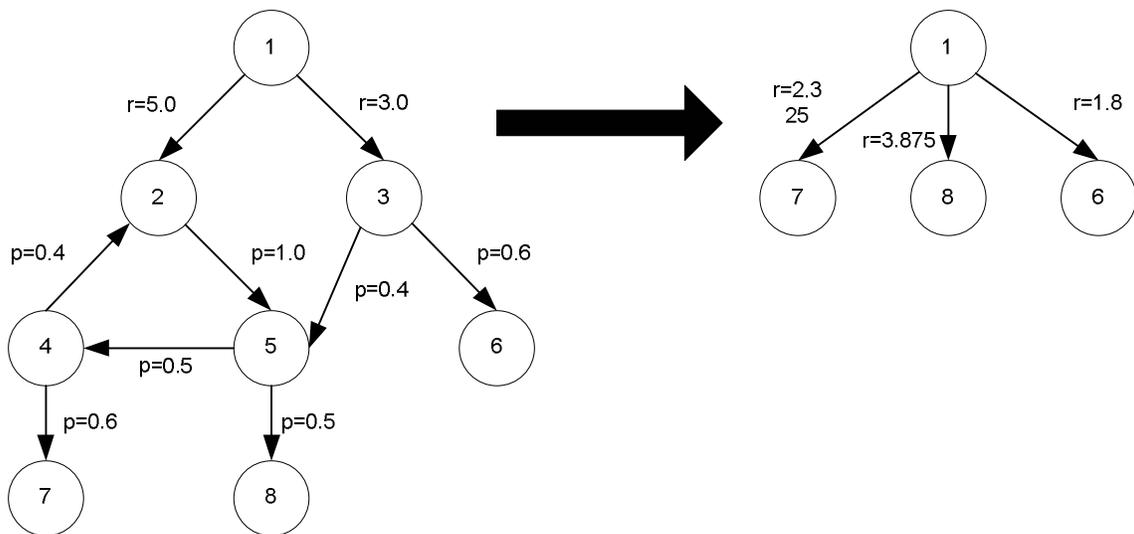


Figure 3-8 An example of a cycle of vanishing states [Kno96]

The second special case that must be considered is again a cycle of vanishing states, Figure 3-9. However, in this case the propagated probability never falls below zero. Cycles such as this are referred to as timeless traps. In these cases, analysis of the underlying Markov chain and hence of the GSPN is impossible. A technique is needed to catch timeless traps. Although not shown in the algorithm of Figure 3-7 for reasons of clarity, PIPE2 keeps a count of the number of attempts to clear the stack of vanishing states. If the number of attempts rises

above a value θ , it is assumed a timeless trap has been found and state space exploration is aborted.

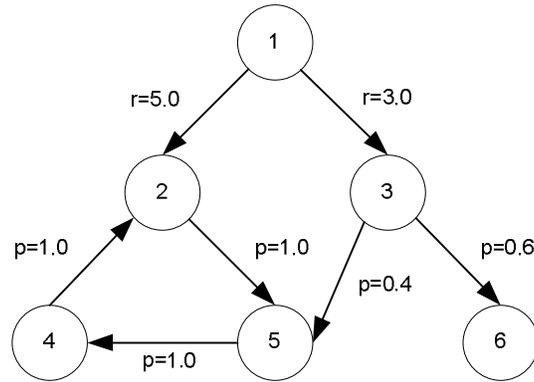


Figure 3-9 An example of a timeless trap [Kno96]

4. Steady State Solution

4.1. *Direct Methods*

Exploring the state space and determining the reachability graph is just the first step in analysing a GSPN. The next stage is the calculation of the steady state distribution π . As explained in section 2.3.4, this involves solving the set of equations:

$$\pi Q = 0, \sum_i \pi_i = 1$$

Eq. 4-1

Before solving the set of equations in Eq. 4-1, they are re-arranged as shown in section 2.4 to give an equation in the form $Ax=b$.

$$Q^T \pi^T = 0$$

Eq. 4-2

There are a number of techniques for solving equations in this form. The most familiar perhaps is Gaussian-Elimination. This involves adding multiples of rows to one another to reduce the augmented matrix $[A \mid b]$ to an upper triangular matrix $[U \mid c]$ and then following this with back substitution to solve for π . There is a problem with this technique however in that it only applies in the case where matrix A is non-singular. This is not the case in this instance and as such the matrix A has to be modified to make it singular. There are two approaches that can be taken, 'replace an equation' and 'remove an equation'. These are described in detail in [Kno96] and will not be covered here. Once one of these steps has been taken it is then of course possible to proceed with Gaussian elimination as normal to calculate the steady state.

Other direct methods for solving equation Eq. 4-2, are LU decomposition and Grassman's algorithm. Both are related to Gaussian elimination and have some advantages in terms of greater accuracy with regards to the double precision arithmetic.

4.2. *Iterative Methods*

Instead of solving equation Eq. 4-2 using direct methods like those in the previous section, one of many iterative methods could be used.

4.2.1. Jacobi's Method

Solving equation Eq. 4-2 actually involves solving n equations of the form:

$$\sum_{j=1}^n a_{ij}x_j = b_i \quad i=1,2,\dots,n$$

Eq. 4-3

This can be re-arranged to solve for the individual x_i values as follows:

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij}x_j \right)$$

Eq. 4-4

This suggests iteration as follows:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij}x_j^{(k)} \right)$$

Eq. 4-5

where $k \geq 0$ and $x^{(0)}$ is an initial guess at the solution.

4.2.2. Gauss-Siedel

An improvement upon Jacobi's method would be to use the newly calculated value of x_i , as soon as it was available, in all the later equations. This is how Gauss-Siedel's method works.

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j < i} a_{ij}x_j^{(k+1)} - \sum_{j > i} a_{ij}x_j^{(k)} \right)$$

Eq. 4-6

Gauss-Siedel results in faster convergence to the solution of the steady state vector.

4.3. *Efficient Memory Utilisation and Computation*

The infinitesimal generator matrix Q requires a substantial amount of memory. It is an nxn matrix where n is the number of tangible states. Each element in the matrix is a double precision floating point number which in Java is 8 bytes. Even if there are just 1000 tangible states, this would result in a matrix which is approximately 8Mbytes in size! The matrix would become impossible to hold in memory even for a relatively small number of states. However, a useful

observation can be made about the infinitesimal generator matrix Q . It is a sparse matrix i.e. most of its elements are zero. Given that this is the case, there is actually no need for an $n \times n$ matrix, we only need to hold non-zero entries. PIPE2 uses the sparse matrix scheme illustrated in Figure 4-1. It also incorporates a further observation; the iterative methods outlined above require *column* access to the matrix Q due to the matrix being transposed before solving for the steady state. In the scheme used by PIPE2 an array of linked lists is used where each list represents a column of matrix Q . Each list contains nodes which record the row they represent, and the value for that row. Only non-zero elements are placed in a list.

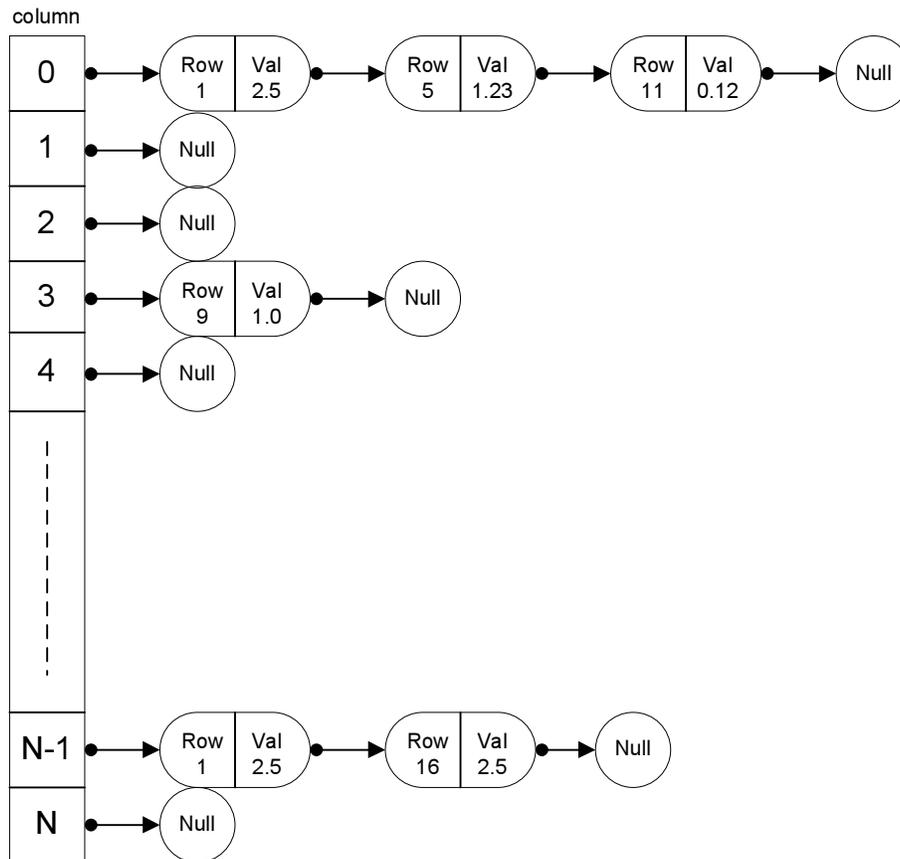


Figure 4-1 The sparse matrix scheme used in PIPE2

Direct methods can make using this scheme awkward as they modify the matrix as they progress. Iterative methods suffer from no such problems as they do not change the matrix Q or A as it has now become. They only modify the solution vector x which is a normal matrix. An additional benefit of the sparse matrix scheme with iterative methods is that there are no longer any computations on elements which are actually zero. This results in more efficient computation of the end result.

PIPE2 uses Gauss-Siedel approximation for solving the steady state solution. To decide whether the values in the steady state vector have converged the following checks are made:

- The residuals $|-Q^T x|$ are checked to see how close to zero they are. The closer they are, the more accurate the result.
- All elements in the steady state vector should be ≥ 0

Once both these tests are passed, the steady state vector is normalised to 1 to meet the requirement,

$$\sum_i \pi_j = 1$$

Eq. 4-7

Further memory efficiency can be achieved by observing that many of the rates in the matrix Q will be the same. Instead of each element storing its associated rate, it could point into a table of rates. This however has not been implemented in PIPE2.

5. Results

5.1. State Space Exploration

PIPE2 fully implements the state space generation algorithm described in section 3. It was tested with examples from [BK02] amongst others.

5.1.1. A simple GSPN

This contrived GSPN was solved as an example in section 2.4. The Petri net and its reachability graph are shown once again in Figure 5-1 and Figure 5-2. Each transition has a rate of 1.0.

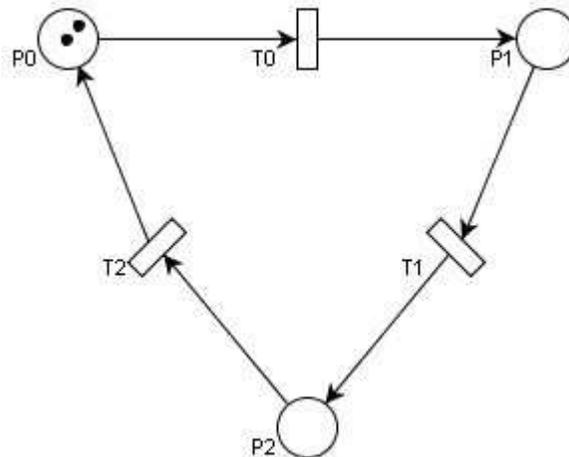


Figure 5-1 A simple GSPN

From the reachability graph we can identify all the tangible states. In fact in this particular example, all the states are tangible. PIPE2 correctly identifies the 6 tangible states as can be seen in the screenshot in Figure 5-3.

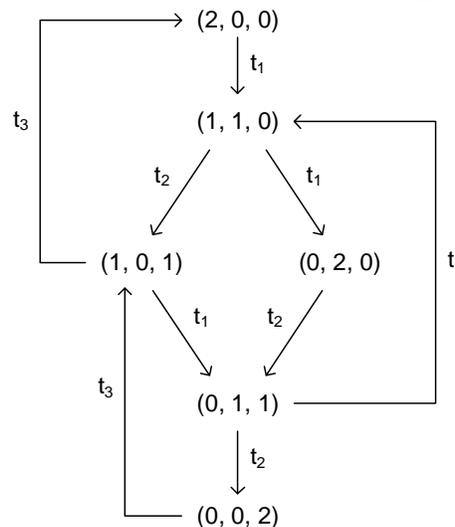


Figure 5-2 The associated reachability graph

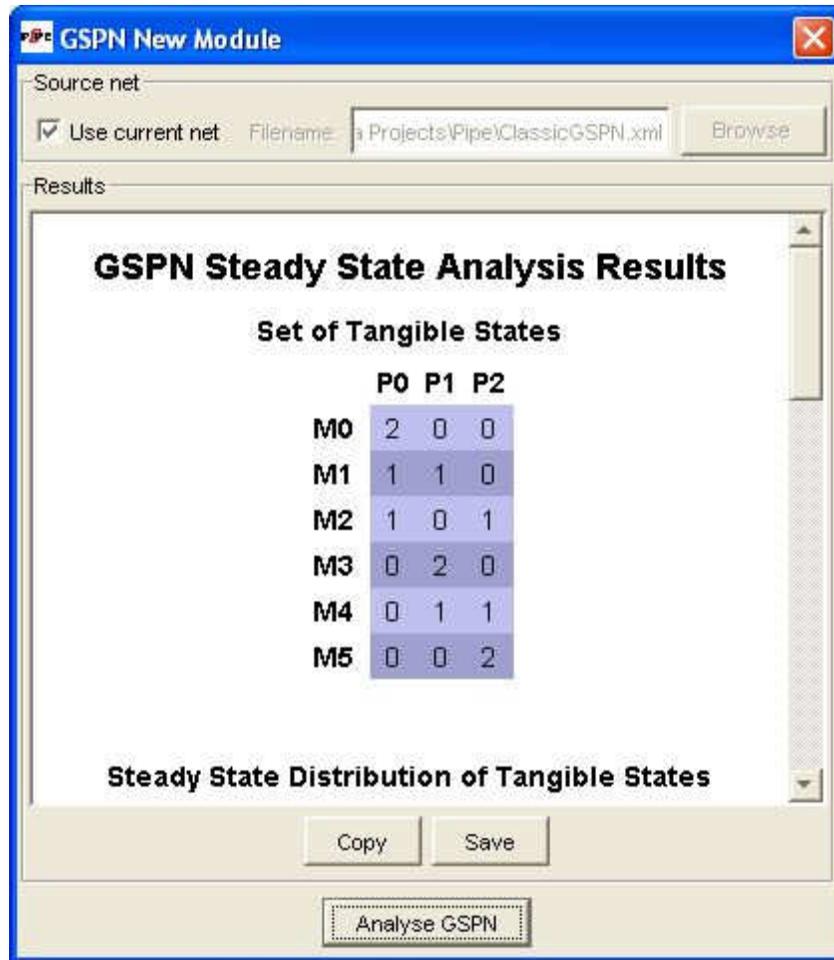


Figure 5-3 The results window from PIPE2 after analysing the GSPN of Figure 5-1

5.1.2. A GSPN with Vanishing States

This example of a GSPN is taken from [BK02]. The GSPN can be seen in Figure 5-4 with its associated reachability graph in Figure 5-5. Each transition t_i has a firing rate i . The only tangible states are M6, M7 and M8 the rest are all vanishing states.

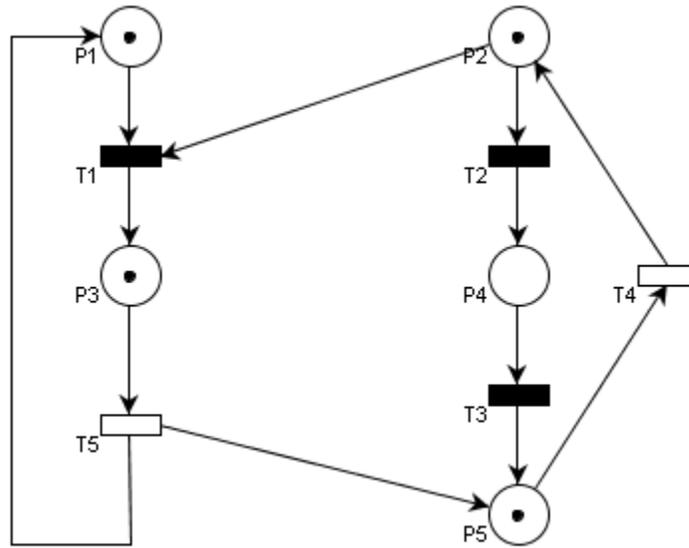


Figure 5-4 A simple GSPN with Vanishing States, transition t_i has firing rate i [BK02]

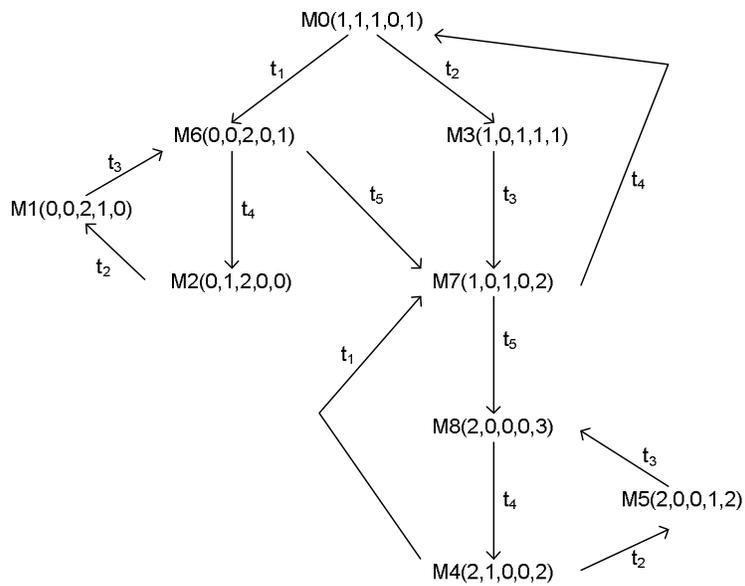


Figure 5-5 The associated reachability graph [BK02]

In Figure 5-6, we can see PIPE2 has correctly identified the three tangible states which it labels as M0, M1 and M2 and eliminated the vanishing states.

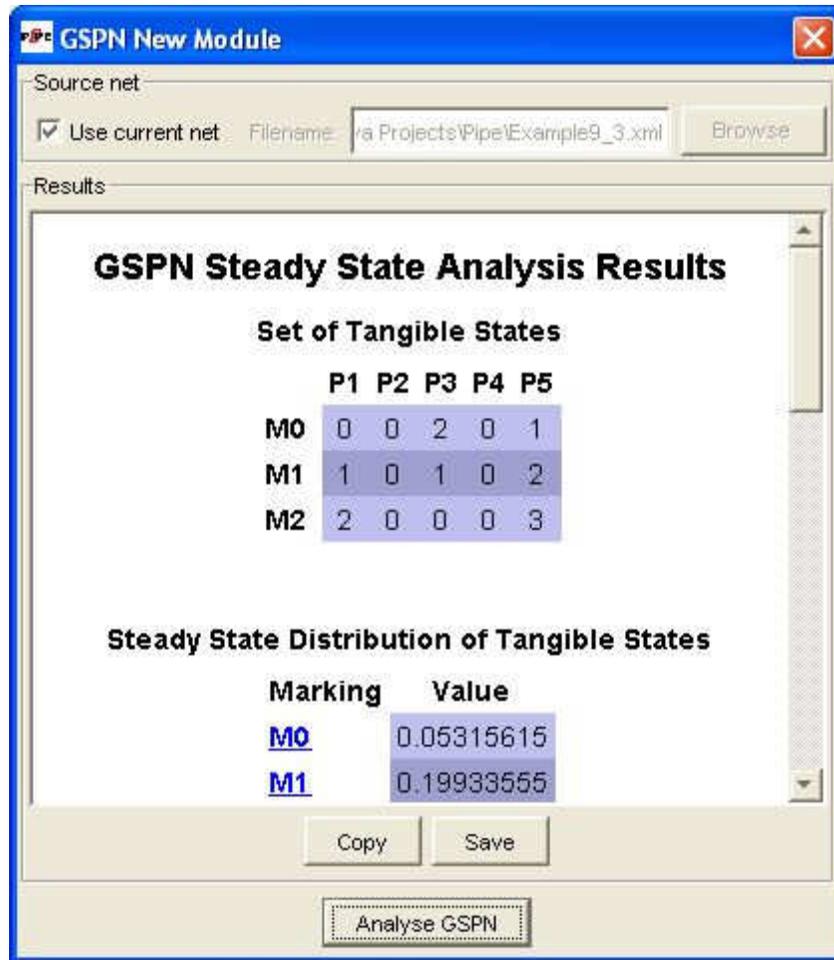


Figure 5-6 The results from PIPE2 after it has analysed the GSPN of Figure 5-4

5.1.3. The Courier Protocol Software GSPN

A more demanding example, Figure 5-7 was also used to test PIPE2. [Kno99] gives details of a GSPN which models the software used in the Courier telecommunications protocol. The GSPN models the ISO application, session and transport network layers from a sender, p1 to p26, to a receiver, p27 to p46. This is a large GSPN with 45 places. It is an excellent model to test PIPE2 with as there are two important parameters in the transport layer which can be used to vary the size of the underlying reachability graph. The first parameter is the sliding window size n (p14), the second parameter is the transport space m (p17). For the purpose of testing PIPE2, m was kept constant at 1 and n was varied. The transition rates are obtained from [Kno99] and are shown in Table 5-1.

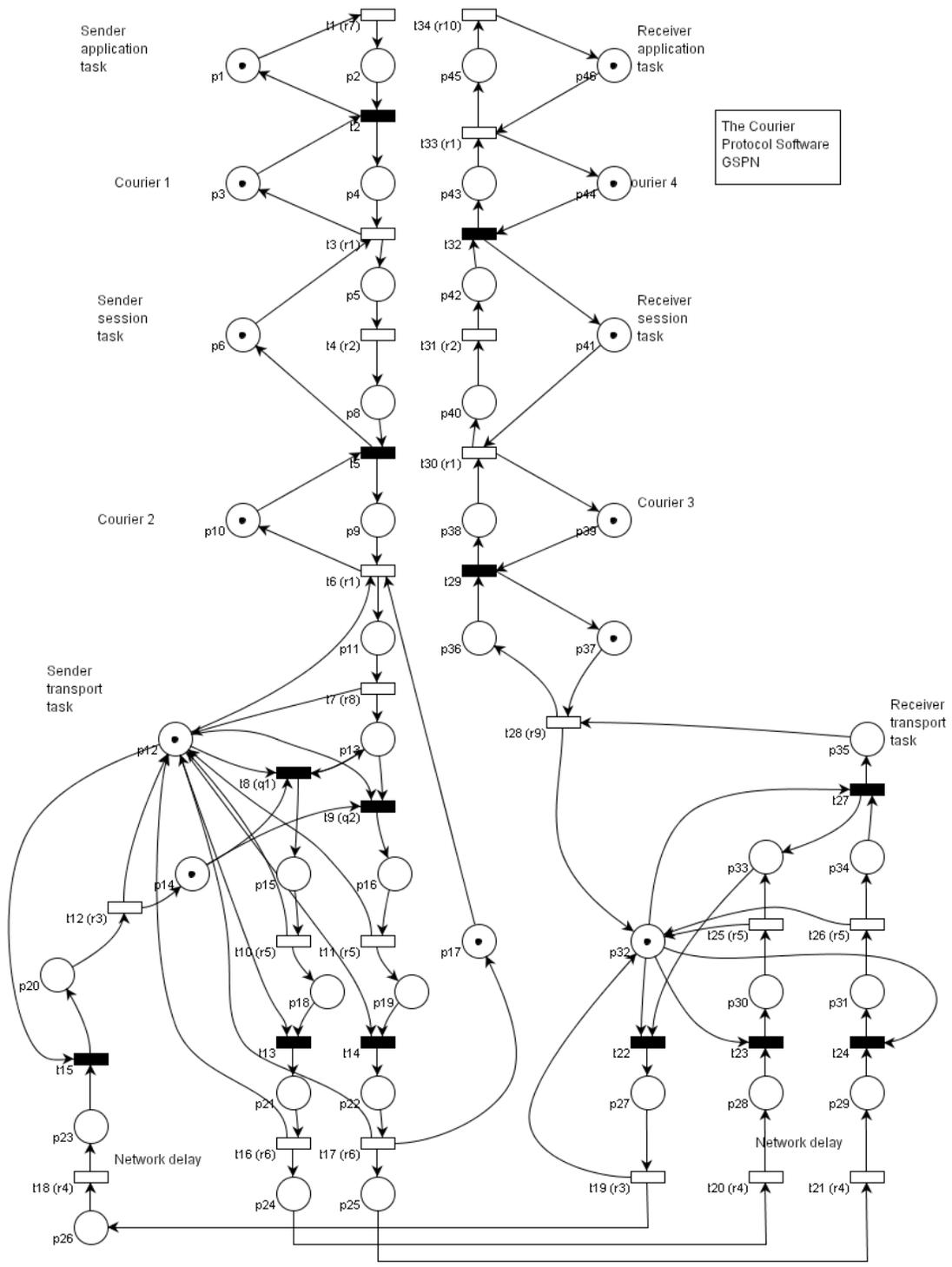


Figure 5-7 The Courier Protocol GSPN Model

Timed Transition Rate	Value
r1	5000/0.57
r2	5000/4.97
r3	5000/1.09
r4	5000/10.37
r5	5000/4.29
r6	5000/0.39
r7	5000/0.68
r8	5000/2.88
r9	5000/3.45
r10	5000/1.25
Immediate Transition Weight	Value
q1 vs. q2	1.0 vs. 1.0

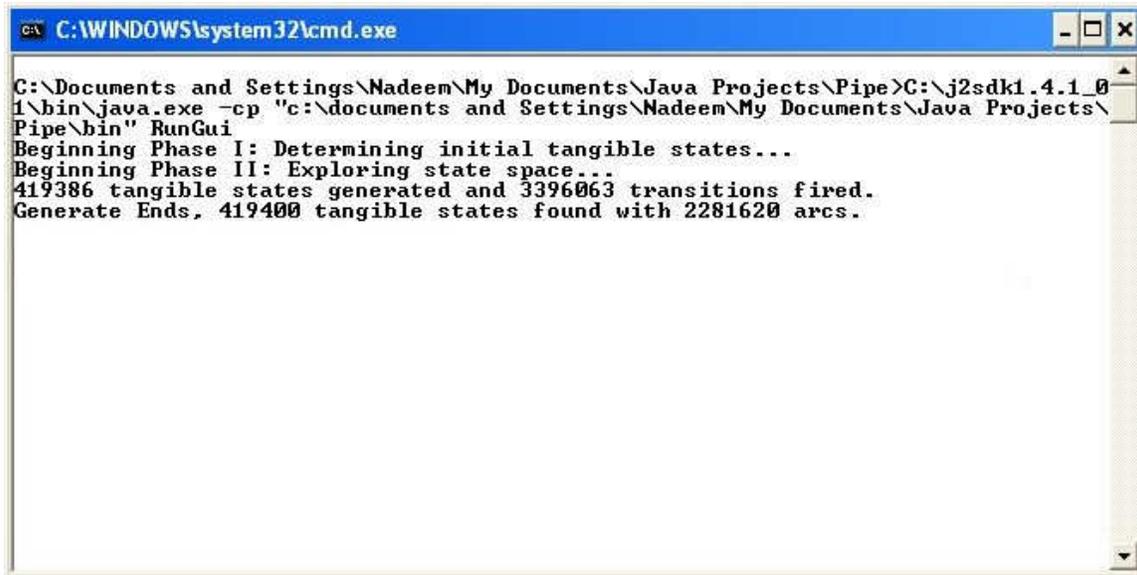
Table 5-1 Transition rates and weights in the Courier Protocol GSPN model [Kno99]

Increasing the number of tokens on p14 (i.e. increasing n) will result an increase in the number of states in the reachability graph and hence an increase in the number of tangible states as in Table 5-2.

Tokens on p14 (n)	Number of Tangible States	Number of Arcs in Reachability Graph
1	11,700	48,330
2	84,600	410,160
3	419,400	2,281,620
4	1,632,600	9,732,330

Table 5-2 The number of tangible states and arcs in the reachability graph for varying n [Kno96]

PIPE2 was tested with increasing values of n to see if it correctly determined all the tangible states and found the correct number of arcs between them. The results are only summarised when there are more than a certain number of tangible states so individual tangible states are not shown. In each case, PIPE found the correct number of tangible states and arcs between them.



```
C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\Nadeem\My Documents\Java Projects\Pipe>C:\j2sdk1.4.1_01\bin\java.exe -cp "c:\documents and settings\Nadeem\My Documents\Java Projects\Pipe\bin" RunGui
Beginning Phase I: Determining initial tangible states...
Beginning Phase II: Exploring state space...
419386 tangible states generated and 3396063 transitions fired.
Generate Ends, 419400 tangible states found with 2281620 arcs.
```

Figure 5-8 Feedback from PIPE2 in the console window after exploring the state space for the Courier Protocol GSPN model with $n = 3$

5.2. Steady State Solution

5.2.1. A Simple GSPN

For testing PIPE2's steady state analysis capabilities, the same examples as from section 5.1 have been used. The first is the simple contrived example GSPN first described in section 2.4. We saw that the steady state distribution for this GSPN was,

$$\pi = \left(\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6} \right)$$

Figure 5-9 shows the results window from PIPE2 after analysis of this GSPN has completed and it can be seen to have determined the steady state distribution correctly.

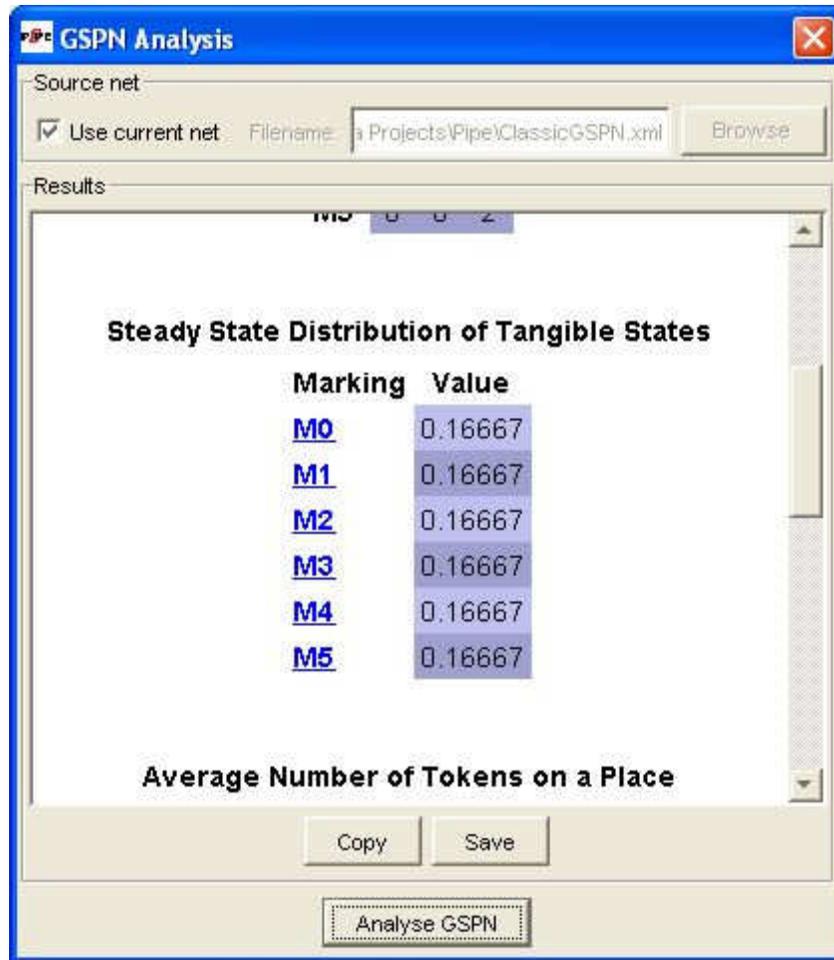


Figure 5-9 The results window from PIPE2 after analysing the GSPN of Figure 5-1

5.2.2. A GSPN with Vanishing States

This GSPN has been described in the previous section and is taken from [BK02]. The correct steady state distribution for this GSPN is given as,

$$\pi = \left(\frac{16}{301}, \frac{60}{301}, \frac{225}{301} \right)$$

In Figure 5-10 we can see that PIPE2 has in fact calculated the correct steady state distribution even for a GSPN where it has eliminated vanishing states 'on the fly'. This indicates that the algorithm for replacing clusters of vanishing states with an effective transition rate between the tangible states is correct.

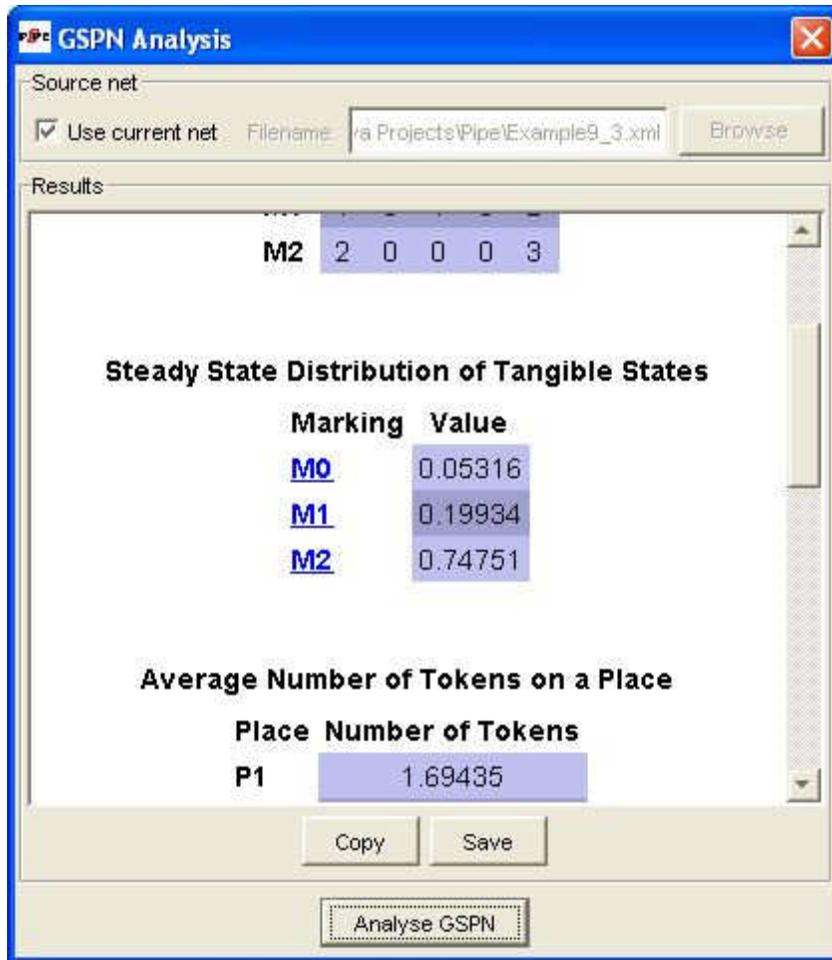


Figure 5-10 The results from PIPE2 after it has analysed the GSPN of Figure 5-4

[BK02] also calculates the transition throughputs. This is a capability also present in the GSPN analysis module in PIPE2 although PIPE2 can only calculate throughput for timed transitions due to data for vanishing states being replaced by effective transition rates. The results are compared in Table 5-3.

Timed Transition	Published Results	Calculated results
t4	4.0	4.0
t5	380/301	1.26246

Table 5-3 Transition throughputs for the GSPN of Figure 5-4

5.2.3. The Courier Protocol Software GSPN

The courier protocol GSPN has been described in the previous section and is once again useful for testing PIPE due to the scalability of the underlying Markov chain and availability of published performance data. Due to the large number of tangible states even in the case where the sliding window size n is 1 (the number of tokens on p_{14}), PIPE2 does not display the steady state distribution. Instead, only the statistical data calculated using the steady state distribution is displayed. PIPE2 calculates the average number of tokens on a place, the token probability

density i.e. the probability of there being π_i token's on place i , and also the throughput of timed transitions. [Kno99] re-prints performance measures first published by Woodside and Li in [WL91]. Woodside and Li suggest the following performance measures:

- λ - the data throughput rate which is given by the throughput of transition t_{21} ,
- The following task utilisation parameters
 - P_{transp1} , the probability that transport task 1 is idle i.e. $P[\pi_{12}=0]$
 - Similarly P_{transp2} , i.e. $P[\pi_{32}=0]$,
 - P_{sess1} , P_{sess2} for p_6 and p_{41}
 - and finally P_{send} , P_{recv} for p_1 and p_{46}

The results can be seen in Table 5-5 and can be seen to agree exactly. The results are only calculated up to $n=2$ as although PIPE2 was able to generate the tangible states and reachability graph, there was insufficient memory to hold the infinitesimal generator matrix for calculating the steady state despite the memory efficient sparse matrix scheme used. This is a known problem and more advanced Petri net analysers such as DNAmaca use disk based schemes for holding large matrices.

	n=1	n=2	n=3	n=4	n=5	n=6
λ	74.3467	120.372	150.794	172.011	187.413	198.919
P_{send}	0.01011	0.01637	0.02051	0.02334	0.02549	0.02705
P_{recv}	0.98141	0.96991	0.96230	0.95700	0.95315	0.95027
P_{sess1}	0.00848	0.01372	0.01719	0.01961	0.02137	0.02268
P_{sess2}	0.92610	0.88029	0.84998	0.82883	0.81345	0.80197
P_{transp1}	0.78558	0.65285	0.56511	0.50392	0.45950	0.42632
P_{transp2}	0.78871	0.65790	0.57138	0.51084	0.46673	0.43365

Table 5-4 Published performance measures for the Courier Protocol Software

	n=1	n=2
λ	74.3467	120.372
P_{send}	0.01011	0.01637
P_{recv}	0.98141	0.96991
P_{sess1}	0.00848	0.01372
P_{sess2}	0.92610	0.88029
P_{transp1}	0.78558	0.65285
P_{transp2}	0.78871	0.65790

Table 5-5 Computed performance measures for the Courier Protocol Software

6. Other Work

6.1. *Bug Fixing*

There were a number of small but serious bugs with PIPE2 before work began on this project. These had to be fixed before any other improvements could be made to PIPE2. This meant understanding the work written by two previous group project teams much of which was uncommented in any way. There was a large amount of redundant and inefficient code which also had to be scrutinised carefully to find the source of bugs.

6.1.1. File Save Bug

The first serious bug present in PIPE2 would manifest itself when saving files. If the option 'Save As...' was chosen, files would be saved correctly under the chosen name. If an existing file was to be saved using the same filename, the user would correctly choose the option 'Save' rather than 'Save As...', however the file would now not be guaranteed to save. This obviously was a serious problem. The bug was eventually tracked down as being an unnecessary and also incorrect test in an 'if' statement. File saves now take place whenever the option 'Save' is selected rather than being a game of Russian roulette.

6.1.2. Edit Arc Weight Bug

Editing arc weights would result in an error message claiming that a number had not been entered. This was nonsense and on repeating exactly the same steps a second time, no error would occur and the arc weight would change correctly. The bug was actually due to objects being added to the window container multiple times instead of just once but the exception handling was so poor that all exceptions were given the same error message. Once exception handling was improved, it was possible to track down the source of the bug and fix it.

6.1.3. Invariant Analysis Bug

One of the pre-existing Petri net analysis modules is 'Invariant Analysis'. This is an analysis which can be carried out on basic Petri nets and as the name suggests, it determines the invariants of a Petri net.

There were two issues with the invariant analysis module. The first was that the identifiers used in the display window for places and transitions did not then get used when reporting back the invariant equations making it impossible to know which places and transitions the invariant equations were referring to. This can be seen in Figure 6-1 and Figure 6-2. The P-Invariant equations in the results window refer to places such as p0 but there is no such place in the place-transition net!

The second issue with the invariant analysis module was that once a Petri net was loaded, any changes to the net were not reflected by changes in the invariant analysis! The file would have to be saved and reloaded for any changes to be recognised by the analysis routine.

Both the bugs in the analysis module have now been rectified. Results now use the same labels as are present on the Petri net itself and analysis now takes place on the net as it is currently without any file save and reload needing to take place.

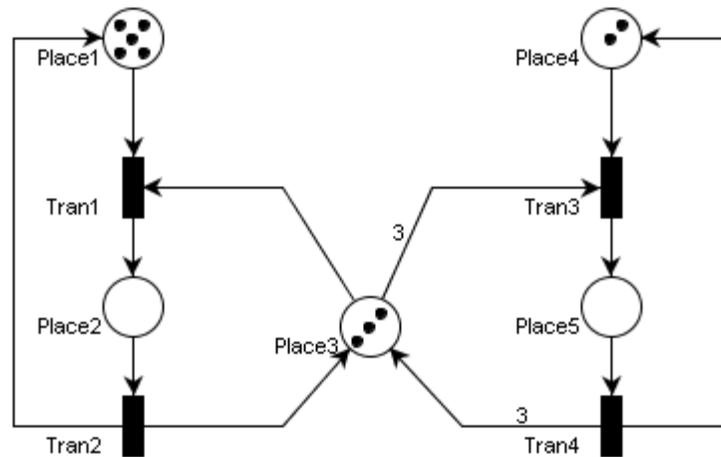


Figure 6-1 A simple Place-transition net

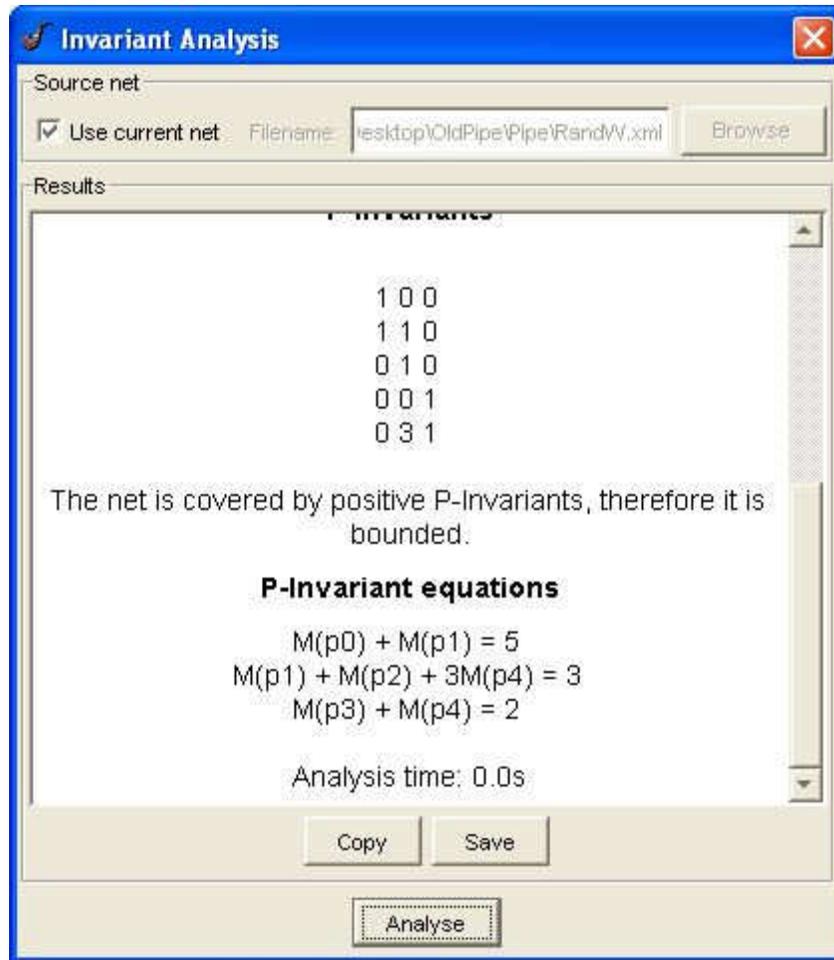


Figure 6-2 The results from invariant analysis of the Place-transition net in Figure 6-1

6.1.4. HTML File Save Formatting

Any analysis modules added by the 2004 group project team use HTML formatting in the results window. This has greatly improved the look of the analysis modules. An additional feature is the ability to save results as an HTML file for later use. This is an excellent addition to PIPE2 but unfortunately, it has a bug which results in all the HTML style information being lost when writing to a file. This actually appears to be a bug in the Java class JEditorPane. It has a method 'getText()' which should return all the text in a JEditorPane window. However, from some reason it does not return any of the text between the <Head> tags even though it returns the tags themselves. As this is where the style information resides, all the style information is lost. This has had to be fixed with a workaround which adds all the style information in again before writing to a file.

6.1.5. Code Optimisation

Whilst reading through the code, a number of areas where code could be optimised were identified. As an example, on many occasions, 'for' loops were found to be very inefficiently programmed. In general they took the form of Figure 6-3.

```
for(int i, i < x, i++){  
    a = method_b(i)  
    if(a == true){  
        ...  
    }  
}
```

Figure 6-3 Example area of code suitable for optimisation

At first this might seem a perfectly reasonable piece of code. The problem actually was down to the method labelled as `method_b()`. This method would carry out a long complex computation and create an entire Boolean array of results. It would then return just one value from this array and discard the rest. The rest of the for loop would then test this result. However, each time around the loop, `method_b()` would have to recalculate *the same array* which could require a few thousand calculations. The loop was re-organised to take `method_b()` out of the loop, it would be calculated just once and the *entire array returned* for testing within the loop. This type of code was found in numerous places. Other 'interesting' programming practices were to have multiple methods all carrying out *exactly* the same calculations but just returning a different step in the process. These were combined into a single method with a simple clear 'if' statement to decide which step in the process should return a result. Most code was found to be completely uncommented. Whenever it was possible, comments were added in to make it easier for future developers to work on the code.

6.2. User Interface Enhancements

A large amount of time was spent on user interface enhancements. These are centred on the arc drawing tool. Whilst this functions correctly, it was lacking features which would make drawing Petri nets an easier process. When drawing an arc between a place and a transition or vice versa, the user would need to have decided before-hand how many points there would need to be on an arc for later manipulation. For example, if the user were to draw an arc straight from a place to a transition and then decide that it needed to be drawn via a different route, the user would only have the option of deleting the arc and starting again. If it was found that further manipulation points were needed, once again the arc would have to be deleted and started again. In Figure 6-4, we see a Petri net as it has been originally drawn. The user has decided to change the path of the arc from T1 to P0 to pass around the outside of the net however there are only two

manipulation points on the arc, the start and end point. These cannot be moved and the only other options for that arc are shown in the pop-up menu to be 'delete' the arc or edit the weight of the arc. As such the only choice for the user is to delete the arc and start again. Even if the user had had the foresight to draw the arc using 3 points, there would be no options for adding more points if needed. Whilst this may not seem like a big issue when drawing a net like that of Figure 6-4, it will clearly hinder progress when drawing a net like that of the courier protocol GSPN model shown previously in Figure 5-7.

There are now two features which have been added to the arc drawing tool. One allows a point on the arc to be split into two points. The second feature splits a segment of an arc into two at the halfway point. Both these feature's work whether the arc is currently straight or curved and appear in the pop-menu in the correct context i.e. if the user selects an arc segment then the 'Split Arc Segment' menu option appears as shown in Figure 6-5.

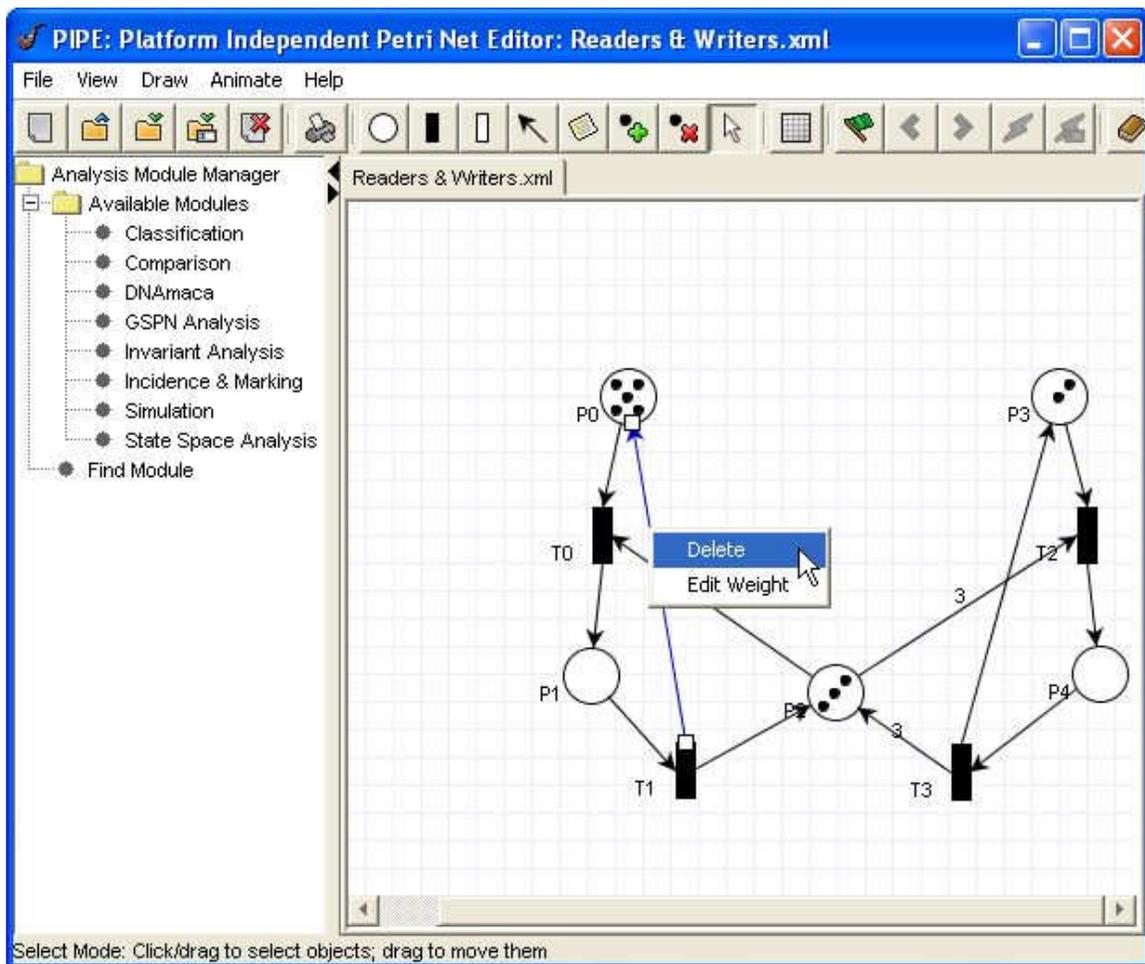


Figure 6-4 The user has selected the arc highlighted in blue. The squares are the available manipulation points.

Arcs in PIPE2 are surprisingly complex. Three classes are involved, 'Arc', 'ArcPath', and 'ArcPathPoint'. An instance of Arc has an associated ArcPath object. An ArcPath object holds a list of 'ArcPathPoint' objects. The split point feature involved identifying the currently selected ArcPathPoint object, duplicating it as another instance, offsetting the co-ordinates of the new instance slightly, and then adding it in at the appropriate point in the list of points held by the ArcPath object. The window then had to be notified to re-draw the arc so the new point would appear.

Splitting an arc segment is a more complex task. Unfortunately, the way arcs are structured in PIPE2 is perhaps not the best way. It would possibly have been better to have a class called ArcSegment. This class would then hold two ArcPathPoint instances for the start and end point of a segment. The ArcPath class would then hold a list of ArcSegment objects. There are many advantages to structuring arcs in this manner. First of all, Java's CubicCurve2D class in package java.awt.geom could then have been used to calculate the Bezier curves for curved arcs. Currently, one of the Arc classes has been written to actually calculate Bezier curves itself.

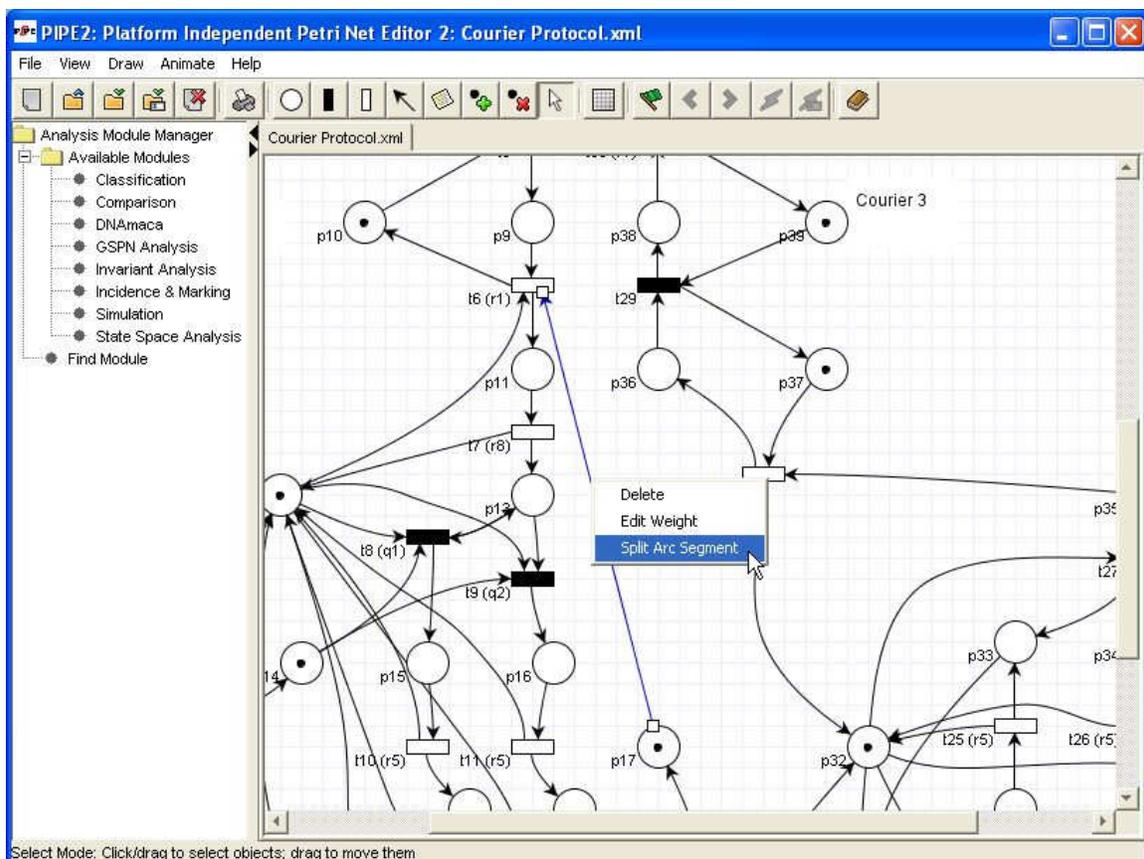


Figure 6-5 The user can now split an arc segment in to two

A second benefit of this alternative structure is that arc segments could then be selected by a user as actual objects in a net just as the complete arc and ArcPathPoints are. Once this could be done, splitting an arc segment would be the same process as splitting an arc path point.

This restructuring of the classes making an arc in a Petri net model was attempted. Unfortunately, it was found that the classes involved in arcs were closely coupled to the classes Place and Transition! The reasons for this are that arcs enter or leave places and transitions at specific angles. Arcs always leave places radially outwards and enter and leave transitions at an angle determined by how many other arcs are entering or leaving that edge of a transition. Whilst this is correct, the previous group project team have made design decisions which have resulted in the place or transition classes actually manipulating arcs instead of providing arc objects with information that allows them to do the work themselves. It very soon became clear that to change the structure of arcs would require a major re-write of too much code for the time available.

An alternative method had to be found to enable a 'Split Arc Segment' feature to be added. A brute force approach was used. When the user clicks on an arc, it is possible to get the mouse co-ordinates. If the user then chooses to split an arc segment, all the points on the arc are compared as pairs in sequence. Each pair has a straight line calculated between them and then the distance from the mouse co-ordinates to the mid-point of each line is calculated. The line which results in the shortest distance to the mouse co-ordinates is then considered to have the correct two points bounding the segment the user wishes to split.

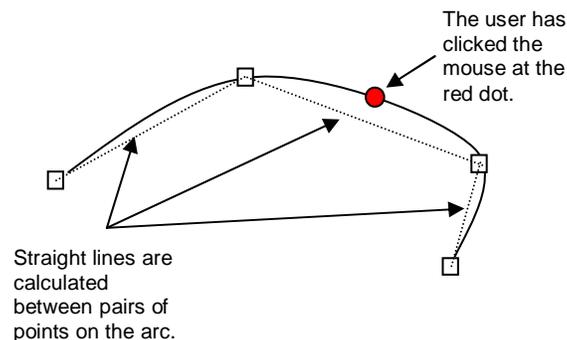


Figure 6-6 The process of splitting an arc segment

Now an extra ArcPathPoint object is created at the midpoint between the points for the selected segment and is added to the ArcPath using the same process as described previously for splitting an ArcPathPoint. This process works very well and is now implemented in PIPE2.

6.3. Product Marketing

In addition to the original requirements, a new website has been created to market PIPE2. From the website it is possible to download the latest release of PIPE2 and also read about the software and Petri nets in general. It can be found at:

<http://pipe2.sourceforge.net/>

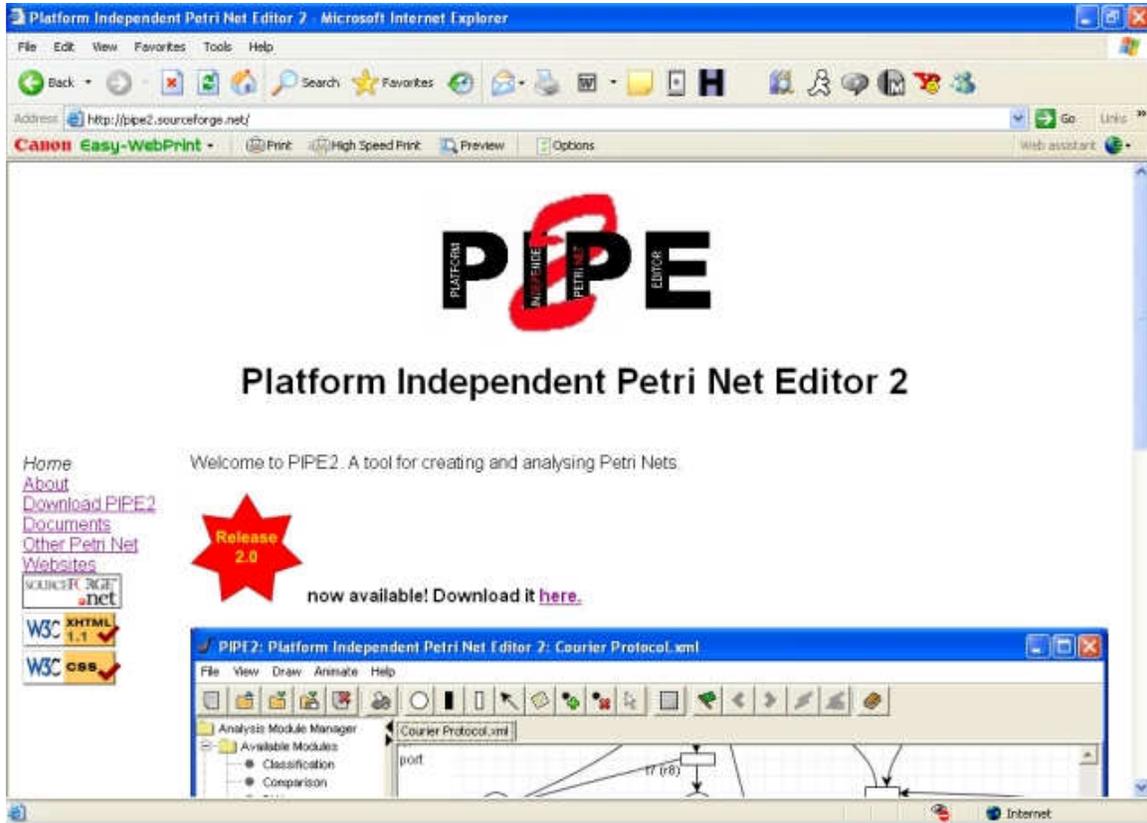


Figure 6-7 The PIPE2 Website

7. Conclusion & Future Work

7.1. Summary

This project has been very successful in meeting all its aims and more. The initial tasks for the project were:

- Fix a number of moderately serious bugs in the user interface
 - Saving files did not always work
 - The invariant analysis module did not give the correct analysis results and did not use the correct labels for the results
 - Changing arc weights resulted in an error which was nonsensical
 - Other as yet unknown bugs that may be found along the way
- Improve the arc editing tool. The options for editing an arc once it was drawn were limited and consisted of being able to delete an arc or change its weight.
- Improve the GSPN analysis capability. The existing capability was so severely limited that it was useless and all analysis of any real models had to be handed off to DNAmaca.

First of all, the bugs listed were found and eradicated. A number of additional bugs were found as work progressed and were removed. At the same time, code was optimised wherever possible and comments were added in to make it easier for future developers to work on PIPE2.

The arc editing tool is now much easier to use. Complex Petri nets such as that for the courier protocol software model are now much easier to draw. The difficulty now lays in the complexity of the model itself and not the tools for drawing the model.

The most important improvement to PIPE2 however is in the GSPN analysis capabilities. PIPE2 was capable of modelling and analysing Generalised Stochastic Petri nets. However, its analysis capability was very severely limited. It could handle at most 1000 states. Not only could it only manage a small number of states, it would incorrectly reject valid GSPN's as being invalid. Furthermore, the analysis was very slow. The original analysis module was compared to the new module for a relatively small model containing just 53 states. The original analysis module took 2minutes 30seconds, PIPE2 can now analyse the same model in 1.5 seconds, 1% of the time it used to take! Also, the number of states the module can deal with is greater than 1million. On this front however there has been some disappointment. Although the state space exploration can cope with over 1million states, the steady state analysis can't keep up. Using the courier protocol GSPN model as an example, scaling the model to 419,400 states leads to over 2.2million arcs in the state graph. Even using the memory efficient sparse matrix scheme, this becomes unwieldy and

memory soon cannot be allocated to hold the whole matrix. This is a problem which has been studied before and schemes exist for coping with this.

In addition to the original project requirements, a website has been created for marketing PIPE2.

7.2. Future Work

There are still a number of ways PIPE2 could be improved which should provide very interesting future software projects. The most obvious extension would be to add a disk based scheme for holding the sparse infinitesimal generator matrix.

Another very useful extension to PIPE2 would be the capability to have token dependent transition rates. This would require users to be able to enter expressions like ' $\#(P1)$ ' meaning the rate is the same as the number of tokens on place P1 or ' $\min(x,y)$ ' meaning the rate is the minimum of x or y where x and y could also be expressions, for every transition.

The way results are presented could also be made to depend on expressions. Instead of just displaying statistics such as token probability distribution or transition throughput, perhaps the user could request that these results are used to calculate further performance statistics they specify by being able to enter an expression along the lines of:

$$x = 400*\text{Throughput}(t1) + 600*\text{Throughput}(t2)$$

The results window would then automatically display the value of x.

These should all be interesting and challenging projects.

Appendix A – User Guide

The following pages provide a brief guide to installing PIPE2 and also using the GSPN analysis module.

Installation

PIPE2 can be downloaded from the following location:

<http://pipe2.sourceforge.net>

The file is a compressed 'zip' file and has to be extracted using an appropriate utility for your operating system. On Microsoft Windows based systems, the commercial tool 'WinZip' can be used although there are also free tools available. On Linux and other Unix based systems the command line tool 'unzip' can be used. Once the files have all been extracted, PIPE2 can be started from the command line as follows:

On Linux: `java -cp ~/yourpath/pipe/bin RunGui`

On Windows: `java -cp "yourpath\pipe\bin" RunGui`

where '*yourpath*' is replaced by the path to the extracted files that you specified during the extraction process. The above assumes you have Java installed, this can be obtained from Sun Microsystems at the website:

<http://java.com>

Analysing a GSPN

Some example Petri nets are included with PIPE2. Select 'Example Nets' from the 'File' menu as in Figure A-1. Select the Petri net 'Classic GSPN'. This will now open in the main window of PIPE2 and should look like Figure A-2. Now double click on 'GSPN Analysis' in the analysis module manager. In a few seconds, a results window should appear and your GSPN analysis is complete. For more complex models that take longer to analyse, progress can be checked by looking at the console window where continuous feedback is provided. Analysis results can be saved for future reference by selecting 'Save' in the results window.

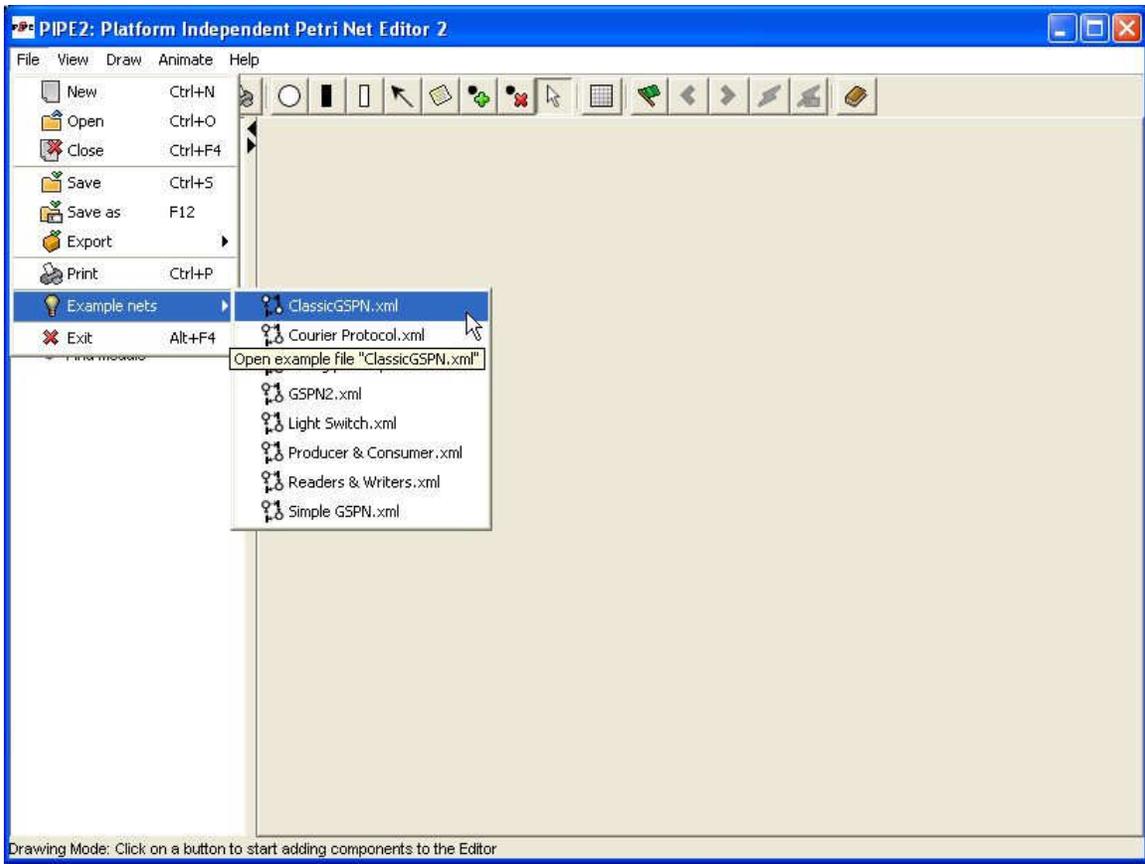


Figure A-1 Choosing an example Petri net

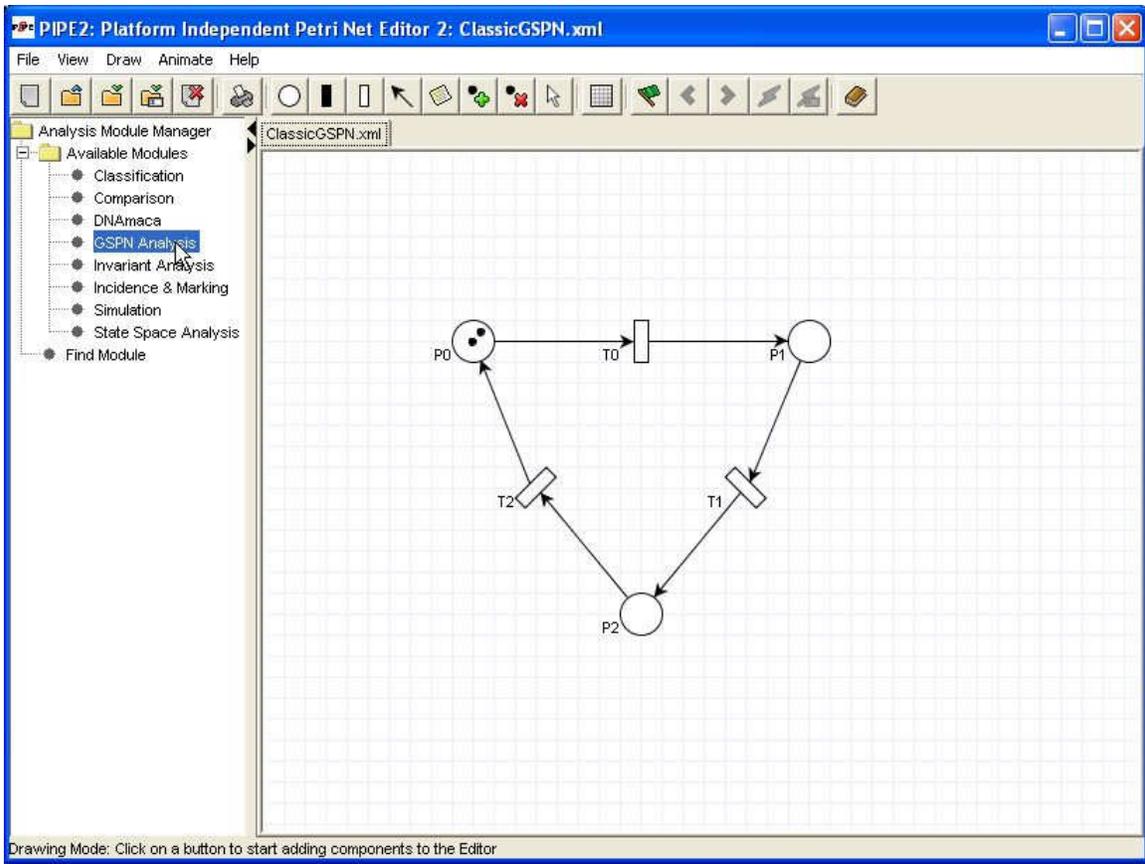


Figure A-2 Selecting the GSPN Analysis module

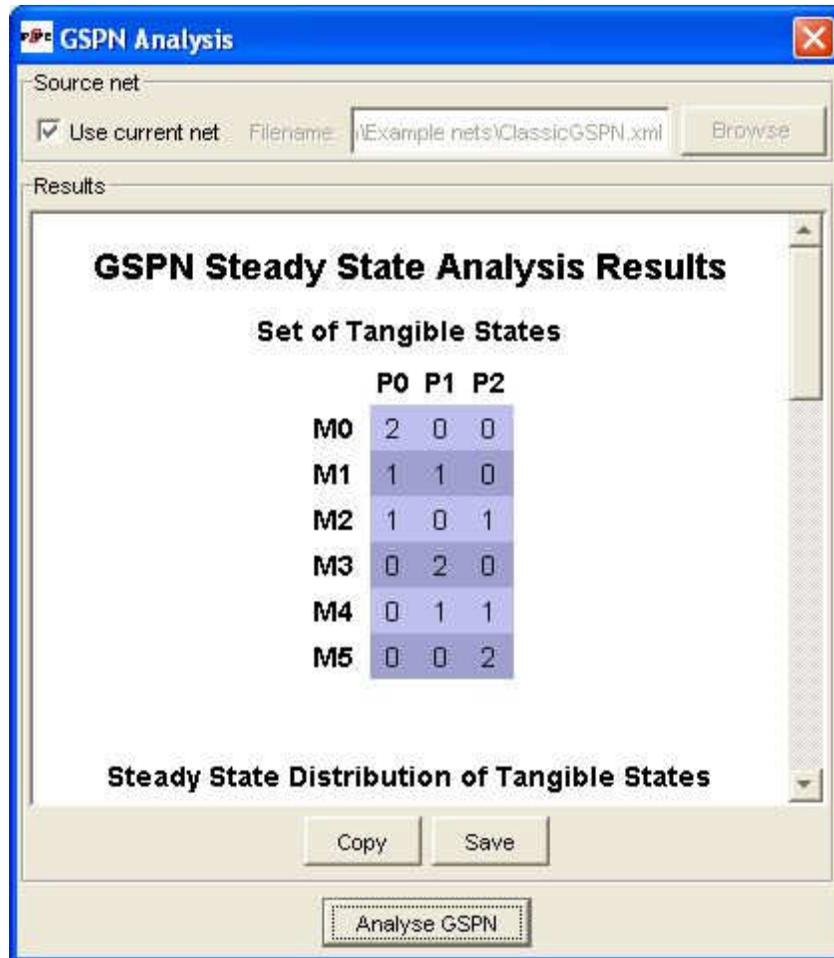


Figure A-3 The analysis results window

Bibliography

[BC+03] **J. Bloom, C. Clark et al** *Platform Independent Petri Net Editor* Group Project Final Report, University of London, Imperial College of Science Technology and Medicine, March 2003

[BC+04] **T. Barnwell, M. Camacho et al** *Petri Net Analyser – Group 4* Project Final Report, University of London, Imperial College of Science Technology and Medicine, March 2004

[BK02] **F. Bause and P.S. Kritzinger** *Stochastic Petri Nets, An Introduction to the Theory 2nd Ed.* Vieweg 2002 ISBN 3-528-15535-3

[Kno96] **W.J. Knottenbelt** *Generalised Markovian Analysis of Timed Transition Systems* MSc thesis, University of Cape Town, June 1996

[Kno99] **W.J. Knottenbelt** *Parallel Performance Analysis of Large Markov Models* PhD thesis, University of London, Imperial College of Science Technology and Medicine, December 1999

[Knu98] **D.E. Knuth** *The Art of Computer Programming Volume 3: Sorting and Searching 2nd Ed.* Addison-Wesley 1998 ISBN 0-201-89685-0

[Sed98] **R. Sedgewick** *Algorithms in C, 3rd Ed.* Addison-Wesley 1998 ISBN 0-201-31452-5

[WL91] **C.M. Woodside and Y. Li** *Performance Petri Net Analysis of Communication Protocol Software by Delay Equivalent Aggregation* Proceedings of the 4th International Workshop on Petri Nets and Performance Models, pages 64-73, Melbourne Australia, 2-5 December 1991 IEEE Computer Society Press