FINAL REPORT

PETRI NET ANALYSER – GROUP 4

15 March 2004

TOM BARNWELL, MICHAEL CAMACHO, MATTHEW COOK,
MAXIM GREADY, PETER KYME, MICHAIL TSOUCHLARIS

ACKNOWLEDGEMENTS

Many thanks to Dr William Knottenbelt for his patience and support as supervisor for this project, and to Nick Dingle for his invaluable help.

TABLE OF CONTENTS

1. INTRODUCTION

The aim of this project is to enhance an existing piece of software written for a 2002/3 MSc Conversion group project, called PIPE. PIPE (Platform Independent Petri Net Editor) is a Java based editing and analysis system for Petri nets. It is to be enhanced through improved and additional functionality and by providing an extensible platform for further development.

1.1. Petri nets

Petri nets are a formalism for modelling concurrent systems, first defined by Carl Adam Petri in 1962. They allow correctness of concurrent systems to be verified using well-defined, provable mathematical techniques and allow the behaviour of a system to be expressed both graphically and algebraically. They support the natural expression of such concepts as synchronisation and communication between processes. The ability they provide to visualise the structure of a system promotes greater intuitive understanding of what is being modelled. Petri nets have since been extended and augmented with additional behaviours, most notably with the addition of time data to produce Generalised Stochastic Petri Nets (GSPNs).

The building blocks of a Petri net are places, transitions, arcs and tokens. Places model conditions or objects. Places may contain tokens, which represent the value of the condition or object. Transitions model activities, which change the value of conditions or objects. For example, firing a transition may destroy a token at one place and create a token at another place. The interconnectedness of places and transitions is represented using arcs. Each arc has one and only one source, and one and only one target. If the source is a place, the target must be a transition, and vice versa.

1.2. PIPE

The original PIPE was conceived as a program to draw, analyse and simulate Petri nets. The intention was to incorporate the many of the features of existing Petri net tools in one package and improve on their shortcomings. A module interface was provided so that external modules could be adapted to be loaded dynamically into PIPE at run time.

The tool was also designed to be fully PNML compliant so as to be compatible with existing tools. The tool was written entirely in Java so as to be fully platform independent.

The program consisted of an editing/animation pane providing basic functions to draw simple Place Transition nets and animate transition actions; and a tree view from which load analysis modules. Modules were provided for invariant, simulation, classification, and state space analysis.

The animation and editing functions were implemented through the use of a Model-View-Control (MVC) pattern where the Petri net data is stored in a “datalayer” class and the view is notified to update itself when changes are made to the data.

There were, however, some notable problems with PIPE. These included:

- The editor pane allowed the drawing of nets but provided no functionality to export the resulting diagrams in a graphical format.
- The editing interface was time consuming, unintuitive, restrictive and inelegant.
- The Invariant Analysis module (adapted from an existing tool called Predator) produced incorrect results.
- Much of the existing code had been produced with a graphical IDE and as such was difficult to read and suffered problems with portability.
- Existing code both lacked clear comments and contained large sections of commented-out code.
- Existing code was both inefficiently implemented and contained many redundant and unused functions.
- The software contained numerous bugs.

2. SPECIFICATION

There were five target areas for development:

- Existing code review and documentation, including optimisation of existing features and fixing of any bugs found.
- Extension of the program to support GSPNs, including representational and editing tools and modules for relevant analyses
- Interfacing with DNAmaca for net analysis
- Improved editing tools – improved interface and functionality for creating nets
- GUI – improved interface and functionality for working with nets

These areas were investigated in the early stages of the project in order to determine their feasibility and to produce estimates for the amount of work required. Some of the issues involved are discussed below. Throughout the project, many additional features were discussed and implemented which were not in the original specification reproduced below; these are discussed later in the report.

2.1. Code review, cleanup and documentation

Having decided to work to extend the existing work done by the 2002/3 MSc group on PIPE, there were several issues to be resolved.

First, it was necessary to become familiar with the structure and workings of their code. Preliminary investigations found that in many areas the existing code was not suitable for extension; in many cases, encapsulation was very poor, with high coupling and low cohesion. The vast majority was not documented. It was therefore decided that, as far as possible without performing a complete rewrite, the code should be reorganised, simplified and made to better follow object oriented program design principles. During this process it should be possible to supply documentation in the form of inline comments to describe what the code is doing, to better promote group work and possible later extension.

During the preliminary testing it became clear that there were some serious bugs in parts of the code; for example, the Petri net classification tool was able to classify a net such that it was a member of a subtype but not a supertype:

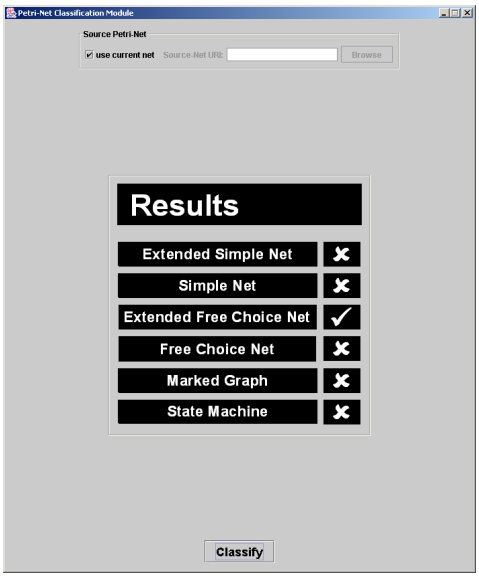


Figure 1 – EFC nets are, by definition, subtypes of ES nets

It was therefore specified that the output of existing modules should be tested for correctness and any errors fixed.

The general overview of the code was that despite starting with good intentions and full of good OO design principles, it had degenerated along the way to become littered with unused functions and unnecessary dependencies between modules, with object references being passed from class to class. The task of cleanup

was therefore a substantial part of the project, but hopefully one that can enable it to be further extended in the future.

2.2. GSPN functionality

Generalised Stochastic Petri nets are an extension of ordinary Petri nets with the addition of timing, to better model real-life systems. They are very widely used and it was considered a basic specification that PIPE should be extended to support these through the support of timed and untimed (immediate) transitions. This was expected to include extending the DataLayer class with additional structures to store the information, and adding appropriate graphical representations to the GUI. This can then provide the basis for a module providing GSPN analysis such as determining the reachability set and transition throughput.

2.3. DNAmaca interfacing

DNAmaca is a Markov chain specification, generation and solution tool, which can be used to perform analysis of GSPNs. It is capable of performing passage time analysis on both semi-Markovian and Markovian models, although our module focuses on the Markovian solver. The results produced by this module are passage time analysis statistics for the current active net, which represent the distribution of how long it takes for a system to go from one state to another. The user enters parameters for their chosen source and target states (such as a list of values representing the number of tokens contained in specified places), and the DNAmaca analysis returns a dataset representing the distribution of times taken for the system to get from the source state to the target state. PIPE should be able to interface with DNAmaca for it to perform analysis upon the Petri net; it should wrap the interface in informative GUI code and should display the returned results to the user.

2.4. Editing tools

No matter how well the analysis is implemented, without an interface allowing nets to be easily drawn and edited, there will be a barrier to the program's use. Through initial testing of the existing version of PIPE, it became clear that there was a lot to be done in this area. There were some barriers to easy net creation; it was therefore specified that creation of nets should be made as easy as possible with minimal restrictions on what the user could do. Proposed inclusions were:

- The ability to select and move objects, singly and in groups
- The ability to align objects linearly
- Enhanced contextual menus
- Contextual controls, for example for token manipulation
- Keyboard and mouse shortcuts for creation and editing of Petri net objects

One of the primary aims was to allow the creation of nets in a style similar to that shown in related books and journal articles, to remove the need for authors to draw the same net twice, first in an analyser and then in a drawing program. This implies the following additional features:

- Segmented arcs – arcs with corners
- Editable curved arcs as an extension to segmented arcs
- The ability to align objects linearly
- Arbitrary labels

2.5. GUI

The following features were specified as targets for inclusion in the project:

- The ability to save nets to graphical file formats such as Postscript and PNG for incorporation into documents, lecture slides etc.
- The ability to print nets
- Animation for transitions firing, specifying the firing delay and number of transitions to animate over
- Alternative selectable component styles, including arrowhead placement and style and place/transition fill/colouring
- Inline help
- Functionality to illustrate different analysis modules

It appears that almost all of the existing GUI code was created using a graphical editor. Unfortunately, this resulted in code that is neither efficient nor readable, and due to the extent to which options were changed from their default values, the cross-platform performance of the program was severely degraded. It was therefore specified that cross-platform performance should be remedied through reimplementation of the code using standard Swing components, and also to provide a consistent interface throughout the program and its modules.

3. METHODOLOGY

3.1. Structure

Since the project is based on existing code, there is less work to be done on structural design and code division. The basic structure of the existing code is:

- DataLayer
  - Code dealing with handling of Petri net data, including loading and saving and all calculations
  - Classes representing Petri net objects
- GUI
  - GuiView: container for graphical display of nets, handling editing events through additional classes
  - All other GUI related items and functionality
- Modules
  - Self-contained modules for analysis of net, using DataLayer functionality

It was unfeasible to reorganise the structure, so this was maintained; there is reasonable code separation along MVC lines, although the editing interface code is intermingled with other GUI code within the `gui` code branch. The level of coupling between classes was reduced wherever possible.

3.2. Coding environment

It was initially decided that a consistent coding environment across the group would be best to enable easier transfer of code and project specifications. PIPE was originally written using Borland's JBuilder, which is proprietary commercial software unavailable to the group. Of the available free IDEs, the open-source Eclipse project[1] was chosen because:

- It is free
- It is well supported under Windows, Linux and Mac OS
- It has a highly mature Java environment
- It has native support for CVS
- It was easily able to import the existing code tree and compile it

Concurrent Version System (CVS) support was especially important because PIPE is already published on SourceForge[2], a popular Open Source community website, with a CVS repository and some useful issue tracking facilities. CVS enables several people to work on the code at the same time, and then to "commit" their changes to the repository, which determines if there are any potential conflicts. Eclipse extends this by allowing line-by-line resolution of conflicting source changes and easy access to all CVS features such as previous revision comparisons and committal annotations. After some initial difficulties gaining contact with the existing project administrator and issues with SourceForge's CVS system, this proved to be a valuable resource, also allowing group members to work on the code from their home computers as well as University machines.

3.3. Overall GUI

3.3.1. Look and feel

Java 2 includes a large library of advanced GUI-related objects under the Swing code tree. They all feature large quantities of options for controlling their appearance and behaviour to enable control of their appearance across platforms, yet Java virtual machines (JVMs) are typically optimised for them in their default state, which typically causes them to be rendered in the most OS-native fashion. By removing most of the code specifying

[1] http://www.eclipse.org
[2] http://www.sourceforge.net

appearance options, it was possible to enable Swing's native platform "Look and Feel" GUI rendering where supported, giving the program the appearance of a native program, while defaulting to the Java Metal appearance where this is not supported in the JVM. This also had the effect of increasing the speed of display of the program as more GUI elements can fall back on native OS display routines. The screenshots of PIPE throughout this report are taken from many different operating systems.

3.3.2. GUI actions

A custom abstract subclass of Swing's AbstractAction class was defined which provides constructors which set the required bound properties within the object to specify an icon, caption, description and keyboard shortcut to actions which can then be inserted into menus and toolbars with this information used to display the item. This GuiAction class was then subclassed for the various types of action in order to provide implementation of actions' invocation. This provided a simple programming interface for creating a consistent and highly functional interface, replacing many virtually identical yet unrelated classes with a class hierarchy.

3.3.3. Toggleable toolbar buttons

The ability to have toggleable toolbar buttons, indicating particular application states, was implemented by including an optional additional state Boolean object in the GuiAction class's bound variables, with appropriate accessor functions. The existence of this boolean was then used to determine if a button was toggleable; if so, it was added to the toolbar as a JToggleButton descendant. This descendant also implements the PropertyChangeListener interface, which allows it to trigger an update of the button's state when the corresponding GuiAction's state changes, by registering itself as a listener to the action.

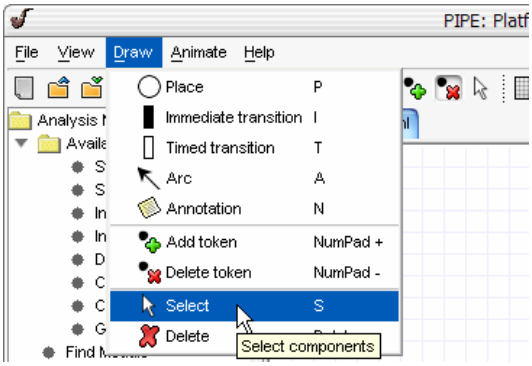


Figure 2 – Listeners allow menu selection to update the toolbar buttons' states

3.3.4. Other GUI features

Many other GUI element were tweaked or modified, including:

- The status bar was relocated from the toolbar to the more usual location at the bottom of the window. This required supplying a parent panel for it to render correctly.
- All status bar text is now stored as constants in the StatusBar class source file to increase consistency and reduce coupling.
- Borders and insets are applied with closer attention to the rendering of subcomponents and with testing across platforms. For example, Windows XP displayed JScrollPane components with a large inset which is now explicitly removed. Some components were displayed with both the component and container borders. This resulted in an unprofessional appearance and was relatively simple to remedy.
- The GuiView object, displaying the net and allowing it to be edited, is now displayed inside a JScrollPane object. With the addition of code to calculate the bounds of the contained net objects, enables the creation of nets of any size and display of large nets on small screens.

3.4. Module GUI

It was determined that the modules' GUI code was duplicated within each module class, with many differences between them in terms of how results were displayed and how options were presented. It was decided to reimplement the common functionality within the GUI source tree, enabling a consistent interface across modules. The interface features chosen for these "widgets" were:

- An encapsulated file browser with a simplified code interface and consolidated common behaviour. This FileBrowser object subclasses the Swing JFileChooser object and provides simplified creation and accessor functions.

Figure 3 – the FileBrowser class. Note that Java provides automatic localisation of this dialog.

- The net chooser/browser panel (PetriNetChooserPanel), allowing the user to specify whether analysis is to be performed on the current net or any net loaded from disk. This was built using Swing components and the FileBrowser object and gives a single function requirement for the code to obtain the required net.

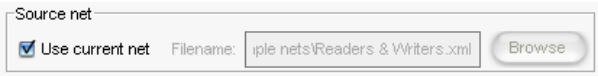


Figure 4 – the PetriNetChooserPanel class

- A panel for displaying arbitrary buttons to perform functions within the module. This ButtonBar object was implemented as a JPanel descendant with a constructor allowing the button captions and corresponding ActionListener objects to be specified, and automatically creates JButton objects and places them within itself.

Figure 5 – the ButtonBar providing navigation buttons for the documentation browser

- The results display panel. This was implemented in such a way that the module would pass an HTML formatted string to the panel, which would then display it in a scrollable pane, giving relevant options. ResultsHTMLPane was implemented as a subclass of Swing's JPanel containing a JScrollPane and a JEditorPane set for display of HTML, with an appropriate hyperlink listener method, with a method for setting the displayed HTML. It also includes a ButtonBar giving buttons for the user to copy the displayed text to the system clipboard or save it to disk, using a FileBrowser. It additionally provides functionality to parse an array of any data type into an HTML formatted table structure, simplifying display of tabulated data.

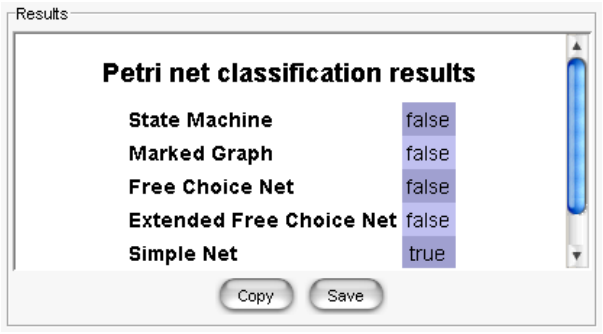


Figure 6 – a ResultsHTMLPane displaying module results including a generated table

- A graph display panel for plotting suitable data. The GraphPanel object extends JPanel with a constructor allowing data to be passed as ArrayList objects and methods to automatically draw a scaled graph on the screen. This can be contained within a GraphPanelPane which provides a ButtonBar and FileBrowser to allow the user to save the graph to a PNG file.

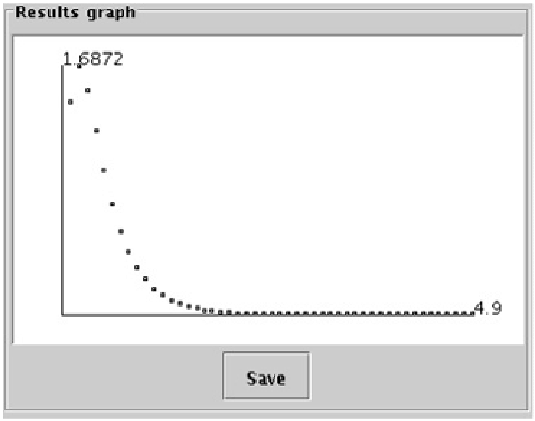


Figure 7 – the GraphPanelPane class

For all other module-specific GUI components, standard Swing components were used in keeping with the intended interface style. Module windows are now modal sub-windows of the editor rather than independent top-level windows, and are designed not to take up more space than is necessary.

Figure 8 – a comparison of the old and new interfaces

3.5. Documentation

The JavaDoc standard is widespread and well supported. It allows the creation of HTML based documentation of the code in a standard format, using a utility included in Sun's Java SDK. It is also supported by Eclipse, which gives assistance in completing the JavaDoc tags correctly.

It was also decided to provide user documentation, giving instructions in how to use the program. This task was chosen due to its suitability for those group members less able to produce code, since due to its nature, this project has a greater emphasis on coding and less on design and analysis. This is also to be produced in HTML format, because it is highly platform independent and also possible to display inside the program due to Java's native HTML renderer.

In order to display the documentation, another subclass of GuiAction was created which, when performed, creates and displays (or activates, if already created) a new window containing a scrolling pane displaying HTML formatted help files. These files make use of an index and many linked pages to display documentation, so the window provides buttons to enable the user to navigate through the history of visited pages and to easily return to the index.

3.6. Enhancements to saving and loading

Several additional data properties and graphical elements have been introduced in the process of augmenting PIPE's capabilities. One of the requirements for these to be useful was that the properties could be preserved and retrieved in the PNML files produced by PIPE.

The extra data-capture facilities introduced were as follows:

1. Petri Net properties
    - Identification of timed and immediate transitions
    - Transition firing rate/weight
2. Graphical display properties
    - Transition orientation
    - Annotations
    - Arc path details (corners, curves)

Introducing these properties into PIPE-produced files required work in the following areas:

- Review of existing code.
- Research into XML and XSL operations and syntax.
- Review of PNML standards to ensure modifications would not conflict with these.
- Co-ordination with graphical elements of the project to ensure the correct data was captured.
- Modification of existing PIPE XML transformation code.
- Modification and cleanup of existing XSL transformation sheets.
- Testing of implementation to ensure all data could be saved and restored correctly.

Design decisions taken in the course of implementing these changes were as follows:

- Store annotations in PNML format using the element tag <labels> at the same level as places, transitions and arc, in accordance with PNML standards. Implement annotations as new a class in PIPE, stored in an ArrayList within the main DataLayer class.
- Store transition orientation, timing and firing rate/weight in PNML as child elements of the transition element.
- Store ArcPath points in PNML as tool-specific child elements of each Arc element.

3.7. Addition of Generalised Stochastic Petri Net analysis

Generalised stochastic Petri nets (GSPNs) are an extension of the basic place-transition Petri nets represented in the original PIPE application that allow modelling of the effects of time on the behaviour of a system by introducing an additional type of transition. The introduction of time as a factor allows a system's performance to be analysed as well as its qualitative properties. Formally, a GSPN is defined as follows:

A GSPN is a 4-tuple (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, $T_1 \neq \varnothing$
- $T_2 \subset$ T denotes the set of immediate transitions, $T_1 \cap T_2 = \varnothing$, $T = T_1 \cup T_2$
- $W = (w_{1, \ldots, w_{|T|}})$ is an array whose entry $w_i \in \mathbb{R}^+$
  - is a (possibly marking dependent) rate of a negative exponential distribution specifying the firing delay, when transition t_i is a timed transition.
  - is a (possibly marking dependent) firing weight, when transition t_i is an immediate transition.

Performance measures that can be produced from GSPNs include mean cycle times for tangible states, sojourn times in tangible states and transition throughput. The structure of GSPNs is such that they can be represented and analysed as a semi-Markov process. In order for quantitative results to be derived, it is necessary to impose some qualitative restrictions on the net, as in some circumstances it is not possible to determine whether the necessary preconditions for employing semi-Markov process analysis techniques have been met.

3.7.1. Preparation

The following areas had to be considered when planning how to implement GSPN support in PIPE:

1. Modifications to saving and loading of PNML files
2. Graphical display of timed net elements
3. The details of these two points are discussed elsewhere in this report.
4. Graphical display of analytical results

   The most logical way of enabling the user to access the results of GSPN analysis was through the PIPE module interface designed by the original PIPE group. Part of the PIPE GUI enhancement work undertaken by the current project team involved creating a set of widget classes to standardise the appearance and simplify the implementation of existing analysis modules. Using these classes made displaying the results of GSPN analysis a straightforward exercise. HTML anchors were included in the text display to allow navigation between tables of results and details of the states those results applied to.
5. Generation of analytical results

   This point took up the majority of the time required to implement GSPN analysis in PIPE. The effort was divided fairly evenly between background research and implementation. Research areas included general Petri net theory, matrices and linear algebra.

3.7.2. Implementation

Following research, the implementation was constructed as follows:

1. An assessment of whether the necessary preconditions for GSPN had been met was written (i.e. a qualitative analysis). Firstly, a test as to whether both timed and immediate transitions are present is performed. This is a straightforward query of the array of transitions in the DataLayer of the net to be tested. Next, a test as to whether 'Condition Equal Conflict' is satisfied is performed. The algorithm checking this condition is contained in `pipe.modules.GSPN.java`, in method `testEqualConflict (DataLayer data)`.

   A GSPN (PN, T_1, T_2, W) satisfies condition EQUAL-Conflict iff

   $$\forall t,t' \in T : \bullet t \cap \bullet t' \neq \varnothing \Rightarrow \{t,t'\} \subseteq T_1 or \{t,t'\} \subseteq T_2$$

2. If the conditions of the qualitative analysis are not satisfied, the analysis terminates. Otherwise a quantitative analysis begins. There are several parts to this.

   2.1. Quantitative analysis of GSPNs entails making a distinction between states that are tangible (i.e. those states where the only enabled transitions are timed transitions) and vanishing (i.e. those states where an immediate transition is enabled). In order to do this it was necessary to determine the state space of the net, i.e. the set of all possible markings of the net. The original PIPE implementation contained a method for traversing the state space of a place-transition net using a tree structure in order to assess certain qualities of the net (`myNode.RecursiveExpansion()`). This method did not retain the net states anywhere, nor did it have any support for timed transitions. It was therefore decided to introduce a new container class to hold state spaces of the net (class StateList – an Arraylist of int[]

marking and arbitrary marking name), and to overload the existing RecursiveExpansion method to add each unique new space to a StateList container and take account of whether timed transitions were enabled when deciding whether a marking should be judged as reachable or not. The reachability set of the net is then separated into tangible and vanishing states.

2.2. Using the lists of tangible and vanishing states, the probability of transition between two specific states is calculated, using the following definition:

$$P[M_k \rightarrow M_r] = \frac{\sum_{i:t_i \in \{t \in T | M_k[t > M_r\} \cap EN_T(M_k)} w_i}{\sum_{j:t_j \in EN_T(M_k)} w_j}$$

where M_k, M_r are two states of the net.

This calculation is implemented in the GSPN module in method `probMarkingAToMarkingB(DataLayer pnmlData, int[] marking1, int[] marking2)`.

2.3. Applying this probability calculation to the entire state space of the net generates the matrix P where:

$$P = \begin{pmatrix} C & D \\ E & F \end{pmatrix}$$

C is the matrix of transition probabilities from vanishing states to vanishing states,

D is the matrix of transition probabilities from vanishing states to tangible states,

E is the matrix of transition probabilities from tangible states to vanishing states,

F is the matrix of transition probabilities from tangible states to tangible states.

2.4. The qualitative preconditions referred to above ensure the existence of a steady state distribution of the embedded Markov chain of the net. This distribution is derived from the global balance equations

$$\tilde{\pi} P = \tilde{\pi} \text{ and } \sum_{M_i \in \hat{T} \cup \hat{V}} \tilde{\pi}_i = 1$$

where $\tilde{\pi}$ is the steady state distribution of the embedded Markov chain

$\hat{T}$ is the set of tangible states

$\hat{V}$ is the set of vanishing states

M_i is a marking

2.5. Research was required to determine how to calculate a solution of the global balance equations. The findings of this research were that linear programming methods would need to be employed as the problem essentially involved solving *n* linear equations with *n* unknowns. The technique for tackling this type of problem is known as Gaussian elimination and involves a process of multiplying matrix rows by a factor then subtracting them to eliminate terms, rearranging rows to ensure a triangular placement of coefficients, and substituting backwards to obtain values for each term. The possibility was considered that an open source implementation of the necessary algorithm might exist, and that this might be used in PIPE to reduce development time. Further investigation did not uncover any suitable code, so a Java implementation of Gaussian reduction was written based on textual descriptions of the algorithm. However, the research was fruitful in assisting with the production of some required intermediate calculations. In particular, it was found that the Jama matrix library contained a method for calculating a matrix inverse. It was therefore decided to include the Jama library within PIPE, as re-implementing this particular calculation would have been an unnecessary distraction from the core development effort.

2.6. The complexity of producing a solution of the global balance equations increases with the number of unknown terms. This number of unknown terms can be reduced, thereby increasing the efficiency of the computation, provided no timeless traps exist (i.e. it is not possible for the net to get to a position where the only reachable states are vanishing states). Petri net theory has shown that satisfaction of Condition Equal-Conflict is sufficient to prove absence of timeless traps; hence our initial qualitative

analysis proves that a reduction of states to be considered can be performed. This reduction takes the following form:

$$\tilde{\pi}P' = \tilde{\pi} \text{ and } \sum_i \tilde{\pi}_i = 1$$

where $P' = F + E*(I-C)^{-1}*D$

C, D, E, F have the meanings described above

I is the identity matrix

The principle behind this reduction is that we know the steady state distribution of being in a vanishing state for the embedded Markov chain of the net will be zero, so the only states that need to be considered are tangible states.

2.7. Once the steady state distribution of the embedded Markov chain of the net has been calculated, it is relatively straightforward to determine further characteristics of the net. In particular:

The mean number of visits to marking M_i between two consecutive visits of marking M_j is

$$v_{ij} = \frac{\tilde{\pi}_i}{\tilde{\pi} j}$$

The sojourn time $t_s(M_i)$ for tangible marking M_i is given by

$$\sum_{j:t_j \in EN_T(M_i)} w_j$$

The mean cycle times $t_c(M_i)$ for tangible marking M_i is given by

$$t_c(M_i) = \frac{\hat{X}}{\tilde{\pi}_i} \text{ where } \hat{X} = \sum_i \tilde{\pi}_i * t_s(M_i)$$

The steady state distribution for tangible states π_i is given by

$$\pi_i = \frac{t_s(M_i)}{t_c(M_i)}$$

2.8. Finally, the throughput (i.e. the mean number of firings at a steady state) of each transition can be calculated. A different approach needs to be taken depending on whether the transition is timed or immediate.

$$\text{Throughput } \dddot{d}_j \text{ of timed transition } t_j = \sum_{M_i \in EN_j} \pi_i w_j$$

$$\text{Throughput } \dddot{d}_j \text{ of immediate transition } t_j = \sum_{M_i \in EN_j \cap \hat{V}} r_i \frac{w_j}{\sum_{k:t_k \in EN_T(M_i)} w_k}$$

where $r = \pi * \tilde{E} * (I-C)^{-1}$,

$$\tilde{E} = \sum_{k:t_k \in \{t \in T | M_i[t>M_j\} \cap EN_T(M_i)} w_k; M_i \in \hat{T}, M_j \in \hat{V}$$

A variety of matrix operations were used to generate these results.

The computational complexity of producing these results is a function of the size of the state space of the net. It was decided to impose a limit of 10,000 nodes on the size of the state space to be analysed, as PIPE is primarily intended a graphical editing tool rather than a high-performance analysis tool. If a user attempts to perform GSPN analysis on a net with more than 10,000 nodes an exception is raised suggesting they use a more appropriate tool such as DNAmaca.

3.7.3. Problems encountered

The extensive nature of changes to PIPE components caused instability in the behaviour of various graphical elements. In particular, for a large part of the project lifecycle it was not possible to save or load nets. This caused difficulty in testing the validity of the results produced by the GSPN analysis module, as it was often not possible to create or access example nets in order to view their output.

3.8. DNAmaca interfacing

The standard DNAmaca interface is via the command line. The following procedure is required:

1. Create Petri net description file in DNAmaca format (.mod file). A full description of this file format can be found in Knottenbelt (1996).
2. Pass the mod file as an argument to a wrapper program 'urta' for the Markovian analyser. This compiles the .mod file into an executable named 'uniform' - uniformisation based passage-time analysis for Markov chains.
3. Run the 'uniform' executable, capturing its standard output. The passage time results are marked with a 'DATA0' tag.

Required features of a module designed to interface with this tool are:

1. Create .mod file from net currently being editing in PIPE.
2. Provide GUI to specify run time parameters to DNAmaca
3. Wrap 'urta' & 'uniform' external commands and their progress in as transparent a means as possible, such that the user is not made aware of their existence
4. Accept and parse results from 'uniform', display in tabular and graphical form within PIPE GUI

An alternative means of accessing DNAmaca is via a client/server interface. However, after investigation of tools provided to us, it was concluded this would be unfeasible.

The principle challenge for this module would be handling the inter-process communication - wrapping the above procedure in such a way that an external command line tool would appear to be an integral part of PIPE.

Another issue was the length of time taken for analysis, and how this would affect the responsiveness of the GUI. For this reason, the decision was taken to run the analysis procedure in its own thread.

The initial dialog presented to the user follows the PIPE standard of a single html pane to display results and user feedback. A separate section allows the user to specify options, which will be included in the generated .mod file or passed as command line parameters.

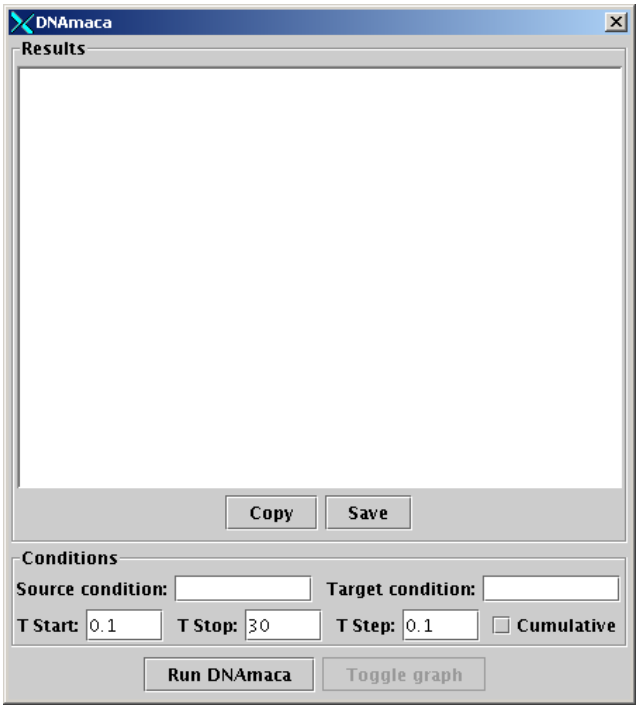


Figure 9 – the initial DNAmaca module interface

Once the user has entered the relevant options, the 'Run DNAmaca' button would be pressed. After some initial checks to ensure the 'urta' script is available on the system path and the net is a valid GSPN, the `generateMod()` method is called. This creates a string containing the .mod file that will be passed to DNAmaca for analysis, extracting information from the DataLayer. An example output is shown below.

```
\model{
	\statevector{
		\type{short}{P0, P1, P2, P3, P4}
	}

	\initial{
		P0 = 1; P1 = 1; P2 = 1; P3 = 0; P4 = 1;
	}
	\transition{T0}{
		\condition{P0 > 0 && P1 > 0}
		\action{
			next->P0 = P0 - 1;
			next->P1 = P1 - 1;
			next->P2 = P2 + 1;
		}
		\weight{1.0}
	}
	\transition{T1}{
		\condition{P1 > 0}
		\action{
			next->P1 = P1 - 1;
			next->P3 = P3 + 1;
		}
		\weight{2.0}
	}
	\transition{T2}{
		\condition{P3 > 0}
		\action{
			next->P3 = P3 - 1;
			next->P4 = P4 + 1;
		}
		\weight{3.0}
	}
	\transition{T3}{
		\condition{P2 > 0}
		\action{
			next->P2 = P2 - 1;
			next->P0 = P0 + 1;
			next->P4 = P4 + 1;
		}
		\rate{5.0}
	}
	\transition{T4}{
		\condition{P4 > 0}
		\action{
			next->P4 = P4 - 1;
			next->P1 = P1 + 1;
		}
		\rate{4.0}
	}
}

\passage{
	\targetcondition{P4 == 3}
	\sourcecondition{P4 == 1}
	\t_start{0.01}
	\t_stop{3}
	\t_step{0.01}
}
```

A thread is then spawned to handle the DNAmaca external process. This thread extends the utility class SwingWorker, which is provided by Sun as a recommended way for implementing lengthy operations within a Swing application.

The SwingWorker thread allows the PIPE GUI to remain responsive, but it still leaves the problem of providing the user with feedback on a possibly lengthy operation. For this, the Swing JProgressMonitor was component utilised. This is shown below.

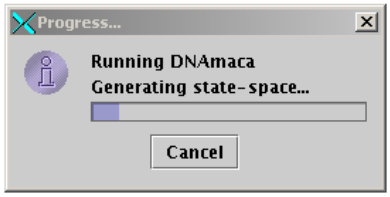


Figure 10 – the DNAmaca progress dialog

Both the status text, indicating to the user the current active task, and the progress bar itself, are updated by parsing the output of the 'urta' & 'uniform' processes, scanning for a list of control strings. As each control string is detected, the relevant properties of the SwingWorker thread are updated.

A separate Timer object handles updating the progress monitor. This object also closes the progress monitor and enables the 'Toggle graph' button when DNAmaca analysis has been completed.

Once the analysis is complete, a table of results is inserted into the html result pane as *x*, *y* values. These results can then be saved, and opened directly within Excel for graphing or analysis. However, a simple graphing feature was added to facilitate visualisation of data. An image showing this is below.

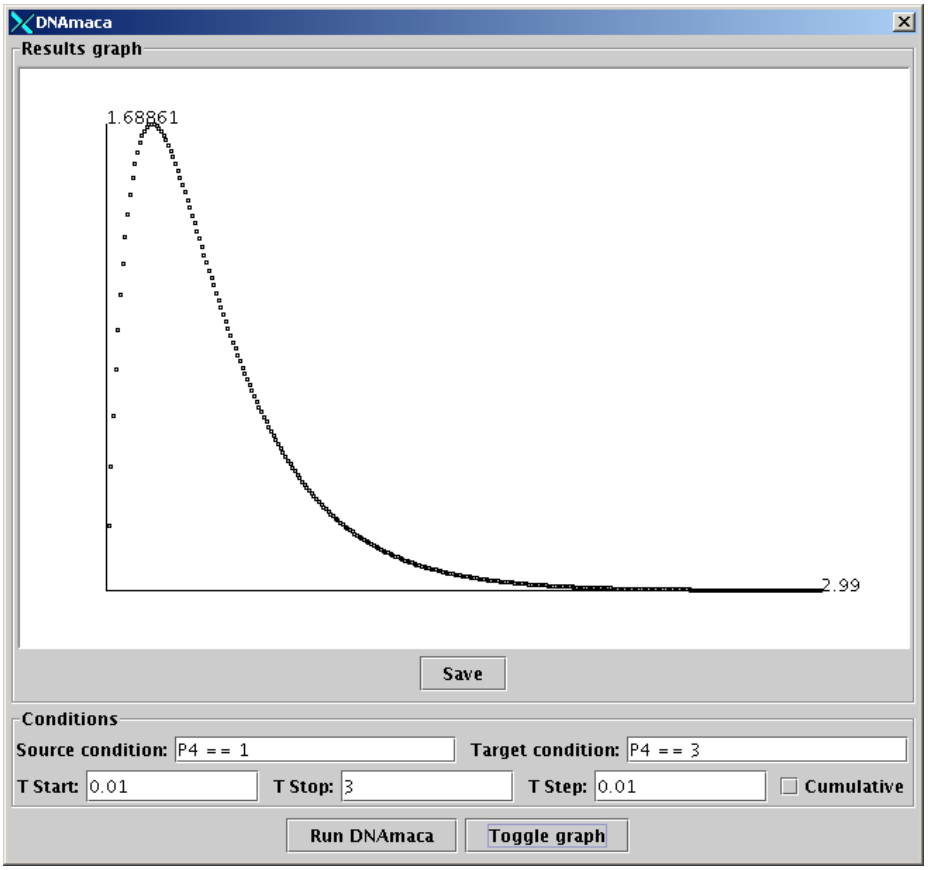


Figure 11 – example passage time analysis graph

The graph can also be saved as a PNG file if desired.

Once the user has finished working with an initial set of results, it is possible to change the parameters and run DNAmaca again (for example, to fine-tune the time range).

3.9. Object selection

Among the most useful new features in PIPE this year is the ability to select objects which have been placed on the net – either individually or in groups – and move them around the screen or delete them. In the original PIPE, this was possible to a certain degree via the 'move' mode, which permitted the user to drag places and transitions around the screen – provided that they were unattached to arcs – and the 'delete' mode which could delete individual objects.

3.9.1. Selection mode

The ‘move’ mode is gone, replaced by a more general ‘select’ mode, which the user can get to by pressing the selection button in the toolbar, using the menu or simply by pressing the S key at any time. In this mode, the user may do a number of selection-related operations, as described below.

The fact that objects are selectable at all is due to a number of modifications in the PetriNetObject class, from which all the net-editable objects inherit. All PetriNetObjects now have `select()` and `deselect()` methods; methods for ascertaining whether the object is currently selected; and a switch to determine whether or not it *can* be selected. Naturally, the user would like some kind of visual feedback as to whether or not an object is selected, and this is achieved by having each object draw itself slightly differently depending on its selection status.

3.9.2. Dragging out a selection: the selection rectangle

The selection rectangle is a feature common to most modern drawing applications, but one which was previously absent from PIPE. It allows the user to click the mouse at a point on the screen and to then *drag out* a rectangular shape to enclose some portion of the editing window. If any portion of a selectable object intersects with the rectangle, then it is itself selected. The selection is updated in real-time by refreshing it every time the user drags the mouse.

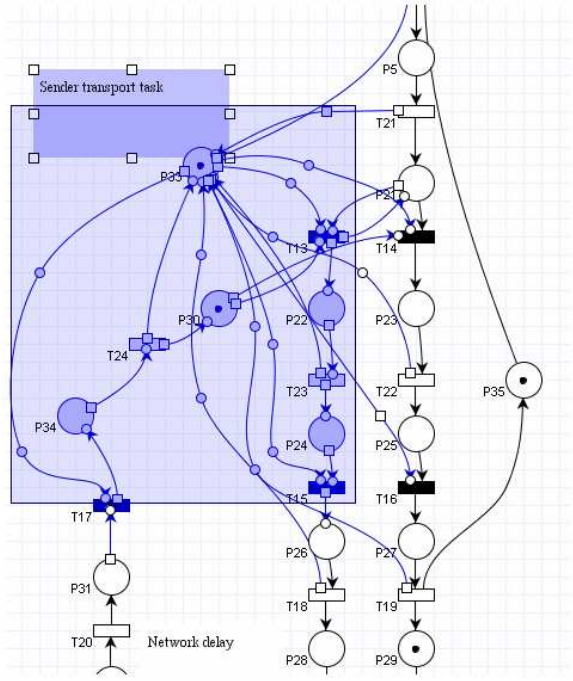


Figure 12 – example of selection rectangle usage

The creating of the selection rectangle (as well as functions involving the current selection) is handled by a new class – `pipe.gui.SelectionObject`. One instance of SelectionObject is added to every GuiView pane object created (i.e. the editing window), and its dimensions are such that it occupies the entire editable area. The selection rectangle (the large semi-transparent rectangle shown in the above screenshot) is thus drawn on the SelectionObject itself, as opposed to in the editing pane, but this is mainly a matter of convenience. Note also that since each GuiView instance has its own SelectionObject, calling the SelectionObject methods will not affect the selections present for other open nets in any way.

When processing the selection (by calling SelectionObject’s `processSelection()` method), those objects contained within the rectangle are selected after first deselecting everything else by calling `clearSelection()` (unless the Shift key is being held down, in which case no deselection occurs). `processSelection()` does this by getting a list of all the components in the GuiView editing pane and then iterating through only the PetriNetObjects (the selectable objects), checking whether the rectangle intersects the bounds of each object, except in the case of arcs where we check for intersection with the arc’s associated ArcPath object.

3.9.3. Clicking on objects and using the Shift key

Whilst dragging out selection rectangles is very useful, often a user will want to select or deselect individual objects, and this can be achieved simply by clicking on them. All subclasses of PetriNetObject implement mouse even listeners, themselves all subclassed from PetriNetObjectHandler, a generic handler which provides all the mouse-related functionality for any editable objects the user will place in the window.

This basic handler has now been expanded to include support for selection of objects upon the user clicking on them whilst in selection mode. Objects may be individually added or removed from the current selection by clicking on them while holding the Shift key. This is adhering to the accepted interface practice of most modern programs.

3.9.4. Movement and deletion of selections

Whereas in the original PIPE the user had to enter either 'move' or 'delete' modes in order to do anything with objects already created, all movement and deletion of existing objects is now accomplished by first selecting an object and then performing the required action. Movement can be achieved by dragging the selection (by clicking on an already selected object and dragging it before releasing the mouse button) and deletion by choosing 'Delete' from the drop-down menu or simply pressing the Delete key.

As mentioned before, the SelectionObject class takes care of most operations involving selections, regardless of whether only one object is selected, or many. In the case of moving selections of objects - by dragging one member of the selection – the member object in question calls SelectionObject's `translateSelection()` method, passing to it as parameters the x and y distances by which it should be moved. As such, individual objects don't have to concern themselves with other selected objects directly. `translateSelection()` operates similarly to `processSelection()` in that it gets a list of the objects contained by the GuiView object, then iterates through them all and – if they are both subclasses of PetriNetObject and are themselves selected – it will translate their positions by the requisite amount.

The deletion of objects upon pressing Delete or choosing it from the menu is similar, in that the SelectionObject's `deleteSelection()` method is invoked, and much like the other selection-wide methods it iterates through all the potentially deletable objects, calling the delete() methods for those which it detects as being selected.

Upon deletion, all PetriNetObjects remove themselves from their parent GuiView container, and also call the relevant DataLayer object's `removePetriNetObject()` method, passing a reference to themselves as a parameter (apart from ArcPathPoint objects, which the DataLayer is oblivious to, and they to it). In this way the internal representation of the Petri net reflects the graphical representation the user sees. It should be noted that the DataLayer, when removing a Place or Transition, will additionally iterate through all the attached Arc objects and remove those too, calling their `delete()` methods as it does so.

3.10. Curved arcs

The specification was to allow drawing of multi-segment arcs containing both curved and straight segments. Any implementation must be both intuitive to use and produce an elegant result, given that the nets can be exported and printed for use in lecture slides as well as being analysed and animated within PIPE. To this end:

1. A line being drawn should render exactly as it will appear once it is drawn, giving live feedback on the result
2. Curved sections of arcs should intersect with places tangentially, and should intersect with transitions at right angles
3. Curved sections of arcs should be collinear with straight sections at the point of intersection
4. In addition to the above two requirements, the whole length of a multi-segment arc should appear smooth at straight-curve intersections and end points
5. The only user input should be the position and type of each intersection point on the line; curves satisfying the above constraints should then draw automatically
6. Points can be removed from multi-segment lines and the type of each point can be altered once the curve is drawn

A variety of algorithms for interpolating curves through a sequence of points were considered and evaluated both against the above criteria and their relative ease of implementation. In particular, to satisfy the final condition above any algorithm had to generate its own control points for curve drawing without input from the user. The decision was taken to use an algorithm for generating natural cubic curves, interpolated across any an

arbitrary number of curved points and returning a set of Bézier control points. The curves can then easily be rendered using built-in Swing and AWT functions. The whole arc can then be drawn using an AWT GeneralPath object using only its `lineTo()` and `curveTo()` functions and the control points be directly overridden to satisfy the aesthetic considerations outlined above. The ArcPath class encapsulates the GeneralPath and provides the custom functionality described here.

The control points are overridden directly rather than simply clamping the gradient at the ends of the cubic interpolation to ensure there are no sharp bends in the curve at any intersections. This also reduces the need for calculating angles as in most cases the control points are calculated from the differences between points rather than the angles. The overall algorithm for drawing a curve is:

```
for (each point)
  if (straight point) // point being drawn TO determines type
    set control points collinear with point & last point
  else
    while (curved point)
      calculate natural cubic interpolating the points
      set appropriate control points for each curved point
for(each point)
  if (point is curved & previous point is straight)
    override first so that it is collinear with previous straight line
  if (point is curved & previous point is straight)
    override second so that it is collinear with previous straight line
for (end points)
  set control point so that c, point, centre of place collinear
  set control point so that arc meets edge of transition at 90°
```

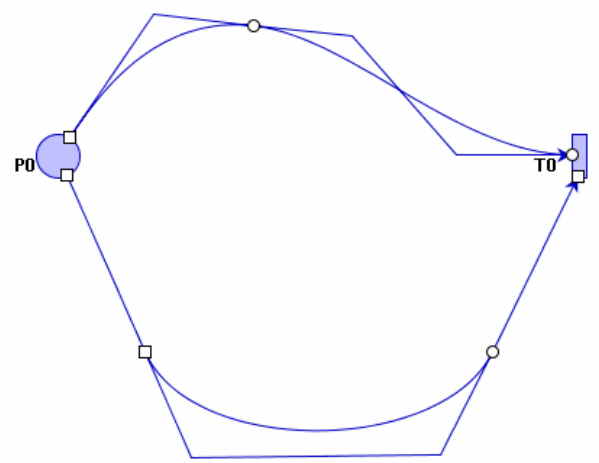


Figure 13 – Diagram illustrating locations of control points. Note that where a curve intersects a place, the centre of the place, the end point of the arc and both control points are collinear to ensure a smooth transition.

An algorithm allowing local control of points was abandoned. This decision was taken because integration with PIPE would add unnecessary extra complexity without adding significant end-user benefits; specifically, appearance of the printed/screen rendered diagrams is not significantly improved.

3.11. Arc-transition entry points

The issue of arc-transition entry points was not mentioned in the original specification, but became clear as the project advanced. It is therefore necessary to specify the problem before describing the methodology used to solve the problem.

3.11.1. Specification

In more complex nets, it is possible to have several arcs entering or exiting one transition. This presents the problem of how to arrange the incoming arcs to ensure the resulting diagram retains clarity, while looking professional (one of the goals of the project). The original version of PIPE employed a simple solution,

positioning the arrows depicting the direction at the centre of the arc. All arcs connecting with a transition would then be drawn to the centre of the transition. This is illustrated below.

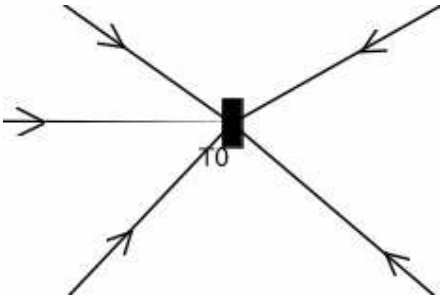


Figure 14 – Original PIPE implementation of arc-transition entry

This solution is now inappropriate for several reasons. Firstly, with multi-segment arcs it no longer makes sense to have the direction indicator at the centre of an arc, since the arc centre is no longer well defined. The solution is to have the arrowhead at the terminating end of an arc. This then brings the problem of several arrowheads converging at a single point (if the arc angle of entry is similar) making the resulting diagram difficult to read.

The second reason is related to the introduction of curved arcs. With curved arcs, it becomes possible to restrict the angle of entry for arcs connecting to a transition to being perpendicular to the transition edge it interfaces with. The intention here is again to increase clarity in the resulting diagram.

For these reasons a new algorithm to calculate the interface points of arcs with transitions was required.

3.11.2. Methodology

The design for the new algorithm made the following requirements.

- Arcs connecting with a transition should be divided into 4 quadrants based on their angle of entry into the transition. The quadrants corresponding to the smaller edges of the transition should have smaller ranges:

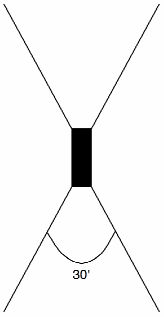


Figure 15 – arc entry quadrant division

- Arcs in the smaller quadrants should be connected at the centre of the relevant transition edge.
- Arcs in the larger quadrants should be spread at equidistant intervals along the transition edge.
- Arcs in the larger quadrants should be ordered by angle, such that arcs do not cross each other where possible.
- The algorithm should deal appropriately with rotated transitions of arbitrary angle.

To illustrate these requirements, a contrived example is shown below using the resulting algorithm.

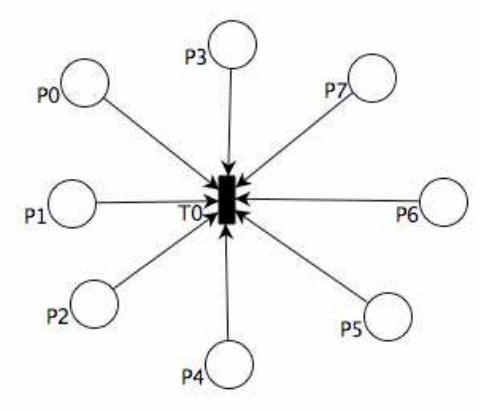


Figure 16 – an example of the desired (and obtained) output

The final algorithm is described below using pseudocode:

```
ArcArray connectedArcs = all connected arcs to thisTransition;
sort connectedArcs by entry angle;

ArcArray top, bottom, left, right;
foreach arc in connectedArcs {
  currentAngle = currentArc.angle – thisTransition.angle;
  if (currentAngle in top quadrant range) add currentArc to top;
  if (currentAngle in bottom quadrant range) add currentArc to bottom;
  if (currentAngle in left quadrant range) add currentArc to left;
  if (currentAngle in right quadrant range) add currentArc to right;
}

transform = rotation transform of thisTransition.angle;

foreach arc in top
  set connecting point of arc to transformed centre of top transition edge;

foreach arc in bottom
  set connecting point of arc to transformed centre of bottom transition edge;

offsetIncrement = transitionHeight / (left.count + 1);
centreOffset = transitionHeight – offsetIncrement;
foreach arc in left {
  set connecting point of arc
      to transformed (centre of left transition edge + centreOffset)
  centreOffset -= offsetIncrement;
}

offsetIncrement = transitionHeight / (right.count + 1);
centreOffset = transitionHeight + offsetIncrement;
foreach arc in right {
  set connecting point of arc
      to transformed (centre of right transition edge + centreOffset)
  centreOffset += offsetIncrement;
}
```

A problem with the above algorithm occurred where an arc angle became greater than 2π. The routine which calculated the arc angle would then wrap the angle returned to zero, which confuses the sorting of angles. The solution to this problem was to ensure that the angle wrapping point always coincided with the edge of one of the quadrants.

The positioning routine was later extended to constrain the entry angle of curved arcs, such that they would be perpendicular to the transition edge to which they connected. An example of this is shown below.

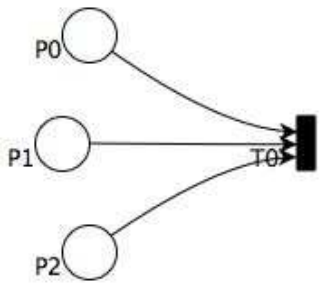
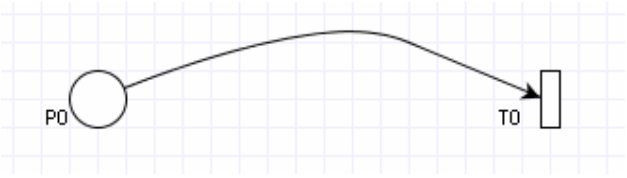


Figure 17 – constrained curved entry to transition

Since the algorithm is capable of recalculating all the connecting positions of arcs for a transition at once, the user is able to drag a place around a transition and see the changes updating dynamically.

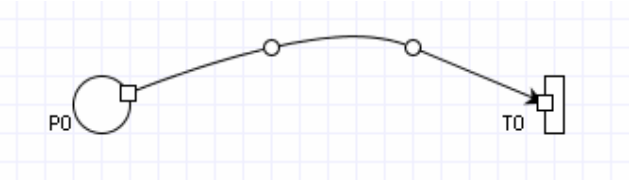
3.12. Arc path manipulation points

Since multi-segmented curved and straight-line arcs are a new feature in this incarnation of PIPE, the arc points (`pipe.datalayer.ArcPathPoint` – subclassed from PetriNetObject) are new. Unlike Places, Transitions, Arcs and Annotations, however, Arc points are not bound to the underlying DataLayer instance representing each net, but rather to the ArcPath object of the associated arc. They provide a useful visual way to represent the control points that determine the shape of the path, and as such it is useful for the user to be able to perform operations on them to modify the shape of the arc after it has been created, rather than having to recreate the arc from scratch.



The image to the right shows a simple 3-segment curved arc with four path points – two at the start and end respectively, and two in the middle.

It was decided that in general, the user wouldn't want arc points displayed constantly, since they would clutter the net view and they are only an aid for shaping the arcs, not really a part of the net itself. However, it would be nice if the points for a specific arc became visible once the mouse moved within a certain distance of it, or when the arc was selected. The proximity detection required for the former case is achieved by using Java Strokes (`java.awt.Stroke`),. Strokes can be used to create a shape that reflects the outline of an existing shape (in this case the arc path), specifying a width for the stroke that represents how thick the outline is. The Arc object's `contains()` method therefore – in addition to its standard containment tests – checks to see whether or not the specified point is within the path outline, and tells the path points to show or hide themselves appropriately.

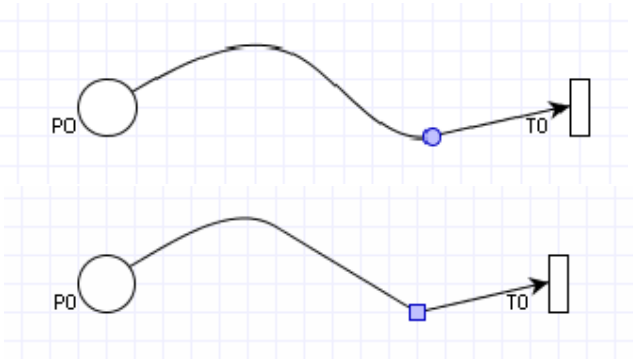


The image to the right shows how the appearance of the arc changes once the mouse pointer is within range. The circles represent the end of curved arc segments, and the squares the end of straight segments (the first point always appears square). When the mouse is moved away, the points hide themselves once more, unless either the arc or the points themselves have been selected, in which case they will remain visible until deselected.

Since they inherit from PetriNetObject and have mouse handlers which subclass PetriNetObjectHandler, ArcPathPoint objects are selectable, draggable and deletable. Additionally, since they use the local GuiView instance as a container, as opposed to their parent Arc object, SelectionObject operations are applicable to them, meaning that they can also be selected with the selection rectangle, and may be dragged and deleted *en masse* at the same time as other objects.

Most of the time, of course, a user will probably just want to perform operations on a single point, dragging/deleting it or maybe even toggling its type between straight and curved. Whenever the user does anything to an ArcPathPoint object, it will signal its parent ArcPath to re-create the path according to the new configuration and repaint it as necessary, ensuring that the appearance of the arc reflects the state/position of the point.

The image to the right shows the same arc again, after the third point has been dragged, reshaping the arc. The point remains selected, thus it is still visible even though the others are not.



The second image shows that the user has used the ArcPathPoint's right-click menu to toggle the point type from curved to square. The new shape of the arc reflects the change.

When an ArcPathPoint object is deleted, it deletes the reference to itself in the parent ArcPath's `pathPoints` list (which is the list that the ArcPath will iterate through as it generates the arc path), thus removing the control point it represents. In general, an ArcPathPoint will refuse to delete itself (via the `delete()` method) if there are only two points left on the arc because they are the required (i.e. the start and end points!). This safety check is only bypassed when the parent Arc object itself is deleted, resulting the in ArcPath object calling the more severe `kill()` method on all of its points as part of its cleanup.

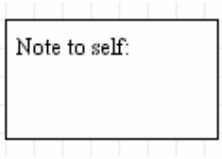
3.13. Arc snapping

It was suggested that creating arcs might be made easier by having the arc-in-progress snap to any Places or Transitions within the vicinity of the mouse pointer – but only to the type to which the arc might actually be attached. This is achieved by modifying the Place and Transition classes to incorporate proximity detection similar to that implemented by the Arc class to determine when to show or hide path points.

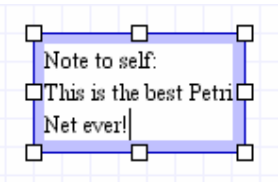
Stroked outlines are created of the Place and Transition object shapes, and checking for intersection with these is added to the standard `contains()` method checking for those objects. A Place/Transition will therefore check to see whether the mouse pointer is contained within the bounds of that outline and additionally whether the local GuiView object is presently creating a new arc. If this is the case, the Place/Transition will acquire a handle to the associated Arc object and instruct it to attach itself (superficially – not at the level of the DataLayer) by setting the arc's target to that Place/Transition, but only if the arc's source is of the opposite type. The arc is then told to update its path, as are any other connected arcs whose position has been affected. When the mouse moves out of range, the arc will cease to be 'attached' to the object and will be free to follow the mouse again.

3.14. Annotations

The `pipe.datalayer.AnnotationNote` class allows the user to add annotations to the net, and as such can be very useful, especially for large, complicated nets. The notes themselves inherit from PetriNetObject, making them automatically selectable and draggable, and the text-editing functionality is provided by a Swing JTextArea, contained within the object. An AWT RectangularShape provides the enclosing rectangle.



The note can be edited by double-clicking it, or via the right-click menu (its AnnotationNoteHandler mouse handler tells it to become editable and focused in both cases). The default `deselect()` method was also overridden so that in addition to deselecting the component it also tells the JTextArea to stop being editable.



In addition to the JTextArea, AnnotationNote also serves as a container for eight resize points (`AnnotationNote.ResizePoint`), implemented as subclasses of JComponent. These are always drawn with their centres at each of the four corners of the rectangular border bounding each AnnotationNote object, and also at the midpoints of the lines of the rectangle. They are only painted if the annotation object is selected, including when it is being edited.

Upon creation, each resize point is given a binary mask which indicates whether it is responsible for the top, bottom, left or right sides of the annotation (or some combination of these). When its mouse handler (`AnnotationNote.ResizePointHandler`) detects a drag, it calls the component's `drag()` method, passing it the Grid-modified mouse coordinates. The `drag()` method then checks the point's movement mask to determine how many of the AnnotationNote's size-adjusting methods to call (e.g. if the point is responsible only for the top of the rectangle, it will call `adjustTop()`, giving it the y-offset to adjust by, etc.).

The points don't bother updating their own positions, as these are updated automatically when the AnnotationNote calls its own `updateBounds()` method. This is important since the AnnotationNote might

refuse to grow any smaller if it means that the text area would become too small to display all the text. Also with regard to this, it was decided that it would be good if the note would resize automatically as the user was typing, to accommodate all the text. Since the note could be made to resize in any direction, it was thought best to simply have the user set any width, and then have the note extend only downwards as required. This also means that the note will automatically grow in height if the user is attempting to decrease its width beyond its ability to contain the text (for all of this, the JTextArea's `getPreferredSize()` method is called, and only the height value of the Dimension object it returns is used). If, however, the user attempts to decrease the height too much, the note will refuse the change since the width doesn't auto-scale. Additionally, there are absolute minimum widths the AnnotationNote will not shrink beyond, even when empty.

The modification of the Annotation's bounds is illustrated in pseudo-code below:

```
updateBounds() {
  newHeight = note.PreferredHeight;

  if ((note.height < newHeight) and (newHeight >= minimumAllowedHeight))
  then note.height = newHeight;

  rectWidth = note.width + RESERVED_BORDER;
  rectheight = note.height + RESERVED_BORDER;

  rectangle.setFrame(RESERVED_BORDER/2, RESERVED_BORDER/2, rectWidth, rectHeight);

  setSize(rectWidth + RESERVED_BORDER,rectHeight + RESERVED_BORDER);

  note.setLocation(MIDDLE_OF_RECTANGLE);

  updatePointLocations(); // update resize point locations.
}
```

Notice that the size of the bounding rectangle (called *rectangle* above) is generated based on the size of the JTextArea component (called *note*). The text area is then positioned within the centre of the rectangle.

The `adjustTop()`, `adjustBottom()` methods called by the resize points check that the height note height will not decrease below its preferred height before changing it, whilst the `adjustLeft()` and `adjustRight()` methods check to make sure that the width doesn't decrease below the minimum allowable width.

3.15. Module manager

One of the much-touted features of the original PIPE was its ability to extend the functionality of the basic program by loading external program modules – this included both the base analysis and classification modules provided with program, as well as any additional custom-made modules written by anyone wishing to add new features to PIPE. The idea was to have a standard Java interface that all modules would have to implement, setting out the basic methods necessary for all modules. PIPE would then use Java's Reflection API to dynamically load the module classes at run-time and call those methods as a way of running the module, having passed it the `pipe.datalayer.DataLayer` object representative of the current Petri net being edited.

The original implementation was, however, fraught with problems, the two main ones being that:

1. The module to load had to be in the JVM's class path, which would usually consist of PIPE's `/bin` directory and the runtime library directory tree
2. PIPE automatically generated the module's package hierarchy based on its directory structure, e.g. if the path to the module's main class (starting from PIPE's `/bin` directory) was `/modules/magic/analysis/ SuperAnalysis.class`, then the full class name generated by PIPE for use with Reflection would be `modules.magic.analysis.SuperAnalysis`. Thus, if a module was not within PIPE's `/bin` directory with a relative folder structure exactly matching its package definition, it would not be possible to access it via Reflection and it would fail to be opened.

The combined effect of the above problems effectively meant that PIPE's ability to load modules after startup was pointless, since the only modules it could possibly load would already have been loaded by the recursive module search that it performs on startup anyway. These two problems have now been addressed, with separate techniques employed in each case:

1. To get around the class path problem, a new class (`pipe.gui.ExtFileManager`) was employed, for loading external classes. It uses a URLClassLoader, which is capable of storing an array of class paths in URL format, and as such a new path can be added as each new module is loaded (provided it actually differs from old paths). How the path is generated is explained below.

2. It is difficult to determine what the top-level package directory of an arbitrary .class file actually is, and without this the class cannot be loaded. Therefore, each module now has a .properties file associated with it (making use of the `java.util.Properties` class), located in the same directory as the directory at the top-level of the module package hierarchy. The properties file contains the fully qualified name of the main class of that module.

The directory containing the properties file is added as a new class path URL, so there is no longer a problem with the JVM being unable to load the module classes. It is also easier to open a module by 'running' its configuration file rather tan having to navigate down to its main class file.

The module manager supports modules with multiple conformant functions contained within them, by using a tree structure to display them. It now automatically expands all of the branches; relabels the default `run()` method with the enclosing module name; and, if a module contains only one analysis function, removes the parent branch so that only the leaf is visible. This greatly enhances the functionality of the module manager by allowing immediate viewing of available analyses, clearer labelling and an uncluttered view.

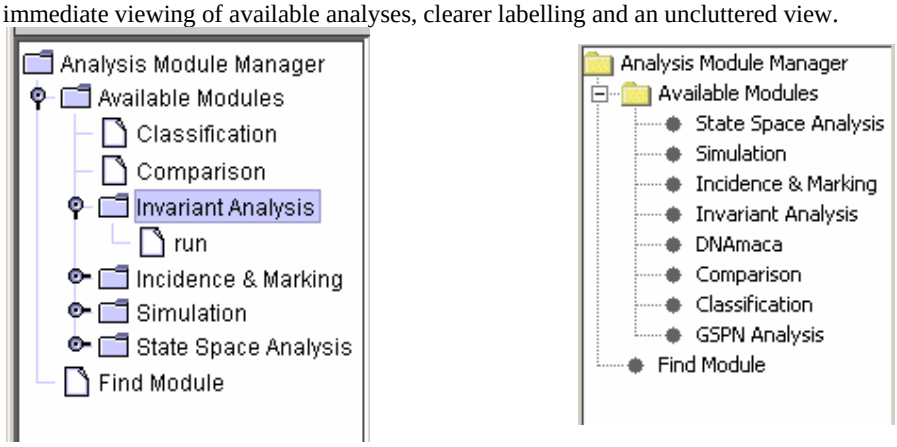


Figure 18 – The old and new module manager interfaces

None of the included modules needed the ability to have multiple independent analysis functions, so this tree simplification algorithm effectively hides this functionality, even though it is still available for future extension modules.

3.16. Grid

It was decided early on to incorporate a 'snap to grid' functionality to facilitate drawing neat nets. This was implemented as a class to contain the grid size, draw the grid and allow components to align to the grid by calling functions to get the nearest grid position for a given *x*, *y* location. The grid cycles through ¼×, ½× and 1× the Place/Transition size, and off. User access is through a menu option and a toolbar button to cycle through the four settings. This provides the necessary functionality without needing the user to specify arbitrary grid sizes, and ensures that components are spaced in proportion to their size.

3.17. Export and printing

Java's enhanced drawing code, known as Java2D, supports drawing of objects in a vector-based format which is intentionally compatible with the PostScript specification. This vector-based specification allows objects to be rendered at varying resolutions without loss of quality. This can be leveraged through Java's printing mechanisms because JVMs supply by default a printer service object which can "print" to a PostScript format stream; by directing this to a file, PostScript compatible files can be created.

By a similar method, Java is able to match the document format with available printers to give the ability to print the current net. This capability is somewhat limited by Java's infamous printing support, which is offset by the ability to print the exported PostScript files giving identical results.

Finally, by rendering the net to an off-screen bitmap, Java's ImageIO classes are able to export it to the PNG bitmap format, which is very widely supported by web browsers.

All of these facilities were implemented in the most generalised way possible, within a static Export class. By doing this, it not only allowed the export functionality to be easily extended from the initial purpose of exporting Petri net images to the ability to export graphs by the same function, but also allows potential re-use of the code in other projects.

The next figure is an example of the output it is capable of.

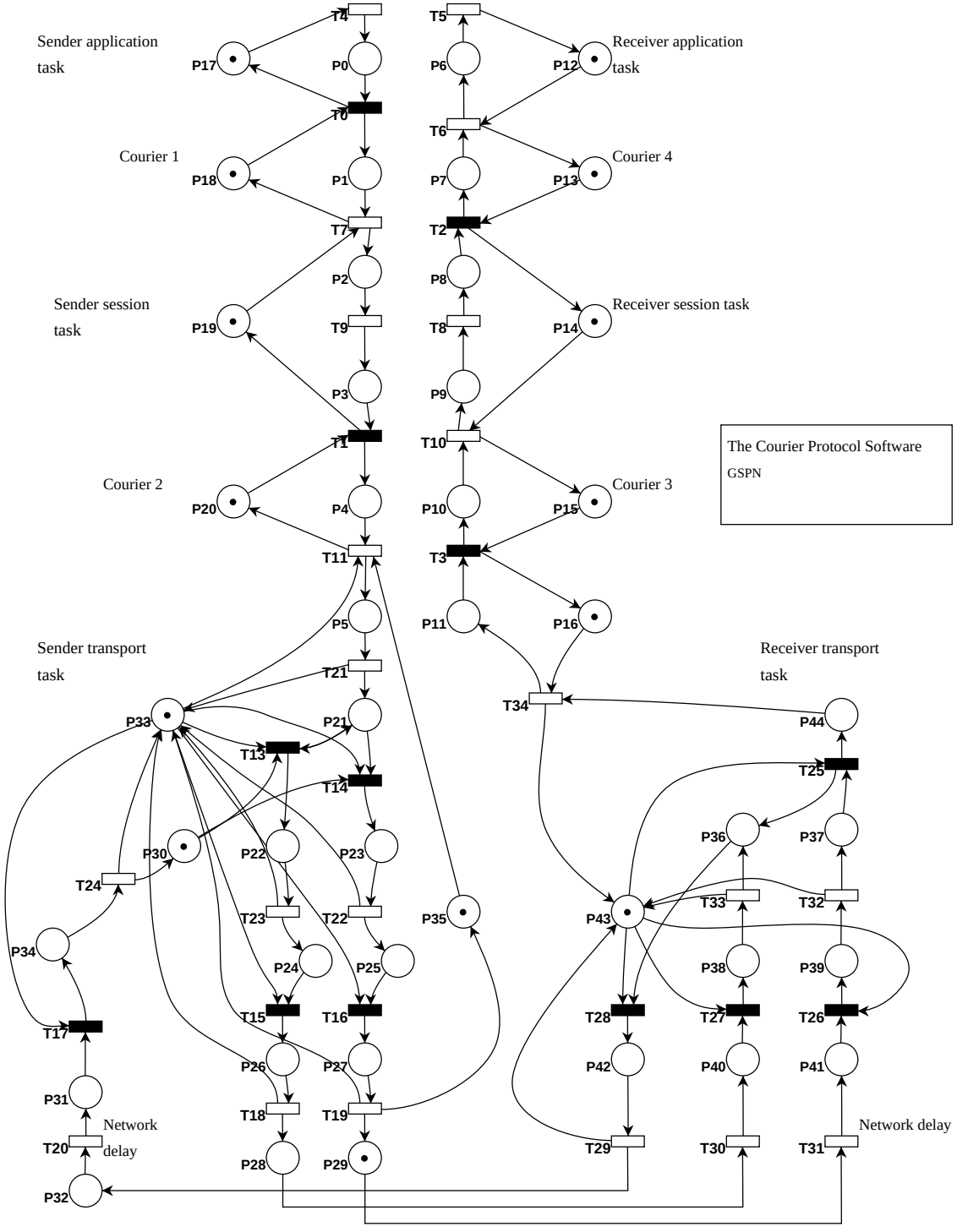


Figure 19 – the Courier protocol

3.18. Example nets

In order to easily demonstrate the usage of Petri nets to represent different concepts, a submenu has been added to the File menu to allow the user to load one of several example Petri nets. This menu is built dynamically on startup by scanning the `Example nets` subdirectory of the program base directory, creating an action object for each file which will load it when performed, and creating a menu item object using this. The example nets include:

- Readers and Writers – models a system in which 3 readers and one writer concurrently access some storage space. The readers cannot access the storage while the writer is writing.

- Simple GSPN – a simple example illustrating the properties of a GSPN taken from Bause and Kritzinger (1995) (p. 180). Primary example used in testing of GSPN module output.
- Dining Philosophers – a representation in Petri net form of the classic Dining Philosophers deadlock example.
- Producer & Consumer – models a producer-consumer system with a limited buffer capacity. The timed transitions T4 and T5 represent the processes of production and consumption. The immediate transitions represent deposit or retrieval of data items from the buffer.
- Courier – a model of the ISO application, session and transport layers of the Courier sliding-window communication protocol (see Dingle, Harrison and Knottenbelt (2002)). Illustrates PIPE's competence in handling substantial net diagrams.

3.19. Animation mode

The original version of PIPE had originally intended to support animated display of net activity through random firing of enable transitions at repeated time intervals. However, it appears this development was stopped after encountering the problems associated with the use of threading with Swing, and reduced to the ability to fire a single random enabled transition instantaneously with a toolbar button.

When implementing true animation, it was still desirable to use threading because it allows the GUI to remain responsive during the animation, which allows animation to be easily terminated. However, rather than using Java's Thread object, the `javax.swing.Timer` class was used. This encapsulates a thread with code to deal with the conflicts with Swing, with the aim of implementing a task at repeated specified intervals; it is therefore ideal for this purpose. When animation mode is enabled, a Timer object is created with an anonymous inner ActionListener object attached, which calls the new `doRandomFiring()` method, as well as maintaining an internal firing counter. By having the random firing encapsulated in a method which deals with all issues (finding enabled transitions, choosing one at random, firing it, and maintaining the data in the DataLayer), it is still possible to have a button for performing a single random firing. The flexibility of the Timer object allows the user to specify the firing delay for the desired effect.

4. ADDITIONAL BUGS FIXED IN PIPE

In addition to the above functionality improvements and extensions, there were many more bugs in PIPE which have been fixed. While innumerable problems were fixed within the course of some other task, without recourse to recording the minutiae of the error and resolution, below are brief mentions of some of the issues resolved.

- Animation history now records random firings

This is an invaluable aid for allowing random firings to be traced backwards to discover the route to a particular state.

- Use of mouse motion listeners to trigger repainting removed

It is apparent that in many of their GUI implementations, the original PIPE team found that there were problems with the components failing to repaint. The reason for this was not determined because, during the course of reimplementing the GUIs more simply these problems disappeared; it is thought that there may be some incompatibilities in the Sun JRE when interfaces are unnecessarily complicated. The manifestation of the problem was that interfaces, especially in the modules, would flicker and, on many occasions, stop flickering in a state where no components were drawn, leaving no interface for the user. The solution was to call the parent frame's `repaint()` method as often as possible; their solution was to do this in a mouse motion listener. The effect of this was to trigger a repaint of the GUI, whether needed or not, every time the mouse was moved, which could be more than 100 times per second. This obviously caused the program to use more CPU time than necessary, and resulted in an unresponsive interface.

- Limitations on number of open files

The original implementation of PIPE used a set of static arrays to maintain pointers to GuiView, DataLayer and File objects for the currently open nets. Its algorithms for handling the creation of new tabs were simplistic, simply shuffling arrays to remove gaps and silently refusing to open more than five tabs, or to allow the final tab to be closed. Its algorithm for determining the name for new tabs was afflicted with this, as it would simply append the string "PetriNet" with the index of the array index into which it was placed, often giving multiple tabs with the same title.

By replacing the arrays with ArrayLists, the restriction upon the number of open tabs was removed. All code dealing with management of references had to be reimplemented, which proved to be a large task due to the level of coupling between classes; but it served as a useful mechanism for exposing this coupling in order to reduce it. There were, in particular, many issues when the final tab was closed, as many functions would then return null references which were not handled by the calling code. These were ultimately remedied through testing.

- Duplication of tab creation code

The process of creating a tab is relatively complicated, yet the difference between creating a new, empty pane and opening an external file is small. This code was therefore consolidated into a single function, which simply creates an empty tab if the filename parameter is null.

- Assumption of working directory

Much of the code in the original PIPE, and some of the code in the current version, makes references to files stored within the same directory sub-tree as the binary program files. For example, button images and example nets are stored this way. It is possible to run a Java program by defining "class paths" beneath which to search for the class files; however, it is not simple for the program to determine which class path it is executing from within. PIPE made use of simple relative file references, which the operating system will append to the current working directory; so, if the working directory was wrong, PIPE would be unable to load any button images or even its XSL files for loading files. This was remedied by determining the class root directory for an instantiated PIPE class (the CreateGui object) through the Class object's security protocols. It is now possible to run PIPE with a command such as:

```
java –cp ~/somefolder/PIPE RunGui
```

with no adverse effects.

- Better handling of file save prompting

In multi-document programs, it is common to prompt the user to save any modified files before closing them, and PIPE is no exception. However, when this check and prompt is done on shutdown, it is usual to be able to abort the shutdown by choosing a 'Cancel' option. This was added to PIPE by modifying the default behaviour to require explicit window destroying, and to not do this if aborted. This also allowed handling of situations where the user opts to save a modified file, but then cancels the file browser.

- Removed HandlerFactory class

While the usage of factories can be good practice because it allows reuse of objects rather than creating duplicates, the factory for mouse handlers in PIPE did not have this ability and as such was pointless, simply serving to slow down execution.

- Invariant analysis module now numbers objects correctly

It was numbering them from 1, whereas everywhere else they are numbered from 0.

- Object label editing cancelling no longer labels the object as “Null”
- Fixed null pointer exception when clicking anywhere but on a node in module tree
- Z ordering of net objects

By drawing net objects onto a Swing JLayeredPane rather than a JPanel, it was possible to enforce Z ordering to give clear separation of objects that should be behind others. For example, arc points should be drawn above arcs and places. This ensures that nets look better and – more importantly – that intersections and containments of objects by other objects are detected when they should be.

- Static CreateGui object

The CreateGui object contains methods for accessing data on the currently open nets and obtaining references to them. However, this meant that it was necessary for every class that might need to obtain a reference from it, to be give a reference to the CreateGui object itself. Since there must be one instantiated for the application to run, and there is no need for more than one to be instantiated, it was modified to be a static class with static methods and member variables. This effectively made it globally accessible and greatly reduced the level of reference passing between classes.

- Forced update of each JComponent at each GuiView repaint event

During the initial review of the PIPE codebase, some odd behaviour was observed relating to the redraw events occurring in the main drawing container, GuiView. For each update event, PIPE would explicitly remove every PetriNetObject from the GuiView. It would then loop through all the PetriNetObjects, restoring them to the GuiView. The only apparent point to this behaviour, which would force a repaint on all contained objects, was to effect the removal of deleted components since they would not be restored.

The solution to this behaviour was to remove all of the above mentioned code, and add code to the delete actions to remove the PetriNetObject from the GuiView container.

- Clipped Drawing of PetriNetObjects

A cosmetic issue with the original version of PIPE was the clipping of the display of PetriNetObjects. This is a result of setting the object bounding box to being the same dimensions as the object itself. Since Java2D drawing primitives are anti-aliased, many of the PetriNetObjects were actually drawing slightly outside their bounding box, and being clipped in the process.

The solution to this problem was to define a constant inset for all bounding boxes of a few pixels, allowing each component to draw successfully.

5. CONCLUSION

5.1. Product evaluation

One of the primary goals of this project was to take an existing program which provided basic functionality, and add the extended functionality which would make it actually useable. The original version of PIPE fell badly short in the areas of presentation (the ability to create professional looking diagrams) and the UI flexibility required of a modern drawing tool.

In these areas, the second version can be considered a success. PIPE now possesses an interface which anyone familiar with the standard drawing UI can pick up and use without application specific knowledge. All of the more counter-intuitive aspects of the previous version have been removed (e.g. the requirement to position places and transitions prior to attaching arcs), and the interface has been streamlined where possible (e.g. consolidation of the move/delete tools into one selection tool).

With regards to the ability of the program to produce professional looking diagrams, every specified goal has been met. Some changes have been trivial such as reverting places to using an unfilled circle, while others have involved major engineering effort such as multi-segment curved arcs. The result is that Petri nets created in PIPE are now of sufficient quality to be included in a publication. The important addition of PostScript output puts PIPE in the position of actually being a useful tool, where previously it had neither the performance to use for serious analysis, nor the capability to output a diagram.

While much of the work specific to Petri net theory had been done by the previous group, significant advances have also been made here. The addition of GSPN support, and the creation of two modules capable of performing analysis on them, opens PIPE up to a whole new class of problems that would have been otherwise inaccessible. The GSPN analysis module took considerable research time due to its theoretical basis, but is now a useful reference tool for smaller nets. The DNAmaca module achieves its goal of masking the underlying inter-process communication involved with an external tool and allows convenient and direct access to powerful C++ based passage time analysis.

Aside from the new features already detailed, a large amount of time was spent fixing various problems with the original version. These ranged from analysis modules producing incorrect results, to more trivial problems such as display issues. Many of the fixes were required to add more advanced functionality. Some changes were made to ensure consistent cross platform behaviour, and the product was tested extensively under Windows, Linux & Mac OS JVMs.

Finally, work was also put into the presentational aspects of PIPE. Button icons with an alpha channel ensure PIPE looks equally at home on different platforms. Several example nets were created, providing a useful starting point for those new to Petri nets or wishing to evaluate the product. The multitude of independent GUIs created for the various modules were replaced with standard components to ensure consistency.

In summary, the great majority of features detailed in the initial specification have made it into the final version of the product. PIPE has progressed from being the somewhat limited application of the first version to a useable tool.

5.2. Further work

There were some planned features that were not implemented due to lack of time. Other features were discussed but not timetabled for inclusion due to the limited timescale. These are summarised below with a short description of the planning that was done for them.

5.2.1. Hierarchical nets

Hierarchical nets are a useful way of building complex nets by representing common sections by a “black box” with inputs and outputs, which can be expanded to show the contained sub-net. This was anticipated to be handled by defining an additional Petri net object type, which would be an interface object. Such an interface would be a source or destination for arcs in a sub-net, providing the location for inputs and outputs with the enclosing net. An additional object would then be used to represent a sub-net, providing input and output locations corresponding to the interfaces in the sub-net. The sub-net could be edited by double-clicking on this representative object, which would open the sub-net in a new tab.

PIPE would not require substantial changes to support this, as demonstrated by the number of new net objects added and the improved ability to deal with multiple tabs. The demand for it is questionable, as there seem to be few significantly large common net sections, and it is probably easier to draw nets without hiding sections since this may lead to errors. There are also issues regarding how such nets would be animated.

5.2.2. Copy and paste

The ability to cut, copy and paste individual net elements, or entire sections of the net, was suggested as a useful feature. It was decided that, due to the highly specific nature of the data, it would not be necessary to use the global system clipboard, but instead a local mechanism could be used. The anticipated implementation was to extend all net objects with serialisation functions, allowing them to save and load themselves concisely to and from a memory stream, including dealing with links between objects. This memory stream could then hold an arbitrary number of objects, which could be recreated in other nets or elsewhere within the same net. However, this serialisation functionality would have to be complete, with implementation in all net objects, for it to be useful. Due to the length of time for which some net objects, particularly arcs, had non-final internal representations, it was not possible to implement this with enough time to be sure of completion.

5.2.3. Undo and redo

The serialisation of objects would also simplify the ability to undo and redo their creation and modification, as a traversable history of changes could be maintained with the addition of some container objects for some group actions such as paste and group deletion. As such, it was an extension of copy and paste and to be implemented after it.

5.2.4. Contextual icon bar controls

It was hoped to implement some contextual buttons which would appear next to the selected object, giving controls for modifying its properties; in particular, it was thought that a place could have buttons for adding and removing tokens, removing the need for those two “modes” in the GUI. While such functionality is available contextually through the popup menu, there was not time to develop a suitable control for this; it is not anticipated that this would be complicated, but since it would essentially be an extra way of doing something already available, it was not a high priority.

5.2.5. Adding points to arcs

It is not currently possible to add extra points to an existing arc that has been connected between a place and transition. The difficulty is in determining the position at which to place the added point; it would ideally be possible to add an extra point at an arbitrary location on the arc, without modifying its shape. For straight line segments this is straightforward; for arcs it requires complicated solving of cubic equations in order to determine the required control points, which would then be different from the values they would have were the arc created sequentially. In addition to the late completion of curved arcs, this made it unfeasible to implement.

5.2.6. Contextual cursors

Because there is still a level of modality in the editing interface, it would be helpful to emphasise the current mode to the user. It was suggested that this would be best achieved by modifying the mouse cursor to indicate the mode; perhaps showing a small place by the arrow when in place mode, for example. However, while Java supports easy creation of icons through its IconImage class (albeit with varying degrees of success across platforms), there is no corresponding class for cursors. This is probably due to the limitations commonly seen in cursor support on various platforms; for example, Windows XP only supports cursors that are 32 pixels square. Custom cursors are therefore handled on a platform-by-platform basis, which is not in keeping with PIPE’s cross-platform nature. Common system cursors such as a crosshair and I-beam are supported cross-platform so they are all PIPE uses.

5.2.7. Passing through extra XML fields

The PNML file format is XML-based, making it highly extensible as programs are free to add any fields they wish to their output files. However, some of them can fail to load the file if these custom fields are removed. It would be preferable if PIPE could maintain these extra data loaded from a PNML file and save them back unchanged; this would make it more likely that the originating program could load the file. However, maintaining this data is not simple, especially if its contents are dependent on other net data. It would require a data structure associated with every net object, as well possibly unassociated blocks, maintaining the tree of additional data.

5.2.8. Mac OS X properties file

Some research was done on making PIPE behave in a more native way under Mac OS X, as Java specifies certain limitations on its behaviour. It was determined that it is possible to override this to provide features such as moving the menu bar from the window to the screen top, through an additional properties file stored along with the executable files; however, this is not necessary for it to be functional and was therefore not a high priority.

5.2.9. Response time analysis

It was thought that it may be possible to write an additional response time analysis module for GSPNs. Initial research was done into the algorithms required; however, completion and testing of the GSPN module was required before adding this and the available time for additions was reduced from initial expectations as the problems with the original PIPE became clear. It was also thought that Java would not be an efficient platform for calculating this, and may be better suited to optimised tools.

5.2.10. Selection glow

Selected objects are currently drawn using hard-coded colours because it is difficult to make use of the system's colour scheme when it is applied to monochrome Net objects. It is quite possible for the system's selection colour to be white with a black background, or black with a coloured background; in this case, it is difficult to distinguish between a monochrome unselected and coloured selected object without analysis of colours. One possibility would be to instead represent selection with a "glow" or surrounding border, which would not be present on unselected objects and therefore easy to distinguish. This could be implemented using Java Strokes.

5.2.11. Handling of occlusion of labels

Place and transition labels, as well as arc weighting labels, can become occluded, especially on more complex nets. It may be preferable to allow the user to manually reposition them; a better solution would be an algorithm capable of determining an optimal position for the label on a locus based on the associated object. This might be a tricky problem to solve.

6. APPENDICES

6.1. References

Aalst, Best, E.: *Application and Theory of Petri Nets 2003*, 24th International Conference, ICATPN 2003 Eindhoven, Netherlands (2003)

Bause F.; Kritzinger P. S.: *Stochastic Petri Nets – An Introduction to the Theory*, unpublished manuscript (1995)

Dingle N. J.; Harrison P. G.; Knottenbelt, W. J.: *Response Time Densities in Generalised Stochastic Petri Net Models*, Proc. 3rd ACM Workshop on Software and Performance (WOSP 2002), Rome, Italy, pp. 46-54 (2002)

Dingle N.: *Production of the Extensible Petri net Editor/Animator 'Medusa'* – MSc project, Imperial College London (2001)

Hall, M.; Brown, L.: *Core Web Programming, 2nd Edition* – Sun Microsystems Press/Prentice Hall PTR (2001)

Hunter, D.: *Beginning XML* – Wrox Press (2001)

International Data Group *JavaWorld* – http://www.javaworld.com

Bloom J.; Clark C.; Clifford C.; Duncan A.; Khan H.; Papantoniou M.: PIPE homepage – http://petri-net.sourceforge.net

Kazmierczak, M.: mkaz.com Linear Algebra section – http://mkaz.com/math/line_alg.html

Knottenbelt, W. J.: *Generalised Markovian Analysis of Timed Transitions Systems* – MSc thesis, University of Cape Town, South Africa (1996)

Oh, H.R.; Chung, W.H.; Kim, M.: *Transformation of Timed Petri Nets for Response Time Estimation* in IEEE Proceedings (Computers and Digital Techniques), Vol. 137, No. 1, pages 74-80 (1990)

Petri, C. A.: *Kommunikation mit Automaten* – PhD thesis, Universität Bonn (1962)

Sun Microsystems Inc: *Java API Documentation* – http://java.sun.com/j2se/1.4.2/docs/api

Sun Microsystems Inc: *Java Tutorial* – http://java.sun.com/docs/books/tutorial

Unknown authors: *Petri Nets World* – http://www.daimi.au.dk/PetriNets

Weisstein, E. W.: MathWorld – http://mathworld.wolfram.com

6.2. Group Work

The project tasks were divided among the group members, and assigned timeslots for their completion, as shown below.

Task	Assigned to*	Week number									
		1	2	3	4	5	6	7	8	9	10
Background reading	123456										
Preliminary code examination	123456										
Code cleanup	12345										
JavaDoc	12345										
Editing tools	125										
Report-I preparation	123456										
GUI	24										
Documentation	6										
GSPN implementation	3										
Report-II preparation	345										
DNAmaca	5										
Optimisation	1245										
Bug fixing	245										
Final Report preparation	123456										

Grey cells indicate allocated timescales; dark grey cells indicate deadlines for reports. This timetable was adhered to as far as possible, with some scope for work to be done outside the timetable as required. Deadlines were set for sub-tasks as required.

Credit for individual added features can be broadly allocated as follows:

Feature	Main contributor*
Curved arcs	1
DNAmaca interfacing	5
Export and printing	4
GSPN analysis	3
Selection	2
Grid	5
Module GUI	4
Annotation notes	2

Almost all tasks were the result of co-operation between group members, and the result of detailed discussions in meetings. Very few implementations were done in isolation, requiring intercommunication between group members. The difficulty of implementing some features was not proportional to their perceived difficulty or the volume of code produced. Much of the work done was not on clear-cut extensions to PIPE.

6.3. Meeting minutes

It was decided before the first formal meeting that, because there was no group member with a strong desire to perform secretarial duties, and the task is neither difficult nor requiring much research, this responsibility would be shared by all members in a turn-based system. This proved to be a useful way of dividing the chore of an unwanted task; the only problem encountered was with maintaining a fast turnaround between meetings and minute publication, as well as an inconsistency in minute formatting.

6.3.1. Minutes 15/1/04

All members of the group present, minutes taken by tb403

Subjects discussed:

* Group members:
1. Tom Barnwell
2. Michael Camacho
3. Matthew Cook
4. Maxim Gready
5. Peter Kyme
6. Michail Tsouchlaris

- Possible IDE's to be used, it was noted that JBuilder is available for use on departmental machines but not for free download for home use, visa-versa for Eclipse. Decision reached for everyone to try out the possible alternatives and take a final decision at the next meeting.
- The need to take a closer look at last year's project in order to see where it can be improved. PK suggested that each member of the group take one sixth of the existing code, work out roughly what it does and describe this to the group at the next meeting. PK agreed to divide up the code between the group.

Next meeting scheduled for Monday 19/1/04.

6.3.2. Minutes 19/1/04

Attendees - all group members present

Minutes taken by Matthew Cook

Agenda

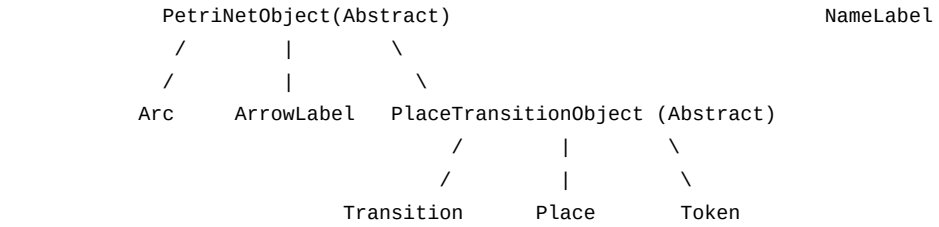
- Each group member to describe to the group their understanding of the section of existing PIPE code allocated to them for analysis over the weekend.
- AOB

Findings as follows:

Pete:

DataLayer/*

These classes outline the various objects that make up a Petri net. The basic inheritance structure is:



So we have an abstract PetriNetObject at the base of the hierarchy, defining base properties like colour, moveable status. Then there's Arc & ArrowLabel. Arcs link to PlaceTransitionObjects, and ArrowLabel objects are the actual arrow that is drawn in the centre of the Arc indicating direction.

The token carrying objects (Places, Transitions and Tokens) have a base class PlaceTransitionObject which is abstract.

All the instantiable classes all have a paintComponent method for drawing themselves (except the tokens of course which are drawn by the places).

The NameLabel class describes labels of PetriNetObject objects.

Maxim:

classification.java

Classifies Petri nets into the six categories found on page 122-3 of the Book.

Has a method for each type returning a bool if it is that type (a net can be a combination of types, in general later ones in the list are supersets of earlier ones). All can be determined from the forwards and reverse incidence matrices, which it pulls from the dataLayer object. It seems to be wrong in several cases. Much possibility for cleanup.

comparison.java

Compares nets, either file to file or current to file (no option to compare two open nets). Seems highly pointless. Compares attributes of place, transition and arc objects - position, name, ID, tokens. Issues with how it does that (dumbly), I'm inclined to say it's pointless except for getting a diff after slightly modifying a net. If we keep it, it needs cleaning.

Matthew:

Datalayer.java

A large class that contains all the details necessary to describe and maintain Petri Net objects. Has arrays to store each type of element present in a net (arrow, place, token etc.), and methods to support adding, removing and getting these elements. Constructors either empty or take an existing PNML file

as arguments. Also contains functions to describe the behaviour of the net when transitions are fired, and details of incidence matrices.

Tom:

Simulation.java/Statespace.java

Documentation poor, lots of code commented out. There appeared to be duplication of GUI functions here.

Michael:

InvariantAnalysis.java/Matrixes.java

These perform analyses on the Petri net objects, but seemed to do lots of GUI work as well. Unsure if the algorithms used are actually correct.

Other points discussed

- General consensus that a number of improvements can be made with the PIPE GUI (e.g. transition alignment, changes to arc behaviour, changing HTML output to stylesheets rather than having each module formatting its own output).
- A library of example nets would be beneficial for demonstration purposes.
- Uncertainty about what direction to take with any further analytical modules.
- The group has to decided to use Eclipse as IDE for this project.

To do

- Speak to Will Knottenbelt re suggested directions for the project [Maxim].
- Speak to James Bloom re gaining access to the PIPE Sourceforge page [Maxim].
- Speak to CSG re installing Eclipse on the network [Maxim].
- Continue familiarisation with Java and Petri nets [all].

6.3.3. Minutes 30/1/04

Agenda

Next steps following successful completion of REPORT-I; AOB

Topics covered

1. Agreed to schedule regular meetings for Mondays and Fridays at 12:00 in the SCR unless otherwise arranged.
2. Initial allocation of work to be done:
    - Maxim: checking and fixing of existing codebase; response time analysis implementation in Java
    - Tom: editor issues
    - Matthew: addition of GSPNs
    - Pete: editor, also DNAmaca implementation (dependent on difficulty, to be ascertained)
    - Michael: invariant analysis
    - Michail: mathematical specialisation
3. Work more appropriate to be done later:
    - Documentation
    - Hierarchical nets
    - All other issues not covered
4. Immediate goals:
    - Draw up specifications for preliminary additions and changes as per above allocations
    - Draw up lists of functions in existing modules which are candidates for movement to a higher utility class to avoid code repetition
    - Get everyone added to the PIPE SourceForge account
    - Get everyone set up in Eclipse, working with Java and set up for CVS access to PIPE@SF
5. Eclipse @ DOC

- Having emailed instructions in accessing the Linux version to the group, Maxim has asked CSG about getting a Windows version on there. Help was offered with importing PIPE into Eclipse for those having trouble.

6. Existing editor framework
    - Given that the existing editor is encapsulated by an object, its functionality should be examined to determine if it is suitable for extension for all of the desired improvements or if it will ultimately be better to rewrite substantial sections of it.

Notes taken by Maxim

6.3.4. Minutes 13/2/04

Agenda

- Progress with current work (all)
- Tasks for next week (all)
- Report 2 (Maxim)
- AOB

Since the last meeting, group members have worked on the following areas:

Maxim

- Created new GUI buttons.
- Added scrolling functionality to display window.
- Tidied up module display widgets.
- General code cleanup.

Pete

- Replaced existing arc class with multi-jointed arcs.
- Converted the guiview to a JLayeredPane to allow use of depth.
- General code cleanup

Tom

- Work on algorithms for curved lines.

Matthew

- Investigating problem with arc display.
- Ongoing development with GSPN analysis.

Michail

- Attempted analysis of state-space module.

Michael

- Selection of elements.

Tasks for next week are as follows:

- Maxim - start work on Report 2
- Pete - continue working on multi-segment arcs, adding curve points to arcs, snapping arcs to place-transitions.
- Tom - finish multi-point curve algorithms.
- Matthew - continue GSPN work.
- Michail - start working on Javadoc/user guide.

Report 2

This is due for submission next week. Maxim will prepare a draft over the weekend and advise the group what each member needs to provide by the next meeting.

6.3.5. Minutes 16/2/04

Agenda

- Work done over the weekend
- Report-II
- Work for the week ahead
- AOB

Topics covered

1. Maxim:

- Drafted Report-II. Needs some sections to be expanded by appropriate people and any other comments; this is to be done over the next few days in time for the Friday deadline. This will be done through email.
- Work ahead: continuing to implement new GUIs for modules; rewriting the GUI tab interface to support arbitrary numbers of tags - using Java Collections rather than static arrays; finishing Report-II.

2. Michael:

- Implemented Delete actions without a discrete mode.
- Work ahead: Selection mode has some issues to be resolved. Module manager needs an overhaul to behave in a less silly way.

3. Matthew:

- Implemented and tested GSPN probability matrix
- Work ahead: sojourn times for GSPNs, possibly completing GSPN functionality by the end of the week.

4. Pete:

- Segmented arcs 90% implemented, just a few issues remaining. Arc points now show when selected and can be moved. Added Tom's curved arcs to the GUI.
- Will now switch to work on DNAmaca interfacing.

5. Tom (absent):

- Arcs

6. Michail:

- No work over weekend.
- Continuing to work on documentation.

7. Other issues to be resolved

- PNML compatibility

  The program does not support copying any unknown data fields from loaded PNML files. This data should be passed through untouched as it breaks compatibility with other Net apps.

- Labels

  An annotation label object should be implemented. Possibly using a JEditorPane for formatted text. PNML supports arbitrary (unlinked) labels.

- Timed transitions

  These need GUI side code to be improved; and need to be drawn differently to untimed transitions.

- Copy/paste/undo/redo

  Some discussion on how to implement this. The plan is for net objects to have some common methods to save and load their representations from an in-memory format, which will then be stored in a linked list for undo/redo and also for clipboard operations. Undo/redo will also need some representation of actions too. The necessity for external clipboard functionality for copying/pasting Net objects is dubious so it may not be worth the effort. Discussion of some kind of multiple object/action container object for encapsulating many actions, eg. paste may include many object creations.

- Response time analysis, DNAmaca

  RTA will probably be done by DNAmaca, reducing the need for an internal Java version. Pete is working on DNAmaca now.

- Printing/save to PNG

  "Simply" a matter of rendering the net to a different canvas, probably not too hard. Task will be assigned when someone suitable is free.

- Add/remove token local toolbar

If, when a place was selected, buttons appeared for adding/removing tokens, it would remove the need for those modal operations, which all agree is A Good Thing. As previous item.

- Use of SourceForge tracker facilities

  Sourceforge offers a bug/feature tracker allowing tasks to be described and assigned to group members. This is a good way of making sure nothing is missed and work is not duplicated, so all agreed to use this, to check it regularly and to assign tasks to themselves and others suitably.

6.3.6. Minutes 20/2/04

Agenda

- Work done over the weekend
- Report-II
- Work for the week ahead
- AOB

Topics covered

1. Maxim:
   - Finished: Report II write-up; various fixes; new Gui for Statespace module.
   - Work ahead: Toggleable toolbar buttons and corresponding radio buttons for menu; de-sillify the first tab once and for all.
2. Michael:
   - Finished: Various bugfixes; CreateGui uses static methods and variables so references to it are no longer required.
   - Work ahead: Various bugfixes, plus context buttons for +/- tokens.
3. Matthew:
   - Ongoing work on GSPN module - almost finished.
4. Peter:
   - Ongoing work on DNAmaca interfacing. Going well.
5. Tom:
   - Arc stuff; Move transition rotation to right-click menu; printing functionality (additionally export to postscript/png).
6. Michail:
   - Continuing to work on documentation.
7. Other issues to be resolved
   - Timed transitions should be unfilled.
   - New right-click delete option.
   - Arc weights need to be displayed alongside arcs.
   - It should be possible for the user to add annotation labels to a Petri net.

6.3.7. Minutes 23/2/04

Progress

Maxim

- Added toggling toolbar buttons.
- Removed limit on number of tabs and disposed of initial tab.
- Modified ‘about’ dialogue box.
- Investigated different file browser with a view to improving appearance for Mac users. Alternative class looked ugly in Windows so will probably not be used.
- General code cleanup.

Michail

- Investigating user documentation.

Pete

- No PIPE work since Friday.

Michael

- No PIPE work since Friday.

Matthew

- Added Labels array to dataLayer (to support net-level labelling). Some problems with DOM to be resolved.
- Debugging of existing PNML code (not possible prior to this weekend due to saving/loading/editing bugs). Fixed incidence matrix bug.

6.3.8. Minutes 27/2/04

Present: Pete, Tom, Max, Michael, Matthew

Taken by Pete

Tom: Has been ill this week and hasn't done any group or coursework.

Max: Has been working on gui stuff, working through the various modules.

Pete: Suggested that it may be more efficient for Max to do the basic dnamaca module gui since he has done the other ones.

Max: Bug in datalayer on loading, looking for xsl files in current directory.

Matthew: Has been working on xsl files, doesn't know why it can't find them though.

Matthew: will look at fixing.

Matthew: has been working on xsl sheets. Added label support in datalayer and xsl sheets.

Max: Comparison module seems fairly useless, but is now fixed.

Matthew: Now has working results for the exponentially weighted firing time. Will need to include external source for this in tree.

Michael: Not done a lot, going to be working on arc weighting label. Would like to have the weighting label draggable along the arc.

Max: Would like the double click to rotate transition changed, possible to right click menu.

Pete: Has been working on dnamaca. Had a meeting with Nick Dingle to discuss problems running dnamaca. Turns out it only runs under Linux. Am now working on running dnamaca as an external process and parsing it's stdout.

Max: Going to be working on printing/exporting

Matthew: Will want some info on arcpaths for saving.

Max: Everyone needs to try and have at least their top priority bug fixed by the end of the weekend, preferably more.

6.3.9. Minutes 1/3/04

Work done over the weekend:

Pete:

- Much arduous hacking of DNAmaca
- Dealing with thread issues in Swing in order to throw up a progress box
- Working on spacing arc transition entry points, proving annoying
- Noted that curved arc entry is still not available

Matthew:

- Modified XSL so that transition orientation is stored
- Completed steady state distribution calculation including transition throughput

Michail:

- Looked into the complexities of HTML in preparation for writing in it

Tom:

- Worked on displaying weight on arcs
- Improved rotation of transitions
- Worked on the ability to hold shift and have the arc preview update accordingly

Michael:

- Arc points are now selectable and draggable
- Setting of weighting now works
- Fixed loading of weighting from file
- Added proximity detection for snapping

Maxim:

- Added PNG export
- Added PostScript export
- Added printing support
- Added example files menu
- More menus for features in general

Work to be done during the week:

Pete:

- Continuing arc transition entry points
- Making DNAmaca work with our nets
- Displaying results from DNAmaca
- Making the grid work better with existing unaligned objects

Matthew:

- Tidying of GSPN code
- Add GUI to GSPN module
- Add arc points saving into XSL

Michail:

- Documentation, especially Invariant Analysis

Michael:

- Popup menu to delete an arc point and toggle its mode
- Adding points to arcs
- Selection glow
- Annotations
- All other issues already on bug tracker

Tom:

- Finish showing weighting on arcs
- Adding keyboard event handler for shift -> arc curve preview
- Arc end point angle calculation for improved arrow head orientation and place entry location
- Supplying data for arc point saving
- Make Delete key delete the last point drawn before arc is completed

Maxim:

- Fix double toggle of object toolbar buttons leaving no button apparently selected
- Delete in-progress arc when object mode is changed
- Other issues on bug tracker
- Work on final report

6.3.10. Minutes 5/3/04

Agenda

Discussion of work completed since the last meeting, and work to be done during the weekend

Topics covered

Work covered during the week:

Maxim:

- fixed bugs with arc mode, mode buttons, animation mode breaking other modes (due to a conflict between mode and object type)
- restored right click menu to module manager, removed beep,
- changed selection colours for places & timed transitions
- saw wjk, demonstrated project, wjk suggested separate buttons for timed & immediate transitions, this implemented

Pete:

- finished code for positioning arcs arc ends on the edge of places & transitions
- continued with DNaMaca (?), now produces correct output for nets generated with PIPE, added some safety checks so that incorrect arcs are not analysed
- now looks for DnAmACa on linux path

Michael

- arcs now snap to nearest place/transtion when drawing
- fixed assigned bugs
- preliminary work on annotation labels
- right click menu now allows user to toggle point type

Michail

- continuing writeup of HTML documentation and help files

Matthew:

- added display for GSPN analysis module, now displays tables of reachability sets etc
- proceeding with code to save arc path point details and transition angles into PNML

Tom:

- substantial rewrite & cleanup of code for creating curved arcs
- fixed code to correctly angle arcs at places and transtions
- added preliminary code for displaying arc weightings when not equal to one
- shift key now toggles the end point of any arc being drawn

To do:

Maxim:

- Start writing report, ensure other members of the group do too

Pete:

- Add code to display graphs of DnAMAcA module output
- start writeup

Michael

- annotation functions
- start writeup

Michail

- complete writeup of HTML documentation and help files

Matthew:

- finish functions for saving and loading arc path points and transtion angles
- add a sizeable example file

- start writeup

Tom

- finish positioning of arc weight labels
- finish code to fix curved arc segments at right angles to transitions
- start writeup

Notes taken by Tom

6.3.11. Minutes 8/3/04

Maxim:

- Fixed png export bug.
- Fixed invariant analysis module gui.
- Fixed simulation module maths bug.
- added shading to the table widget.
- added help window & toolbar/menu icon.
- scaled printing down to fit net on page.

Matthew:

- Added Saving arcpaths is there, loading not there yet.
- Waiting on Tom to add function to receive array of points.
- Added Saving/loading of annotations.
- Created courier example net.

Michael:

- Added support for annotations
- General bug fixing

Maxim: Printing bug of annotations, could we change from courier for font.

Pete: Possible auto resize when typing.

Maxim: Snap to grid a bit illogical? Maybe snap to top left?

Tom: Been working on coursework this weekend, no pipe work done.

Maxim: Asked everyone to do write up, specifically do write up for areas they “own” by Friday.

Discussion regarding mouseover highlighting of end/start arcpoints.

Pete:

- Added graphing of results to dnamaca module, fixed various dnamaca bugs.
- Completed work on arc end point positioning.

6.3.12. Minutes 12/3/04

Maxim:

- Drafted a substantial part of the report, individual sections required saturday am.
- Updated Sourceforge website.
- Fixed several bugs: changing grid size changes annotation size, saved grids now load with weighting, arcs 'wobbling' whilst being drawn, positions of components in saved grids now load correctly, null-pointers on arc deletion ironed out.

Matthew:

- Written up some sections of report.
- Added some html navigation to GSPN module front end.
- Fixed exception to catch oversized nets in analysis module.
- Fixed aspects of loading arc paths from saved nets

Michael:

- Fixed several annotation bugs
- Fixed aspects of loading arc paths from saved nets

Michail:

- Continued update of html help and user documentation

Tom:

- Fixed arcs so that curved end segments meet transitions at right angles
- Modified positioning of weight labels so they remain closer to curved arcs

Pete:

- Written up some of the report on DNAmaca
- Misc improvements to DNAmaca