PIPE

Platform Independent Petri-net Editor

FINAL REPORT

17th March 2003

James Bloom
Clare Clark
Camilla Clifford
Alex Duncan
Haroun Khan
Manos Papantoniou

Supervisor: Dr W. Knottenbelt

Acknowledgements

Many thanks to Dr William Knottenbelt for his patience and support and to Nick Dingle and Michelle Osmond for the time they spent testing PIPE.

Contents

Abstract

Petri nets are a popular graphical way of modelling concurrent systems such as communications protocols, multiprocessor computers etc. With Petri nets it is possible to assess the correctness of systems - for example by verifying that the system cannot deadlock, that there cannot be any buffer overflows etc. Stochastic Petri nets extend ordinary Petri nets by incorporating time. This allows designers to use the same formalism to reason both about the correctness and the performance of systems.

This report discusses the design and implementation of PIPE – Platform Independent Petri Net Editor to edit, animate and analyse Petri nets. Analysis modules may be written applying a module interface to demonstrate the run-time loading and unloading of the modules. Seven modules were written to demonstrate this capability, including integration of the Predator Invariant Analysis module. Other modules included are Invariant Analysis, Simulation, State Space Analysis and Comparison and Classification modules.

The latest XML Petri Net Markup Language was used so that PIPE fully supports all types of PNML standard Petri Nets.

1.0 Introduction

Introduction to Petri Nets

Petri nets were invented in 1962 by Carl Adam Petri and are a formalism for the description of concurrency and synchronisation inherent in modern distributed systems. Since first described by Petri, many variations of his original nets have been defined. Apart from modelling and formally analysing modern distributed systems, Petri nets also provide a convenient graphical representation of the system being modelled.[1]

Place-Transition nets are the basic form of net, often referred to as "ordinary" nets. The graphical representation of a place-transition net contains the following components:

Places:	These are shown by circles and model conditions or objects, for example a program variable. Places may contain one or more tokens.
Tokens:	These are represented by white dots and represent the specific value of the conditions or objects. This could be the specific value of a program variable. Tokens are contained within places.
Transitions:	These are shown by rectangles and model activities that change the values of the conditions and objects.
Arcs:	These are lines that show the interconnections of places and transitions, and hence show which objects are changed by a given transition.

In the place-transition net, a place can only be connected to a transition (by an arc) and vice versa, such that no two transitions or places can be connected to each other. An example of a place – transition net is given below.

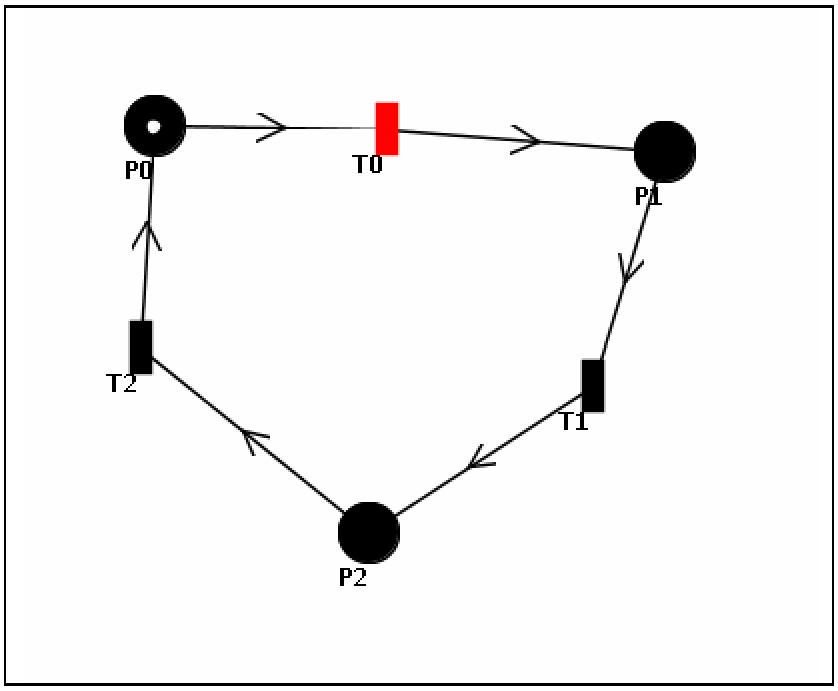


Figure 1.1

Places and transitions might have several input or output elements – there is no restriction on the number of these that can be present. It is also possible to have isolated places or transitions that are elements that are not connected to any others. These do not influence the rest of the net and can be neglected.

Another important concept in describing Petri nets is the idea of arc weighting. This has a significant effect on the dynamic behaviour of a Petri net as it influences which transitions are enabled, and the results of firing these transitions. The arc weighting is displayed next to an arc in the Petri net diagram. If no arc weightings are present then the default value of 1 is assumed (as in Figure 1.2 and Figure 1.3). A transition is enabled if all its input places contain at least the number of tokens specified by the arc weighting on the arc connecting each input place with the transition.

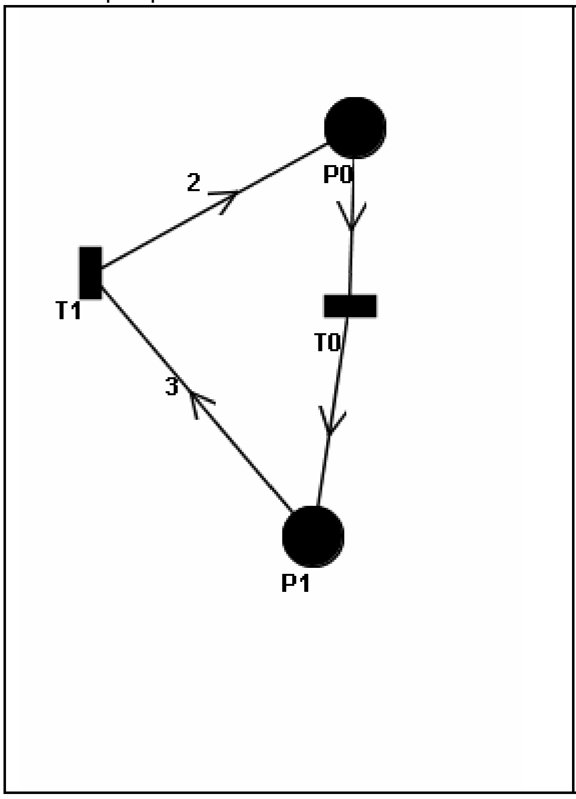


Figure 1.2 Arc Weighting

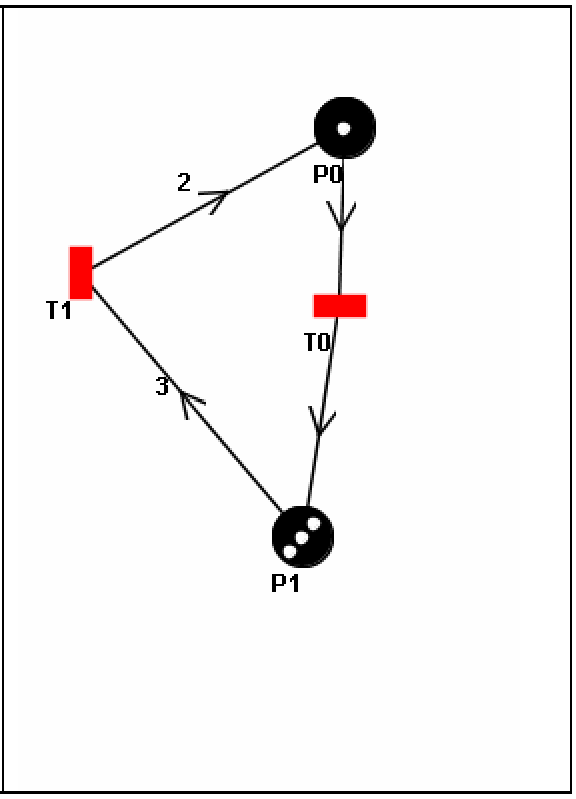


Figure 1.3 Enabled Transitions

An enabled transition may be fired. If a transition is fired then it destroys tokens on its input places. The arc weighting on the arc connecting each input place with the transition gives the number of tokens destroyed on each input place. The firing of a transition also results in tokens being created on the output places. The arc weightings on the arcs connecting a transition with its output places give the number of tokens created on each output place as a result of firing a transition.

Formal Definition of a Place-Transition Net:

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

- P = {p_1,…,p_n} is a finite and non-empty set of places,
- T = {t_1,…,t_n} is a finite and non-empty set of transitions,
- P n T = Ø,
- I^-,I^+ : P × T ? N_0 are the backward and forward incidence functions,
- M_0: P ? N_0 is the initial marking of the Petri net.[2]

The forward and backward incidence functions (I^+ and I^- respectively) describe the connections between places and transitions. If $I^-(p,t) > 0$, then an arc leads from place p to transition t so that p is an input place for transition t. The value of this backward incidence function is equivalent to the arc weighting, and gives the number of tokens required on p to enable a transition. (This is equivalent to the number of tokens lost on p if the transition t is fired). Similarly the forward incidence function $I^+(p,t)$ specifies the number of tokens created on a place p in the case of firing transition t, where p is an output place of transition t.

Other types of Petri nets include Time-Augmented Petri nets where enabled transitions have a probability associated with them, and Stochastic Petri nets where the firing probability follows an exponential distribution.

Motivation

There are many tools available for modelling Petri nets, all of which provide a graphical user interface through which nets can be created and edited, and a variety of tools to perform analysis on those nets. Petri-Net World has a comprehensive list of tools available,[3] however, research performed on a number of these editors revealed that many were difficult to use and unintuitive (Table 1.0). It was clear that a key design consideration of the project would be to provide an easy-to-use application in order to make creating a Petri net simple, fast and efficient.

Existing Petri net tool functionality is however, limited to that which their programmers provide at the time of writing. The main focus and purpose for the project, has been to extend the functionality offered by existing Petri net applications by creating a tool which supports the arbitrary run-time loading and unloading of analysis modules. By using a Module Interface in the application, users have the ability to write their own analysis modules or interface (plug-in) pre-existing analysis modules to PIPE simply by implementing this interface. This means that users are able to perform all types of analysis using one application. By adhering to the latest XML Petri net standards (PNML), users will also be able to run the modules on any pre-existing Petri nets conforming to the same standard. This extensibility will ensure that the application does not become obsolete.

Please note that all diagrams of Petri Nets contained in this report are taken from PIPE.

2.0 Specification

This section details the specification that was laid out at the start of the project and in Report 1. The specification was divided into two parts: the basic functionality of the application and the proposed extended functionality. The overall design and structure of the product seemed to fall logically into three categories; these categories were used to discuss the specification of the application.

Overall Specification

To construct a Petri net application that supports:

- the creation of Petri nets conforming to the latest XML Petri net standard (PNML)
- pluggable analysis modules that implement a standard interface allowing extensibility

2.1 Graphical User Interface

2.11 The Editor

This should allow the user to create, save and load (pre-existing Petri nets) graphical Petri nets using the Petri Net Markup Language (PNML). Within the editor, the user will be able to easily create and edit Petri nets. The editor should be easy to use and as intuitive as possible for the user.

Basic Operations:

- Open/load existing Petri nets from the XML/PNML files
- Save a Petri net to an XML/PNML file
- Create and edit a Petri net easily and intuitively using the standard representation. The above should be implemented via menu items, toolbars, mouse actions and keyboard shortcuts.

Extended Functionality:

- Loading of all types of PNML files, including hierarchical nets
- A multiple document interface
- Printing of Petri nets

2.12 Animation

The application should also contain an animator such that the user can manually fire enabled transitions within the Petri net.

Basic Functionality:

- Enable transitions should be highlighted, so that the user can see which transitions they are permitted to fire.
- The Petri net can easily be returned to its initial state, either at the end of animation or at any point during the animation as required by the user.

Extended Functionality:

To provide as comprehensive an animator as possible, it would be desirable for the user to be able to undo the previous operation/step backward through the transition sequence.

2.2 Pluggable Analysis Modules

The purpose of the modules is to perform analysis on the Petri nets created or loaded into the application.

Minimum Specification:

- An Invariant Analysis module that will analyse a Petri net, determining the P-Invariant and T-Invariant vectors, providing information about boundedness and liveness.

- A simulation module that will run without any input from the user and will operate in conjunction with the analysis modules so that properties of the simulation can be analysed.

Extended Specification:

- Comparison Module - this will be used to compare different Petri nets based on their PNML files. The module will not only confirm whether two Petri nets are the same but will also work out the differences between two Petri nets.

- Classification Module - this module will classify a given Petri net into one of the following types: State Machine, Marked Graph, FC-Nets, EFC-Nets, SPL- Nets, ESPL Nets.
- State Space Analysis Module - to build the reachability tree and to investigate properties such as boundedness, deadlock, safeness and liveness.

- Incorporation of an existing C++ Markov Chain Analyser

2.3 Module and Document Management

The application should be able to load modules as plug-ins to perform analysis on the specified Petri nets. The application should also be able to save and load graphical Petri nets.

Basic Functionality:

- A component for the modules with the module options displayed, allowing modules to be loaded at run time.

Extended Functionality:

- A small module window within the application to display modules loaded and options available for each module. A tree view would be a suitable way of implementing this. The user will be able to select an analysis module from the local file system using a file chooser.

2.4 Other Extensions

To truly reflect the project life cycle, it would be advantageous to perform testing on our application ahead of completion. Given the time constraints of the project, it will be difficult to arrange for a full suite of tests to be performed and to re-work the application to incorporate the findings ahead of the deadline. For an extension of the project, the aim will be to perform limited user interface tests as soon as there is a working interface so that the results can be incorporated.

3.0 Technical Methodology

Overview

The project fell logically into three areas:

- The graphical user interface
- A layer managing the interactions between the GUI and the modules.
- The analysis modules

It transpired during the design stage of the project that the layer sitting between the GUI and the analysis modules would be vital to the successful functioning of the application. This layer would need to act as a sort of adaptor or translator, providing the modules with a standard data format on which to act. This ‘data’ layer would also have to take into account Petri Net Markup Language (PNML) standards so that the application would be able to load any Petri net conforming to this standard. The chosen data representation therefore had to be comprehensive and flexible. The implementation strategy for the GUI was made easier by the extensive features provided by the Java Swing api.

It became clear that the application structure in terms of the relationship between the GUI and the data layer would be best implemented according to a Model-View-Controller architecture.

Model View Controller Architecture

The MVC pattern is simple yet effective, and allows the application to be divided into three distinct areas:

> **Model:** The core of the application, maintaining the state and the data that the application represents. When significant changes occur in the model, it updates all of its views. In PIPE, the data layer is the Model.
>
> **Controller:** The user interface presented to the user to manipulate the application. In PIPE - a Swing container, which controls the individual, Petri nets.
>
> **View:** The user interface, which displays information about the model to the user. Any object that needs information about the model needs to be registered with the model. The view is the graphical representation of a Petri net.

This design pattern has many advantages, in particular clarity of design, modularity, extensibility, and more importantly, its flexibility. This design, its implementation and the advantages it gave us are discussed in greater detail below.

3.1 Data Layer

PNMLData Package

The PNMLData Package contains all the classes used to represent a Petri net. The PNMLData Package has four main roles:

1. Storing all information about a Petri net and calculation of Incidence and markup matrices.
2. Storing the Petri net model in a suitable form for graphical representation.
3. Firing of transitions.
4. Saving or loading a Petri net from a file.

Petri Net Representation

The Petri net model is made up of Places, Arcs, Transitions, Markup Matrices and Incidence Matrices. Places, Arcs and Transitions each have several attributes that characterise their properties, such as id, name, location, etc. These attributes are stored in variables internal to each object. The use of inheritance ensures that common variables and their accessor methods are only coded once.

Each Petri net is encapsulated by an instance of the PNMLData class. The PNMLData class contains all the Petri net Objects stored in ArrayLists enabling the easy addition of new objects. PNMLData has not only methods to access all its internal objects and to return its internal lists in the form of Object Arrays, but also methods to calculate the current markup, initial markup, forwards incidence matrix, backwards incidence matrix, combined incidence matrix and enabled transitions. To increase efficiency the different matrices are only calculated when a get method is called, therefore eliminating the need for continuous updating.

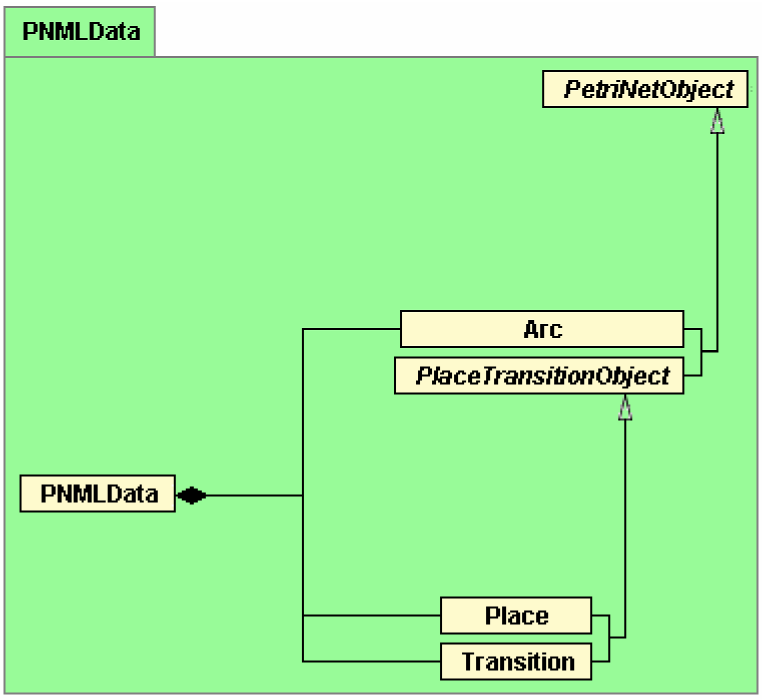


Figure 1.5 Class Hierarchy of PNMLData Package

Animation

The PNMLData class performs the firing of transitions; it has methods that allow either the selection of a specific transition or the firing of a random transition. Random firing is achieved by using Java's random number generator to randomly pick a transition from the set of enabled transitions. Calculation of the result of firing any transition is performed by adding the correct firing vector to the current marking matrix. The firing vector is calculated from the incidence matrix.

Model/View/Controller Design Pattern

Changes to the PNMLData, for example caused by adding a new object or firing a transition is immediately displayed in the GUI through use of Model/View/Controller Design Pattern. PNMLData is used as the model, storing the data for the graphical representation. When PNMLData is changed it notifies the GuiView class, registered as an Observer, which in turn updates its visual representation of the Petri net.

Saving and Loading Petri nets - Builder Design Pattern

The saving of Petri net files is achieved by a Builder Design Pattern approach, whereas loading is achieved by a backwards Builder Design Pattern approach. An Extensible Stylesheet Language Transformation (XSLT) is used to transform the Petri Net Markup Language file into a simple standard XML file. Supporting different input file formats, such as the TREX format is very easy due to the Builder Design Pattern approach. To load a new file format all that is required is a new XSLT. This approach means that a new format can be used to load a Petri net without requiring any change to the Java code. When an input file is read by PNMLData it checks for an XSLT declaration at the start of the file, if none is found it is assumes the file is a legal PNML file. The next stage is to extract the information from the standard XML file; this can be achieved by three approaches:

- JDOM is similar to JAXP but has less functionality, for example it does not support XPath based technologies such as XSLT, XPointer, XInclude and XQuery.
- JAXB has the advantage that the required classes are automatically created from the XML file. JAXB was not adopted because it is still only a beta version and not yet fully supported or documented. This is a distinct disadvantage (as warned by Sun Microsystems) because when a new version of Java is bought out JAXB is planned for an extensive revamp and therefore the saving and loading code would be less compatible with new versions of Java.
- JAXP has all the functionality of JDOM and more. JAXP is built into all versions of Java 1.4.0 as standard and is also very well documented so is easy to use. It was decided to use JAXP for all reading, writing and XSLT transformation of XML.

Dynamic Loading of Modules

One of the advantages of this application lies in its non-specialist nature with respect to the analysis functions available. It requires little knowledge of Petri net analysis, and as all modules are integrated and used in exactly the same way creating and using a new analysis module requires no change in the application code, and can be done whilst the application is running.

The pluggable nature of the Modules is enabled by the Reflection API, which provides information about classes (methods, constructors etc), and methods to instantiate objects and invoke their methods, at runtime, without prior knowledge of them.

Writing Custom Modules

The overall design of the application imposes few restrictions on developers writing custom modules. This means that the changes involved in adapting existing analysis code to use within the application should be minimal, and writing new modules should be straightforward.

All Modules must implement the `Module`interface, which contains only two methods:

```
public void run(PNMLData PetriNet) { ... }
public String getName() { ... }
```

In addition to this, modules must also have a constructor with no parameters.

Each public method that the implementing module has which takes one parameter of type PNMLData will be displayed as a clickable method in the ModuleTree in the UI. It is the intention that these methods perform an analysis function of some description. There is no limit on the number of such methods, but there must be at least one, enforced by the `run` method in the interface.

Developers can extract data about the net being analysed from the PNMLData object and are free to display the results in any format, e.g. a frame displaying numerical data in a table,

The `getName` method supplies the module's display name. There is no equivalent for the methods, so method names should be meaningful to the user.

Plugging Custom Modules into the Application

In order to use a custom module in the application, the module must be on the system CLASSPATH, in a directory structure reflecting its package name. Any modules within the lib/ directory of the application will be displayed in the ModuleTree when the application starts, otherwise the user can add the module into the ModuleTree using the Find Module option.

3.2 GUI

Overview

The research conducted into other Petri net tools (Table 1.0) demonstrated that many were unintuitive to use and basic tasks were hard to complete. This was significant in the design of the GUI, the intention being to provide an effective interface. This was also the motivation behind submitting PIPE to user testing ahead of completion so that the results from usability tests could be assimilated.

Design Considerations

The Java Swing API was chosen as it provides full GUI functionality and mimics the platform it runs on. As PIPE is a cross platform application this was deemed useful for providing a native look and feel.

A GUI shell with a simple, standard interface was designed, with the main focus being to create Petri nets and run analysis modules. For this reason, the drawing pane was designed to be as large as possible so that the user can create larger nets with ease, and there is a "tree" menu to the left allowing users to quickly select, remove or add analysis modules as required.

Users are able to perform tasks using a menu bar, toolbar and mouse actions. The toolbar buttons were designed so that the "selected" image indicates the current mode, and alongside the buttons sits a status bar, which details the mode and the available options for the current mode.

Whilst creating a net, or after loading a pre-existing net, the user can apply a variety of Petri net specific operations via mouse clicks on the individual components. A right mouse click on a component generates a context menu specific to that component. For example, the context menu for a place allows the user to add and remove tokens, name the place, or delete the element.

Implementation

Java Swing

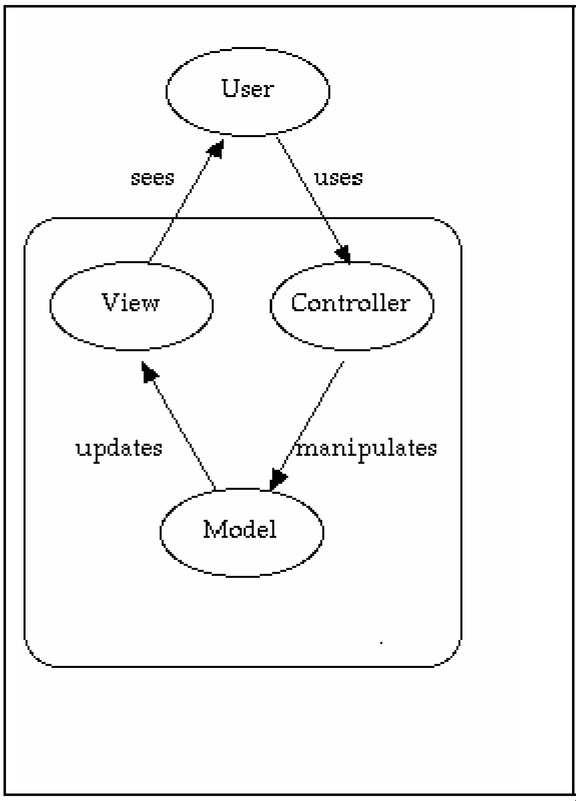
All the components of the GUI extend Java Swing components, enabling them to take advantage of the base class features as well as allowing customisation of the components making them suitable for the needs of PIPE. The Petri net elements (place, transitions, arcs) and their Swing representation required the most thought as it was important to choose a component that reflected the many operations that would be needed; for example, weighting for arcs, adding and removing tokens for places, naming for all Petri net elements. For this, it was decided to extend the JLabel component.

GUI – Data Layer Interaction (Model View Controller Architecture)

The interaction between the GUI and the data layer proved to be one of the most important areas, deserving careful consideration. The graphical representation of the Petri net within the GUI requires a very specific implementation. However, this graphical representation of the Petri net has to be stored in such a manner that the modules can use the data to perform their analysis on the nets.

The data layer and the GUI are both updated frequently and must remain synchronised. If the user moves a Petri net element in the editor pane, the X and Y coordinates need to be updated in the data layer. Also, if the user loads a pre-existing net into the program (the data layer), this new net needs to be drawn in the editor pane.

As a result, it was decided to implement the Model/View/Controller Design Pattern as this seemed ideal for the needs of PIPE. The MVC architecture separates the structure into the **model**, representing the data layer, the **view** being the visual representation of the data, and the **controller**, the interface allowing the user to manipulate the both the data and the visual representation.



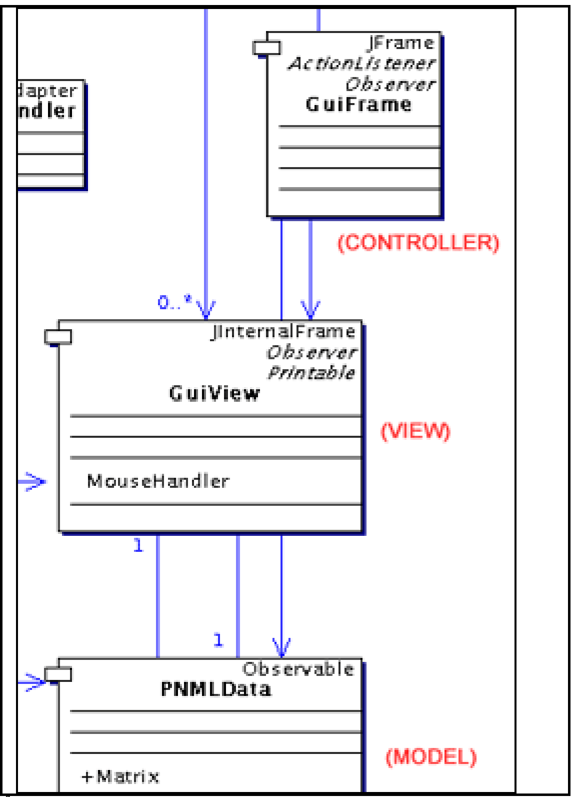


Figure 1.6 Model –View-Controller Architecture[4] Figure 1.7 PIPE implementation of MVC

PNMLData is used as the model, storing the data, whilst the GuiView creates the visual representation of the component from the data in the model and handles user interactions. PNMLData extends the Java Observable class and we register GuiView as an Observer of PNMLData so that PNMLData can automatically notify the view of any changes. By using this architecture, loading other Petri nets becomes easier as GuiView will be updated automatically to reflect the changes as soon as PNMLData has been changed. MVC also allows different models and views to be swapped in and out easily which was particularly useful when it was decided to implement tabbed panes.

Tabbed Panes and Multiple Petri-Nets

PIPE was initially developed with the ability to draw, view, and analyse only one Petri-Net at a time. The ability to manipulate and compare multiple nets is a useful extension to PIPE's functionality. One of the strengths of the MVC architecture is that this extension did not need a dramatic re-engineering of the application. Theoretically, all that was needed was pairs of views and models observing each other. The main technical challenge was to change which pair the whole application was observing in order that the drawing and analysis was done on the correct net. The bulk of the work that the implementation required involved handling multiple views and models and keeping the correct link between them. Arrays of views and models were created with the indices in both arrays used to match the pairs up. The application could then talk to one model-view combination at a time by using that common index.

A multiple document interface (MDI) was explored using a JDesktopPane and JInternalFrames that would be used for the presentation of nets. It was felt that this approach was slightly old fashioned and made the application look unsightly. It was decided to use a JTabbedPane with a JInternalFrame on each tab. This gives the application a modern feel and simplified the implementation of multiple nets, as each tab had an index that could be linked with the relevant model/view pair's index.

Saving and Loading

PIPE required the ability to save and load nets in order to be usable in a productive manner. Java provides a number of classes that enable the implementation of Saving and Loading in a way that would be easily recognisable to the user as common to any commercial application. The JFileChooser class in the javax.swing package provided an easy-to-use mechanism for creating file dialogs for opening and saving files.

For saving, once a filename and directory is chosen, PIPE creates a XML file from the data held in the model part of our MVC architecture. It does this by iterating through each object, that is both shown in the view and held in the model, and adding the correct nodes and ids to the XML DOM. The transforming language XSLT is then used to ensure that the file conforms to the current standard PNML definition. Once the document is completed, a Java File object is created and the document is written to it using an Output Stream.

Loading follows the same processes but in reverse. Once a file is loaded into a new instance of our model (PNMLData) and linked to a new view, the view automatically represents the data due to the aforementioned MVC architecture.

Animation

From a GUI point of view, the animation is relatively straightforward, as a result of the implementation of the MVC architecture. The main steps are detailed below:

1. When the user chooses to enter animation mode, the model is stored so that the user can return to that state (just prior to animation) when they exit animation mode.
2. The user cannot change the Petri net when in animation mode except to fire transitions.
3. The animator queries the model for all enabled transitions. A simple boolean – isEnabled, within the Transition class is set to true if this is the case. The view is then updated to reflect this change, representing the enabled transitions as red. This step is repeated after every firing.
4. The user manually fires an enabled transition by mouse clicking on it. The model (PNMLData) is responsible for the incrementing and decrementing of tokens from the input and output places during the actual firing of a transition and the results are updated in the view.
5. The model is then queried again to update the enabled transitions and the view updates itself to show the new enabled transitions.
6. When the user chooses to exit Animation mode, the stored state of the model (before animation) is restored and the view once again updates itself.

Additional functionality was provided so that the user can step backwards and forwards through a list of previously fired transitions. An arraylist is maintained, storing the history of the animation so that this functionality can be provided easily. To make this more user friendly, it was decided to represent this list as a pane in the GUI, loaded when the user enters animation mode. Screenshot 1.0 f animation mode in the appendices demonstrates this.

It was also decided to add "random fire functionality", an additional feature not laid out in the specification. This allows the user to specify how many times they would like the net to be fired. This then passes the information to the model, which performs the random firing. The results of the firing are then displayed in the view.

3.3 Analysis Modules

3.31 Invariant Analysis Module

The objective of this module was to analyse a Petri net and determine the P-Invariant and T-Invariant vectors. These vectors (called net-invariants) are useful since the complexity of their calculation depends only on the number of places and transitions in the net and not on the size of the reachability set. This is important, because to calculate the reachability set can be quite inefficient in several cases in real life situations.

The module implements the Module interface, and as such it has a run() method which takes as argument an object of type PNMLData. Then it invokes getIncidenceMatrix() on this object to obtain the incidence matrix of the net, which is the difference of the forward and backward incidence matrix. This is represented in Java as a double array of integers.

The module output includes the net invariants of the Petri net under analysis (if there are any), the marking equations and, based on the computed net-invariants, conclusions about the properties of the net (boundedness and liveness).

The module package is called invariant_analysis and it consists of three classes: InvariantAnalysis, IntMatrix and IAResult. The first is the main class and the other two are supporting classes. The purpose of each class is as follows:

- InvariantAnalysis: performs the analysis based on the algorithm presented below.
- IntMatrix: performs matrix operations, and it is derived from the JAMA matrix manipulation package[5]. At first, all unnecessary functions were removed from JAMA, then all variables were converted from double to int (since these are the kind of numbers in the incidence matrix) in order to reduce memory consumption and increase execution speed, thus making the module more efficient, and lastly extra operations such as various checks on rows-columns, like cardinality conditions and finding zero or non-zero elements, comparisons and linear combinations of columns were added. These operations are specifically relevant to the implemented algorithm.
- IAResult: extends JFrame and displays the result in a window, which also gives the user the option to print or save the results.

Background Theory

The forward and backward incidence matrices are defined in Bause and Kritzinger[6] (def. 4.8) as follows:

Given a Petri net PN = $(P, T, I^-, I^+, M0)$ the backward incidence matrix is defined as

$C^- = (c_{ij}^-) \in N_0^{n\times m}$ where

$c_{ij}^- := I^-(p_i,t_j), \forall p_i \in P, t_j \in T$

the forward incidence matrix is defined as

$C^+ = (c_{ij}^+) \in N_0^{n\times m}$ where

$c_{ij}^+ := I^+(p_i,t_j), \forall p_i \in P, t_j \in T$

and the incidence matrix is defined as

$C := C^+ - C^-$

This means that each element of the incidence matrix is the arc weight of a place-transition 2-tupple (in that order), the element row number is the same as the index of the place in the P vector, the column number is the same as the index of the transition in the T vector. For the forward incidence matrix the place is an output place of the transition, and for the backward incidence matrix the place is an input place of the transition. If there is no connection the entry in the matrix is 0.
Enabling and firing of transitions can be expressed in terms of these matrices. For example a transition t_i is enabled at a marking M iff $M \geq C^- e_i$ where e_i is the column vector with 1 at the i^{th} row and 0 elsewhere.

If an enabled transition fires at a marking M the resulting marking M' is $M' = M + C\, e_i$
$C\, e_i$ is the ith column of the incidence matrix C, specifying the influence of transition t_i on its output places. Therefore, markings can be calculated as addition of vectors.

Given a firing sequence $\sigma = t_{k1}, \dots, t_{kj}$, marking M_j can be calculated as $M_j = M_0 + C\sum_{i=1}^{j} e_{k_i}$

The vector given by the summation above is called a firing vector and its dimension is $|T|$.

A vector $v \in Z^n$, $v \neq 0$, is a P-invariant iff $v^T C = 0$.

A PN is covered by positive P-Invariants iff for all p there exists a P-invariant vector ≥ 0, with all its elements > 0 ([BAU95] def. 4.10).

If a PN is covered by P-Invariants then it is bounded[7] if it is not covered by P-Invariants then we cannot conclude about the boundedness, further analysis is required.

Here, covered means all the places must appear in at least one invariant vector.

Similar definitions exist for the T-invariants, which are just firing vectors such that when applied to a starting marking we can arrive to the same marking. We have:

A vector $w \in Z^m$, $w \neq 0$, is a T-invariant iff $Cw = 0$.

A PN is covered by positive T-Invariants iff for all t there exists a T-invariant vector ≥ 0, with all its elements > 0[8]

Covering by T-Invariants is a necessary condition for a PT net to be bounded and live.

So if a PT net is covered by T-Invariants, it is <u>possible</u> to be bounded and live.

To calculate the P- and T-Invariants, the algorithm used was the one from D'Anna and Trigila[9]. There are other algorithms for finding net invariants too, but this one was chosen because it performs reasonably well, and a good description was available in the paper mentioned above, which meant faster implementation. The pseudocode for this algorithm is as follows:

```
/* initialisation of the extended matrix */
      C:= Incidence matrix ⊂ $Z^{mxn}$;
      B:= Identity matrix ⊂ $Z^{nxn}$;
/* The extended matrix is the matrix obtained by adjoining B to the
bottom of C giving a matrix of (m+n) x n dimension */
[phase 1]
while (there exists a non-zero element in C) do
      [1.1] if (there exists a row h in C such that the sets
      $P^+ = \{j \mid c_{hj} > 0\}$, $P^- = \{j \mid c_{hj} < 0\}$
      satisfy $P^+ = \emptyset \vee P^- = \emptyset$)
      then
      [1.1.a] delete from the extended matrix all the columns of
      index $k \subset P^+ \cup P^-$
      else
            [1.1.b] if (there exists a row h in C such that $| P^+ | =
1 \vee | P^- | = 1$)
            then
      [1.1.b.1] let k be the unique index of column belonging to $P^+$
(resp. to $P^-$);
      for $j \subset P^-$ (resp. $j \subset P^+$ ) do
                  substitute to the column of index j the linear
      combination of the columns indexed by k and j with
      coefficients $|c_{hj}|$ and $|c_{hk}|$ respectively
      endfor;
            delete from the extended matrix the column of index k
else
      [1.1.b.1] let h be the index of a non-zero row of C;
            let k be the index of a column such that $c_{hk} \neq 0$;
      for j such $j \neq k$, $c_{hj} \neq 0$ do
                  substitute to the column of index j the linear
      combination of the columns of indices k and j
      with coefficients $\alpha$ and $\beta$ ?defined as follows:
      if $sign(c_{hj}) \neq sign(c_{hk})$ then $\alpha$ ?= $|c_{hj}|$, $\beta$ ?= $|c_{hk}|$
      else $\alpha$ ?= $-|c_{hj}|$, $\beta$ ?= $|c_{hk}|$ endif;
      endfor;
            delete from the extended matrix the column of index k
      endif;
      endif;
endwhile;

[phase 2]
while (the matrix B contains a row of index h with negative elements)
do
let $P^+ = \{j \mid c_{hj} > 0\}$, $P^- = \{k \mid c_{hk} < 0\}$:
if ($P^+ \neq \emptyset$) then
for $(j,k) \subset P^+ \times$ ?$P^-$ do
operate a linear combination on the columns of indices
j and k in order to get a new column having the
h-th element equal to zero,
divide this column by the GCD of its elements,
append this column to the matrix B
endfor;
endif;
delete from B all the columns of index $k \subset P^-$
endwhile;
delete from B all the columns having non-minimal support.
The invariants appear as the columns of the matrix B, the bottom n
rows of the extended matrix.
```

Example – test case for the module:

The following Petri net was used as a test case. It represents the readers/writers problem where the number of readers s = 3, the number of writers t = 2, the initial value of the semaphore n = 3 (n ≥ s to allow all the readers to operate in parallel, in this case n = s, so the readers can operate in parallel).

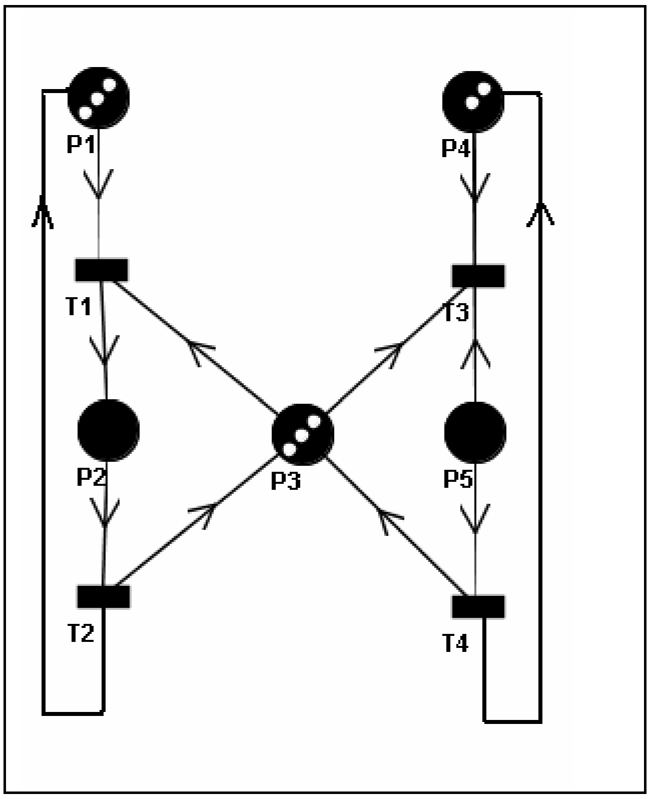


Figure 1.9 Readers/writers Petri net with 3 readers, 3 writers and maximum 3 readers at the same time

The corresponding matrices are

$$C^- = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 3 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad C^+ = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} -1 & +1 & 0 & 0 \\ +1 & -1 & 0 & 0 \\ -1 & +1 & -3 & +3 \\ 0 & 0 & -1 & +1 \\ 0 & 0 & +1 & -1 \end{bmatrix}$$

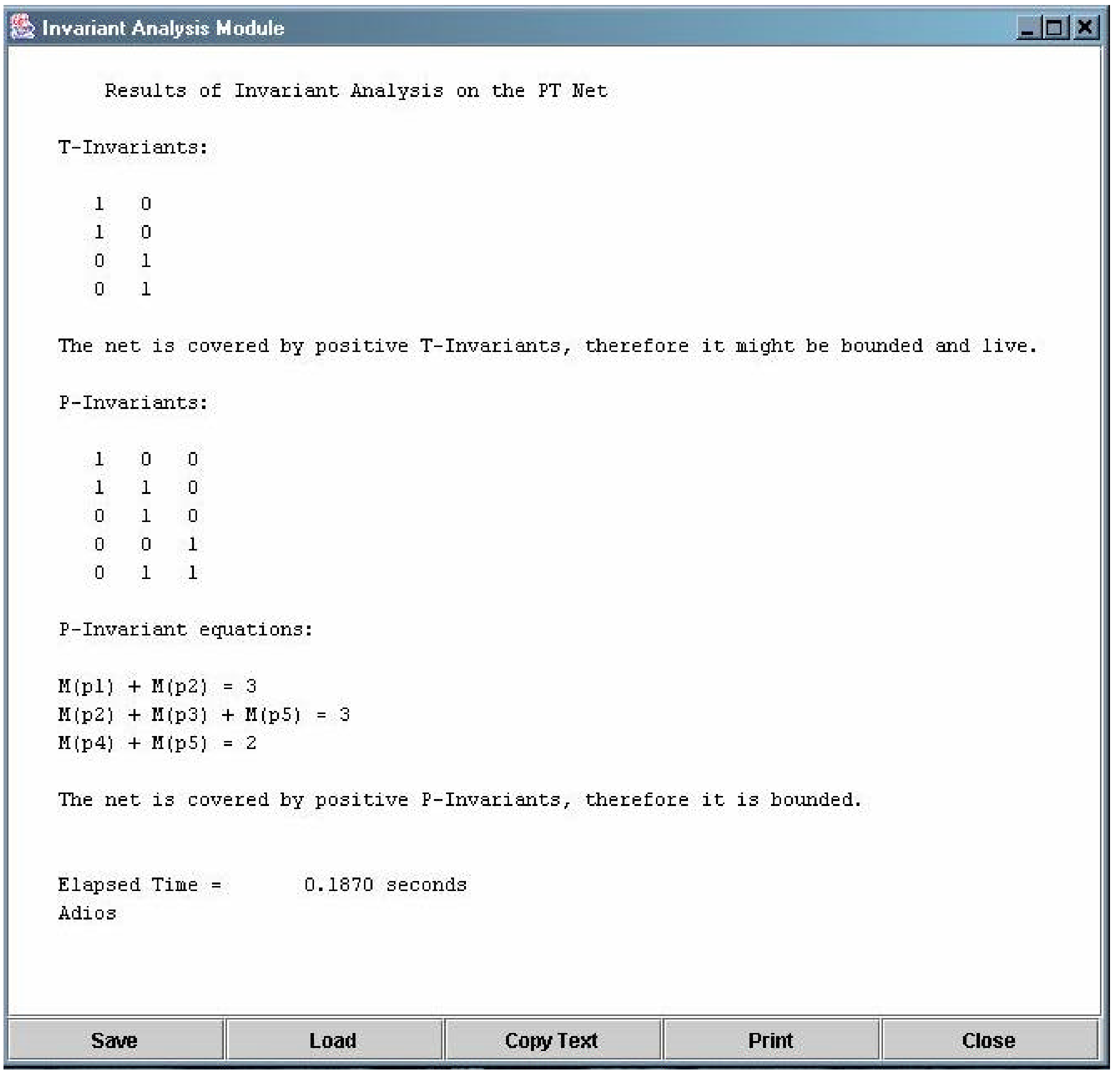
The P-Invariants are $\overrightarrow{v_1} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \overrightarrow{v_2} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \overrightarrow{v_3} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 3 \end{bmatrix}$

The T-Invariants are $\vec{w}_1 = \begin{bmatrix}1\\1\\0\\0\end{bmatrix}, \vec{w}_2 = \begin{bmatrix}0\\0\\3\\3\end{bmatrix}$

And the marking (P-invariant) equations

M(p1) + M(p2) = 3
M(p2) + M(p3) + M(p5) = 3
M(p4) + M(p5) = 2

The results of the module can be seen in the next figure:



```
     Results of Invariant Analysis on the PT Net

T-Invariants:

   1   0
   1   0
   0   1
   0   1

The net is covered by positive T-Invariants, therefore it might be bounded and live.

P-Invariants:

   1   0   0
   1   1   0
   0   1   0
   0   0   1
   0   1   1

P-Invariant equations:

M(p1) + M(p2) = 3
M(p2) + M(p3) + M(p5) = 3
M(p4) + M(p5) = 2

The net is covered by positive P-Invariants, therefore it is bounded.


Elapsed Time =       0.1870 seconds
Adios
```

Figure 2.0: Demonstration of the Invariant Analysis module output

As it can be seen, this agrees with the test case. Also, the performance of the module is satisfactory, well below 1 second. Even with more complex nets with around 30 components the performance was below 1 second in a P4 2.4 GHz. This is important since the second part of the algorithm exhibits exponential complexity.

3.32 Simulation Module

A simulation module was designed and produced to investigate performance characteristics of a Petri net. The motivation for this is that often a full analytical approach to a problem demands a huge amount of resources, while simulations can provide an alternative method to overcome performance issues. The simulation should be able to investigate the performance of Petri nets of a larger scale than those tackled by an analytical approach (such as evaluating the complete state space by constructing a tree of possible markings).

This simulation module studies the performance of a selected Petri net by investigating the average number of tokens per place. It is a Monte Carlo simulation that uses a random number generator (seeded on the system clock) to pick a new marking state of the Petri net. (The functions to fire a random transition and to output the current state of the net exist within the PNML data object). At each new state, the number of tokens per place is recorded, so that the average number of tokens per place can be calculated over a given number of transitions. This cycle can be repeated multiple times for the same initial marking and transition fires to find an average number of tokens on each place after all the cycles. The standard error in this value will reduce in proportion to the square root of the number of cycles which are performed. A 95% confidence interval for the average number of tokens per place is calculated using the intermediate averages from the results of each cycle. The results from this module are shown in its own pane in tabular format, with the average number of tokens and the 95% confidence interval displayed for each place in the net. The random firing of transitions can also be weighted so that transitions with different probabilities can also be taken into account.

3.33 State Space Analysis Module

The State Space Analysis Module illustrates the concept of extensibility, and it analyses the selected Petri net by building a tree of all the reachable markings. This reachability tree is constructed by starting with the initial marking of the place-transition net (see definition 1.1) and adding directly reachable markings as leaves. The reachability tree can then be analysed to determine properties of the initial Petri net such as boundedness, deadlock, and safeness.

Theoretical background to the analysis

The incidence functions used to describe a place-transition net in definition 1.1 can also be described using matrices. The backward incidence matrix, C^- is defined by:

$$C^- = \left(c_{ij}^-\right) \in N_0^{nxm}$$

$$c_{ij}^- := I^-(p_i, t_j), \forall p_i \in P, t_i \in T$$

The forward incidence matrix, C^+ is defined by:

$$C^+ = \left(c_{ij}^+\right) \in N_0^{nxm}$$

$$c_{ij}^+ := I^+(p_i, t_j), \forall p_i \in P, t_i \in T$$

The incidence matrix of PN is defined as C: = C^+ - C^-. These matrices and an initial marking vector can therefore provide a complete description of the initial net. When a transition is fired the marking of a Petri net changes and the formal description of the firing of a transition is given in definition 3.1.

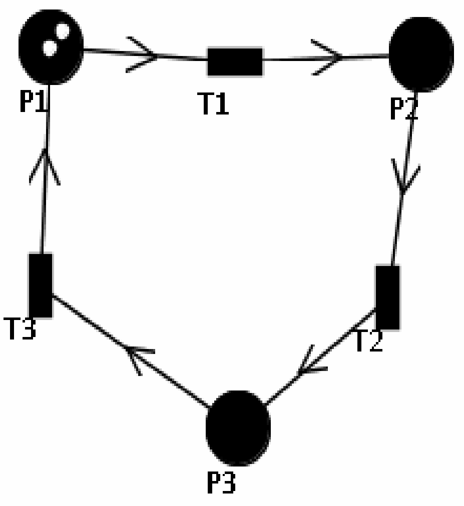
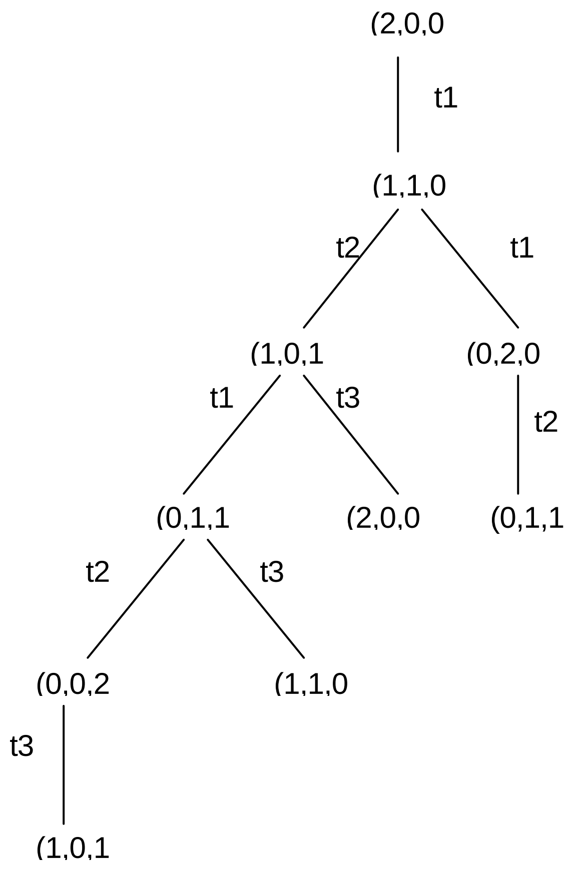
Definition 3.1 *If PN = (P, T, I^-, I^+, M_0) is a place – transition net*

- *A marking of a place transition net is a function M: P ? N_0, where M(p) is the number of tokens on the place p*
- *A set 3" $\subseteq P$ is marked at marking M, iff $\exists p \in$ 3: M(p) > 0; otherwise 3"is unmarked or empty at M*
- *A transition t Î T is enabled at M, denoted by M[t>, iff M(p) = I^-(p,t) "p Î P*
- *A transition t Î T, enabled at marking M, may fire giving a new marking M' where*
  M'(p)=M(p)-I^-(p,t) + I^+(p,t) "p Î P

and this is denoted by M[t>M’. In this case M’ is directly reachable from M and can be written M ? M’.

- *A firing sequence for a PN is a finite sequence of transitions s = $t_1 \ldots t_n$ n=0 such that there are markings $M_1, \ldots, M_{n+1}$ satisfying $M_i[t_i > M_{i+1}$ "I = 1,…n. A shorthand notation for this case is M_1[s> and M_1[s > M_{n+1} respectively. The empty firing sequence is denoted by e such that M[e > M always holds.*

The reachability set contains all markings that are reachable from the initial marking M_0, and analysis of this is useful, since all properties of the net are defined from it. This module therefore analyses the described net by generating a tree of possible markings. The diagrams in figure 3.1 and 3.2 show a Petri net with its associated reachability tree.

	 (2,0,0 t1 (1,1,0 t2 t1 (1,0,1 (0,2,0 t1 t3 t2 (0,1,1 (2,0,0 (0,1,1 t2 t3 (0,0,2 (1,1,0 t3 (1,0,1
Figure 3.1: A Place –transition net	Figure 3.2: Reachability tree for figure 3.1

Design and Implementation

The State Space Analysis module takes a vector of integers and 2 matrices of integers as inputs. These are the Initial Marking vector, and the Forward (CPlus) and Backward (CMinus) incidence matrices respectively, and they are obtained from the PNMLData object (as for the invariance analysis and other modules). The reachability tree is generated by starting with the initial marking of the place-transition net, and adding directly reachable markings (determined from the incidence matrices), as leaves of the initial root node. These new markings are then used and the directly reachable markings from them are also determined. These markings then become the new leaves of the already generated part of the reachability tree. The reachability tree is therefore built recursively, by calling a recursive expansion function on the root node which corresponds to the initial marking of the net. This function continues to call itself until the tree generation is complete.

There are two important conditions which lead to the termination of a particular branch of the tree and these are described in the following section.

Repeated Marking

If a marking is reached that has been previously explored anywhere in the tree, then no further nodes are added on that branch. This leads to the consequence that the reachability tree is not a unique representation of the state space as the representation is dependent on the order in which the possible firings are explored. Other existing Petri net analyser programs which investigate state space analysis tend to build a larger version of the reachability tree, where construction of a particular branch is stopped only if a node is a repeat of a direct ancestor on that branch. This makes the analysis of the tree significantly easier, but since a large reachability tree can be generated from a simple Petri net, this State Space analysis module was designed to build the smaller version of the tree in order to optimise its performance.

No available transitions

If there are no transitions which can possibly be fired from the state contained in the current node then a state of deadlock has been reached and the branch has to terminate.

Unbounded Petri nets

If the marking of a Petri net is greater than that of one of its ancestors on the branch of the reachability tree then places in that marking will be unbounded. This will occur if one or more places in the current marking have more tokens on them than the previous marking, while the numbers of tokens on the other places are the same. In this case, the places with more tokens than in the previous marking are said to be unbounded places and are given the symbol omega in the representation of the tree. A reachability tree containing omegas is called a coverability tree. If the omegas were not added this would lead to a reachability tree with an infinite number of states, since the marking on an unbounded place can continually increase.

Tests on the reachability tree

The state space analysis module performs tests on the reachability tree to investigate boundedness, deadlock and safeness. The test for boundedness checks for any omegas in the coverability tree, which are introduced during its construction to represent unbounded places. The results pane shows a tick or cross to indicate whether a selected net is bound or unbound. The presence of at least one omega means that the net is unbound. Deadlock occurs when it is no longer possible to fire any transitions, and this is also reported in the same way. Tests are also carried out for safeness, which investigate whether it is possible for more than one token to be on any place, so that if a place holds more than one token anywhere in the tree, then the net and the system represented by the net is unsafe.

The algorithm which was designed to construct and investigate the coverability tree is described in the following section.

Algorithm 3.1

```
Inputs: Initial markup vector.
        Cplus and Cminus, the incidence matrices.

Create root of the tree and populate it with the initial markup
vector, and initialise static variables as follows:
  BOOL Found_An_Omega = false;
  BOOL Found_A_Place_With_More_Than_One_Token = false;
  BOOL Found_A_Place_With_No_Enabled_Transitions = false;

Call RecursiveExpansion on the root.
Function RecursiveExpansion:
BOOL A_Transition_Is_Enabled = false;
For each transition {
  If the transition is enabled  {
    Set A_Transition_Is_Enabled = true;
    Do it to produce a new markup vector.
    Check each integer in the variable. If any of them are >1 {
      Set Found_A_Place_With_More_Than_One_Token = true;}

Create a new node using the new markup vector and attach it to the
current node as a child.
/* Need to:  a) Check if any omegas need to be inserted in the markup
vector of the new node. b) Check if the resulting markup vector has
appeared anywhere else in the tree. We must do (a) before (b), as we
need to know if the new node contains any omegas to be able to
compare it with existing nodes. */
// Check if any omegas need to be inserted in the markup vector of
the new node.
    For each of the ancestors to the new node {
      BOOL All_Elements_Greater_Or_Equal = true;
      BOOL Element_Is_Strictly_Greater[number of places] = false;
      For each place p {
        If (M(p) for the new node less than M(p) for the ancestor){
          Set All_Elements_Greater_Or_Equal = false;
          Break out of for loop.}
        If M(p) for the new node strictly greater than ancestor M(p){
          Set Element_Is_Strictly_Greater[p] = true; }
      }
      If (All_Elements_Greater_Or_Equal==true){
        For each place p {
          If (Element_Is_Strictly_Greater[p]){
            Set M(p) for the new node to be omega;
            Set Found_An_Omega = true;} }
      }
    }

// Check if resulting markup has appeared anywhere else in tree.
  BOOL Repeated_Node = false;
    For each node currently in the tree (except this one){
      If (the markup vector for this node == markup for other node){
        // Node has already been seen.
        Repeated_Node = true;}
    }
    If (Repeated_Node==false){
      Call RecursiveExpansion on the new node to evaluate it.
      //If we've seen node before, don't evaluate the branch further.
    } } }
if (A_Transition_Is_Enabled = false){
  Found_A_Place_With_No_Enabled_Transitions = true;
}
```

3.34 Classification Module

The classification module classifies Petri nets based on the connectivity between Place and Transitions. There are six types of Petri net that can be classified, the equation used to determine the classifications were taken from Stochastic Petri nets by Falko Bause and Pieter S Kritzinger,[10]. The equations are as follows:

$$1.\ \text{State Machine} \ \text{iff} \ \forall\, t \in T : |\bullet t| = |t \bullet| \leq 1$$

$$2.\ \text{Marked Graph} \ \text{iff} \ \forall\, p \in P : |\bullet p| = |p \bullet| \leq 1$$

$$3.\ \text{Free Choice net} \ \text{iff}$$
$$\forall p, p' \in P : p \neq p' \rightarrow (p \bullet \cap p' \bullet = \varnothing \ \vee \ |p \bullet| = |p' \bullet| \leq 1).$$

$$4.\ \text{Extended Free Choice net} \ \text{iff}$$
$$\forall p, p' \in P : p \bullet \cap p' \bullet = \varnothing \ \vee p \bullet = p' \bullet$$

$$5.\ \text{Simple net} \ \text{iff}$$
$$\forall p, p' \in P : p \neq p' \rightarrow (p \bullet \cap p' \bullet = \varnothing \ \vee \ |p \bullet| \leq 1 \ \vee \ |p' \bullet| \leq 1).$$

$$6.\ \text{Extended Simple net} \ \text{iff}$$
$$\forall p, p' \in P : p \bullet \cap p' \bullet = \varnothing \ \vee p \bullet \subseteq p' \bullet \ \vee \ p' \bullet \subseteq p \bullet$$

Additional functionality included in the module is the capability to analyse another Petri net by loading any Petri net file that has its associated XSLT file, the default built in XSLT file is for a PNML file. The modules also supports decreasing or increasing the font size, saving the results as an html file, loading previously generated results and printing the results. This is something that was not included in the specifications of Report 1; it was added to improve usability.

3.35 Comparison Module

The comparison module compares two Petri nets based on their attributes. The user can select the combination of attributes used for the comparison. The results are then displayed in a table using HTML; Java supports the displaying of HTML 3.2. After extensive testing it has been found that the Java class file for rendering HTML are unreliable. HTML files that conform to the W3C standard load incorrectly about one in four times. This is a problem has been unable to be overcome.

Additional functionality included in the module is decreasing or increasing the font size, saving the results as an html file, loading previously generated results and printing the results. This is something that was not included in the specifications of Report 1, it was added to improve usability.

3.36 Incidence & Marking Module

The module is additional functionality not included in the specifications of Report 1. It was added not only to improve usability but also to enable easier conformation the results produced by the product. This module therefore enables the user to feel more confident about the validity of the PIPE because the intermediate results used to generate the more advanced results can be viewed. The incidence & marking module displays incidence and marking matrices and the enable transitions set. The module also supports decreasing or increasing the font size, saving the results as an html file, loading previously generated results and printing the results.

3.37 Integration of Predator

In order to demonstrate the ease of integration of other modules to PIPE, the Predator[11] Invariant analysis module (PIAM) was integrated. The integration was made by writing a simple wrapper class, which conforms to the PIPE Module interface and calls the runModule() method of PIAM. The wrapper class also used the savePNML() method of class PNMLData in order to save the object, which contains all the information about the Petri Net under analysis to a path/filename specific to PIAM. This is where PIAM looks for an XML file describing the net, and uses SAX in order to read the information. Also, the wrapper class includes a getName() method and a corresponding static variable of type String, so that the module can identify itself to PIPE.

The integration was successful, in the sense that the module is loaded correctly using the reflection techniques implemented in PIPE, and runs producing an output window. However, there seems to be an internal malfunction to the module, which causes it to produce incorrect results. After careful examination of the PIAM source code it was concluded that this behaviour is likely due to a bug in the module, and fixing it was clearly beyond the scope of the project. The module integration was considered successful, especially given the fact that this module was written for a completely different Petri net editor, without any compatibility with PIPE in mind.

3.4 Other Methodology

Version Control

Throughout the life of the project we used used the Concurrent Versions System[12] to manage our code. CVS enabled us to share our code effectively, allowing multiple developers to work on the same files at the same time, but with a mechanism to prevent one developer overwriting the changes of another, and to allow the changes both have made to be merged together. It provided one central, master copy of the code which everyone could easily get a copy of to work on, and allowed team members to easily update their personal, working copy of the application with any changes made by other developers.

It was found that CVS, although a little unintuitive to start with, made the project much easier to manage. It provided an ongoing history of the development of the project, and a safety net, should code be accidentally deleted or overwritten.

Installer Implementation

In order to make installation easier, an installer jar file was created. This was tested on a Windows 2000 and a Windows XP computer and it appears to work without problems, making the installation process quick and easy. For the installer, the software IzPack was used[13]. Given some more time a similar installer file for UNIX/Linux platforms could be created, and also shortcuts could be added to make the program more user friendly.

3.5 Usability Testing

An extension of the project was to submit PIPE for user testing. The motivation behind this was to reflect the development life cycle and more importantly, to ensure that PIPE is as user-friendly as possible. Petri net tools are very specific, therefore it was necessary that those testing PIPE were familiar with Petri nets and had used Petri net tools before. For this reason, the users chosen were people had written their own Petri net editors in previous projects.

Brief instructions and a feedback form outlining some standard tasks were sent to the users (see appendix for results). The main focus of the testing was to check how easy the application is to use, does it work and what improvements can be made.

Results
The rating scheme used was 1 to 5, with 1 representing excellent and 5, poor. The pertinent results are as follows:

- An overall rating of 1.5
- Specific tasks got ratings in the range of 1 to 3. In general, the users found it very easy to use.
- When asked if they were able to achieve what they intended, the users gave it a rating from 1-3.
- Users liked the animation history
- There were some permission issues with the application that meant that the modules did not run and the saving and loading was problematic.

Improvements
- Users preferred to single click on enabled transitions to fire them rather than the double clicking in place at the time of testing.
- One of the buttons did not show a “selected” image
- The transition labels did not show up unless specified by the user. Therefore the animation history did not necessarily make sense on first using.

- The animation button icons were not intuitive
- The random animation only showed the results at the end of the sequence. Showing the firing as it takes place could enhance it.

Changes Made

- The permissions for saving and loading were checked to ensure that the application works satisfactorily
- The firing of transitions was changed so that a single mouse-click fires the transition
- The random firing button icon was replaced with a text button and made larger. This makes it easier to understand and easier to use.
- Petri net elements now have their ids displayed as a default so that the animation history makes more sense to the user.
- It was decided to make the toolbar dockable so that the user can position the toolbar wherever convenient on the screen to facilitate the drawing and editing process.
- The status bar was enhanced so it provides information for the “mode” that the user is in and what options are available in that mode.

4.0 Conclusion

Difficulties
The main difficulty of the project overall was the interactions of the various parts of the application. This is undoubtedly one of the biggest difficulties of any project and we gave it due consideration, thus allowing us to achieve a working application.

From the GUI point of view, the main difficulty was in choosing the correct component structure for our needs: selecting the right components to extend for the Petri net elements and selecting the right container to hold them. Once this structure was in place and the MVC architecture was finalised, the coding progressed fairly easily. The transition from a single document interface to a multiple document interface was completed efficiently and with relative ease, demonstrating that our program design was flexible and solid.

Code management difficulties within the group were minimised by the use of CVS and a sensible application directory structure.
.

Fulfilment of the Specification and Extensions for the Future

GUI
Both the minimum and maximum specifications laid out in Report 1 were implemented with the exception of printing and cut and paste operations. It was decided over the course of the project that these features would not be an integral part of a Petri net application. The user testing confirmed this. As a result, we were able to focus more on the animation, again meeting the basic and enhanced functionality and adding the random fire operation.

Given more time, the speed of the GUI could be improved. The tabbed panes slow down the overall performance however, it was decided that the functionality this gave outweighed the decrease in performance but this could be improved upon to give the user both. For the animation, an extension for the future would be to implement the animation so that the user could see the random firing as it happens.

The user testing demonstrated that PIPE met the requirements for an intuitive and easy-to-use editor as PIPE scored highly for ease of use and functionality. The results from the user testing were also incorporated into the final product, refining the application even further.

Data Layer
This is very powerful and is designed well to help the programmer (future module builders). It has advanced features built in to help module builders such as function to return all Petri net matrices (current markup, initial markup, incidence, forwards incidence, backwards incidence). Many methods have multiple return types, such as a choice between int or Integer.

Modules
We have produced as part of the user interface a Module Manager, which allows the user to load and unload analysis modules as, required. This has been implemented as an expandable tree dynamically populated with all the available analysis modules residing in the application lib directory when the application started. In addition to collapsing the tree, users can also completely remove modules as well as adding new modules using a file
chooser. The use of the Java Reflection API to do this was relatively straightforward to implement, and has worked well. We did not implement a facility to allow the user to load modules existing outside the JVM, e.g. from a network resource, as this was not deemed a vital feature.
The module interface is flexible and simple requiring only two functions for the future module builder to use to integrate their modules. This allows PIPE to be extensible, preventing it from becoming obsolete.

A summary of the modules provided is provided below:

- More modules than other similar products. Most only have two.
- The Invariant Analysis module works well and is accurate and efficient
- The Simulation module works well providing performance results on the Petri nets. Given time, one might add a progress bar whilst it is analysing the nets.
- PIPE is the only product that has comparison and classification modules therefore increasing PIPE's versatility. There is a small issue with the displaying of HTML in Linux. This could be improved by using labels and text fields but would require a much more complex and flexible approach.
- The State Space Analysis module works well providing accurate results. It has a novel feature showing the shortest path to deadlock. The module calculates boundedness, safeness, deadlock and the shortest path to deadlock. The algorithm will terminate if 10,000 nodes are generated in the state space tree. If this occurs, it's likely that the state space has exploded, and hence other algorithmic approaches are better suited to solving the problem. The module implements the underlying framework to perform similar analysis tasks on the state space tree. Additional tests could be added to the code (such as determining the number of deadlocked markups, etc) comparatively easily, as the necessary framework is already in place.

Overall, PIPE satisfies all the requirements laid out in the project outline and the specification of Report 1. PIPE fully implements an elegant, easy-to-use graphical user interface with full functionality for creating, saving and loading Petri nets conforming to the XML Petri net language standard. PIPE is a Petri net tool that supports the loading of analysis modules at run-timer and supports user defined modules through its simple interface. PIPE also comes with a full suite of analysis modules producing performance statistics, correctness properties and new features such as Petri net comparison and classification.

Sources

[1] Stochastic Petri Nets – An Introduction to the Theory – Falko Bause and Pieter Kritzinger, 1995

[2] Stochastic Petri Nets – An Introduction to the Theory – Falko Bause and Peiter Kritzinger, 1995.

[3] Petri Net World - http://www.daimi.au.dk/PetriNets/

[4] http://www.cs.indiana.edu/~cbaray/projects/mvc3.html

[5] JAMA: A Java Matrix Package- http://math.nist.gov/javanumerics/jama/

[6] Stochastic Petri Nets - An introduction to the Theory, Falko Bause and Pieter S. Kritzinger, 1995

[7] Stochastic Petri Nets - An introduction to the Theory", Falko Bause and Pieter S. Kritzinger, 1995 [DEF 4.8]

[8] Stochastic Petri Nets – An Introduction to the Theory – Falko Bause and Peiter Kritzinger, 1995, [DEF 4.12]

[9] Concurrent system analysis using Petri nets – an optimised algorithm for finding net invariants", Mario D'Anna and Sebastiano Trigila, Computer Communications vol 11, no. 4 august 1988.

[10] Stochastic Petri Nets – An Introduction to the Theory – Falko Bause and Peiter Kritzinger, 1995. p.122

[11] Predator, a hierarchical Petri Net Editor http://www.mark.wass.com/Petrinets.html

[12] http://www.cvshome.org/

[13] IzPack installation software, http://www.izforge.com/