PROJECT REPORT

PETRI NET ANALYSER – GROUP 6

18 MARCH 2002

RAPHAEL GOLDBERG, PETER HOWARTH,
DANIEL JUSTICE, MICHELLE OSMOND,
ZHU TAN, ALEX TRIFUNOVIC

SUPERVISOR: DR WILLIAM KNOTTENBELT

Acknowledgements

Our thanks to Dr William Knottenbelt for his enthusiasm and support, to Michael Wright for his advice on XML, and to Nick Dingle for starting us off on the right track.

Abstract

This report discusses the design and implementation of a Java program EXPT (EXtensible Petri net Tool) to edit, animate and analyse Petri Nets.

Analysis modules may be written by following the specification in the given module interface, and loaded dynamically while the program is running. Two modules were implemented to demonstrate this capability – an Invariant Analysis module and a State Space Analysis module. Modules may also be written to interface between the program and existing analysis software – this was demonstrated by producing an interface to DNAmaca, an existing C++ Markov Chain Analyser.

The latest Petri Net Markup Language (PNML) specification, with subnet (aka module) support, is used in the implementation for both external and internal storage of data. Extensive use is made of the JDOM API to store and manipulate the PNML, allowing a level of portability and, in particular, extensibility, which is not available to most other Petri net tools. Support for multiple Petri net types is included in the design, in such a way that support for more types may be easily added in the future.

§1 Introduction

§1.1 Introduction to Petri nets

Petri nets were invented in 1962 by Carl Adam Petri[1] 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. Place-Transition nets are the basic form of net whose definition is a subset of the definitions of more elaborate types of net.

Place-Transition nets can be graphically described as follows:

Places, drawn by circles, which may contain **tokens**, drawn by black dots. **Transitions**, drawn by rectangles, which are connected to places by directed **arcs** and can **'fire'**, creating one or more tokens on the place pointed to by the arc leading from the transition, determined by the weighting of this arc. One or more tokens are 'destroyed' at the places with arcs pointing to this transition, again determined by the weighting of these arcs. Transitions may only 'fire' when they are **'live'**, ie when the places whose connecting arcs point to the transition contain at least as many tokens as the weighting of the arc (see Figure 1).

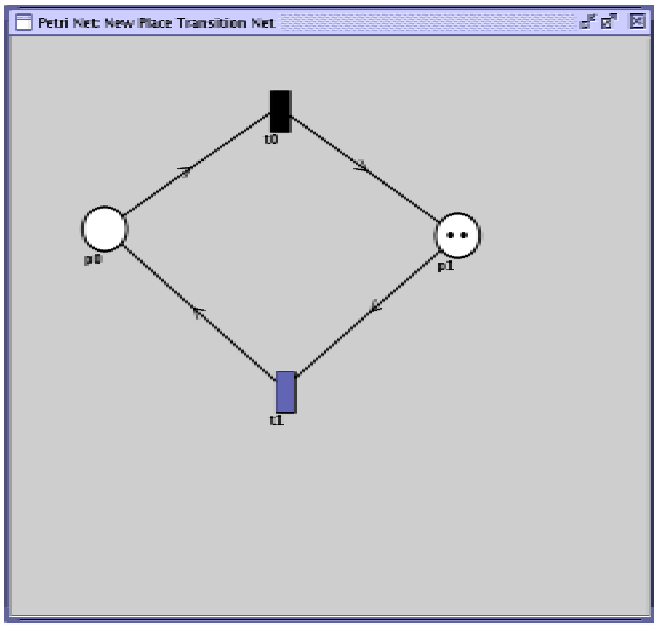


Figure 1

Formal definition:

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

- $P = \{p_1, \ldots, p_n\}$ is a finite and non-empty set of places,
- $T = \{t_1, \ldots, t_n\}$ is a finite and non-empty set of transitions,
- $P \cap T = \varnothing$,
- $I^-, I^+ : P \times T \rightarrow N_0$ are the backward and forward incidence functions respectively. Applied to (p_i,t_j), the functions return the change in the number of tokens on place p_i after transition t_j has fired (assuming that it is live),
- $M_0 : P \rightarrow N_0$ is the initial marking.

More sophisticated Petri nets include Coloured Petri nets with tokens of differing colour, Time-Augmented Petri nets, where an enabled transition fires with a probability $p(t_i, x)$ and Stochastic Petri nets, a subset of the latter, where the probability of firing has an exponential distribution.

§1.2 Tools

Petri net tools are widely available and typically take a Petri net specification, performing analysis and returning results describing properties of the net, which are then used to identify properties of the underlying system modelled by the Petri net itself. A comprehensive list as well as further information can be found in Petri Net World[2].

In the face of evolving theory and techniques, a key consideration for the design of our tool has been extensibility, without which the tool could quickly become obsolete. Important issues are:

- The functionality of the editor, ability to consider different types of net and present a wide range of features to the user.

- The ability to extend the capacity of the program to analyse the nets beyond that which is available with the application. Introduction of analysis modules which can be 'plugged' into the application to perform analysis and then 'unplugged'.

- Representation of the nets using a universally accepted standard.

§2 Project Outline

§2.1 Tool Specification

The aim of this section is to describe the functionality that the finished product will support and possible extensions to this that have been considered.

Taking the points raised in §1.2 into consideration, the following product specification was considered to meet the above mentioned criteria:

- Design and build an elegant and functional GUI that allows the user to both create and edit new Petri nets and to load and edit existing Petri nets.
    - The user may create and edit different types of Petri net.
    - The user may load from disk and edit existing Petri nets.
    - The user may save edited or newly created net to disk.
    - The editor should allow the creation and display of hierarchical subnets, utilising a subnet interface.
    - The user should be able to manually ‘fire’ transitions and invoke a graphical animation of the transition firing and movement of tokens.
    - The editor should be able to animate a sequence of firings to demonstrate a particular property of the net.
- Petri nets should be saved using Petri Net Markup Language (PNML). PNML is based on XML and is becoming the standard for representing Petri nets. This will enable nets to be passed between modules and other Petri net tools.
- The editor should allow the creation and display of hierarchical subnets - including a subnet interface.
- Within the GUI the user should be able to manually fire transitions to animate the displayed net.
- The tool should be able to add modules dynamically while the program is running, so that they appear as new menu options.
- Build a module to analyse the Petri nets using invariant analysis
    - to calculate and display the invariants of the Petri net.
    - as a corollary of calculating the invariants, to determine properties of the net such as boundedness.
- Build a module to analyse the Petri nets using state space analysis
    - To formulate the coverability tree of the Petri net.

- To analyse the coverability tree to determine properties of the net such as liveness

§2.2 Extensions

The following are extensions we are intending to provide in our final specification:

- Through the use of PNML and the JDOM API the tool should be able to cope with new types of nets that are represented as extensions to PNML.
- The interface could be extended to include a method for linking analysis back, e.g. showing how deadlock happens.
- Automated animation within the GUI, with random firing of enabled transitions.
- Develop additional modules:

An interface module to an existing Markov chain analyser.

Finally, the language of implementation was chosen as Java in view of its portability and the wealth of features offered to the programmer. The reduced performance over languages such as C++ was viewed as less significant in the context of the project.

§3 Technical Methodology

§3.1 Introduction

This introduction covers the main technical decisions that needed to be taken in determining the structure and format of data that required saving and parsing between the modules and editor. This key issue had to be decided upon before the main program could be written, and had an important bearing on the capabilities of the program.

§3.1.1 File format[3,4,5,6,7]

As indicated in the initial specification, one of the latest XML Petri net standards was used to represent the nets - the Petri Net Markup Language (PNML). PNML is envisioned as being capable of representing all types of Petri nets, by means of extensions to the basic PNML called Petri Net Type Definitions (PNTDs). It is thus very useful as an interchange format, with the same data file being usable in different applications. Such applications may be written to use slightly different versions or extensions of the basic PNML, but in many cases they may be able to ignore unrecognised data and proceed as normal. Preferably, unrecognised data should be left in the file untouched, so that a more recent or capable program is still able to make use of it. (There is also the related issue that non-PNML data may be present in the file - this should be left undisturbed for similar reasons.)

With the extensibility of PNML to support different net types comes a new challenge and potential, that could not be easily considered with a program-specific file format. These extensions all consist of the same basic PNML, plus varying numbers of additional elements defined in a new schema (PNTD) for that net type. This schema will typically be derived from, and reference, an existing net type such as the basic Place-Transition Net (**ptNet**). Few such PNTDs are available currently, but many more are likely to be defined in the future.

The type of a net is stored as an attribute in the **net** tag (eg `<net id="net1" type="ptNet">`). The program may thus examine the **net** tag and treat the net accordingly for its type.

§3.1.2 Internal representation of Petri nets[7,13,14]

The majority of previous projects in this field have concentrated on building up an internal object hierarchy to represent the Petri net and its elements. This hierarchy would contain classes for Places, Transitions and Arcs, which hold their data in member variables and contain appropriate functions. The program would typically read in a PNML file using SAX or DOM (see below), simultaneously extracting the relevant information and creating objects to represent it. For output, the program would examine the current status of the objects, and then create and output a PNML representation. This

method suffers from being a lossy transfer, in that unrecognised data will not be represented within the program, and is unlikely to be stored in the output file.

However, there is an alternative refinement to this approach. PNML is not restricted to use as a storage and transfer medium - the Document Object Model (DOM) it directly corresponds to can be held in memory and used within a program to represent the data. By maintaining this DOM in memory as the main data structure, all information in the original input file may be stored and even manipulated, whether the particular format was foreseen by the programmers or not. Storing all of the available information enables the program to be written to automatically, read from and allow editing of new net types without (necessarily) any changes to the program. If changes were to be made, they would be far easier to implement, as the basic file input and output program initialisation routines and data storage structures would not require modification.

It was decided, therefore, to maintain the data representing the nets in a DOM, with the display aspects of the program making use of the more usual design methodology. The display uses an object hierarchy which is largely similar to those used in previous projects, but differs in that the objects do not store any local data. Each object contains only a reference to its element in the DOM, and appropriate access functions. Further subclasses of elements will then automatically have access to all additional information they require, simply by querying the DOM.

It was suggested that some frequently accessed information could be stored locally in these classes - however, at the Display level, few functions make frequent use of such stored attributes. The most intensive use would probably be the Animation component, where the overhead due to DOM access is small compared to the hard-coded visual animation timestep. Storing a local copy of data also leads to more update problems, and hence it was decided to restrict all PNML data to the DOM.

§3.1.3 DOM, SAX, JDOM[8,9,10,11]

There are several standard techniques available to Java programmers for reading and writing XML files. The Java libraries from Sun provide two main methods: DOM and SAX.

SAX (Simple API for XML) allows fast parsing of an input XML file, providing an event-based interface which allows the programmer to read in data, and, typically, store it in the program's own internal data structure. It provides output to XML format in a similar way.

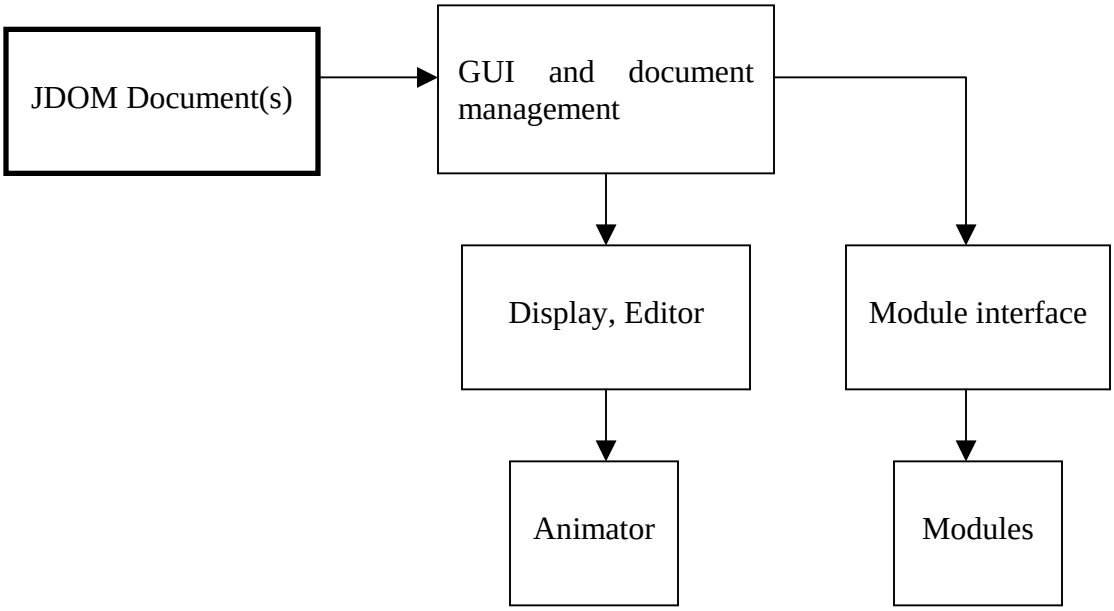
The XML Document Object Model (DOM) is a programming interface for XML documents. Sun’s Java DOM implementation is slower, but more complex, then SAX. It takes an XML document, which in our case is a Petri net, and represents it as an object tree held in memory. No custom programming is necessary on the part of the programmer when reading in and creating this object tree. The nodes of the tree correspond directly to the elements of the XML document. In the DOM specification, each element of an XML

document is represented down to the smallest detail, so it is possible to build up a complete concrete model of the Petri net system. The DOM also defines a set of interfaces that allow access (and modification) to this model.

JDOM is another API which offers an alternative to DOM and SAX, which has been accepted as JSR-102[12]. The JDOM API takes the best concepts from existing API's, and creates a new set of interfaces which incorporates them. It claims to be more efficient and faster in most cases than DOM and SAX, and also takes full advantage of standard Java techniques and resources. The in-memory representation of the DOM tree can therefore be manipulated in a consistent and natural way. For these reasons, JDOM was considered to be the optimal choice for this project, and was used for reading, writing and manipulation of the XML (PNML) data within the program.

§3.2 Overall Program Design

In terms of data flow, the program is based around the JDOM Document structure that holds the net information. Once a data file has been opened, the individual components maintain their own direct link to the relevant Document structure. The initial data flow is shown below:



The actual code can be broken down into a similar structure.

The **Graphical User Interface** and main Application implementation deals with opening and saving documents, initialising other parts of the program, and maintaining state information (such as which editing function is active).

The **Editor** window is initialised with the part of the JDOM Document it has been assigned to deal with (such as a net or a subnet). By inspecting this Document it creates

Objects (PNLabels) which manage the on-screen representation of the elements in the Petri net, such as places and transitions. Each PNLabel maintains a link to its Element within the JDOM Document, so that all changes in the data are reflected directly in the Document – no core information about the net is stored in the PNLabels, to avoid update problems. However the PNLabels do store information about their current display state, and have much of the functionality that would have been expected with a purely object-based data storage system.

Subnets are catered for by a subclass of the Editor window, extending its functionality and display. Further PNLabels are also necessary for the subnet interface.

An **Animator** is initialised for each Editor window – this permits extensibility, as different Animators may be used for different types of Petri net. When running, the Animator changes the state (marking) of the JDOM Document, so that the user may choose to pause the animation, fire transitions of his/her choice, and/or analyse the result. It stores a backup of the Document before running, so that the user may also choose to return to the initial state after watching the animation.

The **Module Interface** manages loading and unloading of modules. When running a module, it determines which Editor window is active, and passes the module a reference to the relevant Document object. The module may then access the data using the JDOM, and possibly transform it into a more convenient format for analysis. A ModuleBridge object was created as a link between the modules and the main program, allowing modules to request that an animation sequence be shown.

§3.2.1 GUI

Look and feel

An initial decision had to be taken whether to use a Single or Multiple Document Interface in the program. A single document interface is easier to implement, and more straightforward for the user to determine what data is being worked on. A multiple document interface allows the user to work on multiple Petri nets simultaneously, eg for visual comparison or copying components between nets. It can also be useful in implementing subnets: multiple windows allow viewing of the subnet at the same time as the main net.

In fulfilling the program specifications, we have adopted a standard format for the positioning of the respective graphical elements, to produce an intuitive interface. The menu bar is placed at the top of the display and the toolbar below this, whilst the main desktop is used to display open documents (Petri nets). All of the menu items are given a mnemonic to allow fast access to the menu items without the use of a mouse. In addition the more common commands are equipped with keyboard shortcuts to allow an experienced user to perform operations rapidly (eg. *Ctrl+x* – cut, *Ctrl+v* – copy). Buttons for selecting a tool or an item to insert that are mutually exclusive are represented in such a way that they appear differently when they are selected. All of the buttons are also

represented on the menus for completeness. A final important point regarding the look and feel of the tool is that it mimics the native platform it is run on.

Basic functionality

The tool allows the creation of different types of net. The types of net available are extensible, as they are specified in a configuration file that may be modified at any time. Into these nets it is possible to insert places, reference places, transitions, arcs, etc. Tools to allow a net to be edited are also provided to the user, making the manipulation of complicated nets easy. These tools allow the user to cut, copy or paste individual elements, or large sections of a net. Once a net has been completed it can be stored to disk and then retrieved at a later date (by using the file menu commands).

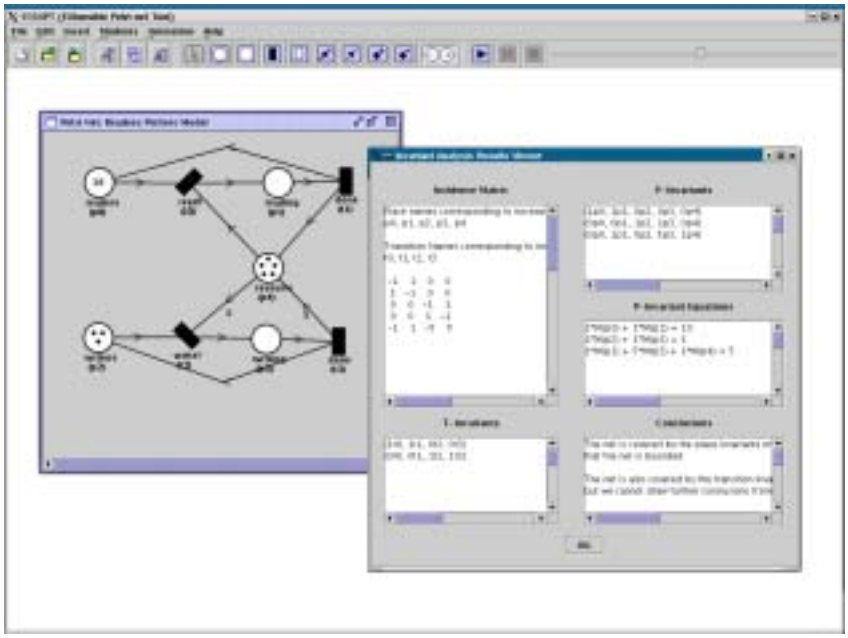


Figure 2: Output of Invariant Analysis module analysing the Readers Writers net

Display

The representation of Petri net elements within the program must provide for the display, animation and editing of the net. The main data is stored within the internal DOM representation (see Figure 3); however it is clear that more functionality is required.

The dominant requirement is the display and interactive handling of the elements, and so it was decided that objects representing Petri net elements should be derived from the standard Jlabel, which provides display, positioning and MouseListener capabilities. Each PNLabel contains a reference to its own element within the DOM, along with functions which provide the interactive capabilities. The exact details of each label type can be found in the Javadoc accompanying the program.

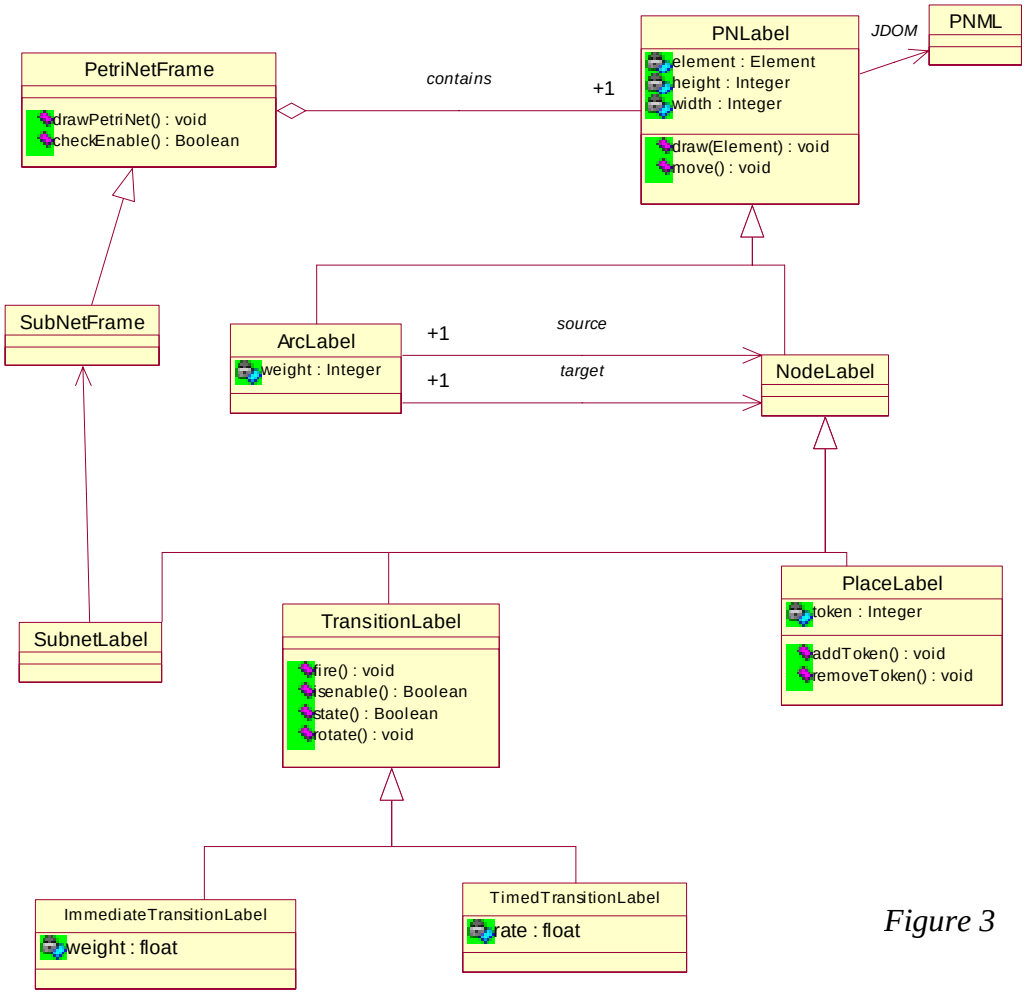


Figure 3

PNLabels are interlinked in a manner corresponding directly to the structure of the Petri net: a NodeLabel will contain references to its input and output ArcLabels, whilst the ArcLabel will contain references to its source and target NodeLabels. This allows for intelligent handling of editing and animation

§3.2.2 Subnets

Alternatives

A review of existing Petri net tools revealed that there was no consistency for the representation of subnets, with many tools not implementing subnets at all. Of those that did they all had their own proprietary implementation rather than the PNML scheme described in a paper on the PNML website[5].

The options available were to either design our own interface, or to try and follow the relatively recent PNML standard. Our own design would probably allow an easier implementation as the XML representation would not be tightly constrained. In addition

it would give us a free reign over the look of the subnet interface to make it intuitive to use.

On the other hand, the PNML scheme has the benefits of being a standard, albeit a one that has not been used yet! It has a consistent modular approach. Within our overall design philosophy of extensibility and using/setting standards, it was decided that modular PNML should be implemented. Producing the first tool to use this would allow us to review the practicality of the scheme.

Subnet interface

The requirements for the support of subnets are:

- creation and saving of a new subnet
- addition of instances of a subnet to a main net and linking them into the net
- flattening of the subnets so that their representation can be interpreted by analysis modules

The key component of the subnet representation is its interface. This contains import and export places that are available for import or export to the main net. Only the interface is available to connect to and visible in the main net.

When a subnet is created the interface is clearly shown separated above a solid line; with the implementation shown below the line. The saved subnet can then be instantiated within a Petri net. The instance appears as a box containing only the nodes within the interface (see Figure 4).

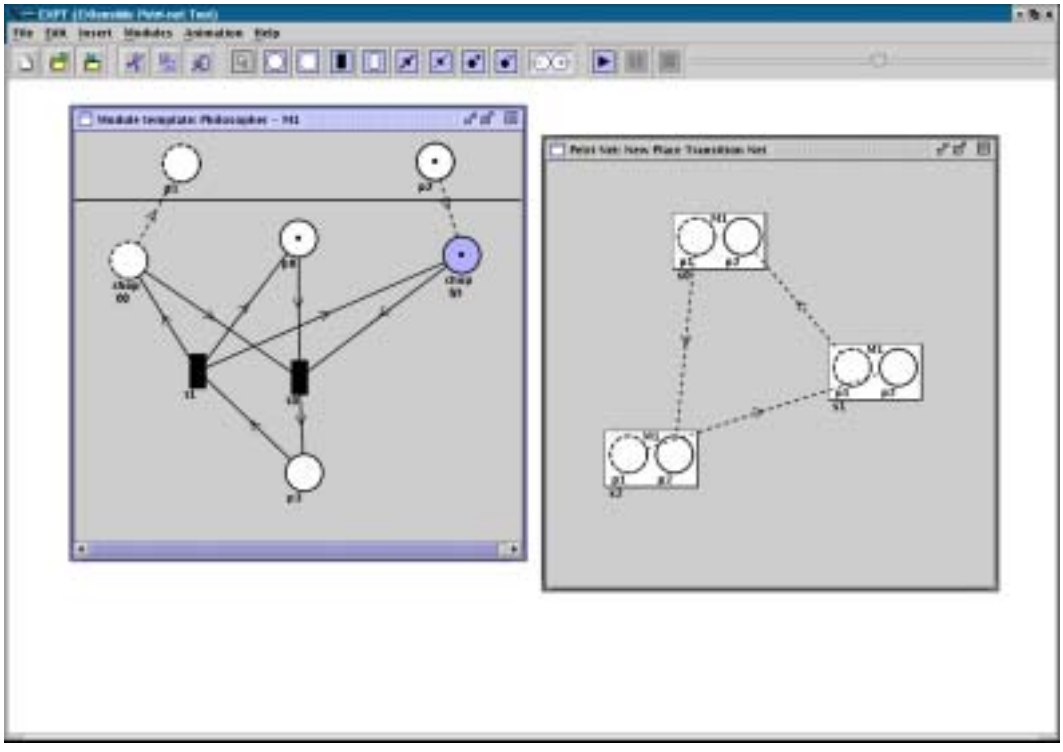


Figure 4

Subnet elements

Early on in the implementation it was decided to constrain the specification to allow only 2 places within the subnet interface. This simplified the implementation without significantly compromising the functionality and still allow us to investigate whether the PNML scheme would work. The generalisation of the interface could be a good future extension to the tool.

To allow code reuse and common behaviour the subnet components were subclassed from existing PNLabels wherever possible. The representation of the instance of a subnet within a net was the most complex as within the PNML it is an element that contains further elements (import and export places). The PNML structure was mirrored in the implementation.

Subnets also introduced references. A reference place contains an attribute indicating which place it refers to. In effect the reference place does not exist within the net, it is just a representation of the place it is referring to. Reference links are graphically represented as dotted arrows.

§3.2.3 Animation

Animation is performed by repeatedly examining the net for active transitions, firing one and displaying the result. Reference texts typically describe changes in state of a Petri net as transformations on the matrices representing the net, producing a new marking (a description of how many tokens are on each place)[15]. While efficient, this technique would require a rewrite of the transformation algorithm if the behavioural characteristics of any part of the net were changed (for example, the introduction of new arc types which impose different requirements on the input places to be active).

The animation technique was designed with the aim of allowing extensibility, so that the behaviour of any individual Petri net element may be changed without necessarily requiring a complete rewrite of the animation code. In the above example, this would correspond to overriding animation code within the new ArcTypedLabel object, but no changes to the original PNAnimator. The animation works as follows:

1. PNAnimator: looks through all the TransitionLabels and queries them to determine which are active.
   a) TransitionLabel: queries its input arcs to find if they are active.
   b) ArcLabel: queries its input places whether they are active.
   c) PlaceLabel: given the arc weight, replies whether it contains enough tokens or not.

2. PNAnimator: chooses a transition to fire from its active list.

3. PNAnimator: requests the chosen TransitionLabel to fire itself
   a) TransitionLabel: requests its input and output arcs to unfire/fire themselves.

b) ArcLabel: removes or adds tokens from their input/output places as required.

The PNAnimator manages the overall animation, so that the mechanics within the NodeLabels may be modified or interrupted as required by the type of net. In the Typed Arcs example, the animation at 1.b) may be interrupted if the arc is an Inhibitor arc, which is active only if its input place has no tokens. Hence the isActive() function in ArcTypedLabel may be modified to check that there are no tokens on the input PlaceLabel, rather than passing the query onto the PlaceLabel in the normal fashion.

More widespread changes in behaviour do require new functionality in the Animator. For example, in a Generalized Stochastic Petri net there are two types of transitions: timed and immediate. If an immediate transition is enabled, it will always take priority in firing over a timed transition. This logic must be built into the Animator, and hence a subclass of PNAnimator must be created.

The animation management all takes place at the PNLabel level of the program – it does not access the DOM directly. A decision had to be made – whether to run the animation entirely at this surface level, leaving the DOM unchanged and storing local data (eg markings) within the PNLabels, or to change the 'real' DOM data during the animation. There are advantages and disadvantages to both techniques.

Storing local data within the PNLabels would give faster access than looking up the information from the DOM; however this is unlikely to be relevant, given that the animation rate must be slow enough for the user to follow what is happening. It would also lead to complications in the display, as the PNLabels must know which data is to be shown on screen at any time: either locally-stored or DOM data.

The main point under discussion, however, was whether the animation should change the DOM or not. Changing the DOM during animation is an advantage if the user wishes to pause the animation and then run analysis on, save, or make modifications to the net in that particular state. It is a disadvantage if (as in most cases) the user wishes the net to return to its initial state after animation.
However, with changes to the DOM, the best of both worlds may be achieved, by making a backup copy of the original DOM and providing an option to restore the net after animation. It was therefore decided to make changes the the DOM during animation, to provide the same advantages as using local data but also allow more flexibility, and avoid update problems.

Subnet animation is more complicated, but may follow the same basic concepts. Elements which reference others (such as ReferencePlaceLabel) must appear to provide the same functionality as the element they reference, so that external requests may be passed on to the 'real' element. Animation of a subnet 'instance' may be achieved by creating the instance as a temporary, separate Document which opens in a new window, so that the animation within the subnet may be displayed.

§3.2.4 Module Interface and Dynamic Loading

One of the main aims of the interface was to enable run-time loading of user-designed modules to increase the analysis power and extensibility of the tool. The following features were considered when designing the module interface:

Each module only requires access to the DOM in which the data for the Petri net is stored, and can be entirely unconnected with the rest of the editor program. Moreover, the module can be dynamically loaded and unloaded and will appear as a menu item while it is loaded. To achieve this, we used the Java Reflection API, which provides a mechanism by which another program can discover information about a Java class which is not known until runtime, to enable the modules to be loaded and run. The module which the user wishes to load can be selected and then the instance of the class will be created by the Reflection API. In order to simplify the process of loading modules, they are limited to having a no-argument constructor. The data and other parameters which need to be passed to the module can be passed as parameters of the `runModule()` method.

There was some discussion on whether to pass the modules a reference to the current JDOM Document, or a copy. The key point here, which may be both an advantage and a disadvantage, is that if given a reference, the module may modify the original data. This is a disadvantage if the modules are untrusted, but an advantage if the modules are trusted – modules could then be written specifically to transform the original data for particular purposes. It was decided to pass a reference due to the larger possibilities for modules it opens up.

Modules should not need to worry about whether the net contains subnets or not. Most existing Petri net analysis modules can only deal with a flattened net (without subnets). If a module cannot deal with subnets, the interface flattens the net within the DOM automatically and passes a new version of the DOM containing the flattened net to the module. The method `isSubnetable()` in the module is called by the module interface to determine whether the module supports the analysis of subnets. The ID and name of all components in the subnet are changed to `_subnetID_componentID`, so the user can still easily locate the position of a component after analysis.

One other significant element is the mechanism for feedback from the modules to the main program. This is a ‘bridge’ object between the module and the display for the net under analysis. The module may request the bridge to perform particular functions – such as the animation of a given sequence of transitions, or refreshing the display completely (if the module has modified the DOM) – and the bridge then calls the appropriate functions in the Animator and/or the Editor window. In this way, the module writers have a well-defined interface, and do not need to know about the internal structure of the program. The methods that can be called in the bridge are included in Appendix E, and described in the Javadoc for the ModuleBridge class.

§3.2.5 Extensibility – Support for different net types

As previously discussed, the use of PNML and the internal DOM representation allows the program to be designed to support multiple net types.

As an initial demonstration of the power of these technologies, the Editor part of the program was written to allow editing of any type of net, regardless of the specific contents of the PNML. This was accomplished by writing a Properties Box which would allow the user to edit the attributes of any PNML element displayed on screen, by examining the contents of the element within the DOM, and dynamically creating editing fields to match. While this does not provide active recognition by the program of the meaning of new attributes, or any change in the way it treats the Petri net elements, it does allow the user to edit any attributes (even non-PNML attributes) that may exist in the document.

Most new types of net will extend existing net types, with this extension perhaps only affecting a few of the Petri net elements – for example, a Petri net with typed arcs (**ptNetArcTyped**) extends the basic **ptNet** by adding a new `<type>` element to the arc definition. In such cases, most of the basic code for **ptNet**s may be inherited unchanged. The level of support for a new net type added to the program may vary depending on the user's requirements and capabilities.

This is catered for in three main design elements:

1. **Creation of Labels and Animators upon loading a new PNML file.**
   This was achieved by requesting a ClassManager object to create all new PNLabels, so that the ClassManager may choose the particular type of PNLabel to create. A ClassManager is created for each Editor window, so that upon loading, it may discover the type of net displayed in that window, and uses Java Reflection to look up and instantiate the corresponding classes of PNLabels. Information on the net type hierarchy, with corresponding Class information, was stored in an external file (in XML format itself), as the PNML documents alone do not provide sufficient information to permit automatic discovery of the hierarchy. This file may be modified by the user to introduce support for new net types, and specify classes to use.

   The type of net may affect the algorithms used to perform animation of the net, so different Animator classes may need to be created. Again, the ClassManager decides which class of Animator should be used, based on the net type. A new Animator is assigned to each Editor window.

2. **Choosing an appropriate Properties Box for a particular type of element, and automatic detection of further (unspecified) attributes.**
   A properties box (for viewing and editing element attributes) was written (as described above) to configure itself automatically by examining the JDOM Element. This has the advantage that all attributes may be discovered and edited, regardless of whether the programmer knows they exist, or will exist in the future. However, a

modification to this approach was necessary, as some attributes are optional: they may not currently appear in the file, but the user may still wish to enter them. An automatic discovery process cannot discover what isn't already present; hence some hard-coding was also necessary to specify which PNML attributes are expected for a particular type of element, and to enforce restrictions on particular fields.

In addition, some elements may not be necessary and should be hidden from the user: this may also be specified in the code for a particular type of Properties box. Element types which require modifications to their properties box are also highly likely to require a custom PNLabel sub-class; therefore it was considered sufficient that a particular PNLabel class contains a hard-coded association with its particular type of Properties Box. This was the approach adopted, for simplicity.

The alternative method under consideration was to produce an externally-configurable properties box, using a configuration file with structure and field information. A further refinement would be to parse and interpret a Schema for a given PNML file, and build a properties box with appropriate fields and restrictions based on this. However, limited numbers of schemas for PNML net types currently exist, in various formats[3]. In addition, complications would arise as there is no direct link within a PNML file which could allow the program to seek out the appropriate schema - this approach would also assume a connection to the internet is available while running the program.

A related issue concerns the Classes for the right-click context menus that appear for each Petri net element – these are unlikely to require modification, but if they do, a similar approach may be used.

3. **Provision of an appropriate PNML format to modules.**
   Different modules will be able to deal with different levels of PNML. The basic PNML has recently been defined to include subnets – however many modules will not be able to interpret this structure, and so the program must be able to expand and eliminate the subnet structure ('flattening') before giving the net to the module. On the other hand, future modules may be written which can deal with and take advantage of the subnet structure, so arbitrarily flattening the net before passing it to any module would be unwise. It was decided that the modules should be able to provide certain information about themselves – therefore, by calling a predefined function in the module, the module manager can find out if the module can support subnets. If so, it should pass the net to the module unchanged. If not, it should flatten the net before sending it to the module (feedback from the module to the main program is possible in this situation with a suitable flattened naming scheme, and interpretation on the part of the ModuleBridge).

   A similar problem arises with different net types – however, as new net types may be defined at any time, it is impossible for the module author to state what future types the module will be able to support. The most likely (if not optimal) solution may be to simply display a statement to the user of which net types are supported, with a

warning that the module may not work as expected with other net types. This is a matter for the individual module author to decide.

§3.3 Modules – Introduction

The properties of a Petri net can be analysed in a number of ways. The tool was tested with three different analysis modules. Invariant Analysis and State Space Analysis modules were written and an interface to an existing Markov Chain Analyser was also constructed.

§3.3.1 Invariant Analysis Module

Background to the analysis

Given the general definition of a Place Transition net PN = (P, T, I^-, I^+, M_0), it is possible to represent the forward and backward incidence functions as matrices. The forward and backward incidence matrices are respectively:

$C^- = (c^-_{ij}) \in N_0^{mxn}$ is defined by $c^-_{ij} := I^-(p_i, t_j), \forall p_i \in P, \forall t_j \in T,$

$C^+ = (c^+_{ij}) \in N_0^{mxn}$ is defined by $c^+_{ij} := I^+(p_i, t_j), \forall p_i \in P, \forall t_j \in T,$

and the incidence matrix of PN is defined as $C := C^+ - C^-$.

A transition $t_i \in T$ is enabled in a marking M, iff $M \geq C^-e_i$, where e_i is the i-th unit vector. If an enabled transition $t_i \in T$ fires in marking M, the successor marking M' is given by

$M' = M + C\underline{e}_i$

Considering $\sigma = t_{k1} \ldots t_{kj}$, $j \in N_0$ a firing sequence of transitions. Marking M_j can be calculated as follows. For each reachable marking the following equation holds:

$M_i = M_{i-1} + C\underline{e}_{ki}$, $i = 1, \ldots, j$

Substituting the right hand side of the equation for i = r into the equation for i = r + 1 yields:

$M_j = M_0 + C\sum^j_{i=1}\underline{e}_{ki}$

The vector $f := \sum^j_{i=1}\underline{e}_{ki}$ is called the firing vector. Since reaching a marking implies the existence of an appropriate firing sequence, the following theorem holds:

$\forall M \in R(PN, M_0) : \exists \underline{f} \in N_0^m : M = M_0 + C\underline{f}$ (*)

Multiplying both sides of (*) with $\underline{v} \in Z^n$ we get:

$\underline{v}^T M = \underline{v}^T M_0 + \underline{v}^T C\underline{f}, \forall M \in R(PN, M_0)$ (*)'

The place invariants, or P-invariants are defined as $\underline{v} \in Z^n$, $\underline{v} \neq 0$ where $\underline{v}^T C = 0$. The P-invariants estabilish a condition on all reachable markings in equation (*)'.

Definition: PN is covered by positive P-invariants iff $\forall p_i \in P$: $\exists$ P-invariant $\underline{v} \in Z^n$ with $\underline{v} \geq 0$ and $v_i > 0$.

The Invariant Analysis Module makes use of the following theorem to determine the boundedness of a PN:

Theorem: PN is covered by positive P-invariants $\Rightarrow$ PN is bounded

Transition invariants (T-invariants) are the firing vectors that enable a marking M to be reached again following a sequence of firings. Consider (*), replacing M with M_0:

$M_0 = M_0 + C\underline{f}$

Then $\underline{w} \in Z^m$, $w \neq 0$, is a T-invariant iff $C\underline{w} = 0$.

Definition: PN is covered by positive T-invariants iff $\forall t_i \in T$: $\exists$ T-invariant $\underline{v} \in Z^m$ with $\underline{v} \geq 0$ and $v_i > 0$.

The Invariant Analysis Module makes use of the converse to the following theorem to determine that a net is not bounded and not live:

Theorem: PN is bounded and live $\Rightarrow$ PN is covered by positive T-invariants.

Proofs concerning the above results and a more detailed discussion of Invariant Analysis can be found in Stochastic Petri Nets – An Introduction to the Theory[15].

Thus, the analysis can be reduced to the computation of the invariants of the matrix C, and so solving the matrix equation:

$A\underline{x} = \underline{0}, \underline{x} \neq \underline{0}$ (*)''

Since $(\underline{v}^T C)^T = C^T(\underline{v}^T)^T = C^T\underline{v}$ the equation $C^T\underline{v} = \underline{0}$ yields the P-invariants and the equation $C\underline{v} = 0$ yields the T-invariants.

Different methods of computation were considered[16] and a decision was made to implement the D'Anna & Trigila algorithm[17] (see Appendix C) for calculating invariants because of its reduced complexity and the ease with which it could be converted to Java code. When applied to matrix A the algorithm produces a minimal generating set of all invariants of A.

Design and Implementation of the module

The module was thought to have three main functions:

- The reading in of the Petri net to be analysed (represented in PNML by the JDOM object) and the initialisation of the incidence matrix
- The analysis of the incidence matrix to determine the properties of the net
- The display of the analysis on the screen, in a separate window from the main application

The module implements the application's standard interface for dynamic loading. The JDOM object representing the Petri net is passed by reference to the module interface, which is implemented in the class InvariantAnalysis. The parsing of the Petri net is done by the class `JDOMreader`, an instance of which is created in the module interface method `runModule()`. The algorithm is then applied to the resulting incidence matrix and its transpose to compute the place and transition invariants. The support classes `Node`, `Int`, `Vector_of_ints`, `Vector_of_nodes` are used in the implementation of the algorithm. Finally, the class `DisplayAnalysis` displays the conclusions in a separate dialog box to the application.

§3.3.2 State Space Analysis Module

This module reads in data from the JDOM and analyses it by building a tree of all the possible markings, before outputting information regarding the characteristics of the Petri net. The data passed from the JDOM consists of:

- Number of Places
- Initial state of the Petri net
- Number of Transitions
- Weighting of each Arc.

The reachability tree is constructed by starting with the initial marking of the net. The directly reachable markings are then added to this as leaves and their directly reachable markings are in turn calculated. These markings are then added as further leaves and so on.The tree is built recursively, and the function proceeds to call itself until all possible markings have been visited. The criteria for stopping the firing of transitions on a particular path and moving on to the next one are:

- No transitions can be fired (deadlock has been reached)
- The marking is a repeat of a previous marking on the same path.
- The marking is not a repeat but is greater than a previous one on the same path, i.e. one or more places contain more tokens than the previous marking, whilst all the remaining places have the same number of tokens. The places with more tokens are assigned the symbol 'omega' as the number of tokens on the place can increase infinitely, if the cycle is repeated.

Once the tree has been constructed, there are a number of tests that can be carried out on it to determine various characteristics of the Petri net. Tests are carried out for boundedness, deadlock, safeness and liveness (see Figure 5 for sample output).

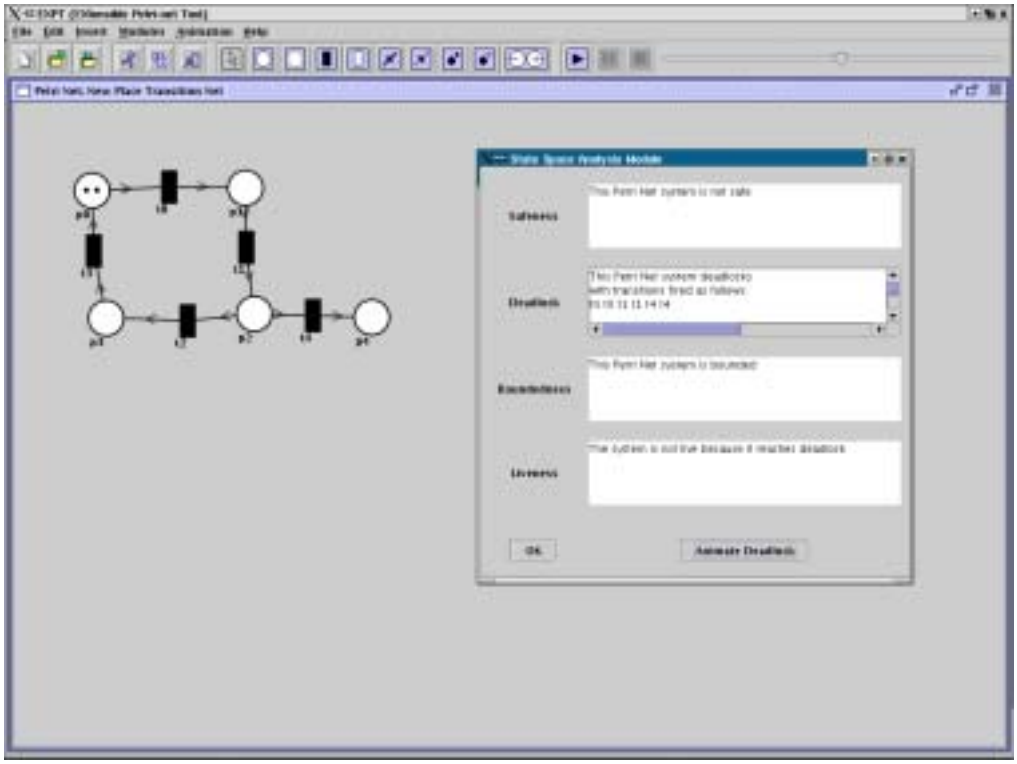


Figure 5

- **Boundedness** is a fairly simple test to implement, as it tests for the presence of omegas within the tree. If there are omegas present then the system is unbounded as there is no finite limit to the number of tokens that can appear on the place marked by the omega. This also means that there are an infinite number of different markings that can be visited.

- **Deadlock** occurs when it is no longer possible to fire any transitions and hence the system is stuck in its current position. If deadlock occurs, the module passes back to the editor a list of the transitions that are the shortest path to deadlock. A button on the display window of the results animates this path to deadlock.

- **Safeness** is a test to determine whether it is possible for more than 1 token to be on any place at any given time. If a place holds more than 1 token at any point in the tree then the system is not safe.

- **Liveness** is a test to check whether the system ever gets to a state where some transitions can no longer be fired. Previous attempts at analysing liveness have carried out this test by converting the tree into a coverability graph, and then testing the coverability graph for the property of strong connectedness and making sure all transitions appeared within each strongly connected cycle.

However the approach taken to liveness was innovative, using new methodolgy not previously used to tackle the problem, which removed the necessity of converting the tree into a coverability graph.

After building the tree, we can define as endpoints, any markings which are the last on any path of the tree. If we can show that all endpoints are live (there are no transitions which can no longer be reached from these markings), then we can conclude that all markings on the tree are live – as they can all reach an endpoint which is in turn live – and therefore that the Petri net as a whole is also live.

All endpoints are only endpoints for one of 3 reasons as stated above:

- **Deadlock** - If the marking is in deadlock, the system is obviously not live as no transitions can now be fired.
- **Unboundedness** - If the marking contains an omega, the system is unbounded and we don't carry out a check for liveness.
- **Repeated marking** - If the marking is a repeat of a previous marking on the same path, we only need to show that the previous marking, higher up the leaf is live for all endpoints in order to conclude that the system is live. The marking higher up the path which is a repeat of the end node also needs to be checked for liveness. The method for checking this node is detailed below in pseudocode.

```
public boolean node_is_live(int a) {
for (all markings i occurring after the node being checked) {
    if (i and a are on the same leaf) {
        if(i is live) {
        return true;
        }
        else if(i is repeated higher up the path and it is live){
        return true;
        }
        else if(i is repeated higher up the path && a doesn't equal i) {
            if(node_is_live(repeated node of i higher up the tree){
            return true;
            }
        }
    }
}
return false;
}
```

A version of the module which builds a smaller tree was also developed, but it requires more calculation to analyse the tree. This is achieved by stopping the construction of any path of the tree when it repeats any marking in the tree rather than only when it repeats a node on the same path. This can shrink the state space being analysed considerably, however this complicates the analysis of liveness significantly.

The analysis of GSPNs using the theories outlined above is made difficult by the fact that such nets possess two forms of transitions. As has been said, the firing of enabled

immediate transitions has priority over that of timed transitions, which leads to the existence of vanishing and tangible markings.

It is not possible to generalise about the attributes of a GSPN from analysis of its underlying Place-Transition net, except to say that if the Place-Transition net is bounded then so is the GSPN[15]. Due to these complications, the state space module is designed for the analysis of Place-Transition nets only, and treats all transitions as immediate irrespective of their attributes.

§3.3.3 Interface to DNAmaca Module

The DNAmaca module[21] was chosen to demonstrate that it is possible to extend our program and link it to new or existing tools that don't necessarily use the same data format or programming language. DNAmaca is a Markov chain analyser written by Dr. William Knottenbelt capable of generating performance analysis results for Generalised Stochastic Petri Nets (GSPNs).

The DNAmaca module automatically generates a description of the net in the model description format of DNAmaca. Additionally, the DNAmaca module automatically adds the performance measures desired, runs DNAmaca itself and finally presents the results back to the user.

An input file contains two sections: the first half describes the structure of the net (the model description) whilst the second details the performance measures. GSPN's are automatically changed into the corresponding DNAmaca module format.

To ensure that the results are always valid, any attempt to generate a model description automatically must deal with the overall effect of firing a transition on the state of the system. Given the general definition of a Place Transition net $PN = (P,T,I^-,I^+,M_0)$, it is possible to represent the forward and backward incidence functions as matrices as described in §3.3.1 above.

The incidence matrix C will just keep the overall effect. Once C has been calculated, it can be used directly in the creation of the DNAmaca input file.

§4 Validation

§4.1 Introduction

The aim of this chapter is to validate the design of EXPT in the light of the aims stated in §1.2. This will involve verifying the correctness of the analysis modules and an assessment of the features and extensibility of the tool.

§4.2 Analysis Modules

The performance of the analysis modules was validated with results produced by the DaNAMiCS tool[7]. The correctness of the Invariant Analysis Module was confirmed on a number of different Petri nets where the results matched those obtained using the DaNAMiCS tool (see Appendix for details). The State Space Analysis Module is limited by the memory available and as a result was only able to give results for some of the nets considered. Where results were obtained however, these were in line with those obtained by DaNAMiCS.

§4.3 Tool Extensibility

EXPT supports two levels of extensibility. This is achieved through support for nets represented in PNML and the ability to dynamically load analysis modules.

- PNML representation – the tool can load and save files containing Petri net representation in PNML format. This was verified by the loading and analysis of sample files from the Petri Net Markup Language website[3].

- Dynamic Loading of Analysis Modules – EXPT can load analysis modules at runtime to perform analysis on a Petri net. This was achieved successfully through the design of the Invariant and State Space Analysis modules. The module interface provides further extensibility through the ModuleBridge class that allows modules to invoke the animator to demonstrate a particular feature of the net. In addition, this allows the module to modify the net (by modifying the underlying JDOM representation) following the analysis.

- The power of dynamic loading of modules was further demonstrated through the development of an interface to DNAMACA, a Markov chain analyser written in C++, which was implemented as an analysis module written in Java. This enables the tool to output performance statistics of a Generalised Stochastic Petri Net.

§4.4 Tool Features

The evaluation of EXPT's features was carried out through a comparison with a number of other tools. A brief summary of these is listed below. For further information and references see the Petri Net World website[2].

- Petri Net Kernel – A Petri net design and build environment implemented in Java and Python. It supports PNML and its chief aim is to provide designers of Petri net tools an editor with an extensive range of features. Developed at the Humboldt University in Berlin[18].

- Renew – A Petri net tool implemented in Java. It offers a number of editing features and a simulation engine. Supports symmetric multi-processor architectures[19].

- Medusa – a Petri net tool supporting runtime loading of analysis modules, offering similar extensibility features to EXPT. Written in Java[13].

- DaNAMiCS – Petri net tool developed at the University of Cape Town. It is written in C++ and offers a wide range of analysis features including Invariant analysis, State Space analysis, Simulation and implements the DNAmaca Markov chain analyser[20].

EXPT was seen to be the most intuitive graphical user interface for those familiar with the standard Windows GUI (assumed to be most users). The DaNAMiCS and Medusa interfaces are not hard to get to grips with though we found that Renew was quite tough going despite numerous options for Petri net design. A useful feature provided by EXPT is the right-click on an element enabling editing of properties as well as manual animation in case of transitions. We have not seen this implemented in other tools. EXPT offers standard functionality such as selecting, cutting, copying and pasting of elements and selections using the selection box that is also present in DaNAMiCS and Renew. Both DaNAMiCS and EXPT support subnets while Renew supports different abstraction concepts based on a class hierarchy in Java.

DaNAMiCS offers the most extensive analysis features though the design of Medusa and EXPT allow for dynamic loading of user designed analysis modules. Unlike Medusa, the module interface of EXPT allows for information to be passed back to the application from the module. In addition, EXPT can load multiple modules at the same time.

§4.5 Concluding Remarks

In this section, the report has attempted to validate the finished product and determine whether the original product specification was met. In view of the strong slant on extensibility, the successful validation of dynamic loading of modules and the support of the latest PNML standards is very pleasing.

Having successfully tested the analysis modules for correctness, the tool offers two types of net analysis as well as an interface to a Markov chain analyser for analysis of GSPNs. Finally, the interface to the user offers a wealth of features, many of which have been found to be an improvement on existing tools.

§5 Conclusion

§5.1 Concluding Remarks on Project

In general the final product satisfies the minimum requirements outlined in §2.1. In addition we have completed our proposed extensions including the ability to cope with new types of nets, linking analysis results back from a module, automated animation and an interface to the Markov chain analyser DNAmaca. With reference to the use of PNML, we have also adopted the new modular-PNML. The added difficulty of incorporating this (and its suggested graphical representation) was clearly outweighed by the obvious extensibility benefits. It should be noted however, that given the complexity of subnets, an instance of a subnet has been restricted to having only one import and one export place. Animation of subnets has also been left as a future extension.

§5.2 Extensions and Improvements for the Future

Subnets

- **Import/Export/Reference transitions** were not implemented due to time constraints.
- **Increasing the limit of one import and one export place per subnet instance**, again due to time constraints.
- **Loading of subnets** could be extended to support distributed storage, so that (for example) a subnet could be loaded from a website which hosted a library of subnets. The current PNML definition allows for this possibility.

Modules

- **Simulator module** – a module which simulates the animation of a net, but does **not** display it, so that statistics may be collected on the behaviour of the net and compared with analytical results.
- **Other possible modules** - a module that converts between different net types, **expands or compresses the markup by adding or removing default-valued** elements, or calculates a graphical layout from Petri net data that may be lacking it, corrupted, or simply messy in appearance.

PNML support

- **Validation** of the PNML is possible given appropriate schemas or DTDs.
- **Different Namespaces** are not distinguished between – for example, the **Properties** Box could be made configurable so that elements in certain namespaces were ignored, and others were made visible.
- **Pages are** included within the PNML definition, but not supported in the current implementation of the editor.
- **GSPNet** is not an official definition – a PNTD for Generalised Stochastic Petri Nets does not yet exist.
- **ptNetArcTyped** is an official definition, but specific functionality is only implemented for arcs of type “inhibitor” (and by default, “normal”) – “read” and “reset” currently behave as “normal” arcs. In addition, a visual indication of arc type may be considered, preferably following any existing conventions.

Animation

- **GSPNAnimator with timed transitions**: when choosing between timed **transitions**, their firing rates should be taken into account using an exponential probability function[15]. The firing rates are currently treated as a simple priority, due to time constraints.
- **Subnet animation,** and interpretation of the flattened ids that are passed back from Modules.
- **Display optimisation** – the current implementation produces difficulties in refreshing the display when the animation is run at full speed, and/or with large nets. A likely solution would be to implement the animation as a separate thread, allowing the Swing event-dispatching thread to keep up with the display.

Editing

- **User-oriented convenience features**, such as the automatic prediction of the next step while adding new elements to a net (eg as in Renew[19]).
- **Cut and paste functionality** – modification to automatically deal with different net types, and subnets.

§6 Sources

[1] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, 1962

[2] Petri Net World, www.daimi.au.dk/PetriNets/

[3] Petri Net Markup Language
http://www.informatik.hu-berlin.de/top/pnml/

[4] Ekkart Kindler, Michael Weber, A Universal Module Concept for Petri Nets. An Implementation-Oriented Approach, June 2001, http://www.informatik.hu-berlin.de/top/pnml/

[5] Matthias Jüngel, Ekkart Kindler, Michael Weber, The Petri Net Markup Language, August 2000, http://www.informatik.hu-berlin.de/top/pnml/

[6] Matthias Jüngel, Ekkart Kindler, Michael Weber, Towards a Generic Interchange Format for Petri Nets, April 2000, http://www.informatik.hu-berlin.de/top/pnml/

[7] Development of Chemical Markup Language (CML) as a System for Handling Complex Chemical Content', Peter Murray-Rust, Henry S. Rzepa and Michael Wright, New J. Chem., 2001, 25, 4, 618-634.
http://www.rsc.org/CFmuscat/intermediate.cfm?FURL=/ej/NJ/2001/B008780G.PDF

[8] W3C - Extensible Markup Language (XML)
http://www.w3.org/XML/

[9] Document Object Model (DOM) Requirements, W3C Working Draft 19 April 2001, http://www.w3.org/TR/DOM-Requirements

[10] Easy Java/XML integration with JDOM, Jason Hunter and Brett McLaughlin, http://www.javaworld.com/javaworld/jw-05-2000/jw-0518-jdom.html

[11] Understanding XML and the Java XML APIs,
http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/overview/index.html

[12] Java Specification Requests, JSR 102 - JDOM 1.0, http://jcp.org/jsr/detail/102.jsp

[13] Nicholas J. Dingle, Production of the Extensible Petri Net Editor/Animator, “Medusa”, Imperial College, 2001

[14] Mark Wass, Predator – A Hierarchical Petri Net Editor,
http://www.mark.wass.com/downloads/dissertation.pdf, Imperial College, 2001

[15] F. Bause & P.S. Krizinger, *Stochastic Petri Nets – An Introduction to the Theory*, 1995

[16] M.B. Powell, *Linear Algebra*, Mathematical Institute University of Oxford, 1995

[17] Mario D'Anna & Sebastiano Trigila, *Concurrent system analysis using Petri nets: an optimised algorithm for finding net invariants*, Computer Communications vol. 11 no. 4, August 1988

[18] Petri Net Kernel, www.informatik.hu-berlin.de/top/pnk/index.html

[19] Renew Petri Net Tool, www.renew.de/

[20] DaNAMiCS - a Petri Net Editor, www.cs.uct.ac.za/Research/DNA/DaNAMiCS/

[21] DNAmaca – A Markov Chain Analyser, Dr William Knottenbelt, www.cs.uct.ac.za/Research/DNA/DaNAMiCS/prop/node4.html