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

Predator – A Hierarchical Petri Net Editor

by

Wass. M.

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

September 2001

Abstract

Petri nets are a widely used modelling technique and a number of Petri net editor tools exist to design Petri nets. However current tools provide little support for hierarchical Petri nets and all analysis features are built into the tools. A Petri net editor was designed and implemented. It provides improved support for hierarchical Petri nets, allowing complex Petri nets to be simplified into smaller subnets. The Petri net editor also incorporates an open architecture enabling the dynamic loading of separately implemented analysis features. An invariant analysis module was implemented to demonstrate the operation of the open architecture.

Acknowledgements

Many thanks to William Knottenbelt my project supervisor who has been a continual source of support and guidance throughout the project.

Thanks also to the Java Tutorial, without which I don't think I would have got very far.

Contents

1 Introduction

Petri nets are popular modelling technique used in many disciplines including the design of concurrent systems, communication protocol design and analysis (Woodside & Li, 1991), and the modelling of manufacturing systems (Ciardo & Trivedi, 1993).

There are a number of Petri net editor tools available, offering a range of features from simple editing to complex simulations. Petri net editors provide only limited support for hierarchical Petri net design, where by a system is decomposed into a set of Petri nets rather than a single net. Such approaches simplify system design, because different components of a system can be designed and modelled individually.

Petri net properties such as liveness, safeness and boundedness, can be tested for all Petri nets. Petri net editor tools often provide built in analysis features to verify these properties. However these are often limited and may not provide the analysis required.

The aim of this project is to design and implement a Petri net editor that will address the current lack of support for hierarchical Petri nets and the limitations of analysis features in current editors. Support for hierarchical Petri nets requires a suitable graphical representation of hierarchical Petri nets and the ability to navigate around a hierarchical Petri net.

To address the limitations of Petri net analysis features offered by Petri net editors, the editor will incorporate an open architecture, allowing the dynamic loading of separately implemented analysis features. This will enable users to dynamically load analysis features as desired, even offering users the opportunity to implement their own analysis features.

Chapter two introduces the background of Petri nets, their properties and analysis techniques. The features offered by existing Petri nets editors are also investigated in this chapter. This is followed by the Design chapter, which discusses the Petri net editor's design issues. Implementation (Chapter 4) describes how the designed editor was implemented. The features of the implemented editor are then tested and discussed with a number of case studies in the Results (chapter 5) and Conclusion (Chapter 6). A user guide for Predator, the Petri net editor tool implemented is provided in chapter 9.

2 Background

This chapter begins by introducing the origin of Petri nets and their basic structure. Hierarchical Petri nets are then introduced before the general properties of Petri nets and analysis techniques to study them. The second part of the chapter considers the properties offered by existing Petri net Editing Tools and discusses the capabilities required by the Petri net Editor to be developed as part of this project.

2.1 Petri Nets

2.1.1 Development of Petri Nets

In 1962 Carl Petri invented Petri nets (http://www.informatik.uni-hamburg.de/TGI/mitarbeiter/profs/petri_eng.html). They have become a widely used modelling technique for systems including Concurrent, distributed and parallel. They have applications in other fields for example manufacturing, where they model flexible manufacturing systems (Ciardo & Trivedi 1993).

Petri nets are mathematical descriptions of systems. They consist of four basic elements: transitions, places, arcs and tokens. Transitions and places are described by sets. Arcs connect transitions and places, and are described by backwards and forwards incidence functions, often termed arc weights, which relate to the movement of tokens between places. Tokens are associated with places and the initial marking describes the initial number of tokens on each place. Markings, the number of tokens on each place, represent subsequent system states. Figure 1 displays the formal definition of a Petri net.

$PN = (P, T, I^-, I^+, M_0)$

P is a finite set of places

T is a finite set of transitions

$P \cap T = \emptyset$

I^- is the backwards incidence function

I^+ is the forwards incidence function

M_0 is the initial marking.

Figure 1. Formal Definition of Place Transition Petri nets

Transitions are the active elements of Petri nets, they can be enabled and once enabled can be fired. The backwards incidence function determines the number of tokens required on an input place before a transition is enabled. For example for an arc from a place p1 to a transition t1 with an arc weight of 3, three tokens are required on place p1 before the transition is enabled. A transition is enabled when this is satisfied for all input places. Upon firing, tokens on all the input places are destroyed and created on all the output places of the transition. The number of tokens destroyed is determined by the arc weight; likewise the number of tokens created on each place is determined by the forward incidence function. This basic Petri net structure is often referred to as a Place Transition Net (Bause & Kritzinger 1995).

The usability of Petri nets is extended by the ability to express them graphically as shown in Figure 2, which illustrates the readers-writers problem. Graphical display simplifies net design, as nets can be constructed using a graphical editor. Their graphical nature also enables Petri nets to be animated allowing visual observation of how systems function.

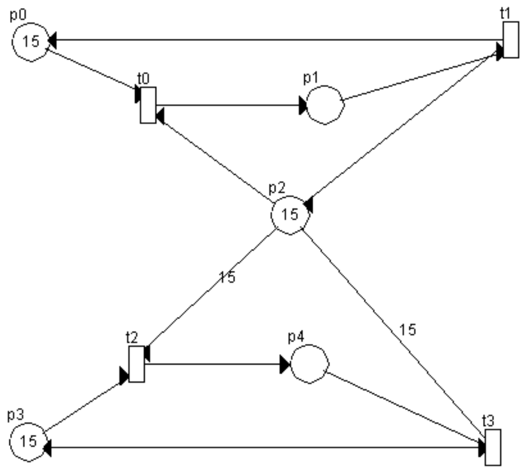


Figure 2. *The readers writers problem*. The readers-writers problem is a common problem associated with concurrent access to files and data. For example a shared database can be accessed by Readers, who obtain data from the database and Writers who both obtain and write data to the database. Multiple readers can access the database concurrently but writers must have exclusive access. Place p0 specifies the number of Readers not reading, while p1 represents Readers that are reading. Transitions t0 and t1 move readers between these states. Similarly place p3 represents the number of Writers not writing, p4 the number of Writers writing and transitions t2 and t3, control movement between these two states. Place p2 represents a semaphore that controls access to the file/database.

There are now many variations of Petri nets they are mostly extensions of Place Transition Nets. Variations include coloured Petri nets, which introduce different coloured tokens, fluid stochastic Petri nets, which instead of tokens associate a level of liquid with places and condition event nets, in which places represent conditions, the presence of a token satisfying the condition. Timed Petri nets introduce the concept of time.

Place transition nets lack the concept of time; it is not known at what time a transition will fire. To perform quantitative analyses on the performance of a Petri net, timing is required as part of the model. Time can either be introduced into a Petri net by specifying sojourn times of tokens on places or associating firing delays with enabled transitions. For the former, tokens generated on places are unavailable for a set time. In the latter case, enabled transitions wait a firing delay before firing. Such nets are Timed Transitions Petri nets.

Stochastic (Molloy, 1982) and Generalised Stochastic (Arjmor *et al.*, 1984) Petri nets are widely used Timed Transition Petri nets. Generalised Stochastic Petri nets further split transitions into two subsets. Timed transitions fire after random exponential firing delays, while immediate transitions fire instantly once enabled. A weight is associated with immediate transitions, it determines the probability of firing if multiple immediate transitions are enabled in the same state. (Bause & Kritzinger 1995).

2.1.2 Hierarchical Petri Nets

Since Petri nets were not originally conceived of as hierarchical structures, Hierarchical Petri nets are not widely supported by Petri net Tools.

By contrast, Process Algebras, such as Performance Evaluation Process Algebra (Hillston, 1996, PEPA homepage: http://www.dcs.ed.ac.uk/pepa) are mathematical calculi for modelling concurrent systems in a compositional way. Compositional approaches allow the decomposition of systems into smaller components, making it easier to model complex systems (and in some cases also easier to analyse if the compositional nature of the system can be exploited).

Providing support for hierarchical structures in Petri nets not only makes the design process cleaner and simple (and more similar to approaches of classical software engineering methodologies), but also allows for compositional analysis techniques from the Process Algebra community to be applied to Petri nets.

Process algebras are textual and based on process calculi, making them relatively difficult to define, whereas Petri nets are easily expressed in a simple graphical way; thus the application of compositional approaches to Petri nets could be advantageous to system designers.

2.1.3 Properties of Petri Nets

Petri nets exhibit properties, which can be verified. Liveness, safeness and boundedness are three important properties of Petri nets.

A net is live, if it is not possible to reach a marking from which a transition will never again be enabled. This means that whatever state or marking the system is in, there will always be a firing sequence such that any transition can be fired from that state. Live systems are free of deadlock and livelock. Livelock occurs when a system is stuck in a subset of states that does not include all transition firings, some functionality of the system can be permanently disabled. Liveness is desirable in most systems because deadlock is avoided and the complete system is always accessible.

A place is safe if in all possible markings, the number of tokens on the place is never greater than one. A Petri net is safe if all places in the net are safe. Safeness is of importance for places that represent conditions. There are two possible states; the condition is either satisfied (token on place) or not satisfied (place empty). It does not make sense for a condition to have more than one token.

Boundedness generalises safeness. A place is k-bounded if in all possible markings, the number of tokens on the place is never greater than k. Further a Petri net is k-bounded if in all possible markings, and for all places, the number of tokens on a place is not greater than k. Boundedness determines that in all markings each place contains a finite number of tokens. Boundedness is important to prevent buffer overflows in systems.

2.1.4 Petri Net Analysis

Petri nets analysis ranges from the simple animation of firing sequences to complex performance analysis. This section briefly describes some of the more common analysis techniques. As an analysis module to perform Invariant analysis is to form part of the project, it is described in more detail.

2.1.4.1 Animation – Token Game Animation

Token Game animation provides the simplest analysis of Petri nets. A Petri net starts in its initial state with all enabled transitions indicated for the user to select one to fire. When a transition is fired tokens are moved accordingly and the process is repeated, another enabled transition can be selected. This type of animation does not provide any definite information on the systems but can be an effective way to view transition firing sequences.

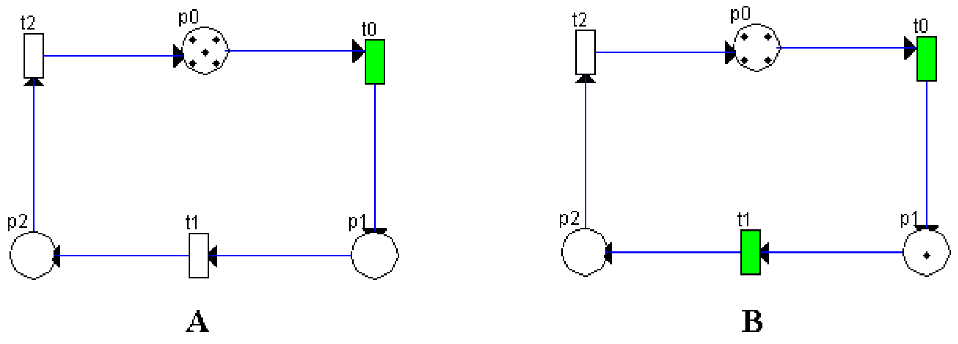


Figure 3. *Example of Token Game Animation*. A) Place p0 has five tokens, so transition t0 is the only enabled transition, which is indicated by its colour. t0 is fired, removing one token from p0 and adding one to p1 as shown in B.
B) Place p1 now also has a token, so both transition t0 and t1 are enabled as indicated.

2.1.4.2 Correctness Analysis

Correctness analysis techniques verify Petri net properties liveness, safeness and boundedness. Invariant analysis and reachability tree analysis provide techniques for verification of these properties.

P invariants

A	p0 + p1 = 15
B	p1 + p2 + 15p4 = 15
C	p3 + p4 = 15

P invariant Explanations

A states that the sum of the number of token on places p0 and p1 is equal
to fifteen in all reachable markings of the net. i.e. the total number of readers is constant

B Similarly states that sum of tokens on p1,p2 and 15 times p4 is always equal to 15.

C The sum of tokens on p3 and p4 is always equal to 15. i.e. The total number of writers is constant.

T invariants

D	t0,t1
E	t2,t3

T invariant Explanations

D The firing sequence t0,t1 returns the system to the marking it was in before the firing sequence. In this case moving a Reader from the state of not reading to reading back to not reading or from reading, to not reading back to reading.

E This sequence performs the equivalent as D but for a writer.

Boundedness, Liveness Properties of the system

Bound – the Petri Net is covered by P invariants because all of the places in the net appear in at least one P invariant. This means that the net is bounded.

Boundeded & Live, - A bounded and live net is covered by T invariants. This net is covered by T invariants, however this doesn’t infer that the net is bounded & live.

Figure 4. *The P and T invariants of the Reader-Writers Petri net.* This figure gives the P and T invariants of the Readers-Writers Petri net (Figure 2). An explanation of each invariant in relation to the Readers-Writer example is also given.

Reachability analysis generates the reachability set (or tree), of a Petri net. The set starts at the initial marking, with a new marking created for each transition. This is repeated for all new markings until all possible marking have been covered. It is then possible to determine if the net is bounded. If so it is also possible to determine if the net is live and to identify home states.

Invariant analysis also identifies if a net is bounded and live. This is done by calculating both P (Place) and T (Transition) invariants. P-invariants occur where for all possible markings the sum of the marking of a group of the places remains constant. Transition invariants identify order independent transitions firing sequences that leave a net's marking unchanged (i.e. From a marking M a firing sequence is followed to return the net to marking M).

To perform Invariant analysis the incidence functions of a Petri net are converted to incidence matrices. The backwards incidence function is converted to the backward incidence matrix (C^-) and the forwards incidence function to the forward incidence matrix (C^+). The incidence matrix C is the result of subtracting C^- from C^+ ($C = C^+ - C^-$). Transition firing can then be expressed using the incidence matrices. It is then possible to express firing of transitions with the incidence matrices as described below.

From a marking M a firing sequence f is followed, where f is the vector of the number of times each transition is fired in the firing sequence, the resultant marking M` can be described as:

$$M` = M + C\,f$$

Thus if M` = M, (i.e. the sequence returns the net to the same marking)
then $C\,f = 0$, thus f is a T invariant which leaves the marking M unchanged.

If equation 1 is multiplied by $v^T \in Z^n$ gives equation 2 below:

$$v^T M' = v^T M + v^T C\,f$$

In this case, if $v^T C = 0$ then $v^T M' = v^T M$, $\forall M \in R(PN, M)$ where R(PN,M) is the set of all possible reachable markings given an initial marking M. v is a P invariant, P invariants can be found be solving $v^T C = 0$.

From P and T invariants it is possible to determine properties a Petri net, such as boundedness and liveness. A Petri net is bound if it is covered by positive P-invariants. This means that all the places of the Petri net must appear in at least one of the p invariants. The Readers-Writers example Petri net (Figure 3) is bounded as described in figure 4.

A bounded and live Petri net is covered by T invariants. This does not infer that a net covered by T invariants is bounded and live, however it can be concluded that a net that is not covered by T invariants is not bounded & live. The net in the readers-writers example is covered by T invariants. Other analysis techniques such as reachability analysis must be performed to identify if the Petri net is bounded and live.

2.1.4.3 Performance Analysis

Performance analysis is possible for Petri nets that incorporate time. Performance analysis can provide statistics such as the mean number of tokens on a place, the probability of being in a subset of markings, the probability of a transition firing given that it is enabled and the throughput of a transition.

Markov chains can be obtained from a Petri net's reachability graph. This enables analytical performance statistics to be obtained. Simulation can also be used for Performance analysis, however results are not exact and many simulations are required to have a high confidence in the results.

2.2 Petri Net Editors

There are approximately forty Petri net Tools listed in the Petri net Tools Data base (http://www.daimi.au.dk/PetriNets/tools/db.html). The functionality provided varies from the most basic of editors to those providing sophisticated analysis capabilities. A sample of the editors available have been obtained and investigated to identify the capabilities common to Petri net Editors. The editors studied include DaNAMiCs, HPSim, INA, VisObjNet144 and the PetriTool The following sections discussion the features provided in these and other Petri net tools. The sources of the other editors referenced are provided in the tools references section of chapter 7.

2.2.1 Basic Features

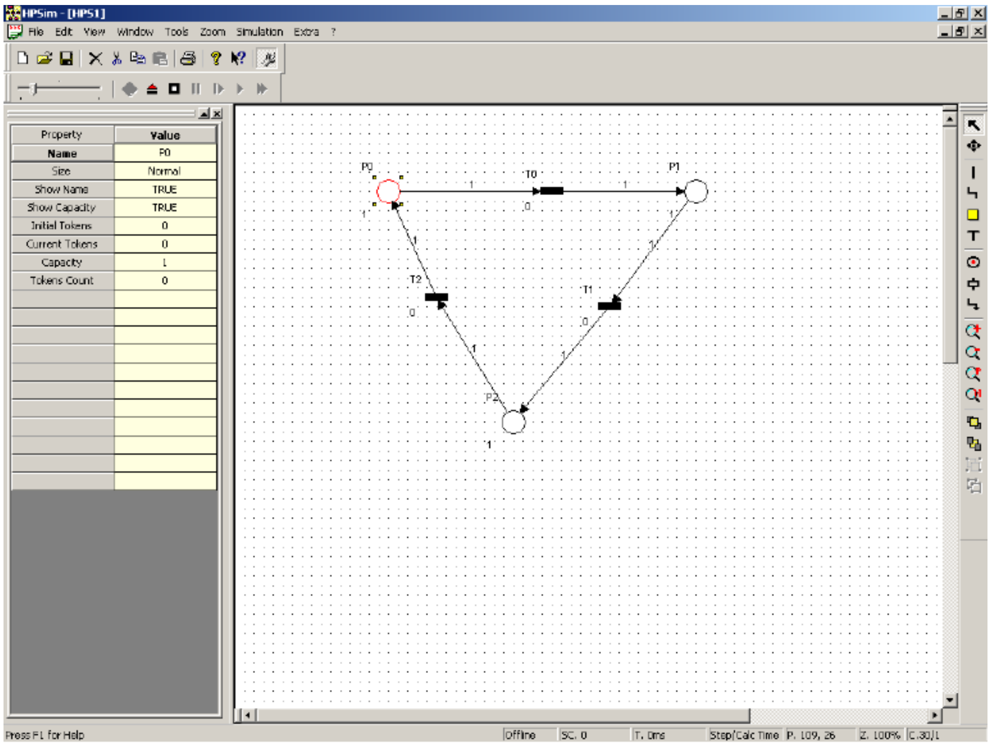
2.2.1.1 Platforms

Of all the editors available the majority run only on UNIX operating systems, a few operate solely on Windows, while fewer still run on both Windows and Linux operating systems. Tools that are portable are mainly implemented in Java. Portability of the tool is desirable because few tools run on multiple operating systems.

2.2.1.2 Editing capabilities

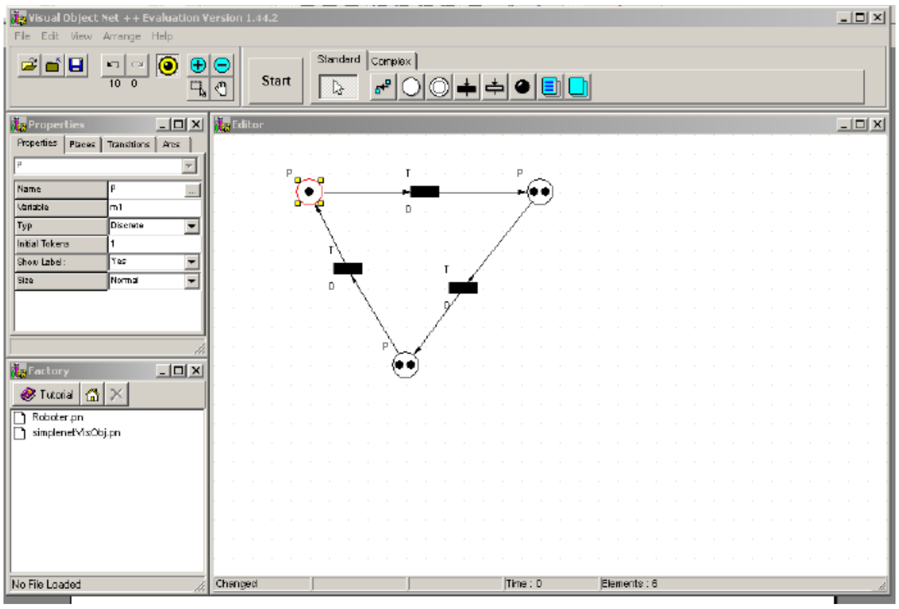
The majority of tools provide a graphical editor for Petri net design, (user interfaces displayed in figures 5 and 6). A few of the tools such as INA and the Model Checking Kit are text based. These tools provide greater support for Petri net analysis rather than design.

The graphical editors investigated all allowed the user to add places, transitions, arcs and tokens to Petri nets and provided features to edit and remove these components. Some provided further options such multiple node arcs, zooming, and printing.



Property	Value
Name	P0
Size	Normal
Show Name	TRUE
Show Capacity	TRUE
Initial Tokens	0
Current Tokens	0
Capacity	1
Tokens Count	0

Figure 5. *A screenshot of the HPSim Petri net editor tool*. HPSim is a Windows based tool providing support for Place Transition nets and Stochastic Petri nets. It offers token game animation and simple performance analysis but no correctness analysis features. Properties of Petri net components can easily be edited using the table to the left of the user interface.



Name	P
Variable	m1
Typ	Discrete
Initial Tokens	1
Show Label:	Yes
Size	Normal

Figure 6. *VisObjNet144 Petri net Editor Tool*. VisObjNet provides similar functions to HPSim, Place Transition and Stochastic Petri nets are supported. Token Game animation, simulation and performance analysis are all incorporated in the tool. Like HPSim Petri net component properties can be edited using a table.

2.2.1.3 Petri Nets Supported

All the editors available support Place Transition nets, which is expected as most Petri net types are extensions of Place Transition nets. Some tools are designed for particular net types, for example CPN which supports coloured Petri nets. Due to their extensive use for performance analysis Stochastic Petri nets are widely supported; at least fifteen tools support them.

2.2.2 Hierarchical Petri Nets

A few Petri net tools provide support for Subnets. Investigation of DaNAMiCS identified that hierarchical models could be designed via the introduction of subnets. Subnets added have to use an existing file and appear on the screen as a blank box (figure 7). Interaction points can easily be chosen but it is difficult to specify which interaction point arcs placed into the subnet should be connected to. To move between levels the respective files for each level of the net have to be opened separately, making it difficult to switch between them and identify how the nets interact (figure 7). It was not possible to investigate any other tools that support Hierarchical Petri nets, however THORN/DE (Schof, Sonnenschein & Wieting, 1995) provides hierarchical Petri net support similar to that provided by DaNAMiCS.

The tools investigated do not provide a suitable graphical representation of hierarchical Petri nets. A representation that explicitly illustrates interaction points and how they are connected to higher levels is required to simplify hierarchical Petri net design. Such a representation should also allow simple switching between different levels of the Petri net without the requirement of opening the files associated with each subnet and editing them independently of the other levels of the net. Provision of such a representation is a central aim of this project.

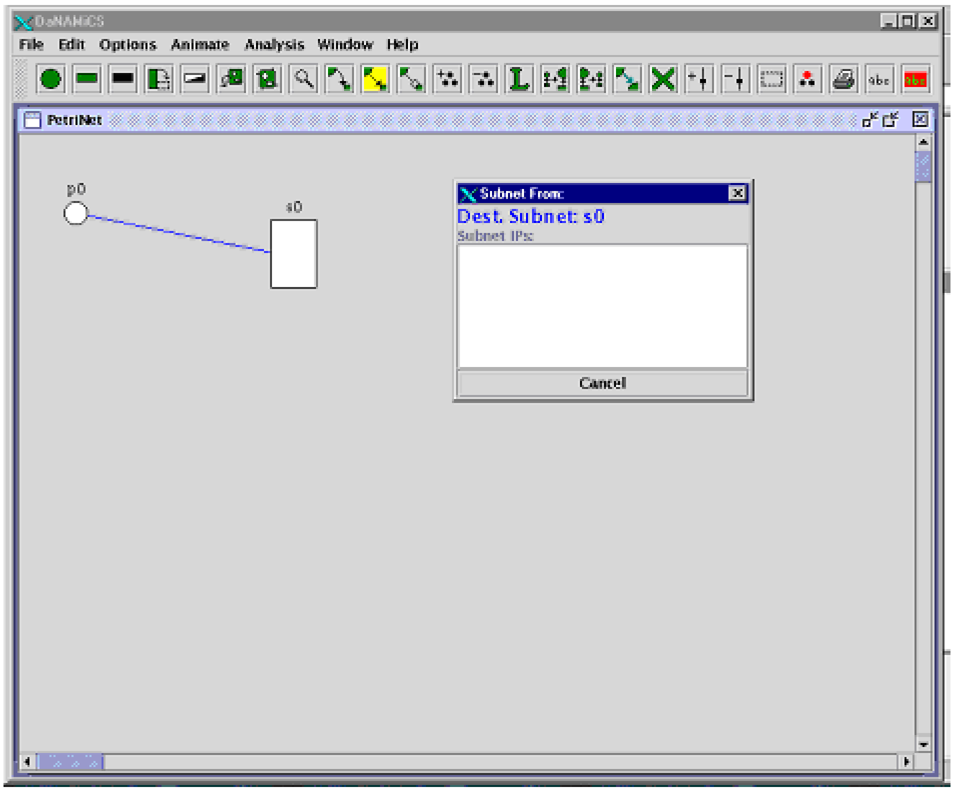


Figure 7. *A DaNAMiCS hierarchical Petri net*. DaNAMiCS provides simple support for Hierarchical Petri nets. In this example a subnet has been added to a Petri net, it is displayed as a white filled box. Arcs can be placed between components of the Petri net and a subnet however it is not clear how to do this. Upon connecting an arc from the place to the subnet shown above, a dialog (shown in figure) appears to select the component of the subnet to connect the arc to. It is not possible to visualise the interaction between the two levels of the Petri net.

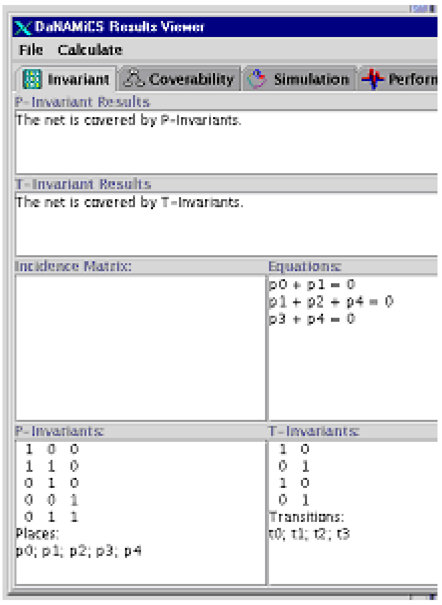
2.2.3 Analysis

There is much variation in the analysis options provided by Petri net editor tools. However, in all cases these analysis features are built into the tools. Users are limited by the analysis features provided in the tool they use. It may be necessary to use multiple tools to perform all the analyses required, this can be problematic if file formats are not interchangeable (see 2.2.4). As analysis features are built in it is not possible for users to easily specify their own analysis methods. This may be desirable if unusual systems are being modelled or a poorly supported Petri net type is being used.

An editor incorporating an open architecture allowing analysis modules (that implement a simple interface) to be dynamically loaded would provide a solution to these problems. Users can load modules performing the analysis they require and could even implement their own modules to perform specific analyses. The development of such an open architecture is an important aim of this project. The aim is that modules once dynamically loaded will add items to a menu of the user interface to run their analysis features.

2.2.3.1 Invariant Analysis

Invariant analysis is present in a few of the tools available. The output of such modules is very similar, for example DaNAMiCs, displays P-invariants, T-invariants, invariant equations and the incidence matrix for the analysed Petri net as shown in the figure 8. It also reports if both P and T invariants cover the net, indicating if the net is bounded and live and bounded. The display of P and T invariants are left in a matrix form from which the invariants must be interpreted. It would be clearer to express the P and T invariants in a format similar to the P invariant equations (see figure 8).



```
P-Invariant Results
The net is covered by P-Invariants.

T-Invariant Results
The net is covered by T-Invariants.

Incidence Matrix:                 Equations:
                                  p0 + p1 = 0
                                  p1 + p2 + p4 = 0
                                  p3 + p4 = 0

P-Invariants:                     T-Invariants:
 1  0  0                           1  0
 1  1  0                           0  1
 0  1  0                           1  0
 0  0  1                           0  1
 0  1  1                          Transitions:
Places:                           t0; t1; t2; t3
p0; p1; p2; p3; p4
```

Figure 8. *DaNAMiCS Invariant Analysis Output for the Readers-Writers Petri net.* This figure DaNAMiCS display of invariant analysis results. P and T invariants, are displayed in the bottom two areas of the dialog, they are displayed in matrix formed and need interpretation. The P invariant equations are displayed more clearly.

2.2.4 File Formats

Petri net tools tend to use their own file format. This is very restrictive if users wish to port Petri nets and use them in different tools to make use of different analysis and simulation capabilities. There have been some attempts by the Petri net community to develop a general interchange file format (Bastide, Billington, Kindler, Kordon & Mortensen 2000).

A number of groups have proposed possible alternate file formats, all of which are XML (extensible markup language) based. The use of XML should make file formats simple to parse particularly as some languages (Java for example) provide support for XML parsing. Most of the proposed formats attempt to separate basic net components such as places and transitions, from Petri net type specific elements such as inhibitor arcs.

The Petri Net Markup Language (Jungel, Kindler & Weber 2000) is one of the proposed file formats. A general section contains all the Petri net type independent data (i.e. places, transitions and arcs), while a separate Petri net Type Definition specifies specific details for different Petri net types. Other proposals include, the use of style sheets to separate style and content (Mailund & Mortensen 2000) and a two level data definition language (Valente & Gribaudo 2000).

A generic file format has yet to be chosen, so this project will aim to use a file format based upon the current proposals but extended to provide support for Generalised Stochastic and hierarchical Petri nets. This will ensure the file format is close to any interchange format eventually chosen.

3 Design

The first section of this chapter provides an outline specification summarising the aims and features intended for the Petri net editor. Subsequent sections discuss programming language choice, object oriented class structures, open architecture design and file format design.

3.1 Outline Specification

The two previous chapters have stated the aims of this project and the features present in Petri net Editors. These have been combined to provide an outline specification of the Petri net editor tool, that this project aims to develop.

3.1.1 General Properties

The tool should be portable. Many existing tools are operating system specific – in fact most are limited to specific brands of UNIX (e.g. Linux, Solaris), so it would be beneficial to operate on a wide range of Windows and UNIX platforms.

3.1.2 Editor Properties

The tool should provide the basic features necessary to design and edit a Petri net. Further features such as zooming and printing should be considered optional, and could be added at a later date. Support for hierarchical Petri nets (3.1.3) and an open architecture (3.1.4) are of greater importance to the project. Places, transitions and arcs should be able to have their properties such as token number, firing rate and weights modified. It should be possible to change the position of components on the net and to have the ability to delete components.

3.1.3 Hierarchical Petri Net Properties

The tool should support hierarchical Petri nets such that interactions between different levels are explicit, and can be switched between effectively. For example clicking on a subnet could cause it to be loaded and displayed by the editor. Interaction points could be identified by painting them a different colour and by displaying them on the outside of the subnet, to show how the Petri net interacts with subnets.

When subnets are introduced users should be able to select what Petri net the subnet contains. There should further be the choice to save the subnet using the already selected file or to save it under a different name specifically for use in the current design.

3.1.4 Open Architecture

The tool should provide an open architecture for the dynamic loading of analysis modules. The interface required for such modules should be simple, for example using a commonly named function to execute and run the module.

An analysis module, for invariant analysis will be implemented to illustrate the capabilities of the open architecture. The invariant analysis module should provide data on both P and T invariants and the boundedness and liveness of Petri nets, similar to that provided by DaNAMiCS.

3.1.5 File Format

A file format closely following proposals for an interchangeable file format (see 2.2.4) should be designed. It should support Hierarchical Petri nets, and should enable tools that use this file format but do not support Hierarchical Petri nets to parse it correctly as a single level Petri net.

3.2 Java Programming Language

For the implementation of an object-oriented tool, the two main choices of programming language are C++ and Java. Java is more suitable for this project. Unlike a C++ implementation a Java implementation will be portable operating on multiple platforms. Java also provides Reflection (See section 4.4) a way to load Java classes into an executing program, which will be necessary for the design of the open architecture. Java also offers XML support, which will be useful for implementation regarding the file format. Java is the clear choice for the implementation and the following sections and chapters will assume this.

3.3 Class Design

This section discusses possible class structures for the Petri net Editor, starting with the basic classes required to represent a Petri net.

3.3.1 An Object Oriented Petri Net

How should the components of a Petri net be modelled as classes in a Petri net Editor? Places, transitions and arcs, are obvious choices for classes.

Likewise subnets should be another class. Figure 9 displays a class structure design representing a Petri net as a group of classes.

An abstract class, PetriComponent, forms a base class from which all other Petri net components, inherit. PetriComponent will provide data members such as position and name, and functions for adding and removing components from a Petri net. Tokens and arcs inherit directly from PetriComponent. Transitions, places and subnets on the other hand, share more features in common that with arcs so they inherit from SolidPetriComponent. For example these components have dimensions, and can be connected to arcs.

The classes Timed and Immediate transitions provide the two subsets of transitions present in Generalised Stochastic Petri nets. However their similarities are contained in Transition, which they both extend.

Tokens are described as a class, but they could also be an attribute of the Place class. They have been assigned a class because it will be simpler to manipulate token position, addition and removal via a separate class, rather than as part of the Place class.

3.3.2 Graphical User Interface Design

The graphical user interface requires all the features that any typical graphical user interface requires, such as menus and toolbars. The section briefly describes the elements that will make up the user interface.

The design or drawing area will form the largest area of the user interface, it is here that Petri nets will be created and edited. To perform functions in the design area one of the options such as transition or edit will be selected. These options will be displayed in a toolbar, preferably adjacent to the drawing area. A further toolbar will provide file opening and saving options.

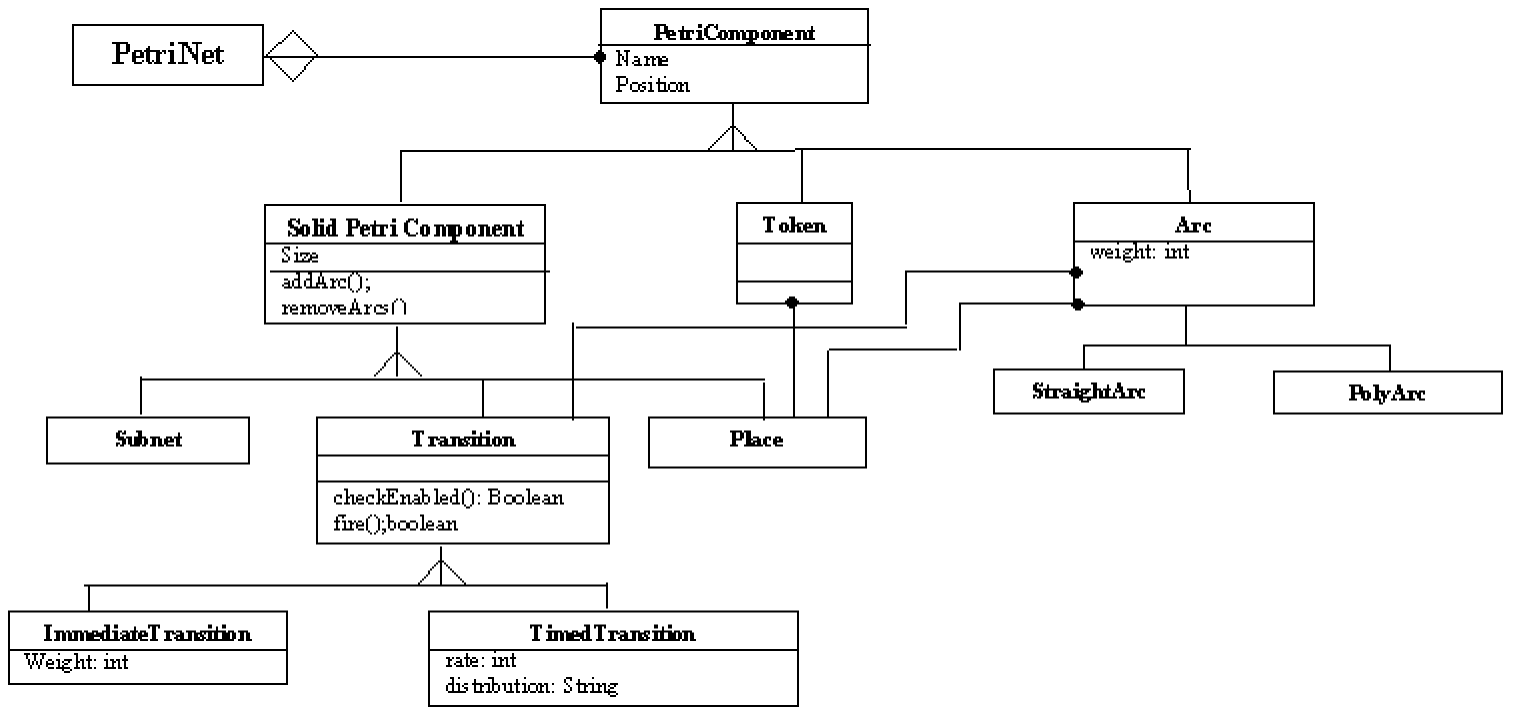


Figure 9. *Object Modelling Technique Class Diagram for a Petri Net.* A Petri Net consists of PetriComponents; places, transitions, arcs and tokens. For clarity only the main data members and functions have been displayed.

A status bar will be placed at the bottom of the user interface to provide information to the user.

Some editing features such as changing arc weights or transition firing rates cannot be carried out in the drawing area alone, because input needs to be obtained from the user. Dialogs are often used to obtain such information, however it would be inconvenient if a dialog appeared every time a small feature needed changing. It is simpler to place a table in the user interface that can display the required details and allow the user to edit them. This method is used effectively in other Tools, such as HPSim and VisObjNet (see figures 5 & 6).

The GUI will provide features to edit Petri nets, such as moving the position of components and modifying components' properties such as token number and weights. Figure 10 displays a possible class structure for the editing features. A different class encapsulates each of the different editing features. These classes all extend the same base class, EditClass. This class will provide functionality to interact with the design area. This design will simplify the addition of further editing features because they can be added by the implementation of a new class, extending EditClass.

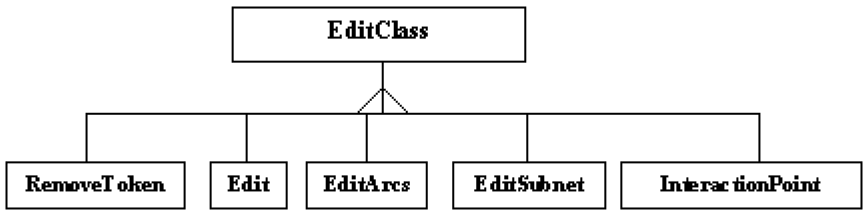


Figure 10. *The Editing Classes of a Petri Net Editor*. This OMT diagram provides a possible class structure for classes implementing the editing features of a Petri net editor. Each editing class encapsulates a different editing feature, for example EditArcs will be used to modify arc positions and weights, while EditSubnet, will provide operations to edit a subnet and its interaction points. Each of the editingClasses extends the same base class, EditClass. The class will have abstract functions to respond to the mouse being pressed, dragged and released. Each of the editing classes will have to implement, these functions. This design will simplify the addition of further editing features, which can be implemented in a new class, which inherits EditClass.

The Java implementation for the user interface is described in detail in chapter four section 2 (4.2).

3.3 Open Architecture Design

The open architecture design must allow analysis modules that perform differing analysis operations to be dynamically loaded and executed by the editor. Modules have to follow an interface, to ensure that they implement the functions that the editor will invoke to run their analysis functions. The interface design is simple so as not to restrict the varying analyses that modules can perform. The simplest interface requires two functions, one to return the name of the module and another to execute analysis.

Once dynamically loaded analysis modules will need to obtain the details of the Petri net being edited. It is simplest if the Petri net is saved to a set file name, which can be specified in the interface. Modules will then know where to locate the file associated with the Petri net they are to analyse.

3.4 File Format Design

None of the proposed Petri net interchangeable file formats (see 2.2.4) support hierarchical Petri nets. It was therefore necessary to design a file format that closely follows the proposals but also provides a format to save hierarchical Petri nets.

A natural way to save a hierarchical Petri net is for each net to have its contents saved in a separate file. Where a Petri net includes a subnet, the subnet can be specified by providing the file name for the subnet. However if the file format is to be used by other Petri net editor tools, that don’t support hierarchical Petri nets, files couldn't be transferred between different tools. It was therefore necessary to design a file format that would hierarchical Petri nets to be flattened and saved as a single file, making them more accessible to other tools.

One of the proposed Petri net interchange file formats, The Petri Net Markup Language (PNML) provides a simple XML format for specifying places, transitions and arcs, a simple example is shown in figure 11. This format formed the basis of the file format that was designed. The format for the arc was not altered. The format for places and transitions needed modifying to identify if they are interaction points. An interaction tag was added to distinguish components as interaction points (figure 13– the transition is an interaction point). The transition format required further modification because the PNML does not allow for timed and immediate transitions present in Generalised Stochastic Petri nets. Attributes for type and weight were added for immediate transitions and type, distribution and rate for timed transitions (figure12).

A new structure had to be designed to accommodate subnets. A subnet tag was added (figures 13 & 14). The name of the subnet is provided as an attribute of this tag. For subnets that relate to different Petri net files, a location tag provides the location of the file (figure 14). If a High level Petri net is saved in a flattened form, the contents of the subnet are contained within a contents tag (as shown in figure 13), which is nested within the subnet tag. Also nested within the subnet tags are graphics and position tags, which are used in the same way as for transitions and places to specify the subnet's location. Offset tags are also nested within the subnet tag, they specify the locations of the subnets interaction points when they are shown on the higher-level net.

```
<?xml version="1.0" encoding="UTF-8" ?>
 <pnml>
    <transition id="t0">
       <graphics>
          <position x="89" y="157" />
       </graphics>
       <name>
          <value>t0</value>
          <graphics>
             <offset x="-2" y="-2" />
          </graphics>
       </name>
    </transition>
    <place id="p0">
       <graphics>
          <position x="300" y="170" />
       </graphics>
        <name>
          <value>p0</value>
          <graphics>
             <offset x="298" y="168" />
          </graphics>
       </name>
       <initialMarking>
          <value>0</value>
          <graphics>
             <offset x="315" y="185" />
          </graphics>
       </initialMarking>
    </place>
    <arc id="a0" source="t0" target="p0">
       <graphics>
          <position x="95" y="171" />
          <position x="315" y="188" />
       </graphics>
       <inscription>
          <value>1</value>
          <graphics>
             <offset x="120" y="196" />
          </graphics>
       </inscription>
    </arc>
  </pnml>
```

Figure11. *An example of a Petri Net Markup Language File*. The above sample file specifies a place, a transition and an arc from the transition to the place. The Petri Net markup language allows an name, position and text position to be saved for transitions, while places also have their initial marking (token number) saved. The source and target of an arc, its start and finish positions and the arc weight are saved within the Arc tag.

a)

```
    <transition id="t0" type="timed" distribution="exponential"
        rate="1.0">
       <graphics>
          <position x="89" y="157" />
       </graphics>
       <name>
          <value>t0</value>
          <graphics>
             <offset x="-2" y="-2" />
          </graphics>
       </name>
  </transition>
```

b)

```
   <transition id="t0" type="immediate" weight="1.0">
       <graphics>
          <position x="113" y="78" />
       </graphics>
       <name>
          <value>t0</value>
         <graphics>
             <offset x="-113" y="-78" />
          </graphics>
       </name>
    </transition>
  </pnml>
```

Figure 12. *XML format for Transitions*. a) The format for a timed transition. Attributes have been added to the transition element. Type identifies that the transition is timed, attributes also identify the distribution and firing rate of the transition.
b) The format for an immediate transition. The type is set as immediate and the weight is specified using the weight attribute.

```
<?xml version="1.0" encoding="UTF-8" ?>
<pnml>
  <subnet id="subnet0">
    <graphics>
      <position x="188" y="148" />
      <size w="50" h="70" />
    </graphics>
    <contents>
      <transition id="t0" type="timed"
          distribution="exponential" rate="1.0">
        <graphics>
          <position x="89" y="150" />
        </graphics>
        <name>
          <value>t0</value>
          <graphics>
            <offset x="-2" y="-2" />
          </graphics>
        </name>
        <interaction />
      </transition>
      <place id="p0">
        <graphics>
          <position x="238" y="146" />
        </graphics>
        <name>
          <value>p0</value>
          <graphics>
            <offset x="236" y="144" />
          </graphics>
        </name>
        <initialMarking>
          <value>0</value>
          <graphics>
            <offset x="253" y="161" />
          </graphics>
        </initialMarking>
      </place>
      <arc id="a5" source="t0" target="p0">
        <graphics>
          <position x="95" y="164" />
          <position x="253" y="159" />
        </graphics>
        <inscription>
          <value>1</value>
          <graphics>
            <offset x="0" y="0" />
          </graphics>
        </inscription>
      </arc>
    </contents>
    <offset x="162" y="119" />
  </subnet>
</pnml>
```

Figure 13. A “flattened” Petri net file containing a single subnet. The subnet is the Petri net specified in figure 11. Its contents are listed between the <contents> tag. The offset tag just before the subnet end flag (</subnet>) specifies the location of the subnet’s transition (which is an interaction point), when it is viewed at the higher level.

```
<?xml version="1.0" encoding="UTF-8" ?>
<pnml>
   <subnet id="subnet0">
      <graphics>
         <position x="188" y="148" />
         <size w="50" h="70" />
      </graphics>
      <location file="H:\msc\IndividualProject\XML thesis
         files\pnml figure.xml" />
      <offset x="162" y="119" />
   </subnet>
</pnml>
```

Figure 14. *A non flattened Petri net File*. This is the equivalent non flattened file format for the flattened file format of figure 13. The subnet contents are not listed in this file but the location of the subnet file is given from which its contents can be obtained.

4 Implementation

The first section of this chapter will address the implementation of the Petri net classes and their incorporation into the Graphical user interface. The following section describes the user interface implementation in more detail. The third section discusses implementation of the open architecture followed by the implementation of the invariant analysis module. For clarity the implemented classes, Java classes, data members and functions are printed in a `Courier` font, functions are distinguished by `functionName()` irrespective of their number of arguments.

4.1 Implementation of Petri Net Classes

4.1.1 Implementation of PetriComponent

Possible Petri net class structures were described in the previous chapter, Petri net components were designed to all inherit a class, PetriComponent. This section describes the implementation of the `PetriComponent` class.

The `PetriComponent` class incorporates the features common to all Petri net components. Petri net components all have a position, name and text position, so these are provided as data members of `PetriComponent`. Petri net components also need to interact with the design area (see 4.2.2) so functions (`startPos()`, `setPosition()` and `finishDraw()`) were added to `PetriComponent` to provide implementation to do this. Finally Petri net components need to be able to paint themselves and when being saved write their details to file, so the functions `paintComponent()` and `writeToFile()` were also added to `PetriComponent`. These main data members and functions of `PetriComponent` are displayed in figure 15.

```
public abstract class  PetriComponent implements Observer {

  Coordinates position;
  String name;
  Coordinates textPosition;

  public abstract void startPos(Coordinates startPos, Drawer d);
  public abstract void setPosition(int x, int y);
  public abstract void finishDraw(Drawer d);
  public abstract void paintComponent (Graphics g);
  public abstract void deleteComponent(Drawer d);
  public void writeToFile(FileWriter out){
  }


  ……………….
}
```

Figure 15. *The main data members and functions of the PetriComponent class*. Position, stores the position of the component in the design area. Likewise the textPosition stores the position that a component's name is displayed relative to its position. Name stores the name of a component. StartPos() is called to set the initial position of a component when the mouse is pressed (See next section), setPosition() is called when the mouse is dragged and finishDraw() when the mouse is released. PaintComponent(), provides implementation to paint the component and deleteComponent() provides implementation to delete Petri Net Components. WriteToFile() is implemented to write a component's details to file.

The individual characteristics of Petri net components also need to be represented, the next section considers what Petri net components need to know about the other components of a Petri net and how this was implemented.

4.1.2 Implementation of An Object Oriented Petri Net

What do the components of a Petri net need to know about each other? Figure 9 demonstrates the relationships between the Petri net component classes but how should these be implemented? Transitions are the active component of a Petri Net, they need to be able to determine if they are enabled and cause change when they are fired. To identify if a transition is enabled it must therefore be aware of the arcs that input into it. The arcs in turn then need to be aware of the places to which they are connected to identify if there are sufficient tokens on the place. Alternatively transitions could also be aware of the places that the arcs are connected to. However such a design introduces redundancy because both arcs and transitions would need to store references for the places that arcs connect to. Similarly when firing, transitions need to know the output arcs connected to them and in turn the arcs must be aware of the places to which they are connected.

For this reason, arcs (implemented in the `Arc` class) contain a reference to their source and target, while the `Transition` class has lists (Vector class used) of references to input and output arcs. A single list is not used because the operations performed on input and output arcs are always different, so it is more efficient than working over a single list and testing each arc to identify if it inputs or outputs to the transition.

From this perspective places do not need to be aware of the arcs that connect them, because they do not perform any operations on the arcs connected to them. This arrangement would be suitable in some cases, however as a graphical editor is being designed it is not. The graphical editor allows places and other components to be moved and manipulated; so when a place is moved, the arcs connected to it must also move. Places therefore also contain lists (Vector class used) of input and output arcs.

As described in Design, places and transitions share similarities such as dimensions and the ability to have arcs connected to them. These similarities were encapsulated in the `SolidPetriComponent` class (extends `PetriComponent`), which provides implementation to add and remove arcs and to modify component position (see figure 16).

The `Token` class inherits from `PetriComponent`. Places need to be able to access the tokens on them, so they are stored as elements of a Vector, `tokens`, in the `Place` class. A separate data member to store the number of tokens is not required as this can be obtained from the size of the vector.

```
public abstract class SolidPetriComponent extends PetriComponent {
  Vector outputArcs;
  Vector inputArcs;

  Dimensions size;

  EditPoint north, east, south, west, centre;


  boolean inSubnet;
  boolean interactionPoint;
  Coordinates netPosition;

  public void removeArcs(Drawer d){}
  public void addEditPoints() {}
  public Vector getEditPoints () {}
  public void update(Observable obs, Object obj){}
  public boolean isOn(Coordinates c) {}
  public void addArc(Arc a, boolean input) {  }
  public void updateArcPositions(){}
  public void setInteraction(){}
  public void setInteractionPosition(InteractionPoint ip){}
  public void switchPosition(){
  }

       …………………………….
}
```

Figure 16. *The main data members and functions of the SolidPetriComponent Class*.

4.2 Graphical User Interface Implementation

The first section provides a brief overview of the swing classes used and extended in the design of the user interface. The subsequent section describes how, these components interact with the Petri net classes described in above.

4.2.1 Swing Components

The Java swing library was used to implement the Graphical User Interface (GUI). Figure 17 displays the class structure used for the GUI implementation. The frame of the user interface is implemented in the `EditorFrame` class, which extends `JFrame`. The design area is implemented in `Drawer`, which extends `JPanel`. A further `JPanel`,

`jPanel4`, contains the class `jToolBar1`, which extends `JToolBar` and implements the file toolbar. `JPanel2` contains `JButtons`, which are used to select editing options. An instance of `Jtable` is used to provide the editing table, which uses the abstract table model `TransitionTableModel`.

A `JSplitPane` instance covers most of the user interface area. There is a `JscrollPane` on each side of the splitPane, which contain the design area on the left and the editing table on the right. This enables the size of the design area and table to be modified and makes both of them scrollable.

A `JLabel` instance is used to provide both the `statusBar` and the `positionBar`. Table1 summarises the classes that the main components of the user interface either extend or are instances of.

Class Name	**Function**	**Java Class**
EditorFrame	GUI frame	JFrame
Drawer	Drawing area	JPanel
JPanel1	Editor toolbar (for editor functions)	JPanel
JPanel2	Contains FileToolBar	JPanel
FileToolBar	Buttons for file functions (open, save etc)	JToolBar
StatusBar	Output information to user	Jlabel
PositionBar	Display position on draw area (Drawer)	Jlabel
JFileChooser1	Display dialog for	JFileChooser
JmenuBar1	Implement menus	JMenuBar

Table 1. *The classes present in the GUI.*

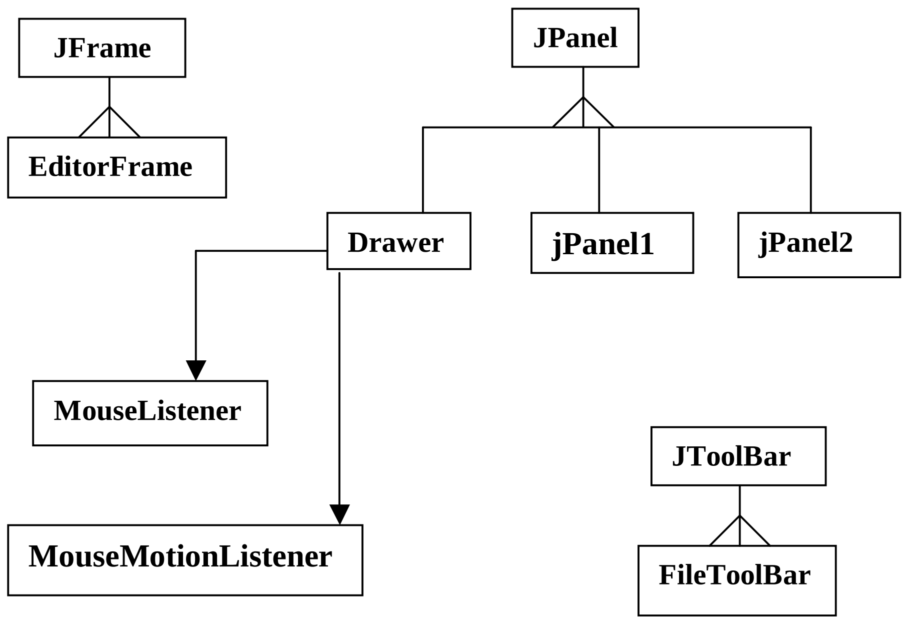


Figure 17. *A Class Diagram of the classes used in the Graphical User Interface.* The class EditorFrame, provides the frame for the user interface, this class extends the swing class JFrame. The other components of the user interface are all part of the EditorFrame. The classes Drawer, jPanel1 and jPanel2 all extend the Swing class, JPanel. These classes implement areas of the user interface. Drawer is responsible for the design area, for this reason it implements MouseListener and MouseMotionListener interfaces, to enable the design area to respond to mouse actions. JPanel1 implements the edit toolbar, which contains buttons to select editing features. JPanel2 is located at towards the top of the user interface and contains the FileToolBar. The FileToolBar extends the Swing class JToolBar. The JsplitPane and JscrollPane classes used in the user interface are not displayed in this figure. The JsplitPane, has a JscrollPane on either side, the scrollPane on the left contains the design area, while the right scrollPane contains the EditTable.

4.2.2 Interaction of Petri Net Classes with User Interface

The previous section described the basic design of the user interface, these features had to be combined with the Petri net classes to produce a functioning editor.

Editing classes were introduced in Design (see chapter 3); these classes provide implementation to perform editing functions upon Petri nets, with each class encapsulating a different feature. All these classes inherit the same base class, (called EditClass in Design, see figure 10). When implementing these classes it was decided that the base class they extend should be `PetriComponent`, the class that all Petri net component classes extend. This was done because most of the classes require data members present in the `PetriComponent` class, such as `position`, and the editing classes have to interact with the `Drawer` instance (the design area), as do the Petri net component classes. This implementation simplifies the implementation of the `Drawer` class and the addition of new editing features as described below.

The `Drawer` class, which provides the design area, also stores all the data concerning the Petri net that it displays. Instances of the `Vector` class are used to store the places, transitions, subnets and arcs of the Petri net.

All design actions (i.e. Petri net component classes and editing classes) such as adding transitions or editing components are performed within the `Drawer` design area, of the user interface. At all times one of the design actions (buttons on the edit toolbar) is selected, and an instance of the respective class is assigned to a `PetriComponent` reference in `Drawer`, called `selectedShape`. For example, if `Arc` is selected then an instance of arc is assigned to `selectedShape`.

`Drawer` implements both `MouseListener` and `MouseMotionListener` interfaces (java.awt.event) to identify mouse actions. `mousePressed()`, `mouseDragged()` and `mouseReleased()` functions are called to act upon the mouse being pressed, dragged and released respectively (as their names might suggest). Each of these functions call a different `PetriComponent` function for the selected shape; `mousePressed()` calls `startPos()`, `mouseDragged()` calls `setPosition()` and `mouseReleased()` calls `finishDraw()`. The classes extending `PetriComponent` implement these functions to provide the correct editing capabilities. Figure 18 demonstrates the `mousePressed()` function from the `Drawer` class and the `startPos()` function from the `Arc` class.

This modular design simplifies the `Drawer` implementation, by requiring each of the `PetriComponent` inheriting classes to implement functions to deal with mouse actions. This means that `Drawer` calls the same functions independent of whether a Petri net component class such as `Arc` is selected or an editing class such as `EditArcs` is selected as demonstrated in figure 18. This simplifies future extensions to the editing capabilities as new features can be added by implementing a new class extending `PetriComponent`, without requiring modification of the `Drawer` class.

a)
```
public void mousePressed(MouseEvent e) {
      e.consume();
      startPos.x = e.getX();
      startPos.y = e.getY();
      theFrame.statusBar.setText("Mouse Pressed at (" + startPos.x +
","
                                   + startPos.y + ")");
      //…………
      selectedShape.startPos(startPos, this);
}
```

b)
```
public void startPos(Coordinates startPos, Drawer d){

    // check if starts on a transition or a place
    int num = d.transitions.size();
    for (int i=0; i < num; i++) {

      Transition temp = (Transition)d.transitions.elementAt(i);
      if (temp.isOn(startPos)) {
        //set the arc so that it originates from the centre of the
        position.x = temp.position.x + (temp.size.w/2);
        position.y = temp.position.y + (temp.size.h/2);
        arcTran = temp;
        drawingPosition();
        return;
      }
    }

    num = d.places.size();
    for (int i=0; i < num; i++) {

      Place temp = (Place)d.places.elementAt(i);
      if (temp.isOn(startPos)) {
        position.x = temp.position.x + (temp.size.w/2);
   position.y = temp.position.y + (temp.size.h/2);
        arcPlace = temp;
        drawingPosition();
        return;
      }
      drawingPosition();
    }

       finishPos.x = position.x =startPos.x;
       finishPos.y = position.y =startPos.y;
 }
```

Figure 18. *Sample of mousePressed() function the Drawer class and startPos() from the Arc class.* a) mousePressed() is called when the mouse is pressed within the Drawer (design area). It calls startPos() for the selected PetriComponent, selectedShape. b) This sample of code from StartPos() in the Arc class, checks if the arc begins on a place or transition and sets variables appropriately.

4.3 Hierarchical Petri Net Implementation

Once a basic editor had been implemented, support for hierarchical Petri nets needed to be implemented. The main features that had to be addressed were the selection and identification of interaction points, the ability to add subnets to a Petri net and most importantly the mechanism to move between different levels of a Petri net effectively.

4.3.1 Interaction Points

Selecting interaction points was one of the simpler tasks. A Boolean variable, `interactionPoint`, was added to the `SolidPetriComponent` class (figure 16). This variable is set true when a place or transition is an interaction point and false otherwise.

A class `InteractionPoint` was implemented to provide the ability to select and deselect interaction points. This class extends `PetriComponent` like the other editing classes. The `startPos()` function was implemented to identify if the mouse is pressed over a place or transition and if so its `interactionPoint` status is toggled (i.e. set false if was already true, and set true if it was previously false).

To distinguish interaction points from other transitions and places, the `paintComponent()` functions of transitions and places were modified, such that interaction points are filled green.

Interaction points are displayed on the outside of subnets (figure19), they also have a location inside the subnet. To store both of these positions a second position called `subnetPosition`, was added to the `SolidPetriComponent` class. `subnetPosition` stores the position of the interaction point on the outside of the subnet.

A reference to a subnet, `theSubnet` was added to `SolidPetriComponent`, it refers to the subnet the component belongs to, if any. Components need to know which subnet they belong to for displaying their full name when interaction points are viewed (see figure 19) and when writing to file in save operations. A Boolean variable, `inSubnet`, is used to identify if a subnet is the Petri net currently displayed in the design area or if it is present as a subnet. This variable is used to ensure that only the correct components are painted, for example, only interaction points are painted when the net is displayed as a subnet, but all components are displayed when viewed as the main Petri net. (figure 19).

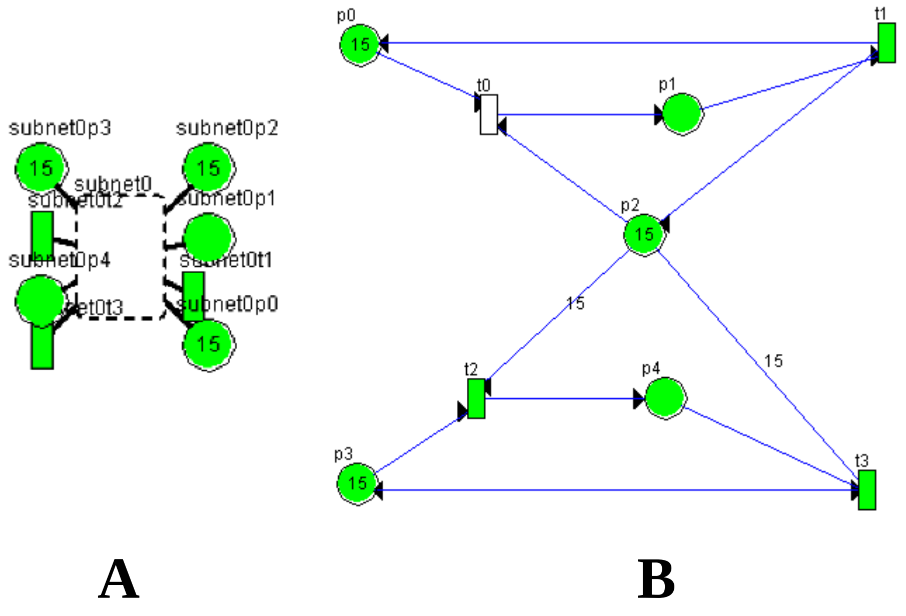


Figure 19. *A simple Petri net demonstrating the positions of interaction points.*
A) The representation of a subnet in a hierarchical Petri net. The interaction points of the subnet are shown on the outside of the subnet. They display their full name, which is the name of the subnet concatenated with their own name (e.g. place p3 is labelled subnet0p3).
B) The Petri net that the subnet represents, in this example, the reader writers net from Figure 2 has been used. Eight of the places and transitions have been set as interaction points. All the places and transitions, not just the interaction points are visible as demonstrated by transition t0, which is displayed in the B but not in A.

4.3.2 Addition of Subnets

The class `Subnet` encapsulates the ability to add subnets to a Petri net and to store and manipulate the contents of a subnet. `Subnet` was implemented to inherit `SolidPetriComponent`, because like places and transitions, subnets can be added to Petri nets. They can also be selected for editing and have their position altered, and implementation to do this is provided in `SolidPetriComponent`. The `Subnet` class contains `Vectors` to store its places, transitions, subnets and arcs.

To provide a choice of how a subnet is added the `finishDraw()` function of the `Subnet` class, displays a `JOptionDialog` prompting the user to either select a blank subnet or a file to use for the subnet. If a file is chosen, an instance of `XmlFileReader` (see 4.6) is created to parse the file.

Subnets need to display their interaction points, so the higher level Petri net can identify them. When a subnet is added, interaction points are identified as a subnet file is parsed and the `Subnet` function `setInteractionPositions()` is called. This function sets an interaction point's position on the outside of the subnet to one of eight preset positions. The eight interaction points on the subnet in figure 19A demonstrate these positions.

4.3.3 Switching between levels of the subnet

Implementation was required to enable moving up and down hierarchical Petri nets, to display and edit different levels of the Petri net. Importantly the hierarchical structure must be maintained while navigating a hierarchical Petri net. To ensure this the implementation providing movement between levels of a hierarchical Petri net, leaves the hierarchical structure as it is and just copies the relevant subnet that is to be displayed to the instance of `Drawer` (i.e. to the design area). Two vectors `currentLevel` and `higherNets`, were added to `Drawer` class, they both store references to subnets. They are required to ensure the hierarchical structure is maintained and any editing is both displayed on the design area and added to the correct subnet.

The `EditSubnet` class was implemented to manage moving down a hierarchical Petri net. It extends `PetriComponent` like the other editing classes. When the mouse is clicked it identifies if it was clicked on one of the displayed subnets, if so that subnet is selected and loaded (figure 20 includes a sample of the `loadSubnet()` function). When a subnet is loaded a reference to the selected subnet is added to the end of the `currentLevel` vector. This ensures that once the subnet is loaded, the last element of this vector will refer to the subnet that is being displayed. This reference is used to ensure that any changes made at this level are carried out on this subnet. A reference to the level that is being left is added to the `higherNets` vector. The `higherNets` vector identifies the levels that were displayed before the current level on display. This vector can then be used when the navigating back up the hierarchy (see below). The contents of the selected subnet are then copied to the `Drawer` vectors so it can be displayed. While this is done, the `inSubnet` variables of the places and transitions are set to false, because the components are no longer being displayed as if they are inside a subnet. The positions of interaction points also have to be set to ensure they are displayed at the correct locations.

A `Jbutton` called the `backButton` was added to the user interface to move back up the hierarchy. A separate class like `EditSubnet` was not required to implement moving back up the hierarchy because the path followed down the hierarchy is reversed and there is no choice as to which level to return to. To move up a single level, the contents of the net being displayed have their variable `inSubnet` set to true, the last element of the `higherNets` vector is removed and its contents copied to the instance of `Drawer`, the last element of `currentLevel` is also removed. Figure 21 displays a sample of the function that performs this.

```
public void loadSubnet(Subnet selSubnet, Drawer d){
    Subnet tempSubnet = new Subnet();
    d.checkLevel(false);
    tempSubnet.copyToSubnet(d);
    d.higherNets.addElement(tempSubnet);
    d.clearCanvas();
     selSubnet.copyToDrawer(d, false);
 }
```

Figure 20. *loadSubne()t function from the EditSubnet class.* This function is called when a subnet has been selected to load, the contents of the selected subnet are copied to the drawer so as to display them and the previous level is added to the higherNets vector.

```
  void jButton9_actionPerformed(ActionEvent e) {
    //This button moves back a level;
    if (drawArea.higherNets.size() != 0){
      Subnet tempSubnet = (Subnet)drawArea.higherNets.lastElement();
      //Switch elements currently on drawer to be inSubnet
      int num = drawArea.transitions.size();
      for (int i =0; i < num; i++){
        Transition tempTran =
              (Transition)drawArea.transitions.elementAt(i);
        tempTran.inSubnet = true;
        if (tempTran.netPosition != null)
          tempTran.switchPosition();
      }

      num = drawArea.places.size();
      for (int i =0; i< num; i++){
        Place tempPlace = (Place)drawArea.places.elementAt(i);
        tempPlace.inSubnet = true;
        if (tempPlace.netPosition != null)
          tempPlace.switchPosition();
      }

      tempSubnet.copyToDrawer(drawArea, true);
      drawArea.higherNets.remove(tempSubnet);
      drawArea.checkLevel(true);
      repaint();

drawArea.currentLevel.remove(drawArea.currentLevel.lastElement());
    }
    else return;
  }
```

Figure21. *actionPerformed() function to move to a higher Level.* A jButton is used to move back up the hierarchy. When pressed it sets the inSubnet variable of the components of the displayed net to true and copies the higher level subnet to the Drawer (This is the last element of the higherNets Vector).

4.4 Open Architecture

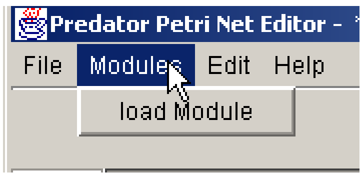
The Reflection API, part of the Java programming language, enables executing programs to dynamically load classes. The capabilities of Reflection were exploited in the implementation of the open architecture.

An interface was implemented that modules to be dynamically loaded must follow. It is shown in figure 22. The `Module` interface follows the design described in Design (chapter 3). It contains two functions, `runModule()` which executes the module's analysis functions and `getModuleName()`, which returns the module's name. The third element of the interface is the string, `inputFileName`, a constant, which is initialised to "current.xml" the file modules parse to obtain the Petri net that they are to analyse.

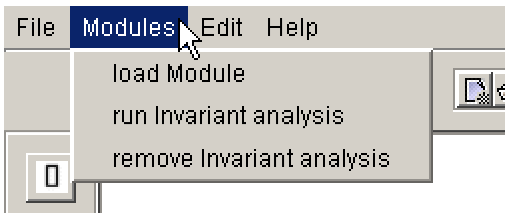
```
public abstract interface Module {
        public abstract void runModule();
        public abstract String getModuleName();
        static final String inputFileName = "current.xml";
}
```

Figure 22. *The module interface*. Analysis module to be dynamically loaded must implement the module interface. RunModule() is called to execute the analysis functions, getModuleName() returns the name of the analysis module and inputFileName is set to the file name that modules parse to obtain the structure of the Petri net.

To load a module, the load module menu item (figure 23) from the module menu must be selected. A File Chooser enables selection of the class to be dynamically loaded. The name of this class is then passed to the `createObject()` function, a function of `EditorFrame`(figure 24), which creates and returns an instance of the class. This object is then assigned to a data member of `EditorFrame`. Upon loading a module, menu items are added to the module menu to run and remove the module, as shown in figure 23.



A



B

Figure 23. *The module menu*.
A) The module menu without any modules loaded. Load module is selected to load an analysis module.
B) The module menu with the Invariant analysis module loaded. A menu item to run the invariant analysis module and another to remove it are added to the module menu.

```
static Object createObject(String className) {
      Object object = null;
      try {
          Class classDefinition = Class.forName(className);
          object = classDefinition.newInstance();
      } catch (InstantiationException e) {
          System.out.println("Error loading Module");
          System.out.println(e);
      } catch (IllegalAccessException e) {
          System.out.println("Class Couldnot be accessed");
          System.out.println(e);
      } catch (ClassNotFoundException e) {
          System.out.println(e);
         System.out.println("Class could not be found");
      }
      return object;
   }
```

Figure 24. *Use of Reflection to dynamically load classes*. The createObject() function is passed a string containing the name of the class to dynamically load and returns an instance of this class. This function is adapted from the Java Tutorial www.java.sun.com/tutorial.

To run a module and to obtain the name of a module, the respective functions in the module class first have to be identified and then invoked. Figure 25 demonstrates the `Drawer` class function `runAModule()`, which is called to execute a module's analysis features. Before analysis is performed the current Petri net is saved to a file called current.xml (the file that modules load Petri nets from). The `runModule()` method of the analysis class is then obtained using the `getMethod()` function (from the Reflection library), which returns the method to a Method variable (called `runModuleMethod` in Figure 25). The method can now be invoked using `runModuleMethod.invoke(module,arguments)`, which is passed the instance of the class upon which the method is to act and an array of arguments, which in this case is empty. This results in calling the `runModule()` function, which should perform the analysis. To obtain the module name, a similar function is used to invoke the module's `getModuleName()` method.

```
private void runAModule(){

  //save the Petri net to current.xml
    saveFile("current.xml");

    Class[] parameterTypes = new Class[] {};
    Method runModuleMethod;
    Object[] arguments = new Object[] {};
    Class moduleClass = module.getClass();
    try {
      runModuleMethod = moduleClass.getMethod("runModule",
                                                 parameterTypes)
                                              ;
      runModuleMethod.invoke(module, arguments);
    } catch (NoSuchMethodException t) {
        //System.out.println(t);
        System.out.println("Error loading module!");
    } catch (IllegalAccessException t) {
        //System.out.println(t);
        System.out.println("Error invoking method!");
    } catch (InvocationTargetException t) {
        System.out.println(t);
        t.printStackTrace();
    }
}
```

Figure 25. *runAModule() function from EditorFrame.* runAModule(), initially saves the file to Petri net to the file current.xml. It then obtains the runMethod() Method from the desired module class and subsequently invokes it to perform the analysis. This function was adapted from functions in the Java Tutorial www.java.sun.com.

When a module is removed, the menu items are removed from the module menu and the reference to the module instance in `EditorFrame` is reset to null, removing all references to the module.

4.5 Invariant Analysis Module

The class structure of the Invariant analysis module is simple compared to the editor itself. The module consists of three classes, `Analysis`, `InvariantXMLFileReader` and `InvariantDialog`. `Analysis` implements the `Module` interface (figure 22) and implements the Invariant analysis algorithm. `InvariantXMLFileReader` parses current.xml initialising the data fields of an instance of `Analysis`.

When the `Analysis` method `runModule()` method is called, an instance of `InvariantXMLFileReader`, is created, the file is parsed and the algorithm executed. The algorithm determined by D'Anna & Trigila (D'Anna & Trigila 1988) for finding invariants was implemented. An outline of the algorithm is reproduced below.

4.5.1 Invariant Analysis Algorithm

The same algorithm is used to calculate both P and T invariants. The incidence matrix C is the input for the algorithm. C is used for calculating T invariants, while the transpose of C is used for the calculation of P invariants.

The algorithm:

Initialisation

- The Incidence matrix C has dimensions m x n , the number of places and transitions respectively.
- An identity matrix, B, of dimensions n x n is constructed
- The extended matrix is formed by combining C and B, by writing C above B. The resulting extended matrix has m+n rows and n columns.

Phase 1 - removes non-zero elements from C

- **While** there are non zero elements in C **do**
    - **If** there is a row h in C such that either sets, P+ and P- are the empty set. Where P+ is all the positive elements in row h and P- is all the negative elements in row h) **then**
        - Delete from the extended matrix all columns where row h has a non zero column.
    - **Else** (if there is a row h in C with $|P^+| = 1$ or $|P^-| = 1$ **then**
        - Set k equal to the unique index of the column belonging to P+ (algorithm reversed if $|P^-| = 1$)
        - **For** (j in P-) **do**
            - Substitute to the column of index j the linear combination of the columns indexed by k and j with the coefficients |C(hj)| and (C(hk)| respectively.
        - Delete the column of index k from the extended matrix.
    - **Else**
        - set h equal to the index of the non-zero row of C, and k to the index of the column so C(hk)!=0
        - **for** (j where J!=k and C(hj)!= 0) **do**
            - substitute to column with index j the linear combination of the columns with indicdes k and j with coefficients α and β such that α and β are: if (sign(C(hj)) != sign sign(C(hk)) then $\alpha = |C(hj)|$ and $\beta = |C(hk)|$ else $\alpha = -|C(hj)|$, $\beta = |C(hk)|$.
            - Delete from the extended matrix column with index k

Phase 2

```
• While (B has arrow with index h containing negative elements) do
    o P$^-$ = negative elements in row h
    o P$^+$ = positive elements in row h
    o If P$^+$ is not empty
        ▪ For (j,k) in P$^+$ x P$^-$ do
            • Perform a linear combination on columns j ank
              k to obtain a new columns with the h-th
              element equal to zero,
            • Divide the new column by the greatest
              common denominator of its elements
            • Append the column to B
    o Delete from B all columns with indexes in P-

• Delete from B all columns having non-minimal support
```

4.5.2 InvariantDialog

InvariantDialog extends JDialog. It displays the results of the invariant analysis. P and T invariants, P invariant equations and information regarding the liveness and boundedness of the Petri net are displayed in separate text fields.

4.6 File Format

Files are written using a `FileWriter` instance. A function `writeToFile()` is called for each component of the Petri net to save its details to file.

XML file parsing to read files when opening or adding subnets was aided by the new JAXP1.1 library (www.java.sun.com/xml/) which provides classes to perform XML parsing. The class `XmlFileReader` was implemented to handle parsing, it extends `DefaultHandler`, a class from this library. The `XmlFileReader` class creates a `SAXParser` (a class form the JAXP library) instance to parse the XML file, and also provides the implementation to interpret the XML tags identified by the `SAXParser`. This is done using three simple functions, `startElement()`, `endElement()` and `characters()`. The `SAXParser` calls `startElement()` when a start tag is encountered. `startElement()` identifies what the tag is, any attributes it may have and provides implementation to act upon this information. A sample of `startElement(),` is shown in figure 26, it shows that when a place tag is encountered, a new `Place` instance is created and it name is set.

`EndElement()` is called when a terminating tag is encountered. Like `startElement()`, `endElement()` identifies what the tag is and functions are called to process this information. Figure 27, demonstrates this for an end place tag (</place>). Finally `characters()` is called when there are parameters between tags. This function is implemented to identify such parameters and act accordingly.

```
public void startElement(String namespaceURI,
                         String sName,
                            String qName,
                         Attributes attrs)
  throws SAXException
  {
    String eName = sName;   //element name
     if ("place".equals(eName)){
      newComponent = new Place(0);
      if (attrs != null){
        newComponent.name = attrs.getValue(0);
      }
    }

     if(“transition”.equals(eName)){
            //......................................
     }

  }
```

Figure 26. *startElement() Function in XmlFileReader.* When the SAXParser encounters the start of a tag, startElement() is called, which provides the implementation to parse the tag. The section of code demonstrates how a place tag (<place>) is parsed.

```
public void endElement(String namespaceURI,
                           String sName, // simple name
                           String qName  // qualified name
                       )
    throws SAXException
    {
        String eName = sName;

      if("place".equals(eName)){
         newComponent.addEditPoints();
         if (subnetFlag){

      if(((SolidPetriComponent)newComponent).interactionPoint){

             newSubnet.setInteractionPosition((SolidPetriComponent)n
             ewComponent);

           }
         ((Place)newComponent).inSubnet = true;
         newComponent.theSubnet = newSubnet;
         newSubnet.places.addElement(newComponent);
         newComponent.theSubnet = newSubnet;
         }
         else
         d.places.add(newComponent);
      }

      ………
}
```

Figure 27. *endElement() function in XmlFileReader.* When the SAXParser encounters an end element the endElement() function in the DefaultHandler (XmlFileReader) is called to parse the tag. This section of code demonstrates the actions taken when an end place tag is reached (</place>).

5 Results & Case Studies

This Project has designed and implemented a Petri net editor called Predator. In this chapter a number of case studies are used to investigate the capabilities of the Predator Petri net editor.

5.1 Basic Editor Features

An initial aim of the project was to provide simple Petri net editor functions. These include, adding arcs, places, transitions and tokens. A simple Graphical user interface was designed to incorporate these features; it is shown in figure 28.

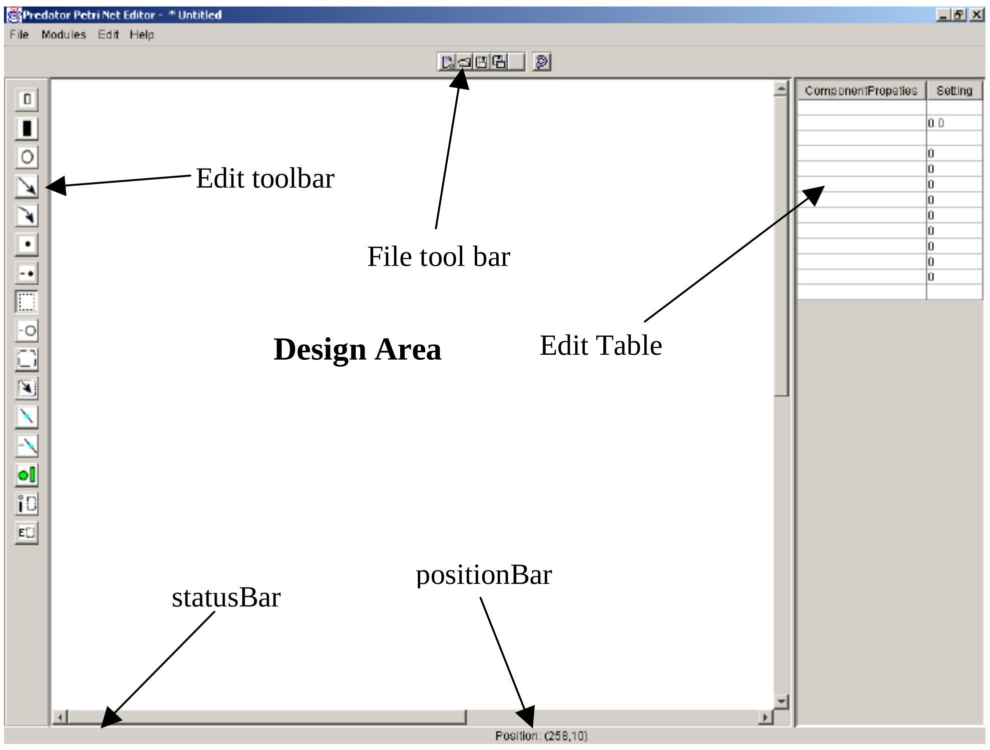


Figure 28. *The Predator Graphical User Interface*. The arrows indicate the components of the interface.

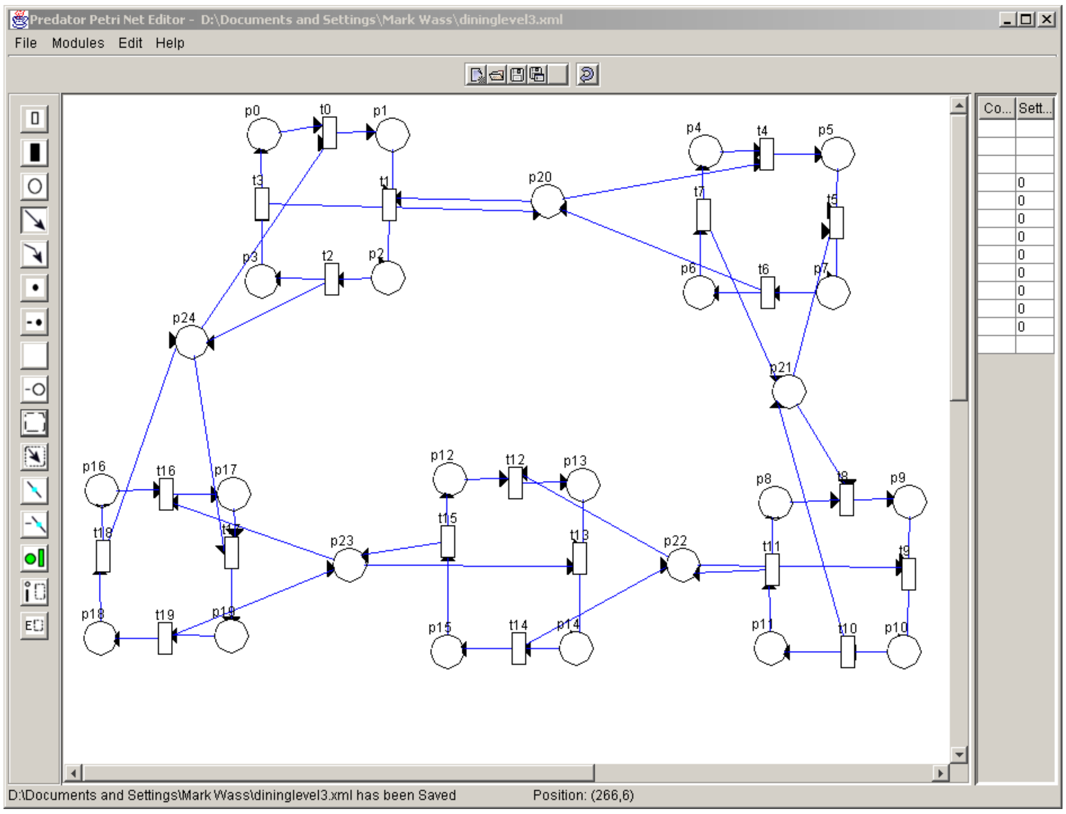
The user interface was implemented to provide a large design area, with the editing options easily accessible in the edit Toolbar and on the edit Table. All the operations required in designing a Petri net could be done using these three elements of the user interface.

The ability of Predator to design Petri nets will be demonstrated by the case studies in the following sections. Full details for using the Predator Petri net editor are provided in the user guide (chapter 9).

5.2 Hierarchical Petri Nets

The Predator Petri net editor enables the design of hierarchical Petri nets via the addition of subnets. Predator allows subnets to be added to any other Petri net, allowing multiple level Petri nets to be designed. The implementation requires only single mouse clicks to navigate the various levels of a hierarchical Petri net. To move down a level, the editSubnet icon must be selected, any subnet can then be viewed by clicking on it. To move up a level a single click on the back level icon is required. Predator allows interaction points to be visibly seen on the outside of the subnet. This feature enables arcs between levels to easily be specified and observed as shown in the dining philosophers example below (figure 31).

The dining philosophers is a common computing problem, used to demonstrate deadlock. The system consists of five philosophers who are either thinking or eating. There is a fork/chopstick between each of the philosophers. To eat a philosopher must pick up the two forks directly next to him. A single level Petri net of this problem is displayed in figure 29. From this it is evident that each of the philosophers has the same structure enabling them to be represented by a single Petri net (figure 30). This Petri net can then be added to the system as a subnet for each of the philosophers (figure 31). Comparison of the single level net and the hierarchical net demonstrate the advantages of hierarchical Petri net design. The display of hierarchical net is simpler and much clearer. It is easier to understand what the Petri net is modelling.



Co...	Sett...
	0
	0
	0
	0
	0
	0
	0
	0
	0

D:\Documents and Settings\Mark Wass\dininglevel3.xml has been Saved

Figure 29. *A single level dining Philosophers Petri net*. This Petri net models the dining philosophers problem as a single level. Places p20, p21, p22, p23 and p24, represent the chopsticks. The rectangular looking parts of the net are philosophers; each has four places and four transitions. For example, p0 represents the first philosopher thinking, t0, picking up the chopstick to the right of the philosopher. p1 represents the state when this chopstick has been picked up. t1 models the action of picking up the second chopstick, this time form the left. This leads to place p2, modelling the philosopher eating. The subsequent places and transition, t2, p3 and t3 model releasing the chopsticks returning the philosopher to the thinking state.

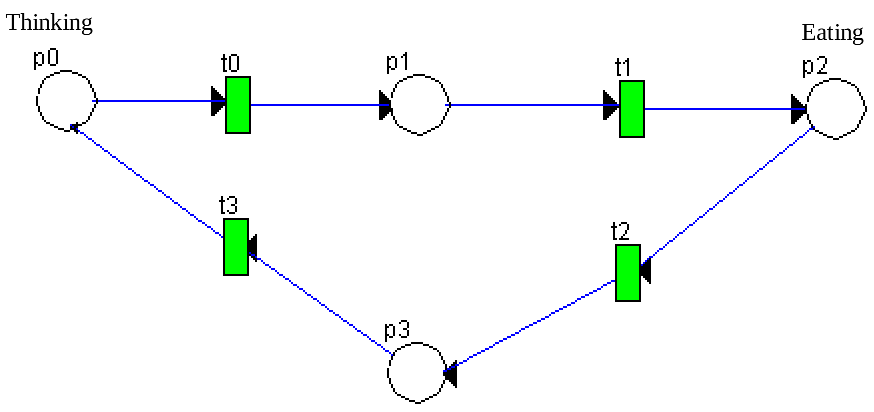
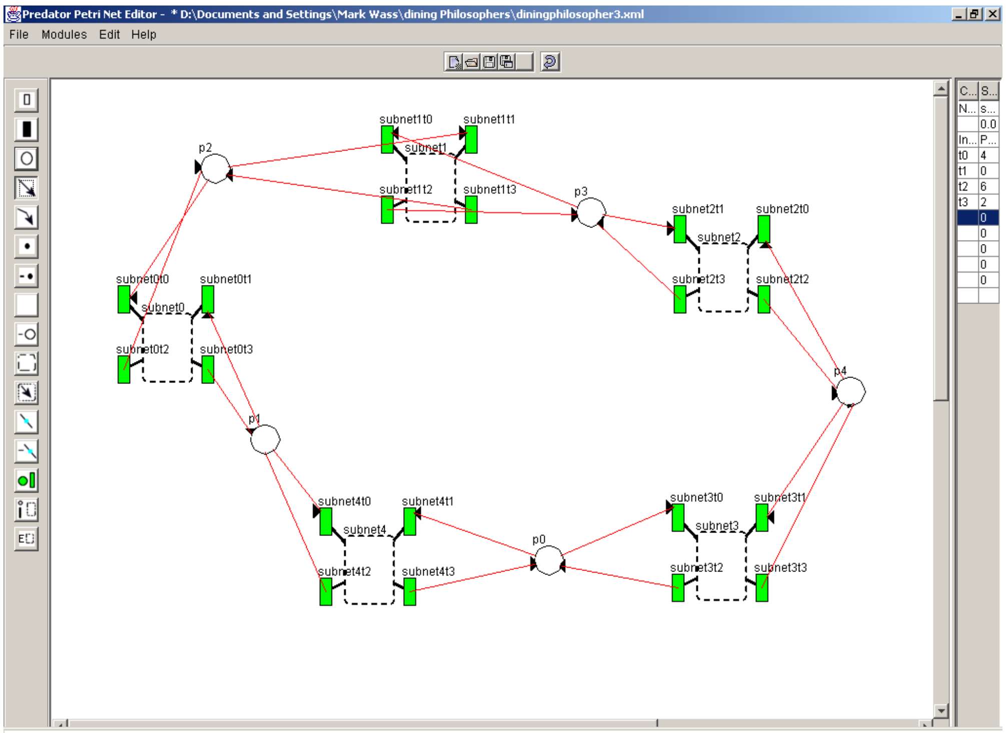


Figure 30. *A Petri net representing a single Dining Philosopher*.



C...	S...
N...	s...
	0.0
In...	P...
t0	4
t1	0
t2	6
t3	2
	0
	0
	0
	0
	0

Figure 31. *Hierarchical Dining Philosophers*. This Petri net presents a hierarchical design of the dining philosophers problem. Each of the philosophers is represented by a subnet, with four interaction point transitions. Transition t0 of each philosopher picks up the chopstick to the right. t1 picks up the chopstick to the left of the philosopher, while t2 and t3 replace the chopsticks to the right and left of the philosopher respectively. Each of the chopsticks is represented by a place. A token on these places represents that the chopstick is available.

This figure demonstrates how Predator represents subnets and their interaction points. Each subnet is a dashed rectangle, with its interaction points placed on the outside of the subnet. The interaction points are distinguished from the components of the higher level net by the thick line connecting them to the subnet. All interaction points are painted green to distinguish them from the other components of a Petri net.

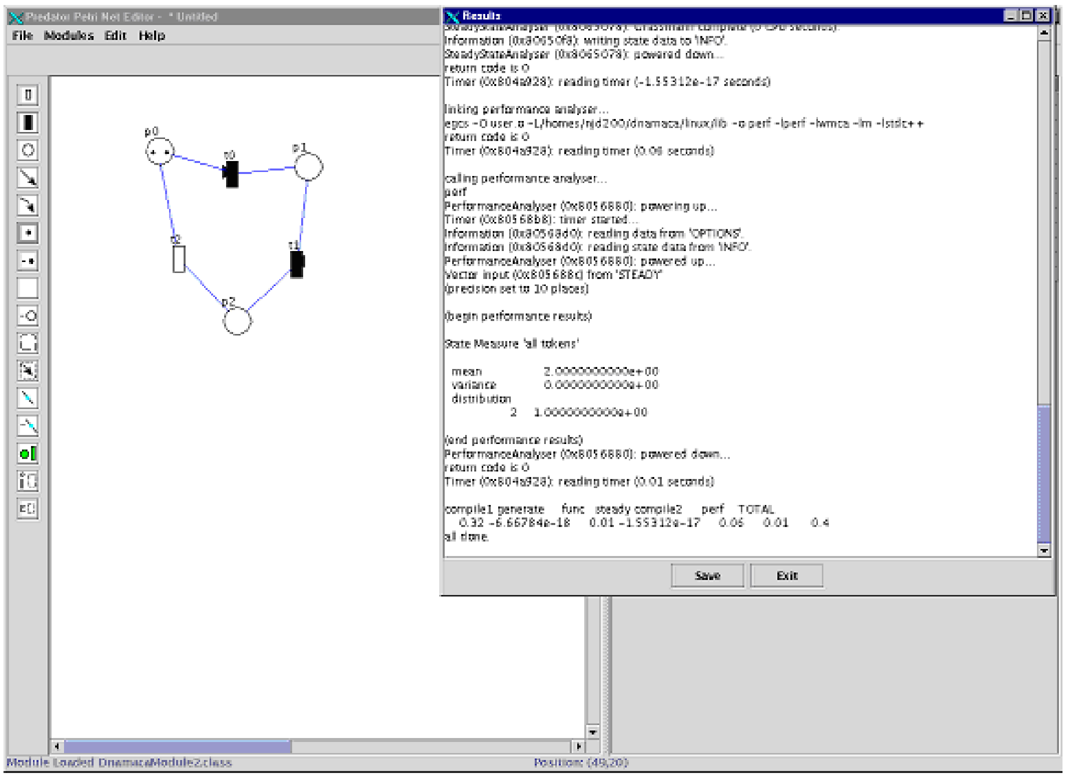
5.3 Open Architecture

The aim of the open architecture was to enable the Petri net editor to dynamically load classes, create instances of them and invoke their methods. The initial step in demonstrating the function of the open architecture was to attempt to load a `testModule` class. This class implemented the `Module` interface, the `runModule()` function was kept simple; it only output a string to the standard output. This class was successfully loaded and executed demonstrating that the open architecture functions correctly. The invariant analysis module was also successfully loaded by the editor and executed on a number of sample Petri nets (A full description of the invariant analysis module follows in the next section).

The open architecture was further tested because another module implementing the `Module` interface has been implemented independently of this project (Dingle 2001). This module interacts with a pre-existing, Markov chain Petri net performance analysis program Dnamaca (Knottenbelt, 1996). To demonstrate the open architecture this module was dynamically loaded and run on a simple Petri net, as shown in figure 32.

5.4 Invariant Analysis Module

Invariant analysis is a well-supported analysis technique present in some Petri net editors. Thus the invariants for common examples of Petri nets such as the readers-writers problem and the producer-consumer problem are well known. This provides examples with which to test the invariant analysis implementation.


Predator Petri Net Editor - * Untitled
File Modules Edit Help
p0
t0
p1
t2
t1
p2
Results
Save
Exit
Module Loaded DnamacaModule2.class
Position: (49,20)

```
Information (0x8065Of8): writing state data to 'INFO'.
SteadyStateAnalyser (0x8065078): powered down...
return code is 0
Timer (0x804a928): reading timer (-1.55312e-17 seconds)

linking performance analyser...
egcs -O user.o -L/homes/njd200/dnamaca/linux/lib -o perf -lperf -lwmca -lm -lstdc++
return code is 0
Timer (0x804a928): reading timer (0.06 seconds)

calling performance analyser...
perf
PerformanceAnalyser (0x8056880): powering up...
Timer (0x80568b8): timer started...
Information (0x80568d0): reading data from 'OPTIONS'.
Information (0x80568d0): reading state data from 'INFO'.
PerformanceAnalyser (0x8056880): powered up...
Vector input (0x805688c) from 'STEADY'
(precision set to 10 places)

(begin performance results)

State Measure 'all tokens'

 mean            2.0000000000e+00
 variance        0.0000000000e+00
 distribution
           2    1.0000000000e+00

(end performance results)
PerformanceAnalyser (0x8056880): powered down...
return code is 0
Timer (0x804a928): reading timer (0.01 seconds)

compile1 generate    func   steady compile2    perf    TOTAL
    0.32 -6.66784e-18    0.01 -1.55312e-17    0.06    0.01    0.4
all done.
```

Figure 32. *The Markov Chain Performance Analysis module loaded into the Predator Petri net Editor and run on a simple Petri net.* This screenshot demonstrates the Markov chain analysis module run on the Predator Petri net editor. This screenshot of the Petri net editor may look different from other because it was run under Linux, as the Markov chain analysis module required this. All other screenshots are taken from Windows execution of Predator

The invariant analysis module was initially tested with the simple Petri net shown in figure 33,A. It correctly identified the P and T invariants of the net as shown in table2. The addition of places, transitions and arcs to this Petri net provided another simple test. Figure 33, parts b-e, display how the Petri net was modified and table 2 presents the results obtained from invariant analysis on them.

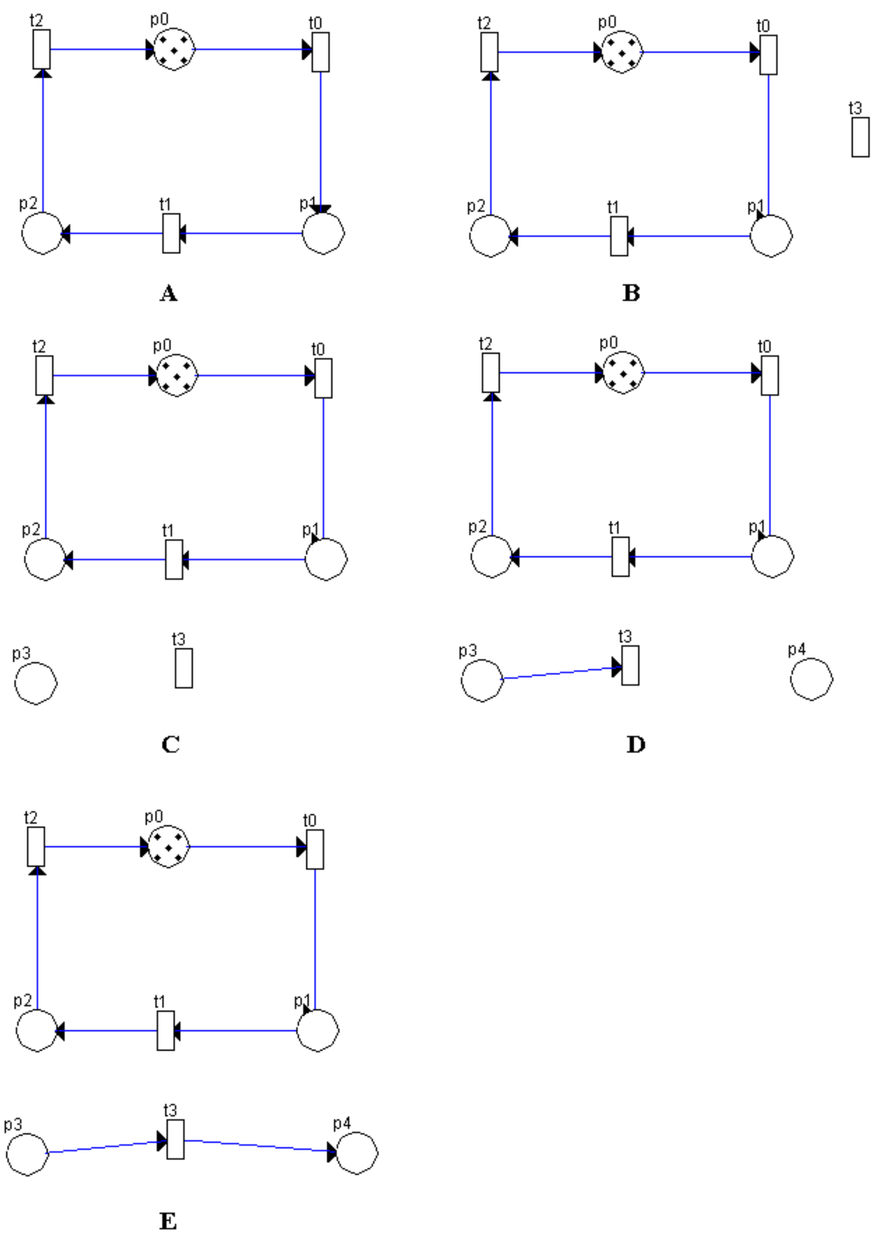
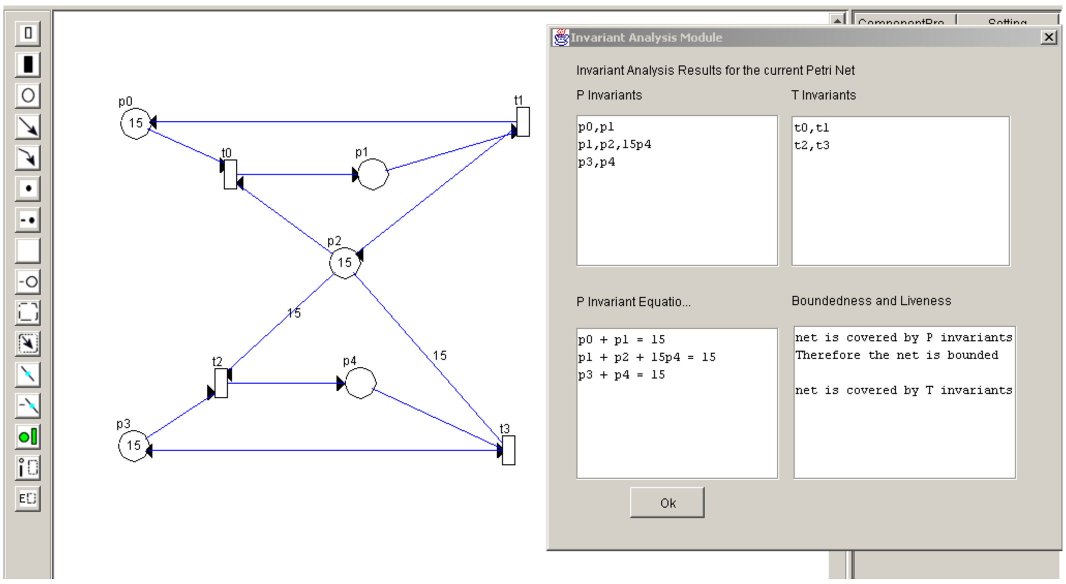


Figure 33. *Simple Petri nets used for Invariant Analysis*. Each of the Petri nets labelled A to E were used to test the invariant analysis module. The expected invariants and results are displayed in Table 2.

The readers-writers problem was also used to test the invariant analysis module. A Petri net for the readers-writers problem was introduced in chapter 2. For this Petri net there are three P invariants, described in Background (2.1.3.2) and shown in table 2. The results obtained from the invariant analysis module are shown in figure 34, they match the expected results. Figure 35 provides a clearer illustration of the dialog used by the invariant analysis module to display its results.



Invariant Analysis Results for the current Petri Net

P Invariants

```
p0,p1
p1,p2,15p4
p3,p4
```

T Invariants

```
t0,t1
t2,t3
```

P Invariant Equatio...

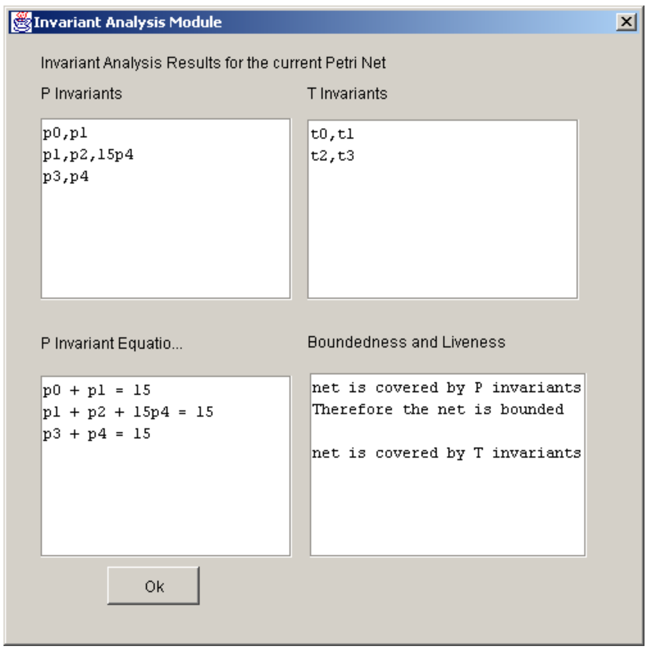
```
p0 + p1 = 15
p1 + p2 + 15p4 = 15
p3 + p4 = 15
```

Boundedness and Liveness

```
net is covered by P invariants
Therefore the net is bounded

net is covered by T invariants
```

Figure 34. *The output from the invariant analysis module for the Readers Writers Problem*. The information is output on four text fields. The P and T invariants are displayed in the top text fields. The bottom left displays the P invariant equations and the bottom right displays data on the boundedness and liveness of the system.



Invariant Analysis Results for the current Petri Net

P Invariants

```
p0,p1
p1,p2,15p4
p3,p4
```

T Invariants

```
t0,t1
t2,t3
```

P Invariant Equatio...

```
p0 + p1 = 15
p1 + p2 + 15p4 = 15
p3 + p4 = 15
```

Boundedness and Liveness

```
net is covered by P invariants
Therefore the net is bounded

net is covered by T invariants
```

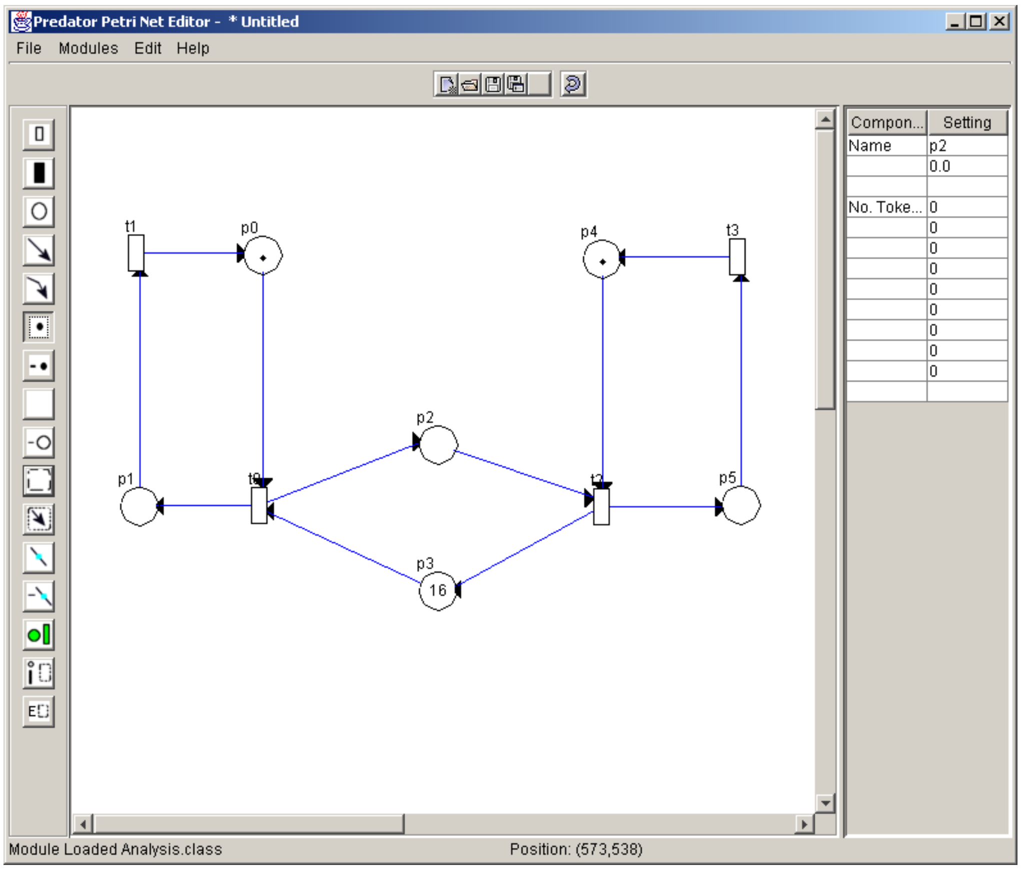
Figure 35. *The InvariantDialog displaying the invariant analysis module results*. This dialog displays the same results for the readers-writers problem as in figure 34.

Petri net	Known Invariants	Results from Analysis module
Simple Net (Fig 33. A)	p0 + p1 + p2 = 5	p0 + p1 + p2 = 5
	t0,t1,t2	t0,t1,t2
Simple Net (Fig 33, B)	p0+p1+p2 =5	p0 + p1+ p2 =5
	t0,t1,t2 t3	t0,t1,t2 t3
Simple Net (Fig 33, C)	p0+p1+p2 =5 p3 =0	p0+p1+p2 =5 p3 =0
	t0,t1,t2 t3	t0,t1,t2 t3
Simple Net (Fig 33, D)	p0+p1+p2 =5 p4 =0	p0+p1+p2 =5 p4 =0
	t0,t1,t2	t0,t1,t2
Simple Net (Fig 33, E)	p0+p1+p2 =5 p3 +P4 =0	p0+p1+p2 =5 p3 +P4 =0
Readers Writers (Fig 34)	p0 + p1 = 15 p1 + p2 + 15p4 = 15 p3 + p4 = 15	p0 + p1 = 15 p1 + p2 + 15p4 = 15 p3 + p4 = 15
	t0,t1 t2,t3	t0,t1 t2,t3
Producer Consumer (fig 36)	p0 + p1 = 1 p2 + p3 = 16 p4 + p5 = 1	p0 + p1 = 1 p2 + p3 = 16 p4 + p5 = 1
	t0,t1,t2,t3,t4	t0,t1,t2,t3,t4

Table 2. *P and T invariants of the tested Petri nets*. This table provides a comparison between the known invariants (provided from texts or from invariant analysis with DaNAMiCs Petri net editor) with those obtained from the Invariant analysis module implemented.

The invariant analysis module was further tested using the Producer consumer problem. Figure 36 displays the expected invariants for the Produce consumer problems (as described in figure 36) and the results obtained from the invariant analysis module.

The invariant analysis module has been tested on a limited number of Petri nets, all of which suggest that the implementation correctly identifies both P and T invariants. However further testing on more complex Petri nets is needed.



Compon...	Setting
Name	p2
	0.0
No. Toke...	0
	0
	0
	0
	0
	0
	0
	0
	0

Module Loaded Analysis.class

Position: (573,538)

Figure 36. *The Producer Consumer Problem*. The left side of this Petri net represents the producer while the right models the consumer. The two components are connected via a buffer that in this case allows up to sixteen items to be produced at once. The buffer ensures that if there a no items produced the consumer cannot consume any and if the buffer is full, then the producer can't produce until the consumer consumes at least one item.

6 Conclusion & Future Work

The two central aims of this project were to design a Petri net Editor that supported hierarchical Petri nets and provided an open architecture to facilitate the dynamic loading of Petri net analysis modules. These aims were met as discussed in the second (6.2) and third (6.3) sections of this chapter. The basic infrastructure required to support these two features was also implemented as discussed in the next section.

6.1 General Features

Predator, the Petri net editor implemented provides the basic features for designing Petri nets that are available in most editors (described in chapter 2). The various examples provided in the results (see chapter five) illustrate the ability to design Petri nets with the Predator Petri net editor.

Support for hierarchical Petri nets and an open architecture were placed ahead of additional editing features such as zooming and printing. Obviously additional features such as these will improve the ability to design nets with the editor but it would not be difficult to add them in the future.

6.2 Hierarchical Petri Nets

The support for Hierarchical Petri nets provided by other Petri net editors is very basic and difficult to navigate the various levels of the Petri net (discussed in chapter 2). This project set out to provide improved support for hierarchical Petri nets, which was to be achieved primarily by making interaction points between different levels of a hierarchical Petri net explicit and by providing a simple way to navigate between the many levels of a hierarchical Petri net. The Predator Petri net editor has provided improved support for hierarchical Petri nets; interaction points can easily be selected/deselected, and are also visible on the outside of subnets. The implementation also enables single mouse clicks to enable moving between different levels of a hierarchical Petri net.

The dining philosophers example (results, chapter 5) demonstrates the advantage of designing complex systems using a compositional approach, especially if the system contains repetitive units, like the philosophers in this example.

Predator currently limits the number of interaction points a subnet may have to eight. This was done to simplify the positioning of interaction points on the outside of subnets but could be a limitation in the design of complex systems. An implementation providing unlimited interaction points could be added in the future.

This project has only explored the ability to design, edit and store hierarchical Petri nets. This has demonstrated advantages in the design of complex systems but much greater benefits could come from the use of compositional approaches to testing and analysis of Petri nets. Process algebras exploit their compositional nature in analysis techniques. It may be possible to adopt them for hierarchical Petri net analysis. This provides an area of future research. The predator Petri net editor could be a useful utility for such research as it provides support to design hierarchical Petri nets and the open architecture allows the dynamic loading of user defined analysis modules.

6.3 Open Architecture

The aim of incorporating an open architecture into the design of the Petri net editor was to provide users the power to implement, load and use analysis techniques to suit their needs. All existing Petri net tools rely upon built in analysis techniques, which limit the user to use only those provided with a particular editor or to use multiple editors to provide all the analysis capabilities they require.

The open architecture implemented using Java Reflection, fulfils it aims. This has been demonstrated by the ability to dynamically load and execute two different analysis modules, both offering different types of analysis techniques. One of these modules was implemented separately from this project (Dingle 2001). This module demonstrates that users will be able to implement their own modules. Further as this module modifies a pre existing analysis program Dnamaca, it demonstrates that new implementations are not required but only modification to enable existing programs to parse the file format and implement the Module interface.

6.4 Invariant Analysis Module

In the results (chapter 5), the invariant module was demonstrated to function correctly for a limited number of Petri nets. It is not possible to verify that the implementation will correctly analyse every Petri net, but the simple tests, suggest that the module does perform invariant analysis correctly. The implementation of this module was an important feature of demonstrating the capabilities of the open architecture as discussed above.

6.5 File Format

The file format developed is based upon the Petri Net Markup Language (Jüngel et al., 2000). The file format has been modified to support Stochastic and Generalised Stochastic Petri nets, and further to support hierarchical Petri nets. Two different file formats were used to provide hierarchical Petri net support; a flattened form saving a hierarchical net as a single file and the other saving each net as a separate file. The flattened structure was added to provide the complete structure of a hierarchical Petri net in a single file, so that editors or analysis modules that do not support hierarchical Petri nets can parse a hierarchical Petri net as if it consisted of single level.

6.6 Concluding Remarks

The Predator Petri net editor has met the aims of this project. The open architecture is a powerful feature of Predator enabling users to perform whatever type of Petri net analysis they wish to. There is also much that can now be explored in the field of Hierarchical Petri nets, whose structure could be exploited in system design and analysis.

7 Bibliography

7.1 Book & Journal References

Ajmone-Marsan, M., Conte, G. & Balbo, G. (1984) *A Class of Generalised Stochastic PetriNets for the Performance Evaluation of Multiprocessor Systems*. ACM Transactions on Computer Systems, **2**:93-122.

Bause, F. & Kritzinger, P.S. (1995*) Stochastic Petri nets – An Introduction to the Theory*.

Ciardo G. & Trivedi, K.S. (1993) *A decomposition approach for stochastic reward net models*. Performance Evaluation **18**:37-59.

D'Anna, M. & Trigila, S. (1988) *Concurrent System Analysis Using Petri nets: An Optimised Algorithm for Finding Net Invariants.* Computer Communications, **11**, 215—220.

Deitel & Deitel (2001) *Java How To Program.* 3rd Edition Prentice Hall.

Dingle (2001) *The production of the extensible Petri net Editor/Animator – "Medusa"*. Master Thesis, Imperial College.

Sundsted, T. (1996) *Examining HotSpot, an object-oriented drawing program*. Java World , December 1996

Meeting on XML/SGML based Interchange Formats for Petri nets (2000).

Hillston., J. (1996) *A Compositional Approach to Performance Modelling*. Cambridge University Press.

Jüngel, M., Kindler, E. & Weber, M. (2000) *Towards a Generic Interchnage Format for Petri nets – Position Paper*. http://www.informatik.hu-berlin.de/top/pnml/

Magee, J. & Kramer, J. (1999*) Concurrency StateModels & Java Programs*. Wiley.

Mailund & Mortensen (2000) *Separation of Style and Content with XML in an interchange format for higher level Petri nets*.

Molloy, M.K. (1982) *Performance analysis using stochastic Petri nets*. IEEE

Transactions on Computers, **31**:913-917.

Knottenbelt,W.J. (1996) *Generalised Markovian Analysis of Timed Transition Systems*. Masters Thesis, University of Cape Town.

Valente, A. & Gribaudo, M. (2000)*Two level interchange format in XML for Petri nets and other graph-based formalisms.*

Woodside, C.M. & Li., Y. (1991). *Performance Petri Net Analysis of Communication Protocol Software by Delay-Equivalent Aggregation.* Proceedings of the 4th International Workshop on Petri nets and Performance Models, IEEE Computer Society Press, 64-73.

7.2 URL references

Websites that were particularly useful references are listed below.

Carl Petri's Homepage - http://www.informatik.unihamburg.de/TGI/mitarbeiter/profs/petri_eng.html

Petri net World - http://www.daimi.au.dk/PetriNets/

The Petri Net Tool Database - http://www.daimi.au.dk/PetriNets/tools/db.html

The Java Tutorial http://www.java.sun.com/tutorial/

www.petrinets.org

7.3 Tool References

A number of Petri net Editor tools were tested and some are referenced in the text. The URL of these tools' homepages are given below.

CPN http://www-src.lip6.fr/logiciels/mars/CPNAMI/

DaNAMiCS http://www.cs.uct.ac.za/Research/DNA/DaNAMiCS/

HPSim http://home.t-online.de/home/henryk.a/petrinet/e/hpsim_e.htm

INA http://www.informatik.hu-berlin.de/~starke/ina.html

THORN/DE http://www.offis.uni-oldenburg.de/projekte/dns/project_dns.htm

Model checking kit http://wwwbrauer.in.tum.de/gruppen/theorie/KIT/

Visual Object Net++
http://www.systemtechnik.tu-ilmenau.de/~drath/visual_E.htm

8 Appendix

Source code for the Predator Petri net editor will be available online at http://mark.wass.com/Petrinets.html .

9 User Guide

This chapter describes how to use Predator Petri Net Editor to design and edit Petri nets. A number of the examples use the readers-writers problem.

9.1 Getting Started

9.1.1 Loading

Predator is provided in a JAR file called PEditor.jar. To run on command line on either Windows or Linux type e.g. c:\java –jar PEditor.jar, or use the batch file provided.

9.1.2 Initial Screen

Figure 9.1 demonstrates Predator's user interface. From here Petri nets can be designed, edited and loaded.

Components of the User interface

drawing area	- where Petri nets are displayed.
EditorToolBar	– where editing tools are selected.
editTable	- used to edit properties of Petri components
FileToolBar	- used for file operations, e.g. loading and saving files.
Menu Bar	- menus for opening files, loading modules and help
Status bar	- displays information about operations such as file opening and saving
Position Bar	- displays the position of the cursor on the screen

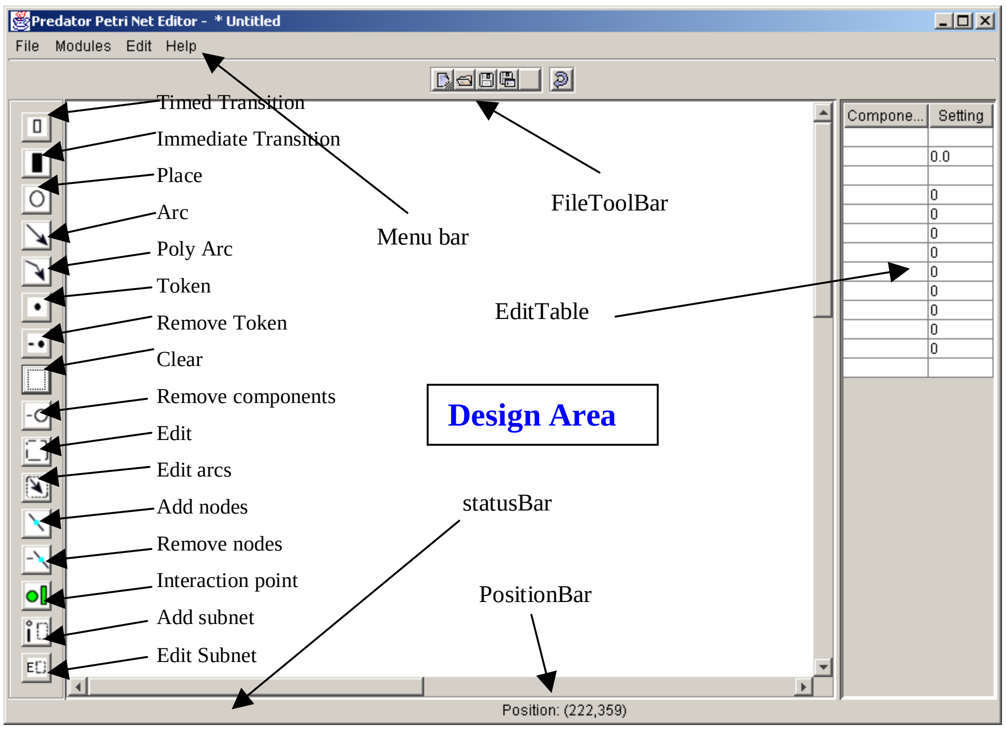


Figure 9.1. *The Predator Petri Net Editor Tool User Interface*. The EditToolbar provides editing options. The File Toolbar provides buttons to open and save files. The editTable displays editable features of selected Petri net Components. The Design area is the area where Petri nets are displayed. The position of the cursor is displayed in the position bar. User information is displayed in the status bar.

9.2 Designing A simple Petri Net

To add components to a Petri Net the relevant button on the EditToolBar is selected and the mouse clicked at the location the component is to be added, as displayed in Figure 9.2. For example select Transition and add one to the display.

To add Places, first select the place icon, and like transitions click on the design area where places are to be added. Figure 9.3 displays the editor with places added.

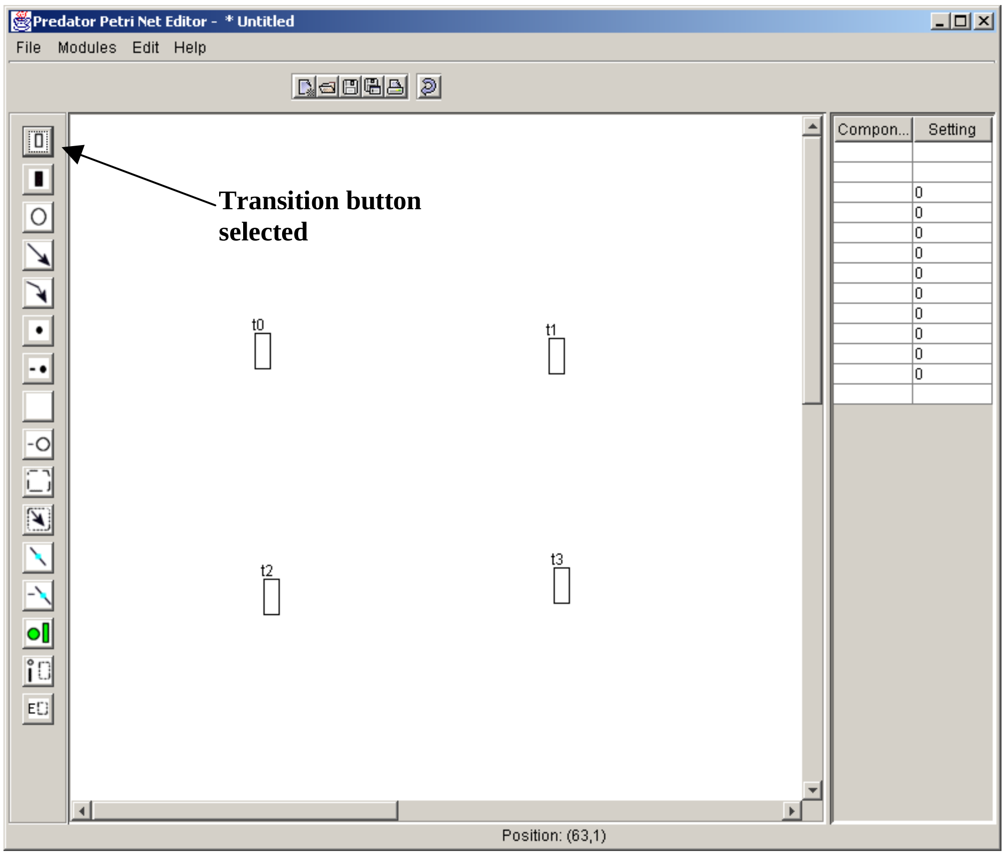


Figure 9.2. *Transitions added to a Petri Net*. The Transition icon from the editToolBar is selected as shown. The mouse is then clicked on the design area where transitions are to be added.

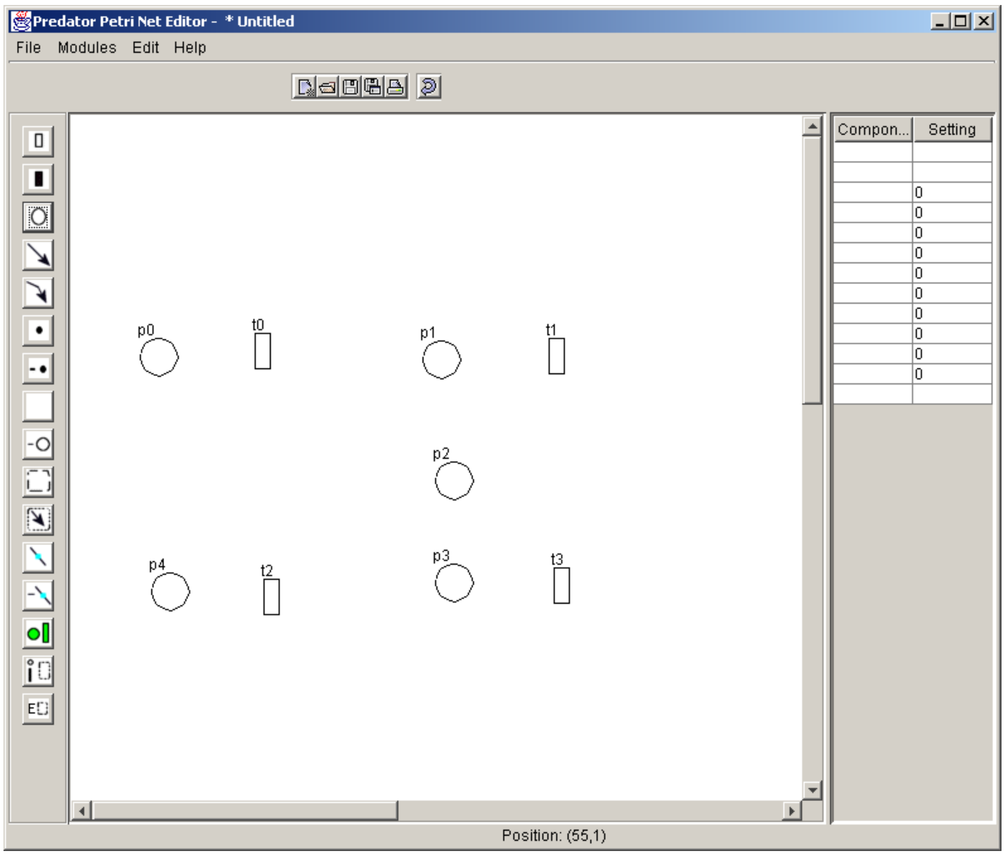
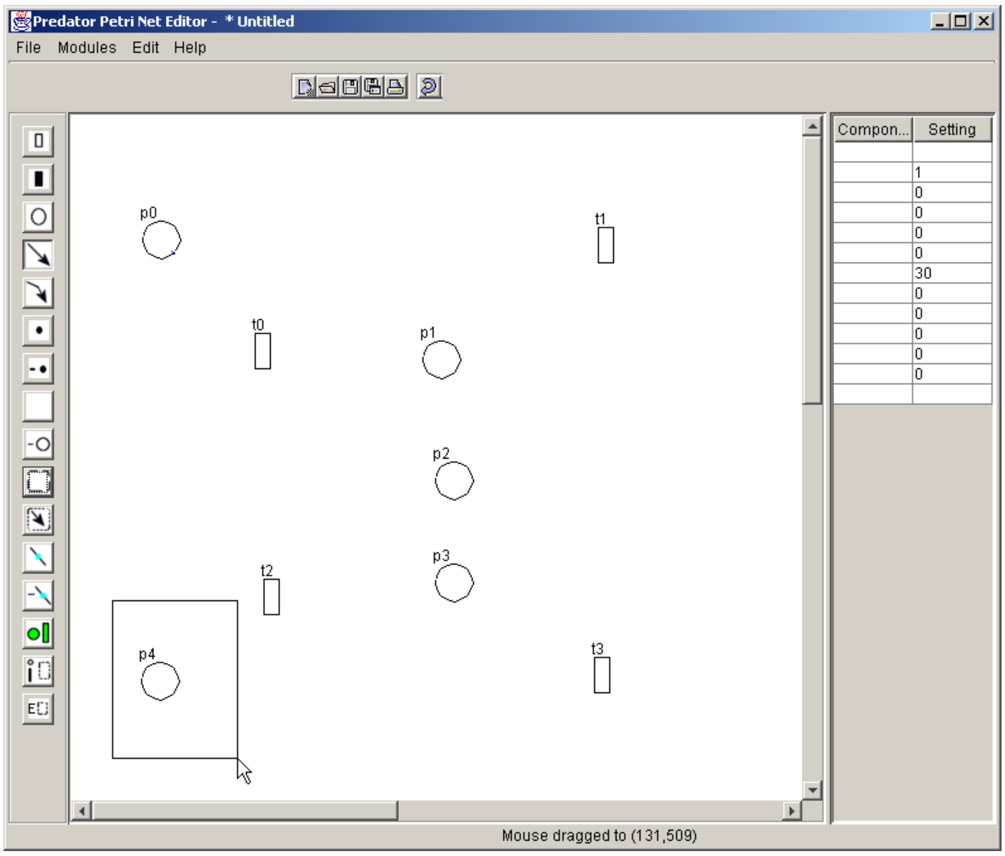


Figure 9.3. *Places added to the Design Area.*

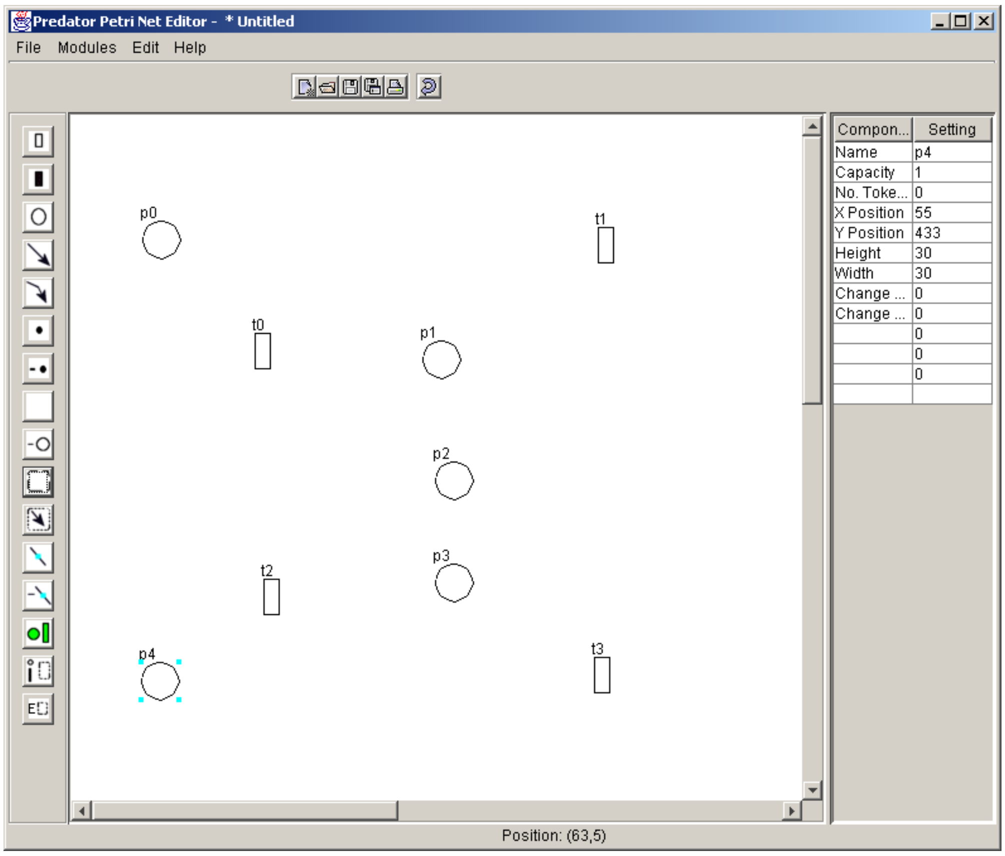
The position of components can be edited. To do this first select the edit icon from the editToolBar then click and drag the mouse to cover the components that are to edited, as shown in the figure 9.4. Once the mouse is released edit points are drawn at the corners of the selected components, as shown in the figure 9.5.



Compon...	Setting
	1
	0
	0
	0
	0
	30
	0
	0
	0
	0
	0

Mouse dragged to (131,509)

Figure 9.4. *Editing*. A place is being selected for edit. The edit icon has been selected and the mouse has been dragged over the place that is to be selected. The rectangle shows the selected area.



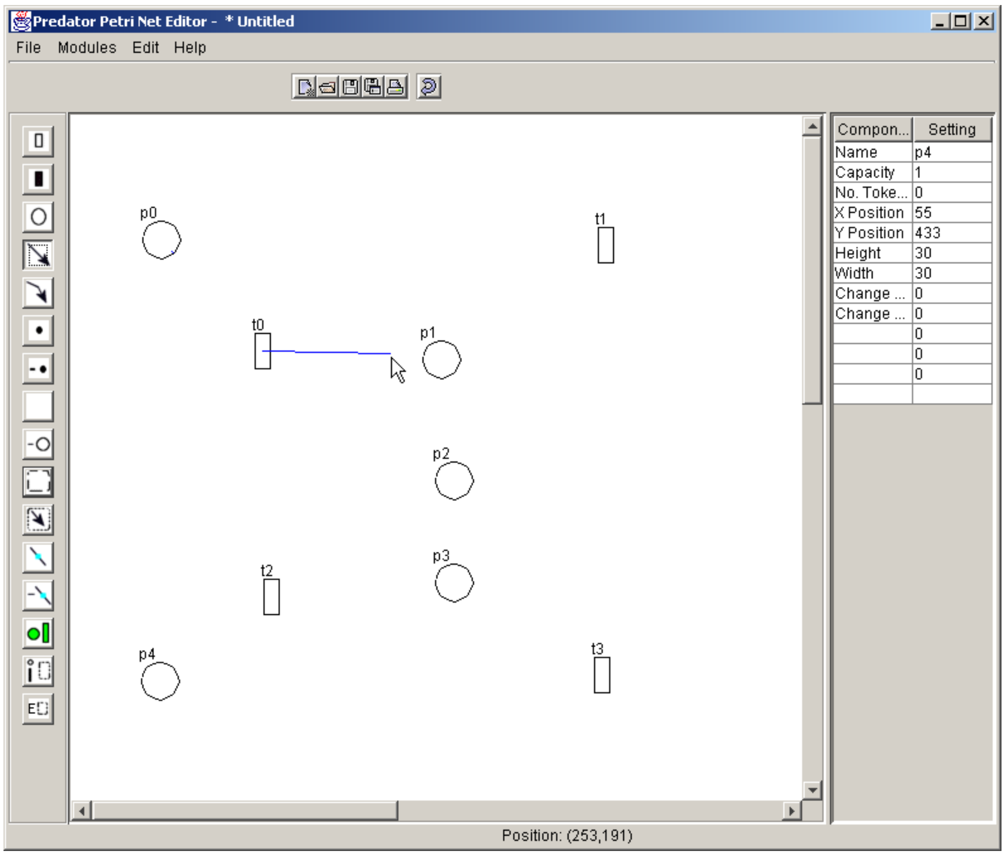
Compon...	Setting
Name	p4
Capacity	1
No. Toke...	0
X Position	55
Y Position	433
Height	30
Width	30
Change ...	0
Change ...	0
	0
	0
	0

Position: (63,5)

Figure 9.5. *Selected Components*. The selected components are painted with editpoints. Place p4 is selected.

9.2.1 Adding Arcs

Arcs connect places and transitions. To add an arc select the arc option and click on place or transition that the arc is to start at. Then drag the arc to the place or transition at which it is to finish (see figures 9.6 & 9.7).



Compon...	Setting
Name	p4
Capacity	1
No. Toke...	0
X Position	55
Y Position	433
Height	30
Width	30
Change ...	0
Change ...	0
	0
	0
	0

Figure 9.6. *Adding an arc.* Then mouse is pressed on the place or transition that it is to start and dragged over to the place/transition where it is to finish.

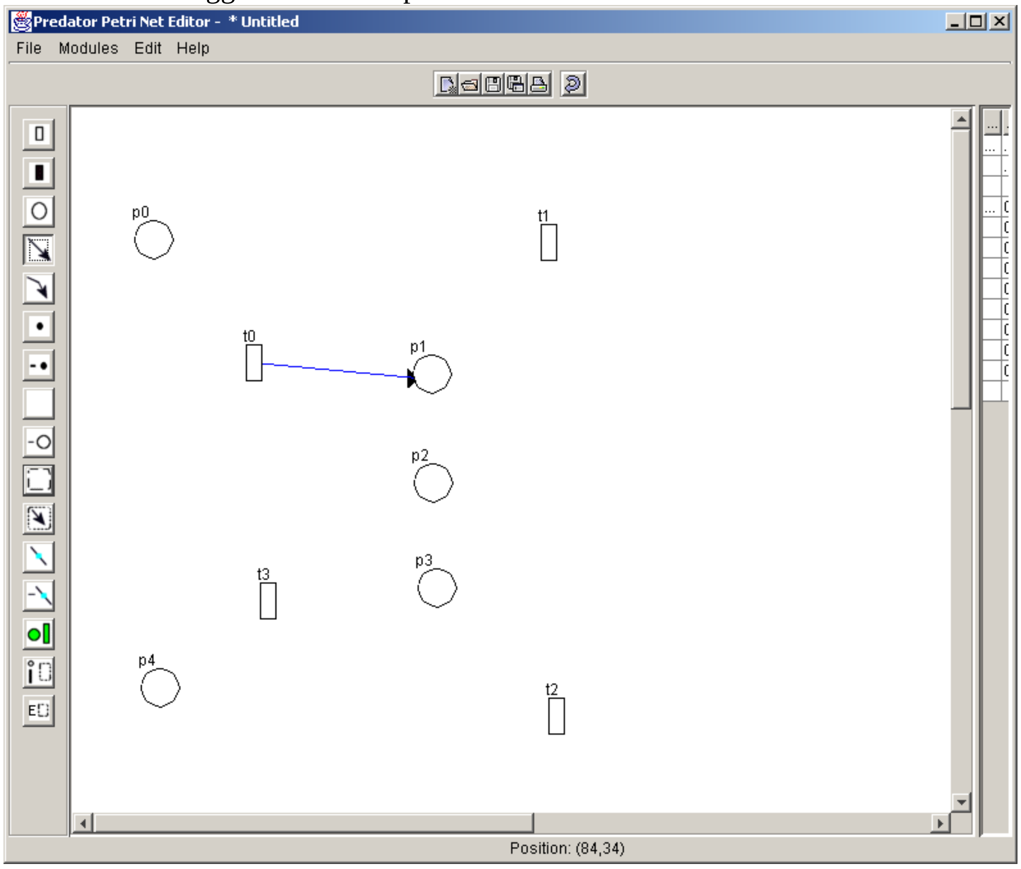


Figure 9.7. *Connecting an Arc to an input Place/transition.* The arc was added by releasing the mouse over the place p1.

9.2.2 Tokens

Tokens can be added to places. To add a tokens to places, select **token** from the **editToolBar.** Click on a place to add a token, repeat this for the number that are to be added (see figure 9.8).

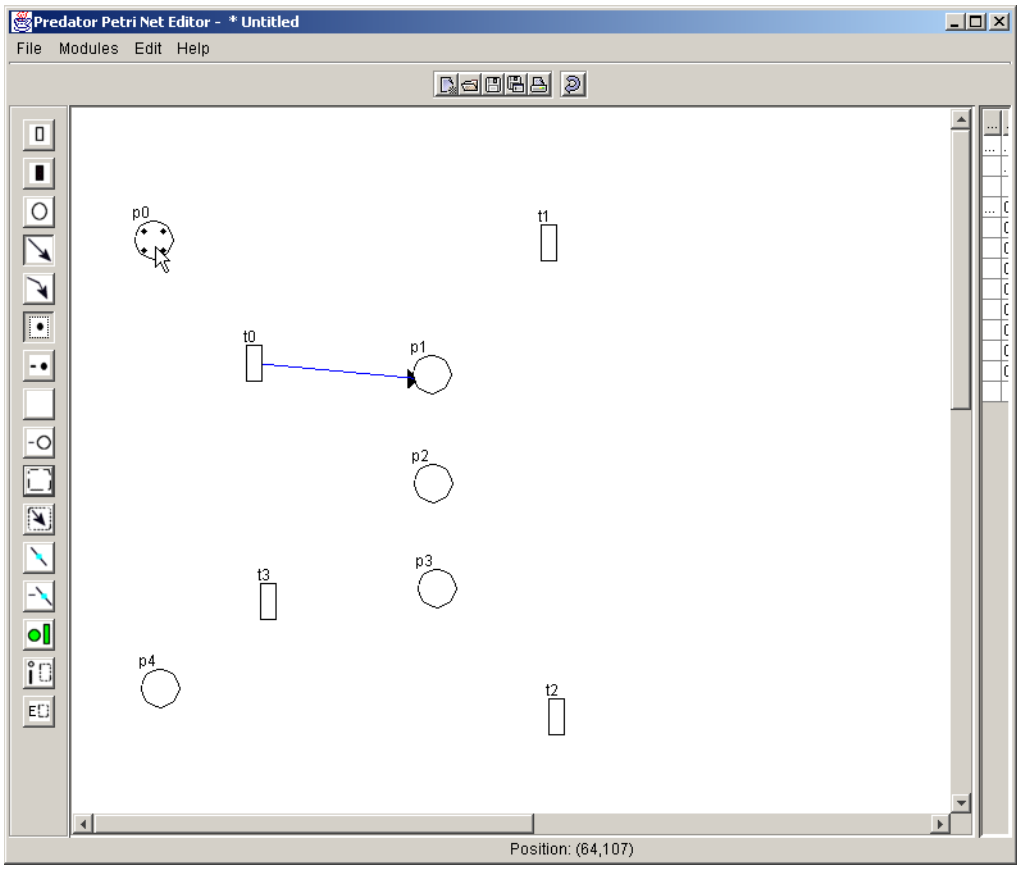
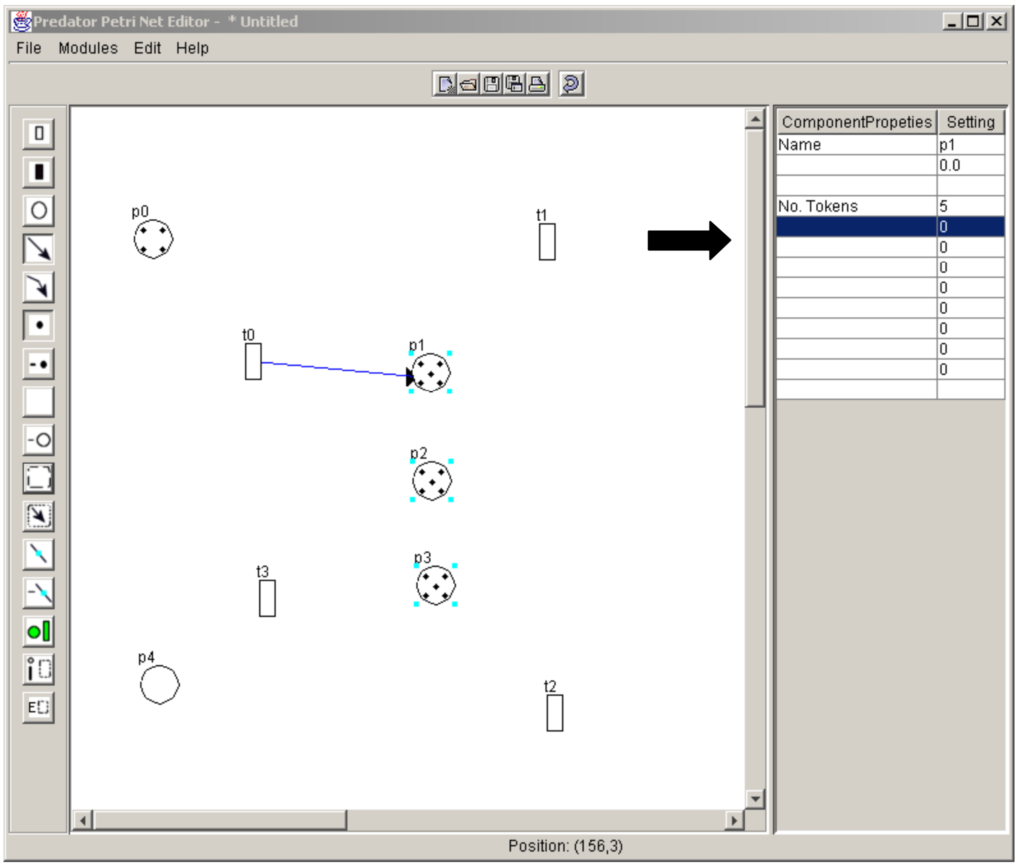


Figure 9.8. *Adding Tokens to Places.* Four tokens have been added to place p0. The token icon on the editToolbar is selected, tokens are added by clicking on a place.

Tokens can similarly be removed by selecting the **removeTokens** icon from the **editToolBar,** and clicking on places.

Token number can also be changed when **edit** is selected from the **editToolBar.** The number of tokens on multiple places can be changed at the same time. First the places must be selected. The token number on the

editTable can then be used to set the number of tokens on all the selected places, as shown in the figure below (9.9).



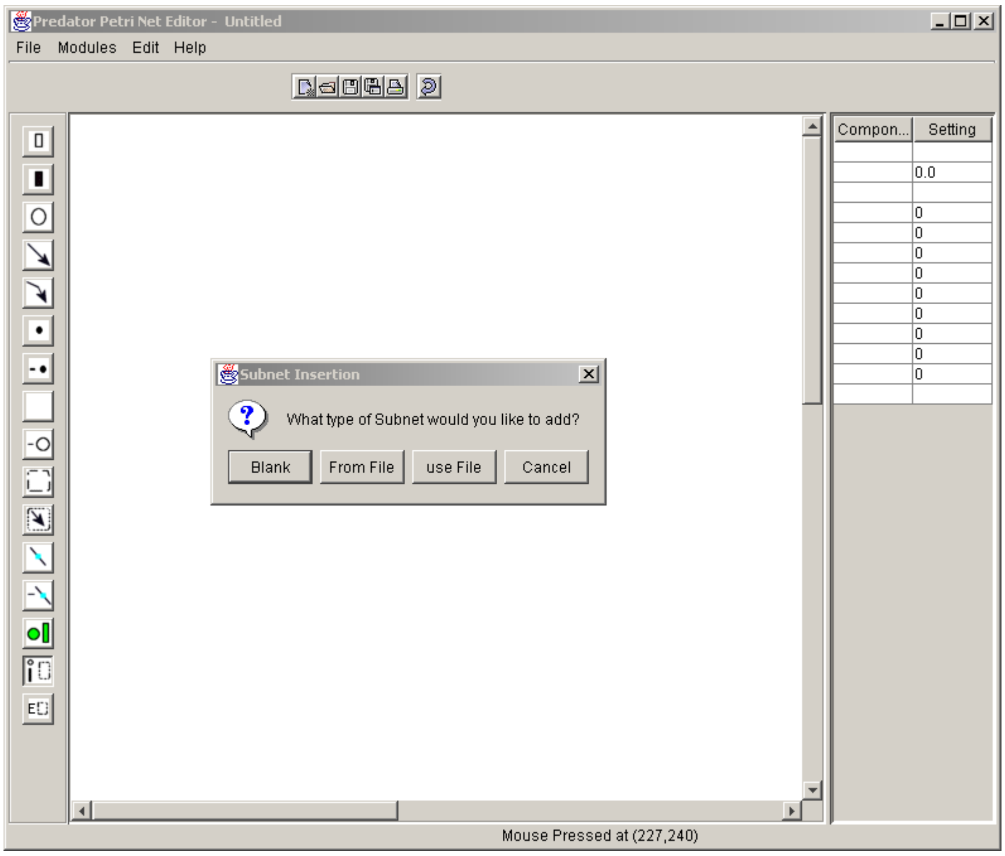
ComponentPropeties	Setting
Name	p1
	0.0
No. Tokens	5
	0
	0
	0
	0
	0
	0
	0
	0

Figure 9.9. *Adding Tokens to multiple Places.* The edit icon from the editToolBar has been selected. The mouse was pressed and dragged to select places, p1,p2 and p3. The editable (indicated above by large arrow) was used to set the token number of these places to 5.

9.3 Subnets

9.3.1 Adding Subnets

To add a subnet select the subnet icon from the **edittoolbar**. Then click on the screen to add a subnet. A dialog will appear (see figure9.10) giving the option to add a subnet from a file, use a file, add a blank subnet or cancel the addition.



What type of Subnet would you like to add?

Compon...	Setting
	0.0
	0
	0
	0
	0
	0
	0
	0
	0
	0

Mouse Pressed at (227,240)

Figure 9.10. *Adding a Subnet.* Subnets can be added by selecting the subnet icon from the editToolBar. The dialog is used to identify how the subnet should be added. A blank subnet can be added or one from a file. If from file is selected then when the Petri Net is saved the subnet will be saved to a different file, not the source file. When use file is selected, the subnet will be saved back to the source file.

The options from file and use file then present FileChooser dialogs (figure 9.11) to select the file to use for the subnet. The difference in these two options occurs when the Petri Net is saved. From file, saves the subnet as a different file, where as use file, saves the subnet using the file selected. Once selected a subnet is added as shown in figure 9.13.

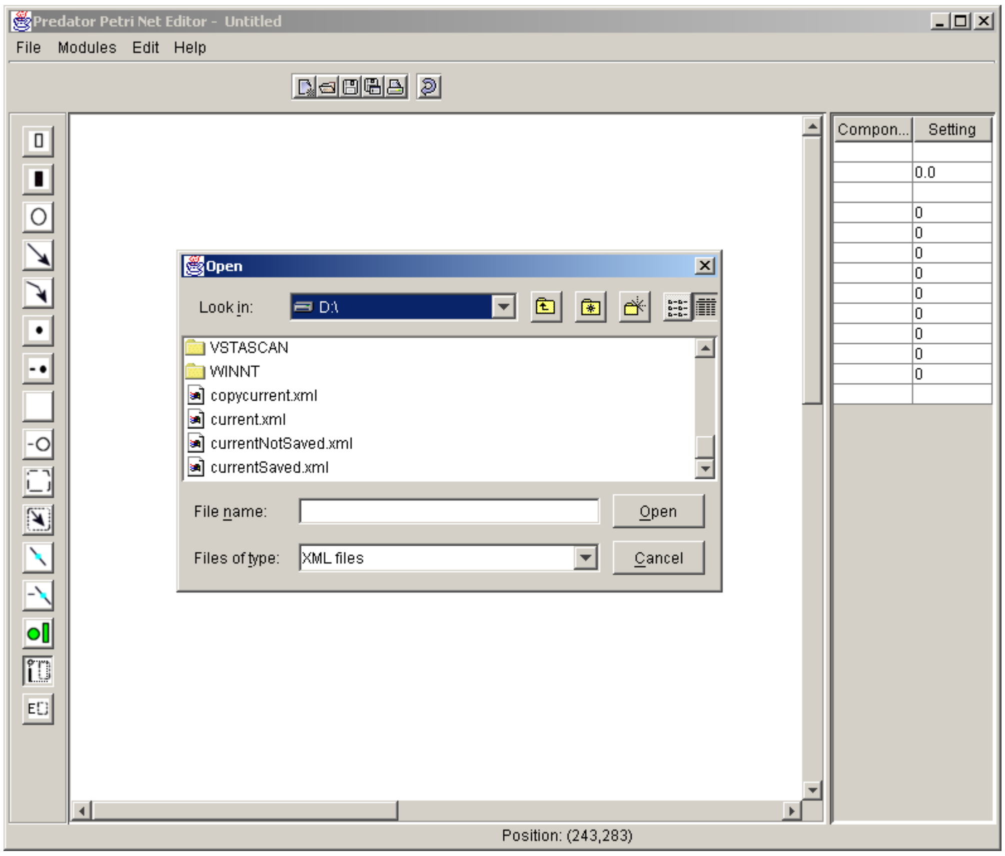


Figure 9.11. *Choosing a subnet*.When from file or use file are selected, a file chooser dialog is displayed to obtain the name of the file to use for the subnet.

9.3.2 Subnet Interaction Points

Transitions and places of a Petri net can be set as interaction points by selecting the interaction point icon from the **edittoolbar.** The interaction point status of places and transitions can then be toggled by clicking on them. Interaction points are filled green to distinguish them.

A subnets interaction points are displayed on the outside of the subnet. This enables arcs from the current level of the Petri net to be connected to the interaction points and thus interact with the subnet.

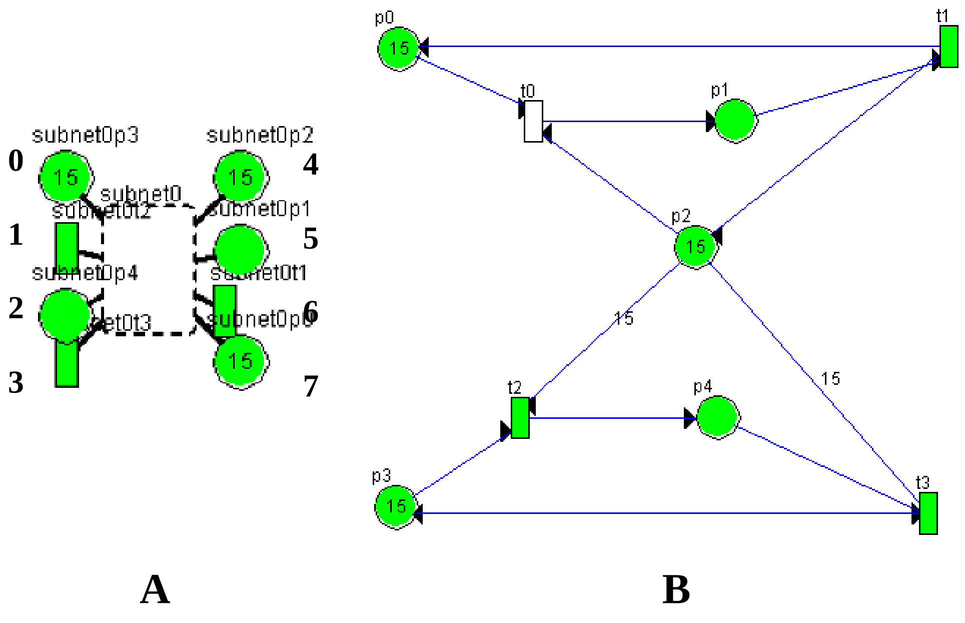
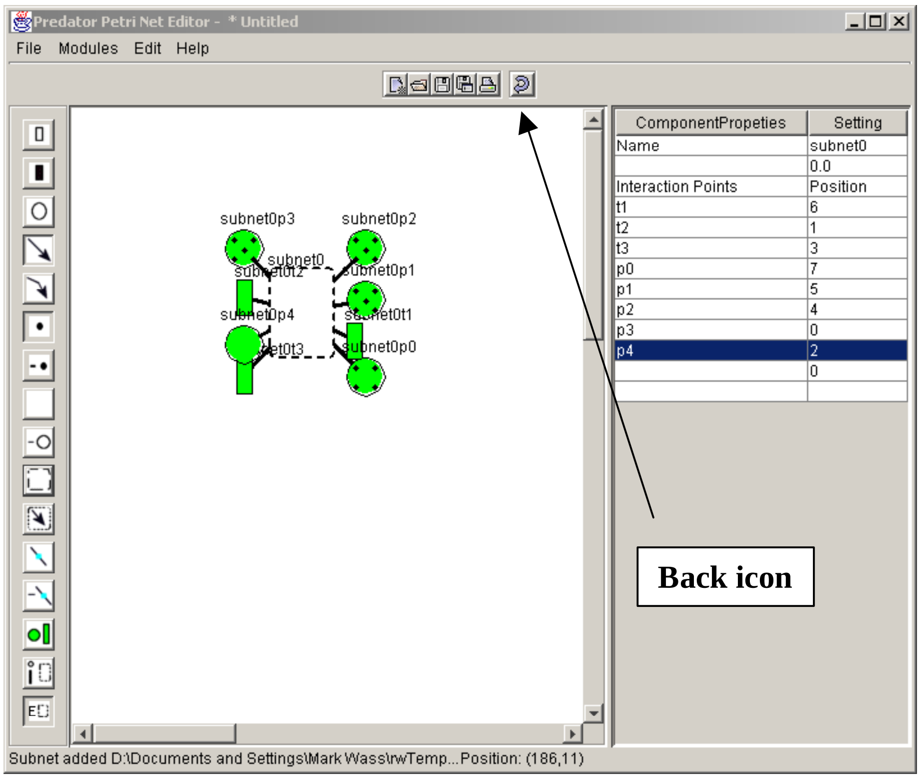


Figure 9.12. *Subnet interaction Points.* A) How a subnet is displayed in a higher level net. The interaction points are displayed at set positions, on the outside of the subnet. B) ThePetri Net that makes up the subnet.

9.3.3 Editing a Subnet

Subnets can be edited by selecting the editSubnet icon from the **editToolBar**. To move the position of interaction points click on one of the interaction points. The positions of each of the interaction points are then displayed in the **editTable,** they can be set to any of eight locations from 0 to 7. The numbers in the figure above indicate the numbers of each position.



ComponentPropeties	Setting
Name	subnet0
	0.0
Interaction Points	Position
t1	6
t2	1
t3	3
p0	7
p1	5
p2	4
p3	0
p4	2
	0

Subnet added D:\Documents and Settings\Mark Wass\rwTemp...Position: (186,11)

Figure 9.13. *Editing the position of Interaction Points.* The editSubnet icon from the editToolbar has been selected. To modify the locations of the interaction points, one of them must be clicked on. The positions of the interaction points are then displayed in the editTable. Position values ranging from 0 to 7 can be enetered. If there is already an interaction point at the chosen position, then the interaction points are swapped.

To edit the contents of a subnet, click on the subnet, this will display the contents of the Petri net for editing. If this level contains subnets they can also be clicked on to move further down the hierarchy. The **back** icon (indicated in the figure above) is used to move back up the hierarchy.

9.3.4 Saving Hierarchical Petri Nets

Hierarchical Petri nets can be saved using a separate file for each subnet, or the whole hierarchical net can be saved to a single file. To save Petri nets using a single file, **save** is selected from the **File** menu. Otherwise to use a separate file for each subnet, **saveSeparate** is selected(figure 9.14).

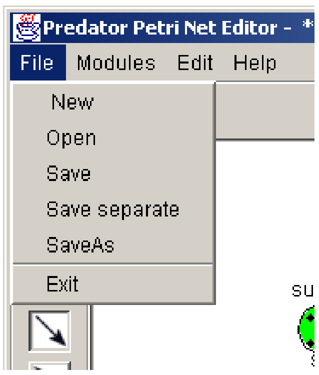
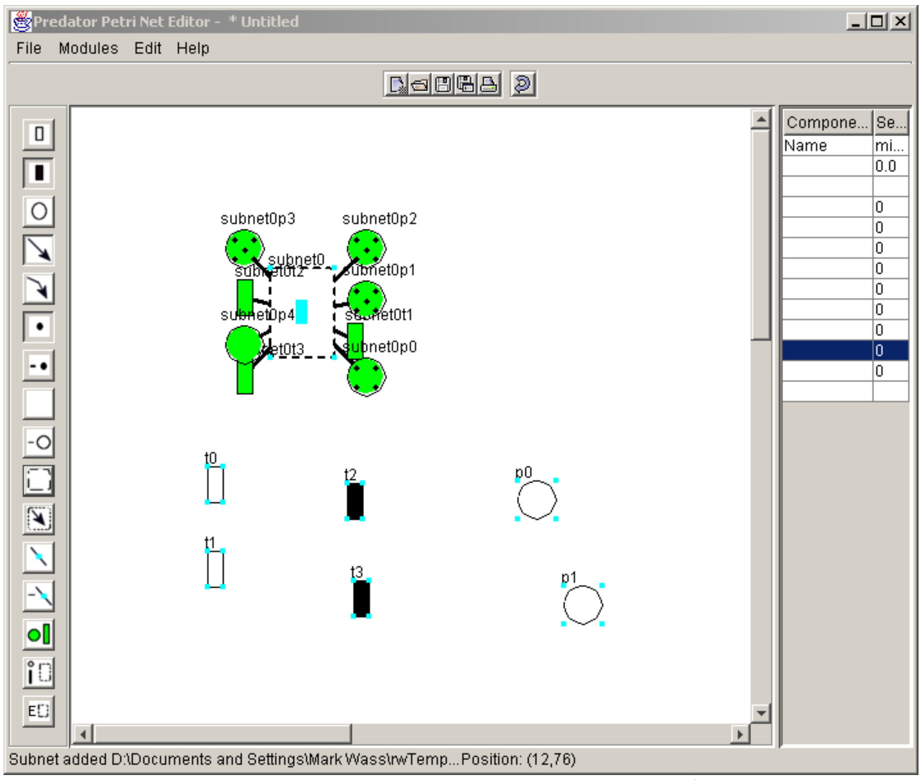


Figure 9.14. *Saving Hierarchical Petri nets*. To save a hierarchical Petri Net as a set of files choose **Save separate** from the file menu. To save as a single 'flattened file' choose **Save**. To save a file under a different name choose **saveAs**.

9.4 Editing Petri Net Components

9.4.1 Editing Transitions, Places and Subnets

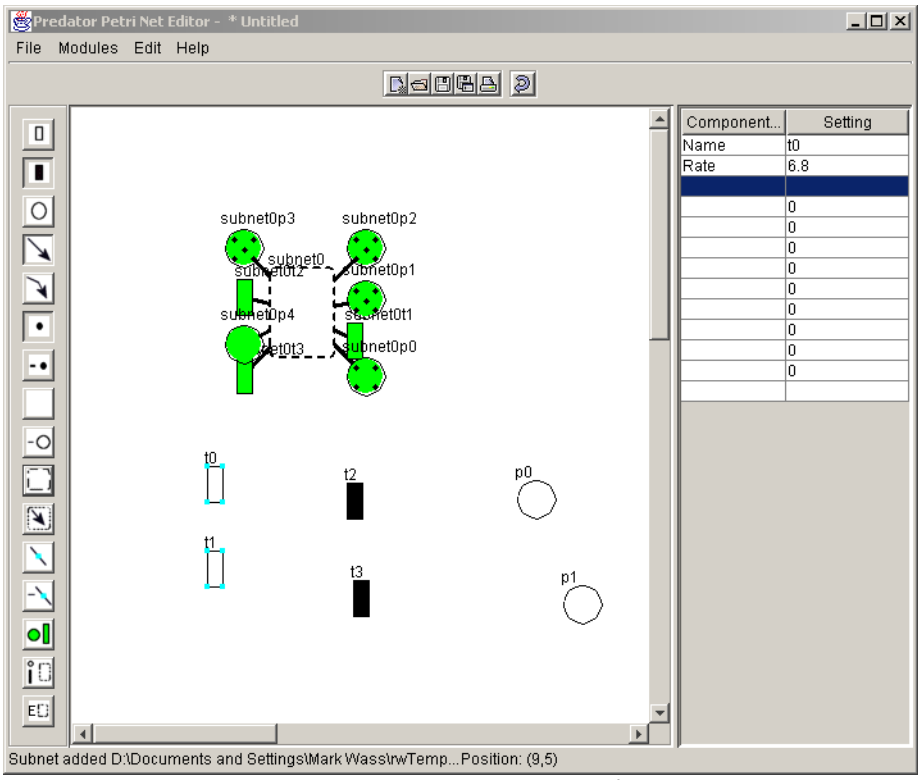
Selecting the edit icon (on the editToolBar), enables the positions of transitions, places and subnets to be changed. It also enables the renaming of transitions and places. To edit one of these components click on the mouse and drag to completely cover the objects that are to be selected. To move them, press the mouse on one of the selected objects and drag it to the desired position, see figure 9.15. Some examples of selection have been shown in earlier diagrams (setting token number of multiple places).



Subnet added D:\Documents and Settings\Mark Wass\rwTemp...Position: (12,76)

Figure 9.15. *Multiple Selection*. All the components of this Petri Net have been selected. They can be moved by pressing the mouse on top of one of the selected components and dragging to a new position.

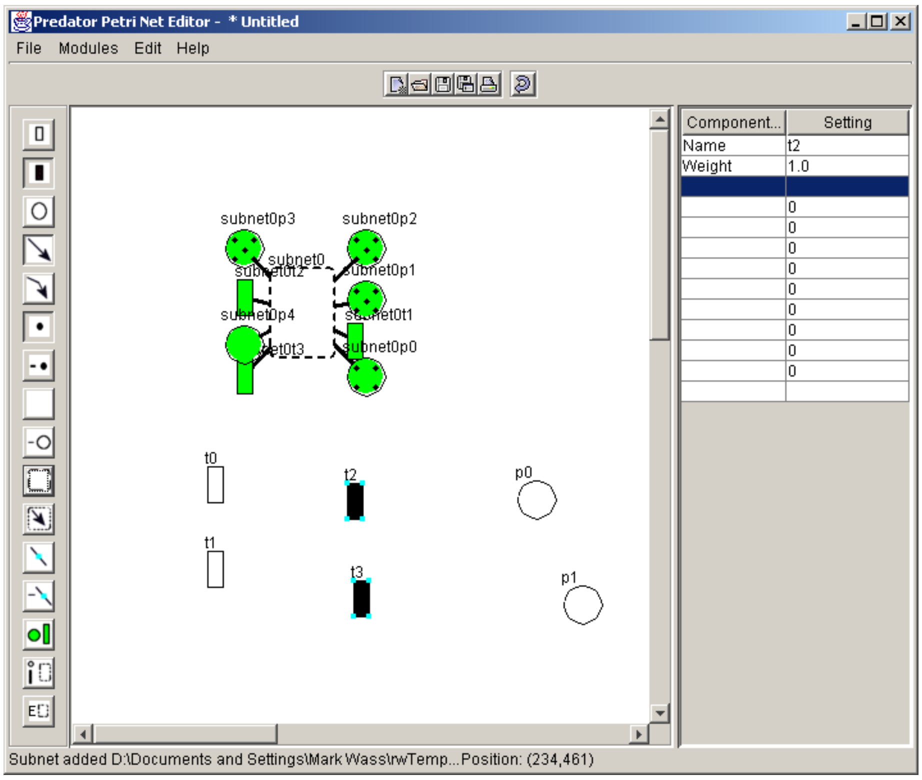
The figure above demonstrates that Petri Net components can be selected together. If only places or a single type of transition are selected then further properties can be edited in the editTable. For timed transitions their firing rate can be edited. Immediate transitions can have their weight edited. Editing of these properties is shown in the following two figures (figures 16 & 17).



Component...	Setting
Name	t0
Rate	6.8
	0
	0
	0
	0
	0
	0
	0
	0
	0

Subnet added D:\Documents and Settings\Mark Wass\rwTemp...Position: (9,5)

Figure 9.16. Selecting Timed Transitions. In this figure two timed transitions have been selected. Entering a new rate in the rate column of the EditTable can alter their firing rate. Like multiple selection their positions can also be changed.



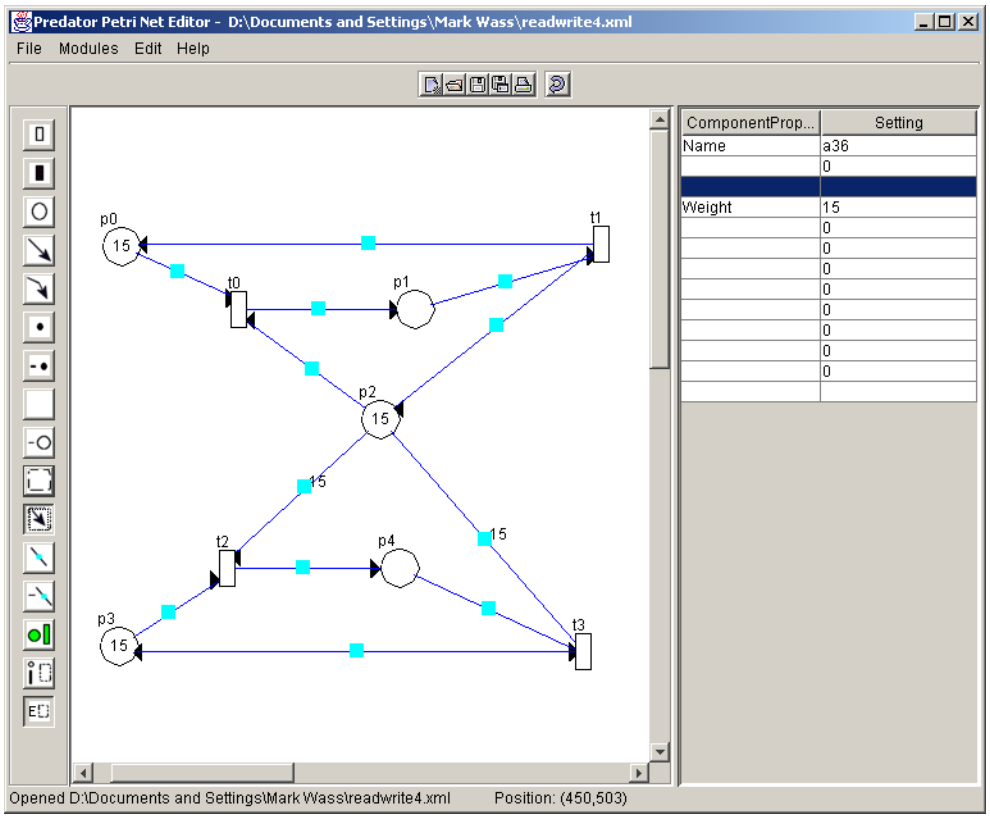
Component...	Setting
Name	t2
Weight	1.0
	0
	0
	0
	0
	0
	0
	0
	0
	0

Subnet added D:\Documents and Settings\Mark Wass\rwTemp...Position: (234,461)

Figure 9.17. *Selecting Immediate Transitions*. In this figure two immediate transitions have been selected. Entering a new weight in the weight column of the editTable can set their weight.

9.4.2 Editing Arcs

To edit arcs select the **editArcs** icon. This will display a square in the middle of each arc as shown in figure 9.18, below. To modify the weight of an arc click on its square, and change the weight displayed in editable.


Predator Petri Net Editor - D:\Documents and Settings\Mark Wass\readwrite4.xml

ComponentProp...	Setting
Name	a36
	0
Weight	15
	0
	0
	0
	0
	0
	0
	0
	0

Opened D:\Documents and Settings\Mark Wass\readwrite4.xml Position: (450,503)

Figure 9.18. *Editing Arcs*. The EditArc icon has been selected from the **editToolBar**. EditPoints are displayed in the middle of all arcs. To modify the weight of an arc click on its edit point and enter a new weight in the Weight column of the **ediTtable**.

9.4.3 Removing Petri Net Components

Components can be removed by selecting the remove components icon from the edit toolBar. Componenets are then removed by clicking on them. All arcs associated with places and transitions are also removed when they are deleted. The contents of subnets are also removed when subnets are removed. To clear the canvas click the **clear** icon on the **editToolBar**.

9.4.4 Opening, Saving and New Files

The **File Menu** and **File ToolBar** enable the user to open and save files and create new files.

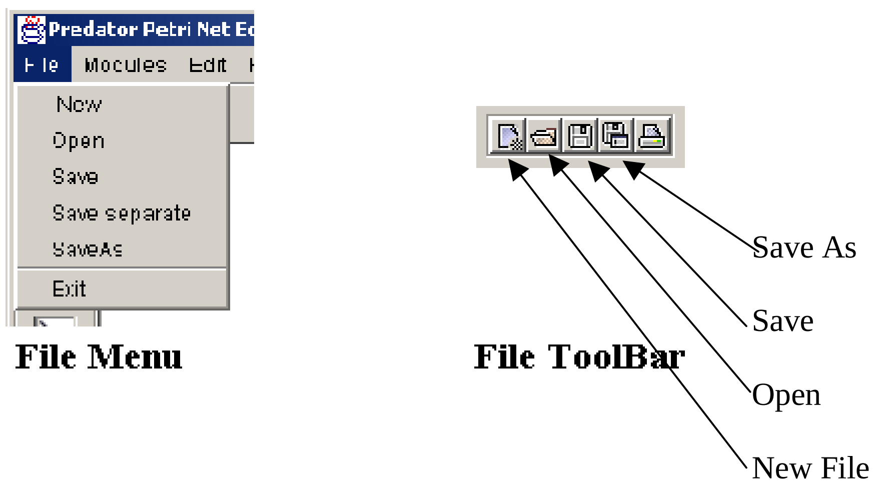


Figure 9.19. *The FileMenu & File ToolBar*.

New – allows the user to create a new file, if the current file has been modified then the user is prompted to see ifthey want to save the file

Open – open a Petri Net file – displays a File chooser dialog for the user to choose a file to open.

Save – save the current Petri Net, if the net is hierarchical it will be saved in as a single file rather than a set of files.

SaveSeparate – saves hierarchical Petri nets as a set of files rather than a single file.

SaveAs – saves a file under a different name. For hierarchical Petri nets the type of format previously selected is used.

9.5 Petri Net Analysis – Open Architecture

The Predator Petri net editor provides an open architecture, for the dynamic loading of analysis modules. The current version supports a maximum of four modules. This enables users to write their own analysis modules and upload them to the Editor.

9.5.1 Module Interface

Modules that are to be uploaded must follow a basic interface shown below:

```
public abstract interface Module {

    public abstract void runModule();

    public abstract String getModuleName();

    static final String inputFileName = "current.xml";

}
```

The runModule() method is called to execute the analysis module. GetModuleName(), is called by the editor to obtain the name of the analysis module.

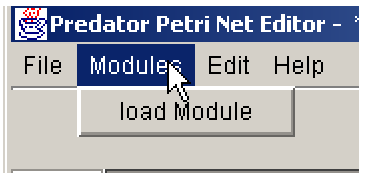
9.5.2 Loading an Analysis Module

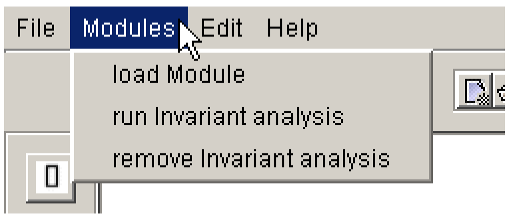
Modules can only be loaded if they are added to the PEditor jar file. The command to do this is:

```
jar uvf PEditor.jar  filename
```

To load an analysis module select **load module** from the **Module** menu (figure 9.20). A dialog or filechooser will be displayed for you to enter the name of the class you wish to load.

Two menu items are added to the **module** menu when a module is loaded. One is labelled **run m*oduleName***, the other labelled **remove *moduleName***, where ***moduleName*** is the name returned by the modules getModuleName function.





A B

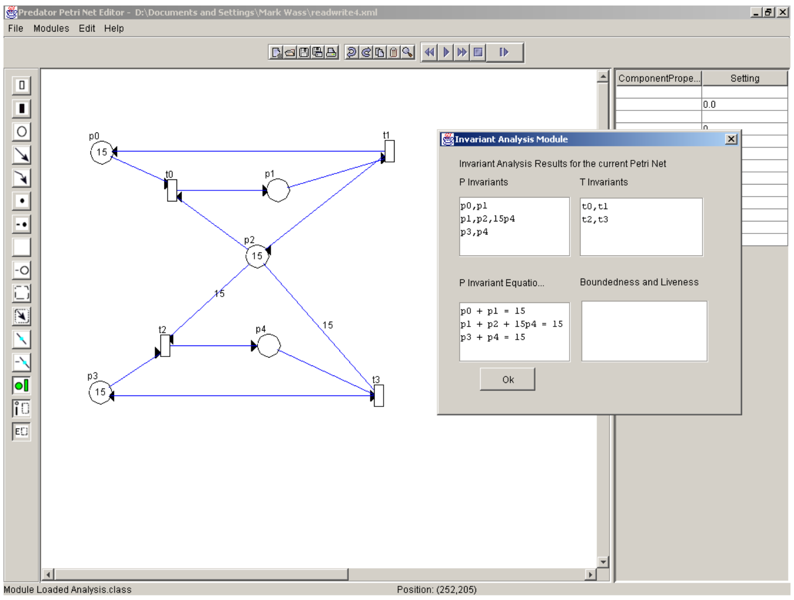
Figure 9.20. *The Module Menu.* A). The module menu when no modules are loaded. The load module menu item is pressed to load a module. B) the module menu once the invariant analysis module has been loaded. Two menu items are added for each module loaded; one to run the module and the other to remove the module.

9.5.3 Running An Analysis Module

To run an analysis module select **run m*oduleName*** from the **module** menu. To remove a module select the appropriate **remove *moduleName,*** from the **module** menu.

9.6 Invariant Analysis Module

An analysis module to perform invariant analysis has been implemented to work with the open architecture. The module can be loaded as described in the previous section. When executed the module displays a dialog (shown in figures 9.21 & 9.22), giving the P and T invariants as well as the P invariant equations and information on the liveness and boundedness of the Petri Net.


Predator Petri Net Editor - D:\Documents and Settings\Mark Wass\readwrite4.xml
File Modules Edit Help
ComponentPrope... Setting
0.0
p0
15
t1
t0
p1
p2
15
15
t2
p4
15
p3
15
t3
Module Loaded Analysis.class
Position: (252,205)

Invariant Analysis Module

Invariant Analysis Results for the current Petri Net

P Invariants

```
p0,p1
p1,p2,15p4
p3,p4
```

T Invariants

```
t0,t1
t2,t3
```

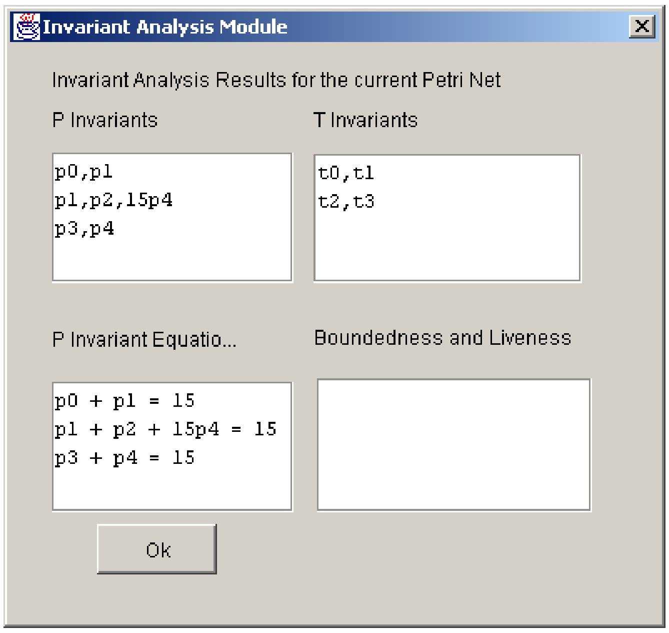
P Invariant Equatio...

```
p0 + p1 = 15
p1 + p2 + 15p4 = 15
p3 + p4 = 15
```

Boundedness and Liveness

Ok

Figure 9.21. *Invariant Analysis Results*. The invariant analysis dialog, for the readers writers problem. When run the invariant analysis module displays its results in a dialog.



Invariant Analysis Module

Invariant Analysis Results for the current Petri Net

P Invariants

```
p0,p1
p1,p2,15p4
p3,p4
```

T Invariants

```
t0,t1
t2,t3
```

P Invariant Equatio...

```
p0 + p1 = 15
p1 + p2 + 15p4 = 15
p3 + p4 = 15
```

Boundedness and Liveness

Ok

Figure 9.22. *The invariant analysis dialog*. The P and T invariants as well as the P invariant equations and statements on the boundedness/liveness of the system are displayed.