Imperial College of Science,

Technology and Medicine

(University of London)

Department of Computing

Production of the Extensible Petri Net Editor/Animator "Medusa"

by

Nicholas J. Dingle

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 formalism for the analysis of concurrent systems and as such there are a plethora of existing tools which allow users to edit, animate and analyse a range of Petri net classes. These tools are essentially limited, however, to the functionality incorporated into them when they are written. The main aim of the project was therefore to produce a basic Petri net editor/animator tool which could be arbitrarily extended by the user. This was achieved by designing an architecture which would allow the program to load user-designed modules about which nothing is known until runtime. The project also provided an opportunity to design a tool which offered new features not present in existing pieces of software and which also corrected known flaws in these tools. In particular, the animator designed as part of Medusa incorporates a novel backwards animator which allows the user to step backwards through the sequence of transitions which they have fired.

Two modules were produced as part of the project. The first of these was designed to use graph theory to analyse Place-Transition nets for properties such as liveness and boundedness. The second allowed Medusa to interface with an existing Markov chain analyser called Dnamaca. This permitted Medusa to be used to perform performance analysis on Petri nets. The implementations of these modules and of the Medusa tool were validated against a set of known results.

Results were also produced from an invariant analysis module designed by a third-party. This was intended to demonstrate that the aim of designing an extensible tool with a set interface had been achieved successfully. From this it would be possible to conclude whether or not other parties could design and implement their own modules which would work correctly.

Acknowledgements

I would like to express my thanks to the Dr. William Knottenbelt, my supervisor, for initially suggesting the topic and for his help and enthusiasm throughout the project's duration. I would also like to thank my family and friends for their love and support during the whole of the MSc course.

Table of Contents

Title Page . . . . **i**

Abstract . . . . **ii**

Acknowledgements . . . . **iii**

Table of Contents . . . . **iv**

1 Introduction **1**

1.1 Motivation 1

1.2 Report Outline 2

2 Background **5**

2.1 Introduction 5

2.2 Overview of Petri Net Theory 5

2.2.1 Place-Transition Nets 5

2.2.2 Generalised Stochastic Petri Nets (GSPNs) 9

2.2.3 Other Petri Nets 10

2.3 Existing Tools 11

2.3.1 DaNAMiCS 12

2.3.2 Renew 12

2.3.3 Petri Net Kernel 12

2.4 Justification for Medusa 13

3 The Medusa Editor/Animator Architecture **16**

3.1 Introduction 16

3.2 The Editor's Architecture 16

3.3 The Animator's Architecture 18

3.3.1 Forwards Animation 19

3.3.2 Backwards Animation 21

4 **Extensibility** **24**

4.1 Introduction 24

4.1.1 Overview of How Medusa's Extensibility Works 24

4.2 The Petri Net Markup Language 26
4.2.1 JAXP or JAXB? 29
4.2.2 XML Parsing in Medusa 30
4.3 Reflection: Loading and Running Modules 31

5 Design and Implementation of a Module for the Analysis of Place-Transition Nets Using Graph Theory 35
5.1 Introduction 35
5.2 Properties of Petri Nets Which Can Be Analysed Using Graph Theory 35
5.2.1 Analysis of Place-Transition Nets Using Graph Theory 37
5.2.2 Analysis of GSPNs Using Graph Theory 40
5.3 Implementation of the Module 41
5.3.1 Analysis of the Coverability Graph 41
5.3.2 Validation of the Chosen Algorithm 45
5.3.3 Completing the Analysis 46

6 **Design and Implementation of a Module to Interface With Dnamaca 48**
6.1 Introduction 48
6.2 Dnamaca Input Files 49
6.2.1 Dnamaca Model Descriptions 49
6.2.2 Dnamaca Performance Measures 52
6.3 DnamacaModule 53
6.3.1 Model Description Generation 54
6.3.2 Performance Measures 57
6.3.3 How DnamacaModule Invokes Dnamaca and Displays Results 58

7 Validation of Concept and Design Through Analysis of Generated Results 61
7.1 Introduction 61

7.2 Graph Theory Analysis Module 61
7.3 DnamacaModule 64
7.4 Running a User-Designed Module 66
7.5 Conclusion 70

8 **Conclusion** **72**
8.1 Conclusions 72
8.2 Opportunities for Future Work 73

Appendix A: Medusa User Guide **75**

Bibliography **79**

Chapter 1: Introduction

1.1 Motivation

Petri nets are a widely used formalism for the analysis of concurrent systems and as such there are a plethora of existing tools which allow users to edit, animate and analyse a range of Petri net classes. These tools are essentially limited, however, to the functionality incorporated into them when they are written. It is impossible, therefore, for the user to use the program to perform some function not supported by it (for example a certain type of analysis) unless they have access to the source code. Even if they do have such access there is no guarantee that they will be able to understand and modify it to suit their needs.

Medusa was conceived to address this problem. The main aim of the project was therefore to produce a basic Petri net editor/animator tool which could be arbitrarily extended by the user without access to its source code. This would enable users to investigate the properties in which they are interested. This will be achieved by designing an architecture which will allow the program to load user-designed modules about which nothing is known until runtime. As Java provides a mechanism suitable for this, the tool was conceived from the beginning as being programmed in this language. The project also provided an opportunity to design a tool which offered new features not present in existing pieces of software and which also corrected known flaws of these tools. In particular, the animator designed as part of Medusa incorporates a novel backwards animator which allows the user to step backwards through the sequence of transitions which they have fired. Such a feature is not found in the tools currently available.

This project aims to produce two modules which will be used to validate the finished editor. The first module will use graph theory to analyse Petri nets. If this is successful it will demonstrate that it is possible to extend Medusa's functionality through purpose-designed analytical modules. The second module will allow Medusa to interface with Dnamaca, a Markov chain analyser. If this

module is successful it will show how the extensibility of Medusa allows it to make use of the analytical tools of existing applications.

In addition, Medusa will be tested with a module designed as part of another MSc project (see [MW01]). The design of this module has proceeded independently from Medusa except that details of the interface mechanism were shared. If Medusa is capable of loading and running this module successfully it will show that the main aim of this project has been accomplished - namely that it is possible for a user to design modules suitable for their needs when they only know the details of the interface.

In order to assess the success of these modules examples from a variety of sources will be analysed. The results provided in these sources will then be compared with those generated by the modules. As will be shown below, there are no discrepancies between the published and generated results and hence the concept and design of Medusa has been validated.

1.2 Report Outline

The layout of the report is:

Chapter 2 provides the background information to this project. An overview of Petri net theory is provided which covers Place-Transition and Generalised Stochastic Petri Nets as these are the types supported by Medusa. Some other common types of Petri net are introduced for completeness. This is followed by a brief description of some existing Petri net tools which leads on to the justification for the production of yet another such program.

Chapter 3 details the architecture of the Medusa editor/animator. An Object Orientated approach is used as well as some fragments of Java code. Medusa presents Petri nets to the user in graphical form and allows elements to be added and removed as desired. The functionality of the Medusa animator is also described. This animator is capable of traditional forwards animation where the

user selects which enabled transition they wish to fire and then updates the graphical display of the net accordingly. It also allows the user to undo the most recent firing – a feature known as backwards animation. The various ways in which this could have been accomplished are described before the implementation of the method using a stack is justified.

Chapter 4 deals exclusively with the way in which Medusa can be extended with the addition of user-created modules. This is accomplished through the use of the Java Reflection mechanism. A description of Reflection in general is given before the implementation used by Medusa is considered in more detail. Central to the process of running a module is the parsing of the description of the Petri net currently opened in Medusa from a Petri Net Markup Language (PNML). This requires the use of an XML parser and so the two main ways of doing this in Java (JAXP and JAXB) are detailed. Finally, the decision to adopt the JAXP method is justified.

Chapter 5 explains how a module which uses graph theory to analyse a Petri net's attributes was produced. The mathematical theory which underlies this process is described in detail to ensure that the reader is aware of the issues which influenced the implementation. The key issue faced when designing this module was the implementation of an algorithm capable of describing the strongly connected components of a graph and it is on this topic that the second part of this chapter concentrates. The chapter ends by showing that the implementation of the algorithm adopted produces correct results.

Chapter 6 describes a module which allows Medusa to interface with the Dnamaca Markov chain analyser. The prime motivation behind this was to correct a flaw in the implementation of the DaNAMiCS tool which attempts to perform the same function. It also provided an opportunity to show that Medusa could be made to interface with existing tools through its modular system. The implementation of the module uses the incidence function description of Petri nets to create correct Dnamaca model definitions. DnamacaModule also runs Dnamaca automatically with the model description it has generated. The way in which this was accomplished is described.

Chapter 7 validates the concept and implementation of Medusa as an extensible editor. This is achieved by using the modules described above to produce results by running them with Medusa and comparing these results against published versions. This process not only shows that it is possible to produce an extensible editor capable of running modules but also demonstrates that the specific implementations of the modules are correct. Particularly important is the fact that a module produced by a third-party, Mark Wass, is run successfully by Medusa and produces correct results. This demonstrates that the main concept behind Medusa, as detailed in the motivation above, has been fulfilled by the implementation.

Chapter 8 summarises what has been achieved in the course of this project, presents some conclusions and raises issues which could be addressed in future work. It is concluded that the project has been a success because of the results presented in Chapter 7. These demonstrate conclusively that it is possible to design modules for Medusa without access to the source code which function correctly.

Appendix A provides a user guide for the Medusa editor/animator.

Chapter 2: Background

2.1 Introduction

The aim of this chapter is to introduce the reader to the project's background. First an overview of the mathematical modelling formalism of Petri nets is provided which covers the nets supported by Medusa as well as introducing briefly some of the other forms which exist. This is followed by a section detailing the existing pieces of Petri net modelling software. This section will identify why there is the need for another piece of software in what is already a well-provided field.

2.2 Overview of Petri Net Theory

Petri nets were invented by Carl Adam Petri in 1962 as a formalism for describing and reasoning about concurrent systems [BK95]. They have been used to model a variety of such systems, including communication protocols, parallel programs, multiprocessor memory caches and distributed databases [KNO99]. Petri initially described Place-Transition nets but numerous other classes of nets have since been defined to allow more sophisticated reasoning. One such class of nets is Generalised Stochastic Petri Nets (GSPNs) which introduce time as a variable and thus allow performance analysis of the modelled system to be conducted. This chapter is mostly concerned with Place-Transition nets and GSPNs as these are the two forms of Petri nets which can be modelled with the Medusa editor, but brief mention will also be made of some of the other types which exist.

2.2.1 Place-Transition Nets

Place-Transition nets are the basic type of Petri net from which all other types are derived. As described in [BK95], they consist of four elements:

- **places**, which are represented by circles and model conditions or objects such as program variables.
- **tokens**, which are represented by black dots. These are contained within places and represent the specific value of the condition or object which that place represents. The initial arrangement of tokens on places is known as the **initial marking** of the Petri net.
- **transitions**, which are represented by hollow rectangles and model activities which change the values of conditions and objects.
- **arcs**, which are represented by lines connecting places and transitions. These indicate which objects are changed by which activities. As Place-Transition nets are bipartite, arcs may only connect places to transitions or transitions to places, but not places to places or transitions to transitions. An arc may have a **weight**, which specifies how many tokens are created or destroyed when a transition to which it is attached is fired.

All of these can be seen in the illustration of a simple Place-Transition net in Figure 2.1.

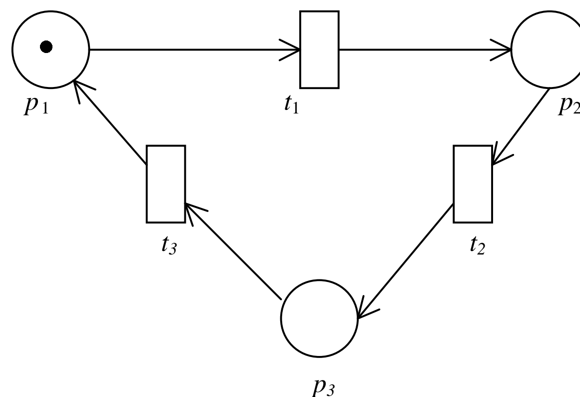


Figure 2.1: A Place-Transition net.

A Place-Transition net can be defined formally using functional notation:

Definition 2.1 *A Place-Transition net is a 5-tuple* $PN = (P,T,\Gamma^-,\Gamma^+,M_0)$ *where*

- $P = \{p_1, \ldots, p_n\}$ *is a finite and non-empty set of places,*
- $T = \{t_1, \ldots, t_m\}$ *is a finite and non-empty set of transitions,*
- $P \cap T = \emptyset$,
- $\Gamma^-,\Gamma^+ : P \times T \rightarrow \mathbb{N}_0$ *are the backward and forward incidence functions respectively. If* $\Gamma^-(p,t) > 0$, *an arc leads from place p to transition t, whilst if* $\Gamma^+(p,t) > 0$ *then an arc leads from transition t to place p,*
- $M_0 : P \rightarrow \mathbb{N}_0$ *is the initial marking.* [BK95]

The Petri net in Figure 2.1 could therefore be described as PN = $(P,T, \Gamma^-,\Gamma^+,M_0)$ where

- $P = \{p_1, p_2, p_3,\}$,
- $T = \{t_1, t_2, t_3,\}$,
- $\Gamma^-(p_1,t_1) = 1, \Gamma^-(p_2,t_2) = 1, \Gamma^-(p_3,t_3) = 1$. *All other values of* Γ^- *are zero,*
- $\Gamma^+(p_1,t_3) = 1, \Gamma^+(p_3,t_2) = 1, \Gamma^+(p_2,t_1) = 1$. *All other values of* Γ^+ *are zero,*
- $M_0(p) = 1$ *if* $p = p_1$, $M_0(p) = 0$ *otherwise,* $\forall p \in P$.

The dynamic behaviour of a Petri net is determined by rules concerning the enabling and firing of transitions, as described in [BK95]. When the arcs connecting a transition to its input places have a weight of one, the transition is enabled if all of its input places are marked with at least one token. Only an enabled transition may fire. When it does one token on each of its input places is destroyed and one token is created on each of its output places. It is possible, however, for an arc to have a weight greater than one. In this case, if the arc is an input arc then the transition is only enabled if the number of tokens on the place to which it is connected is equal to or greater than its weight. When then transition is fired, the number of tokens destroyed on the place is equal to the arc's weight. If it is an output arc, firing the transition creates the number of tokens on the output place equivalent to the arc's weight. The numerical values of each of the Γ^- and Γ^+ functions correspond to the weights of the arcs connecting

places and transitions. By convention arc weights of 1 are not shown explicitly [BK95]. In Figure 2.1, it can be seen that t_1 is the only enabled transition as there is one tokens on p_1. The effect of firing t_1 is shown in Figure 2.2.

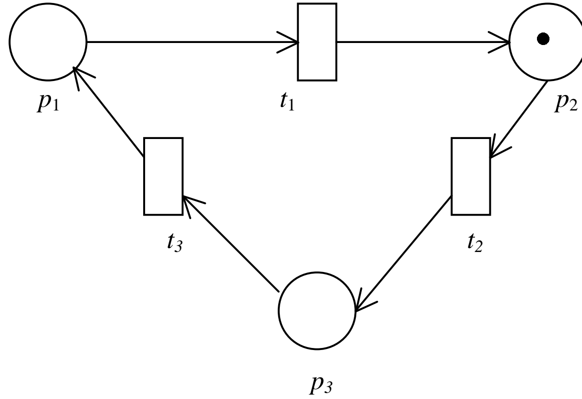


Figure 2.2: The effect of firing the enabled transition of the net in Figure 2.1.

The rules for the enabling and firing of transitions can be formalised thus:

Definition 2.2 *If* $PN = (P,T,\Gamma^-,\Gamma^+,M_0)$ *is a Place-Transition net*

- *A marking of a Place-Transition net is a function* $M : P \rightarrow \mathbb{N}_0$ *, where* $M(p)$ *is the number of tokens on place p*
- *A set* $\tilde{P} \subseteq P$ *is* **marked** *at marking M, iff* $\exists p \in \tilde{P} : M(p) > 0$*; otherwise* $\tilde{P}$ *is* **unmarked** *or* **empty** *at M*
- *A transition* $t \in T$ *is* **enabled** *at M, denoted by* $M[t >$*, iff* $M(p) \geq \Gamma^-(p,t)$ $\forall p \in P$
- *A transition* $t \in T$*, enabled at marking M,* **may fire** *yielding a new marking M' where*

  $$M'(p) = M(p) - \Gamma^-(p,t) + \Gamma^+(p,t) \qquad \forall p \in P$$

  denoted by $M[t > M'$*. In this case M' is* **directly reachable** *from M and we write* $M \rightarrow M'$*. Let* $\rightarrow^*$ *be the reflexive and transitive closure of* $\rightarrow$*. A marking M' is* **reachable** *from M, iff* $M \rightarrow^* M'$*.*
- *A firing sequence of PN is a finite sequence of transitions* $\sigma = t_1 \ldots t_n$ $n \geq 0$ *such that there are markings* $M_1, \ldots, M_{n+1}$ *satisfying* $M_i[t_i > M_{i+1}$ $\forall i = 1, \ldots, n$*. A shorthand notation for this case is* $M_1[\sigma >$ *and*

$M_1[\sigma > M_{n+1}$ respectively. The empty firing sequence us denoted by ε and $M[\varepsilon > M$ always holds. [BK95]

The firing of a transition when the Petri net has one marking creates a new marking. The set of all markings which are reachable from M_0 is known as the **reachability set** of the Petri net and the connections between the markings in this set are represented by the reachability graph [BK95]. The use of such graphs in the analysis of the attributes of a given Petri net, for example whether or not it is live, is examined below in Chapter 5.

Place-Transition nets do not contain any concept of time and as such cannot be used as a performance analysis formalism [KNO99]. There are, however, a number of time-augmented Petri net formalisms which can be used to model performance. One of the most widely used is the Generalised Stochastic Petri Net (GSPN) and it is these which Medusa supports.

2.2.2 Generalised Stochastic Petri Nets (GSPNs)

GSPNs have two types of transitions: immediate and timed. An enabled immediate transition fires in zero time whilst enabled timed transitions fire after a random exponentially-distributed delay (usually designated as rate λ_i for transition t_i). Timed transitions are represented as hollow rectangles whilst immediate transitions are filled. If only timed transitions are enabled, the probability of one transition t_i which is a member of the set of enabled transitions ($EN_T(M)$) firing is given in [BK95] as:

$$\frac{\lambda_i}{\sum_{j:t_j \in EN_T(M)} \lambda_j}$$

When $EN_T(M)$ contains only one immediate transition, that transition fires with probability 1.0. If, however, it contains more than one such transition, the relative frequency with which they fire is determined by using their assigned weights. Given the simultaneously enabled immediate transitions $t_1, \ldots, t_n$ with

corresponding weights $w_1, \ldots, w_n$, the probability of t_i firing is given in [KNO99] as:

$$\frac{w_i}{\Sigma_{k=1} w_k}$$

A GSPN can be defined formally thus:

Definition 2.4 *A GSPN is a 4-tuple GSPN =* (PN, T_1, T_2, W) *where*

- $PN = (P, T, I^-, I^+, M_0)$ *is the underlying Place-Transition net,*
- $T_1 \subseteq T$ *is the set of timed transitions,* $T_1 \neq \emptyset$*,*
- $T_2 \subseteq T$ *denotes the set of immediate transitions,* $T_1 \cap T_2 = \emptyset$, $T = T_1 \cup T_2$,
- $W = (w_1, \ldots, w_{|T|})$ *is an array whose entry* $w_i \in \mathbb{R}^+$ *is either*
  - *a possibly marking dependent rate of a negative exponential distribution specifying the firing delay, when transition* t_i *is a timed transition, i.e.* $t_i \in T_1$*, or*
  - *a possibly marking dependent firing weight where transition* t_i *is an immediate transition, i.e.* $t_i \in T_2$. [BK95]

As described in [BK95], a GSPN has two types of markings. Immediate transitions fire in zero time and as such the time spent in markings with enabled immediate transitions is also zero. These markings are known as vanishing markings as a random observer will never see them. However, markings which enable only timed transitions will be observed as they are not left immediately. These are known as tangible markings.

2.2.3 Other Petri Nets

Two other common forms of Petri nets are Coloured Petri Nets (CPNs) and Queuing Petri Nets (QPNs). As Medusa does not support either only a brief outline is presented below. Readers wishing to know more are directed to the bibliography, especially those texts which comprise the sources for this chapter.

One problem with Place-Transition nets and GSPNs is that their graphical representations can become very confused for large or complex systems. Coloured Petri Nets are intended to remedy this problem and are fully described in [BK95]. In a CPN tokens are assigned different colours and the transitions have different firing rules based on the colours of the tokens on their input places. They have not been included in Medusa because they add no expressive power; it is possible uniquely to unfold every CPN into an uncoloured Petri net representing the same model [KNO99].

As described in [KNO99], Queuing Petri Nets are an attempt to overcome the difficulties faced when modelling queues with GSPNs. Queuing Petri Nets integrate the concept of queues into a coloured version of GSPNs (CGSPNs). To summarise briefly, a queued place has two components: the queue and the depository for tokens which have been serviced at this queue. Tokens which arrive at a queued place are placed in its queue for service and are not available to output transitions until they have exited the queue and been placed in the depository. QPNs are therefore a convenient way of modelling queues using Petri nets but their use is somewhat specialised and they have, for that reason, not been supported by Medusa.

2.3 Existing Petri Net Tools

There are a huge number of pre-existing Petri nets tools. For example, one web-page contains links to 43 different pieces of software designed explicitly for the creation and analysis of Petri nets.[1] A number of examples tools which illustrate the range available are given below before some of the problems with existing tools are elucidated. This leads to a justification as to why yet another piece of software is needed in what would seem to be an over-crowded field.

[1] See http://www.aut.utt.ro/~mappy/petri/home.html

2.3.1 Data Network Architecture - Modelling Concurrent Systems (DaNAMiCS) [DAN]

DaNAMiCS is an improved version of DNAnet, another Petri net tool. It supports Place-Transition, Generalised Stochastic and Coloured Petri Nets, and allows the user to insert subnets into other nets. It has an animator which allows the user to fire enabled transitions but has no facility to undo a firing. Its analysis suite is highly comprehensive and includes a simulator as well as invariant and graph-theory based analysis tools. It also has the ability to perform steady-state analysis of a Petri net by exporting it to Dnamaca, a Markov-chain analyser.

2.3.2 Reference Net Workshop (Renew) [REN]

In contrast to DaNAMiCS, Renew is simple and does not offer the same range of analysis tools as DaNAMiCS. Its only tool is a simulator. It does, however, incorporate some interesting features. The design is intended to be of open-architecture and the source code is freely distributed so that users with knowledge of Java programming can customise it as much as they want (including adding analysis tools). Also, it has the ability to export nets in an XML-based format which could be read or produced by other tools without knowledge of the internal architecture of Renew.

2.3.3 Petri Net Kernel [PNK]

Like Renew, Petri Net Kernel is much simpler than DaNAMiCS. As its name suggests it is not intended as a complete tool in its own right but as the basic unit around which the user can construct, in Java, a more sophisticated application. It uses XML heavily both to save nets created in it by the user and to define valid net classes and tools created by the user to suit their particular needs. The only analysis tool which it includes is a simulator.

2.4 Justification for Medusa

Given this range of existing software, why is there a need for another tool to be produced? The production of Medusa can be justified on two levels: it corrects known flaws in existing tools' implementations and it offers services which no other application does.

As described above, DaNAMiCS is able to perform steady-state analysis by exporting nets to the Dnamaca Markov chain analyser. There is a flaw in its implementation, however, which causes it to generate incorrect results when a place is both the input and the output place of a transition. If a user does not have access to DaNAMiCS' source code there is no way to rectify this. Medusa, however, offers the ability to solve the problem as a user can write a module to perform the conversion correctly. Furthermore, the conversion in DaNAMiCS is entirely automated and therefore does not allow the user to investigate exactly the properties of the net in which they are interested. A Dnamaca interface module could improve upon this by allowing the user to customise the Dnamaca file which it produced before passing it to Dnamaca. Of course, this does not apply exclusively to Dnamaca. A similar module can be produced to allow the conversion of Petri nets into a format suitable for input into another tool.

Medusa attempts to offer a range of services which no other tool does. Tools like DaNAMiCS offer an enormous range of analysis options, but that range is still limited and if the user wishes to do something not included in this range then they must either find another tool or obtain the source code and modify it appropriately. The design of Medusa, therefore, is intended to produce a tool with an open architecture, supporting basic features like editing and animation but allowing the user to create modules to perform specific tasks without having to start from scratch or modify somebody-else's code. The Petri Net Kernel aims to do something similar but the basic functionality which it includes is less than that of Medusa. Consequently it requires more work to produce the same results.

The XML format of Renew is a very interesting feature as it promotes interoperability between tools. It allows Petri nets to be defined textually and so

to be inputted to and outputted from other tools. The problem is that it is an entirely arbitrary format created by the programmers of Renew and therefore requires that other programmers have heard of the tool and know of its XML format if they are to incorporate it in their own programs. A better solution would be to adopt a generally recognised XML format about which others are likely to have heard. This ensures that a ‘standard’ output format genuinely is standard. The Petri Net Markup Language [PNML] is an attempt to introduce just such a format and it is this which Medusa uses as a standard output format.

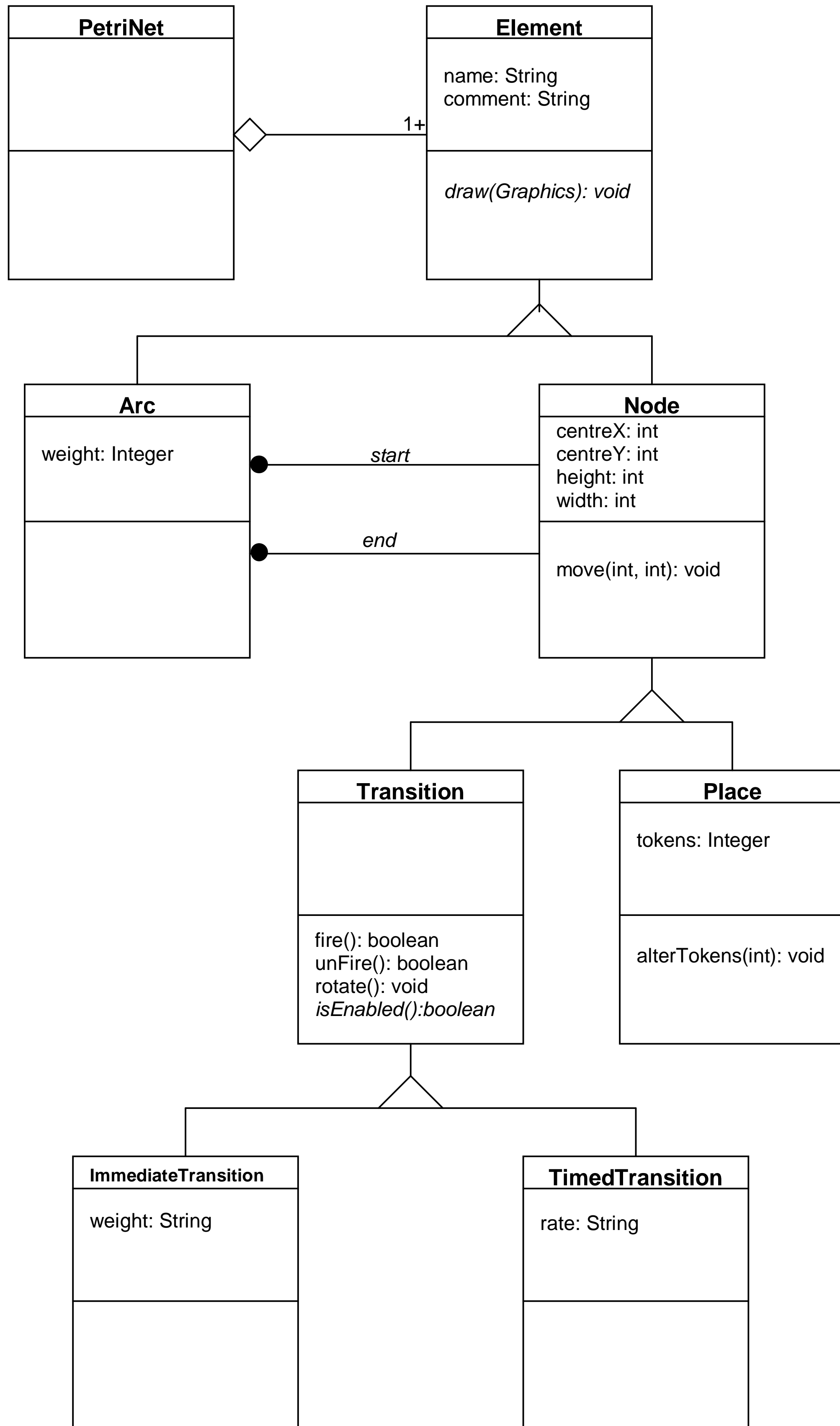


Figure 3.1: Object diagram of the structure of a Petri net as represented in Medusa

Chapter 3: The Medusa Editor/Animator Architecture

3.1 Introduction

The aim of this chapter is to provide details of the architecture of the basic Medusa package. The program is composed of two main parts, the editor (which allows the user to enter Petri nets through a graphical interface) and the animator (which allows the user to move tokens around a net). The layout of this chapter reflects this structure by considering each in turn.

For the editor, the emphasis is on the internal representation of Petri nets used by Medusa. This is fundamental to the success of the project as a whole: if Medusa does not represent Petri nets correctly then it will fail even if other features are a success.

For the animator, the way in which transitions are checked to see if they are enabled is detailed, as is the way in which firing them is handled. Together these allow the user to perform forwards animation on the displayed Petri net. A special feature of the Medusa animator is its ability to perform backwards animation. There were two main ways in which this could have been implemented and so both options are explored before the choice adopted is justified.

3.2 The Editor's Architecture

The editor is the core component of Medusa as it is through this that the user can add to and remove elements from the net which they are editing. This is accomplished through the Graphical User Interface (GUI) shown in Figure 3.2.

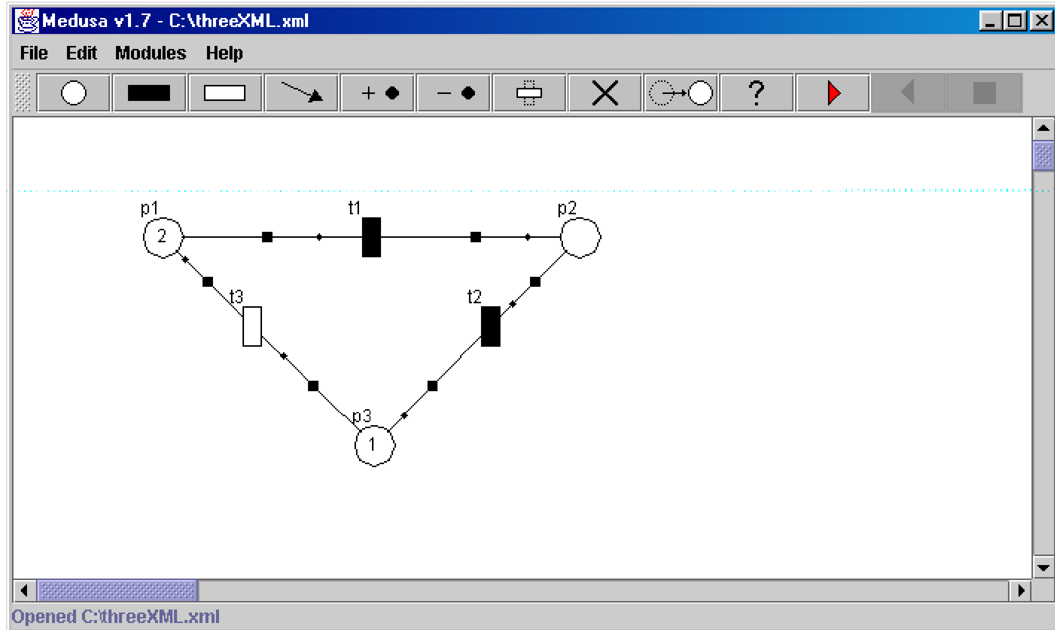


Figure 3.2: The appearance of the Medusa GUI.

The row of buttons below the menu bar allows the user to select which function they wish to perform. The large white panel below these buttons, known as the `PetriNetPanel`, displays the Petri net currently being edited. By clicking with the mouse on a button and then clicking on a location on the `PetriNetPanel` the appropriate action, such as adding a place, is performed at that location.

The object diagram in Figure 3.1 shows how Petri nets are represented in Java classes behind this `PetriNetPanel`. Note that not every function which exists in the source-code is shown on this diagram – there are, for example, a number of functions in the `PetriNet` class which deal with the addition of various types of element through the GUI which are not reproduced as they do not affect the way in which Medusa represents the nets.

Figure 3.1 expresses the structure of Medusa's internal representation very succinctly. The `PetriNet` stores its constituent `elements` in four `Vectors`, one each for `Places`, `Arcs`, `TimedTransitions` and `ImmediateTransitions`. `Vectors` are an ideal collection class for this as they can be iterated over in the

same manner as C++ arrays, which makes it easy to locate specific elements. They can also be resized dynamically. This means they can be initialised containing no elements but then as the user adds elements to the Petri net these elements can be added to the correct `Vector`. Elements can also be removed from `Vectors` and a `Vector` will resize itself automatically when this happens. This is used when the user wishes to delete elements from the net as the selected element is removed from the `Vector` in which it is located.

The storage of the transitions which make up the net in two separate `Vectors` could be criticised as in order to access all transitions, for example when drawing all transitions, first one `Vector` and then another must be iterated over. This is not a flaw, however, as it simplifies the process of checking exclusively for one type of transition – for example when checking if any immediate transitions are enabled before doing so for timed transitions (see below).

3.3 The Animator's Architecture

Medusa's animator has two modes of operation: forwards (or manual) animation and backwards animation. The first mode can also be found in the animators of existing tools like DaNAMiCS. It allows the user to select which transition they would like to fire and then performs the act of firing if the selected one is enabled, altering the marking of the net accordingly. All of this presented on the graphical display and so as the user fires a sequence of transitions they can observe the tokens moving around the net.

The second mode (backwards animation) is not supported by many tools. Its effect is to undo the firing of the most recently fired transition. This allows the user to step backwards through the sequence of fired transitions in order to correct mistakes or experiment with a different sequence. The various ways in which this could have been implemented are considered below before the actual way in which it was done is described and justified.

3.3.1 Forwards Animation

When the animator is started Medusa waits for the user to click on the `PetriNetPanel`. If a click occurs, its coordinates are compared with those of all transitions. If it occurs within a transition and that transition is enabled, the transition is fired and the display updated accordingly.

Checking to see if a mouse-click event occurs within a transition is straightforward as mouse-clicks have an (x, y) location in the same way as all `Nodes` of a Petri net. As the centre points and sizes of all `Nodes` on screen are known, whether or not a click occurs over a certain transition can be easily computed.

Having located the transition which the user wishes to fire, Medusa then attempts to fire that transition. The first step is to check that the transition is enabled. As described in Chapter 2, this is only the case if the all of the transition's input places are marked with at least the number of tokens specified by the arcs connecting these places to the transition. There is a further complication if the net is a GSPN and the user wishes to fire a timed transition as it is necessary first to ensure that there are no enabled immediate transitions. This is because immediate transitions fire in zero time and so fire before timed transitions. The pseudocode for the function which is used by timed transitions to check if they are enabled is shown in Figure 3.3. Note that the routine used by immediate transitions is identical except that the status of other immediate transitions is not checked first.

If the transition is enabled then it can be fired. First, the numbers of tokens specified by the weights of the connecting arcs are destroyed on the transition's input places. Then the correct numbers of tokens are created on the transition's output places, again according to the weights of the arcs connecting the transition to these places. The complete algorithm for achieving this is shown is Figure 3.4. It starts with a call to the routine in Figure 3.3 which checks if the transition is enabled. If it is not, the firing routine terminates, otherwise the creation and destruction of tokens proceeds.

```
public boolean isEnabled() {
      for (every immediate transition) {
            if (that transition isEnabled())
                  return false;
      }

      for (every arc) {
            if (that arc ends at this timed transition) {
                  //get the place at the start of the arc;

                  if (the number of tokens on that place is
                      less than the weight of the arc) {
                        return false;
                  }
            }
      }
      return true;
}
```

Figure 3.3: Pseudocode used by a timed transition to identify if it is enabled

```
public boolean fire() {
      if (isEnabled()) {
            for (every arc) {
                  if (that arc ends at this transition) {
                  //get the place at the start of the arc
                  //and alter the number of tokens on it
                  //by the weight of the arc
                  }
            }

            for (every arc) {
                  if (that arc starts at this transition) {
                  //get the place at the end of the arc
                  //and alter the number of tokens on it
                  //by the weight of the arc
                  }
            }
            return true;
      }
      return false;
}
```

Figure 3.4: Pseudocode representation of the routine by which the animator assess if a transition can fire and then performs the firing if it can

As this is a direct implementation of the rules for the dynamic behaviour of Petri nets as outlined in Chapter 2 this scheme gives the correct results. In order to aid the users of Medusa, when the animator is in use the transitions which are currently enabled are highlighted in red. This is shown in the screenshot in

Figure 3.5. Note that t_1 is an immediate transition and as such prevents t_3, a timed transition, from being enabled even though there is one token on p_3.

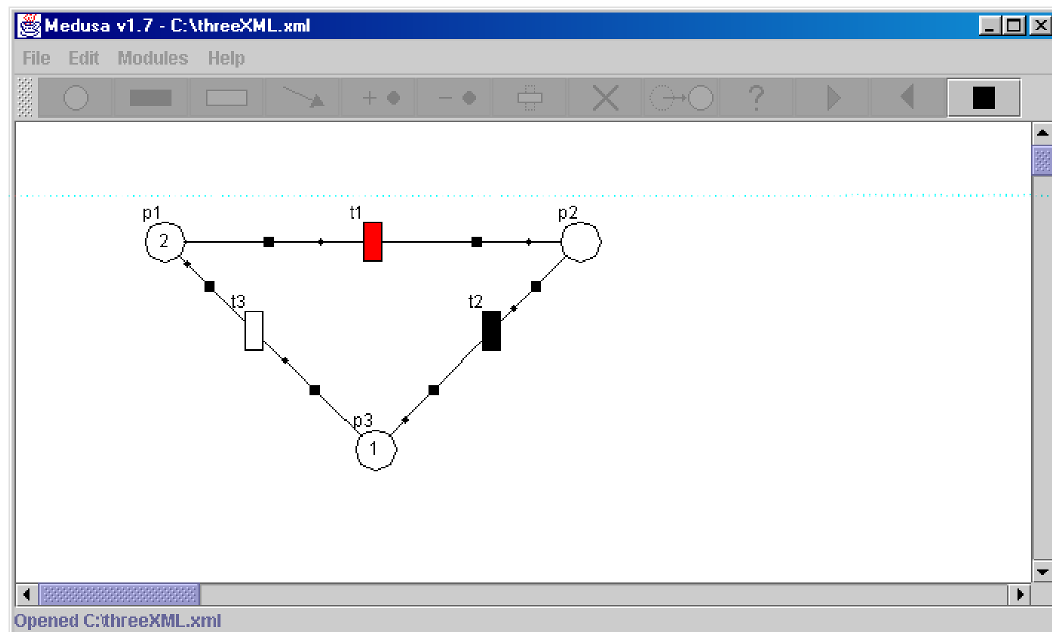


Figure 3.5: Screenshot showing Medusa whilst the animator is being used. Note that the only enabled transition, t_1, is highlighted.

When the user ends manual animation the net reverts to the marking which existed prior to the animator being enabled.

3.3.2 Backwards Animation

A novel feature of the animator provided with Medusa is the ability to do backwards animation. This is a feature lacking from DaNAMiCS. This gives the user the ability to backtrack whilst animating so they can retrace their steps whilst animating to try different choices or to correct mistakes. This is achieved by undoing the effect of firing the most recent transition to have been fired. There were two possible ways in which this could be achieved, namely by redoing all but the final transition which has been fired or by using a stack.

In this first method, a list of all transitions which have been fired is maintained. When the user wishes to step backwards through the animation, the net reverts to the initial marking and the effect the firing of each transition in the list is applied until the penultimate firing is reached. The resulting marking can then be applied to the net as a whole. The problem with this method is that it is inefficient: if *n* transitions are fired the cumulative effect of firing *n*-1 transitions must be calculated to undo one firing. Hence the larger the value of *n* the longer this process will take.

A much more efficient implementation is achieved through the use of a stack. Every time a transition is fired, details about it are pushed on to a stack. When the user wishes to perform backwards animation details about the last transition to be fired are popped off the top of the stack and, using these details, the transition is identified and then 'unfired'. The general operation of a stack is shown in Figure 3.6. This method is more efficient as stepping backwards to the previous marking only requires the calculation of the effects of 'unfiring' one transition, no matter how many transitions have been fired up to that point. For this reason it was this method which Medusa implements.

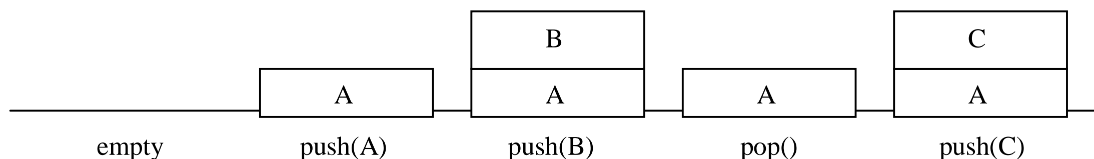


Figure 3.6: General operation of a stack

The implementation of the stack method was aided by the fact that Java provides a ready-made `Stack` ADT with `Object push(Object item)` and `Object pop()` methods. In Medusa, every instantiated `PetriNet` object has a `Stack` called `trace` which is used when animating. Whenever a transition is fired, its name is `push`ed on to `trace` and when the user wishes to step backwards the top name is `pop`ped off and used to identify the transition to be 'unfired'. The pseudocode representation of the routine for retrieving the correct transition is shown in Figure 3.7.

```
public void stepBackwards() {
      if (there is at least one transition on the stack){
            String name = (String) trace.pop();
            Transition t = getTransition(name);
            if (t is a valid transition) {
                  t.unFire();
            }

      }
}
```

Figure 3.7: Pseudocode of the function used for retrieving the transition whose firing is to be undone.

Once the transition has been identified the effects of its last firing can be undone. This process of ‘unfiring’ a transition is the exact opposite of firing it – the number of tokens specified by the output arcs’ weights are destroyed on the output places and the number of tokens specified by the input arcs’ weights are created on the input places. The marking of the net which existed before that transition was fired is therefore restored.

This chapter has examined the structure of the two basic components of Medusa. The aim of the project, however, was to create an extensible editor to which the user could add functionality through modules. The next three chapters, therefore, will detail the mechanism by which Medusa supports this and describe the production of two such modules. The success of the implementation is assessed in the penultimate chapter.

Chapter 4: Extensibility

4.1 Introduction

The aim of this chapter is to detail the mechanism by which Medusa supports the addition of user-designed modules about which nothing is known at compile-time. This mechanism contains two import elements. Firstly, XML is used to define saved nets. The format of the language used is the Petri Net Markup Language (PNML) which is a proposed standard for the interchange of Petri nets between various tools. There were a variety of ways in which files containing nets defined in this language could be parsed by Medusa. The section below lays out the advantages and disadvantages of each method. Secondly, the loading and running of modules is handled through the Java Reflection API. An overview of this mechanism is given as well as details of its specific implementation in Medusa. The Java interface which all modules must implement in order to be compatible with Medusa is given in this section.

Taken together these two elements define the interface through which Medusa interacts with user-designed modules. In order for Medusa to be considered a success, therefore, it should only be necessary for users wishing to design their own modules to understand the contents of this chapter. The only information to which a third party need have access is this description of the interface. If their modules adhere to the guidance given herein they should be able to be loaded and run by Medusa with no problems.

4.1.1 Overview of How Medusa's Extensibility Works

This section gives a step-by-step guide to the process of loading and running a module. In order for Medusa to run a module, the module must implement the `Module` interface shown in Figure 4.1. The use to which each method is put is covered in the step-by-step guide

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

Figure 4.1: The `Module` interface.

When a module is loaded and run the following events occur:

1) When the user selects "Load Module" from the "Module" menu, a standard Swing `file_chooser` dialog is opened. From this, the user selects which Java `.class` file containing the desired module. An instance of the module thus specified is created through Reflection. Modules must have a parameterless constructor.

2) A "Run Module" option is added to the "Module" menu. The "Module" menu will now resemble the screenshot shown in Figure 4.2.

3) When the user selects "Run Module" from the menu, Medusa saves the Petri net being currently edited under the name `current.xml` and Reflection is again used to invoke the `runModule()` method of the module.

4) This causes the module to run. Typically it will have to parse `current.xml` into its internal representation of a Petri net before executing further.

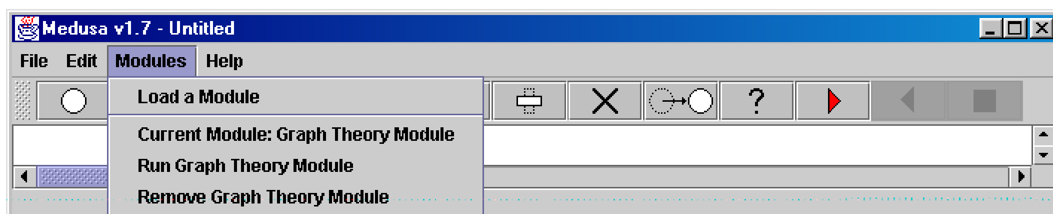


Figure 4.2: Screenshot showing the appearance of the Module menu when a module has been loaded and is ready to be run.

4.2 The Petri Net Markup Language [JKW08/00] [PNML]

The Extensible Markup Language (XML) is a meta-markup language which defines the rules by which users can define their own markup languages based on tags [ERH99]. These tags define the meaning and structure of the elements in the document in which they occur rather than the way in which that document is formatted. This ability to customise a language to suit specific needs has lead to XML becoming used in a variety of fields. As described in [ERH99], there are XML languages for describing topics as varied as chemical formulae,[1] musical scores[2] and job advertisements[3].

XML is easy to read and write as it is non-proprietary and composed entirely of ASCII characters [ERH99]. It can also be understood by a human reader. This makes it ideal for use as a language for exchanging data between pieces of software as the applications involved need only understand how to read XML and not the proprietary data formats which each uses (many applications exploit this – see [ERH99]). As Medusa is designed to be extensible, such a language is ideal as it allows it the saving of nets in a way which permits other tools to load them. Medusa will also be able to load nets created in another tool in the same fashion. Given that Petri nets are a commonly used modelling formalism, it should come as no surprise that attempts are being made to introduce a standard format for the interchange of Petri nets. This format is called the Petri Net Markup Language [JKW08/00] [PNML].

PNML is not the first time that XML has been used as a language for the definition of Petri nets, however. As detailed in [KW00], the Renew application supports the export of nets in XML. The problem with the format, however, is that it is geared exclusively towards the needs of Renew and as such contains data concerning the way in which that tool displays the nets which would be of no use to another piece of software. For example, it saves the colour of element and the font used to display their labels. Medusa, however, does not store either.

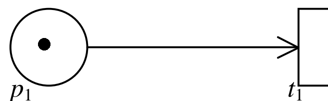
[1] See http://www.xml-cml.org/
[2] See http://www.oasis-open.org/cover/mnml199906.html
[3] See http://www.hr-xml.org/channels/home.htm

At the same time, PNML is designed as a generic language for representing Petri nets without worrying how each tool represents them (graphically or otherwise). It is also being developed as a standard for all tools and is not just implemented by one of tools amongst the range available. It therefore seems ideally suited to the role of being the exchangeable saved-file format of Medusa.

PNML is not the only attempt to define a standard Petri net exchange language. A number of other text-based formats have been suggested. These include the Abstract Petri Net Notation (APNN) [BKK94] and the format used by Design/CPN [LM00]. Neither were suitable however. APNN defines the structure of the net without any tool-specific implementation details. However, the way in which GSPNs were represented is not very intuitive as it does not have transitions which were explicitly "timed" or "immediate" but merely of priority "0" or "1" [APNN]. APNN also lacks a way of representing the graphical layout of a Petri net and so a tool using it as a file format would have to employ a graph-drawing algorithm to draw nets saved in this language. The complexity of such algorithms would greatly increase the difficulty of programming such a tool. The Design/CPN format was rejected for the same reasons as the Renew XML format – namely that it is a proprietary format which is only used by a single tool.

PNML has two further attributes which makes it the most attractive choice for Medusa's file format. Firstly, it supports the addition of user-defined Petri net types [JKW0/00]. This means that although it does not explicitly support timed and immediate transitions as used in GSPNs, it can easily be modified so to do and will still be understood by other tools. Secondly, it is an XML based format. This is a great advantage as there are standard APIs for parsing to and from XML in Java which could be implemented. Had APNN been chosen for example, the construction of a parser suitable for reading files saved in that format would have been much more time-consuming. In order to illustrate the PNML format, Figure 4.3 shows a net and its corresponding PNML description. As can be seen PNML is easy to understand. In between the `<place id="p1">` and `</place>` tags, everything about that place is described. The value of its initial marking is contained between tags of that name, whilst its on-screen location is found in the `<position />` tag, which is marked as containing graphical information.

Transitions are similarly described, whilst arcs record their start and end as the nodes which they connect.




```
<?xml version="1.0"?>
      <net id="n1" type="null">
            <name>
                  <value>C:\PNML Example.xml</value>
            </name>

            <place id="p1">
                  <graphics>
                        <position x="141" y="79" />
                  </graphics>
                  <name>
                        <value>null</value>
                        <graphics>
                              <offset x="-15" y="-15" />
                        </graphics>
                  </name>
                  <initialMarking>
                        <value>1</value>
                        <graphics>
                              <offset x="0" y="0" />
                        </graphics>
                  </initialMarking>
            </place>

            <transition id="t1" type="timed"
                        distribution="exponential" rate="1.0">
                  <graphics>
                        <position x="206" y="77" />
                  </graphics>
                  <name>
                        <value>null</value>
                        <graphics>
                              <offset x="-15" y="-15" />
                        </graphics>
                  </name>
            </transition>

            <arc id="a1" source="p1" target="t1">
                  <graphics>
                        <position x="170" y="75"/>
                  </graphics>
                  <inscription>
                        <value>1</value>
                        <graphics>
                              <offset x="-15" y="+20"/>
                        </graphics>
                  </inscription>
            </arc>

      </net>
```

Figure 4.3: A Petri net and its corresponding PNML description

4.2.1 JAXB or JAXP?

There are two packages for handling XML input and output in Java available from Sun Microsystems: Java Architecture for XML Binding [JAXB] and the Java APIs for XML Processing [JAXP].[1] JAXB creates a two-way mapping between XML documents and Java objects. It does this through a user-provided schema which defines how XML elements relate to the attributes of the Java classes which they describe [JAXB]. This is then used by the JAXB compiler to generate classes which have the in-built ability to be created from XML and to create an XML representation of themselves through their `unmarshal()` and `marshal()` methods respectively. This removes the need for the user to write their own code to parse and furthermore guarantees that the XML which is produced will be valid [JAXB].

JAXP is a package which provides a variety of methods for parsing XML, all of them using Crimson as the reference implementation. It provides Java classes which are used by the programmer to implement two of the most common standards for XML parsing: Document Object Model (DOM) and the Simple API for XML version 2 (SAX2). As described in [JDC], DOM parses the entire XML document into a `Document` object in memory. This object contains a tree of `Nodes` which correspond to the elements between the tags of the XML representation. These nodes can then be manipulated to extract or modify the data held in the XML document [JDC]. It is also possible to create new XML documents through the DOM API.

SAX2 employs a very different method which is also described in [JDC]. As a SAX2 parser reads through an XML document it calls event handlers when certain tag types, for example those marked as the start (<..>) or the end (</..>), are encountered. In order to implement a SAX2 parser, therefore, the user has to specify what should occur when these handlers are called. SAX2 has no in-built means of creating XML documents as DOM does, and it does not “remember” what has been read by previous executions of the same event handler unless the

[1] Both are available from http://java.sun.com/xml

user provides appropriate storage variables [JDC]. If the XML document is very complicated this method can be difficult, but SAX2 has the advantage over DOM in that it is faster and that it uses less memory. This is because it does not have to create and maintain a representation of the entire document in memory.

It was decided to use the SAX2 parsing method in Medusa. As the complexity of a PNML document (defined as this different types of XML element which it can contains) is fairly low even for large Petri nets, the added complexity of JAXB and DOM out-weighed their benefits. The lack of an in-built XML document generator in SAX2, a feature which is present in JAXB and DOM, was not a draw-back as generating a PNML description of a net from the Medusa representation can be accomplished without too much trouble using standard ASCII character-to-file writing techniques (see below).

It must be recognised, however, that JAXB has much to recommend it, especially the ease with which objects can be created from XML documents and vice-versa. It suffers, however, from being an experimental technology which does not work under Windows. This negates the platform-independence which Medusa enjoys as a Java application. Also, it has to be written into the application from the beginning.

4.2.2 XML Parsing in Medusa

This comprises two parts: writing out files in XML and parsing them back in to recreate saved nets.

From Figure 4.3 it will be seen that a PNML representation of a Petri net can be generated easily from the way in which they are described internally be Medusa. It is achieved simply by writing out ASCII characters to a file through a standard output stream, first of all describing all places (the contents of the `places` Vector) then all transitions (the `immediateTransitions` and `timedTransitions` Vectors) and finally all arcs (the `arcs` Vector). The

attributes of each place, transition or arc correspond to the fields of the PNML in an obvious way. The Medusa name of each place becomes its PNML "id", whilst the comments become the PNML "name". Similarly, the location of each element stored by Medusa is written in the `<position>` tag. The division of transitions in Medusa between two `Vectors`, one for immediate and one for timed, does not create a problem even though a PNML document written in this manner will not describe the transitions in ascending numerical order. All transitions are still defined before they are used in the descriptions of the arcs and so no problems arise.

Parsing the data back in is slightly more complicated, however. The SAX2 method parses the document as it is read based on the contents of the tag which it is currently reading. For example, when a `<place>` tag is encountered, the XML parser calls the `startElement()` handler. The values stored in the XML are then read out as they are encountered and stored in appropriately named local variables of the parser. When a `</place>` tag is encountered, signifying that the end of that place has been reached, the `endElement()` handler is called and a new `Place` object is created using the values from these variables. The parser then continues through the document. This method is safe as it can be guaranteed that well-formed PNML will not start the description of another Petri net node in the middle of another node description – for example, all the information needed to describe a place is always contained between a `<place> </place>` set of tags and they are never interleaved. This method has proved successful for retrieving complex models such as the Courier protocol as described in Chapter 6.

4.3 Reflection: Loading and Running Modules

Having described the use of XML in Medusa as a saved-file format, the mechanism by which user-designed modules are loaded and then executed is described. This process makes use of the Java Reflection API which provides a mechanism by which another program can discover information about a Java

`class` which is not known until runtime. As described in [CWH01], this information includes:

- The class of an object.
- The class's modifiers, fields, methods, constructors, and superclasses.
- Which constants and method declarations belong to an interface.

Reflection also permits the user to manipulate classes and objects about which nothing is known until runtime. This includes:

- The creation of an instance of a class (an object).
- Getting and setting the value of a field of the resulting object.
- Invoking a method on that object [CWH01]

Medusa uses these abilities of the Reflection API when it loads and runs modules. As all modules must implement the `Module` interface the method which runs the module is known to Medusa at compile-time. The name of the module's constructor is not known, however, so this must be discovered before an instance of the module can be created.

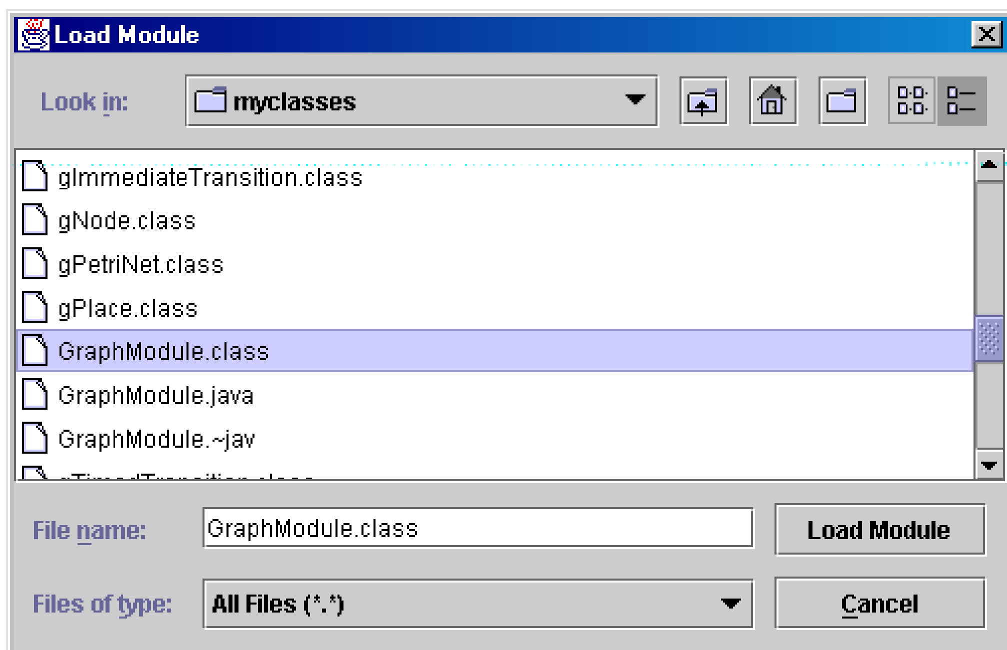


Figure 4.4: The `Dialog` in which the user selects which module to load.

The name of the module which the user wishes to load is selected, as detailed above, through a `file_chooser Dialog` as shown in Figure 4.4. The value which is returned from this dialog is the name and location of the module file which includes the `runModule()` method. It is passed as `className` in to a function which uses Reflection to create an instance of this class. The implementation in Medusa limits the modules to having a no-argument constructor, otherwise the process of loading a module would have become very complicated indeed. This is not a problem, however, as the name of the XML file they must parse is specified in the `Module` interface which they implement. Any other information required to execute the model should be specified by the programmer or inferred from the contents of the PNML description.

Once the module has been instantiated in the manner it is also run using Reflection. The general scheme for invoking a method through Reflection, detailed in [CWH01] is:

1) Create a `Class` object which corresponds to the class of the object which contains the method you wish to invoke.

2) Create a `Method` object by invoking `getMethod` on the `Class` object. The `getMethod` method has two arguments: a `String` containing the method name, and an array of `Class` objects which correspond to the parameters of the method.

3) Invoke the method by calling `invoke`. The `invoke` method has two arguments: an array of arguments to be passed to the invoked method and an object which class declares or inherits the method.

The body of `runModule()` is written by the writer of the module and may call other functions in the Module in the normal manner. Note that `runModule()` is parameterless and as such the arrays mentioned in 2) and 3) above will be empty.

The `action_listener` of the "Run Module" item on the "Module" menu is connected to a `doAMethod()` method which is uses Reflection as detailed above to execute the methods of a module. Clicking on "Run Module", therefore, causes `runModule()` to be called and the function to run. The Object which is passed in when it is called is an instantiated `Module` and the method name is `runModule`. When `runModule()` is called by this mechanism, the module executes according to the code written in that function.

The next two chapters detail the kinds of tasks which can be performed by this code. First of all, the graph theory analysis shows how modules can be written to perform specific tasks themselves. The DnamacaModule, however, shows how the extendable nature of Medusa allows it to interface with existing pieces of software. This module also provides the opportunity to correct flaws in an existing tool which attempts to accomplish the same thing.

Chapter 5: Design and Implementation of a Module for the Analysis of Place-Transition Nets Using Graph Theory

5.1 Introduction

The choice of modules to be implemented was chosen to demonstrate the possibilities offered by Medusa's extensibility. The graph theory analysis module was chosen as a topic as it demonstrates that Medusa's functionality could be extended by the provision of purpose-written modules. Such a module would allow the user to investigate exactly the attributes of the Petri net in which he was interested. This chapter starts with an introduction to the mathematics of graph theory. The analysis of the Petri net is accomplished through the analysis of its reachability set. This section is included to ensure that the reader understands the principles on which this module functions. The central implementation issue for this module was the choice of algorithm used to check the coverability graph for strongly connected components. A number of options were available but it is felt that the one selected best suits the needs of this method of analysis. In order to show that the implementation of the selected algorithm is correct, this chapter closes by comparing a set of correct results for a graph against those produced by the implementation.

5.2 Properties of Petri Nets Which Can Be Analysed Using Graph Theory

One way in which the properties of a Petri net are often analysed is through the application of graph theory to their reachability sets. The term 'reachability set' was defined informally in Chapter 2 as the set of all markings which are reachable from the initial marking, though a formal definition of it is given below. There are numerous properties of Petri nets about which one may be interested. For Place-Transition nets, the list below compiled from [BK95] illustrates the most common:

- **boundedness**. A net is **bounded** if there is a finite limit on the number of tokens on every place. Obviously, source (transitions which have no input places and can thus always fire) elements prevent a net from being both bound if it is live.
- **safeness** is of interest if the places in a net represent conditions, and so it follows that the presence or absence of tokens represents these conditions as being satisfied or not. A net is **safe** if there is at most one token on each place.
- **liveness** concerns the firing of transitions. If a net is **live** then no reachable marking exist such that a transition is never enabled again. In the same way that sources prevent boundedness, sinks (places which have no outgoing arcs) prevent liveness in a bounded net.

These properties can be formalised thus:

Definition 5.1 *Let* $PN = (P, T, I^-, I^+, M_0)$ *be a Place-Transition net*

- *The* reachability set *of PN is defined by* $R(PN) := \{M \mid M_0 \rightarrow^* M\}$. *If PN denotes an unmarked Place-Transition net of if we want to consider parts of the reachability set, the set of reachable markings for a given marking* $\tilde{M}$ *will be denoted by* $R(PN, \tilde{M}) := \{M \mid \tilde{M} \rightarrow^* M\}$. *Thus for a marked Place-Transition net we have* $R(PN) = R(PN, M_0)$.
- *PN is a* bounded Place-Transition net, *iff* $\forall p \in P : \exists k \in \mathbb{N}_0 : \forall M \in R(PN) : M(p) \leq k$.

  PN is safe, *iff* $\forall p \in P : \forall M \in R(PN) : M(p) \leq 1$.
- *A transition* $t \in T$ *is* live, *iff* $\forall M \in R(PN) : \exists M' \in R(PN) : M \rightarrow^* M'$ *and* $M'[t>$.
- *PN is* live, *iff all transitions are live, i.e.* $\forall t \in T, M \in R(PN) : \exists M' \in R(PN) : M \rightarrow^* M'$ *and* $M'[t>$.
- *A marking* $M \in R(PN)$ *is a* home state, *iff* $M' \in R(PN) : M' \rightarrow^* M$. [BK95]

A necessary condition for a net to be live and bounded is that it is strongly connected. Otherwise a net is said to be weakly connected. Informally, it is said that two nodes, x and y, of a net are weakly connected if and only if x can be reached from y or y can be reached from x. The nodes are strongly connected if and only if x can be reached from y and y can be reached from x. As a Place-Transition net can be treated as a directed graph, this can be formalised as follows:

Definition 5.2 *Let* $PN = (P,T,\Gamma^-,\Gamma^+,M_0)$ *be a Place-Transition net*

- *Input places of transition t are defined as:* $\bullet t := \{p \in P \mid \Gamma^-(p, t) > 0\}$,
- *Output places of transition t:* $t\bullet := \{p \in P \mid \Gamma^+(p, t) > 0\}$,
- *Input transitions of place p:* $\bullet p := \{t \in T \mid \Gamma^+(p, t) > 0\}$,
- *Output transitions of place p:* $p\bullet := \{t \in T \mid \Gamma^-(p, t) > 0\}$,
- $F \subseteq (P \times T) \cup (T \times P)$ *given by* $F := \{(x, y) \mid x, y \in P \cup T : x \in \bullet y\}$ *is called the* flow relation *of PN.*

  Let F^* *denote the reflexive and transitive closure of F, i.e.* $x, y, z \in P \cup T$:

  a) $(x, x) \in F^*$

  b) $(x, y) \in F \Rightarrow (x, y) \in F^*$

  c) $(x, y) \in F^*$ *and* $(y, z) \in F^* \Rightarrow (x, z) \in F^*$
- *PN is* weakly connected *iff* $x, y \in P \cup T : xF^*y$ *or* yF^*x,
- *PN is* strongly connected *iff* $x, y \in P \cup T : xF^*y$ *and* yF^*x.

  [BK95]

5.2.1 Analysis of Place-Transition Nets Using Graph Theory

There are two common ways in which the reachability set of a Place-Transition net can be drawn. The first is as a reachability tree, whose nodes are the markings of the net. Figures 5.1 and 5.2 show a Place-Transition net and its corresponding reachability tree.

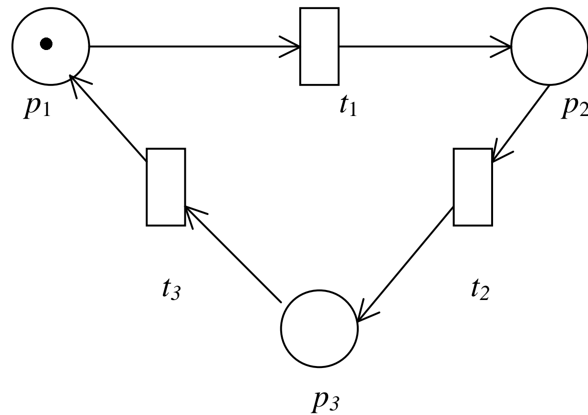


Figure 5.1: A Place –Transition net.

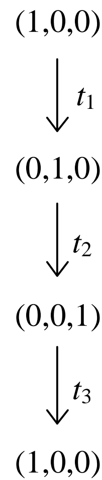


Figure 5.2: The reachability tree of the net in Figure 5.1.

The process by which the reachability tree is constructed is described in [BK95]. The starting node is the initial marking of the net. From this, the directly reachable markings are added as leaves and their directly reachable markings are in turn calculated. These are then added as further leaves and so on until a previously generated marking is encountered.

The second format is as a reachability graph. This is a direct transformation of a reachability tree achieved by deleting duplicate nodes and connecting the remainder up appropriately [BK95]. The reachability graph which corresponds to the reachability tree in Figure 5.2 is shown in Figure 5.3.

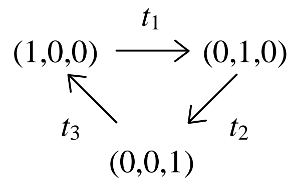


Figure 5.3: The reachability graph of the net in Figure 5.1.

The reachability graph of a bounded net like that shown in Figure 5.3 is relatively straightforward to generate but problems are encountered when trying to generate for unbounded nets as they are infinitely large. To overcome this and make the generation of reachability graphs for unbounded nets possible, the symbol ω is used to in texts such as [BK95] to represent the infinite marking on an unbounded place. By using ω and Algorithm 5.1, a finite representation of the reachability tree can be generated for both bounded and unbounded nets. This is known as a coverability tree, and in the case of a bounded net is identical to that net's reachability tree [BK95].

Algorithm 5.1 *to generate the coverability tree of a Place-Transition net:*

X := {M_0} *// M_0 is the root of the coverability tree*
while X ≠ Ø do
begin

Choose x ∈ X.
∀t ∈ T : x[t > do
create a new node x' given by x[t > x' and connect x and x' by a directed arc labelled with t.
Check ∀p ∈ P :
If there exists a node y on the path from M_0 to x' with y ≤ x' and y(p) < x'(p) then set x'(p) := ω.
X := {x | x is a leaf of the coverability tree generated so far, in x at least one transition is enabled and there is no non-terminal node y with y = x }

end [BK95]

The coverability tree can then be converted into a coverability graph in the same way as a reachability tree is converted into a reachability graph. Analysis of this

coverability graph allows us to deduce the following things about the Place-Transition net to which it belongs:

- the net is **bounded** if and only if no node in its coverability tree is marked with the symbol ω [BK95].
- if the net is bounded, it is **live** if and only if all transitions appear as a label in all final strongly connected components of the coverability graph. A strongly connect component is final if there are no arcs which leave that component [BK95].
- if the net is bounded, a **home state** exists if and only if its coverability graph contains exactly one final strongly connected component. It is not the case, however, that if a net is unbounded then it cannot have a home state [BK95].

One problem with graph-theory analysis is that the coverability graphs generated for even relatively simple nets can be very large – this is called the state space explosion problem in [BK95]. There exists a method which addresses this problem (invariant analysis) but it is beyond the scope of this project. A module capable of performing invariant analysis on complex nets has been produced and full details of it are given in Chapter 7.

5.2.2 Analysis of GSPNs Using Graph Theory

The analysis of GSPNs using the theories outlined above is rendered difficult by the fact that such nets possess two forms of transitions. As has been said, the firing of enabled immediate transitions has priority over that of timed transitions, which leads to the existence of vanishing and tangible markings (see Chapter 2). Is it impossible to generalise about the attributes of a GSPN from analysis of its underlying Place-Transition net, except to say that if the Place-Transition net is bounded then so to is the GSPN [BK95]. Due to these complications, the graph theory module is designed for the analysis of Place-Transition nets only.

5.3 Implementation of the Module

The first step is to construct the coverability tree as per Algorithm 5.1. The coverability tree must then be converted into a coverability graph. Once this has been accomplished it can be analysed. This analysis centres around the identification of the strongly connected components of the graph, a task for which there exist a number of common algorithms. The following section will therefore concentrate on describing the selection of the algorithm which was adopted as this was the key issue faced.

5.3.1 Analysis of the Coverability Graph

The coverability graph is analysed to discover if the net is bounded and live and if the net has a home state. This section will describe how such analysis is performed in this module. Checking to see if the net is bounded is relatively simple. As will be recalled, the net is bounded if no marking in the coverability graph contains the symbol ω. This translates rather obviously into the pseudocode shown in Figure 5.4.

```
private boolean isBounded() {
      for (every node in the coverability graph) {
            for (every place in that node) {
                  if (the marking on that place is "ω")
                        return false;
                  }
      }
      return true;
}
```

Figure 5.4: Pseudocode representation of the function for analysing if a net is bounded.

Conducting the analysis necessary to check for the existence of the other two conditions is more involved, however. As per the definition of the three conditions above, this analysis is only necessary if the net is bounded. If the net is not bounded then analysis will terminate here. Assuming the net is bounded, the strongly connected components of the coverability graph must be identified.

The coverability graph can be treated as a **directed** graph as defined in Definition 5.3.

Definition 5.3 *a* **directed** *graph or* **digraph** *is a pair* $G = (V, E)$ *where:*

- *V is a set whose elements are called* **vertices** *or* **nodes**,
- *E is a set of ordered pairs of elements of V which are called* **edges**, **directed edges** *or* **arcs**,
- *For an arc* (v, w) *in E, v is its* **tail** *and w is its* **head***:* (v, w) *is represented in diagrams as* $v \rightarrow w$ *and is written vw.* [BG00]

The classic algorithm for identifying strongly connected components in a directed graph was designed by R. E. Tarjan:

Algorithm 5.2 *Tarjan's algorithm to detect the strongly connected components of a directed graph* $G = (V, E)$ [NSS94]

```
procedure VISIT(v);
begin
        root[v] := v; InComponent[v] := False;
        PUSH(v, stack);
        for each node w such that (v, w) ∈ E do begin
                if w is not already visited then VISIT(w);
                if not InComponent[w] then root[v] := MIN(root[v], root[w])
        end;
        if root[v] = v then
        repeat
                w := POP(stack);
                InComponent[w] := True;
        until w = v
end;

//Main program
begin
        stack := ∅;
        for each node v ∈ V do
                if v is not already visited then VISIT(v)
end.
```

Tarjan's algorithm applies a recursive function *VISIT* to every node in the graph which has not already had *VISIT* applied to it [NSS94]. The algorithm aims to

find the root of every strongly connected component in the graph. The root of each strongly connected component is defined as the first node which *VISIT* enters in that component [NSS94]. To accomplish this the algorithm performs two interleaved traversals of the graph undergoing analysis [NSS94]. The first is a depth-first search of all edges [NSS94]. The second is accomplished using a stack on which each node is stored when it is discovered by the first traversal [NSS94]. When a root of a strongly connected component is found all of its descendants which are not part of a previously identified strongly connected component are marked as belonging to the root's component [NSS94]. When the root of a component is exited all the nodes down to this root are popped off the stack and are taken as forming that component [NSS94].

There have been a number of proposed refinements of Tarjan's algorithm, for example two are presented in [NSS94]. These focus on improving its efficiency by reducing the number of nodes stored on the stack during the second traversal and thus finding the strongly connected components faster. The first of the refined algorithms does not use the stack when exploring strongly connected components which consist of only one node. The second is a further refinement but is of little use in this particular application as it only stores the roots of the components and not the nodes which make them up. This would make checking for liveness very hard. For the purposes of this module, however, Tarjan's original algorithm would be sufficient as the size of the coverability graph is constrained by the state-space explosion problem and so the improvements are unlikely to have too great an impact.

There is also a variation on Tarjan's algorithm by M. Sharir presented in [BG00]. This takes an array of adjacency lists as its representation of a directed graph. Each vertex of the graph is assigned a unique integer identifier which is used as the index in an array of adjacency lists. Each entry in the array is a linked list of integers which records to which vertices the vertex is connected. This can perhaps be more easily understood when presented as in Figure 5.5.

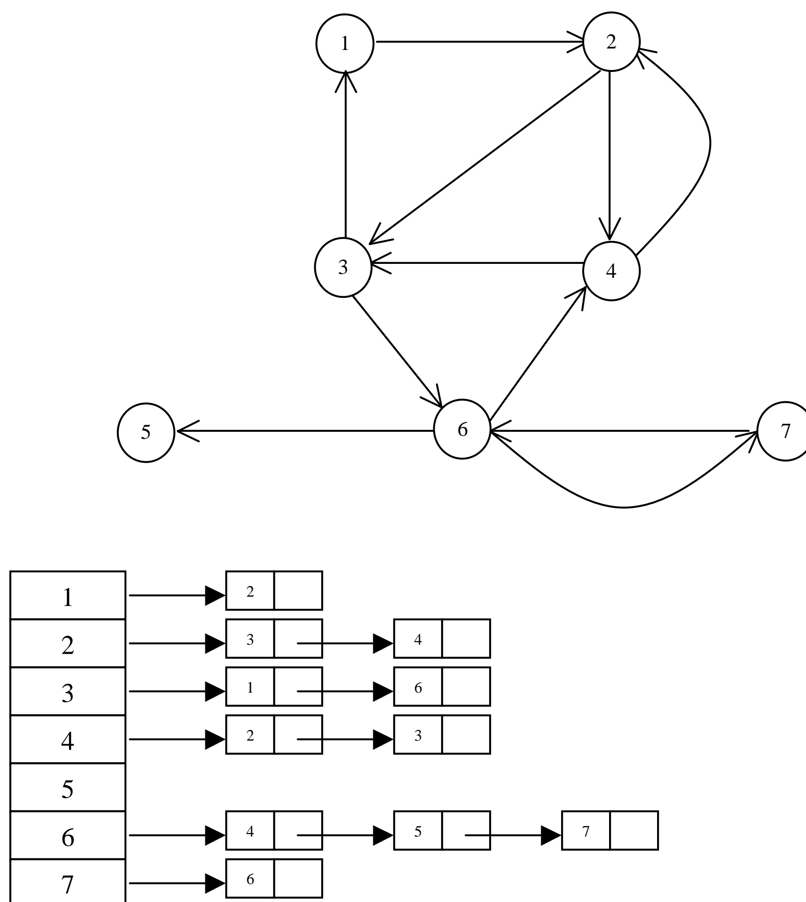


Figure 5.5: A directed graph and its corresponding adjacency list representation [BG00].

This algorithm described in [BG00] is divided into two phases. First of all, a depth-first search of the directed graph G represented as an array of adjacency lists is conducted and the vertices are pushed onto a stack as they are encountered [BG00]. Once this search is completed the algorithm moves into the second phase. The transpose graph of G, denoted G^T, is employed in the second phase. G^T is formed by reversing the direction of every arc in G, which can be accomplished from the adjacency list structure of G [BG00]. A depth-first search of G^T is then performed and from this the strongly connected components are identified [BG00]. Once again a stack is employed to accomplish this. The strongly connected components are identified by the index of the vertex encountered by the algorithm in that component [BG00].

It was decided to select the algorithm which is presented in [BG00] because of the provision of extensive pseudocode and explanatory notes in that text. This greatly aided implementation. The improved versions of Tarjan's algorithm presented in [NSS94] have much to recommend them, especially their improved efficiency over the other methods. It is arguable, however, whether or not their improvements would be noticeable given the limited scale of the problems which this module, constrained as they are by state-space explosion, could be expected to address. Furthermore, the algorithms are not expanded upon in too much detail and so implementation would have been more time consuming.

5.3.2 Validation of the Implementation of the Chosen Algorithm

The implementation of this algorithm can be validated against an example provided in [BG00]. Consider the directed graph in Figure 5.6. Its strongly connected components are (1,2,4,6), (5,7) and (3).

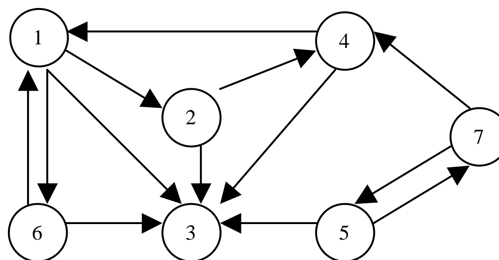


Figure 5.6: A directed graph with strongly connected components [BG00].

The implementation outputs the identifier of each vertex along with the first vertex of that vertex's strongly connected component. The result generated is:

```
Vertex 1 is in the same scc as vertex 2
Vertex 2 is in the same scc as vertex 2
Vertex 3 is in the same scc as vertex 3
Vertex 4 is in the same scc as vertex 2
Vertex 5 is in the same scc as vertex 5
Vertex 6 is in the same scc as vertex 2
Vertex 7 is in the same scc as vertex 5
```

It can be seen, therefore, that the implementation of the algorithm correctly identifies three strongly connected components consisting of vertices (1,2,4,6), (5,7) and (3). In this case the first vertex of each component is 2, 5 and 3 respectively. This shows that the implementation used in the module is correct.

5.3.3 Completing the Analysis

Having settled upon the choice of the algorithm to identify which vertices belong to which strongly connected components, these components can now be analysed to check for liveness and the existence of home states. This requires that final strongly connected components are identified from amongst the strongly connected components. Recall that a final components is one which has no outgoing arcs. The pseudocode for a function which checks if a component is final is given in Figure 5.7. It assumes that the graph is defined in the adjacency list structure shown in Figure 5.5.

```
public boolean isFinal() {
      for (every vertex in the component) {
            if (the vertex is connected to another not in
            the same component) {

                  //this is achieved by comparing the
                  //vertices in the vertex's adjacency list
                  //with those vertices known to be in this
                  //component

                  return false;
            }
      }
      return true;
}
```

Figure 5.7: Pseudocode for a function which checks if a strongly connected component is final.

As a home state only exists if the coverability graph contains exactly one final strongly connected component, the existence of home states can be checked for by assessing each component using the routine in Figure 5.7 and recording how

many "true" values are returned. If the number is not one, the net with that coverability graph does not have a home state.

For a net to be live every transition must appear as a label in every final strongly connected component. Using the routine in Figure 5.7 the final components are identified and analysed by the routine shown in Figure 5.8.

```
public boolean isLive() {
      for (every final strongly connected component) {
            List transitions = new List();
            for (every marking in the component) {

                  //retrieve the transition which fired to
                  //produce the marking - this is stored
                  //when the graph is constructed

                  transitions.add(transition.name);
            }
            //compare the list for this component with a
            //pre-generated list of all transition names

            if (there is a transition of the net which does
            not appear in "transitions")
                  return false;
      }
      return true;
}
```

Figure 5.8: Pseudocode representation of the function which checks if the coverability graph is that of a live Place-Transition net.

Chapter 6: Design and Implementation of a Module to Interface With Dnamaca

6.1 Introduction

DnamacaModule was selected as a topic to demonstrate that it is possible to design a module to allow Medusa to interface with a pre-existing piece of software. This also presented the opportunity to correct a flaw in another Petri net editor (DaNAMiCS) which also produces output files in Dnamaca format, but not in a correct manner. It was hoped to incorporate features lacking from DaNAMiCS as well, such as a front-end which would allow the user to specify exactly the performance measures which they wished to have analysed.

Dnamaca is a Markov chain analyser written by Dr. William Knottenbelt capable of generating performance analysis results for GSPNs [KNO96]. The theory of Markov analysis is not covered here as it is not essential to the understanding of the design of DnamacaModule, but readers wishing to know more are directed to [KNO96], [KNO99] and [BK95]. Very broadly, Markov analysis deals with the states of a system, for example what the probability of it being in a certain state at some time is. For Petri nets, the states of the system are the markings through which it passes when enabled transitions are fired.

Dnamaca functions by parsing an input file containing a textual description of the net to be analysed into its internal representation, performing the analysis on this representation and producing an output file detailing the results. The key issue was the design of a module which would render a Petri net produced in Medusa into an input file to be read and processed by Dnamaca. DnamacaModule, therefore, is responsible for automatically generating a Dnamaca input file from the Petri net currently being edited in Medusa, allowing the user to add the performance measures desired, running Dnamaca itself and finally presenting the results back to the user. The process can be summarised thus:

1) DnamacaModule parses the `current.xml` file produced by Medusa
2) From this, it automatically generates a description of the net in the model description format of Dnamaca.
3) It then gives the user chance to add the performance measures to be analysed. These and the model description are contained in a file called `current.mod`.
4) Dnamaca is invoked by DnamacaModule with `current.mod` as its input file.
5) The results produced by Dnamaca held in the `current.mod.out` file are presented back to the user.

This chapter comprises three main sections. An overview of how nets are inputted into Dnamaca is given, including details of how DaNAMiCS is flawed. The implementation of DnamacaModule is then covered, including how this flaw was rectified and how invoking another command line program is done in Java. The success or otherwise of the implementation is assessed in the chapter which follows.

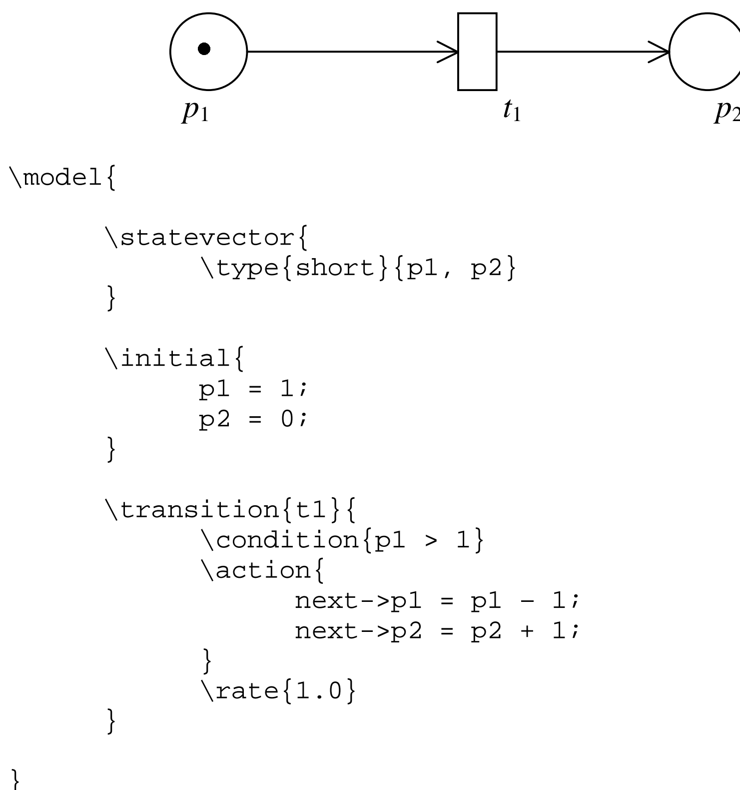
6.2 Dnamaca Input Files

This section shows how a Petri net is defined in an input file for Dnamaca. An input file contains two sections: the first half describes the structure of the net (the model description) whilst the second details the attributes of the net which the user wishes to have analysed (the performance measures).

6.2.1 Dnamaca Model Descriptions

As the Markov chain which underlies a GSPN may contain a very large number of states it would not be practical to describe each one of these individually [KNO99]. Instead, Dnamaca employs a high-level description which specifies the components of the system (the state description vector), the system's initial state and the rules by which it moves between states [KNO99]. These correspond

respectively to the places which make up the net, the initial marking of these places and a description of each transition including the conditions under which it fires and the resulting state when it does [KNO99]. It is possible to write a program which generates such a description automatically when given a Petri net – indeed this is what the DaNAMiCS export function and DnamacaModule both do. Figure 6.1 shows a simple Place-Transition net and its equivalent model description.




```
\model{

	\statevector{
		\type{short}{p1, p2}
	}

	\initial{
		p1 = 1;
		p2 = 0;
	}

	\transition{t1}{
		\condition{p1 > 1}
		\action{
			next->p1 = p1 – 1;
			next->p2 = p2 + 1;
		}
		\rate{1.0}
	}

}
```

Figure 6.1: A Place–Transition net and its corresponding Dnamaca model description.

The way in which a transition is described merits some explanation. The `condition` describes the state which the system must be in for this transition to be enabled. It can be seen that it corresponds to the number of tokens which must be present on the transition's input places to enable it. The `action` describes the state which results from the transition firing in terms of the number of tokens which are created and destroyed on the transition's output and input places.

It is in the action section that model descriptions generated automatically by DaNAMiCS can exhibit flaws. Consider the net in Figure 6.2. Notice that the arc connecting t_1 and p_1 is directed both ways, and that there is a token on p_1 and as such t_1 is enabled. This means that t_1 can always fire and when it does one token is destroyed on the transition's input places and one is created on each of its output places. As p_1 is both the input and output place of t_1, when the transition is fired the number of tokens on p_1 does not change.

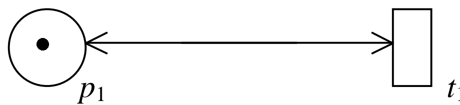


Figure 6.2: A net which would be described incorrectly by DaNAMiCS.

The correct description for t_1 would be:

```
\transition{t1}{
      \condition{p1 > 1}
      \action{
      }
      \rate{1.0}
}
```

Notice that the action description is empty. This is because it describes the state which the system will be in after the transition fires, and as has been shown above the firing of t_1 leaves the system's state unchanged. If the net is entered into DaNAMiCS and then exported as a Dnamaca model description, the following is produced:

```
\transition{t1}{
      \condition{p1 > 1}
      \action{
            next->p1 = p1 - 1;
            next->p1 = p1 + 1;
      }
      \rate{1.0}
}
```

This appears to be correct as it describes what occurs: a token is destroyed on p_1 and then another is created there. Dnamaca, however, deals with the state which results from the firing of a transition and so expects a place to feature at most

once in the action description. When it parses a description with a repeated place it only takes in the final one. This means that if Dnamaca was given the description above it would interpret it as meaning that the firing of t_1 increases the number of tokens on p_1 by one each time.

It is not hard to see how such an error could occur. In most cases, considering the effects of firing a transition on first its input and then its output places would give correct results. To ensure that the results are always valid, however, any attempt to generate a model description automatically must deal with the overall effect of firing a transition on the state of the system (that is to say, the markings on the places). An approach which achieves this, and thus avoids replicating the mistake of DaNAMiCS, has been implemented in DnamacaModule and is detailed below.

6.2.2 Dnamaca Performance Measures

Unlike Dnamaca model descriptions, which can be generated automatically from a representation of a Petri net, the performance measure part of the input file must be supplied by the user. Performance measures are either state measures or count measures. State measures are used to indicate to the performance analyser the real expressions for which the user wishes to have results generated. This includes measures such as the average number of tokens on a place [KNO99]. They are described in the form:

```
\statemeasure{<identifier>}{
      \estimator{<any combination of “mean”, “variance”,
                 “stddev” and “deviation”>}
      \expression{<expression to be analysed>}
}
```

Count measures are used to indicate in which rates of event occurrence the user is interested, for example the rate at which a transition fires and produces throughput [KNO99]. They are expressed thus:

```
\countmeasure{<identifier>}{
      \estimator{mean}
      \precondition{<a boolean expression>}
      \postcondition{<a boolean expression>}
      \transition{<either "all" or a list of those
      required>}
}
```

It is the responsibility of the user to specify those which they desire. In DaNAMiCS this is not possible as the whole Dnamaca input file is generated automatically with a set of default performance measures. DnamacaModule, however, incorporates a mechanism which allows the user to specify the measures which they wish to have analysed through a series a GUIs. This system is detailed below.

6.3 DnamacaModule

Because performance measures are supplied by the user, they are not hard to accommodate. The automatic generation of a correct model description is a far more involved process and it is on this that the bulk of this section will focus. DnamacaModule is much more than a textual generator as it not only creates the correct input file but also runs Dnamaca with this file and presents the user with the results produced. How this invocation of another program is achieved is detailed below.

The first thing which DnamacaModule must do is parse the `current.xml` input file. It accomplishes this by using the same SAX2 parser method as Medusa does to load saved nets. DnamacaModule's internal representation of a net is identical to Medusa's (see Figure 3.1). Medusa's class structure was repeated as it could easily be applied to the task of generating Dnamaca input files, but functions such as those responsible for drawing the Petri net were unnecessary and were consequently deleted.

6.3.1 Model Description Generation

DnamacaModule utilises the incidence matrices of the net which is to be analysed to generate the model descriptions. These matrices are also used in invariant analysis as described in [BK95]. There are three of these matrices: the backward incidence matrix (C^-), the forward incidence matrix (C^+) and the incidence matrix (C). The matrices C^- and C^+ describe the incidence functions I^- and I^+ in matrix form. C^- describes the number of tokens which are destroyed on each place for each transition, whilst C^+ describes the number of tokens which are created [BK95]. For the Petri net in Figure 6.1, the corresponding matrices would be:

$$C^- = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad C^+ = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

The incidence matrix C is the combination of C^- and C^+. To be formal [BK95]:

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

For example, the C matrix for the net in 6.1 would be:

$$C = \begin{pmatrix} -1 \\ +1 \end{pmatrix}$$

The columns of the C matrix therefore describe the cumulative effect on the markings of all places created by the firing of a transition and it is this which DnamacaModule exploits. It will be recalled that the problem with the existing implementation in DaNAMiCS is that it considers first the input places and then the output places, which can lead to incorrect results with bi-directed arcs. It has already been said that the way to overcome this was to consider the overall effect of firing a transition when it is described, and the use of the C matrix permits this. This can be seen if the matrices for the net in Figure 6.2 are considered:

$$C^- = \begin{pmatrix} 1 \end{pmatrix} \qquad C^+ = \begin{pmatrix} 1 \end{pmatrix} \qquad C = \begin{pmatrix} 0 \end{pmatrix}$$

The value of the corresponding C matrix reflects the fact that the cumulative effect of t_1 firing is to leave the marking of p_1 unchanged – the number of tokens changes by 0.

The C matrix, therefore, provides an ideal mechanism for describing the overall effect of firing a transition and thus ensuring that bi-directed arcs are described correctly. In order to exploit this, DnamacaModule first calculates the C^- and C^+ matrices and then subtract one from the other to find C. All three matrices are represented as two-dimensional matrices in DnamacaModule's Java code. They are declared as:

```
int[][] C = new int[i][j]
```

where i is the number of places in the system and j is the total a number of transitions (immediate and timed). The relevant matrix is then passed into a routine which assigns the values based on the structure of the Petri net. Figure 6.3 shows the pseudo-code version of the function which calculates the C^- matrix, but the routine which assigns C^+ is identical save for the fact that it uses output places not input places.

```
private void calcC⁻(int[][] C⁻) {
      for (every transition tj) {
            for (every place pi) {
                  if (pi is an input place of tj) {
                        C⁻[i][j] = weight of arc from pi to
                                   tj;
                  }
                  else {
                        C⁻[i][j] = 0;
                  }
            }
      }
}
```

Figure 6.3: The pseudocode of the function which calculates the C^- matrix

The calculation of C is then achieved by matrix subtraction. A pseudocode representation of DnamacaModule's code is shown in Figure 6.4.

```
private void calcC(int[][] C, int[][] C+, int[][] C-) {
      for (every place pi) {
            for (every transition tj) {
                  C[i][j] = C+[i][j] - C-[i][j];
            }
      }
}
```

Figure 6.4: The pseudocode representation of the algorithm for calculating the *C* matrix.

Once *C* has been calculated it can be used directly in the creation of the Dnamaca input file. When describing transition t_j, a "next->p_i" line is only written out if the value of (pi, t_j) in *C*, which corresponds to the (i^{th}, j^{th}) element, is not equal to zero. Otherwise, the firing of transition t_j does not affect the marking of p_i and so nothing needs to be described. The pseudocode representation of the Java code for implementing this is shown in Figure 6.5.

```
for (every transition tj) {
      for (every place pi) {
            if(C[i][j]>0) {
                  write("next->pi = pi +" C[i][j]);
            }
            else if(C[i][j]<0) {
                  write("next->pi = pi -" abs(C[i][j]));
            }
      }
}
```

Figure 6.5: The pseudocode representation of the way in which DnamacaModule writes transition definitions.

Note that the `abs` expression in the `else if` statement is necessary to return the absolute value of *C*[*i*][*j*] as it will have a negative value and subtracting a negative number has the same effect as addition.

6.3.2 Performance Measures

Once the model description has been generated, DnamacaModule must collect the performance measures from the user. This process is achieved through a number of Swing `Dialogs` with fields corresponding to the elements which make up the measures (see Section 6.2.2). Figure 6.6 shows the form which these `Dialogs` take.

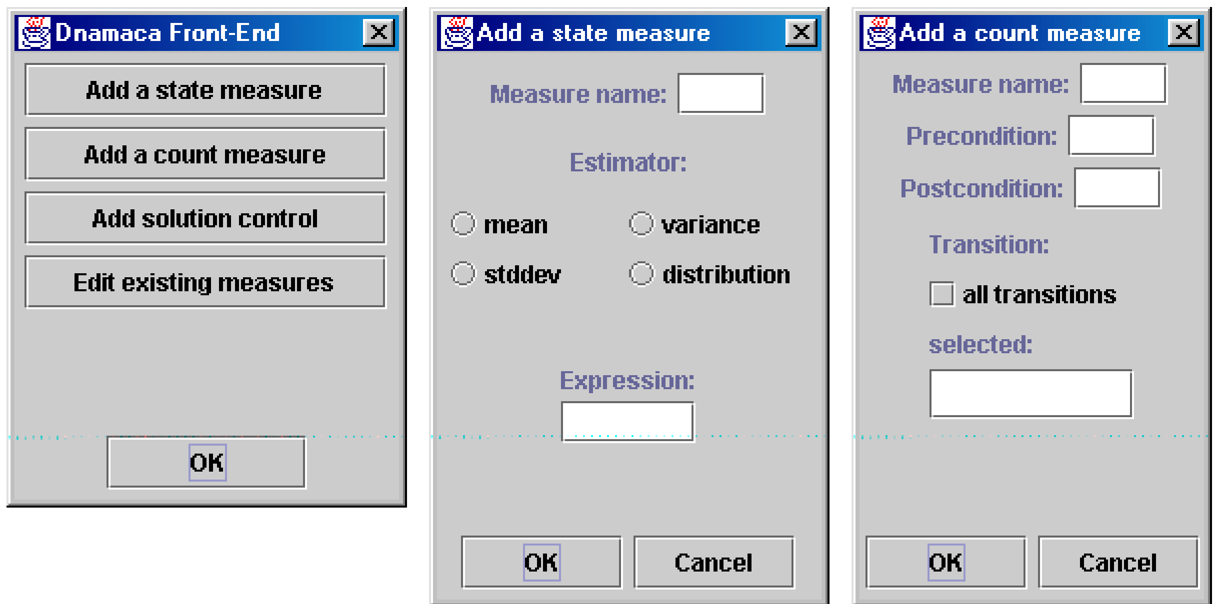


Figure 6.6 Screenshot of the `Dialogs` through which the user inputs the desired performance measures.

The fields in which the expressions are specified by the user are `JTextFields`, the contents of which can be retrieved as a `String` and written to a file. When the OK button is clicked on the `Dnamaca Front-End`, the model description and any performance measures are written to a Dnamaca input file called `current.mod`.

6.3.3 How DnamacaModule Invokes Dnamaca and Displays Results

When the model description has been generated and the performance measures specified, DnamacaModule automatically runs Dnamaca with the completed input file. It is worth explaining how this is accomplished. Dnamaca is a command-line program which is invoked with:

```
dnamaca <input filename>
```

Java offers a ready-made mechanism through which programs can invoke command-line applications, the intricacies of which are explained fully in [DAC00]. In order to run Dnamaca from DnamacaModule it is necessary first to retrieve the existing Java Runtime Environment as reference to a `Runtime` object [DAC00]. It is then possible to use the `Runtime.exec(String)` method to run Dnamaca where the command shown above is passed in as the `String` variable [DAC00]. As the file produced by DnamacaModule is always called `current.mod` this is a straightforward matter.

As [DAC00] explains, however, there are a number of pit-falls to this apparently simple procedure. Most importantly, the output streams produced by the program being invoked by `Runtime.exec()` must be handled and the invoked process's return value passed back to the invoking program or else the invoking program will hang. The invoked program will generate two output streams: one on the standard output stream for its results and one on the standard error stream if any problems occur. The output streams of the invoked program are treated as input streams by the invoker and so they must be redirected to the correct output streams again by the invoking program [DAC00].

DnamacaModule handles the Dnamaca's standard output stream first and only handles the error stream when Dnamaca stops writing to the standard stream. This is one of the methods described in [DAC00], but [DAC00] also details how threads can be used to read both output streams concurrently. This solution has much to commend it in terms of elegance but it was felt that the extra complexity which was involved, especially the introduction of threads into a program with a

Swing GUI (which is by definition not thread-safe [CWH01]), was greater than the possible benefits. The solution adopted handles all the output generated by Dnamaca in the correct way as the error stream is only written to when an error occurs, in which case no more writing to the standard output stream takes place.

The final task which DnamacaModule must perform is the presentation of the generated results back to the user. Dnamaca writes the results both to the screen and to a file. This file's name is always of the form `<inputfilename>.mod.out`, so the results from an execution of DnamacaModule are always to be found in `current.mod.out`. In order to display them to the user this file is read and displayed in a Swing `Dialog` as shown in Figure 6.7.

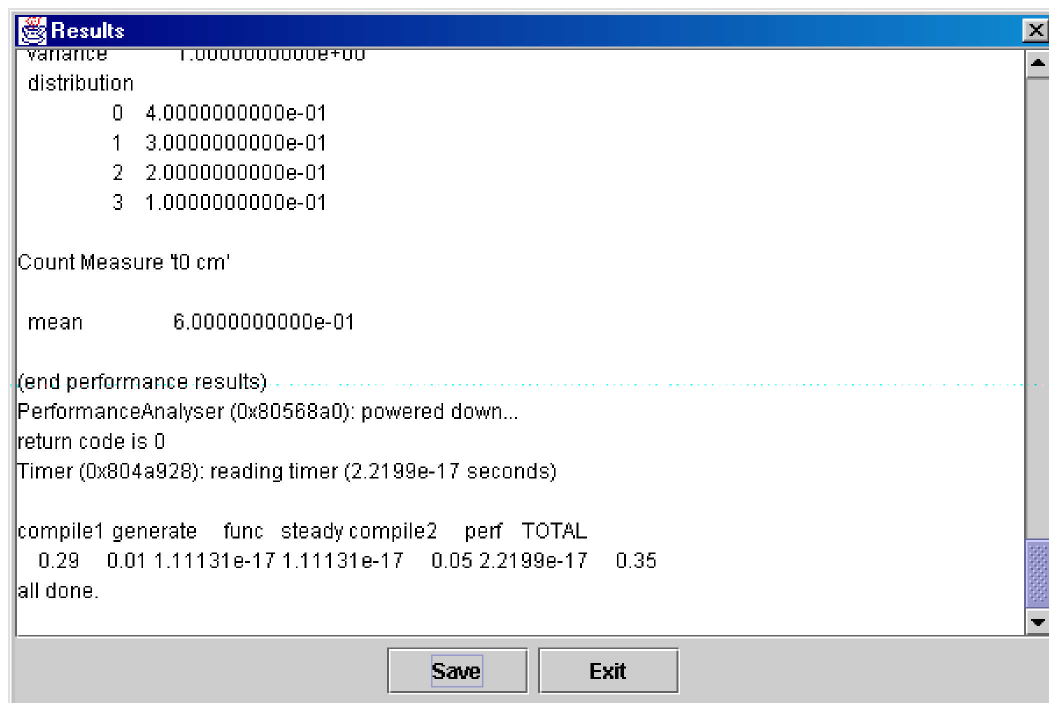


Figure 6.7: A screenshot showing how DnamacaModule presents the results generated by Dnamaca to the user

Having detailed the production of two modules a consideration of how successful the project has been must be undertaken. For DnamacaModule, this means assessing how accurate the model description which it generates are. This process will also assess Medusa's architecture because the model descriptions produced by DnamacaModule depend entirely on the description of the Petri net entered into Medusa. Ultimately, the aim of this validation process is to

demonstrate whether or not the main aim of the production of an extensible editor has been achieved. This is the topic of the following chapter.

Chapter 7: Validation of Concept and Design Through Analysis of Generated Results

7.1 Introduction

The aim of this chapter is to attempt to validate the design of Medusa and the modules. This will be achieved by using the modules to analyse models with documented results and comparing the results produced. If the results match then it can be concluded that the implementations are correct, whilst if they differ the implementations are flawed. This process will also evaluate how successful this project has been in achieving its aim of producing an extendable tool.

A module designed as part of [MW01] will be used in this process. This module was designed to be compatible with Medusa as it implements the same interface and takes PNML as an input file. It should, therefore, be able to be loaded and run by Medusa without problems and generate correct results. If this proves to be the case it can be concluded that the project has achieved its aim and that it is possible to extend Medusa knowing only the details of how it interfaces with its modules.

7.2 Graph Theory Analysis Module

The models chosen to validate this module were taken from [BK95]. The first to be tried was the simple net shown in Figure 7.2 with the corresponding reachability graph shown in Figure 7.3. The module calculated the reachability graph and from this identified that the net was bounded as no marking contained the ω symbol. Furthermore, the strongly connected component algorithm detected that there was only one such component and that this was final. This meant that the net had a home state. The net was correctly described as live as every transition appears as a label in that component. Figure 7.4 shows a screenshot of the module returning these results.

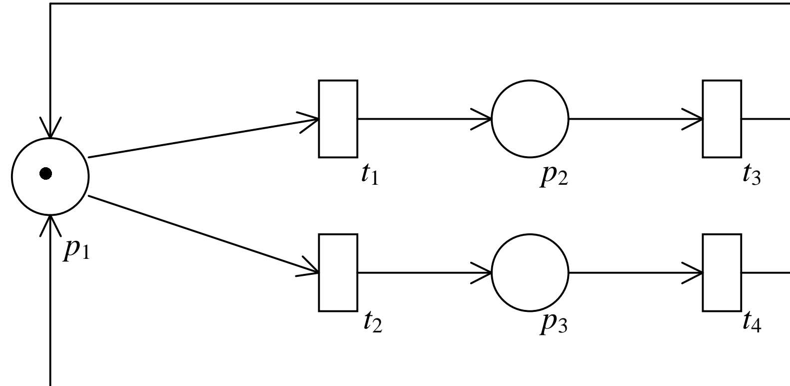


Figure 7.2: A Place-Transition net [BK95].

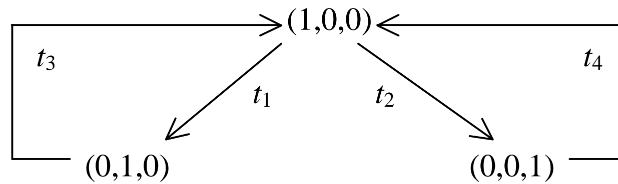


Figure 7.3: The reachability graph of the Place-Transition net in Figure 7.2 [BK95].

Medusa v1.7 - C:\JBuilder3\myclasses\twoOptions.xml
File Edit Modules Help
Results
Running GraphModule
Entering buildCoverabilityTree()
Exiting buildCoverabilityTree()
Checking to see if the net is bounded . . .
Net is bounded, moving on . . .
Entering convertToGraph()
Exiting convertToGraph()
Entering strongComponents()
Exiting strongComponents()
Vertex 1 is in the same scc as vertex 1
Vertex 2 is in the same scc as vertex 1
Vertex 3 is in the same scc as vertex 1
There are 1 strongly connected components
scc containing vertex 1 is final
There is a total of one strong connected components
Therefore the net has a home state
Checking labels . . .
In scc containing vertex 1, all transitions appear as labels
Therefore the net is live
Results:
The net is bounded, live and has a home state
GraphModule finished
Save Exit
Opened C:\JBuilder3\myclasses\twoOptions.xml
Start JAVA JBuilder Medusa v1.7 -... 12:15

Figure 7.4: Screenshot of Medusa analysing the net in Figure 7.2 with the graph theory module.

The module also described correctly the complex unbounded net shown in Figure 7.5. In this case as the net is unbounded the module does not attempt to check for liveness or the existence of a home state. It does, however, construct the reachability graph and recognise that the net is unbounded from this. Again, Figure 7.6 shows a screenshot of Medusa and the results returned by the module.

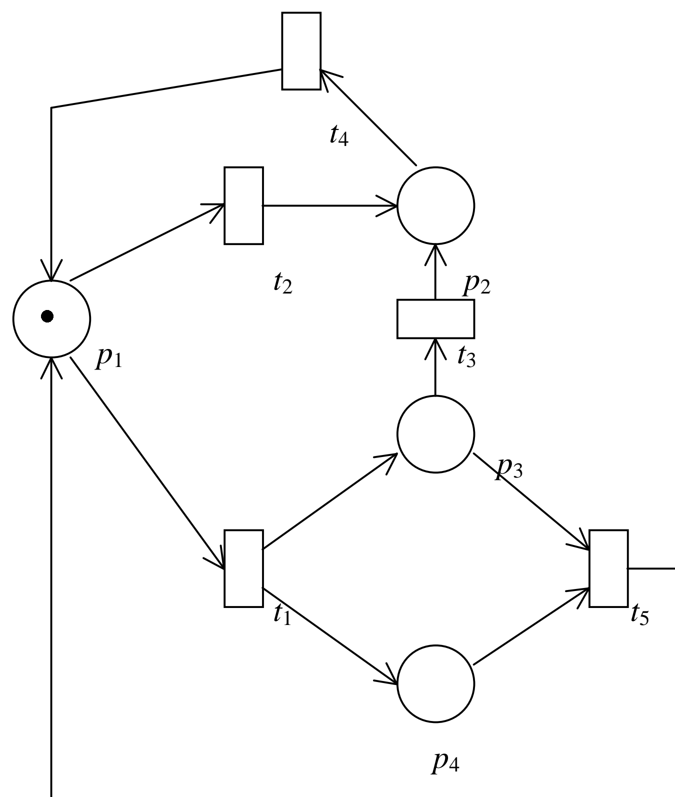


Figure 7.5: A complex, unbounded Place-Transition net [BK95].

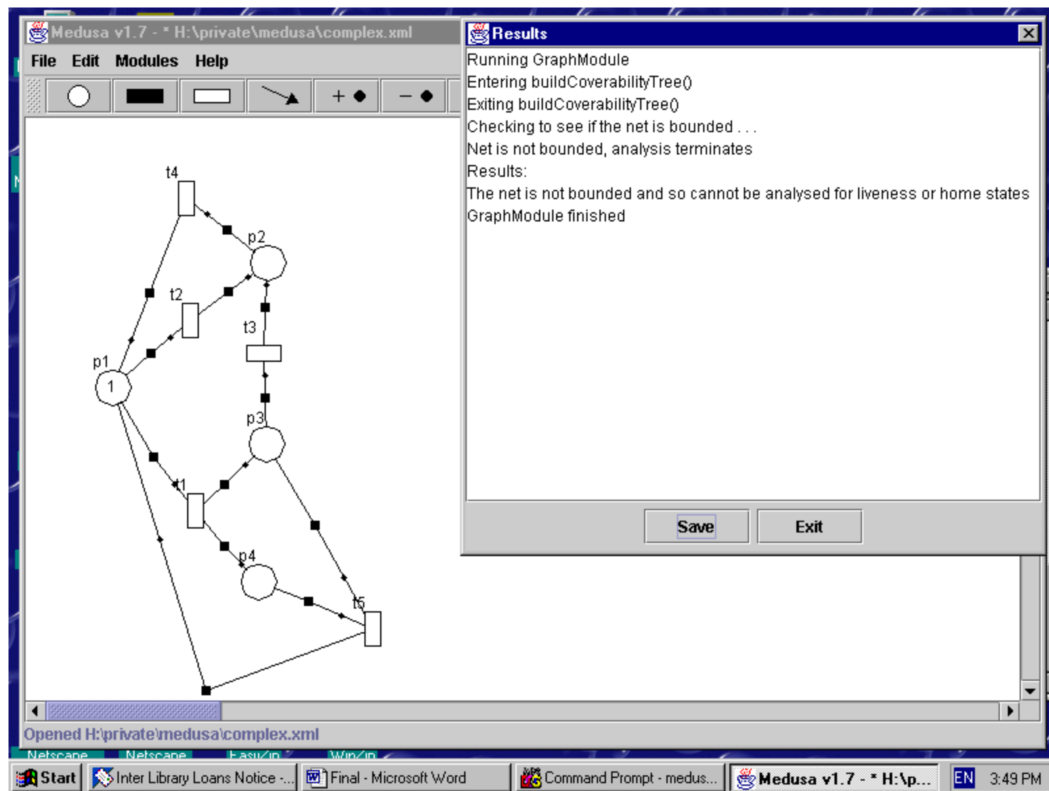


Figure 7.6: Screenshot of Medusa analysing the net in Figure 7.5 with the graph theory module.

7.3 DnamacaModule

It was decided to use the GSPN model of the Courier communications protocol shown in Figure 7.1 to validate DnamacaModule as it is a complex model which contains a bi-directed arc. DnamacaModule was designed specifically to describe these correctly as DaNAMiCS cannot. There are also a set of performance results readily available to which those generated by DnamacaModule can be compared. If the results are the same it can be concluded that DnamacaModule generates valid model descriptions and also that Medusa is capable of editing complex nets successfully.

The Courier model was entered into Medusa from the diagram reproduced in Figure 7.1. DnamacaModule was then loaded and run with the necessary performance measures entered through its GUI.

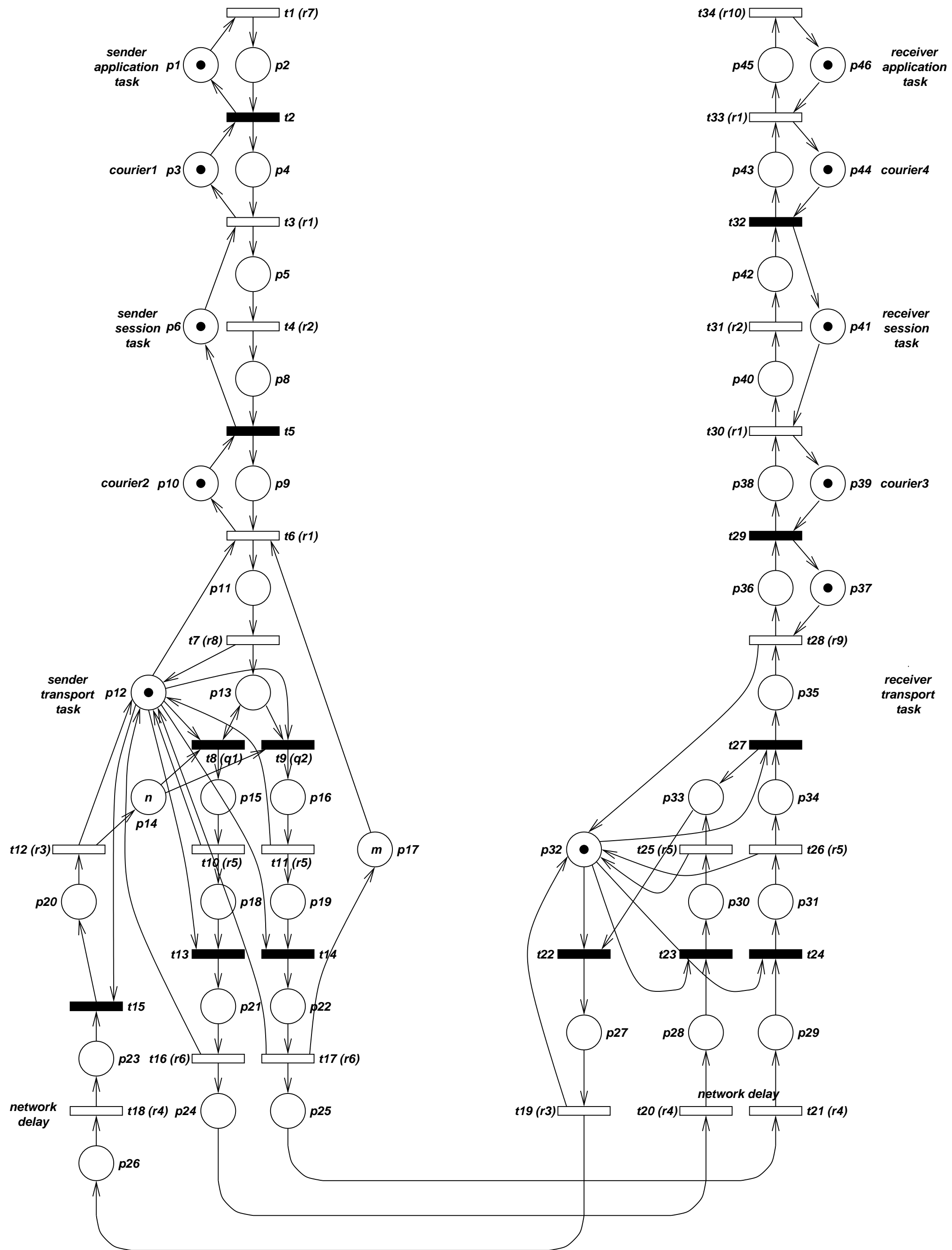


Figure 7.1 Diagram of the GSPN model of the Courier protocol [KNO99].

The performance measures used are identical to those in [KNO99]. The data throughput rate (λ) of the protocol is given by the throughput of t_{21}. The P_{xxxx} measures determine task utilisation, for example P_{transp1} is the probability that p_{12}

is marked. Similarly, P_{transp1} is defined for p_{32}, P_{sess1} and P_{sess2} for p_6 and p_{41}, and P_{send} and P_{recv} for p_1 and p_{46} [KNO96].

The results generated by DnamacaModule are reproduced in Table 7.1 along with the published results from [KNO99]. The variable m (the marking on p_{17}) was fixed at 1 to conform with that used to generate the known results, whilst n (the marking on p_{14}) was tried at 1 and 2. Unfortunately, although the published results cover the range $1 \leq n \leq 6$, the complexity of the model prevented the generation of results for $n > 2$ by Dnamaca.

As can be seen, the results generated from DnamacaModule are in complete agreement with the published results. From this it can be concluded that it is possible to edit complex nets using Medusa and that DnamacaModule generates correct model definitions from nets entered in Medusa even if that net contains bi-directed arcs.

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

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

Table 7.1: Published results (left) and those generated from the model description produced by DnamacaModule (right).[1]

7.4 Running a User-Designed Module

The aim of this project was to produce an editor which could be extended by the user without access to its source-code. In order truly to validate the success of the implementation, therefore, it is necessary to test if this has been achieved. A

[1] Results from C.M. Woodside and Y. Li., 'Performance Petri net analysis of communication protocol software by delay-equivalent aggregation' in *Proceedings of the 4th International Workshop on Petri nets and Performance Models* (Melbourne, Australia: IEEE Computer Society Press, 2nd-5th December 1999,) pp. 64-73, reproduced in [KNO99].

suitable module was produced as part of another MSc Computing Science project [MW01]. It had been agreed prior to the beginning of the projects that they should implement the same interface so that modules could be swapped between them in order to validate the concept of both. Other than this there was no discussion as to the internal designs of either the tools or the modules which both projects produced. This replicates the constraints under which modules would be produced in the real world – the user would not be intimately familiar with the internal works of Medusa in the way that its author is.

The module designed as part of [MW01] is capable of performing invariant analysis on a Petri net. Invariant analysis can be used for solving problems such as the Reader-Writer problem presented in [BK95]. Consider the Place-Transition net in Figure 7.6. This represents a system with several processes reading and writing a shared file. Readers never modify the file and so more than one may access it at the same time. Writers, however, do modify the file and so when they access it all other readers and writers must not be allowed to access the file. The synchronisation required could be achieved by a semaphore. A possible solution may be to initialise the semaphore with a value of n and every time a reader wanted access it would decrement it by 1 and then increment it by 1 when it finished. A writer, however, would decrement and increment the semaphore by n. Invariant analysis can be used to prove whether or not this solution is correct.

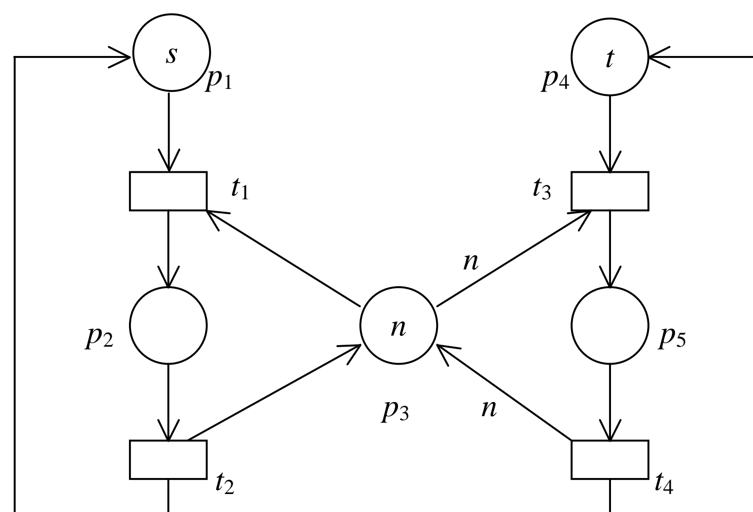


Figure 7.6: Petri net representation of the Reader-Writer problem [BK95].

In Figure 7.6, the number of tokens on p_1 and p_2 represent the number of readers which are not reading and reading respectively. Similarly, p_4 and p_5 model the same conditions for the writers whilst p_3 represents the semaphore. The results of the invariant analysis published in [BK95] give the following P-invariants for the markings on the various places. Recall that $M(p_i)$ is the marking on place p_i and that an invariant is something which holds true for all possible states of the system.

$$M(p_1) + M(p_2) = s$$
$$M(p_4) + M(p_5) = t$$
$$M(p_2) + M(p_3) + nM(p_5) = n, \forall M \in R(PN)$$

This means that the sum of the markings on p_1 and p_2 is always equal to s no matter which state the system is in. This means that the total number of readers is constant. Similarly, the total number of writers is always t [BK95]. The third equation can be solved to yield the following:

a) $M(p_2) \geq 1 \Rightarrow M(p_5) = 0$

b) $M(p_5) \geq 1 \Rightarrow M(p_2) = 0$

c) $M(p_5) \leq 1$ [BK95]

These show that if a reader is reading, no writer is writing and if a writer is writing no reader is reading (a) and b) respectively) [BK95]. It also demonstrates that there is at most one writer writing (c)) [BK95]. From this it can be concluded that the solution adopted is correct.

The screenshot in Figure 7.7 shows the results of performing this invariant analysis on the Reader-Writer problem where s = 3, t = 2 and n =2. The equations which are produced (displayed in the lower left-hand box) are as follows:

$$M(p_1) + M(p_2) = 3$$
$$M(p_2) + M(p_3) + 2M(p_5) = 2$$
$$M(p_4) + M(p_5) = 2$$

These are in exact accordance with the published results for the net. Furthermore, the module correctly identifies the net as being bound. In order to verify that this one result is not an anomaly, the procedure of running the module with Medusa was repeated a number of times and the values of s, t and n were varied. For $s = 1$, $t = 1$ and $n = 1$:

$$M(p_1) + M(p_2) = 1$$
$$M(p_2) + M(p_3) + 1M(p_5) = 1$$
$$M(p_4) + M(p_5) = 1$$

Likewise for $s = 2$, $t = 2$ and $n = 2$:

$$M(p_1) + M(p_2) = 2$$
$$M(p_2) + M(p_3) + 2M(p_5) = 2$$
$$M(p_4) + M(p_5) = 2$$

And finally for $s = 3$, $t = 3$ and n 3:

$$M(p_1) + M(p_2) = 3$$
$$M(p_2) + M(p_3) + 3M(p_5) = 3$$
$$M(p_4) + M(p_5) = 3$$

In all cases these correspond to the published values for this problem.

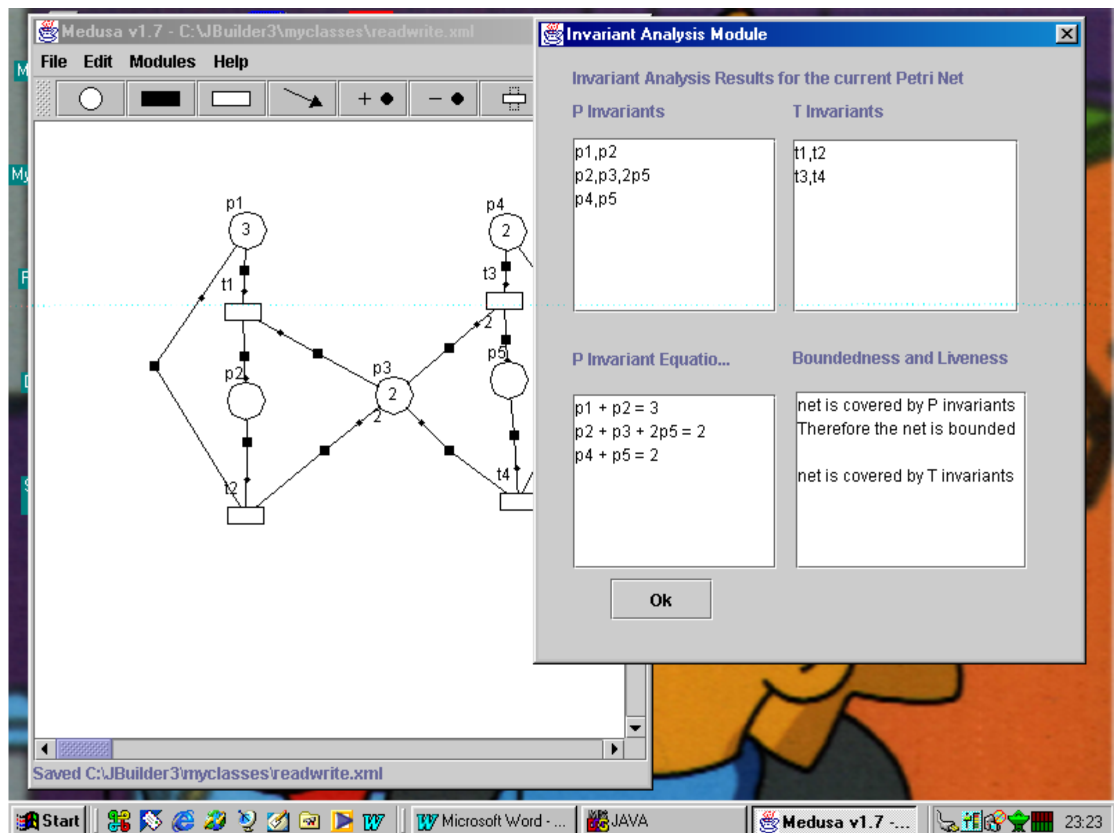


Figure 7.7: A screenshot showing Medusa running the Invariant Analysis Module on the Reader-Writer problem.

7.5 Conclusion

The results generated by the combination of Medusa and the three modules used all comply with known results. From this a number of things can be concluded. Firstly, this shows that the specific implementations of each module are correct. Secondly, Medusa is capable of being used to edit complex GSPNs like the Courier model successfully and the representation which it produces is sound. If this was not the case then errors would be expected in the results produced. As the modules have successfully generated results when used with other editors then, if errors had occurred, Medusa would have been at fault.

More broadly, however, the success of operating all three modules with Medusa shows that this project has succeeded in its aim of producing an extensible Petri

net tool. Particularly gratifying is the success of the invariant analysis tool as this was designed with knowledge of only PNML and the `Module` interface. That it functioned correctly when run by Medusa shows that it is possible to design modules for the tool when only the way in which it interfaces with these modules is known. This was what the project set out to achieve.

Chapter 8: Conclusion

8.1 Conclusions

This report has covered the production of an extensible Petri net editor/animator called Medusa. The aim of this project was to improve upon existing Petri net tools by creating a piece of software whose functionality could be extended through the addition of modules by users. It was hoped that this could be achieved even if the user was given only the specifications of the `Module` interface and details of the XML format used to describe saved nets. Furthermore, the design of a new Petri net tool created the opportunity to incorporate new features which are not present in other tools as well as to correct known mistakes in other implementations.

The architecture of Medusa has been described. The animator incorporates a novel feature in its ability to perform backwards animation, which allows the user to step backwards through the sequence of fired transitions. This feature is not present in the animators of existing tools like DaNAMiCS.

Medusa is capable of loading nets designed in other tools and also saving nets in a format which can be read by other tools. This is achieved through the use of the Petri Net Markup Language, an XML-based language which is being developed as a proposed standard for the description of Petri nets by software tools. The fact that it is a standard and not a proprietary format promotes the extensibility of Medusa by allowing it to be used in conjunction with other tools which support PNML.

The results from the three modules described in this report validate the design of Medusa and of the respective modules. The results obtained from DnamacaModule's analysis of the Courier protocol are particularly pleasing. Courier is a very complicated model with forty-five places and thirty-four transitions but Medusa proved capable of being used to create correctly such a model. DnamacaModule successfully converted this model into a representation

which could be analysed by Dnamaca even though the model contained a bi-directed arc. This shows that the flaw exhibited by DaNAMiCS when attempting to describe such arcs has been avoided in this implementation. The performance results for the Courier protocol obtained from an execution of DnamacaModule correspond exactly to the published figures. This demonstrates how Medusa can interface successfully with existing tools through specially designed modules.

The results from the graph theory analysis module show that it is possible to write a self-contained module capable of performing complex analysis on nets created with Medusa. Once again, the results for the problems analysed were the same as those which had been published. This validates the implementation of the mathematical theories used in this module. It also demonstrates that the implementation of Medusa is sound.

The final proof of the success of Medusa as an extensible piece of software came when a module designed by another party was successfully loaded and run. This showed that the main aim of this project has been achieved: namely that Medusa should be capable of running modules designed by users with no knowledge of Medusa's internal architecture. The only information shared between the two projects was details of the XML and Reflection interface.

8.2 Opportunities for Future Work

Thanks to Medusa's extensibility, the scope for future work is enormous as any number of modules could be written to perform various functions. There are, however, certain additions which could be made to the Medusa editor/animator in order to increase the functionality which it offers.

There are a number of other Petri net formalisms which could be added to Medusa, for example Coloured Petri Nets. It will be recalled that they offer no extra expressive power over the two types of net currently used in Medusa but they can reduce the complexity of large nets. Similarly, a version of Medusa which was capable of representing Queuing Petri Nets would be a valuable

extension. To achieve either of these, however, would require a major rewrite of the source-code of Medusa as it could not be achieved through a module.

Medusa could also be extended to support subnets. Like Coloured Petri Nets, subnets add no expressive power but they do simplify the graphical representation of the net. This is achieved by replacing sections of the net with "black boxes" which are themselves Petri nets but whose internal workings are not visible to the user. An implementation which allowed subnets to be imported from various files into a single Petri net would be particularly useful.

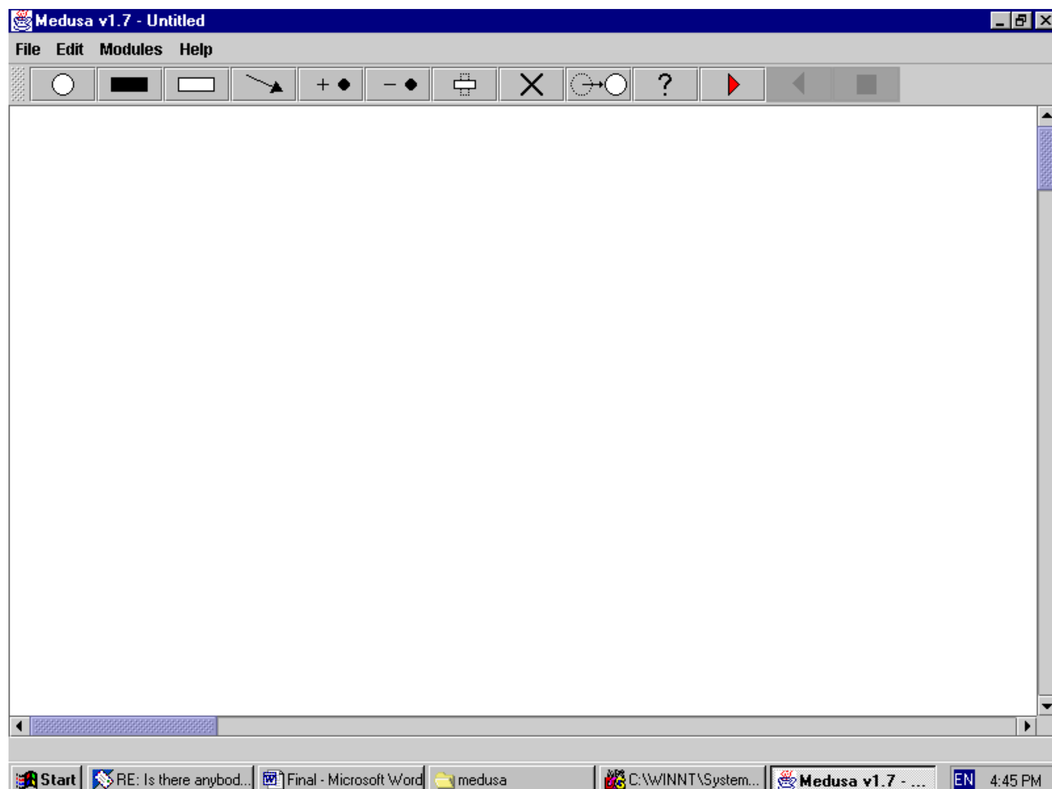
Medusa's animator could also be improved upon. The basic functionality of forwards and backwards animation is sound but some form of automatic animation could be added. This could allow the user to enter a sequence of transitions which they wished to see fired and then have the animator display that sequence. An even greater bonus would be if the animator could be connected to the extension mechanism in some way so that results from the execution of a module could be displayed. This would be particularly effective if the module was one which identified a sequence of transitions which lead to deadlock as the analyser could then display that trace in graphical form.

Appendix A: Medusa User Guide

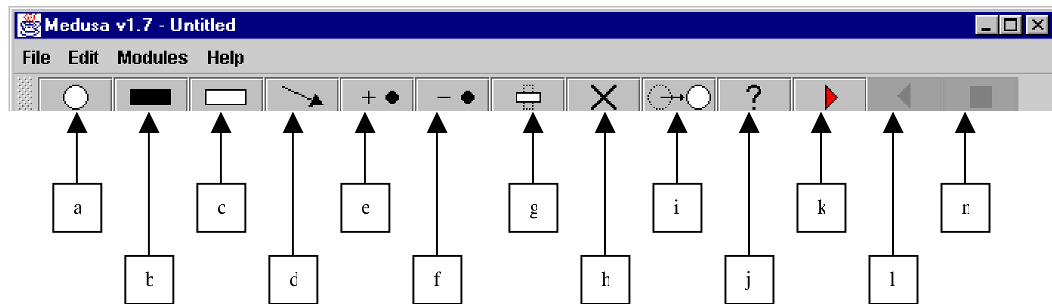
1. Unzip the `medusa.zip` archive. This will create a folder labelled `medusa` containing everything necessary to run Medusa.

2. Go into the `medusa` directory. To run Medusa in Windows, type `medusa.bat` in a command prompt or double-click on the `medusa.bat` file. To run Medusa in Linux, type `./medusa.ksh` in a console window.

3. Medusa will start, displaying a window like this:



4 What do the buttons do?

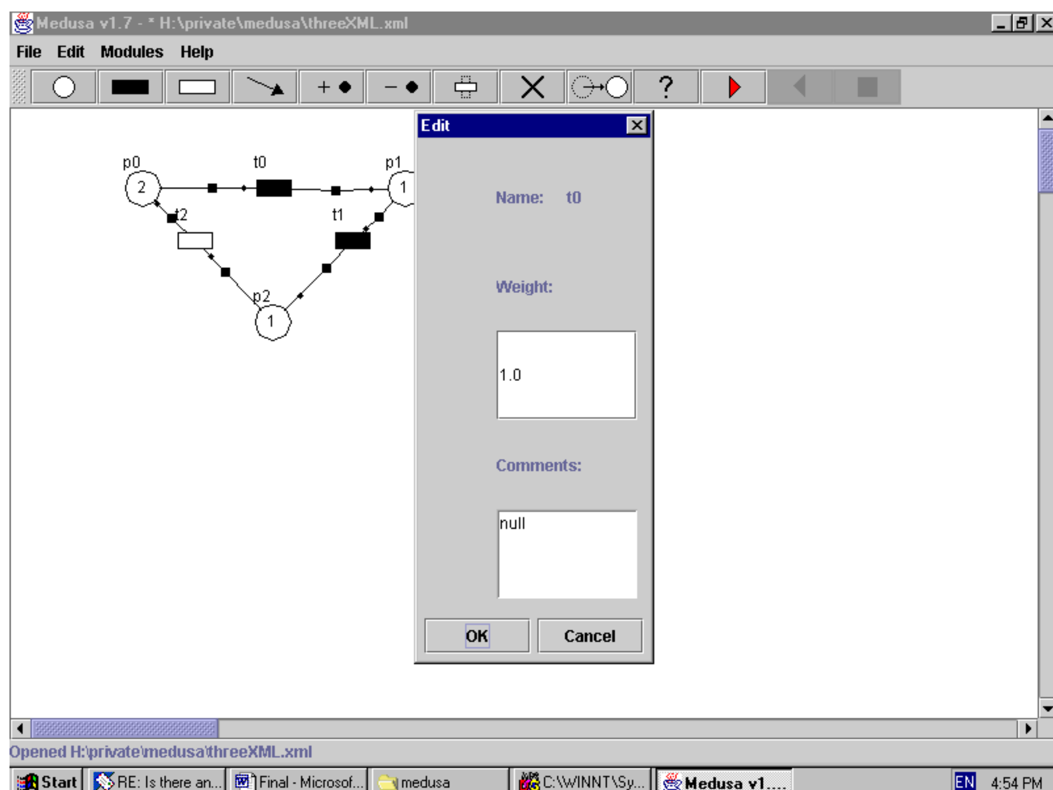


- a = Add a place
- b = Add an immediate transition
- c = Add a timed transition
- d = Add an arc
- e = Add a token to a place
- f = Remove a token from a place
- g = Rotate a transition
- h = Delete an element of the Petri net
- i = Move an element
- j = Edit an element's attributes
- k = Start animating
- l = Step back one firing
- n = Stop animating

5 To add a place, click on the appropriate button. Then click on the location where you wish to place it. Transitions can be added in the same way. To add an arc, click and hold on the node you wish it to start from and release over the node where you wish it to end. This is also the mechanism used to move an element.

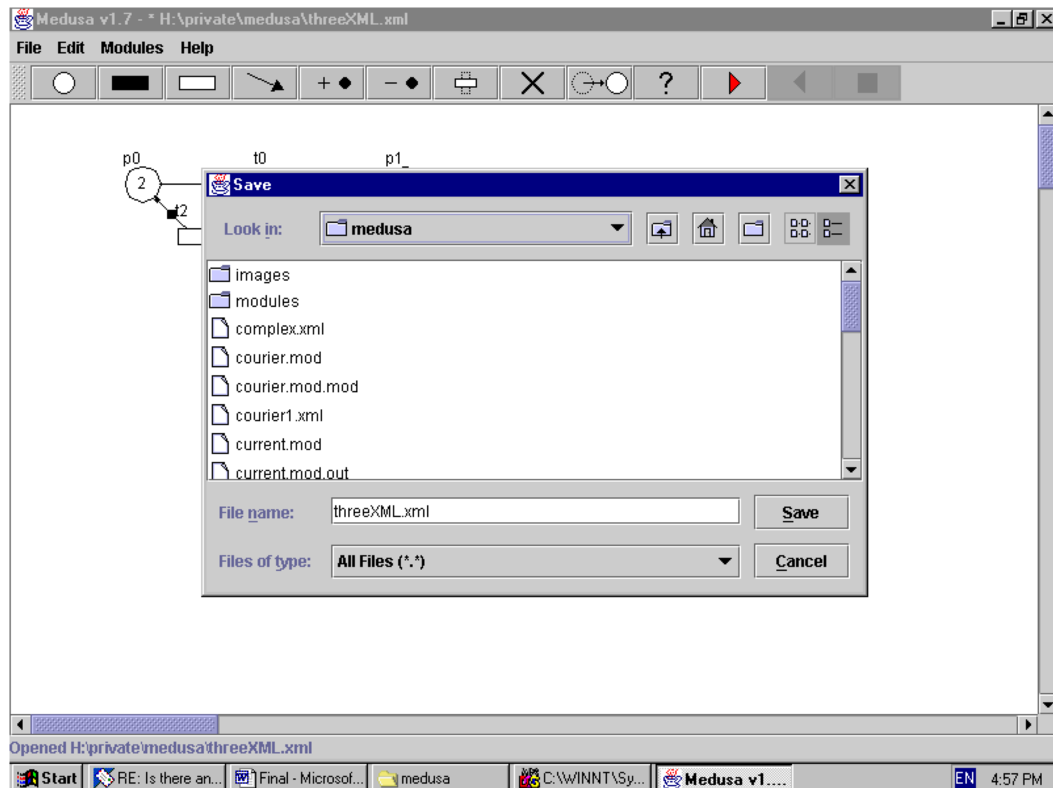
6 Tokens can be added and removed from a place by pressing the appropriate button and then clicking repeatedly on a place until the correct number appears.

7 To edit the attributes of a place, first click the "Edit an element's attributes" button. Then click over the element you wish to modify. A box will be displayed showing that element's attributes and this allows you to change them using the keyboard. An example is shown below.



8 To begin animating, click on the "Start animator" button. Transitions eligible to be fired will be highlighted in red. When one is clicked on, it will fire and the marking on the net will change accordingly. To undo a firing, click the "Step back one firing" button (black triangle). To stop animating and restore the initial marking, click the "Stop animator" button (black square).

9 To save the net, select “Save as XML” from the “File” menu. A dialog will be displayed allowing you to select the name and location of the saved file:



10 To open a saved file, select “Open XML File” from the “File” menu. Again, a dialog will open allowing you to select the file to load.

11 To show or remove the grid, select “Toggle Grid” from the “Edit” menu. To enable/disable the automatic snapping of elements to the grid, select “Toggle Snap to Grid” from the “Edit” menu.

12 To load a module, select “Load Module” from the “Module” menu. A dialog will open as when opening a saved net. Select the module from this. To then run it, select “Run” from the “Module” menu. The module will then execute and prompt for any further input.

13 To remove a previously loaded module select “Remove” from the “Module” menu.

Bibliography

[BG00] Sara Baase and Allen Van Gelder, *Computer Algorithms – Introduction to Design and Analysis*, 3rd Edition (Reading, Mass.: Addison-Wesley, 2000)

[BKK94] Falko Bause, Peter Kemper and Pieter Kritzinger, *Abstract Petri Net Notation* in 'Forschungsbericht Nr. 563 des Fachbereichs Informatik der Universität Dortmund' (University of Dortmund, Germany, 1994) from http://ls4-www.informatik.uni-dortmund.de/QPN/APNN_article/PNN.ps

[BK95] Falko Bause and Pieter S, Kritzinger, *Stochastic Petri Nets – An Introduction to the Theory* (Braunschweig/Wiesbaden, Germany: Advanced Studies in Computer Science, Friedr. Vieweg & Sohn Verlag, 1995)

[CWH01] Mary Campione, Kathy Walrath, Alison Huml, *The Java(TM) Tutorial: A Short Course on the Basics (The Java(TM) Series)* (Palo Alto, CA: Sun Microsystems Inc., 2001) available on-line at http://java.sun.com/docs/books/tutorial/

[DAC00] Michael C. Daconta, *When Runtime.exe() won't*, from http://www.javaworld.com/javaworld/jw-12-2000/jw1229traps.html

[DAN] DaNAMiCS Homepage: http://www.cs.uct.ac.za/Research/DNA/DaNAMiCS/

[ERH99] Elliotte Rusty Harold, *XML Bible* (Foster City, CA: IDG Books Worldwide Inc, 1999)

[JAXB] *The Java^tm Architecture for XML Binding (JAXB) User Guide Early Access Draft* (Palo Alto, CA: Sun Microsystems Inc., May 2001) from http://java.sun.com/xml/ jaxb/index.html

[JAXP] *The Java^tm APIs for XML Processing Version 1.1 Documentation* from http://java.sun.com/xml/xml_jaxp.html

[JDC] *The Java Developer Connection^tm Tech Tips*, June 27th 2000, from http://developer.java.sun.com/developer/TechTips/2000/tt0627.html

[JXML] *Java^tm Technology and XML*, from http://java.sun.com/xml

[JKW06/00] Mathias Jüngel, Ekkart Kindler and Michael Weber, *Towards a Generic Interchange Format for Petri Nets*, presented the 'Meeting on XML/SGML based Interchange Formats for Petri Nets' at the 21st International Conference on the Application and Theory of Petri Nets, Aarhus, Denmark June 26th-30th 2000. From http://www.daimi.au.dk/pn2000/Interchange/position.html.

[JKW08/00] Mathias Jüngel, Ekkart Kindler and Michael Weber, *The Petri Net Markup Language* (Humboldt-Universität zu Berlin, 31st August 2000) submitted to the Algorithmen und Werkzeuge für Petri-Netze (Algorithms and Tools for Petri Nets) Workshop, Koblenz, Germany, October 2000, from http://www.informatik.hu-berlin.de/top/pnml/ index.html

[KNO96] William J. Knottenbelt, *Generalised Markovian Analysis of Timed Transition Systems* (Unpublished MSc thesis, University of Cape Town, 1996)

[KNO99] William J. Knottenbelt, *Parallel Performance Analysis of Large Markov Models* (Unpublished PhD thesis, Imperial College, University of London, 1999)

[KW00] Olaf Kummer and Frank Wienberg, *The XML File Format of Renew* (University of Hamburg) presented at the 'Meeting on XML/SGML based Interchange Formats for Petri Nets' at the 21st International Conference on Application and Theory of Petri Nets, Aarhus, Denmark, June 26-30, 2000 from http://www.daimi.au.dk/pn2000 /Interchange/detailed.html

[LM00] Regnar Bang LyngsØ and Thomas Mailund, *Textual Interchange Format for High-Level Petri Nets* (University of Aarhus) presented at the 'Meeting on XML/SGML based Interchange Formats for Petri Nets' at the 21st International Conference on Application and Theory of Petri Nets, Aarhus, Denmark, June 26-30, 2000 from http://www.daimi.au.dk/pn2000/Interchange/detailed.html

[MW01] Mark Wass, *Predator - A Hierarchical Petri Net Editor* (Unpublished MSc thesis, Imperial College, University of London, 2001)

[NSS94] Esko Nuutila and Eljas Soisalon-Sohininen, *On Finding the Strongly Connected Components in a Directed Graph* (Helsinki: Laboratory of Information Processing Science, Helsinki University of Technology, 1994) from http://citeseer.nj.nec.com/cache/papers2/cs/549/http:zSzzSzwww.cs.hut.fizSz~enuzSzpszSzipl-scc.pdf/ nuutila94finding.pdf

[PNK] The Petri Net Kernel Homepage: http://www.informatik.hu-berlin.de/top/pnk/ index.html

[PNML] The Petri Net Markup Language Homepage: http://www.informatik.hu-berlin.de/top/ pnml/index.html

[REN] The Renew Homepage: http://www.renew.de/