

IMPERIAL COLLEGE LONDON  
DEPARTMENT OF COMPUTING

# **Extension of PIPE2 to Support Coloured Generalised Stochastic Petri Nets**

Written by Alex Charalambous

Supervised by  
Dr William Knottenbelt

**Final Year Report**

June 14, 2010

## Abstract

In this report we demonstrate how we have extended the Platform Independent Petri net Editor 2 (PIPE), an open-source system modelling tool for Generalised Stochastic Petri Nets (GSPNs), to support the design, animation and analysis of Coloured Generalised Stochastic Petri Nets (CGSPNs). We review existing standard ways to represent CGSPNs and design/implement a new approach that offers a more intuitive form of graphical representation. To accompany this, we also propose a new PNML structure for importing/exporting CGSPNs between applications.

The new infrastructure of PIPE allows users to implement and plug in analysis modules to directly analyse CGSPNs. Also, in order for CGSPNs to be analysed using PIPE's powerful pre-existing analysis modules, we have devised an algorithm that is capable of transforming any CGSPN to a GSPN. This unfolding mechanism also enables users to export their CGSPNs to tools that do not support CGSPNs so that they can also be used for analysis.

Through a case study of a hospital's Accident and Emergency department, designed by Susanna Wau Men Au-Yeung [2], we show how these changes allow users to create more realistic models while retaining a compact graphical representation. Finally, by implementing a test framework for PIPE, we protect the existing and new functionality from being broken by changes made to the open source project in the future.

## **Acknowledgments**

I would like to thank my supervisor, Dr William Knottenbelt for his constant support and motivation throughout the project development. I would also like to thank my second supervisor, Nicholas Dingle for devoting time to provide me with valuable advice and constructive criticism. I would like to acknowledge PhD student Nikolas Anastasiou for providing me with feedback throughout the design stages of the project. Finally, I would like to thank my parents for their continuous financial and moral support throughout my four years as a student in London.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Contributions . . . . .	2
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	Introduction . . . . .	3
2.2	Petri Nets . . . . .	3
2.3	Petri net Tools . . . . .	11
2.4	PIPE2 . . . . .	14
<b>3</b>	<b>Overview of Architectural Changes</b>	<b>21</b>
3.1	Introduction . . . . .	21
3.2	Packages . . . . .	22
<b>4</b>	<b>Supporting the design and editing of CGSPNs</b>	<b>26</b>
4.1	Introduction . . . . .	26
4.2	Add/Edit/Remove token classes . . . . .	26
4.3	Marking a place with different token classes . . . . .	31
4.4	Add/Edit Multiple Arc Weights On A Single Arc . . . . .	38
4.5	Extending the PNML interchange format to allow saving/loading of CGSPNs . . . . .	41
<b>5</b>	<b>Supporting the animation of CGSPNs</b>	<b>46</b>
5.1	Introduction . . . . .	46
5.2	Existing Architecture . . . . .	47
5.3	Design decisions and challenges . . . . .	48
5.4	Overview of new functionality . . . . .	48
5.5	New Architecture . . . . .	48
5.6	Final result . . . . .	49
<b>6</b>	<b>A Novel Graphical Representation for CGSPNs</b>	<b>50</b>
6.1	Motivation . . . . .	50
6.2	Design considerations and challenges . . . . .	52
6.3	Implementation and Architecture . . . . .	54
6.4	Final Result . . . . .	57
<b>7</b>	<b>Supporting the unfolding of CGSPNs</b>	<b>62</b>
7.1	Motivation . . . . .	62
7.2	Design considerations and challenges . . . . .	63

7.3	Implementation and architecture . . . . .	63
7.4	Final Result . . . . .	65
<b>8</b>	<b>Supporting the analysis of CGSPNs</b>	<b>66</b>
8.1	Introduction . . . . .	66
8.2	Existing Functionality . . . . .	66
8.3	Existing Architecture . . . . .	66
8.4	Design decisions and challenges . . . . .	67
8.5	New Architecture . . . . .	68
8.6	Final Result . . . . .	68
<b>9</b>	<b>Testing Framework</b>	<b>69</b>
9.1	Motivation . . . . .	69
9.2	Design considerations and challenges . . . . .	69
9.3	Implementation . . . . .	70
9.4	Final Result . . . . .	75
<b>10</b>	<b>Evaluation</b>	<b>76</b>
10.1	Vastly increased the modelling power of PIPE through the design, anima- tion and analysis of CGSPNs . . . . .	76
10.2	Quality of Structural Changes . . . . .	80
10.3	Comparison with state of the art . . . . .	81
10.4	Added ability to transform any CGSPN to a GSPN to allow analysis by non-CGSPN compatible modules/tools . . . . .	87
10.5	Retained PIPE's ease of use and core functionality . . . . .	89
10.6	Added ability to save/load CGSPNs . . . . .	90
10.7	Present tool to Petri-net experts to get feedback on the features and its ease of use . . . . .	90
<b>11</b>	<b>Conclusion</b>	<b>92</b>
11.1	Future Work . . . . .	93
	<b>Bibliography</b>	<b>93</b>

# Chapter 1

## Introduction

### 1.1 Motivation

Petri nets are a formalism for the description and analysis of concurrent processes. They provide an intuitive graphical representation of the dynamic behaviour of systems which is backed by a well defined mathematical theory. The graphical component allows for visualization of the state changes in the model, while the theoretic component allows accurate modelling and analysis of system behaviour [11]. This allows Petri nets to be easily understood and yet be automatically analysed by a computer. Such properties have made Petri-Nets an extremely powerful and popular modelling paradigm [13] that is commonly applied in various fields e.g. modelling of manufacturing systems [26] and airport control systems [24].

A Coloured Generalised Stochastic Petri Net (CGSPN) is an extension of the ordinary Petri net model that vastly improves its modelling power by facilitating the clean and compact realisation of more complex real life systems. The existing method for representing CGSPNs involves drastically changing the graphical representation of a Petri net to cater for this improvement. In particular, the governing rules of the dynamic behaviour of a Petri net have been replaced by more complex ones involving mathematical expressions. The graphical representation is no longer as intuitive and often requires explanation.

There are many Petri net tools that allow users to graphically design Petri net models, simulate the models through an animated sequence and analyse their models in a number of different ways. However, although certain tools allow users to perform certain tasks, users will rarely find a single tool that can cater to all their needs. Instead, many find themselves having to import/export their Petri nets between tools that are suited for a particular task and when that is not possible they must re-create the Petri nets manually. Furthermore, the few CGSPN-compatible tools that exist are lacking in other vital features and have adopted the more complex expression-based graphical representation that makes the design of GSPNs more complicated than strictly necessary. The Petri net community seems to be lacking a complete Petri net tool which has all the features users usually require, integrated into one application. Section 2.3 analyses all available tools and concludes that a tool initially developed by Imperial College London, “Platform Independent Petri net Editor 2 (PIPE2)”, is the perfect candidate to be extended for this purpose.

## 1.2 Contributions

One of PIPE’s most prominent deficiencies is its lack of support for CGSPNs, preventing it from being able to compactly represent large classes of real world systems. Furthermore, since the introduction of CGSPNs, the Petri net community has been limited to only one possible representation for such models. In this project we aim to make the following contributions:

- Extend PIPE2 to support the design, animation and analysis of CGSPNs (Chapters 4 and 5). Hence we greatly enhance the modelling power of a tool that already has one of the most comprehensive feature sets in the Petri net community (Section 2.3). The current procedure of creating and editing non-coloured Petri nets in PIPE2 are left completely unchanged to avoid unnecessary complications for users wishing to model simpler GSPNs.
- Introduce a novel way of representing CGSPNs (Chapter 6) that is as simple and intuitive as that of ordinary Petri nets. Due to their similarities, this representation guarantees that any user familiar with GSPN models will be able to understand, design, simulate and analyse their own CGSPN models with little or no training.
- Propose a new standard interchange format for importing/exporting our representation of CGSPNs between other tools (Section 4.5).
- Implement an “unfolding mechanism” (Chapter 7), which allows users to transform any CGSPN implemented in PIPE2 into a GSPN. This automatically enables the net to be analysed using existing analysis modules. In addition, it allows for the net to be exported to the current standard interchange format (Section 2.4.4) and analysed by external tools that do not support CGSPNs.
- Implement a testing framework (Chapter 9) for PIPE2 to ensure existing functionality has not broken but also to ensure that in the future, developers do not break any parts of the functionality.

# Chapter 2

## Background

### 2.1 Introduction

The purpose of this section is to introduce basic concepts relating to this project and presenting the most relevant work done in the field of Petri nets. In particular, we begin by introducing various types of Petri nets, explaining what they are, what they do and how they behave (Section 2.2). As the goal of this project is to extend a Petri net tool (PIPE2), we first explain why we have selected to extend that particular tool (Section 2.3) before investigating the tool in more detail (Section 2.4). This section is aimed to give all the information required to understand the specification listed in this document and to begin its implementation.

### 2.2 Petri Nets

Petri nets are a formalism for the description and analysis of concurrent processes which arise in distributed systems [19]. Petri nets provide an intuitive graphical representation (directed bipartite graph) of the dynamic behaviour of systems which is backed by a well defined mathematical theory. The graphical component allows for visualization of the state changes in the model, while the theoretic component allows accurate modelling and analysis of system behaviour [11]. This allows Petri nets to be easily understood and yet still be automatically simulated by a computer. Such properties have made Petri-Nets an extremely powerful and popular modelling paradigm [13].

#### 2.2.1 Place Transition Nets

The simplest and most basic form of Petri Nets are known as “Place Transition Nets” or “Ordinary Petri Nets” and consist of the following components [3]:

- **Place:** Models a condition or object and is depicted by a circle. Places may contain tokens.
- **Token:** Models the value of the place it exists in and is depicted by a black dot.



- **Transition:** Models activity that can occur that changes the value of places and is depicted by a rectangle.
- **Arc:** Connects transitions to places and indicates which places may be altered by which transitions. It is depicted by straight lines between places and transitions.

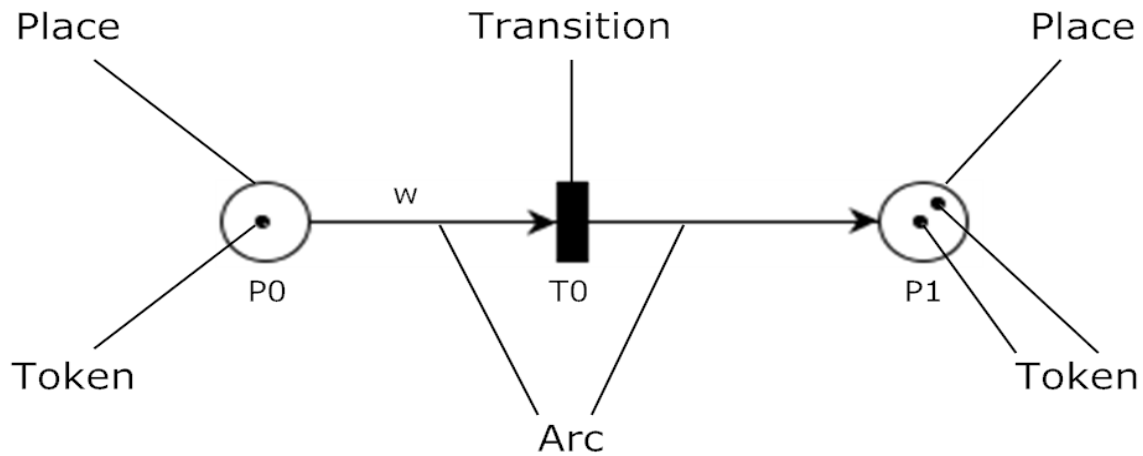


Figure 2.1: A basic Place Transition Net

In Figure 2.1, P0 is known as the **input place** of T0 and P1 as the **output place** of T0. Similarly, T0 is known as the **output transition** of P0 and the **input transition** of P1.

The number sometimes written next to an arc is known as the **arc weight**. If no number exists, then an arc weight of 1 is inferred. For example the arc between P0 and T0 has a weight of  $w$ , whereas the arc between T0 and P1 has a weight of 1.

### Firing of transitions

During the execution of a Petri net, events may occur that change the distribution of tokens among places, thus changing the state of the model. This is known as firing a transition and can only be done if the transition is enabled:

- A transition is enabled if its input place is marked with the same number of tokens (or more) as the weight of the arc that connects them.
- An enabled transition may fire. This event will:
  - Destroy the number of tokens depicted by each input arc-weight from their corresponding input places.
  - Create the number of tokens depicted by each output arc-weight from their corresponding output places.

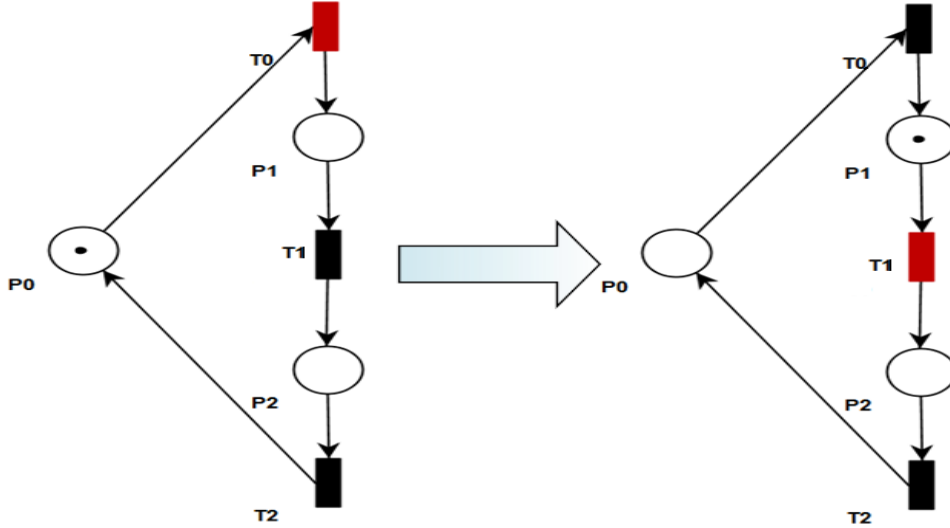


Figure 2.2: A transition being fired (enabled transitions are coloured in red)

On the left hand side of Figure 2.2, only transition T0 is enabled. Since there is no arc weight depicted on neither of the arcs in the place transition net, an arc weight of 1 is assumed. As place P0 is the only place marked with a token, T0 is the only enabled transition. The right hand side of Figure 2.2 shows the state of the place transition net after T0 has fired. A token has been destroyed from P0 (the input place of T0) and created in P1 (the output place of T0). Now T0 is no longer active as its input place is no longer marked with a token but T1 is, as its input place (P1) is now marked with a token.

A place transition net can be formally defined as follows [3]:

**Definition 1.** A Place Transition net (*P-T net*) is a 5-tuple  $PN = (P, T, I^-, I^+, M_0)$ :

- $P = \{p_1, \dots, p_n\}$  is a finite and non empty set of places,
- $T = \{t_1, \dots, t_m\}$  is a finite and non-empty set of transitions,
- $P \cap T = \emptyset$ ,
- $I^-, I^+ : P \times T \rightarrow \mathbb{N}_0$  are the backward and forward incidence functions.
- $M_0 : P \rightarrow \mathbb{N}_0$  is the initial marking.

The marking of a Petri net represents the assignment of tokens to places [11].

For further clarification we shall also formally define the input and output elements of a place and a transition as follows[1]:

**Definition 2.** Let  $PN = (P, T, I^-, I^+, M_0)$  be a Place-Transition net where:

- Input places of transition  $t$ :  $\bullet t := \{p \in P \mid I^-(p, t) > 0\}$ ,
- Output places of transition  $t$ :  $t \bullet := \{p \in P \mid I^+(p, t) > 0\}$ ,
- Input transitions of place  $p$ :  $\bullet p := \{t \in T \mid I^+(p, t) > 0\}$ ,
- Output transitions of place  $p$ :  $p \bullet := \{t \in T \mid I^-(p, t) > 0\}$ , the usual extension to sets  $X \subseteq P \cup T$  is defined as  $\bullet X = \cup_{x \in X} \bullet x$ ,  $X \bullet = \cup_{x \in X} x \bullet$ .

## 2.2.2 Stochastic Petri Nets

Stochastic Petri Nets (SPNs) are a variation of the original place transition net model that allow quantitative analysis of models that exhibit stochastic and discrete behaviour. This is achieved by adding the notion of time into the original Petri net model.

A Stochastic Petri net can be formally defined as follows [3]:

**Definition 3.** A Stochastic Petri net (SPN) is a 2-tuple  $SPN = (PN, \Lambda)$  where:

- $PN = (P, T, I^-, I^+, M_0)$  is the underlying Place-Transition net
- $\Lambda = \{\lambda_1, \dots, \lambda_m\}$  is a finite and non-empty set of transition rates where  $\lambda_i$  is the, (possibly marking dependent) transition rate of transition  $t_i$ .

The firing time of transition  $t_i$  follows an exponential distribution where the cumulative density function is:

$$F_{x_i}(x) = 1 - e^{-\lambda_i x}$$

SPNs are isomorphic to continuous time Markov chains; the number of states in the Markov chain corresponds to the reachability graph of the SPN [23].

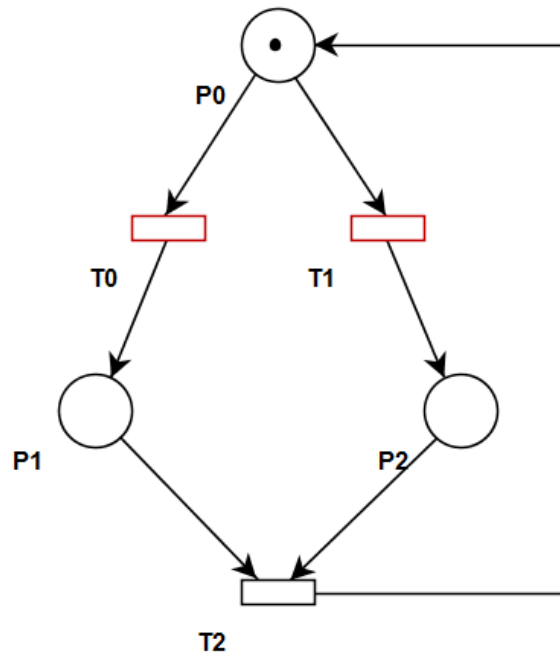


Figure 2.3: A simple SPN

In Figure 2.3 both T0 and T1 are enabled. Firing of T0 will disable T1, preventing it from firing and vice versa. With an initial marking of  $M_0 = (1,0,0)$ , if T0 fires the SPN will change to a marking of:  $M_1 = (0,1,0)$  whereas if T1 fires the SPN will change to a marking of  $M_2 = (0,0,1)$ . The transition that will fire and hence the marking that the SPN will end up in depends on the transition rates of T0 and T1 ( $\lambda_0$  and  $\lambda_1$  respectively). In this example the expected time for T0 and T1 to fire is  $\frac{1}{\lambda_0}$  and  $\frac{1}{\lambda_1}$  respectively. To help explain the usefulness of Petri nets and their applications in real life scenarios, Figure 2.4 shows a model of a hospital's Accident and Emergency department. The tokens represent

initially healthy individuals who fall ill and are then serviced by the hospital. This is a simplified model of the research done in the paper [2] and is a minor adaptation from one of the examples packaged with the Petri-net tool PIPE2 (see Section 2.4).

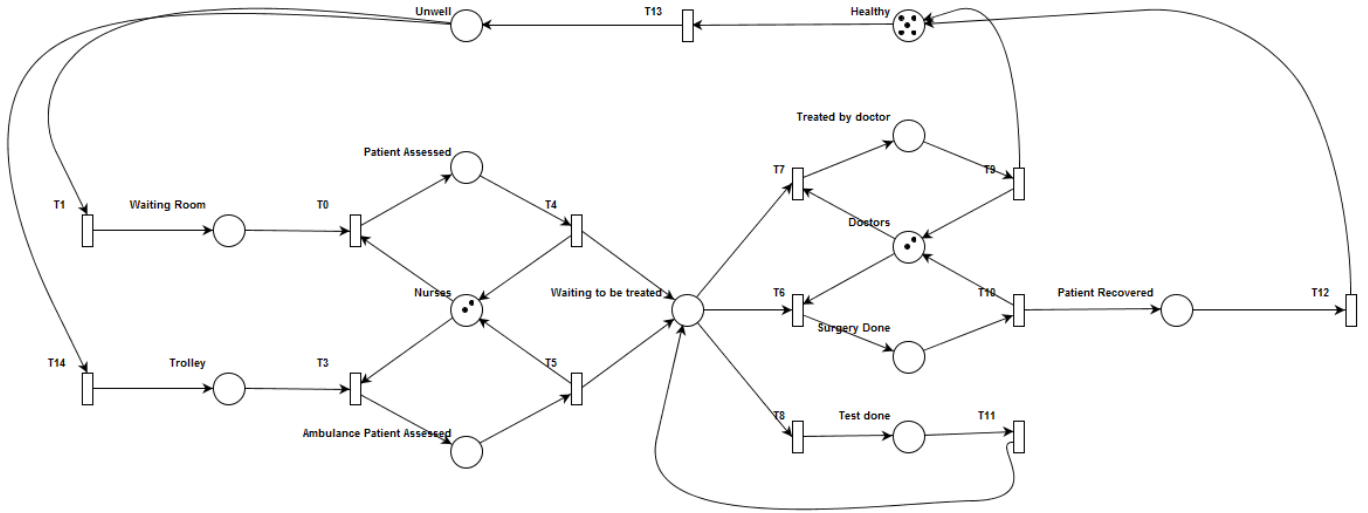


Figure 2.4: Modelling a hospital using a SPN

### 2.2.3 Generalised Stochastic Petri Nets

GSPNs [17] are an extension to the SPN model that consist of two different classes of transitions [3]:

- Immediate transitions: Once enabled, fire immediately and do not take up any time. Depicted by a solid rectangular box.
- Timed transitions: Once enabled, fire after a random, exponentially distributed time (as in SPNs). Depicted by an empty rectangular box.

Immediate transitions have priority over timed transitions.

A Generalized Stochastic Petri net (GSPN) can be formally defined as follows [3]:

**Definition 4.** A Generalized Stochastic Petri net 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,  $T_1 \neq \emptyset$
- $T_2 \subset T$  denotes the set of immediate transitions,  $T_1 \cap T_2 = \emptyset$ ,  $T = T_1 \cup T_2$
- $W = (w_1, \dots, w_{|T|})$  is an array whose entry  $w_i \in \mathbb{R}^+$  is either:
  - The (possibly marking dependent) rate of a negative exponential distribution which specifies the firing delay: Occurs when transition  $t_i$  is a timed transition.
  - A (possibly marking dependent) weight which specifies the firing frequency: Occurs when transition  $t_i$  is an immediate transition.

In the case where two immediate transitions are enabled, the transition with the highest frequency has priority. Likewise, in the case where two timed transitions are enabled, the transition with the largest transition rate has priority.

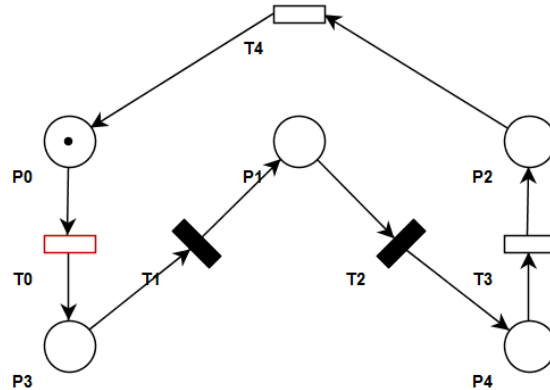


Figure 2.5: A simple GSPN

In Figure 2.5, transitions T0, T3 and T4 are timed transitions and each would have an associated firing delay. Transitions T1 and T2 are immediate transitions and each would have an associated firing frequency.

#### 2.2.4 Coloured Petri Nets

One disadvantage to the Petri net models we have presented so far, is that there is a lack of expressiveness due to the fact that there is only one type of token available. This is apparent in the model of a hospital's Accident and Emergency department (Figure 2.4) where a Place Transition net is only able to model one general type of patient. It would be much more powerful and useful if within the model you could distinguish between different classes of patients such as critically injured patients (who would be serviced at a higher priority) and patients with minor injuries. Such expressiveness would allow for more powerful and accurate models of real world scenarios.

Such desired features were brought to life when Kurt Jensen [14] defined a new variation to the standard Petri Net model called Coloured Petri nets (CPNs). In this model, a new type, called the colour is attached to a token.

A Coloured Petri net (CPN) can be formally defined as follows [3]:

**Definition 5.** A Coloured Petri net is a 6-tuple:  $CPN = (P, T, C, I^-, I^+, M_0)$  where:

- $P$  is a finite and non empty set of places,
- $T$  is a finite and non-empty set of transitions,
- $P \cap T = \emptyset$ ,
- $C$  is a colour function defined from  $P \cup T$  into finite and non-empty sets,
- $I^-, I^+$  are the backward and forward incidence functions defined on  $P \times T$  such that:  $I^-(p,t), I^+(p,t) \in [C(t) \rightarrow C(p)_{MS}]$ ,  $\forall (p,t) \in P \times T$ ,

- $M_0$  is a function defined on  $P$  describing the initial marking such that  $M_0(p) \in C(p)_{MS}, \forall p \in P$

NOTE: A multi-set (also known as a bag) is identical to a set with the exception that individual elements of the set may occur more than once. In the definition above,  $C(p)_{MS}$  represents the set of all finite multi-sets over  $C$ .

For further clarification we shall also formally define the preset and postset of a transition as follows[1]:

**Definition 6.** The preset  $\bullet(p, c)$  of  $p \in P$  and  $c \in C(p)$  is:

$$\bullet(p, c) := \{(t, c') | t \in T, c' \in C(t) : I^+(p, t)(c')(c) \neq 0\}.$$

The preset  $\bullet(t, c')$  of  $t \in T$  and  $c' \in C(t)$  is:

$$\bullet(t, c') := \{(p, c) | p \in P, c \in C(p) : I^-(p, t)(c')(c) \neq 0\}.$$

The postset  $(p, c)\bullet$  of  $p \in P$  and  $c \in C(p)$  is:

$$(p, c)\bullet := \{(t, c') | t \in T, c' \in C(t) : I^-(p, t)(c')(c) \neq 0\}.$$

The postset  $(t, c')\bullet$  of  $t \in T$  and  $c' \in C(t)$  is:

$$(t, c')\bullet := \{(p, c) | p \in P, c \in C(p) : I^+(p, t)(c')(c) \neq 0\}.$$

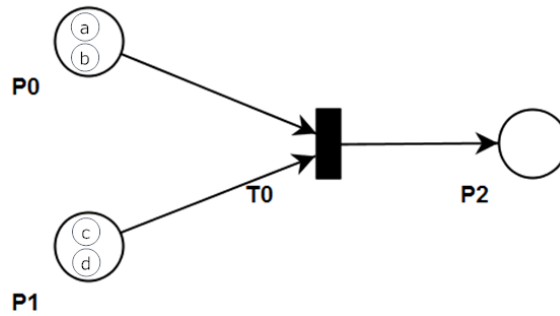


Figure 2.6: A simple CPN

In Figure 2.6 place P0 is marked with two tokens of colours a and b and P1 is marked with two tokens of colours c and d. With the introduction of different types of tokens, the enabling and firing of transitions becomes more complex, as the next state depends on which colours were destroyed and created in each place. The behaviour of a CPN can be defined as follows:

- **Enabling of transitions in CPNs**

A transition  $t \in T$  is enabled in a marking  $M$  with respect to a colour  $c' \in C(t)$ , denoted by  $M[(t, c') >$ , iff  $M(p)(c) \geq I^-(p, t)(c')(c), \forall p \in P, c \in C(p)$  [3].

- **Firing of transitions in CPNs** An enabled transition  $t \in T$  may fire in a marking  $M$  with respect to a colour  $c' \in C(t)$  yielding a new marking  $M'$ , denoted by  $M \rightarrow M'$  or  $M[(t, c') \downarrow M'$ , with  $M'(p)(c) = M(p)(c) + I^+(p, t)(c')(c) - I^-(p, t)(c')(c), \forall p \in P, c \in C(p)$  [3].

In Figure 2.6, the firing of T0 could destroy and create different combinations of tokens, for example: destruction of a and c and the creation of a new colour f or the destruction of b, c and d and the creation of a token of colour b and so on. Each combination is known as a **firing mode** and is defined by the backward and forward incidence matrices. Note that in Figure 2.6 we have used the most common convention of classifying various types of coloured tokens (using letters). For the remainder of the project however, we will choose to classify the various types of tokens as coloured versions of the tokens in Place Transition nets. In Figure 2.7 another representation for showing CPNs is depicted where instead of using letters, the different types of tokens are shown as different colours.

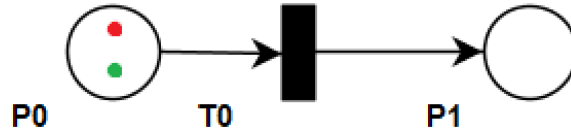


Figure 2.7: A simple CPN before it is unfolded

### Unfolding of a CPN

It is possible to unfold a Coloured Petri net to represent it as an ordinary Petri net. Consider the simple CPN in Figure 2.7 and its firing modes (Table 2.1). Given the firing modes and the current marking of P0, the firing of T0 means P1 could end up with three possible markings. The unfolding process will depict each of those markings as places of their own as shown in Figure 2.8.

$I^-(p_0, t_0)(red)$	$I^-(p_0, t_0)(green)$	$I^+(p_1, t_0)(red)$	$I^+(p_1, t_0)(green)$
0	1	0	1
1	0	1	0
1	1	1	1

Table 2.1: Firing modes for Figure 2.7

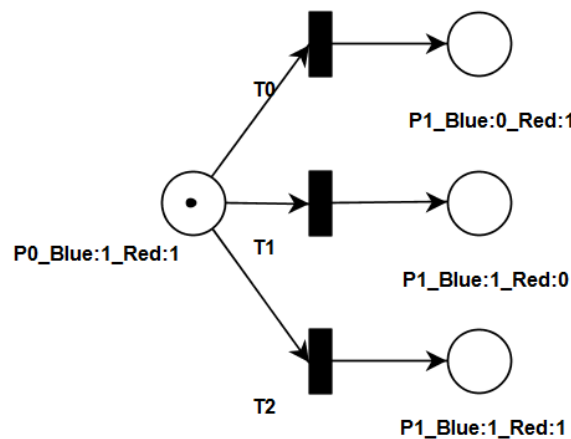


Figure 2.8: The CPN shown in Figure 2.8 unfolded to a Place Transition net

## 2.2.5 Coloured Generalized Stochastic Petri Nets

A coloured Generalized Stochastic Petri net [14] is an extension over the Generalized Stochastic Petri Net model to support coloured tokens. The resulting net is able to model immediate transitions, timed transitions whilst incorporating the ability to express complex synchronization patterns by using many types of tokens. Firing of transitions follow the same rules as CPNs with the addition that there is now either a firing frequency or a firing delay associated with the transition, depending on the type of transition involved.

A Coloured Generalized Stochastic Petri net (CGSPN) can be formally defined as follows:

**Definition 7.** A Coloured Generalized Stochastic Petri net is a 4-tuple:  $CGSPN = (CPN, T_1, T_2, W)$  where:

- $CPN = (P, T, C, I^-, I^+, M_0)$  is the underlying Coloured Petri 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,  $T_1 \cap T_2 = \emptyset$ ,  $T = T_1 \cup T_2$
- $W = (w_1, \dots, w_{|T_1|})$  is an array whose entry  $w_i$  is a function of  $[C(t_i) \rightarrow \mathbb{R}^+]$  such that  $\forall c \in C(t_i) : w_i(c) \in \mathbb{R}^+$ :
  - is the (possibly marking dependent) rate of a negative exponential distribution which specifies the firing delay with respect to colour  $c$ , if  $t_i \in T_1$ .
  - a (possibly marking dependent) firing weight with respect to colour  $c$ , if  $t_i \in T_2$ .

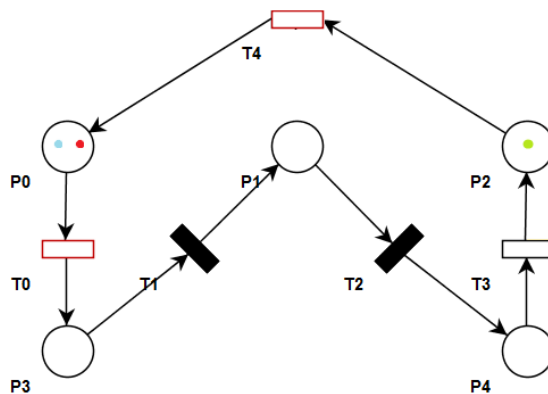


Figure 2.9: A simple CGSPN

## 2.3 Petri net Tools

### 2.3.1 Introduction

As Petri nets are becoming more and more popular, an increasing number of tools have been developed offering a variety of features to users. Using such tools, users are able to graphically design Petri net models, simulate the models through an animated sequence



and analyse their models in a number of different ways. The problem with the tools that are currently available is that although you can find a tool that allows users to separately perform these tasks, users will rarely find a single tool that can cater to all their needs. Instead, many find themselves having to import/export their Petri nets between tools that are suited for a particular task and when that is not possible they must re-create the Petri nets manually. The Petri net community seems to be lacking a complete Petri net tool-kit which has all the features users usually require, integrated into one application. The purpose of the research in this section is to find the tool that has the greatest potential of realizing this goal and the purpose of this project is to bring that tool one step closer to achieving it. Ideally, such a tool should have the following features:

- **Multi-Platform:** By limiting the user-base of the tool to specific operating systems, people would be discouraged from using the tool because they would not be able to easily share their Petri net models with colleagues and other interested parties. For this project, we assume that a multi-platform tool is fully functional on UNIX, Windows and Mac OS X operating systems as these make up the vast majority of computer usage around the world.
- **Open Source:** New ideas and extensions to the ordinary Petri net model are constantly being introduced within the Petri net community and so any successful tool must be available for extension in order to remain at the forefront of Petri net development. The larger the development team, the more likely the tool will be kept up to date and an open source project is the best way to obtain a large development team.
- **Embedded Graphical Animation:** A significant reason for Petri net’s popularity is that it offers an intuitive graphical component for simulation and modelling [13]. This vital component should not be missing from a “complete” list of features.
- **Structural Analysis:** Petri nets are applied in real life applications to model the behaviour and performance of systems. This is enabled in part by tools which make use of the mathematical theory underlying Petri nets, to analyse models for various structural properties such as boundedness or freedom from deadlock.
- **Stochastic Petri net support:** To analyse complex systems in a multitude of ways and provide quantitative, accurate information about their performance, the use of timed transitions is a necessity.
- **Coloured Petri Net support:** To allow for even greater expressiveness and to make complex models more intuitive, coloured tokens should be supported.

### 2.3.2 Petri net Tool Comparison

Currently, a total of 73 Petri net tools have been registered with the Petri Nets Tool Database [20]. Below is the complete list of these tools, rated against the criteria listed in Section 2.3.1:

	Multi-Platform	Open Source	Embedded Graphical Simulation	Structural Analysis	Stochastic Petri Net Support	Coloured Petri Net Support	Score
ALPHA/Sim			✓			✓	2
ARP		✓		✓			2
Artifex			✓	✓			2
CoopnBuilder	✓	✓	✓				3
COSA BPM	✓		✓			✓	3
CPN-AMI		✓		✓			3
CPN Tools		✓	✓			✓	3
ExSpect			✓		✓		2
FLOWer	✓						1
F-net			✓	✓	✓		3
GDToolkit		✓					1
Geist3D		✓					1
GreatSPN		✓	✓	✓	✓	✓	5
Helena		✓					1
HiQPN-Tool		✓	✓		✓		3
HiSim	✓	✓	✓				3
HPSim		✓	✓		✓		3
INA		✓		✓			2
Income Suite	✓	✓	✓				3
JARP	✓	✓	✓				3
JFern	✓	✓	✓				3
JPetriNet	✓	✓		✓			3
Kontinuum							
LoLA		✓					1
Maria		✓					1
MISS-RdP					✓	✓	2
The Model-Checking Kit		✓					1
Netlab		✓	✓	✓			3
Nevod			✓				1
Opera		✓	✓	✓			2
ORIS		✓	✓				2
P3		✓	✓				2
PACE					✓		2
PED		✓	✓				2
PEP		✓	✓				3
PetitPetri	✓	✓		✓			2
Petrogen		✓	✓				2
Petri-lld	✓	✓	✓				3
Petri Net Kernel	✓	✓	✓				3
Petri .NET Simulator			✓				1
Petri Net Toolbox			✓	✓	✓		3
PetriSim		✓					1

	Multi-Platform	Open Source	Embedded Graphical Simulation	Structural Analysis	Stochastic Petri Net Support	Coloured Petri Net Support	Score
PIPE2	✓	✓	✓	✓	✓		5
PNetLab		✓	✓	✓		✓	4
PNML Framework	✓	✓	✓				2
PNSim	✓	✓	✓	✓			4
PNetalk		✓	✓				2
Poses++		✓					1
Predator	✓	✓			✓		3
PROD		✓					1
ProM framework	✓	✓		✓			3
PROTOS				✓	✓		2
QPME	✓	✓			✓		3
Renew	✓	✓	✓				3
Romeo							
SEA		✓	✓				2
SIPN-Editor	✓	✓					2
Simulaworks, Petri Nets Simulator					✓		1
SNAKES	✓	✓					2
Snoopy	✓	✓	✓		✓		4
SPNP		✓			✓		2
StpnPlay		✓			✓		2
SYROCO		✓					1
TimeNET		✓	✓	✓	✓	✓	5
Tina	✓	✓	✓				3
TAPAAL	✓	✓	✓				3
Visual Object Net ++		✓	✓	✓			3
VisualPetri		✓	✓				2
WebSPN	✓	✓	✓		✓		4
WINSIM		✓				✓	2
WoPeD	✓	✓	✓				3
XRL/flower	✓	✓					2
YAWL	✓	✓	✓				3

Figure 2.10: Petri net tool comparison (score represents the number of required features the tool supports)

It is important to note that although some of these tools support CPNs, none of them claim to support CGSPNs. This highlights the importance of implementing CGSPNs in a Petri net tool. Figure 2.10 shows that three tools within the database score a 5 against the selected criteria. For this reason we further analyse the top ranked tools as shown in Figure 2.11 [6].

	GreatSPN 2.0	TimeNet 4.0	PIPE2
PNML			✓
<b>GUI</b>			
Undo		✓	✓
Redo			✓
Copy/Paste	✓	✓	✓
Arc Weight	✓		✓
<b>Features</b>			
Capacity Limitation	✓		✓
Priorities	✓	✓	✓
Server Semantics	✓	✓	
<b>Token Game</b>			
Step by step	✓	✓	✓
Backwards			✓
Continuous	✓	✓	✓
<b>Structural Analysis</b>			
Net comparison			✓
Place Invariants	✓	✓	✓
Transition Invariants	✓		✓
Siphons and Traps	✓	✓	✓
Qualitative Properties	✓		✓
<b>Analysis</b>			
Transient Analysis	✓	✓	
Transient Simulation	✓	✓	

Figure 2.11: Comparison of top ranked tools.

Of the three tools compared above, PIPE2 seems to have the most features. PIPE2 offers the most user friendly GUI out of the three and surpasses the competition when it comes to structural analysis. Although lacking in transient analysis and simulation, PIPE2 still seems like the wise option thanks to its ability to export to PNML format; if need be, such analysis could be handled by other tools. In such cases the user also has the option to create his/her own analysis module as PIPE2 offers a pluggable module interface. Furthermore, neither GreatSPN [7] nor TimeNET [12] are multi-platform and are only commercially available (free to academic institutions only). This may act as a barrier to independent researchers/developers, limiting the tool's development. Finally, although both tools support CPNs, they do not support CGSPNs which limits their expressibility. Although it is missing CPN support, PIPE2 seems to be the most viable candidate to become a fully featured, complete Petri-net tool. With the integration of CGSPN support in PIPE2 and some work on the analysis features, PIPE2 could become a one-stop gateway to the average users's needs.

## 2.4 PIPE2

### 2.4.1 Introduction

Platform Independent Petri net Editor 2 (PIPE2) is a tool developed in Java 1.4 for creating, saving, loading and analysing Petri nets including Generalised Stochastic Petri nets [6]. The tool was initially developed in 2003 as a MSc Group Project at the Department of Computing, Imperial College London under the name PIPE. Since then it has gone through many changes and enhancements and is still actively been developed by Imperial College London, Universitat de les Illes Balears [18, 6] and an external user community. With over 28,000 downloads, PIPE2 is becoming increasingly popular.

## 2.4.2 Version Selection

Officially, the latest version of PIPE2 is PIPE 2.6; however, the fact that multiple parties and institutions develop PIPE simultaneously has caused problems. More specifically, 2.6 is a branch originating from the 2.4 version and does not incorporate the extremely useful features of 2.5. This leaves a predicament as to which branch should be extended. Version 2.6 has vastly improved PIPE's analysis capabilities adding extra modules and Performance Trees whereas 2.5 has made significant improvements on the user interface, fixing various annoyances and adding undo/redo/copy/paste capabilities. Hopefully, the next version of PIPE will integrate both versions, but for the purpose of this project, 2.5 is best suited because a more robust and user friendly GUI will be much more valuable in our goal of extending the GUI to support CGSPNs. For this reason the analysis in the remainder of this chapter is based on PIPE 2.5 with references to 2.6 where appropriate.

## 2.4.3 Features

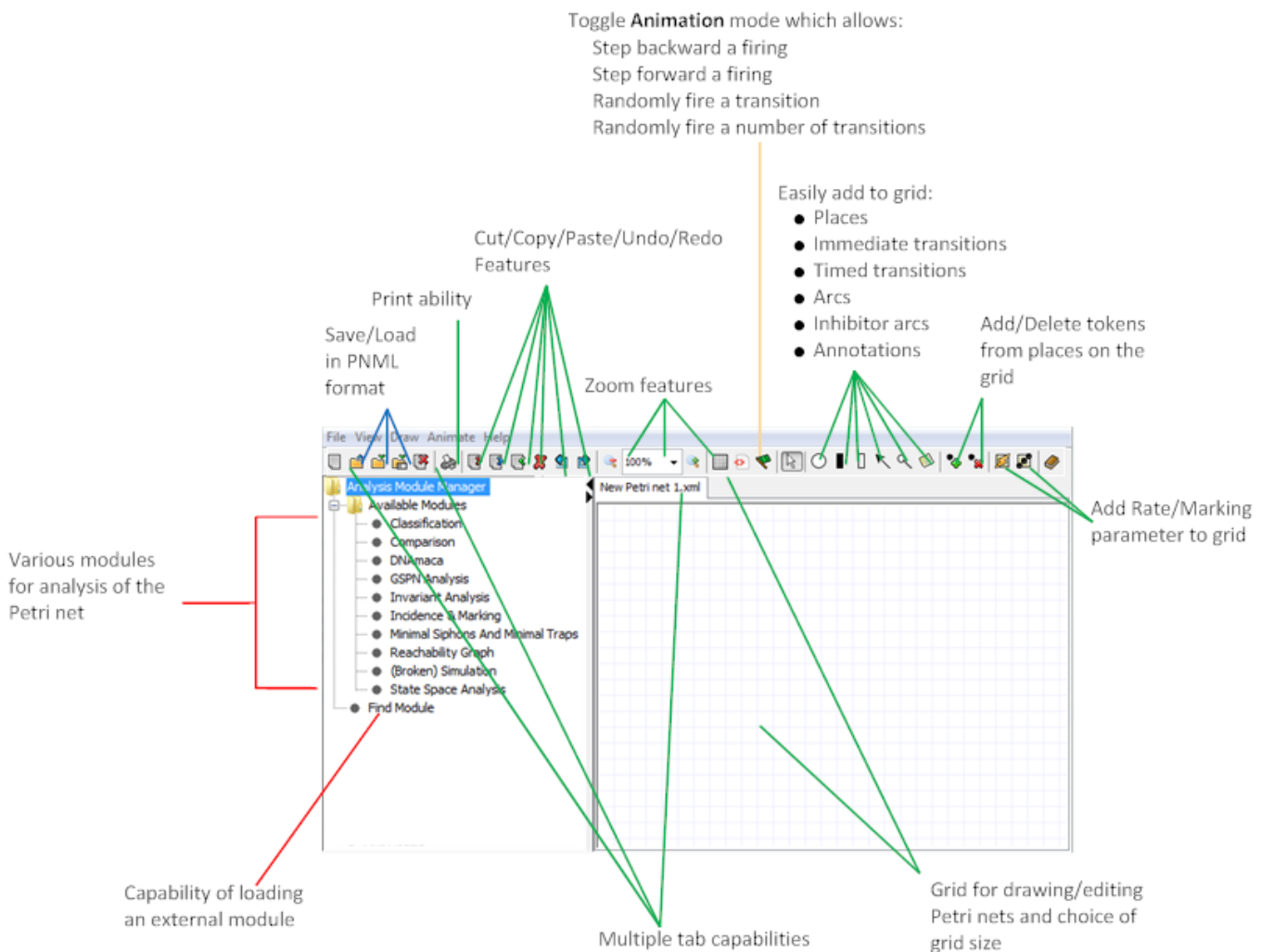


Figure 2.12: The PIPE2 interface

In addition to the ideal features we looked for (Section 2.3.1) before choosing to extend this tool, PIPE2 also offers other important and useful features [21]:

- Free to download and use
- Simple and intuitive interface
- Supports the modelling of Generalised Stochastic Petri nets
- Provides several analysis modules including advanced GSPN analysis
- Is capable of handling hundreds of thousands of states and eliminating vanishing states ‘on the fly’
- Generates reachability graphs
- Conforms to PNML (Petri net Markup Language), which has become the industry standard interchange format so it can be integrated with other tools.
- Allows export to Postscript/PNG files
- Zoom in/out feature
- Edit/Undo features
- Native support for CVS

#### 2.4.4 Component Overview

For clarity, PIPE2 can be functionally separated into various components (Figure 2.13) and in this section we will analyse each of these to understand PIPE better.

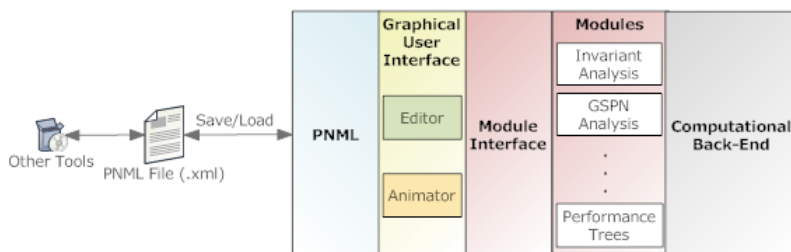


Figure 2.13: Components within PIPE2

#### PNML

Petri Net Markup Language is a Petri net interchange format, designed so that Petri nets can be imported/exported between various applications. It is implemented using XML technology and has become popular thanks to its:

- **Flexibility:** Is able to represent any kind of Petri net by storing Petri nets as labelled graphs. This allows information to be stored in labels attached to the places, the arcs or the net itself [4].

- **Compatibility:** Since different types of Petri nets can be represented, PNML ensures that as much information as possible can be exchanged between these different types. It does this by providing a conventions document [8] used to define a label with a particular meaning [4].
- **Extensibility:** Is structured in a way that it may be modified to support different types of Petri nets that may be introduced.

```

<pnml>
<net id="Net-One" type="P/T net">
  <place id="P0">
    <graphics>
      <position x="210.0" y="135.0"/>
    </graphics>
    <name>
      <value>
        P0
      </value>
    </name>
    <initialMarking>
      <value>
        1
      </value>
    </initialMarking>
    <capacity>
      <value>
        0
      </value>
    </capacity>
  </place>
  .
  .
  .
</net>
</pnml>

```

Figure 2.14: The PNML representation of P0 in Figure 2.1 as given by PIPE2

Some common labels used in PNML are:

**<name>**: The identifier of an element.

**<initialMarking>**: Describes the initial marking.

**<capacity>**: Describes the capacity of a place.

**<graphics>**: Has to do with the graphical representation of a PNML document and encapsulates information such as the position and offset of various items.

PIPE2 utilizes PNML by using it to save/load any net created in the GUI. When the user clicks on “save” in the GUI, the `DataLayerWriter` class generates an XML document with elements representing all the objects of the net. Likewise when a user selects a PNML file to load, the `PNMLTransformer` parses and transforms the file into objects that can be handled by PIPE.

## Graphical User Interface

With multiple classes implemented within the `pipe.gui` package, the GUI provides an interface where users can easily draw and edit Petri nets. The GUI consists of two parts:

- **Editor**

- Undo/Redo/Copy/Paste functionality: Features vital for any user friendly interface are accessible via the standard keyboard shortcuts, from the button panel or from the menu.
- Multiple Tabs: Allows users to work on more than one Petri net at a time and is implemented by adding additional views to the MVC model and switching view where appropriate.
- Grid: Objects such as places, arcs and transitions can be added onto the grid using buttons on the button panel. Once on the grid the user can shift them around and edit their properties by right clicking on them.
- **Animator:** PIPE2 allows users to toggle the “Animation Mode” (also known as the token game). This is implemented by creating a new model which updates every time a user fires an enabled transition by clicking on it. When the animation mode is toggled off the previous model is restored and the user is able to see the Petri net he/she was previously working on. This mode allows users to:
  - manually fire any of the enabled transitions (highlighted in red) within a Petri-net
  - automatically execute random transitions by specifying firing delay and the number of firings
  - step backwards and forwards through the various states caused by each firing transition

Other self-explanatory features of the GUI can be seen in Figure 2.12.

## Modules

PIPE2 has a number of modules capable of carrying out quantitative and qualitative analysis. The set of modules is extendible as a module interface exists which developers may implement to add new modules. Currently the following modules are available in PIPE2:

- **Classification:** Uses the structure of a given Petri net to determine if it can be classified as one of the following: EFC-Nets, ESPL Nets, SPL-Nets, FC-Nets, Marked Graph, State Machine.
- **Comparison:** Works out if two Petri nets (derived from PNML files) are functionally the same, otherwise it displays their differences.
- **DNAmaca interface:** Allows PIPE to interact with DNAmaca - a tool for generalised markovian analysis of timed transition systems. Hence the module is able to produce passage time analysis statistics for a Petri net, which represent the distribution of the time it takes for a system to go from one given state to another.
- **GSPN analysis:** Explores the state space of a Petri net to calculate the average number of tokens in a place, the throughput of timed transitions and the token probability density.

- Invariant analysis: Calculates place invariant vectors, transition invariant vectors, as well as providing marking equations and information about boundedness and liveness.
- Incidence & marking: Displays the forward, backward and combined incidence matrices. Also generates the marking matrix and the set of enabled transitions.
- Minimal Siphons And Minimal Traps: Generates the minimal siphons (sets of places that never gain a token once none of their places is marked) and traps (sets of places that never lose all tokens once at least one of their places is marked) for a given Petri net [6]. These can then be used to check properties such as liveness and reachability.
- Reachability graph: Graphically shows every possible firing sequence of a given Petri net and gives information about boundedness, safeness and deadlock-free properties [6].
- Simulation: Since full analysis of a Petri net may require a huge amount of resources this module gives performance results on a simulation which computes the average number of tokens per place with the 95% confidence interval for each place in the net [6].
- State space analysis: Builds a tree of all the reachable markings in a Petri net to calculate properties such as boundedness, safeness and deadlock-free. Can also detect and output the shortest path to any existing deadlock.

## Computational Back-End

This component is used to describe the classes that support the above components. Although less visible to the user, this component provides the necessary calculations which allow the other components to function correctly. Determining whether or not a transition is enabled and processing fired transitions are examples of such calculations.

### 2.4.5 Architecture

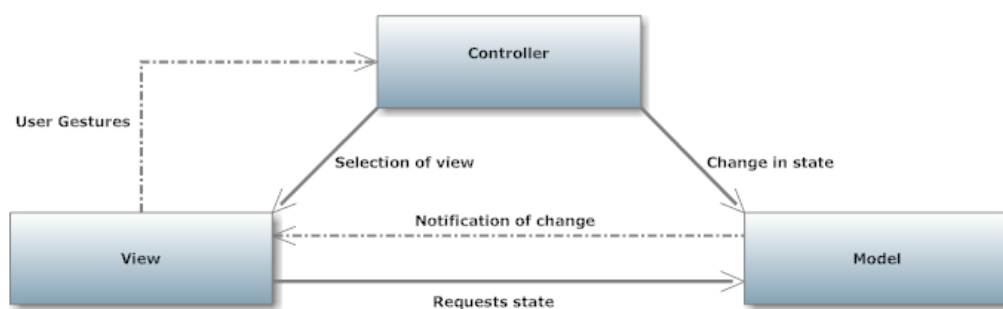


Figure 2.15: Model View Controller design pattern

PIPE was initially designed to follow the MVC design pattern [5]. The design pattern consists of the following components:



- Model: Represents the overall state and the data the application represents. In PIPE2, the `DataLayer` class in the package `pipe.common.dataLayer` acts as the model and constantly notifies its observers (the view) of any changes made to it e.g. a place has been added to the grid.
- View: Renders the model so that users can view the model and send gestures. In PIPE2 classes `GuiFrame` and `GuiView` in package `pipe.gui` both act as views allowing users to interact with the application.
- Controller: Receives input and responds by invoking methods on the model. The swing components within PIPE2 act as the controllers, mainly the `GuiFrame` class in the `pipe.gui` package.

As explained above, the fact that `GuiFrame` seems to act as both the observer and the controller, partly violates the MVC pattern and should be restructured.

## Chapter 3

# Overview of Architectural Changes

### 3.1 Introduction

In this chapter we give a general overview of the PIPE architecture and all the changes that had to be made to extend PIPE.

The major challenge throughout this project was understanding what the various classes and methods are responsible for so we could comprehend which modules had to be modified. PIPE fails to clearly separate the various packages and many cyclic dependencies exist between them (examined in Chapter 10). For this reason, it was impossible to examine each package and class in isolation and hence very difficult to understand which classes/packages would be affected by a particular change. This forced us into another approach for making our modifications, where a key change was made to a particular class causing a domino effect of errors on various other classes/packages which were taken care of one by one. As each error was tackled, the class involved was examined and understood. This approach may have led to a good understanding of the interactions between PIPE's classes and packages, but was very challenging at times. For example, one key change we made was changing the marking of a place from an `int` to a collection of other items (outlined in Section 4.3). This change caused so many errors in inter-dependent classes, that it took several days of non-stop programming to overcome these and get PIPE to compile. This meant we were blindly making changes without having the luxury of checking each change for correctness.

To achieve our target we had to understand, modify and extend 51 of the 204 PIPE classes. In addition we have added one new package and 22 new classes.

	Old PIPE	New PIPE
Lines of Code	~36000	~41000
Packages	27	28
Classes	204	226

Figure 3.1: Overview of changes

## 3.2 Packages

In the subsequent chapters, we carefully analyse the major changes that were made to achieve a certain task. However, it was deemed impractical to examine all the small and uninteresting changes that had to be made to a multitude of different classes to cater for our new functionality. For this reason, in this section we present compact diagrams depicting all the PIPE packages and their classes to give an overview of exactly what was modified and added to cater for our new functionality. Note that the figures below are focusing on the changes made rather than the architecture of the system. For this reason, along with the fact that we are describing a very large number of classes, we have not used UML diagrams here. PIPE's architecture will be discussed in more detail in the chapters that follow.

Referring to Figures 3.2 - 3.9:

- Blue coloured classes: Existing classes in PIPE that were modified
- Purple coloured classes: New classes that were added to PIPE
- Red letters depict the reason a specific class was modified/created. The following letters on the diagram show why each class was created/modified:
  - *D*: To support the design of CGSPNs. Details of changes are available in Chapter 4.
  - *A*: To support the animation of CGSPNs. Details of changes are available in Chapter 5.
  - *G*: To support an elegant representation of CGSPNs. Details of changes are available in Chapter 6.
  - *U*: To support the unfolding of a CGSPN to a GSPN. Details of changes are available in Chapter 7.
  - *L*: To support the analysis of CGSPNs. Details of changes are available in Chapter 8.

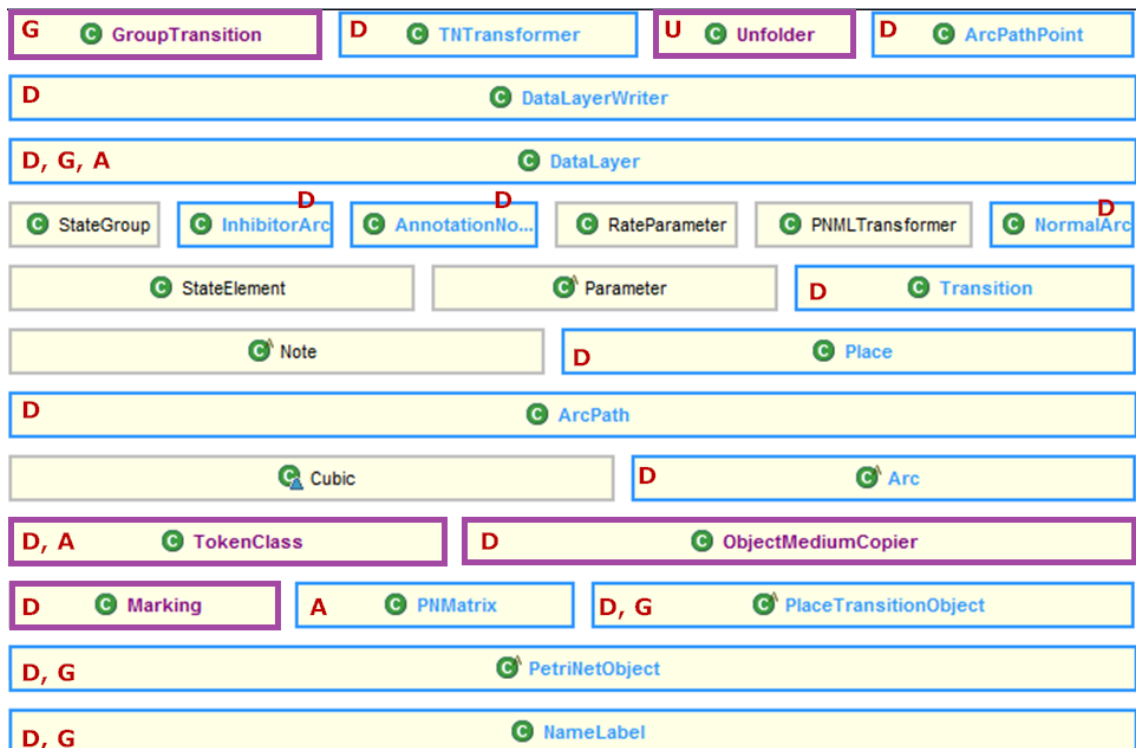


Figure 3.2: Package dataLayer

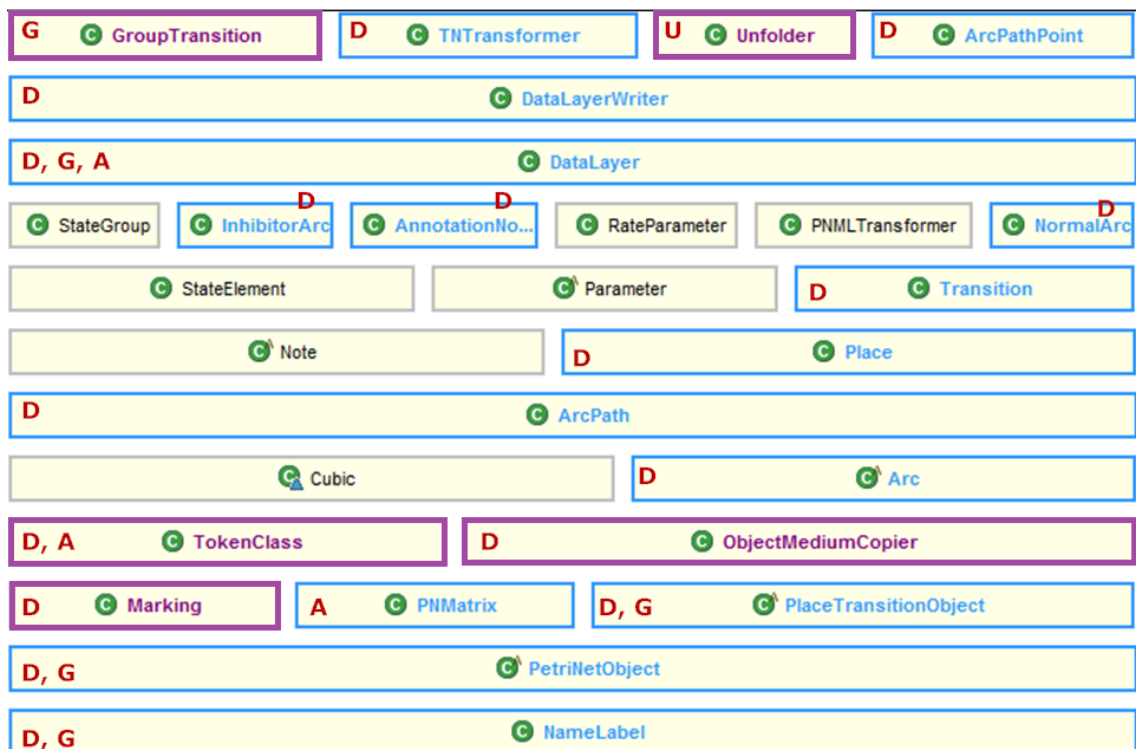


Figure 3.3: Package dataLayer.calculations

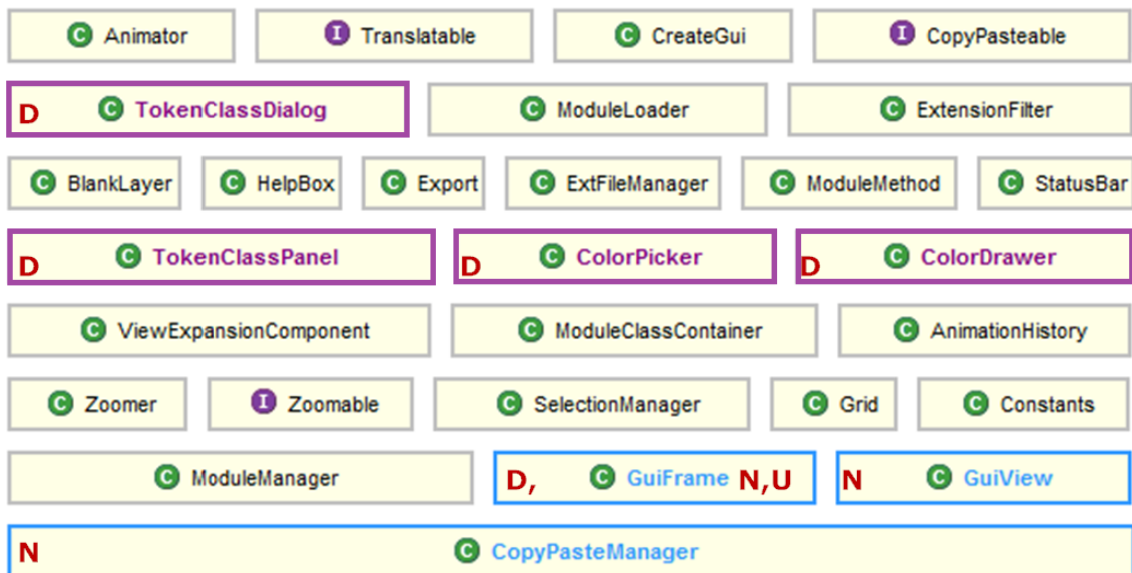


Figure 3.4: Package `gui`

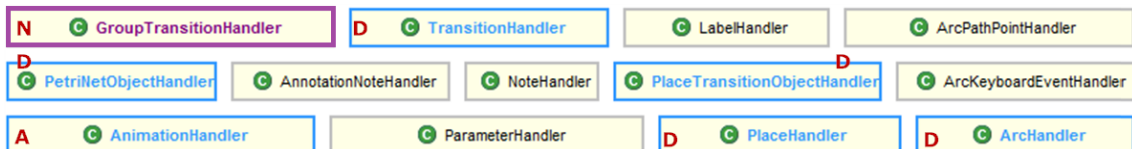


Figure 3.5: Package `gui.handler`

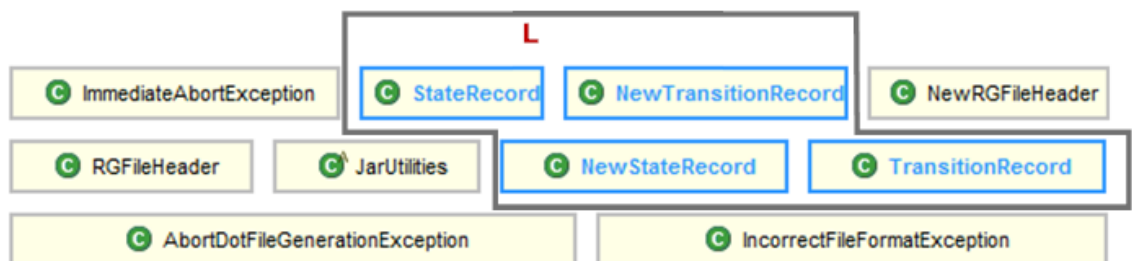


Figure 3.6: Package `module`

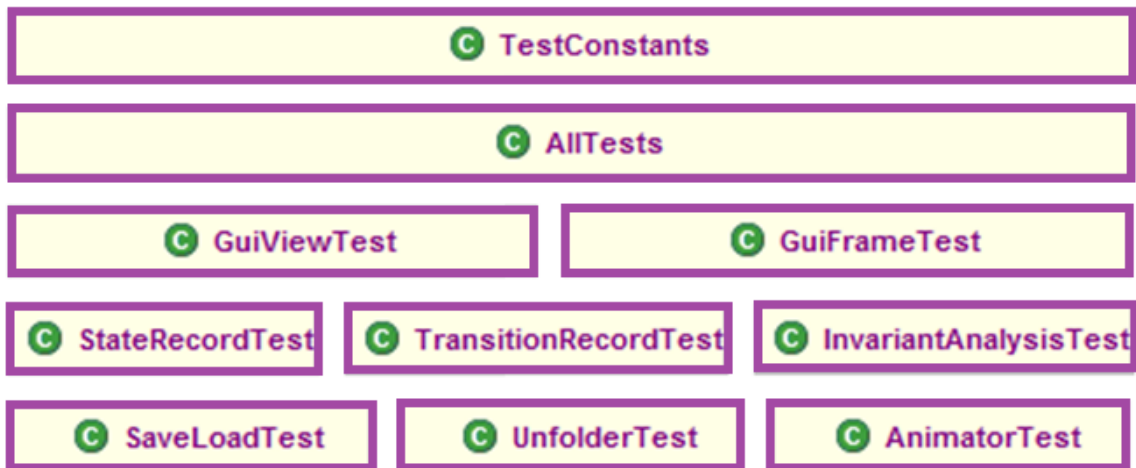


Figure 3.7: Package `test`, PIPE's new test framework (Chapter 9)

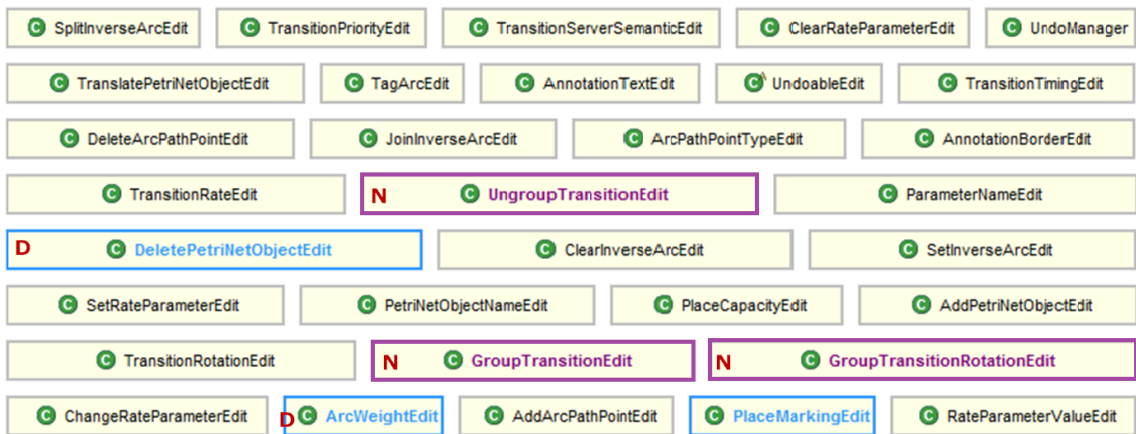


Figure 3.8: Package `gui.undo`

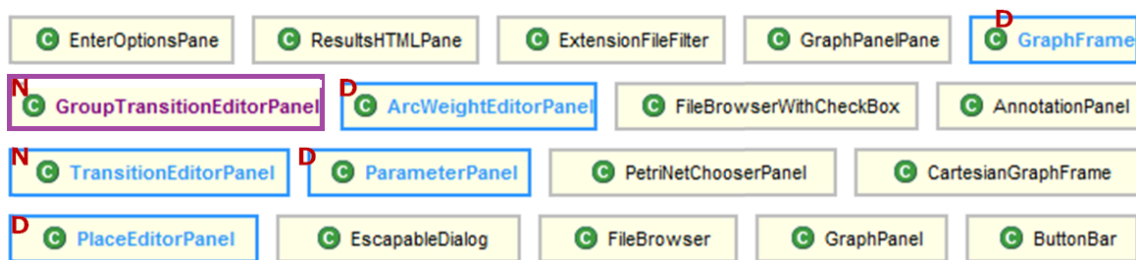


Figure 3.9: Package `gui.widgets`

## Chapter 4

# Supporting the design and editing of CGSPNs

### 4.1 Introduction

We would like to extend PIPE to allow the design and editing of CGSPNs. This will enable users to draw up and visualise graphical representations of complex models. Ultimately, users will be able to represent complex realistic models using compact and concise models.

### 4.2 Add/Edit/Remove token classes

The first place to start when creating a CGSPN is adding the functionality for creating different types of tokens. We shall refer to different token types as ‘token classes’. Currently there is no concept of token classes in PIPE. A token can only be of one type (a black token). This section explains how we were able to allow users to create new token classes and then edit/delete them at will.

#### 4.2.1 Design decisions and challenges

- To guarantee the integrity of a Petri net, validations must exist that will ensure that any new token class has a unique, non-empty name and that at least one token class is enabled at any one point.
- The user should have a way of enabling/disabling previously defined token classes but should not be allowed to disable a token class that is being used.
  - This would require the program to track which places are currently marked with specific token classes. A token class can only be disabled if no place is currently marked with it.
- How should we represent coloured tokens?

- **Option A:** The standard form of coloured Petri nets as presented in [3] takes the following form:

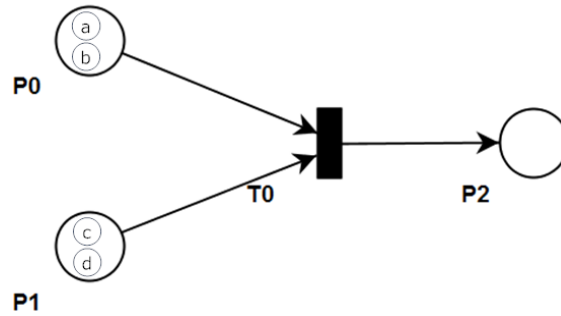


Figure 4.1: A basic CPN

Tokens are transformed from simple dots to circles denoting the class of each token, i.e. in the diagram there are 4 token classes (a, b, c, d). We could implement CGSPNs using this notation as is done in the Petri net tool CPN Tools.

- **Option B:** Tokens are simple dots as before. To differentiate between various token classes we colour each token differently depending on the token class it belongs to.

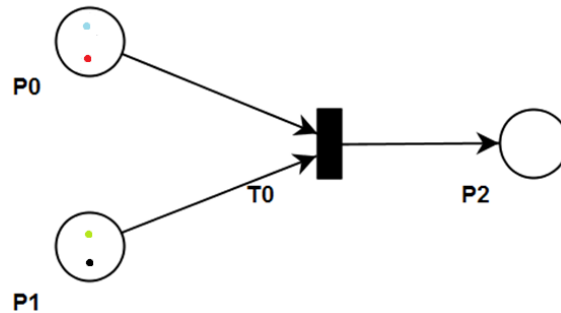


Figure 4.2: A basic CPN

Although there are no obvious reasons for selecting **Option A** over **Option B**, **Option B** has the huge advantage of allowing for a less cluttered interface. Using **Option A**, each token must be expanded in size in order for the letter to fit. This would waste unnecessary space and would make it impossible to fit several tokens in a single place. For this reason **Option B** was chosen and token classes will be represented using normal tokens coloured depending on the token class they belong to.

#### 4.2.2 Overview of new functionality

The goal is to have a button on the toolbar which brings up a window where a user can easily add/edit/remove token classes. The window should allow the user to define the



name of a token class, choose a colour for it and decide on whether or not he/she would like to enable it in the current tab. In the back-end, when a new token class is created a new instance of a class: `TokenClass` is generated.

### 4.2.3 New Architecture

A class: `TokenClass` was created with the following properties:

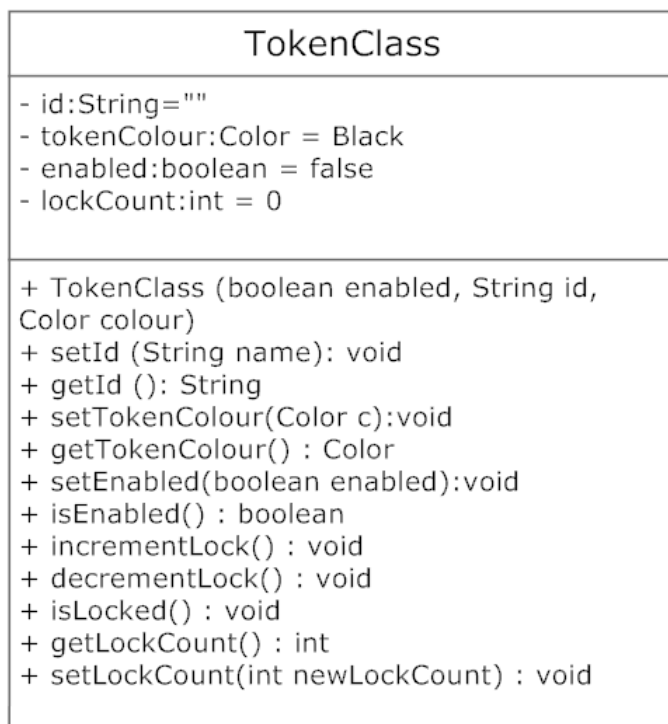


Figure 4.3: Class: tokenClass

The lock count serves as an integrity check. It is incremented each time a token class is being used in a place. This way the program can ensure that a user does not disable a token class that is currently in use.

### Token Class Editor window

To create this window I created 4 new classes:

- `TokenClassDialog`: This class is an action listener for `TokenClassPanel` and controls the OK/Cancel option offered within the panel.
- `TokenClassPanel`: This class generates a table with 3 columns(Enabled (Yes/No), Name, Colour). It allows the user to easily define and edit new token classes.
- `ColorDrawer` and `ColorPicker`: Allow user to select a colour from a palette and display selection in the table above.

Included in the `TokenClassPanel` are multiple validation checks on the user input to ensure that any new token classes have a unique, non-empty name and that at least one token class is enabled at any one point. It also denies the user the ability to disable a token class that is currently in use (by calling the token class' `getLockCount()` method).

## Integration

Having created a suite of classes capable of allowing a user to edit and create new token classes, this functionality now had to be integrated into the existing code. To add the Token Class Editor to the GUI, we had to make a few simple modifications to the `GUIFrame` class. A new button was added to the toolbar which creates an instance of `TokenClassPanel`. The class `TokenClass` was added to the `pipe.dataLayer` package as it is now an integral part of the model since the state of the model is now partly defined by the token classes available in a Petri net. The class `dataLayer` (the model in the MVC architecture) was also modified. Only the modifications are listed in Figure 4.4

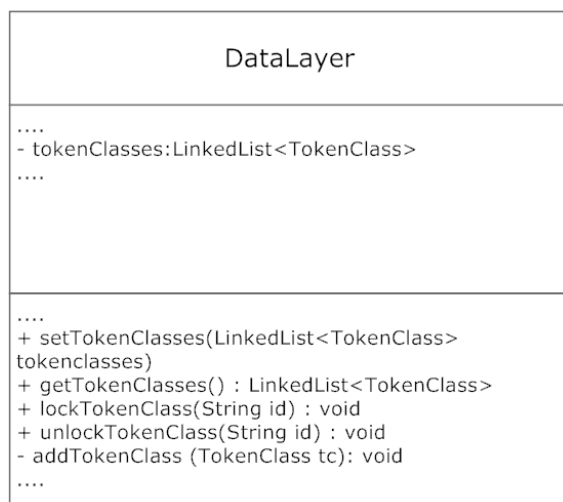


Figure 4.4: Modifications to class: `DataLayer`

New token classes are created by the user in the `TokenClassPanel`. In the back-end, the panel passes these new token classes to the `DataLayer` using the method `setTokenClasses`. The functions relating to the lock are for place objects to use when they make use of a specific token class. This way the model knows how many places are making use of each token class. If no place is using token class the lock is zero and the user will be allowed to edit/delete the token class.

#### 4.2.4 Final result

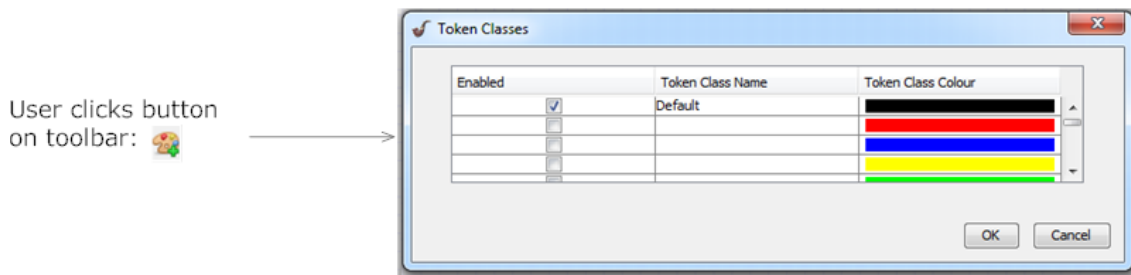


Figure 4.5: The Token Class Editor window

When the user clicks on the “Specify token classes” option under the “Draw” menu or on the toolbar button, the dialog in Figure 4.5 is displayed. The user is able to modify the contents of the table directly. Upon input the user is informed of any validation errors such as that shown in Figure 4.6. In such cases the input reverts to the previous input.

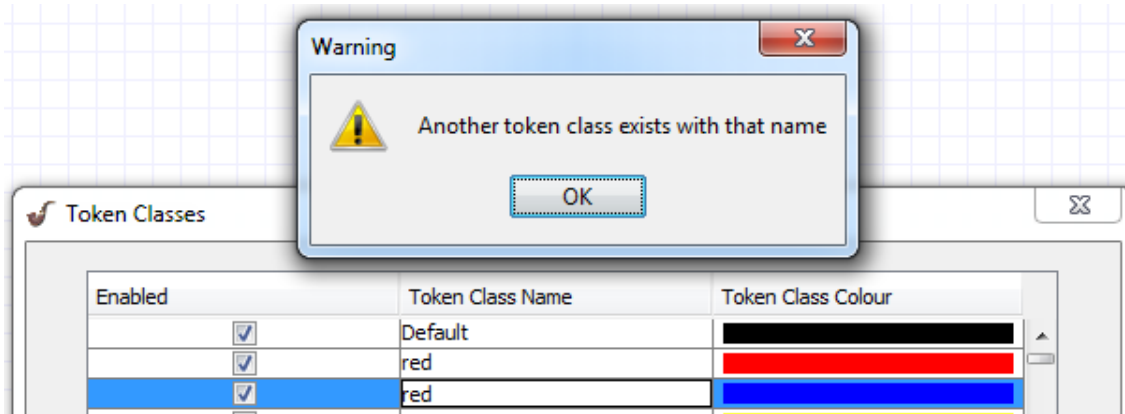


Figure 4.6: An example of a validation error

The user can choose a colour by clicking once on a cell in the colour column of the table. A new dialog appears (see Figure 4.7) giving the user 3 different methods for choosing a colour.

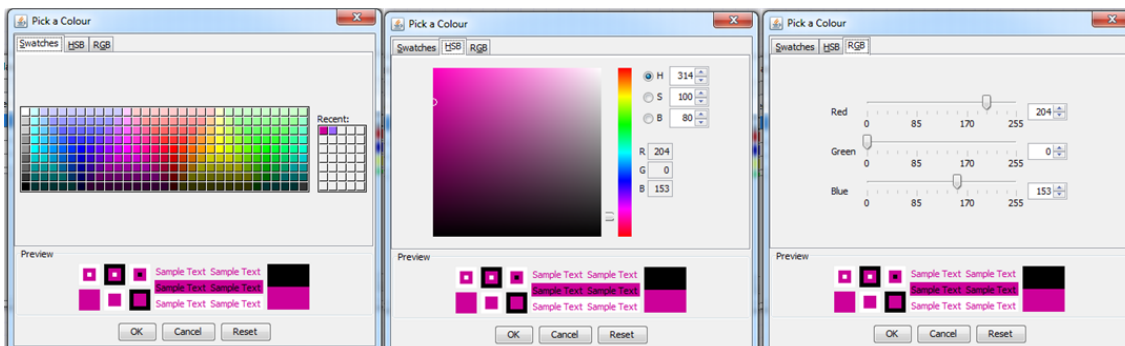


Figure 4.7: Colour chooser

### 4.3 Marking a place with different token classes

Having implemented the ability to create new token classes within PIPE, we must now design and implement a mechanism for the user to mark places with a specific token class. In particular, we must enable the user to:

- Mark places with any of the previously defined token classes.
- Set a limit on the total number of tokens in a place.
- Edit the marking of a place as easily as before when dealing with the simpler GSPNs.

#### 4.3.1 Existing Functionality

The existing PIPE only allows a user to mark places with one type of token. As a result, the user interface simply presents the user with two buttons, one for adding tokens to a place and one for removing tokens from a place.



Figure 4.8: The current method of adding/removing tokens

The process of adding a token to a place always results in a black token being marked on the place. Due to space limitations, if more than 5 tokens are added to a place then individual tokens are all replaced with a number depicting the marking of that place.

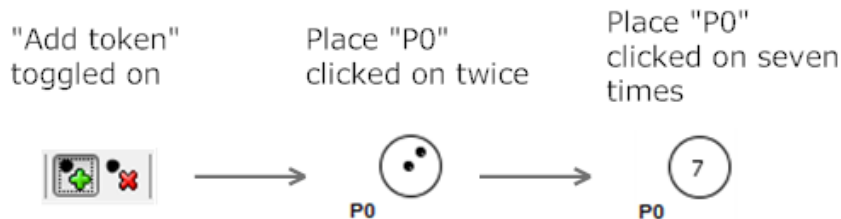


Figure 4.9: The current method for marking places

There is also an alternative method for adding/removing tokens to a place. The following dialog is available when a user right clicks on a place and selects "Edit place".

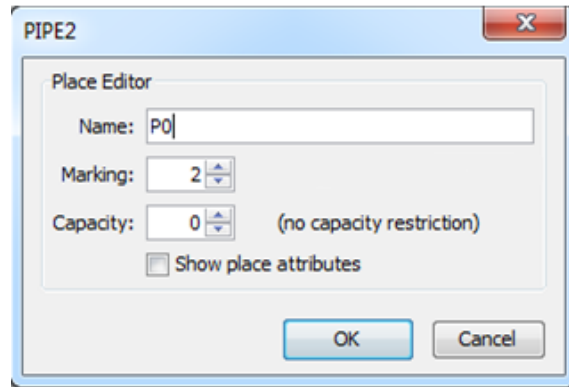


Figure 4.10: The current edit place dialog

The capacity field allows the user to set a limit on the total number of tokens that can exist in this place.

### Usability features

- A user can undo/redo the marking of a place by using the shortcut keys or the buttons on the toolbar.
- A user can increment/decrement the marking of a place by hovering over the place on the grid and scrolling up/down.

### 4.3.2 Existing Architecture

Currently, the marking of a place is not modelled by an object, but instead is represented as a simple `int` property of the place object. When a user carries out the steps depicted in Figure 4.9 the place's marking is updated. The place's marking is only incremented if the new marking does not exceed the place's capacity. Once an update has been made, the object's `repaint()` method is called which draws the tokens depending on the value of this `int`.

### Usability features

Undo/Redo features are implemented in the package `pipe.gui.undo`. This contains the following vital classes:

- **UndoManager**: Responsible for keeping track of any event occurring so that when the user clicks on undo/redo the appropriate action is taken. The events the **UndoManager** keeps track of are all of type **UndoableEdit**.
- **UndoableEdit**: Abstract class containing `undo()` and `redo()` functions. Any new type of action must extend one of these and define its own `undo()` and `redo()`.

The functionality of changing the marking via mouse scroll is also found in the **PlaceHandler** class under the `mouseWheelMoved()` method.

### 4.3.3 Design decisions and challenges

- Should a new token class be available in other tabs?
  - Each Petri net in each tab is trying to model something different and hence is associated with its own unique token classes. For this reason, the token classes should not be shared between tabs.
- Currently the problem of overloading a place with too many tokens is simply overcome by replacing all tokens with a number. However, with the introduction of an arbitrary number of token classes there would have to be space for more than one number in each place.
  - Each token class has its own number. Each number is displayed vertically below the other. When the vertical space runs out we can add more numbers in another column.

### 4.3.4 Overview of new functionality

An ideal way to preserve the trivial method of creating a simple Petri net is to allow the marking of places with coloured tokens to take place in an almost identical way as before. The proposed method is to have a combo box from which the user can select the token class he/she wants to work with. Once selected, the user can add/remove tokens exactly as before with the method discussed in Section 4.3.1.

Furthermore, the user can edit the marking of a place by right clicking the place and selecting “Edit Place”. Here, the user should be able to edit the marking of each individual token class within this place. Finally, as before, the capacity will define the maximum number of total tokens allowed in the place and can be set from the same menu.

The undo/redo and all shortcut features should work as before.

### 4.3.5 New Architecture

A class `Marking` was created consisting of an integer (to represent the marking of a place) and the token class (Figure 4.3) associated with this marking.

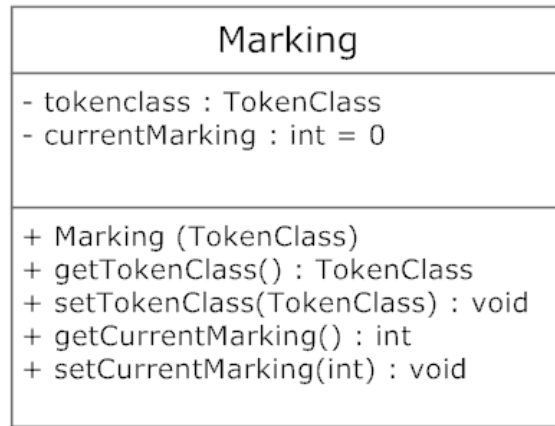


Figure 4.11: Class: Marking

To allow for the user to add/remove a specific type of token class to a place, a combo box was implemented. The choices of the combo box are the token classes the user has previously defined. The combo box was implemented in the view/controller `GuiFrame` (recall that the current implementation has no clear distinction between view and controller). The choices of the combo box are generated by the model's `getTokenClasses()` method. Once a selection is made the model is updated with a new "Active token class". This means that any time a user tries to add/remove a token from a place, it will add/remove tokens of that specific type. In order for this to work the model also updates the active token class on all places so that when the `PlaceHandler`'s `addListener()` is invoked (i.e. an action commanding it to add tokens) it knows which coloured token to add.

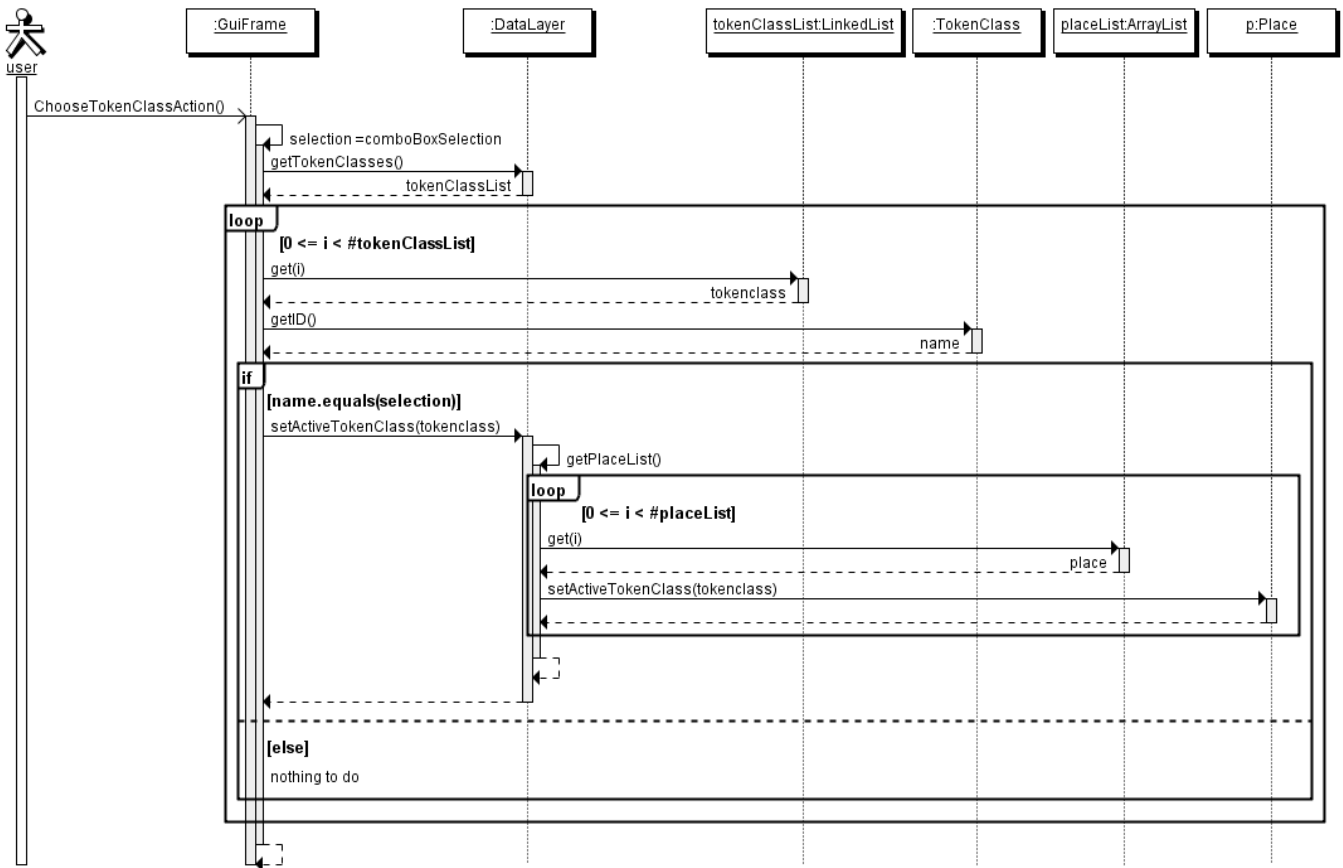


Figure 4.12: User chooses active token class

The “Edit Place” window was also completely revamped. The class `PlaceEditorPanel` was altered so that instead of offering one input box for the marking of a place there are now multiple boxes: one for each token class.

## Integration

Having defined a class `Marking`, we must now alter the `Place` class to hold a collection of these. As discussed in 4.3.3, a `LinkedList` will be used for this. A private method `getTotalMarking()` was also defined as a helper method since calculating the total number of tokens in a place was a common task.



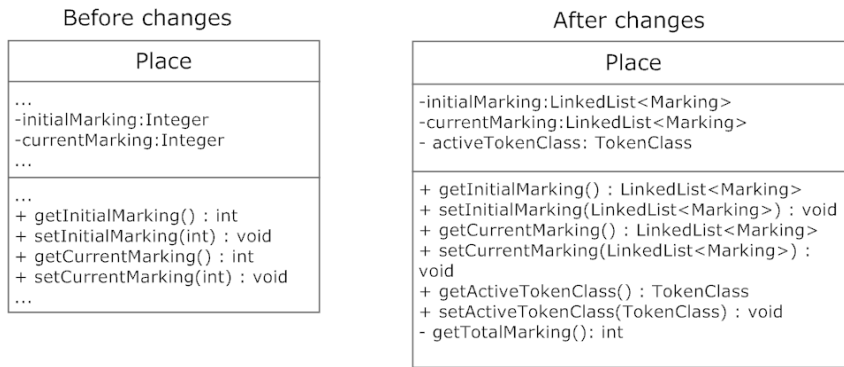


Figure 4.13: Class: Place

We also had to modify `PlaceHandler` (package `pipe.gui.handler`) which handles any action made to the place on the GUI. One of these actions is adding new tokens to a place via the button on the toolbar or the mouse scroll. The class was modified to generate a new linked list of markings based on which active token class is selected and using it as the new marking of the place.

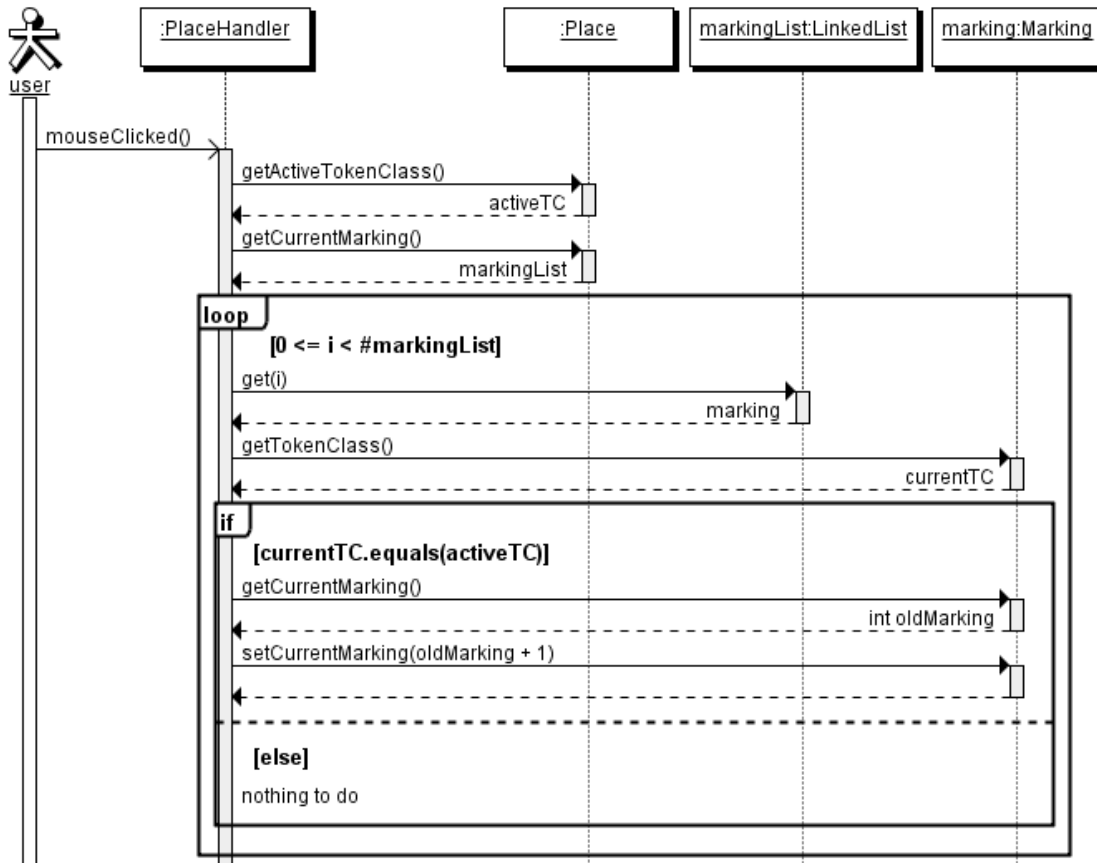


Figure 4.14: User adds a token to a place

## Undo/Redo

The undo/redo feature for marking a place currently works by storing the old integer value of the marking and replacing it with the current marking where necessary. Intuitively we must replace that integer with a `LinkedList` of objects of type `Marking`. The class responsible for the undo/redo of a place is: `PlaceMarkingEdit`.

### Problems Encountered

With the implementation so far, every time a user changes the marking of a place, the old marking is stored in the `PlaceMarkingEdit` object. This way the current marking of the place changes and when the user clicks on undo, the current marking is replaced with what we had previously stored in the `PlaceMarkingEdit` object. In reality however, because Java passes objects by reference, the old marking we have stored for undo/redo purposes is simply a reference to the current marking and hence they will always have the same value. This method worked before, because the marking was a primitive type (`int`), and Java chooses to copy such variables rather than reference them.

To overcome this problem, we initially thought of doing a deep copy of the current marking so that `PlaceMarkingEdit` has its own copy. However, each marking is associated with a `TokenClass`, which we would like to keep the reference to since any changes made to any `TokenClass` (e.g. a change in colour) must be reflected in all places. Since we were looking for something in between a deep and a shallow copy, we defined a new class: `ObjectMediumCopier`. The `mediumCopy()` method of `ObjectMediumCopier` creates a copy of the integer value of each marking while retaining the reference for the token class of that marking. The `PlaceMarkingEdit` first makes a medium copy of the marking that was passed to it before storing it for any future undo/redo action.

### 4.3.6 Final result

After defining multiple token classes, the user can now select which one he would like to work with by using a combo box on the toolbar (Figure 4.15).

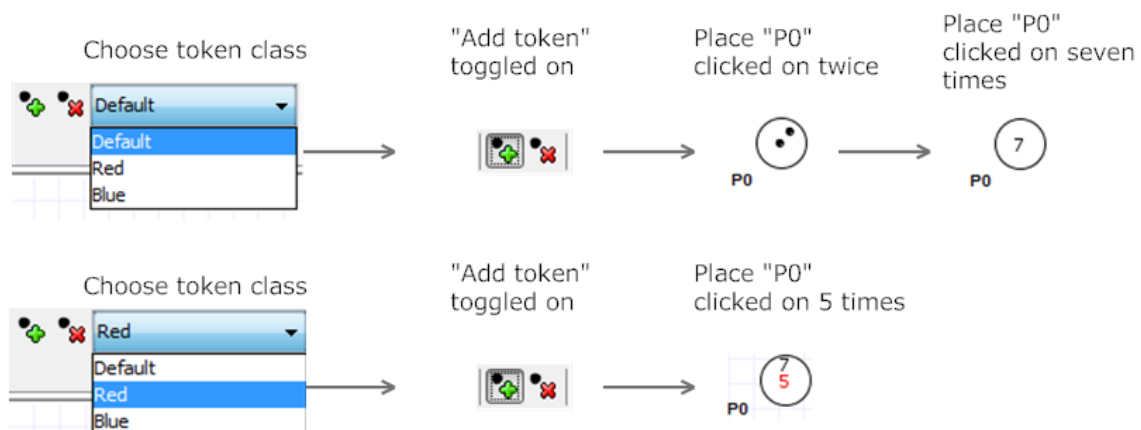


Figure 4.15: Process for adding different tokens to a place

Once selected, the user can then add an arbitrary number of tokens to any place and tokens of that type will appear in the place.

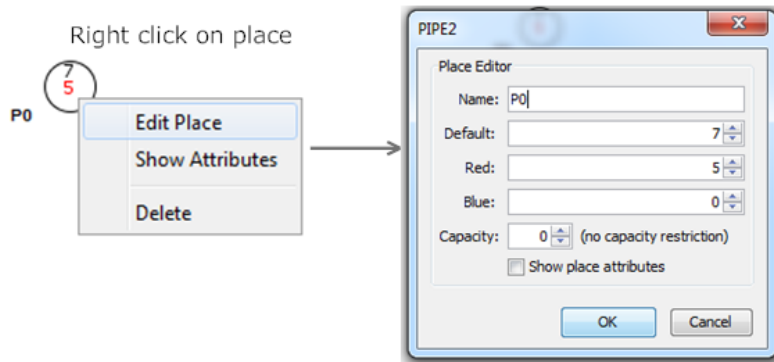


Figure 4.16: The new edit place dialog

There are validations in place to ensure the integrity of the Petri net, such as:

- Ensure that only positive integers are allowed as input.
- Ensure that the total marking does not exceed the set capacity of the place (a capacity of 0 denotes no restriction).

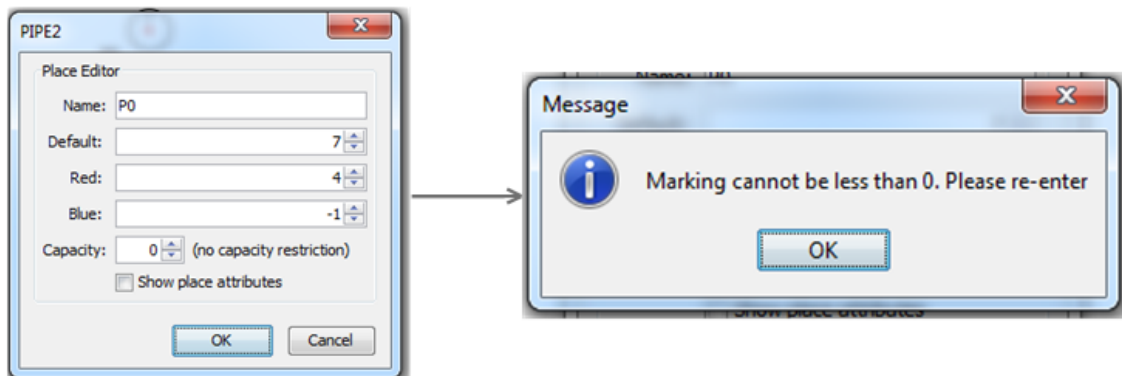


Figure 4.17: An example of one of the implemented integrity checks

#### 4.4 Add/Edit Multiple Arc Weights On A Single Arc

Having implemented the ability to create new token classes within PIPE, we must now design and implement a mechanism for the user to edit arc weights to cater for specific token classes. Let us consider the Petri net shown in Figure 4.18. Currently the user can only edit the arc weight of black tokens, but what if the user wanted to fire both a red and a black token at once? The arc weight must therefore be expanded to support multiple token classes.

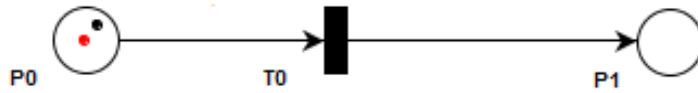


Figure 4.18: Example showing the need for multiple arc weights

In particular we would like to:

- Allow for multiple arc weights on a single arc (One for each enabled token class.
- Allow for editing of such multiple arc weights.
- Display such multiple arc weights in an intuitive and clear manner on the GUI as to reduce cluttering.

#### 4.4.1 Existing Functionality

Currently the user can edit the arc weight of an arc by right clicking on it and selecting “Edit Weight”.

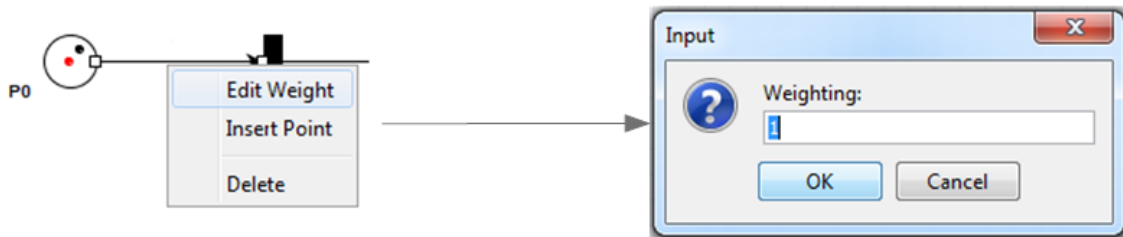


Figure 4.19: Current mechanism for editing arc weight

#### 4.4.2 Existing Architecture

Currently the weight of an arc is a property of the abstract class `Arc`. Extending this class is a `NormalArc` class which contains functions for changing the arc weight. The “Edit Weight” menu option (Figure 4.19) is controlled by the `ArcHandler` class. The `ArcHandler` implements an `actionListener` that calls upon the class `ArcWeightEditorPanel` which is the class involved for the resulting popup.

#### 4.4.3 Design decisions and challenges

- How will we display multiple arc weights at once without cluttering the interface?
  - **Option A:** We could hide all arc weights unless a user hovers over an arc. This could prove tedious for somebody wanting to understand how a model works as the user would have to hover over all the arcs one by one.
  - **Option B:** Display the weight for each class one under the other until a threshold number whereby a new column begins alongside the others.

**Option B** seems like the wiser option, as it will allow users to understand the model at a glance without having to examine each arc one by one.

#### 4.4.4 Overview of new functionality

The method of editing an arc weight will remain the same with the only difference being the “Edit arc weight” window must have multiple input fields for each token class. Furthermore, the labels of the arc weights must be altered to display each token class’ arc weight.

#### 4.4.5 New Architecture

Instead of the weight of an arc being a simple integer, it now consists of a collection of integers, each of a different token class. We would like to distinguish between the arc weights of different token classes and hence colour each label in a different colour. This was accomplished using the same technique as with the places. The arc weight was changed from an `int` to a `LinkedList` of objects of type `Marking`. For each entry in this linked list there is also an entry in the `NameLabel` linked list used to display the coloured label on the GUI. The `setWeightLabelPosition()` in class `Arc` was also heavily modified to display the arc weights in a tabular fashion. The undo/redo features were implemented in an almost identical manner to the place marking undo/redo feature detailed in Section 4.3.5.

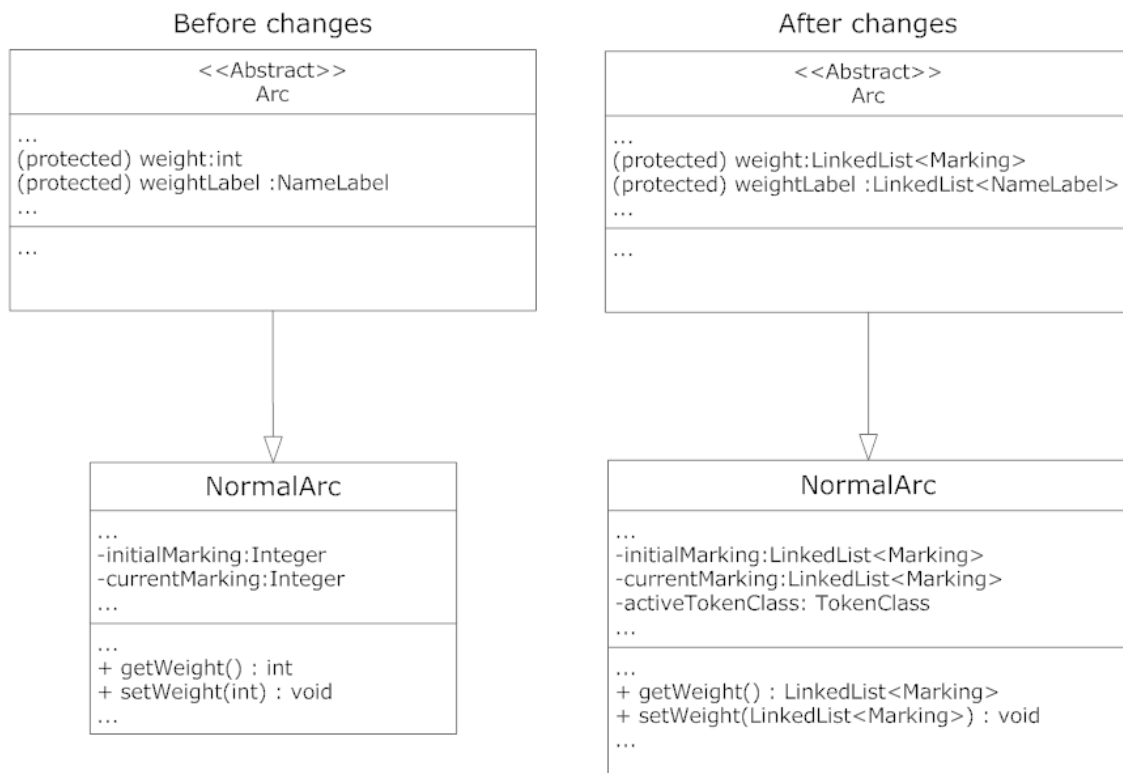


Figure 4.20: Main changes to classes `Arc` and `NormalArc`

The `EditWeightAction` was heavily modified to display a new window allowing the user to enter multiple arc weights.

#### 4.4.6 Final result

The user can edit the arc weight for each token class by right clicking on an arc and choosing the “Edit arc weight” option (Figure 4.21).

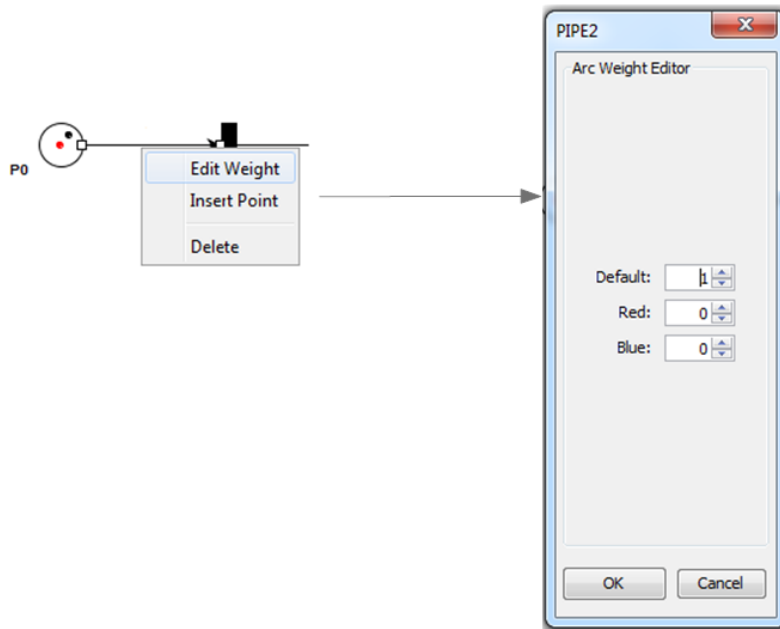


Figure 4.21: New Edit Arc Weight window

The label for each token class is displayed in the colour that the token class is set to.

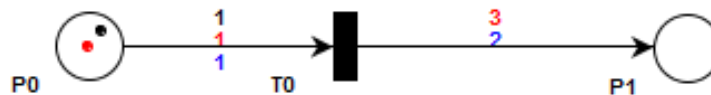


Figure 4.22: Multiple arc weights

## 4.5 Extending the PNML interchange format to allow saving/loading of CGSPNs

### 4.5.1 Introduction

As explained in the background section, PIPE currently uses the standard PNML format to save/load Petri nets. We would like to add the ability to save/load CGSPNs. Unfortunately, our research shows that there does not seem to be a standard PNML format

for specifying CGSPNs. Furthermore, the tools that do support CGSPNs have implemented their save/load features in a unique format and do not conform to PNML. For these reasons we extend the current PNML format to support CGSPNs.

### 4.5.2 Existing Functionality

A user can save the Petri net that is currently open in a tab by clicking on File – Save/Save As or on the save button on the button toolbar. A Petri net can be loaded in the same manner. The resulting PNML file follows the structure outlined in Section 2.4.4.

### 4.5.3 Existing Architecture

#### Saving

The user interacts with the GUI (and hence the `GUIFrame` class) to save a net. Once the button is pressed `GUIFrame`'s `saveNet` is called which in turn creates a `DataLayerWriter` object and calls `savePNML` on it. `DataLayerWriter` has all the necessary methods to save a Petri net. It takes in a `DataLayer` object and transforms all its transitions, arcs and places into XML elements. The `savePNML` method takes a File as input to which it will write each XML element. The result is a PNML representation of the Petri net saved as a .xml file.

#### Loading

Method `transformPNML()` in class `PNMLTransformer` takes as input the file name of an XML file. It then parses and transforms the file, and generates a `Document` object. Method `createFromPNML()` in class `DataLayer` takes the document object from above and calls the `addPetriNetObject()` method for each element. That in turn adds the loaded element into the model and onto the GUI.

### 4.5.4 Design decisions and challenges

- We must somehow extend the PNML to support CGSPNs; however we would like the new definition to be completely backward compatible.
  - This can be accomplished by adding new XML elements (outlined in Section 4.5.5) for anything that did not exist before but keeping the specification of pre-existing elements the same.
  - In particular we must create a new type of element for defining token classes. Such an element should define the token class name/id, whether or not its enabled/disabled and its red green and blue values (collectively defining its colour).
  - Old style PNML formats can still be loaded and non-CGSPNs should be saved in the old format.

- We must adapt the specification of the marking of a place as well as the arc weight of an arc. Previously the PNML declared both of these as integer values; however now each of these can have several values from different types of token classes. We have two options for this:
  - **Option A:** Define a new PNML element called **Marking**. This would associate a previously defined **TokenClass** id with a number denoting the marking of that token class on a place (and likewise for the weight of an arc). We could then define an XML list of such markings for each place.
  - **Option B:** If a place is marked with 1 red token and 2 blue tokens we could define the marking as `Red,1,Blue,2`.

Although **Option A** seems like a more elegant solution it would overload the PNML file with extra elements for each combination of token classes with places and arcs. One apparent disadvantage of **Option B** is that the commas make it seem unstructured; however, the implementation of **Option A** would also involve XML lists of **Marking** elements. The only difference between our lists and those defined by the XML schema are that we use commas instead of spaces to separate elements of the list. Furthermore, **Option B** is more human readable allowing the user to immediately understand how that particular place is marked whereas **Option A** would force the user to track down **Marking** elements elsewhere in the file one by one to finally decipher the place marking. Finally, to allow backward compatibility using **Option A** we would have ended up defining completely different methods for dealing with old PNML files and new PNML files which would have made the code messy. For these reasons we have decided that **Option B** seems like the best choice for defining place markings and arc weights.

#### 4.5.5 New Architecture

##### Changes to support Saving new PNML format

The `DataLayerWriter` class had to be modified in the following ways:

- `createTokenClassElement(TokenClass inputTokenClass, Document document)`: This method takes in a token class and a PNML document. It was created to transform token classes into XML elements and then append them onto the document.
- `createPlaceElement(Place inputPlace, Document document)`: This method transforms a place into an XML element. It was modified to allow for the new comma separated format we have introduced to model the marking of a place.
- `createArcElement(Arc inputArc, Document document)`: This method transforms an arc into an XML element. It was modified to allow for the new comma separated format we have introduced to model the arc weight of an arc.
- `savePNML`: We extend this method to iterate through all token classes in the model and add them to the PNML by calling on `createTokenClassElement` multiple times.

We also added a new member `isCGSPN` that checks to see if more than one token class is enabled in this Petri net. If only one token class is enabled then this Petri net can be



treated as a GSPN and should be saved that way to retain backward compatibility. All methods above will only use our new PNML definition if `isCGSPN` is true.

### Changes to support Loading new PNML format

To allow for loading we tracked down the `createFromPNML()` method in class `DataLayer`. We added a new clause so that if the element passed in is of type `TokenClass`, the private method `addTokenClass()` is called. That method in turn adds the new token class to the model. Finally, methods responsible for transforming XML elements into actual objects in the model and on the grid had to be modified. These methods were `createArc()` and `createPlace()` which were modified to handle the new comma separated format. For example if the PNML had "Red,1" for the place marking, a new instance of `Marking` was created with the token class set to Red and the marking set to 1. Note that at this point a token class of type Red should already have been declared earlier in the PNML and hence now we can reference our new `Marking` to that token class (by searching through token classes by name). As there may be multiple markings for a place, a loop is used to add each such entry into a `LinkedList<Marking>` which was stored as the marking of the place or the arc weight of the arc respectively.

#### 4.5.6 Final result

The mechanisms for saving/loading a CGSPN are no different to those that were previously used to save/load a GSPN. However, the actual PNML differs in the following ways:

- Token classes are declared as separate elements where their name, colour and whether or not they are enabled appear as attributes.

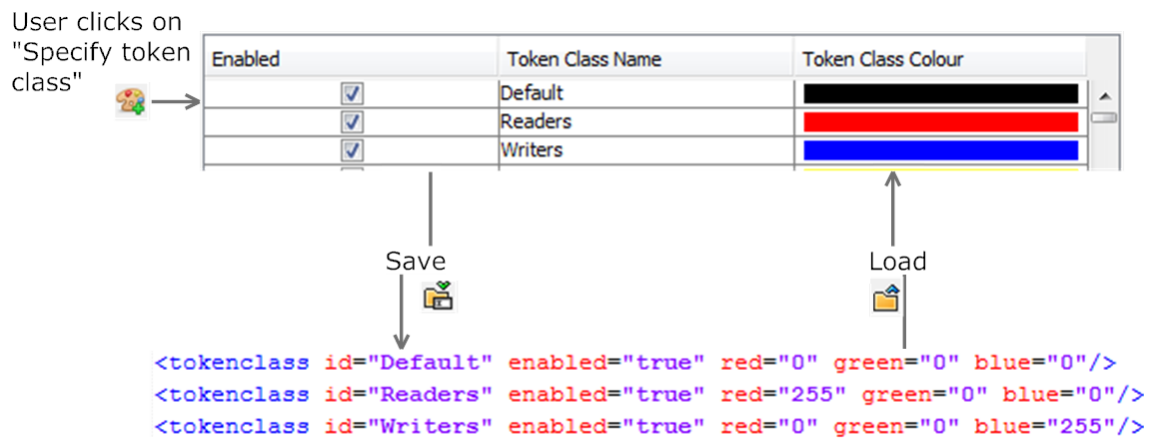


Figure 4.23: PNML Definition of token classes in readers/writers problem

- Token classes are displayed alongside arc weights and places separated by a comma.

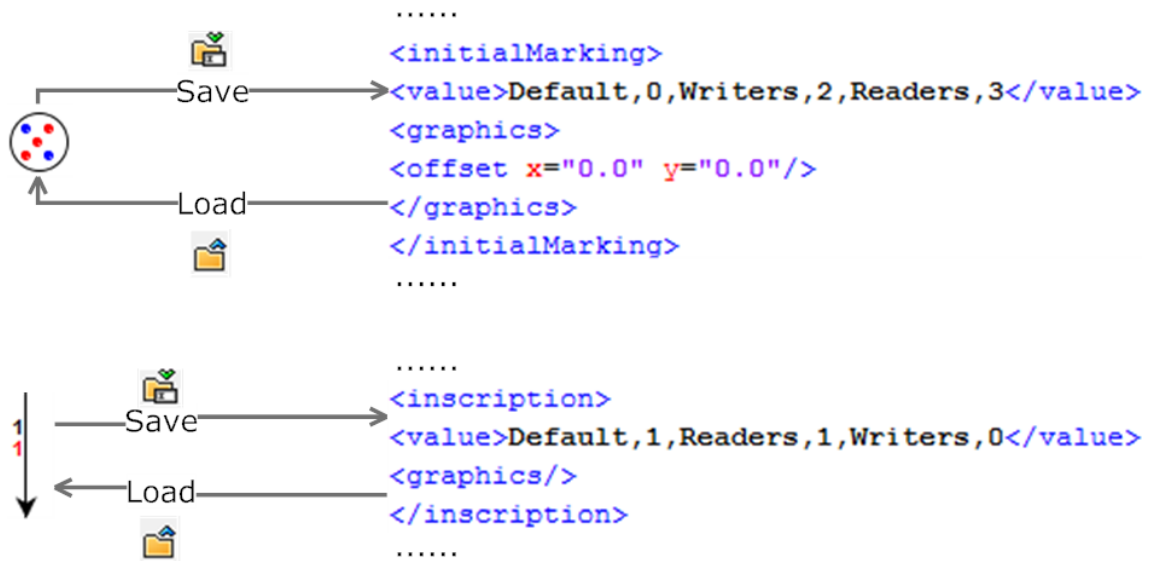


Figure 4.24: PNML Definition of arcs and places

Please note that in the screenshots above, only selected parts of the PNML file are shown as we are only interested in the changes made for CGSPNs. It should also be noted that if only one token class exists in the Petri net model then it is saved using the old PNML format to ensure backward compatibility.

## Chapter 5

# Supporting the animation of CGSPNs

### 5.1 Introduction

PIPE offers an “animation mode” that allows users to animate the firing of transitions and change the state of a Petri net. This allows users to simulate a chosen model and watch it changing states in real time. Currently, a user can:

- Manually fire enabled transitions.
- Randomly fire enabled transitions.
- Step back and forth through previously fired transitions.
- Randomly fire a specified number of transitions, each within a specified time interval.

We would like to allow for the same functions to work with CGSPNs.

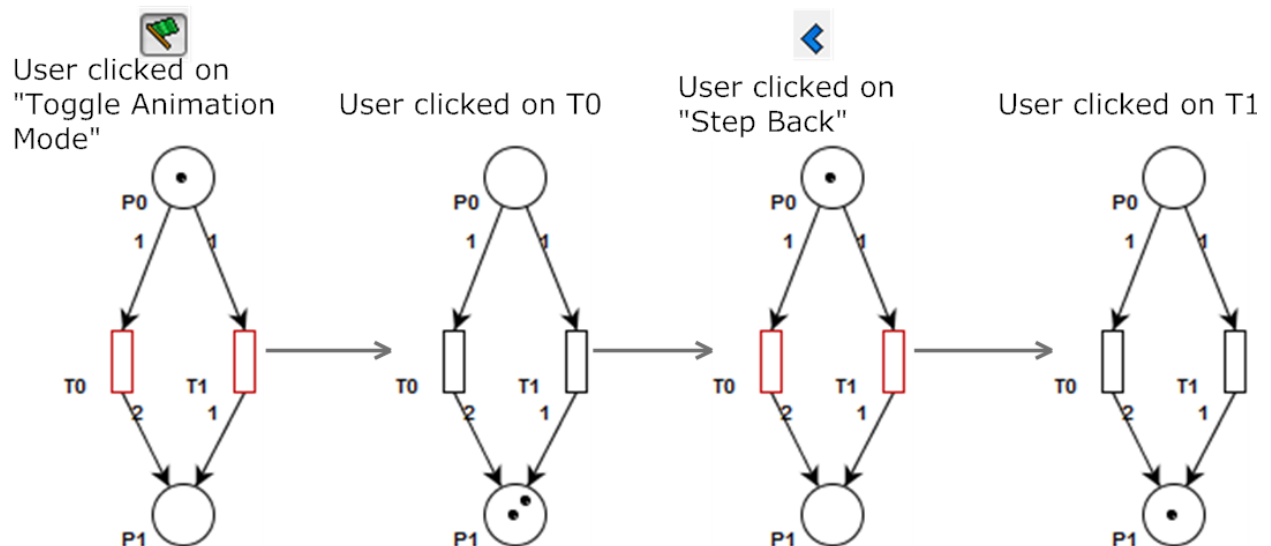


Figure 5.1: An example of the animation of a GSPN

## 5.2 Existing Architecture

As described in the Background chapter, an integral part of any Petri net are its backwards and forwards incidence functions. They are responsible for specifying the relationship and connections between transitions and places. The forwards incidence function between a place  $p$  and a transition  $t$  defines the weight of the arc between them and hence the number of tokens created on place  $p$  once transition  $t$  is fired. If the incidence function is  $\leq 0$  then such an arc does not exist. The backwards incidence function follows the same rules only that it defines the number of tokens destroyed on place  $p$  once transition  $t$  is fired. It is clear that these are the vital calculations that need to occur for the animator to know what will occur once a transition is fired. In PIPE, the backwards and forwards incidence functions are defined as matrices that specify the backwards and forwards incidence functions for every place in the model. Since the matrices are defined for the whole model these functions are found in the `DataLayer` class. Furthermore, the `DataLayer` also contains methods that make use of these matrices to calculate which transitions are enabled and to enable them when necessary. Overall the most important functions which the `DataLayer` has pertaining to animation of Petri nets are:

- `createMatrixes`: Simply calls all the functions required to calculate all incidence matrices for the Petri net.
- `createForwardIncidenceMatrix`: Calculates the forward incidence matrix of the Petri net.
- `createBackwardIncidenceMatrix`: Calculates the backward incidence matrix for the Petri net.
- `createIncidenceMatrix`: Calculates the incidence matrix by subtracting the forward incidence matrix from the backward incidence matrix. In this matrix lies an entry for each place and transition pair which specifies the tokens to be added to that specific place if that specific transition is fired.
- `createInitialMarkingVector`: Calculates a vector of initial markings for each place in the Petri net.
- `createCurrentMarkingVector`: Calculates a vector of current markings for each place in the Petri net. These will be modified each time a transition fires.
- `createCapacityVector`: Calculates a vector of place capacities in the Petri net. These will ensure that the firing of a transition does not cause a marking of a place that exceeds its capacity.
- `getRandomTransition`: Chooses an enabled transition of the Petri net at random.
- `fireTransition`: Fires a specified (enabled) transition. Will use the incidence matrix to calculate the new marking of the affected places. Also makes a call to `setMatrixChanged`.
- `getTransitionEnabledStatusArray`, `getTransitionEnabledStatus` and `setEnabledTransitions` : Collectively calculate whether a transition is enabled given a specific marking.
- `restoreState`: Restores Petri net to previous stored marking.

- **setMatrixChanged**: Is called any time a firing occurs so that the incidence matrices are recalculated.

Each of the aforementioned matrices are of type `PNMatrix` which holds a two dimensional int array and functions for common calculations made on the incidence matrices.

Finally, PIPE has a class `Animator` that is responsible for calling these functions on the model during the simulation of a Petri net.

### 5.3 Design decisions and challenges

- The current implementation only calculates the incidence matrices based on one token class (the default black one). We had to come up with a way to elegantly calculate the incidence matrices for all token classes.
  - After carefully going through the code, the best way to accomplish this was to shift methods and members pertaining to the calculation of incidence matrices to the class `TokenClass`. This makes sense because each token class has its own marking in each place and its own arc weight going to or from that place.

### 5.4 Overview of new functionality

Each token class will have its own incidence matrices that define the effect of firing a transition. No visible changes will be made to the GUI of PIPE but the changes explained below will allow the correct animation of a CGSPN.

### 5.5 New Architecture

Deletions from DataLayer

DataLayer
...
+incidenceMatrix :PNMatrix = null
+forwardsIncidenceMatrix :PNMatrix = null
+backwardsIncidenceMatrix :PNMatrix = null
...
...
+createForwardIncidenceMatrix() : void
+createBackwardsIncidenceMatrix() : void
+createIncidenceMatrix() : void
+getIncidenceMatrix() : int[][]
+setIncidenceMatrix(PNMatrix matrix) : void
+getForwardsIncidenceMatrix() : int[][]
+setForwardsIncidenceMatrix(PNMatrix) : void
+getBackwardsIncidenceMatrix() : int[][]
+setBackwardsIncidenceMatrix(PNMatrix) : void
...

Additions to TokenClass

TokenClass
...
+incidenceMatrix :PNMatrix = null
+forwardsIncidenceMatrix :PNMatrix = null
+backwardsIncidenceMatrix :PNMatrix = null
...
...
+createForwardIncidenceMatrix(ArrayList<Arc> arcsArray, ArrayList<Transition> transitionsArray, ArrayList<Place> placesArray) : void
'
+createBackwardsIncidenceMatrix(ArrayList<Arc> arcsArray, ArrayList<Transition> transitionsArray, ArrayList<Place> placesArray) : void
'
+createIncidenceMatrix(ArrayList<Arc> arcsArray, ArrayList<Transition> transitionsArray, ArrayList<Place> placesArray) : void
'
+getIncidenceMatrix() : int[][]
+setIncidenceMatrix(PNMatrix matrix) : void
+getForwardsIncidenceMatrix() : int[][]
+setForwardsIncidenceMatrix(PNMatrix) : void
+getBackwardsIncidenceMatrix() : int[][]
+setBackwardsIncidenceMatrix(PNMatrix) : void
...

Figure 5.2: Functions and members moved from DataLayer to TokenClass

Having shifted the functions shown in Figure 5.2 all the other related methods/classes (listed in Section 5.2) were modified to iterate through each token class and call the incidence functions of each token class. Since these functions need to iterate through all arcs, transitions and places in the model we also had to pass these to the `tokenClass` (hence the extra parameters in Figure 5.2). Another vital change that had to take place was to change the `currentMarkingVector` and `initialMarkingVector` from an `int[]` to a `LinkedList<Marking>[]` since that is how we now represent markings (Section 4.3). Each item in the array is a linked list containing the various markings from all token classes for that specific place. Finally, the calculation of the new incidence matrices after a transition is fired, is now done for each token class that is involved in the firing. No changes had to be made to the `Animator` class as that simply controls the execution of the methods we have altered in this section.

## 5.6 Final result

Whenever a transition is fired, PIPE now uses the arc weights for each individual token class to determine how many tokens of each token class should be destroyed/created. As shown in Figure 5.3, the animation is now fully functional with CGSPNs. Users can manually/randomly fire transitions or choose to fire a specified number of transitions at a specified time interval. They are also able to “Step back” and “Step forward” as before.

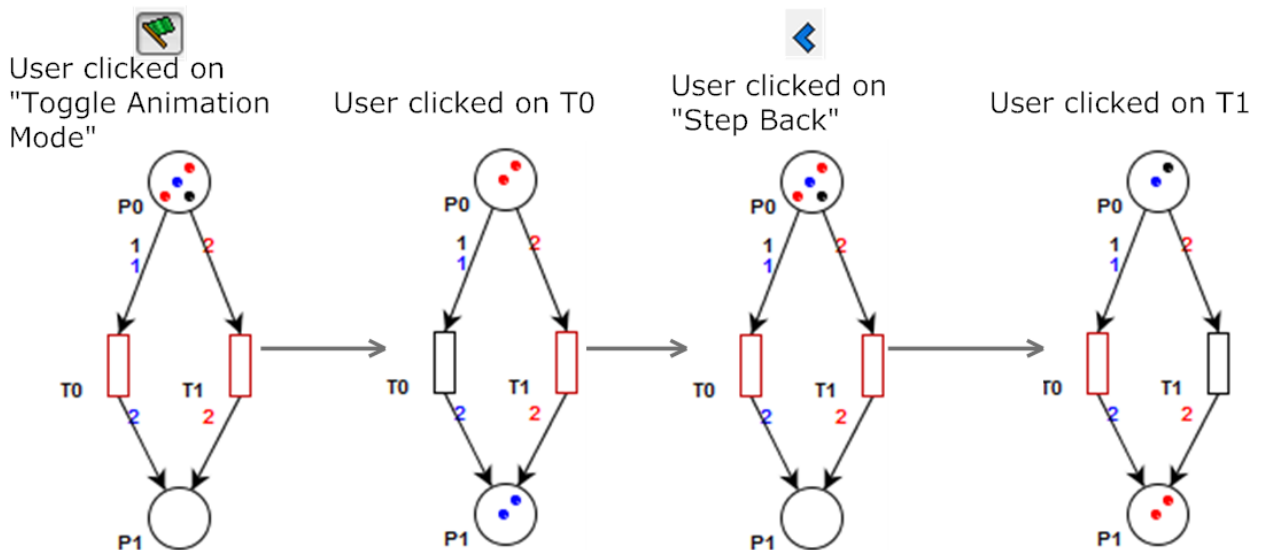


Figure 5.3: An example of the animation of a CGSPN

We have implemented a way for users to simulate CGSPNs in PIPE; however, we could do better. In the following chapter we analyse why our current representation is not ideal and seek new methods for representing CGSPNs that will make our CGSPN models more compact.

## Chapter 6

# A Novel Graphical Representation for CGSPNs

### 6.1 Motivation

We have now implemented all that is necessary to design and animate CGSPNs; however, the result is not as elegant as we had hoped. Let us analyse the example given by [3]. We would like to analyse the Petri net model of a dual processor where both processors access a common bus, but not simultaneously.

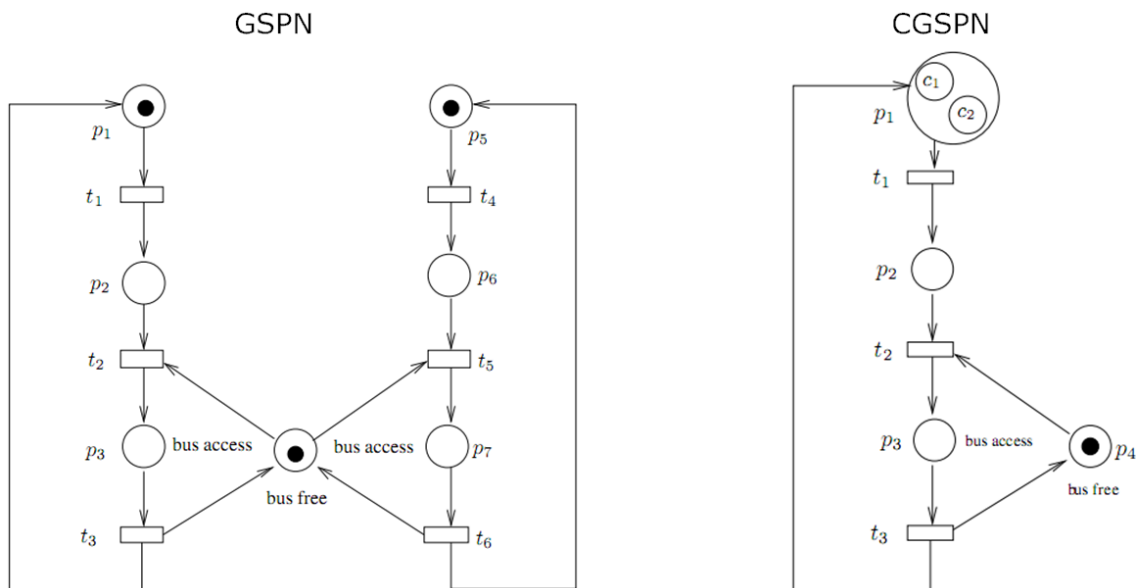


Figure 6.1: The Dual Processor problem as a GSPN and a CGSPN as given by [3]

Using our current implementation, the result in PIPE of the coloured version of the dual processor model is shown in Figure 6.2.

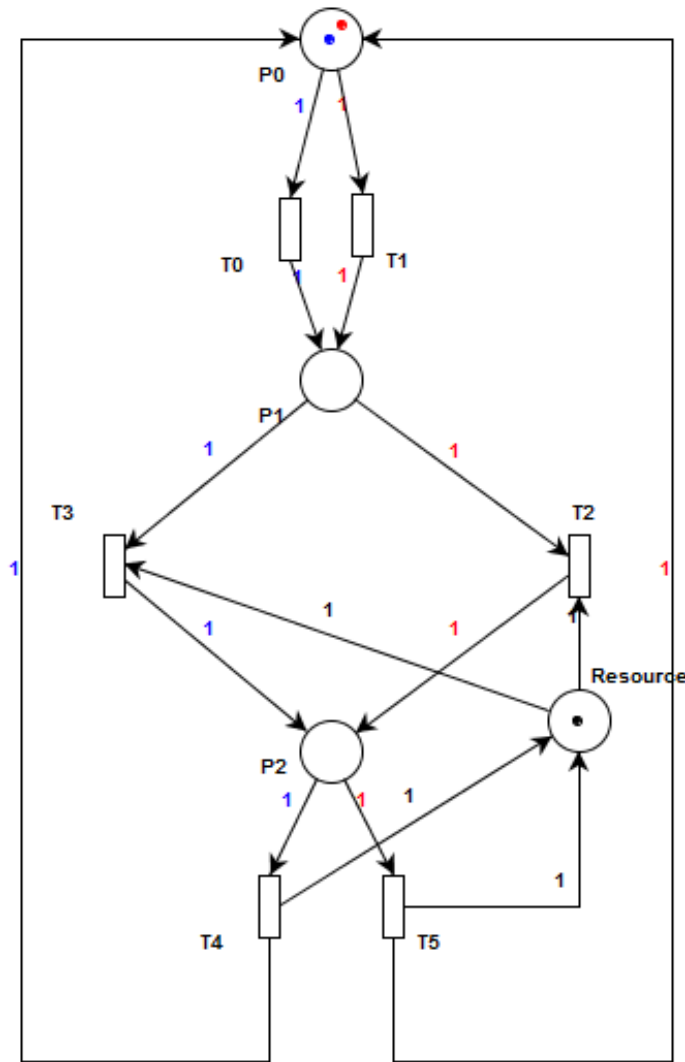


Figure 6.2: The Dual Processor problem modelled using our current implementation

It is obvious that our version seems much more complex than that in Figure 6.1. The reason for this is that Bause and Kritzinger have given the most compact form of the CPN but have chosen to omit underlying details of this model. In reality, each transition in Figure 6.1 has two firing modes. One which fires a token class of type A and one that fires one of type B. These “firing modes” are present in our model as separate transitions and their corresponding arcs. For example the two firing modes of the first transition in Bause and Kritzinger’s model are represented in our model by transitions T0 and T1. T0 fires a red token corresponding to a token of type A in the other model and likewise, T1 fires a blue token representing a token of type B. These firing modes are not graphically represented anywhere in the Bause and Kritzinger model but are simply inferred.

Although we have successfully managed to compact the places in our Petri net (from 7 places to 4 places), we have failed to compact the transitions and arcs. We would like a way to make our model look more like that in Figure 6.1. That can be accomplished by somehow grouping the transitions so that a single transition can represent multiple firing modes.



## 6.2 Design considerations and challenges

We considered two possible methods for modelling our CGSPNs in a more compact way. Each method is presented as a subsection below and analysed to find the best solution.

### 6.2.1 Arc expressions

This is the standard method of defining firing modes in CPNs and is the way in which the few tools that offer support for CPNs do so (e.g. CPN Tools, Section 10.3). We could change the way in which arc weights work in PIPE so that instead of arc weights being defined by integers they are actually defined by expressions. For example an arc weight could be defined as  $x + y$ . This is like saying that you would like to fire 1 token of type  $x$  and 1 of type  $y$ . Multiple firing modes are defined by binding each variable ( $x$  and  $y$ ) to each available token class.

#### Advantages

- Where appropriate, multiple transitions and arcs can be compacted together

#### Disadvantages

- Such an implementation would require the complete redesign of the way in which arc weights are defined and presented in PIPE. This would make the task of designing a simple Petri net unnecessarily more complex.
- Is significantly less intuitive than the integer representation of arc weights where users can graphically see which tokens will be fired.
- Referring to Figure 6.3,  $x$  and  $y$  can each be bound to token classes  $a, b$  and  $c$  respectively. If the user requires a different number of tokens to be fired depending on what token classes  $x$  and  $y$  are eventually bound to then he would have to declare a new transition. This requires an extra transition and defeats the purpose of the compacting procedure. This could possibly be overcome by defining a function instead of a simple expression but this would further complicate matters.
- The popup for the selection of a firing mode is not an elegant solution as the user must examine options presented to him/her on demand. It would be easier on the user if these were just available graphically.

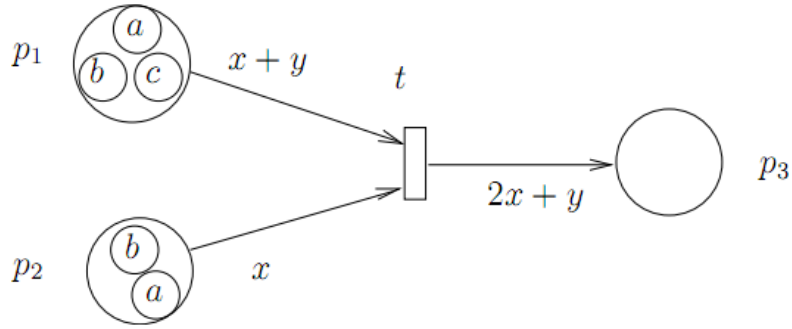


Figure 6.3: A CPN with arc expressions taken from [3]

### 6.2.2 Grouped Transitions

Due to the deficiencies of the “Arc expressions” method explained above, we sought to come up with an alternative way of handling multiple firing modes. One of the goals of the proposed method is to keep the design of Petri nets and the current representation of arc weights intact. The method has the following properties:

- Each firing mode is defined as a separate transition with its corresponding arcs. The user can very easily edit the firing mode by simply editing the arc weights as was always the case in PIPE.
- The user can select any number of transitions that have identical input and output places and choose to “Group” them. The transitions will all be replaced by a single object called a Group Transition. This object must somehow look different so it may be distinguished from normal transitions.
- The user can select any Group Transition and choose to “Ungroup” it. The Group Transition will be replaced by all the transitions it was made of.
- Under animation mode any Group Transitions that have enabled Transitions in them are highlighted. If a user wants to choose a firing mode for the Group Transition then he/she must click on it. The Group Transition “Ungroups” and all the original transitions (and hence all the possible firing modes) are displayed to the user. The user clicks on any of the enabled firing modes to fire the transition.
- After firing a transition the Group Transition is grouped once again.
- The arcs connected to a Group Transition should not have arc weights as these arcs represent a collection of underlying arcs, each with their own arc weights.
- A Group Transition cannot be deleted but can always be ungrouped. If a user wants to delete firing modes from it then he/she can ungroup the Group Transition and then delete any of the transitions.

#### Advantages

- Where appropriate, multiple transitions and arcs can be compacted together.

- This method does not alter the intuitive way in which simple Petri nets are designed.
- The way in which firing modes are presented to a user upon firing a transition is elegant and non-intrusive.
- By keeping the representation of arc weights as numbers rather than expressions, a CPN is as easy to understand as a simple Petri net.

### Disadvantages

- To define each firing mode the user must create a separate transition and its corresponding arcs. Although this is the most basic and intuitive way to define a firing mode, some may argue that it is more time consuming to design.

## 6.3 Implementation and Architecture

### Class GroupTransition

Any object that is to be displayed on the grid is considered a `PetriNetObject` and should extend this class. Objects that also have labels, must extend the `PlaceTransitionObject` class which is an extension of the former (Figure 6.5). For this reason `GroupTransition` was included in the `DataLayer` package and extends a `PlaceTransitionObject` object. A `GroupTransition` differs from other `PlaceTransitionObject` objects. Although it is displayed on the grid, it is not part of the state of the Petri net and hence is not referenced in the `DataLayer` class. This is because the actual transitions within a group transition are those that define the Petri net model. A group transition is essentially a visual enhancement used to simplify the look of the Petri net model. The `GroupTransition` was modelled after the `Transition` class due to the similarities between them (particularly the fact that they can be enabled and hence highlighted). Some of the major differences between the `Transition` class and the new `GroupTransition` class are displayed in Figure 6.4.

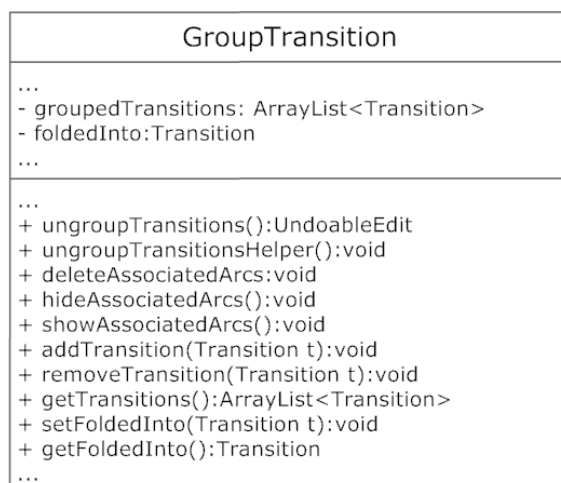


Figure 6.4: Unique functions in GroupTransition

Two very important members of `GroupTransition` are:

- `ArrayList<Transition> groupedTransitions`: This is a collection of all the transitions that this group contains. Note that `ArrayList` was chosen as we will not be adding/deleting transitions to this array and hence the `ArrayList` offers the fastest access.
- `Transition foldedInto`: A `GroupTransition` folds a number of transitions into one. This attribute is the transition that all other transitions have folded into. It was included as a member to help with undo/redo as explained in Section 6.3.

The `unfoldTransitions()` is responsible for disassociating the transitions with this group and restoring them on the grid.

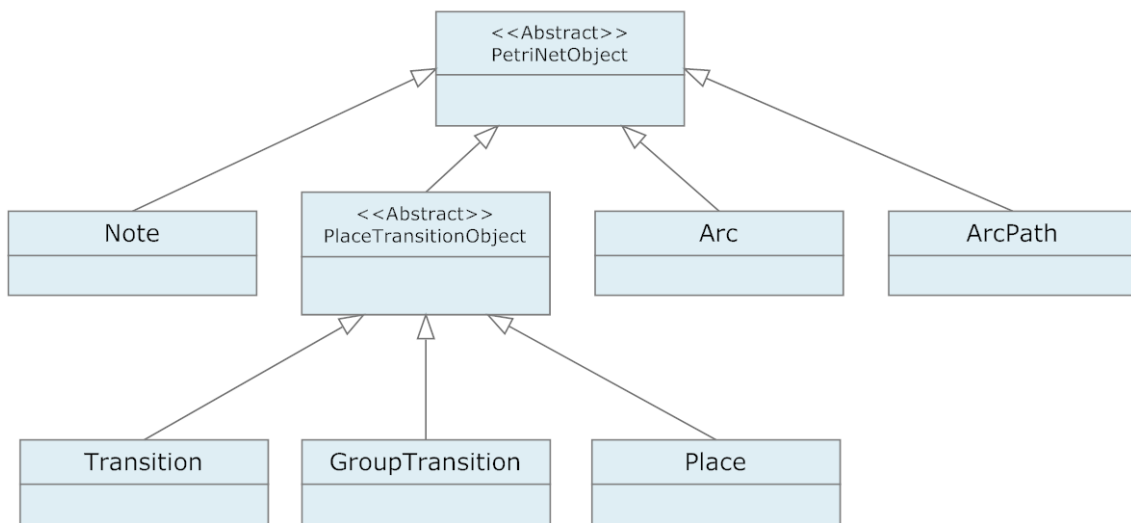


Figure 6.5: All objects extending `PetriNetObject` can be displayed on the grid

## Transition

Major changes also had to be made to the `Transition` class.

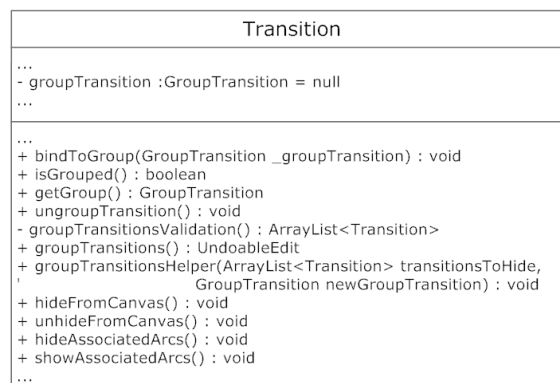


Figure 6.6: New functions in `Transition`

If a transition is enabled then it must also enable the group transition it belongs to, hence the member `groupTransition` was added to the class. The `groupTransitionsValidation()` method ensures that only transitions with the same input and output place can be grouped together. Finally, the `groupTransitions()` method creates a new `GroupTransition` that contains all transitions currently selected on the canvas.

### GroupTransitionHandler

A handler was also created in the package `pipe.gui.handler` to handle interactions with the object on the GUI; more importantly to allow the user to select the option of ungrouping it. The handler also allows the user to select the “Edit Transition” option which instantiates a `GroupTransitionEditorPanel` object.

### GroupTransitionEditorPanel

This class is responsible for displaying a window where the user can alter the name of the `GroupTransition` and rotate it a selected amount of degrees.

### GuiFrame and GuiView

These classes were modified to add two buttons on the toolbar. One button is the Group All function which iterates through all transitions in the Petri net and automatically groups transitions together. This is done by examining all transitions of the Petri net, identifying those that have common parents and children and grouping them together. The Ungroup All button will ungroup any `GroupTransitions` on the grid using a similar technique. Shortcut keys were also added for these two buttons (*ctrl + shift + g = Group*, *ctrl + shift + h = Un-group*).

### AnimationHandler

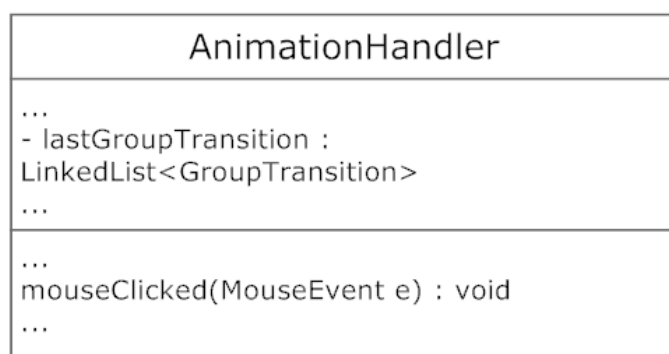


Figure 6.7: Properties modified in class `AnimationHandler`

The animation handler class had to be altered so that when in animation mode and a group transition is clicked on, the group transition is temporarily un-grouped. Before

being un-grouped the group transition is saved in the lastGroupTransition field. Once the user fires a transition, the transitions are grouped back using the group transition stored in lastGroupTransition.

## Undo/Redo Features

- **GroupTransitionEdit:** The GroupTransition is saved. The undo calls a method in GroupTransition for ungrouping itself. The redo collects all the transitions from the old GroupTransition and calls upon them to regroup.
- **GroupTransitionRotationEdit:** Saves the old angle. If undo() is called the object is rotated by +angle degrees and if redo() is called, it is rotated by -angle.
- **UngroupTransitionEdit:** The GroupTransition is saved. The undo() collects all the transitions from the old GroupTransition and calls upon them to regroup. The redo calls a method in GroupTransition for ungrouping itself.

## 6.4 Final Result

### 6.4.1 Design

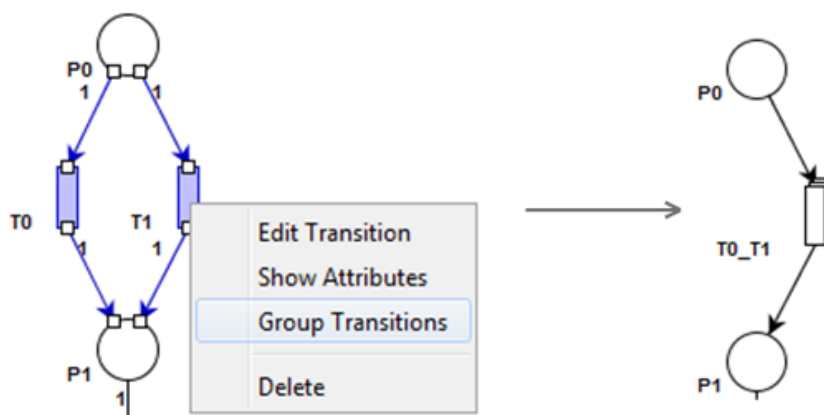


Figure 6.8: Grouping two transitions

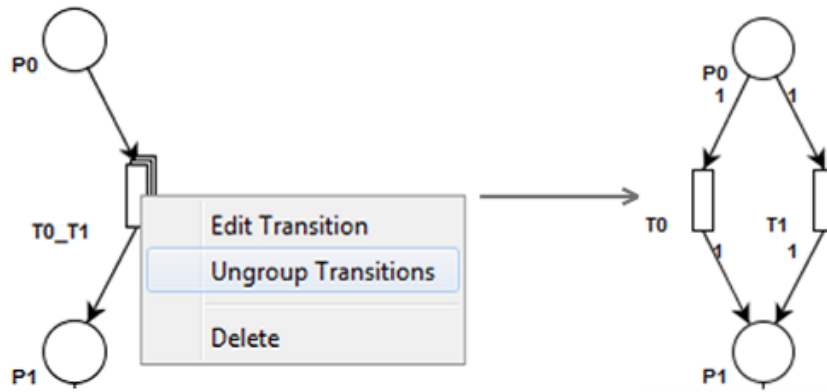


Figure 6.9: Un-grouping a group transition

Only transitions that have identical input and output places can be grouped as only those transitions can act as “firing modes” for one transition.

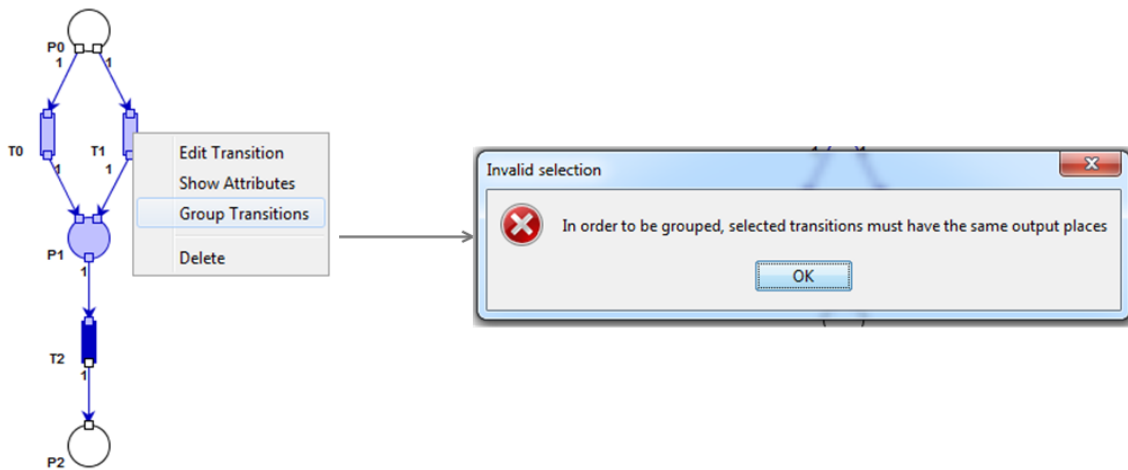


Figure 6.10: Validation error when grouping incompatible transitions

The Group and Ungroup buttons on the toolbar save the user some time and automatically group/ungroup any compatible transitions.

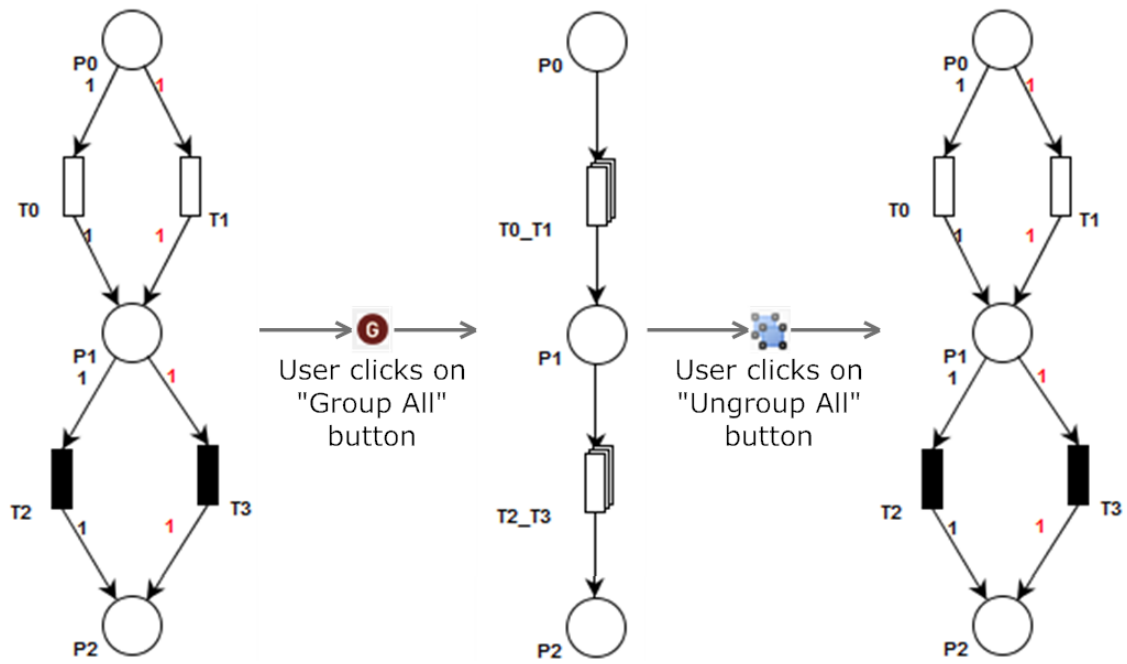


Figure 6.11: Group/Ungroup All functionality

We shall now try modelling the dual processor problem again using this new implementation.



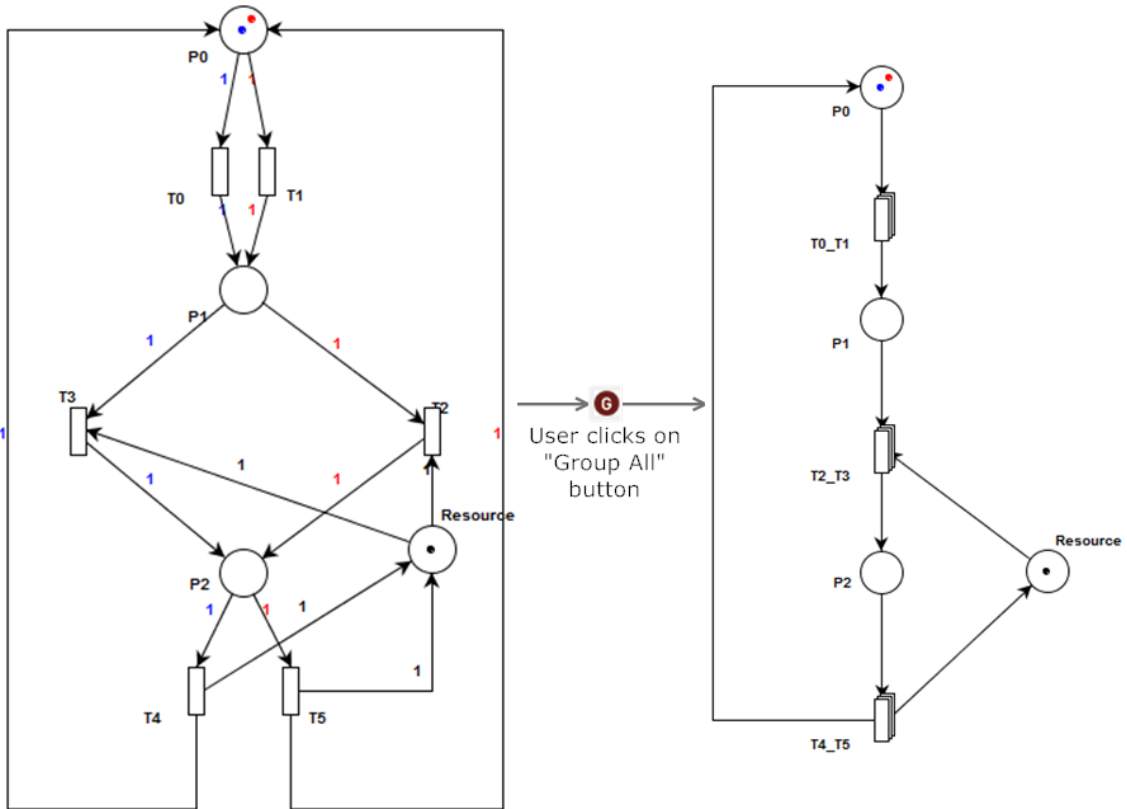


Figure 6.12: The result of grouping our previous model

The result is exactly what we hoped for as the CPN is modelled in a very elegant and compact way.

### 6.4.2 Animation

When switching to animation mode, the group transition is highlighted red when enabled.

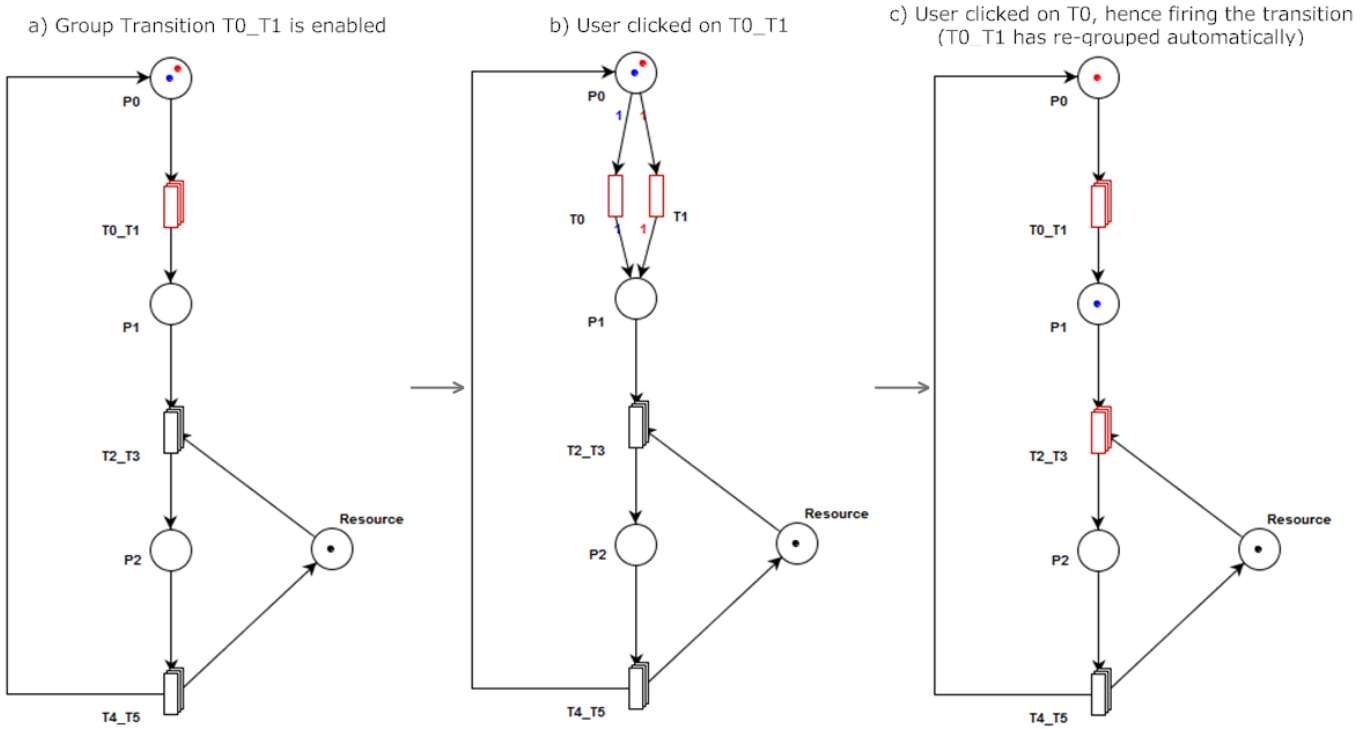


Figure 6.13: Animation using group transitions

In step (b) the user has chosen to fire group transition T0\_T1 and hence has clicked on it. The group transition then unfolds, displaying to the user all the available firing modes. In step (c) the user has chosen one of the enabled firing modes and the transition fires. The group transition is automatically re-grouped to retain the compact design of the CGSPN.

## Chapter 7

# Supporting the unfolding of CGSPNs

As explained in Section 2.2.4, unfolding refers to the transformation of a CGSPN to a GSPN. The resulting model has the same meaning as the folded version but is often much larger and more complicated when represented graphically.

### 7.1 Motivation

Having implemented CGSPNs in PIPE we are now able to model very complex models in a much more compact and elegant way. However, we have lost out on two very important features that were previously present in PIPE:

- Ability to export a Petri net to various other applications through the PNML standard
  - As we were forced to define our own extension to the PNML format for saving/loading CGSPNs, other tools will not be able to import models of this type. Even if there was a standard format, very few Petri net tools support CGSPNs and hence we would not be able to use their analysis features.
  - Note that we can still export GSPNs to other applications, since we have made sure that the exported PNML is equivalent to that before our changes were implemented.
- Ability to analyse a Petri net using all the pre-existing PIPE analysis modules
  - Although new analysis modules can be written to analyse CGSPNs in PIPE it would make a lot more sense if any pre-existing modules can somehow be made to work with CGSPNs.

By adding the ability to unfold a CGSPN to a GSPN, a user can analyse any Petri net exactly as was done before. The user can also export his/her CGSPN model (transformed to a GSPN) to any other Petri net tool that supports the standard PNML format.

## 7.2 Design considerations and challenges

- The unfolded Petri net will expand existing places into several new places. How should these places be named?
  - An appropriate convention would be to name the new places as: *OldName-TokenClass*. For example, suppose two token classes exist named *Red* and *Blue* and one place named *P0* exists, marked with a red and a blue token. The two new places would be called *P0\_Red* and *P0\_Blue*.
- Should the unfolded Petri net be available graphically?
  - Initially this did not seem necessary. The user will mostly use this for transforming a CGSPN to a GSPN so it can then be analysed by a module or another application. This process can be done in the background, creating a functionally correct Petri net that can be analysed without ever appearing on the GUI.

It was finally decided that the user would want to see the unfolded Petri net. When a Petri net is unfolded, the places present in the CGSPN will be expanded to several new places. These new places will automatically be given unique names. If the user cannot see the Petri net model then analysis modules will reference these new places by their assigned name but the user will not know what the module is talking about.

## 7.3 Implementation and architecture

### 7.3.1 Algorithm

1. Expand any transitions that have input/output arcs with arc weights of different token types.
2. For each transition  $t_i$  do:
  - For each transition  $t_j$  ( $t_i \neq t_j$ ), find any common input/output place,  $p$  they have. For each such place  $p$ :
    - (a) Create a new place for each type of token class that can reach those places.
    - (b) Name it using the following convention: *OldName-TokenClass*.
    - (c) Mark it with the marking that the old place had for that particular token class.
    - (d) Shift the arc on transition  $t_j$  so that it is now connected to the new place instead of place  $p$ .
    - (e) Mark place  $p$  for deletion.
3. Delete all places  $p$  that have been marked for deletion.

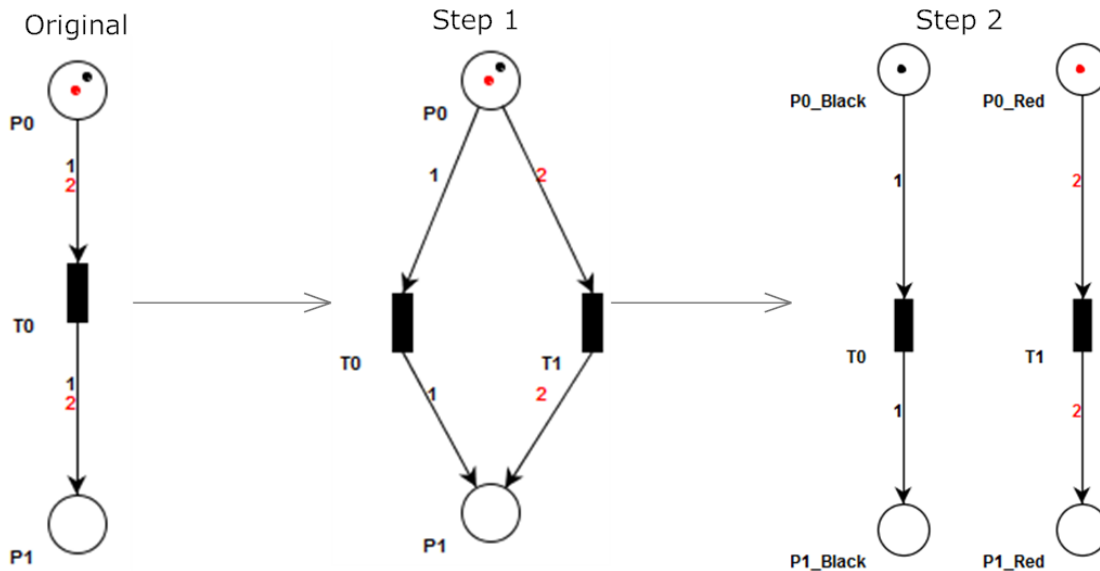


Figure 7.1: The algorithm step by step

### 7.3.2 Architecture

#### Class Unfolder

A class Unfolder was created:

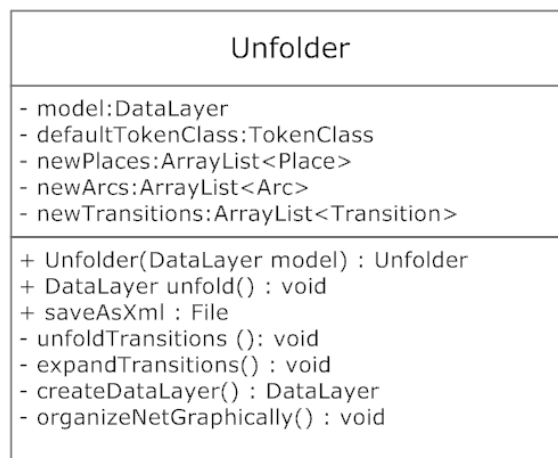


Figure 7.2: Class Unfolder

It was placed in the `DataLayer` package since it instantiates its own `DataLayer`.

The `unfolder` is instantiated by being passed an existing Petri net model (a `DataLayer` object). When the `unfold()` function is called the private methods presented in Figure 7.2 are called sequentially. The result is a new `DataLayer` object representing the unfolded Petri net. The option is then given to save the new net in PNML format by calling `saveAsXml`.

## GuiFrame and GuiView

These classes were modified to add a button “Unfold Net” on the toolbar. The net that is open in the current tab is passed to the unfolder and a new tab is generated with the unfolded net. The user can then analyse it or save it as he/she pleases.

## 7.4 Final Result

To unfold a CGSPN, the user must click on the “Unfold Net” button on the toolbar. The unfolded net will appear in a new tab. The places and transitions on the grid may not be optimally organized as the placement of elements on the GUI is done algorithmically. Below we show the result of the unfolder on a CGSPN of the readers and writers problem where multiple readers may access a resource simultaneously but a writer can only access a resource if nothing else is currently accessing it. Note that the resulting model was slightly manually rearranged to make it look neater.

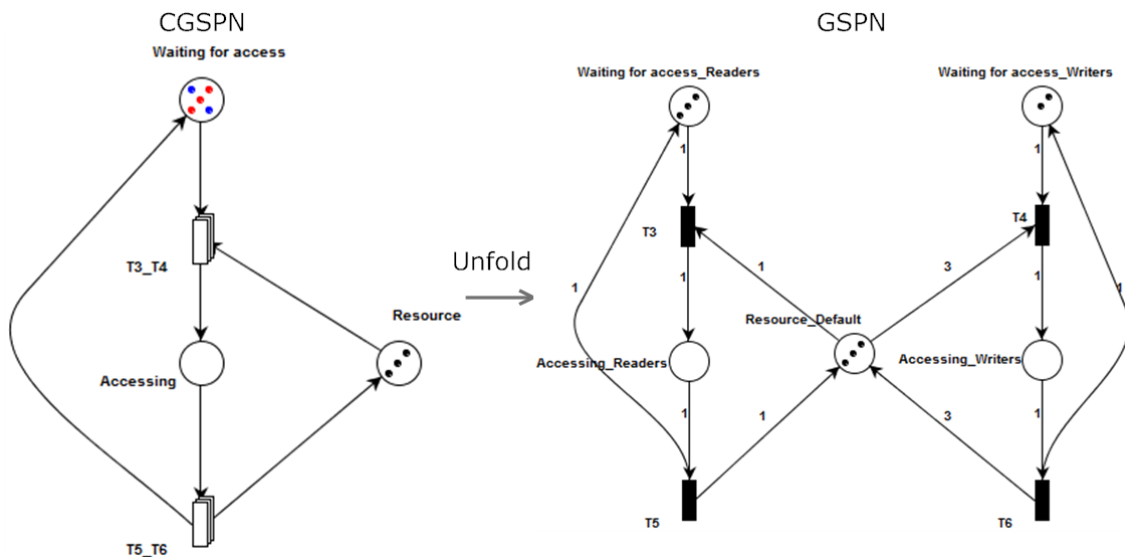


Figure 7.3: Unfolding of the readers/writers problem

## Chapter 8

# Supporting the analysis of CGSPNs

### 8.1 Introduction

One of the major reasons for implementing the Unfolder (Chapter 7) was to allow the existing analysis modules in PIPE to work with CGSPNs simply by first unfolding them to GSPNs.

### 8.2 Existing Functionality

The available modules are analysed in Section 2.4.4. Whilst working on a Petri net, a user is able to double click on any of the available modules. What happens next depends on the implementation of the `run()` method in each analysis module.

### 8.3 Existing Architecture

Currently, there is a `Module` interface which all modules must implement. The methods that must be implemented are the `getName()` and `run()` methods. The latter contains all the logic behind each analysis module.

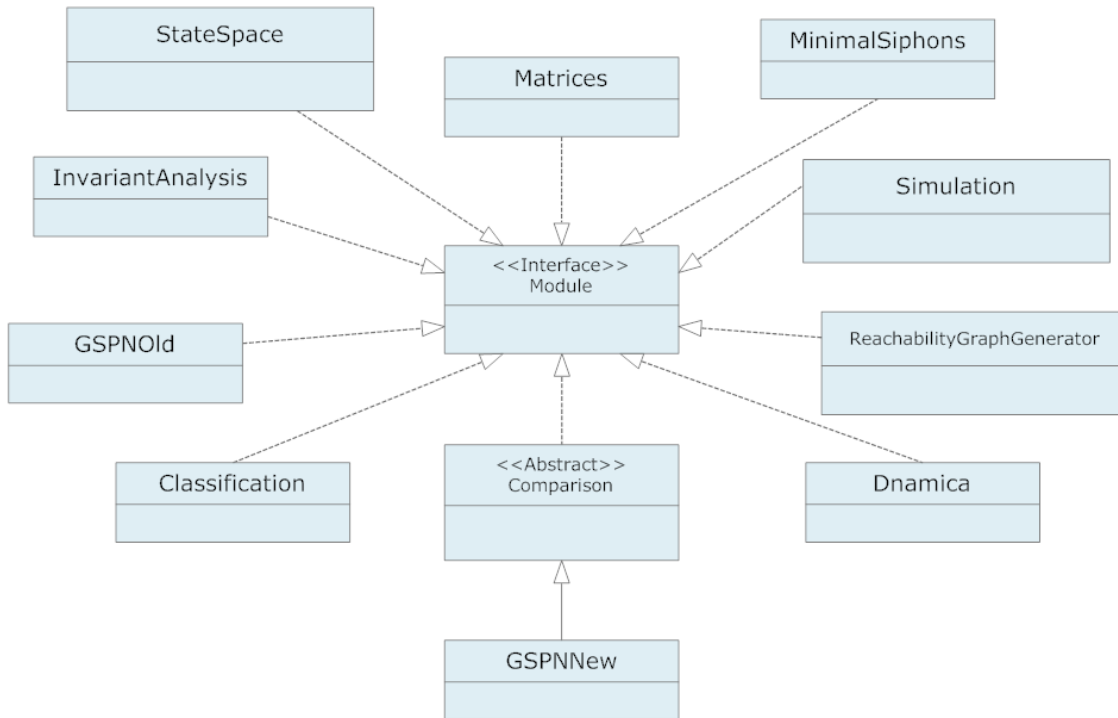


Figure 8.1: UML of Analysis modules

## 8.4 Design decisions and challenges

- Conceptually we would like to “plug-in” each module into a new element that lies between the model and the module. This new element would check to see if the model is compatible with the module, if not then the model is first unfolded into a GSPN and then passed to the module.
  - **Option A:** A very elegant way of implementing this would be to change the Module interface to an abstract class. This way an abstract method could be implemented that would check compatibility between the Petri net and the module. If incompatible, this method would go on and unfold the Petri net before allowing the module to analyse it.
  - **Option B:** The check for compatibility is done in each module and the `Unfolder` is called where necessary.

Although **Option A** is a much cleaner method we soon came to realise that converting the `Module` interface to an abstract class had one major flaw. Existing modules that extend other classes could not be made to extend our new abstract class due to the fact that Java does not allow a class to extend multiple classes. We could have modified the existing modules to no longer extend the classes they used to extend; however, it became apparent that robbing programmers of the option of extending their modules from various classes would limit their flexibility. Hence it was eventually decided to follow **Option B**.



## 8.5 New Architecture

Having already implemented the main enabler for analysis (the **Unfolder** described in Chapter 7) we now had to slightly modify the modules to take advantage of its functions. No change was made to the existing architecture of the modules; however, the code of each module had to be re-factored to be compatible with some of the changes we have made to PIPE. Such changes include:

- Change any calls made to the **DataLayer** requesting the incidence functions of a Petri net to be made to the individual token classes (of type **TokenClass**) instead as that functionality had been shifted there (Section 5.5).
- Change any calls made to the **DataLayer** requesting the marking of places to expect a **LinkedList<Marking>** instead of an **int []**.
- Change any calls made to the **DataLayer** requesting the weights of arcs to expect a **LinkedList<Marking>** instead of an **int []**.

## 8.6 Final Result

With many back-end changes in place, the user is now able to analyse CGSPNs by double clicking on the required analysis module. If the selected analysis module is CGSPN compatible then the analysis module is executed normally, otherwise the CGSPN is automatically unfolded and then analysed.

## Chapter 9

# Testing Framework

Throughout the project we carried out Test Driven Development (TDD). This involved creating tests after the design of a feature and before its implementation. When the tests were created they were expected to fail. The code was then written and constantly modified until the tests would succeed.

### 9.1 Motivation

Besides all the obvious reasons for creating automated tests during any software development process, the fact that we are dealing with an open source project makes it all the more important because:

- Unit tests often help a user understand somebody else's method/class and how it works
- It is unrealistic to assume that a user will never break existing functionality when implementing changes to the project. It is also unrealistic to assume that a user is able to check all aspects of the system manually after changes have been made to ensure he/she has not broken anything. System tests are a way for the user to check for such breaks.

Having entered a project with a very limited set of tests (which were imported from another version of PIPE), we did not have these benefits and quickly came to realize how this can repel potential contributors to the project. By implementing our own testing framework we would not only offer these advantages to the PIPE community but also would ultimately cut our own development time down considerably by detecting most breaks and bugs immediately throughout the development life cycle.

### 9.2 Design considerations and challenges

- A robust test framework should contain both unit and system tests. In a project with 204 existing classes it would be impossible to implement unit tests for each unit of functionality in our time constraints. For this reason we have defined a unit test

as a less granular test on the overall functionality of a collection of methods, such as that of “Saving/Loading”. Unfortunately this means that when such a test breaks the user may not be able to quickly detect the exact method that is at fault.

- Most of our new functionality has to do with adding new elements to the GUI, what is the best way to test this?
  - **Option A:** Allow the test framework to gain control of the mouse and keyboard to replicate all UI actions. This can be accomplished using the Abbot Java GUI Test Framework.
  - **Option B:** Make calls to the appropriate `actionPerformed` methods of the various classes to replicate UI actions.

**Option B** has the disadvantage of not checking the correctness of the toolbar buttons; however, such errors are extremely easy to debug manually as the user will notice something is wrong once he/she clicks the button. For each test in **Option A**, the test code is made harder to understand because the test code has to instruct the test framework where the mouse must move to execute a particular action. Additionally if the user moves the mouse while the test is running, the test fails and users may not realise why. Considering these disadvantages we choose **Option B**, which gives us more robust and easy to understand tests.

- How do you test animation mode? If you automatically calculate the results of each step in the animation won't you risk replicating bugs from the code into your tests?
  - This problem was overcome by loading known Petri nets for which we know what result each step in the animation should produce. After each animation step, the test checks the current state of the Petri net against a hard coded expected state of the Petri net.

## 9.3 Implementation

We made use of the `JUnit` test framework to carry out our tests. The framework automatically executes any methods that begin with the string “test”. Furthermore, before running each such method it executes the `setUp()` method and after each test method is finished it executes the `tearDown()` method. In our case these two methods were used to create a new instance of `PIPE` before each test and to kill that instance upon completion.

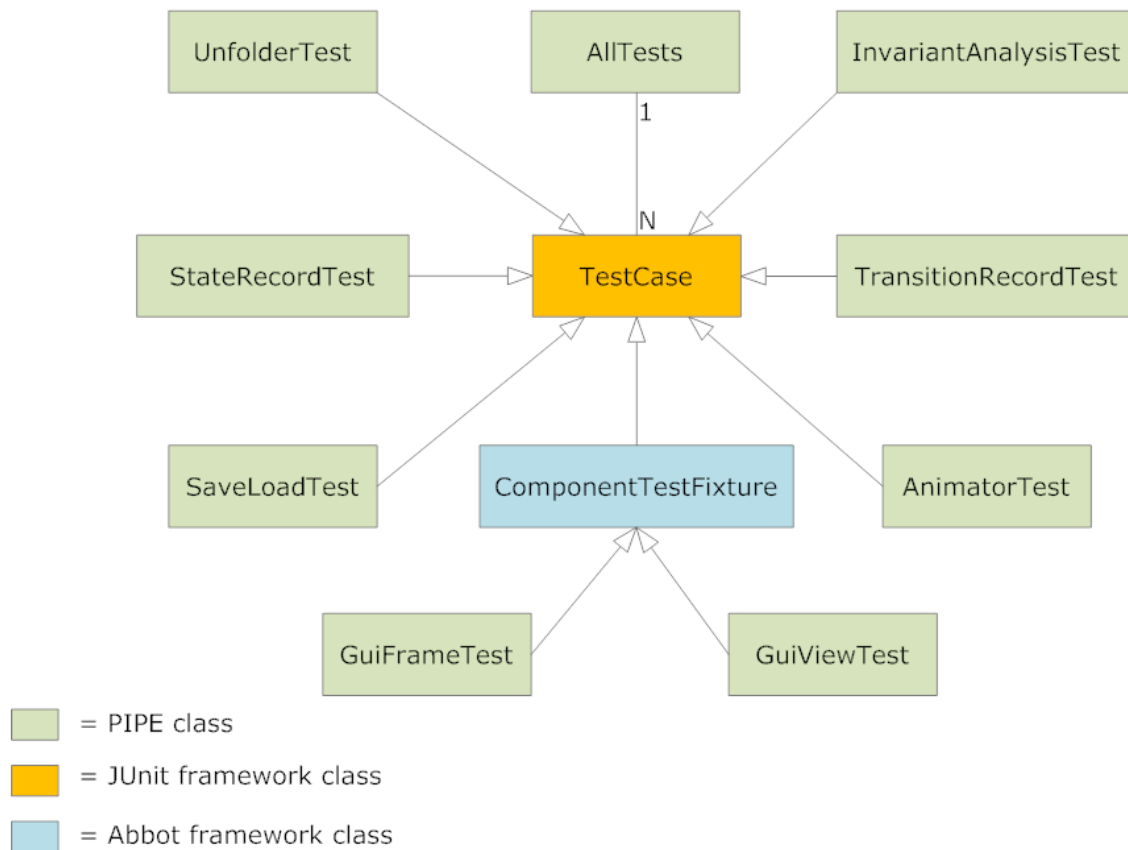


Figure 9.1: Structure of tests in PIPE

### 9.3.1 Imported Tests

We got hold of the other branch of PIPE (explained in section 2.4.2) and imported some of its tests into our branch. These tests are:

- **InvariantAnalysisTest**: Tests the “Invariant” analysis module.
- **GuiFrameTest**: Checks to see if the place button on the GUI functions correctly.
- **GuiViewTest**: Tests that users can select multiple items on the grid and drag them. It also tests the zoom functionality.
- **StateRecordTest**: Tests the `StateRecord` class that is involved with the “Reachability Graph Generator” analysis module.
- **TransitionRecordTest**: Tests the `TransitionRecord` class that is involved with the “Reachability Graph Generator” analysis module.

These existing tests, check a small subset of the existing functionality. They all ran successfully in our version.

### 9.3.2 AnimatorTest

This class ensures that manual/random firing of transitions occurs correctly by comparing the result of each firing against a set of expected results.

AnimatorTest
+ AnimatorTest(String name) + testAnimation() : void - setUp() : void - tearDown() : void - checkEnabled(LinkedList<String> transitionNames) : void - checkPlaceMarking(String placeName, int defaultMarking, int redMarking, int blueMarking) : void - checkTransitionVisibility(LinkedList<String> shouldBeVisible) : void - checkGroupTransitionCorrectness(LinkedList<String> shouldBeVisible, LinkedList<String> shouldBeEnabled, int expectedNumber) : void

Figure 9.2: Class AnimatorTest

- **checkEnabled:** The method receives a linked list of transitions that should normally be enabled. Using a for loop of assertions, it makes sure that each of these transitions is in fact enabled and highlighted on the GUI and that no other transitions are enabled or highlighted. If such an assertion fails, the user is told which transition of the model is not enabled/disabled as it should be.
- **checkPlaceMarking:** The method is passed a place name and the expected marking it should have. It gets an instance of this place from the model and checks the inputs against the actual place marking. Any discrepancies are explained to the user.
- **checkTransitionVisibility:** This aims to ensure that when firing a group transition, its firing modes (which are transitions) are exposed as expected. Takes a linked list of Strings that represent the names of transitions that should be visible on the grid at this instance. It gets an instance of each of these transitions from the model and ensures that they are in fact set to visible. Any discrepancies are explained to the user.
- **checkGroupTransitionCorrectness:** This aims to ensure that after firing a group transition, its exposed firing modes are contracted back into a grouped transition. It also checks the number of group transitions that should be present on the GUI and finally checks that the expected group transitions are enabled and no others. It accomplishes these tests by getting all group transitions from the model and checking them against the inputs of expected results.
- **testAnimation:** All the above methods can be considered as helper functions for this method. It loads up a specific “test model” and begins firing specific transitions on it. After each firing, it runs several of the helper functions to ensure the correctness of the model at that instance. It also checks the “Step back”, “Step forward”, “Fire a number of random transitions” and undo/redo features in the same way.

### 9.3.3 SaveLoadTest

This class aims to ensure that saving a net generates a correct PNML file and vice versa.

SaveLoadTest
<pre>+ SaveLoadTest(String name) + testLoad() : void + testSave() : void - setUp() : void - tearDown() : void - checkTransitionConnections(String transition, LinkedList&lt;String&gt; parents, LinkedList&lt;String&gt; children) : void</pre>

Figure 9.3: Class SaveLoadTest

- **checkTransitionConnections:** The method is passed the name of a transition and lists of the parents and children that should be connected to it. The method gets an instance of the transition from the model and then examines all its parents and children. Finally, the transition’s actual children and parents are checked against the expected values that were received as inputs (using assertions). Any discrepancies are clearly explained, stating which transition failed the test, what the expected parents and children were and what the actual children and parents are.
- **testLoad:** This method loads up a known “test model” and checks its whole structure using the **checkTransitionConnections** method. If PIPE is functioning correctly, then once the file is loaded into PIPE, the model should be identical to what is hard-coded into the test code.
- **testSave:** If **testLoad** has succeeded then we can use it to test the save method. This function loads our test model and saves it under a temporary file. It then checks the freshly saved file against our test model file line by line to ensure that the new file is identical to the old.

### 9.3.4 UnfolderTest

UnfolderTest
+ UnfolderTest(String name) + testProcessor() : void + testReadersWriters() : void + testSimpleCGSPN() : void - setUp() : void - tearDown() : void - checkPlaceMarking(String placeName, int expectedMarking) : void - checkTransitionConnections(String transition, LinkedList<String> parents, LinkedList<String> children) : void - checkArcWeight(String childName, String parentName,int expectedArcWeight) : void

Figure 9.4: Class UnfolderTest

- **checkTransitionConnections:** Identical to the same method found in the **SaveLoadTest** class explained above.
- **checkPlaceMarking:** Identical to the same method found in the **AnimatorTest** class explained above.
- **checkArcWeight:** Is passed a child, a parent and an expected arc weight. This method iterates through all arcs in the model to find one which has the specified child and the specified parent. It then asserts that the weight of this arc is equal to the expected arc weight received as input to this method.
- **testProcessor, testReadersWriters, testSimpleCGSPN:** These classes each load a particular test model and unfold it. They then make use of the above helper functions to check the resultant Petri net against a set of expected results. Any discrepancies are clearly stated to the user.

### 9.3.5 System test

We created a class **AllTests** that runs all of our unit tests to ensure that the system as a whole is fully functional.

## 9.4 Final Result

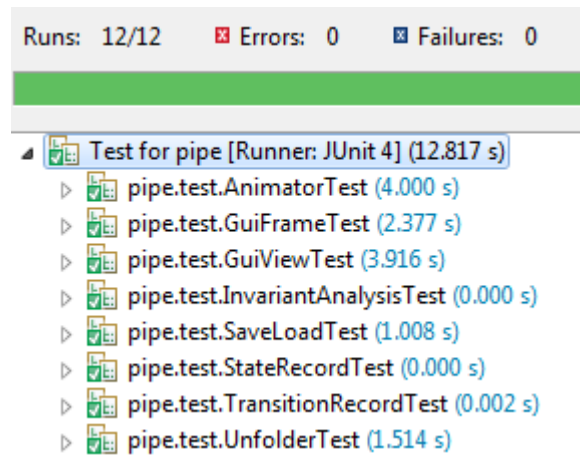


Figure 9.5: A successful run of class `AllTests`

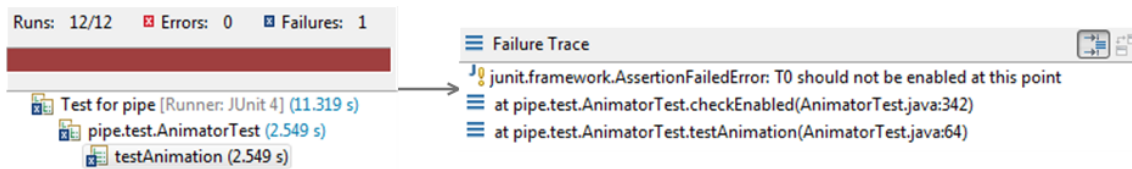


Figure 9.6: A failed run of class `AllTests`



# Chapter 10

## Evaluation

### 10.1 Vastly increased the modelling power of PIPE through the design, animation and analysis of CGSPNs

In this section we set out to prove that we have implemented what is necessary to design CGSPNs in PIPE and show how this has increased its modelling power.

#### 10.1.1 Case Study

Since it is difficult to quantitatively evaluate the design of CGSPNs we shall do so through a case study. We shall use the example presented in the Background section of a hospital's Accident and Emergency department. In Figure 10.1, the tokens represent initially healthy individuals who fall ill and are then serviced by the hospital.

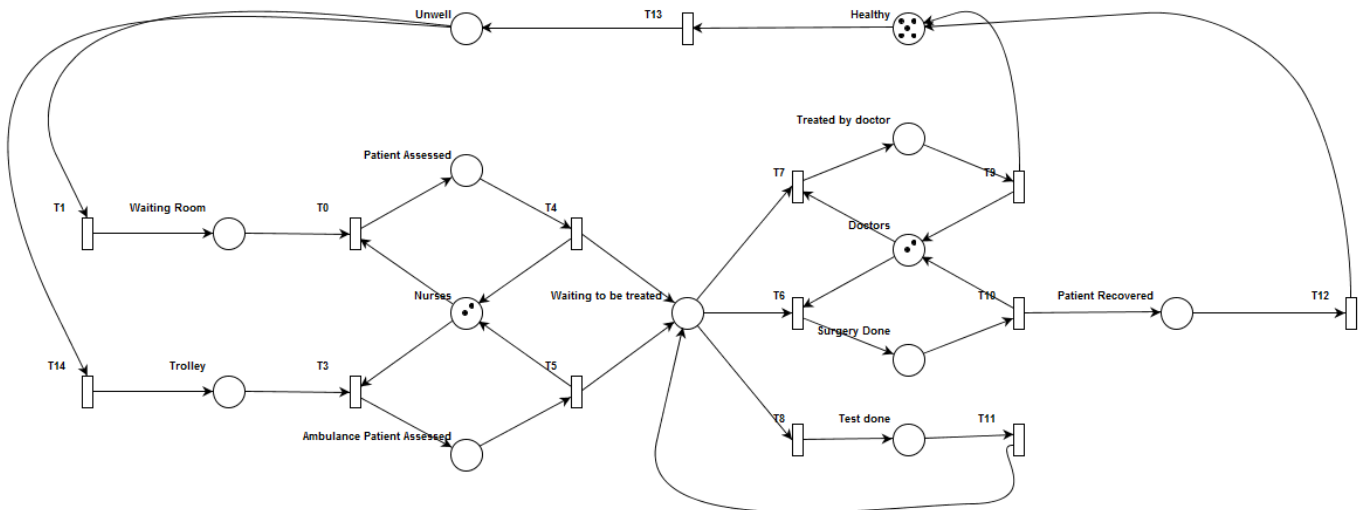


Figure 10.1: Modelling a hospital using a GSPN

We will now leverage our new functionality to improve the model by splitting patients into

two groups: patients who are critically ill and patients who are not. Such a separation allows for a more realistic model since in a real life scenario, you would expect a patient with a critical injury to be attended to faster than a person who came in for a check-up. The fact that any patients with critical injuries will be attended to faster can be modelled in a Petri net by increasing the rate of firing transitions that have to do with such patients. We shall show that using CGSPNs we are able to add this new feature to the existing model and yet retain an almost identical graphical representation.

### 10.1.2 Methodology

- We create new classes of patients through the “Specify Token Classes” and assign them to a particular token colour:
  - Critical patients: Red tokens.
  - Non-critical patients: Blue tokens.
  - Unassigned (default): Black tokens.
- We create multiple firing modes for each transition, by creating new transitions and grouping them together.
- We ensure that different types of patients are serviced differently by increasing the firing rate of transitions that fire red tokens, so that critical patients are attended to at a faster rate. This is done by right clicking on such transitions and increasing their rate.
- We can now simulate our model by switching to animation mode.

### 10.1.3 Results and Conclusions

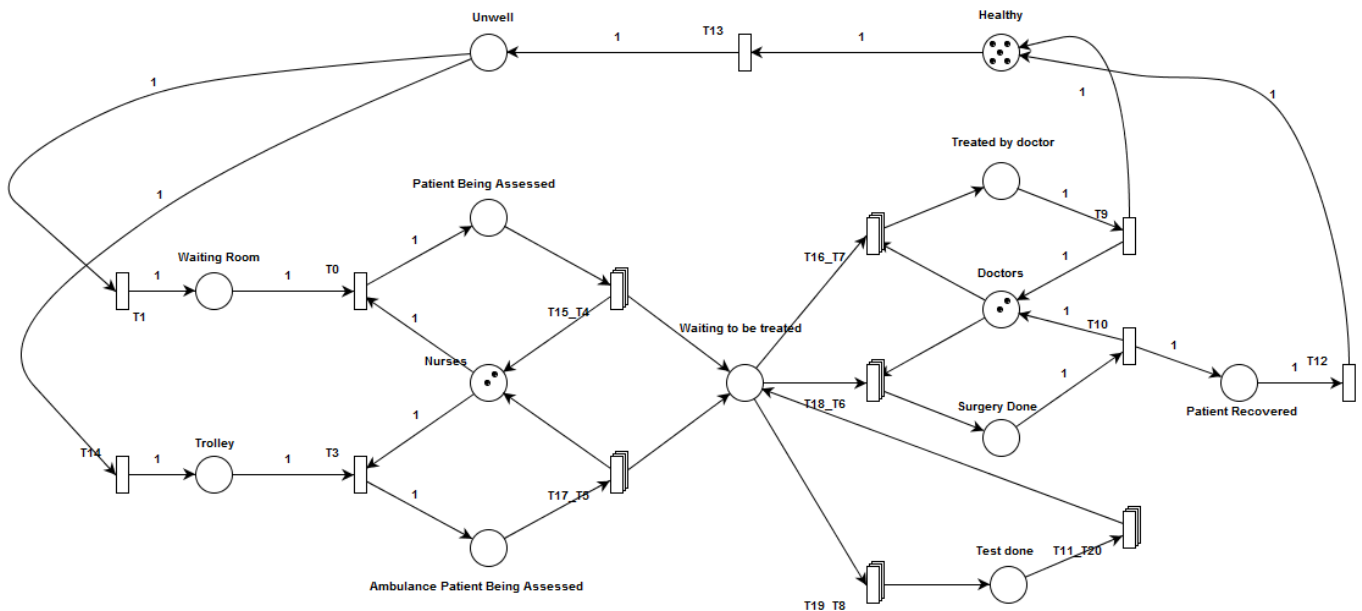


Figure 10.2: New representation of the hospital

Figure 10.2 shows our approach of representing firing modes. We have just added a new variable to the equation (that of injury classification) and yet our model is almost identical to what we had before (Figure 10.1). We are able to accomplish this by compacting several firing modes into one “Grouped Transition”. Underlying each of these grouped transitions, are the individual possible firing modes that are responsible for firing red/blue tokens where appropriate. These can be seen by clicking the “Ungroup All” button, the results of which are shown in Figure 10.3.

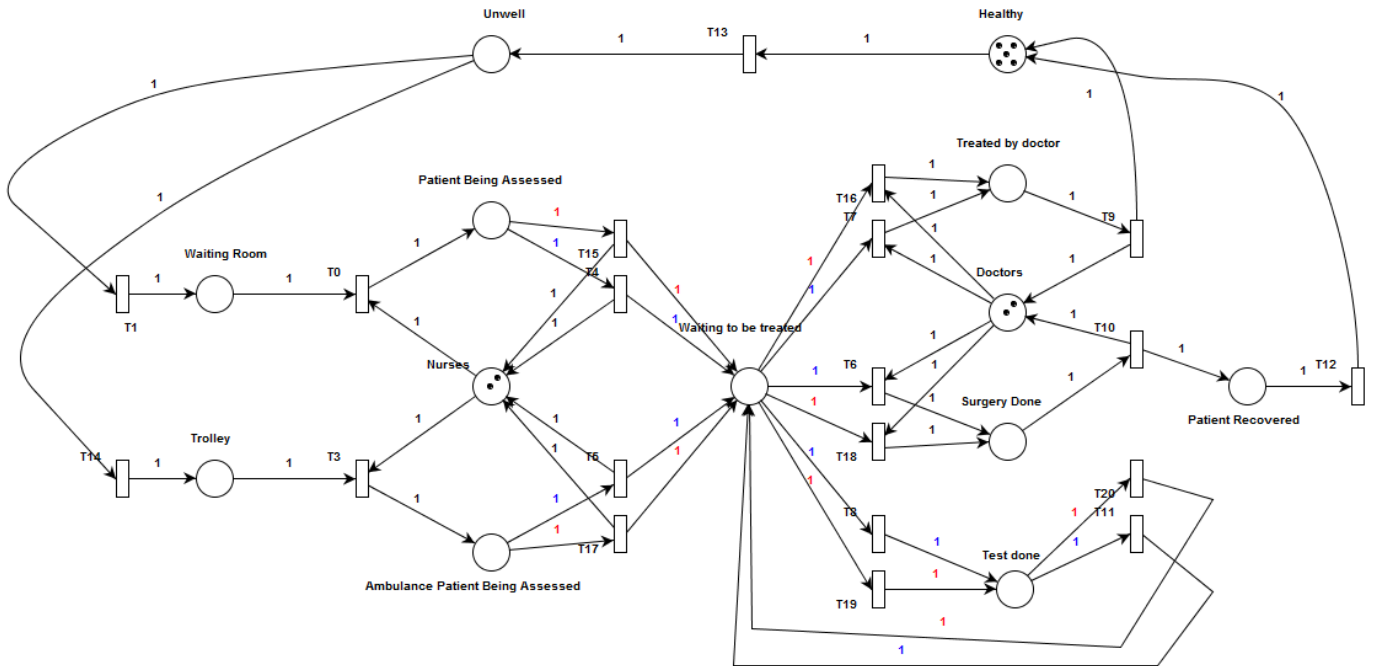


Figure 10.3: New representation of the hospital with exposed firing modes

For more clarity, let us consider the particular case where patients have just been assessed. At this point, they are classified as critical or non-critical patients. This means that only blue or red tokens will ever enter the “Waiting to be treated” place. Group transition T15\_T4 (Figure 10.2) consists of two firing modes:

- T15: If fired, will take a critical patient to the “Waiting to be treated” place.
- T4: If fired, will take a non-critical patient to the “Waiting” place.

### Animation

In animation mode, the user can pick out any enabled transition/group transition to fire. If a transition is selected, it is fired, otherwise the group transition temporarily unfolds to expose its underlying transitions. The user can then select which transition to fire and this will cause the group transition to fold back.

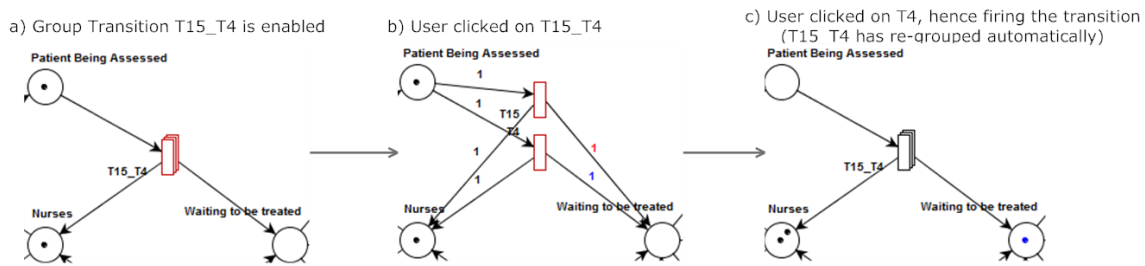


Figure 10.4: Animation of a CGSPN

Proof that the automatic firing of a number of random transitions is fully functional is presented in Section 10.1.3 where the analysis module `Simulation` performs this procedure to analyse the time spent by various token classes in each place.

### Quantitative analysis of servicing different types of tokens

From here onwards each type of patient should be serviced at different rates. We expect the critical patients to be moved around the hospital at a much faster rate than the non-critical patients. For example, if there were 1 critical and 1 non-critical patient in the waiting room heading for surgery, the critical patient should be attended to faster. This can be accomplished by making transition T18 an immediate transition (guaranteeing a red token will always be fired before a blue one) or simply increasing the rate of the transition (by right clicking and choosing edit transition). By doing this on all transitions that deal with critical patients we can ensure that critical patients are generally serviced at a faster rate than non-critical patients. To ensure that this is indeed the case, we can use the `Simulation` analysis module, the results of which are shown in Figure 10.5.

Place	Average number of tokens
Ambulance Patient Being Assessed_Unassigned	0.36064
Doctors_Unassigned	0.61039
Healthy_Unassigned	0.58941
Nurses_Unassigned	1.52248
Patient Being Assessed_Unassigned	0.11688
Patient Recovered_Unassigned	0.14985
Surgery Done_Unassigned	0.44256
Test done_Critical Condition	0.32368
Test done_Non-Critical Condition	0.2957
Treated by doctor_Unassigned	0.94705
Trolley_Unassigned	0.23377
Unwell_Unassigned	0.31568
Waiting Room_Unassigned	0.49051
Waiting to be treated_Critical Condition	0.33866
Waiting to be treated_Non-Critical Condition	0.3956

Figure 10.5: Analysis results of simulating 1000 firings over 50 runs

The multiple runs should ensure that on average the same amount of red/blue tokens

should reach the “Waiting to be treated” place. However, as the red tokens (critical patients) are serviced more frequently, we expect them to spend less time in this place. This is reflected in the results as at any one time there are less critical patients in the “Waiting to be treated” place than non-critical patients.

## Conclusion

With relative ease, we have managed to add a new patient class and associated routing characteristics to an existing model. We have managed to design a CGSPN modelling a hospital and animate it to observe the patients, nurses and doctors movement. We have also analysed the model to see on average how many patients/nurses/doctors are in each place (using the `Simulation` module). A similar result could have been achieved using the old version of PIPE but that would mean almost doubling the existing places and transitions in an already complicated model. This would make it almost impossible for someone to easily understand the graphical representation of the model. With our new implementation, we can continue to add new types of patients (e.g. medium priority patients) and the resulting model would still look exactly the same. A similar approach could be taken to model different types of doctors such as surgeons and GPs.

### 10.1.4 Known bugs/deficiencies

There are a few things we wished we had implemented better, but were unable to due to the time constraints of the project:

- Arrangement of elements when CGSPNs are automatically unfolded, are not always arranged in a neat way. This problem could possibly be addressed by using available packages specialising in automated graph layout, such as `Graphviz` [10].
- Excessive marking of places (5 token classes or more) is not handled gracefully. Currently such markings exceed the area they are confined to (the place) and can make the model look messy. A possible fix would be to replace these multiple markings with a special symbol. If such a symbol exists in a place then the user knows that he/she must hover over the place to see its marking.
- During animation, when users want to fire a transition within a group transition they click on the group transition to expand it and then on the transition they wish to fire. Once fired, the transitions are all contracted back into a group transition automatically. This automatic grouping will not occur if a user clicks on a group transition to expand it but then chooses to fire another transition in another group transition. This can be fixed by keeping a full history of all expanded group transitions during animation rather than just storing the last expanded group transition.

## 10.2 Quality of Structural Changes

PIPE was created with an MVC pattern in mind but unfortunately the various releases by various individuals have led to a rather inconsistent structure. PIPE does not make a clear separation between the model, view and controller. For example the class `GuiFrame`

seems to act as both the view and the controller. Being an open source project, it is clear that PIPE must undergo major refactoring to allow contributors to understand the code easily. However, as this was beyond the scope of this project, our goal was to ensure that we did not make matters worse.

A major goal throughout the project was to avoid cyclic dependencies between packages. Cyclic dependencies prevent an entity from being understood or reused in isolation. We used a tool called Structure101 [22] to analyse such dependencies between the various packages. Figure 10.6 shows the dependencies that were detected before and after our changes. Any classes involved in such cyclic dependencies are said to be part of a tangle. Although we have generally increased the number of calls between these tangled packages we have not involved new packages in the tangle. This way we have ensured that any future attempts to refactor the code will not be made any more complicated.

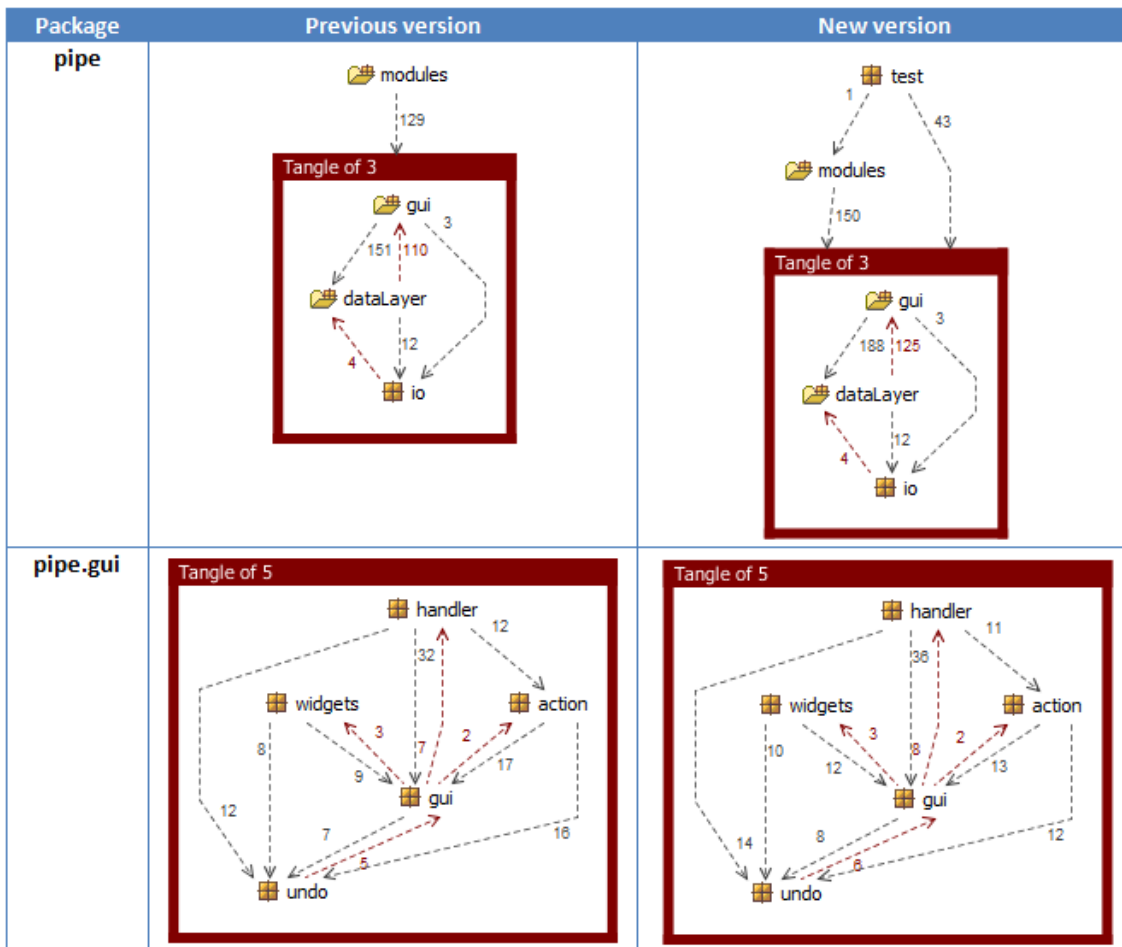


Figure 10.6: Structural dependencies of PIPE before and after our changes

### 10.3 Comparison with state of the art

In this section we compare our design and animation of CGSPNs with that of one of the most popular CPN tools, namely “CPN Tools” [15]. Note that we cannot guarantee that

we are using CPN Tools in the most efficient way and we are basing this evaluation on our understanding of the tool after many hours of self training. This is because we were unable to find experts of the tool to assist us and hence relied on the tool’s training material and examples to understand it.

### 10.3.1 Defining token classes

To define new token classes in CPN Tools the user must click “Declarations” and define new token classes in the form shown in Figure 10.7. To do the same in PIPE the user must click the “Specify token classes” button and fill in the table shown in Figure 10.8.

```

▼Declarations
  ▼Standard declarations
    ▼val n = 3;
    ▼colset R = index r with 1..n;
    ▼colset RES = index res with 1..3;
    ▼colset W = index w with 1..2;
    ▼colset RxW = union READ:R + WRI:W;
  
```

Figure 10.7: Defining token classes CPN Tools





Enabled	Token Class Name	Token Class Colour
<input checked="" type="checkbox"/>	Default	
<input checked="" type="checkbox"/>	Readers	
<input checked="" type="checkbox"/>	Writers	
<input type="checkbox"/>		

Figure 10.8: Defining token classes in PIPE

### Conclusion

The reason there are 4 token classes in Figure 10.7 is that users must also define expressions for combinations of these classes. For example, since some places hold both readers and writers we must define a new token class RxW, which we then define to represent the union of the classes R and W.

Under CPN Tools, users must be familiar with a certain convention for defining new token classes. The expressions can get very complicated incorporating further in-built methods such as union, product and intersection. PIPE does not require the use of such expressions and hence we can conclude that it is easier for new users to learn.

### 10.3.2 Marking a place

To mark a place with tokens in CPN Tools, the user must carry out 3 steps:

- Define which token classes may enter this place. This is done by clicking on a place, pressing *Tab* and typing the name of the token class (e.g. RxW).
- Define marking by:
  - Right click and hold button on place
  - Drag the mouse to the *Show Marking* area on the popup that appears.
  - Fill in the marking box with expressions of the form  $1 \text{ 'READ}(r(1))++ \dots$

To mark a place in PIPE, the user must right click it and choose *Edit Place* and then fill in each field with the number of tokens of each type to be added.

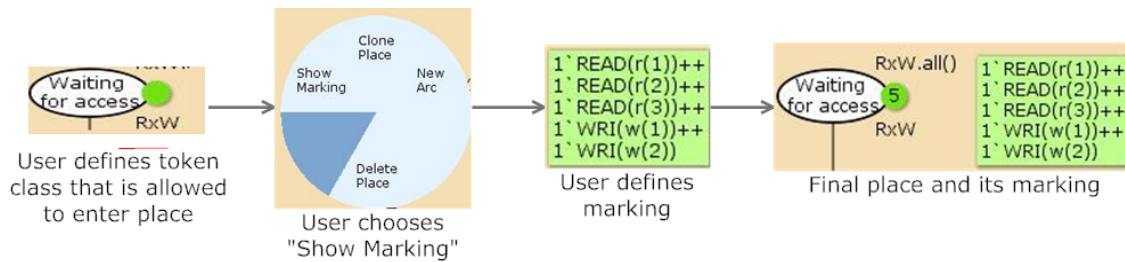


Figure 10.9: Marking a place in CPN Tools

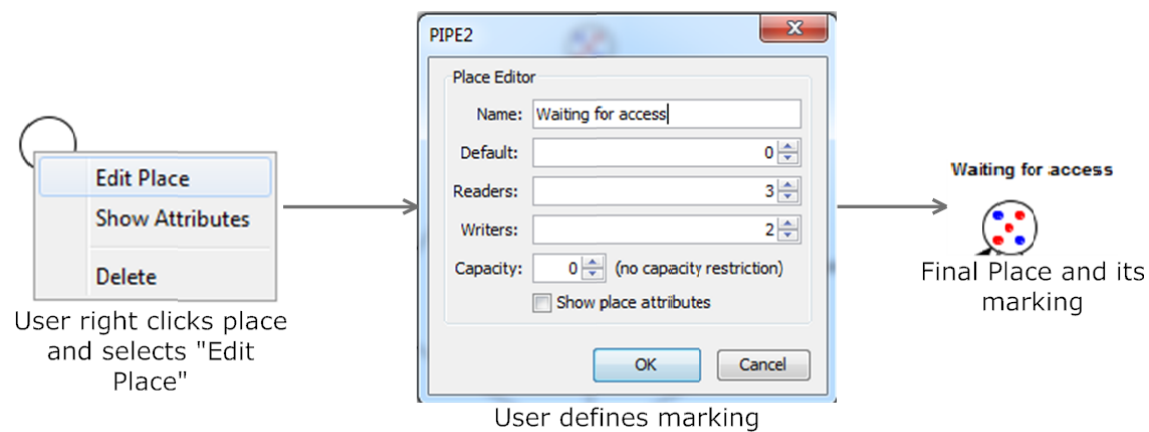


Figure 10.10: Marking a place in PIPE

## Conclusion

Marking a place in CPN Tools is a much more complex task made overly complicated by the fact that instead of defining a simple integer for each token class, an expression must be defined. Aspects of the user interface are also very unintuitive e.g. it took a very long time to discover how to edit the marking of a place.



### 10.3.3 Animation

To enter animation mode in CPN Tools the user must select the *Simulation* option from the toolbar on the left and drag it onto the canvas. A panel then appears by which a user can select the “Fire a transition” button.

To fire a transition in CPN Tools the user must bind variables defined in the arc weight to particular tokens in the input places.

To enter animation mode in PIPE the user must select the **Toggle animation** button. To fire a transition, the user must click on a group transition and then the transition to be fired. The number of tokens to be fired (arc weight) is denoted on the arcs.

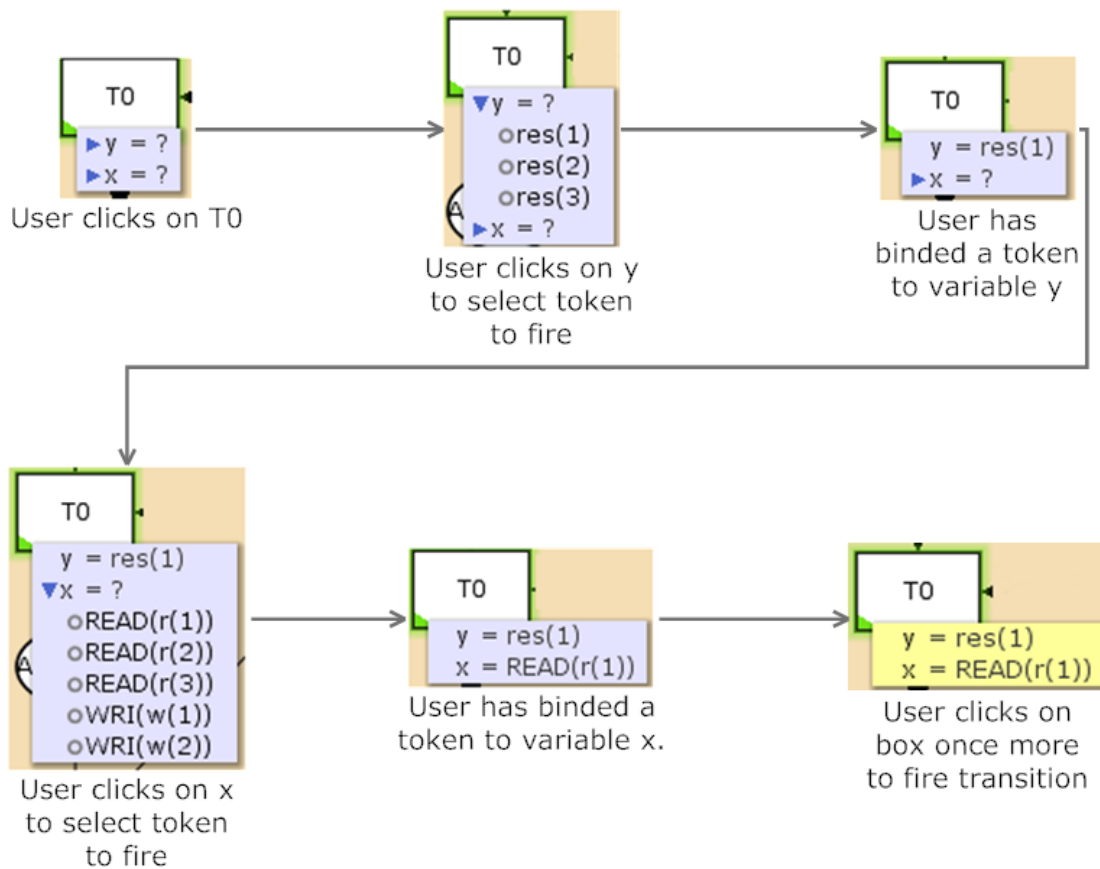


Figure 10.11: Firing a transition in CPN Tools

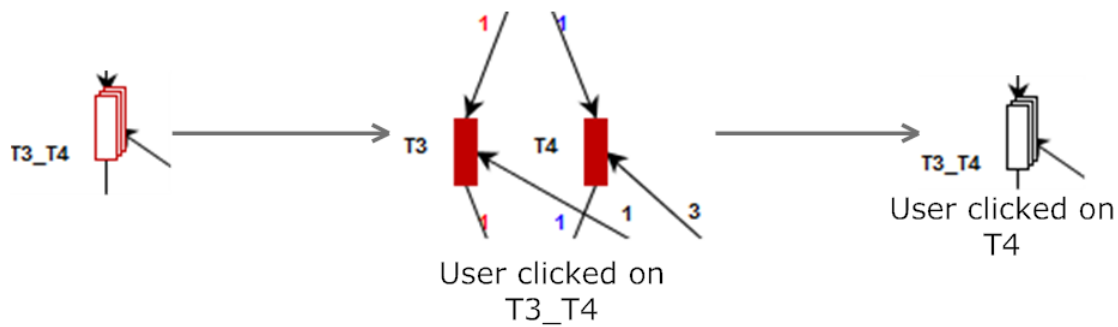


Figure 10.12: Firing a transition in PIPE

It should be noted that in the CPN Tools figure we are only firing one token of each type. To define different actions based on whether a reader or writer token is selected special functions must be defined. In that case the arc expression would read  $function(x)$  instead of  $x$  but the bindings would still have to occur.

## Conclusion

CPN Tools requires more clicks to fire a transition compared to PIPE. However, CPN Tools seems to offer a more powerful model since it allows users to number the various readers and writers. Hence you can differentiate between each reader and each writer and choose which one will move to the next place. Although this is indeed an advantage over PIPE, we could still replicate such behaviour by simply defining more token classes in PIPE to differentiate between each reader and writer. Choosing between a slightly more powerful model and a simpler/faster animation is a matter of opinion and hence we cannot declare a winner under this criteria.

### 10.3.4 Conclusion

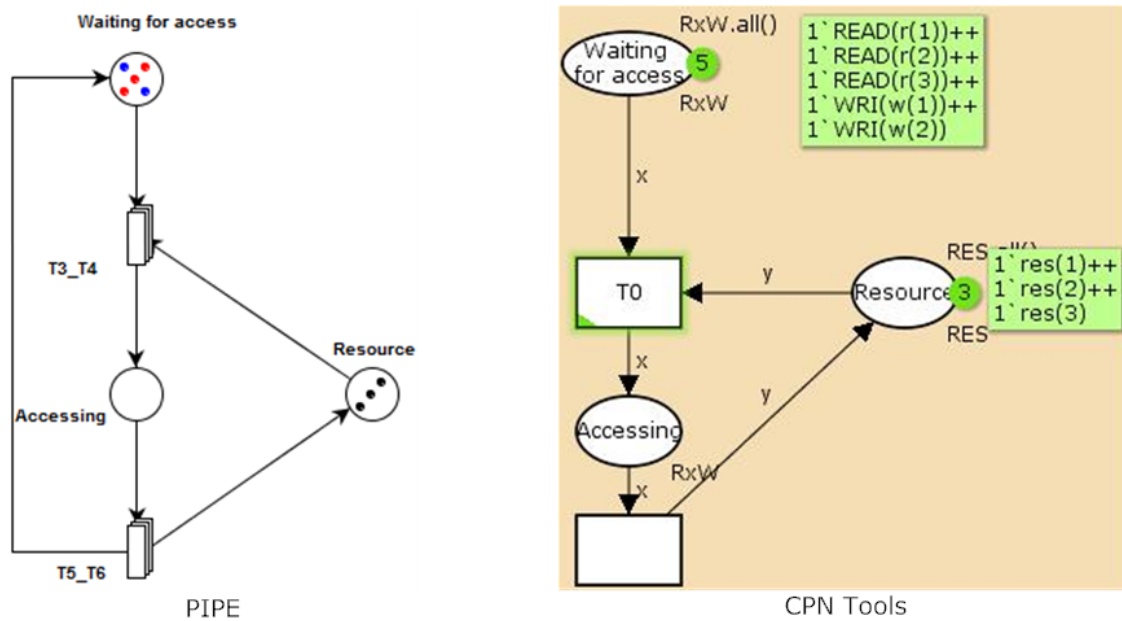


Figure 10.13: Readers/Writers problem in CPN Tools and PIPE

It seems as if CPN Tools can boast a slightly more powerful model thanks to its ability to identify specific members of a token class. An appropriate analogy would be to imagine a token class as an array and a token as an element of that array. Although both PIPE and CPN Tools can define arrays of different types and use elements of the array as they wish, CPN Tools is able to choose particular elements in the array (i.e. `Array[2]`) whereas PIPE does not distinguish between each element (i.e. `Array[x]`). As we have explained above, this deficiency can easily be overcome by PIPE with the definition of a new token class for each element of the array.

CPN Tools does not come across as a user friendly tool. Common functions such as saving/loading a Petri net require dragging toolbar buttons into the canvas and conventions such as the undo action (`ctrl+z`) have been implemented using their own methods (*right click+drag+select option from popup*). Additionally, the tool requires users to learn all the conventions for defining expressions and formulas (for token classes, arc weights and firing modes).

PIPE can boast a very clean and intuitive interface that allows users to design CGSPNs with little to no training. Our representation of CGSPNs does not require users to make use of complex expressions or formulas and the graphical model allows users to clearly understand the model at a glance. Care has been taken to provide consistency allowing users to copy/paste/undo/redo/save/load with the standard shortcut keys or the standard toolbar menus (e.g. File -> Save).

## 10.4 Added ability to transform any CGSPN to a GSPN to allow analysis by non-CGSPN compatible modules/tools

The best way to evaluate the unfolding mechanism of a CGSPN is to unfold one for which we already have the correct unfolded version. More specifically, we must:

- Step 1: Find a CGSPN along with its unfolded version (a Place Transition net which describes the exact same model)
  - We shall use the Dual Processor problem presented earlier (as defined by [3]) where both processors access a common bus, but not simultaneously.

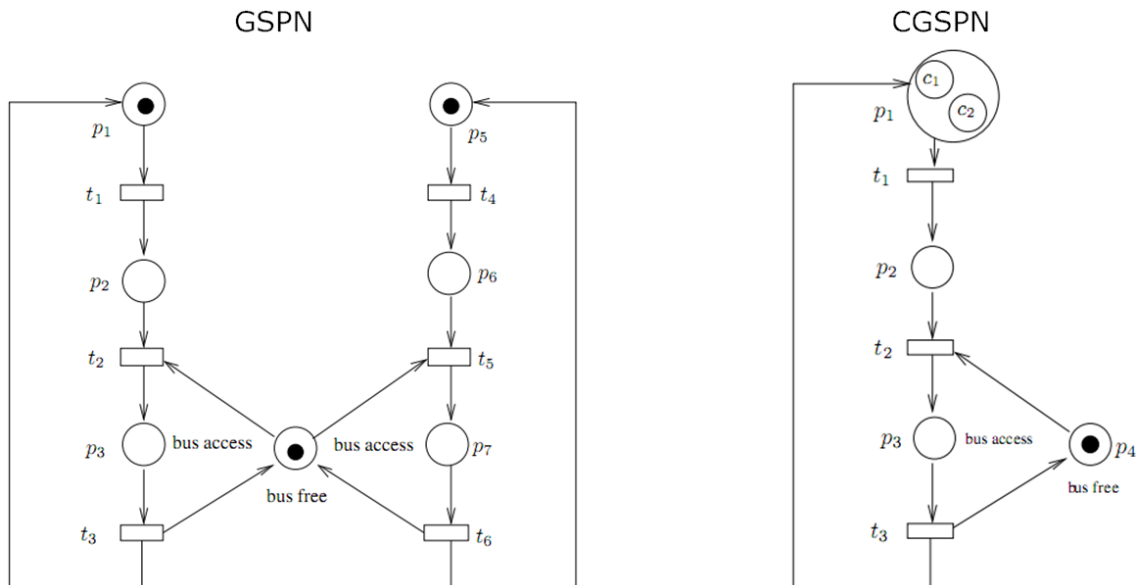


Figure 10.14: Dual Processor problem represented as a CGSPN and GSPN

- Step 2: Model the CGSPN in PIPE

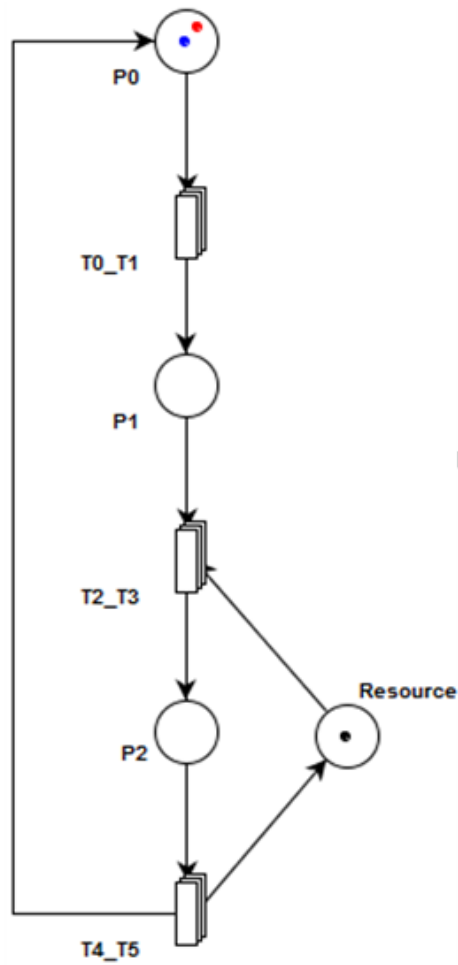


Figure 10.15: Dual Processor problem represented as a CGSPN in PIPE

- Step 3: Unfold the CGSPN via PIPE and compare the unfolded Place Transition net with that found in step 1 By clicking on the “Unfold” button whilst having open the CGSPN in step 2, the unfolded version was generated.

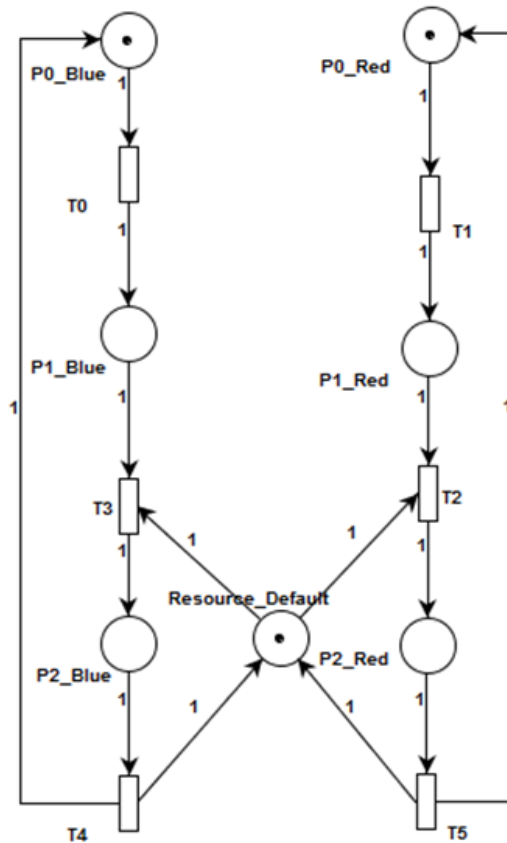


Figure 10.16: Dual Processor problem unfolded to a GSPN in PIPE

The results are identical to our example net with the only difference being the place and transition names. The same type of check has already been made during the design and implementation stage of the unfold (Section 7.4) where we unfolded the readers and writers problem to ensure the result is as expected. Additionally, our test framework checks the unfolding mechanism against three different models.

## 10.5 Retained PIPE's ease of use and core functionality

In this section we ensure that the changes we have made have not downgraded PIPE's functionality in any way and have not tarnished its ease of use.

### 10.5.1 Retain PIPE's functional correctness

Before any changes were made we implemented a test framework (Chapter 9) that tested the design/animation of GSPNs and before adding each new functionality we would extend the test framework appropriately. As this was not a foolproof method (it is impossible for tests to check every possible scenario) we had to make manual checks, loading and animating various Petri nets along the way. Such manual checks were done excessively after our implementation was complete and all tests now succeed.

### **10.5.2 The process of creating/editing GSPNs remains almost completely unchanged**

The users should not have to be burdened with all the difficulties that come with coloured tokens unless they are working with a CGSPN. Initially we planned on evaluating this by documenting the exact mouse clicks and button presses required to build a simple GSPN in PIPE and ensure that the same procedure will result in the same GSPN under the new implemented version. This method was deemed unnecessary as the method of designing and animating a GSPN has remained unchanged. Although the users can carry out additional actions to add colour to a Petri net, they go through the exact same procedure and same number of clicks (as in PIPE's previous version) to design a GSPN.

## **10.6 Added ability to save/load CGSPNs**

We would like to ensure that once a CGSPN is saved and loaded back into the program, we get the exact same net. This was checked by creating a Save/Load test in our testing framework (as outlined in Chapter 9). The module opens a pre-defined Petri-net and records all its properties including:

- Marking of places
- Properties of transitions
- Arc connections and weights between transitions and places
- Positions of all above objects
- Token classes

These properties are all checked against known expected values. Finally the CGSPN is then saved under a different name, and is checked against the file loaded, line by line to ensure they are identical. Additionally, our new PNML definition's syntactical XML correctness was also validated via an online XML Validator tool [25]. The success of carrying out this test with multiple test files means that this criteria has been met.

## **10.7 Present tool to Petri-net experts to get feedback on the features and its ease of use**

In any project, getting feedback from the end-user is a crucial step. Throughout the implementation we were closely conversing with academics who have great knowledge and experience with Petri nets to ensure that our new functionality was intuitive, functional and easy to use. Such people include Nick Dingle a Research Associate at Imperial College London who developed a Petri Net editor for his MSc in Computing Science that triggered the beginning of PIPE [9]. Nicholas Anastasiou, a PhD student at Imperial College London who is currently working on performance model construction from location tracking data has also reviewed the work on a regular basis and plans to use this new implementation with his work in the future. Thanks to them, any lack of functionality and difficulties of

usability were pointed out during the project development to ensure that the final result presented in this report fully meets the user's expectations.



# Chapter 11

## Conclusion

Working with PIPE has given me an appreciation for this powerful tool for designing and simulating complex systems. One of the greatest challenges faced throughout this project was understanding what the 204 existing classes did and how they worked. This led me to realise the importance of following good software engineering design practices and correctly adhering to a decided upon software architecture. This is even more important in an open source environment, where users are constantly modifying other people's code who are in many cases no longer involved in the project. In my opinion, any open source project should have:

- Comments explaining the code.
- Tests: To ensure changes to the code will not break existing functionality but also to assist others in understanding what various methods do.
- SVN/CVS: To ensure changes can be rolled back and users can examine (and understand) changes between different versions.
- Detailed documentation/reports: So that users know where in the code changes have been made and hence know where to look if modifications must take place.

Throughout this project, we have designed and implemented a way for users to design, animate and analyse CGSPNs in PIPE2. Through a case study (Section 10.1.1), we show how these changes have significantly increased PIPE's modelling power whilst keeping the graphical model intact. By using our own method for representing such models (Chapter 6), we introduce an alternative to the somewhat complicated existing and accepted form of representation (Section 6.2.1). Our unfolding mechanism (Chapter 7) ensures that users can export any CGSPN to GSPN analysis modules or even external tools that do not support CGSPNs. Although our case study involves modelling people's movements within a hospital, PIPE2 can now be used in a wide range of fields that make use of CGSPNs. Some interesting applications of CGSPNs under research include the conjunction of UML with CGSPNs to provide a more comprehensive method for designing software [16] and the automated Petri net performance model construction from location tracking data [1]. The latter, is part of a PhD project currently in progress at Imperial College London that will use the new version of PIPE to visualise and analyse CGSPN performance models of systems. Different token colours will be used to model different phases of customer service and customer classes.

## 11.1 Future Work

- As explained in Section 2.4.2, PIPE2 has been split into two branches by two different educational establishments. This means features can be found in each branch that do not exist in the other. This project has extended one of these branches with the ability to support CGSPNs. A project is currently being undertaken by Imperial MSc Student Matthew Cottingham to integrate this branch with the other, to create a complete fully featured version. This project will also aim to refactor the code, cleaning up the messy software architecture that has resulted after many years of modifications from various people.
- In this project we have implemented an unfoldier that will transform a CGSPN to a GSPN. An interesting idea would be to reverse engineer that functionality so that GSPNs that exhibit certain properties may be automatically folded into a CGSPN to automatically make candidate models more compact.
- Currently grouped transitions (Chapter 6) are GUI elements and are not part of the model itself. Although excluding them from the model is indeed a correct approach, it comes at a cost. When loading a saved net, the group transitions are no longer present and the user must select the “Group All” button to regroup transitions. As this often requires some manual rearrangement on the grid, this can get tedious. It would be useful if we could add PIPE specific information to the PNML file allowing it to load the group transitions as well.
- If we had more time we would have also liked to overcome the known problems and deficiencies listed in the evaluation section (Section 10.1.4). Hopefully these can be addressed before the next release of PIPE2.

# Bibliography

- [1] N. Anastasiou. *Automated Construction of Petri Net Performance Models from Location Tracking Data*. MSc Thesis, Imperial College London, 2009.
- [2] Susanna Wau Men Au-Yeung. *Response Times in Healthcare Systems*. PhD thesis, Imperial College London, UK, 2008.
- [3] Falko Bause and Pieter S. Kritzinger. Stochastic Petri Nets: An Introduction to the Theory. *SIGMETRICS Perform. Eval. Rev.*, 26(2):2–3, 1998.
- [4] Jonathan Billington, Sren Christensen, Kees Van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, Michael Weber, and Coremedia Ag. The Petri Net Markup Language: Concepts, Technology, and Tools. In *In Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003)*, pages 483–505. Springer, 2003.
- [5] James Bloom, Clare Clark, Camilla Clifford, Alex Duncan, Haroun Khan, and Manos Papantoniou, 2003. Platform Independent Petri Net Editor.
- [6] Pere Bonet, Catalina Lladó, Ramon Puigjaner, and William J. Knottenbelt. PIPE v2.5: A Petri Net Tool for Performance Modelling. In *23rd Latin American Conference on Informatics (CLEI 2007)*, October 2007.
- [7] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudó. Greatspn 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. 24:47–68, 1995.
- [8] PNML Conventions Document. <http://www2.informatik.hu-berlin.de/top/pnml/download/pnml/conv.rng>.
- [9] Nicholas J. Dingle. Production of the Extensible Petri Net Editor/Animator MEDUSA. In *Masters thesis*, 2001.
- [10] J. Ellson, E.R. Gansner, E. Koutsofios, S.C. North, and G. Woodhull. Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools. In M. Junger and P. Mutzel, editors, *Graph Drawing Software*. Springer-Verlag, Berlin/Heidelberg, 2004.
- [11] Paul A. Fishwick. *Handbook of Dynamic System Modeling (Cpaman & Hall/Crc Computer and Information Science)*. Chapman & Hall/CRC, 2007.
- [12] Reinhard German, Christian Kelling, Armin Zimmermann, Gnter Hommel, Technische Universitt Berlin, and Fachgebiet Prozedatenverarbeitung Und Robotik. Timenet - A Toolkit for Evaluating Non-Markovian Stochastic Petri Nets. *Performance Evaluation*, 24:69–87, 1995.

- [13] G. Horton. A New Paradigm for the Numerical Simulation of Stochastic Petri Nets with General Firing Times. In *Proceedings of the European Simulation Symposium*, 2002.
- [14] K. Jensen. Coloured Petri Nets and the Invariant Method. *Theoretical Computer Science*, 14:317–336, 1981.
- [15] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer*, 9(3):213–254, 2007.
- [16] Jens Bk Jrgensen. J.b.: Coloured Petri Nets in UML-Based Software Development Designing Middleware for Pervasive Healthcare. Technical report, In Kurt Jensen (Ed.): Proc. of the Fourth International Workshop on Practical Use of Coloured Petri Nets and the CPN, 2002.
- [17] G. Conte M.A. Marsan and G. Balbo. A Class of Generalized Stochastic Petri Nets for Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93–122, 1984.
- [18] Marc Meli, Catalina M. Llad, Ramon Puigjaner, and Connie U. Smith. An Experimental Framework for PIPE2. *Quantitative Evaluation of Systems, International Conference on*, 0:239–240, 2008.
- [19] C. Adam Petri and W. Reisig. Petri Net. *Scholarpedia*, 3(4):6477, 2008.
- [20] Petri Net Tool Database. <http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html/>.
- [21] PIPE Homepage. <http://pipe2.sourceforge.net/>.
- [22] Structure 101. <http://www.headwaysoftware.com/products/structure101/index.php>.
- [23] W. M. P. van der Aalst, K. M. van Hee, and H. A. Reijers. Analysis of Discrete-time Stochastic Petri Nets. *STATISTICA NEERLANDICA*, 54(2):237–255, 2000.
- [24] Bernd Werther. Colored Petri Net Based Modeling of Airport Control Processes. *Computational Intelligence for Modelling, Control and Automation, International Conference on*, 0:108, 2006.
- [25] XML Validation. <http://www.xmlvalidation.com/>.
- [26] Armin Zimmermann. Modeling of Manufacturing Systems and Production Routes Using Colored Petri Nets. In *Proc. of the 3rd IASTED Int. Conf. on Robotics and Manufacturing, Cancun*, pages 380–383, 1995.