

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

---

# PIPE — The Great Re-Plumbing

---



*Author:*  
Sarah TATTERSALL

*Supervisor:*  
Dr. William KNOTTENBELT

June 2014

Submitted in part fulfilment of the requirements for the degree of Master of Engineering  
in Computing of Imperial College London



## Abstract

Stochastic Petri nets (SPNs) are a high-level formalism for modelling and analysing the behaviour of complex concurrent systems. Their notion of time provides a convenient level of abstraction above low-level Markov processes. Their main strengths are their convenient graphical representation and their natural expressions of synchronisation.

The Platform Independent Petri net Editor (PIPE) is an open-source tool for building and evaluating Petri nets. Since 2003 features have been added by many contributors with a general lack of consideration for the quality of the software engineering and the impact on the correctness of other features. Whilst the user interface appears polished the underlying codebase is disjoint and buggy. A quantitative analysis of the codebase reveals it to be highly tangled containing 40 cyclic relationships and 12 904 code quality issues.

The present work presents a complete renewal of PIPE's underlying software architecture as well as new analysis modules which exploit the features of multi-core processors. We develop new back-end models of a Petri net and its underlying Markov chain and deploy various best-practices and design patterns such as the publish-subscribe and model-view-controller. Using a Backus-Naur Form grammar, we have renewed the support for functional expressions, which substantially extends the expressive power of Petri nets. As a result of these and other changes the number of cyclic relationships and code quality issues has been reduced to 13 and 937.

To improve build quality a rigorous test suite of 1042 unit and integration tests has been deployed with a continuous integration server. The development process was modernised and streamlined by migrating from CVS to Git and using Maven as a build system.

The new analysis modules include a state space exploration module which uses a multi-threaded MapReduce-style state explorer that is around 200x faster than the original implementation when running on 8 virtual cores. It can generate a million state reachability graph in under 6 minutes. A parallel Gauss-Seidel steady state solver was developed which when running on 4 virtual cores is up to 5x as fast as its sequential counterpart.

We have developed a robust, reliable and powerful platform upon which future generations of researchers can use and extend. Indeed scientists at the International Computer Science Institute in Berkeley California are already in the process of customising our work to build and solve neural cognitive models.



## Acknowledgements

I would like to express my thanks and appreciation to:

- Dr. William Knottenbelt, my supervisor, for always making time for me. Our project meetings were a real source of inspiration and always good fun.
- Professor Peter Harrison, my tutor, for his support, guidance and continuous belief in my abilities over the past four years at Imperial College London.
- Stephen Doubleday for his enthusiasm, invaluable feedback and for promoting the use of PIPE 5 at ICSI and Google.
- Christoph Beck, a developer at Stan4J, for providing me with a free license for academic purposes.
- My friends and family for their tremendous support this year.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Objectives . . . . .	3
1.3	Contributions . . . . .	3
1.4	Report structure . . . . .	4
<b>2</b>	<b>Background</b>	<b>6</b>
2.1	Markov Theory . . . . .	6
2.1.1	Stochastic processes . . . . .	6
2.1.2	Markov process . . . . .	7
2.2	Petri nets . . . . .	9
2.2.1	Anatomy . . . . .	9
2.2.2	Enabling and firing . . . . .	11
2.2.3	Coloured Petri nets . . . . .	13
2.2.4	Stochastic Petri nets . . . . .	14
2.2.5	Other extensions . . . . .	17
2.3	Petri net analysis . . . . .	17
2.3.1	Exploring the state space . . . . .	17
2.3.2	Solving the steady state . . . . .	18
2.3.3	Performance analysis . . . . .	22
2.4	Petri Net Markup Language . . . . .	22
<b>3</b>	<b>PIPE 4 - an open-source Petri net editor</b>	<b>24</b>
3.1	Using PIPE 4's editing features . . . . .	25

3.2	Editing a Petri net . . . . .	27
3.2.1	Petri net component menus . . . . .	27
3.2.2	Functional expressions . . . . .	27
3.3	Animation . . . . .	28
3.3.1	Firing transitions . . . . .	28
3.3.2	Animation history . . . . .	29
3.4	Analysis modules . . . . .	29
<b>4</b>	<b>Analysing the PIPE 4 software architecture</b>	<b>31</b>
4.1	Model-view-controller structure . . . . .	31
4.2	Plug in modules . . . . .	33
4.2.1	The IModule interface . . . . .	33
4.2.2	The state space exploration algorithm . . . . .	35
4.2.3	The implementation of the state space exploration . . . . .	38
4.2.4	Steady state at equilibrium . . . . .	40
4.3	Quantitative code analysis . . . . .	43
4.3.1	Static analysis . . . . .	43
4.3.2	Cyclic dependencies . . . . .	44
4.3.3	Documentation . . . . .	44
4.3.4	Coding style . . . . .	46
4.4	Critical aspects of functionality . . . . .	52
4.4.1	Global state . . . . .	52
4.4.2	Functional expressions . . . . .	52
4.4.3	Infinite server semantics . . . . .	54
4.4.4	Transition throughput analysis . . . . .	56
<b>5</b>	<b>Modernising the build process</b>	<b>60</b>
5.1	Version control . . . . .	60
5.2	GitHub . . . . .	63
5.3	Travis CI . . . . .	71
5.4	Automating the build process via Maven . . . . .	73
<b>6</b>	<b>A new software architecture for PIPE 5</b>	<b>75</b>
6.1	Creating a new Petri net model . . . . .	75
6.2	Petri net component software architecture . . . . .	79

6.2.1	The publish–subscribe design pattern . . . . .	79
6.2.2	The acyclic visitor design pattern . . . . .	80
6.2.3	The builder design pattern . . . . .	85
6.2.4	Domain Specific Language . . . . .	85
6.3	A new implementation of functional weights . . . . .	87
6.3.1	ANTLR v4 . . . . .	87
6.3.2	Grammar . . . . .	88
6.3.3	Implementing an API . . . . .	88
6.4	Input/output . . . . .	91
6.4.1	Parsing XML in Java . . . . .	91
6.4.2	Implementation . . . . .	93
6.5	Documentation . . . . .	97
6.5.1	GitHub pages . . . . .	97
6.5.2	Javadoc . . . . .	99
<b>7</b>	<b>Re-engineering the existing view code</b>	<b>100</b>
7.1	Modifying the architecture . . . . .	100
7.1.1	Applying a model–view–controller architecture . . . . .	100
7.1.2	Eradicating the ApplicationSettings class . . . . .	101
7.1.3	Replacing custom code with third party libraries . . . . .	102
7.2	Changes in displayed components . . . . .	103
7.2.1	Deleting the tokens . . . . .	103
7.2.2	Changing the rate parameters . . . . .	104
7.3	Useful errors . . . . .	106
<b>8</b>	<b>New analysis modules</b>	<b>107</b>
8.1	State space exploration . . . . .	107
8.1.1	Sequential state space exploration . . . . .	108
8.1.2	The explored states data structure . . . . .	114
8.1.3	A parallel implementation . . . . .	116
8.1.4	Improving the performance . . . . .	117
8.1.5	PIPE 5 reachability graph UI . . . . .	120
8.2	State space exploration module . . . . .	122
8.2.1	A sequential implementation . . . . .	122



8.2.2	A parallel Jacobi implementation . . . . .	122
8.2.3	A parallel asynchronous Gauss-Seidel implementation . . . . .	127
8.3	Performance Analysis . . . . .	127
<b>9</b>	<b>Evaluation</b>	<b>129</b>
9.1	Quality metrics . . . . .	129
9.1.1	Reduction in tangles . . . . .	130
9.1.2	Static analysis . . . . .	133
9.1.3	An improved test suite . . . . .	134
9.1.4	Documentation . . . . .	136
9.2	Analysing the new modules . . . . .	137
9.2.1	Analysing the state space explorer module . . . . .	137
9.2.2	Scalability . . . . .	138
9.2.3	Steady state solver results . . . . .	142
9.3	Testimonial . . . . .	146
9.4	Strengths and limitations . . . . .	146
<b>10</b>	<b>Conclusion</b>	<b>148</b>
10.1	Achievements . . . . .	148
10.2	Lessons learned . . . . .	149
10.3	Future work . . . . .	150
10.3.1	Improving the convergence time of the steady state solver . . . . .	150
10.3.2	Reversed Compound Agent Theorem . . . . .	151
10.3.3	Cloud based modules . . . . .	151
10.3.4	An improved graphical interface . . . . .	151
	<b>Appendices</b>	<b>158</b>
<b>A</b>	<b>An example PNML listing</b>	<b>159</b>



# Chapter 1

## Introduction

The Petri net, like the steam engine, is a technological breakthrough that changed the world.

Peter Singer (paraphrased)

### 1.1 Motivation

Stochastic Petri nets are a powerful mathematical modelling language for describing distributed systems. Their notion of time provides a convenient level of abstraction above low-level Markov processes. Their main strengths are their convenient graphical representation and their natural expressions of synchronisation. They are often chosen over Markov chains to model real world problems because of their strengths and the isomorphism that exists between them. Interesting applications include the modelling of computer virus life cycles [1], modelling information flow in security policies to determine the possible ways in which information may be compromised [2] and modelling of extreme unforeseen events, such as very heavy rainfall at water sewage plants which can lead to waste water overflowing into the streets [3].

The Platform Independent Petri net Editor (PIPE) was created by Imperial College London in 2003 for graphically creating, simulating and analysing Petri nets. In the last eleven years PIPE has been maintained and extended by the open-source community and Imperial

College London. Whilst there are many Petri net tools that exist [4], PIPE differentiates itself from these by offering 13 plug-in analysis modules.

Sadly over the years PIPE has become unmaintainable due to years of neglect of the core codebase, a lack of good testing, advances in the Java language, and new more developer friendly technologies becoming available. The code itself has become bloated, difficult to understand, over-complicated and riddled with bugs. When adding a new feature developers pay little attention to the quality of the software architecture they leave behind or the impact their changes make on other features and analysis modules. PIPE is in desperate need for a re-design. Reviews on the SourceForge home page indicate that whilst a good application it has some fundamental flaws; one quotation indicates this clearly [5]:

*“My students have been using PIPE2 in a master-level course on concurrent systems modelling and analysis for five years. It has served us well so far: it is user-friendly and easy to use for beginners and has a nice set of supported analyses. I am however very concerned about the current evolution of the tool. PIPE 3 (2010) and PIPE 4 (2011) have been released with hardly any information on what has been added, changed or improved. Some very serious bugs . . . have been reported (see bug tracker) with no apparent reaction so far from the developers. Unless you are really interested in the new features and is [sic] willing to live dangerously, I would recommend that you stick to release 2.5 rc5.”*

All releases, including the latest release of PIPE 4, are only available as a stand-alone graphical user interface (GUI) application but the International Computer Science Institute (ICSI) have expressed a deep interest in having a set of back-end models on which they can perform various analyses. Their fundamental interests lie in Petri nets for the application of neural networks so it is key to them that PIPE supports a good extendable API. They wish to analyse large Petri nets without a graphical front-end, something which is just not possible due to the tangled nature of the Petri net logic in PIPE 4’s codebase. Furthermore there has been some level of interest from Google US for PIPE as an analysis tool for natural language processing, but PIPE 4 is a long way away from being usable by such a large profile customer.

In order for PIPE to remain and progress as a useful tool for analysing Petri nets the development process needs to be modernised and the logic of a Petri net and its analysis

needs to be abstracted from the rest of the code. Without these changes future developers of PIPE will find it incredibly difficult to add new features and analysis modules without continuing to introduce a series of bugs which harm the quality of the product.

## 1.2 Objectives

With the aim to increase the quality of software engineering behind PIPE the objectives we set out for this project were as follows:

- Perform a stringent analysis and critique of the existing codebase to ascertain the state that it was in.
- Perform a complete redesign of PIPE's architecture to introduce a set of Petri net models and re-engineer the existing codebase to be compatible with these new classes.
- Implement new analysis modules by designing algorithms which exploit features of multi-core processors.
- Streamline and modernise the build process of PIPE, including implementing unit and integration testing frameworks.

## 1.3 Contributions

In this dissertation we present PIPE 5, the latest version of the Platform Independent Petri net Editor, which contains an all new back-end and two exciting new parallel analysis modules. The main contributions to this new version are summarized below:

- A quantitative code quality analysis using static code analysers which highlight a total of 12904 code quality issues in PIPE 4's design and 40 cyclic dependencies throughout the project.
- A set of new back-end classes and interfaces for creating, modifying, simulating and analysing a Petri net. This includes an enhanced way to read and write Petri nets and a domain specific language which makes the creation of a Petri net read like an

English sentence. These classes are thoroughly tested via integration and unit tests both on the development machine and a continuous-integration server.

- Integration of these models into the existing PIPE GUI. This required a significant number of changes to the GUI classes and they now support a publish-subscribe model-view-controller architecture.
- The streamlining of PIPE's development process. This has been achieved by migrating from CVS to Git, moving the project from SourceForge to GitHub, setting up a continuous integration server for the new test suite and using Maven as a dependency manager and build system.
- Development and advanced profiling of a new state space exploration algorithm which exploits multi-core processors to gain speedups over 200x the original PIPE 4 algorithm and is up to 4.32x as fast as our re-written sequential algorithm when running on 8 virtual cores. Our algorithm can now process a million state Petri net in under 6 minutes when run with 8 virtual cores.
- Development of a parallel asynchronous Gauss-Seidel steady state solver which is up to 5x as fast as the sequential Gauss-Seidel algorithm when run on 4 virtual cores.

## 1.4 Report structure

Chapter 2 contains an introduction to Markov processes and Petri nets and follows by explaining the theory behind their simulation. It details algorithms for analysing the underlying structure of a Petri net, and takes a brief look into different iterative models for solving the steady state. Finally it describes the Petri Net Markup Language (PNML) standard for reading and writing Petri nets to an XML file.

Chapter 3 describes PIPE 4's use and existing features. Following from this Chapter 4 analyses the architectural design and codebase of PIPE 4 using several static analysis tools. It provides quantitative results and details critical aspects of the functionality and design.

PIPE 5's new build process is shown off in Chapter 5 providing details of the migration to Git, GitHub's most useful features and the introduction of a continuous integration server.

It explains the use of Maven as a build system to provide easy builds and releases.

Details of the new PIPE 5 architecture can be found in Chapter 6. It explains how we addressed and fixed some of the critical problems that were highlighted about PIPE 4 and demonstrates architectural design decisions taken. Following from this Chapter 7 details the restructuring of the existing view code to incorporate these new models into the front-end GUI.

Chapter 8 explains the redesign of the state space exploration and steady state algorithms and details new parallel implementations which exploit multi-core processors.

We evaluate the success of this project in Chapter 9 by providing a testimonial from ICSI, rerunning the quantitative analysis tools on the new architecture and evaluating the the performance results of the new parallel algorithms designed.

Finally Chapter 10 concludes by summarising the project achievements reflecting on the work performed, and detailing areas of future work.

# Chapter 2

## Background

As a research tool, the Petri net is invaluable.

Noam Chomsky (paraphrased)

In this chapter we explore the key knowledge required for understanding and analysing Petri nets. As an isomorphism exists between Petri nets and Markov chains we begin by detailing the theory of a Markov process.

### 2.1 Markov Theory

#### 2.1.1 Stochastic processes

We define a random variable,  $\chi$ , as a variable whose value is the result of a random experiment [6]. It is said to be discrete if the set of possible values for  $\chi$  is countable. We define its probability mass function as:

$$p_{\chi}(x) = P(\chi = x).$$

Likewise if the set of possible values of  $\chi$  is uncountable then we say the random variable



is continuous and define its cumulative distribution function as:

$$F_\chi(x) = P(\chi \leq x).$$

A stochastic process can be defined as a family of random variables  $\{\chi(t)\}$  where each state can take a value belonging to the state space of  $\chi(t)$ . If the state space is discrete we call the process a chain, and write  $\chi_t$  rather than  $\chi(t)$ . Furthermore  $\{\chi(t)\}$  is indexed by time and if the time index set  $\{t\}$  is countable then we call this a discrete time process or chain.

### 2.1.2 Markov process

For a discrete time process the Markov property states that given the current state,  $\chi_t$ , the distribution of any future state,  $\chi_v$ , does not depend on the past history  $\{\chi_u : u < t\}$ . A Markov process is therefore the class of stochastic processes who satisfy the Markov property.

For a discrete time Markov chain (DTMC) the Markov property states for a time index set  $\{t\}$  of counting numbers  $\{0, 1, 2, \dots\}$ :

$$P(\chi_{n+1} = j | \chi_0 = i_0, \dots, \chi_n = i_n) = P(\chi_{n+1} = j | \chi_n = i_n).$$

This evolution can be described by 1 step transition probabilities that make up the characteristic transition probability matrix  $Q$  where:

$$q_{ij} = P(\chi_{n+1} = j | \chi_n = i) \text{ for } i, j \geq 0.$$

Similarly we can define a continuous time Markov chain (CTMC) as a stochastic process that satisfies the Markov property:

$$P(\chi_{t+s} = j | \chi_u = i_u, u \leq t) = P(\chi_{t+s} = j | \chi_t = i_t)$$

where  $\{\chi_{t+s}, \chi_u, \chi_t\} \in \Omega$  and  $\Omega$  is a countable set.

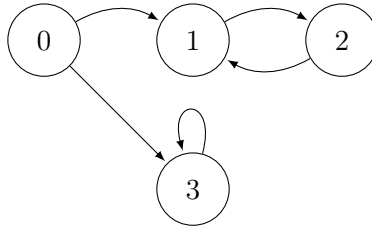


Figure 2.1: A Markov chain in which states 1 and 2 are closed because no other state can be reached from them and state 3 is absorbing.

This evolution can also be described by a generator matrix  $A$  where:

$$a_{ij} = \begin{cases} \mu_i q_{ij} & \text{for } i \neq j, \\ -\mu_i = \sum_{j \in S, j \neq i} a_{ij} & \text{for } i = j \end{cases}$$

where  $\mu_i$  is the rate at which state  $i$  transitions into state  $j$ .

In order to define useful properties on Markov chains we must cover some definitions:

- **Reachable** — if starting in some state  $i$  and following transitions from state to state we arrive at a state  $j$  then  $j$  is said to be reachable from  $i$ .
- **Absorbing states** — these are states which can never be transitioned out of. For example in Figure 2.1 state 3 is an absorbing state.
- **Closed states** — a set of states  $C$  is closed if no state outside of  $C$  can be reached from a state in  $C$ . For example in Figure 2.1 states 1 and 2 are closed.
- **Irreducible** — a Markov chain is irreducible if the set of all states is the only closed set.
- **Recurrent** — if having once been in state  $i$ , the chain returns to  $i$  with probability 1 then the state is said to be recurrent. States which are not recurrent are said to be transient. If all states are recurrent the Markov chain is defined to be recurrent.
- **Periodic** — a state is periodic with period  $m > 1$  if the consecutive returns to a state  $j$  occur at multiples of  $m$  steps. In an irreducible Markov chain all states are periodic or aperiodic.

- **Positive recurrent** — If a state  $j$  has  $m_j < \infty$  and the probability of being in state  $j$ ,  $\pi_j$ , is greater than 0 then it is positive recurrent.

Therefore an irreducible, aperiodic DTMC with state space  $S$  and one step transition matrix  $Q$  is positive recurrent if the equations below have a solution:

$$\pi_j = \sum_{i \in S} \pi_i q_{ij} \text{ for } j \in S \text{ (balance equations)} \quad (2.1)$$

$$\sum_{i \in S} \pi_i = 1 \text{ (normalising equation)}. \quad (2.2)$$

Then  $\pi$  is referred to as the steady state probability of  $\chi$ . Similarly an irreducible, positive-recurrent CTMC has a steady state solution if:

$$\sum_{i \in S} \pi_i a_{ij} = 0 \text{ for } j \in S \text{ (balance equations)}$$

$$\sum_{i \in S} \pi_i = 1 \text{ (normalising equation)}.$$

## 2.2 Petri nets

A Petri net is a directed bipartite graph used to describe and analyse the concurrent processes which arise in distributed systems. They were first invented by Carl Adam Petri in 1965 [7] and have since been expanded on to solve a variety of different problems.

### 2.2.1 Anatomy

The simplest type of Petri nets are place-transition nets. They consist of the following items:

- **Places** — represented by a circle, these describe the states of the system.
- **Tokens** — represented by a black dot in a place; they move from place to place by enabling and firing the transitions.

- **Transitions** — represented by a rectangle, describe a way for tokens to move from place to place.
- **Arcs** — represented by an arrow with a triangular tip between places and transitions. Places must be connected to transitions and vice versa. Arcs that connect a place to a transition are known as inbound, and those that connect a transition to a place are known as outbound. The arcs can have token weights associated with them, if there is no weight then the default is assumed to be 1.
- **Inhibitor Arcs** — represented by an arrow with a circular tip. These must start at places and end at transitions, they have no associated weight.

Formally we can define place-transition nets as a 5-tupule  $PN = (P, T, I^-, I^+, M_0)$  [8] where:

- $P = \{p_1, \dots, p_n\}$  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 respectively and specify the connection between places and transitions based on arc weights. If  $I^-(p, t) > 0$ , an arc connects place  $p$  to transition  $t$  and vice versa with  $I^+$ .
- $M_0 : P \rightarrow \mathbb{N}_0$  is the initial marking and depicts the number of tokens in each place.

Using this notation we can define the place-transition net depicted in Figure 2.2 as follows:  $PN = (P, T, I^-, I^+, M_0)$  and,

- $P = \{p_1, p_2, p_3, p_4, p_5\}$
- $T = \{t_1, t_2, t_3\}$
- $I^-(p_1, t_1) = 1, I^-(p_3, t_2) = 1, I^-(p_4, t_2) = 2$ . All other values of function  $I^-$  are 0.
- $I^+(t_1, p_2) = 1, I^+(t_1, p_3) = 1$ . All other values of function  $I^+$  are 0.

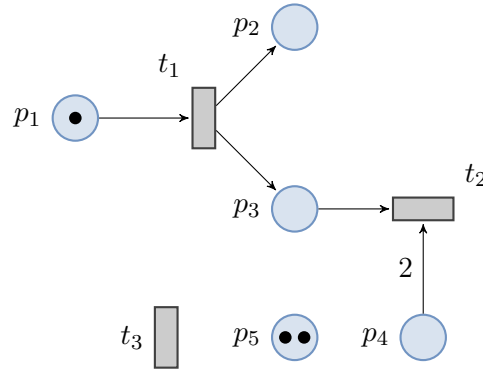


Figure 2.2: An example place-transition net.

$$\bullet M_0(p) = \begin{cases} 1 & \text{if } p = p_1, \\ 2 & \text{if } p = p_5, \\ 0 & \text{otherwise.} \end{cases}$$

It follows from this that given a place-transition net  $PN = (P, T, I^-, I^+, M_0)$ , we can define [8]:

- Input places of a transition  $t$  (preset of  $t$ ):  $\bullet t = \{p \in P \mid I^-(p, t) > 0\}$ ,
- Output places of a transition  $t$  (postset of  $t$ ):  $t \bullet = \{p \in P \mid I^+(t, p) > 0\}$ ,
- Input transitions of a place  $p$  (preset of  $p$ ):  $\bullet p = \{t \in T \mid I^+(t, p) > 0\}$  and
- Output transitions of a place  $p$  (postset of  $p$ ):  $p \bullet = \{t \in T \mid I^-(p, t) > 0\}$ .

We can also extend this to sets  $X \subseteq P \cup T$  where  $\bullet X = \cup_{x \in X} \bullet x$  and  $X \bullet = \cup_{x \in X} x \bullet$ . For example the place-transition net in Figure 2.2 yields  $\bullet p_1 = \emptyset$ ,  $\bullet p_2 = \{t_1\}$ ,  $\bullet \{t_1, t_2\} = \{p_1, p_3, p_4\}$ ,  $\{t_1, p_4\} \bullet = \{p_2, p_3, t_2\}$ , and so on.

### 2.2.2 Enabling and firing

A transition is enabled if all the input places are marked with at least as many tokens as specified by their connecting arc weight. It is this enabling that gives Petri nets their notion of synchronisation. We define the enabling degree of a transition  $t$  as the number

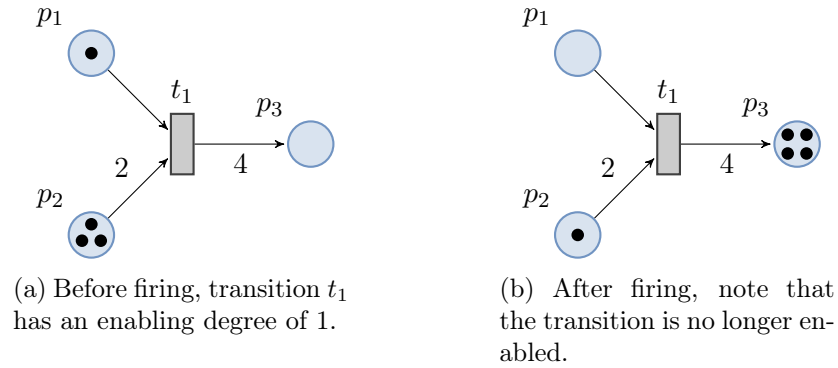


Figure 2.3: An example Petri net before and after firing  $t_1$ . Observe that to fire  $t_1$  requires 1 token from  $p_1$  and 2 tokens from  $p_2$ . When firing it removes these tokens from the places and puts 4 new tokens into  $p_3$ .

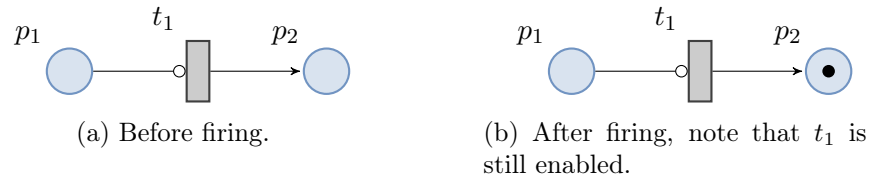


Figure 2.4: An example Petri net with an inhibitor arc before and after firing. If  $p_1$  had contained a token then  $t_1$  would never be enabled.

of times  $t$  can fire given the current marking [9]. Any enabled transition is eligible to fire.

When a transition fires it removes the number of tokens specified by the inbound arc from each input place and creates the relevant number of tokens in any places outbound arcs connect to. An example of firing can be seen in Figure 2.3. We define two transitions to be in conflict if the firing of a transition reduces the others enabling degree to 0 [9]. Conflicts in a Petri net represent choice of the system being analysed.

Inhibitor arcs are a special type of inbound arc that only enable a transition if their place contains no tokens. A firing example can be seen in Figure 2.4.

We can define the enabling and firing behaviour formally on a place-transition net  $PN = (P, T, I^-, I^+, M_0)$  as follows [8]:

- A marking of a place-transition net is a function  $M : P \mapsto \mathbb{N}_0$ , where  $M(p)$  denotes

the number of tokens in  $p$ .

- A set  $\tilde{P} \subseteq P$  is marked at a marking  $M$ , iff  $\exists p \in \tilde{P} : M(p) > 0$ ; otherwise  $\tilde{P}$  is unmarked or empty at  $M$ .
- A transition  $t \in T$  is enabled at  $M$ , denoted by  $M[t >$ , iff  $M(p) \geq I^-(p, t), \forall p \in P$ .
- A transition  $t \in T$ , enabled at marking  $M$ , may fire yielding a new marking  $M'$  where:

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

denoted by  $M[t > M'$ . We say  $M'$  is directly reachable from  $M$  and write  $M \rightarrow M'$ . Furthermore define  $\rightarrow^*$  to be the reflexive and transitive closure of  $\rightarrow$ . A marking  $M'$  is said to be reachable from  $M$  iff  $M \rightarrow^* M'$ .

- A firing sequence of  $PN$  is a finite sequence of transitions  $\sigma = t_1, \dots, t_n$  for  $n \geq 0$ , such that there are markings  $M_1, \dots, M_{n+1}$  satisfying  $M_i[t_i > M_{i+1}, \forall i = 1, \dots, n$ .

### 2.2.3 Coloured Petri nets

Coloured Petri nets are an extension of place-transition nets, whereby more than one token type can be defined in the Petri net. Each token represents a different data value and is given a different colour. This allows more complex real life problems to be modelled.

Formally a coloured Petri net,  $CPN = (P, T, C, I^-, I^+, M_0)$ , extends place-transition nets in the following way [8]:

- $P$  and  $T$  are defined as per Section 2.2.1.
- $C$  is a colour function defined from  $P \cup T$  into finite and non-empty sets.
- $I^-$  and  $I^+$  are the backward and forwards incidence sets defined in Section 2.2.1, except now:

$$I^-(p, t), I^+(p, t) \in [C(t) \rightarrow C(p)_{MS}] \forall (p, t) \in P \times T.$$

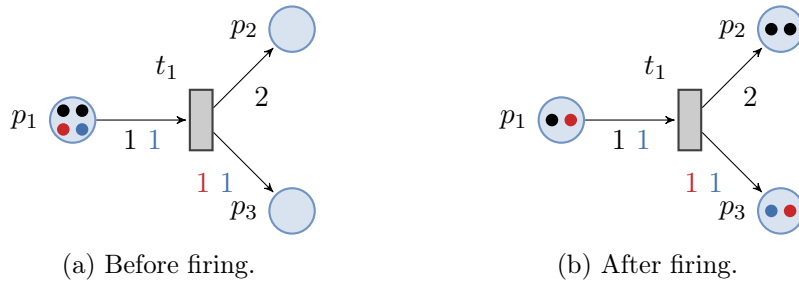


Figure 2.5: A coloured Petri net before and after firing transition  $t_1$ . In order for  $t_1$  to be enabled it requires that  $p_1$  contains at least 1 black token and 1 blue token with all other colours being irrelevant. On firing  $t_1$  produces 2 black tokens in  $p_2$  and a red and blue token in  $p_3$ .

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

The same properties apply for firing and enabling transitions as in Section 2.2.2 except now an arc can be labelled with more than one colour. If a subset of coloured token weights are specified on an arc then all other coloured weights default to 0. If no colour is specified the default is 1 black token. An example of a coloured Petri net is displayed in Figure 2.5.

## 2.2.4 Stochastic Petri nets

In order to analyse the performance of a distributed system we introduce the notion of time through stochastic Petri nets (SPNs).

In a stochastic Petri net the time in which it takes a transition,  $t$ , to fire,  $\chi$ , is exponentially distributed with rate  $\lambda$ . The set  $\Lambda$  is added to  $PN = (P, T, I^-, I^+, M_0)$  where  $\Lambda = (\lambda_1, \dots, \lambda_n)$  and  $\lambda_i$  is the transition rate of transition  $t_i$  [8]. To distinguish between timed and immediate transitions in a Petri net diagram they are represented as empty and filled rectangles respectively.

If more than one transition is enabled at any given time in the case of Figure 2.6, then the transitions race each other. The one that fires first wins and so removes the tokens from its input places and produces the specified number of tokens in its output places. In



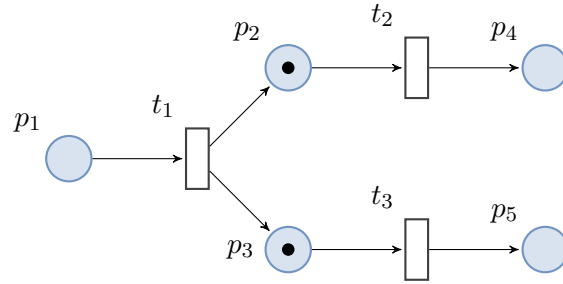


Figure 2.6: A stochastic Petri net with both  $t_1$  and  $t_2$  enabled.

Figure 2.6 the probability that  $t_2$  will fire is given by:

$$\begin{aligned} P[t_2 \text{ fires at } M_0] &= P[\chi_2 < \chi_3] \\ &= \frac{\lambda_2}{\lambda_2 + \lambda_3}. \end{aligned}$$

If a timed transition has an enabling degree greater than 1, then different semantics are possible for the firing [9]:

- **Single-server semantics** — the enabling sets of tokens are processed serially meaning new delays will be generated on firing the transition if it is still enabled.
- **Infinite-server semantics** — every enabling set of tokens is processed as soon it forms in the input place. The firing delay is calculated at the time of arrival so the the timers for each of these tokens will run down to zero in parallel.
- **Multiple-server semantics** — enabling sets of tokens are processed as soon as they form in the input places of the transition up to a maximum degree of parallelism. If the enabling degree is greater than the degree of parallelism new enabling sets of tokens will only be set when the number of concurrently running timers decreases below this value.

To summarise this more clearly these three semantics are best detailed with an example relating to Figure 2.7, where the activity firings will take 3, 2 and 4 seconds [9]:

- **Single-server semantics:**

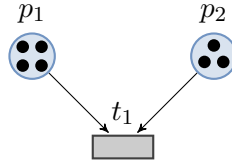


Figure 2.7: A timed transition with enabling degree 3.

1. At time 0  $t_1$  is enabled and the first activity starts.
2. At time 3 the first activity ends, taking three seconds;  $t_1$  fires and the second activity starts.
3. At time 5 the second activity ends, taking two seconds;  $t_1$  fires and the third activity starts.
4. At time 9 the third activity ends after four seconds;  $t_1$  is disabled.

- **Infinite-server semantics:**

1. At time 0  $t_1$  is enabled and the activity starts.
2. At time 2  $t_1$  fires because of the completion of the second activity.
3. At time 3  $t_1$  fires because of the completion of the first activity.
4. At time 4  $t_1$  fires because of the completion of the third activity;  $t_1$  is now disabled.

- **Multiple-server semantics:** Here we assume the degree of parallelism is 2.

1. At time 0  $t_1$  is enabled and the first two activities start.
2. At time 2  $t_1$  fires because of the completion of the second activity. The third activity is now allowed to start.
3. At time 3  $t_1$  fires because of the completion of the first activity.
4. At time 6  $t_1$  fires because of the completion of the third activity;  $t_1$  is now disabled.

Generalised stochastic Petri nets (GSPN's) are another extension of place-transition nets

allowing for both timed and immediate transitions. Immediate transitions will always fire before timed transitions, so in the case where both could be enabled, an immediate transition will always take priority.

### 2.2.5 Other extensions

Other extensions of Petri nets are as follows:

- Combining the rules of coloured Petri nets and generalised stochastic Petri nets into a coloured generalised stochastic Petri net.
- Giving immediate transitions priorities. This places an ordering on which enabled immediate transition can fire.
- Giving places a capacity. This stops an enabled transition being able to fire if any of its outbound places are already at maximum capacity.
- Adding functional expressions in the form of arc weights and transition rates. These allow the weights and rates to alter based on properties of individual places and are particularly important in Petri net design as they allow more expressive models to be created. For example if we wish to simulate the action of clearing all items in a buffer then it is very simple with a functional expression; the arcs weight is set to  $\#(p_0)$  which will always remove every token from  $p_0$  when the transition is fired.

## 2.3 Petri net analysis

Petri nets can be analysed in many different ways. In this section we explain the two most common analysis techniques.

### 2.3.1 Exploring the state space

One of the most common ways to analyse a Petri net is to generate its state space and display it in the form of a reachability graph where the nodes contain the markings of the

Petri net. Two nodes  $M$  and  $M'$  will be connected with a directed arc iff  $M[t > M'$  for some  $t \in T$ .

The traditional way to generate the reachability graph is to perform a sequential breadth first search starting with the initial marking of the Petri net and adding directly reachable markings as children.

When exploring the state space states are classified into two types:

1. **Vanishing** — a state is vanishing if any immediate transitions are enabled. Since immediate transitions take no time to fire the vanishing state will instantaneously transition to the next state, meaning that no time is spent in it.
2. **Tangible** — if no immediate transitions are part of the enabled set then a state is said to be tangible, meaning that some time will be spent in the state.

An example of a Petri net's reachability graph can be seen in Figure 2.8.

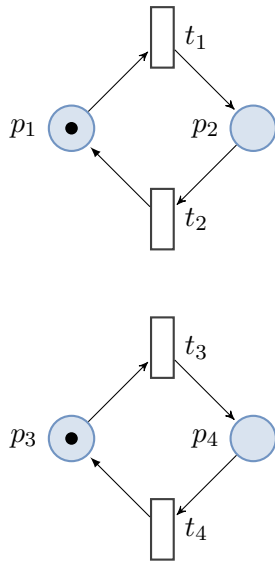
If a Petri net is unbounded, as in the case of Figure 2.9, the generation of its reachability graph will fail because the state space is infinite. To address this problem the symbol  $\omega$  is introduced to represent the marking of an unbounded place. A graph that contains unbounded places is known as a coverability graph and the full algorithm for generating it can be found in [10].

### 2.3.2 Solving the steady state

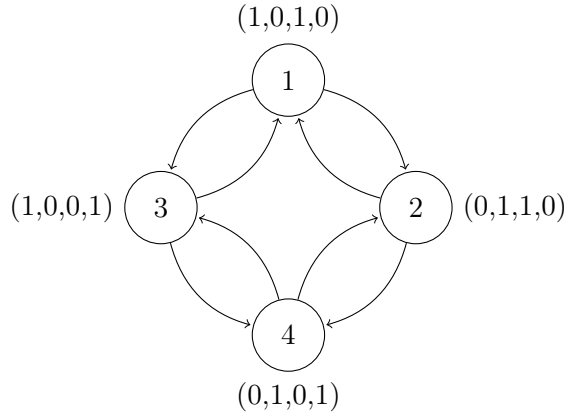
Once we have generated the reachability graph of a Petri net, and thus explored fully its state space, we can solve to find its steady state. The steady state of a Petri net represents proportion of time the system spends in each state at equilibrium. As explained in Section 2.1.2 we can find the steady state distribution,  $\pi$ , for a CTMC by solving the equations of the form:

$$\begin{aligned}\pi A &= 0, \\ \sum_{i \in S} \pi_i &= 1\end{aligned}$$

where  $A$  is the generator matrix.



(a) A 4 state Petri net.



(b) Reachability graph of the Petri net to the left. The states have been numbered and their given markings (i.e. how many tokens each place contains) are shown next to them in the order  $(p_1, p_2, p_3, p_4)$ .

Figure 2.8: An example Petri net reachability graph which represents the 4 possible states of the Petri net on the left.

Similarly for a DTMC we can solve Equation (2.2) in matrix form as:

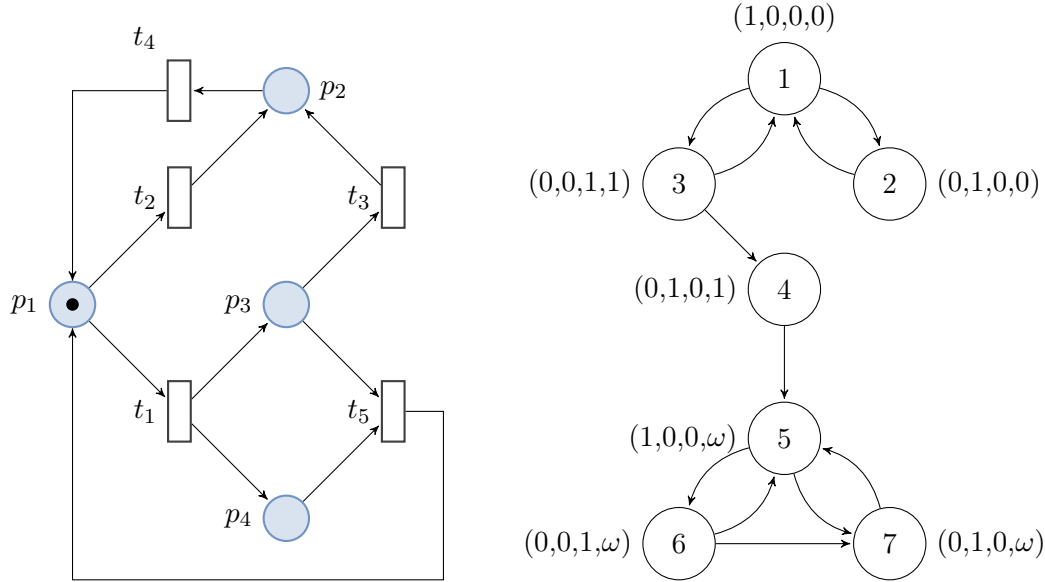
$$\pi Q = \pi. \tag{2.3}$$

Since several standard algorithms for solving linear systems of the form  $Ax = b$  exist the above equation for a CTMC can be rewritten as:

$$A^T \pi^T = 0, \tag{2.4}$$

$$\sum_{i \in S} \pi_i = 1. \tag{2.5}$$

Two different strategies exist to solve linear systems of the form  $Ax = b$ . The first strategy involves direct methods which yield an exact solution to the equations. Gaussian elimina-



(a) A Petri net with an infinite state space. (b) Coverability graph of the Petri net to the left.

Figure 2.9: An example of an unbounded Petri net from [8] whose state space is infinite and its corresponding coverability graph.

tion is one such direct method that solves the set of linear equations through matrix row operations as follows [11]:

1. Reduce the matrix  $A^T$  into an upper triangle matrix through a series of row operations. For the  $k$ th row all elements directly below it are eliminated making  $a_{jk} = 0$  for  $j > k$  by subtracting multiples of the  $k$ th row from each row below.
2. Solve via back substitution.
3. Normalise if desired.

Unfortunately for sparse matrices direct methods suffer from a problem known as matrix fill-in: initial zero elements get set to a non-zero value during the execution of an algorithm. For Gaussian elimination this means that the matrix is no longer sparse and so causes the algorithm to suffer from terrible memory performance.

The second type of algorithms are iterative methods. These start from an initial approx-

imation and iteratively aim to better the solution until a desired accuracy is reached. If convergence is known to be quick, these types of algorithms can often be a lot faster than direct methods and have the added advantage of only requiring space proportional to the size of  $x$ . Since they do not alter the matrix directly there is no risk of fill-in and no computation is needed for zero elements.

The most simple iterative algorithm is known as the Power method [6] and can solve linear equations in the form:

$$Ax = b \text{ or } xA = b.$$

The algorithm repeats the following until convergence:

1.  $x^{n+1} = x^n A$
2. if  $\frac{\|x^{n+1} - x^n\|_2}{\|x^n\|_2} < \epsilon$  then the algorithm has converged and return  $x$ .

For a DTMC this method is guaranteed to converge if the matrix  $Q$  in Equation (2.3) is aperiodic as it mimics the behaviour of the DTMC until convergence is reached. This method is however often slow to converge [12].

Another iterative method is the Jacobi algorithm [11]. At each iteration it makes successive corrections to the approximation of  $x$  via the formula:

$$x_i^k = \frac{1}{a_{ii}} \left( b_i - \sum_{i \neq j} a_{ij} x_j^{(k-1)} \right) \text{ where } k \text{ denotes iteration step } k. \quad (2.6)$$

The Jacobi method is called a method of simultaneous corrections since no element of an approximation  $x^k$  is used until all the elements of  $x^k$  have been generated. This means that the algorithm can be performed in parallel.

The Gauss-Seidel algorithm [11] behaves in exactly the same way as the Jacobi algorithm but uses previously calculated values of  $x^k$  in the same iteration to yield faster convergence

times. That is:

$$x_i^k = \frac{1}{a_{ii}} \left( b_i - \sum_{j<i} a_{ij} x_j^{(k)} - \sum_{j>i} a_{ij} x_j^{(k-1)} \right).$$

### 2.3.3 Performance analysis

Having calculated the state space of a Petri net and solved for its steady state at equilibrium, more meaningful high-level performance measures can be calculated.

For a given performance measure  $m$ , we can calculate expected (mean) metrics using the equation [13]:

$$E[m] = \sum_{i=1}^n \pi_i v_i, \quad (2.7)$$

where  $n$  is the number of states in the system,  $\pi_i$  is the steady state distribution and  $v_i$  is a function of the elements of the  $i$ th state descriptor.

For example if we are interested in the throughput of a particular timed transition,  $j$ , we can apply Equation (2.7) in the following manner [8]:

$$d_j = \sum_{s_i \in EN_j} \pi_i \lambda_j, \quad (2.8)$$

where  $d_j$  represents transition  $j$ 's throughput,  $EN_j$  is the set of all states in which transition  $j$  is enabled and  $\lambda_j$  is the transition rate of  $j$  in state  $s_i$ .

## 2.4 Petri Net Markup Language

The Petri net Markup Language (PNML) was devised as an XML standard so that Petri nets could be exported from one set of tools and into another [14].

Each Petri net file may contain several Petri nets. The naming of the Petri net components as XML elements comes from a predefined set of conventions. The PNML for a Petri net



is shown in Listing A.1 in Appendix A.

The aforementioned Petri net components can optionally have labels to give extra meaning to them. They often represent the name of the object and some other object specific information such as the number of tokens in a place. Labels come in two forms:

- **Annotations** — an unlimited amount of text which will be displayed near the component to which it belongs. For example place names, arc weights and so on.
- **Attributes** — these are domain limited values which are not normally displayed textually. Instead they represent something about the component for example timed or immediate transitions and will affect how component object itself is displayed.

Petri net components and annotations are also allowed graphical information which determines where they are rendered by an application. For places and transitions this is their physical positioning and for an arc it is a list of intermediate points.

## Chapter 3

# PIPE 4 - an open-source Petri net editor

The Petri net is like the phone. To be without it is ridiculous.

Kevin Mitnick (paraphrased)

The Platform Independent Petri net Editor, more commonly known as PIPE, is an open-source tool developed by Imperial College London in 2003 and written in Java for building and analysing Petri nets. Today Petri nets play a substantial part in solving real world problems such as [1–3] and so PIPE has proven to be a very popular tool amassing over 89 000 downloads on SourceForge alone [15].

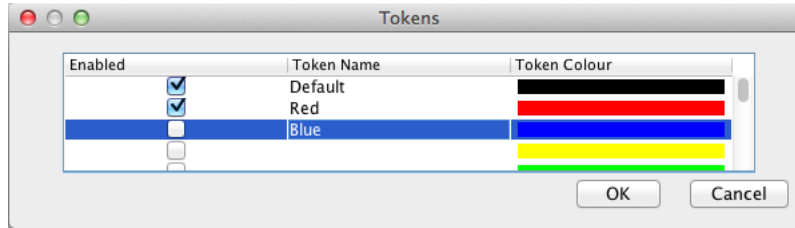
Whilst there were, and still are, many other existing tools for analysing Petri nets available online, PIPE differentiates itself from these by providing an extensive API for writing additional modules. These additional modules allow for further analysis tools to be written and by used anyone at a later date.

Before this project PIPE 4, specifically version 4.3.0, was the latest release of PIPE and throughout the report references to PIPE 4 refer to this release.

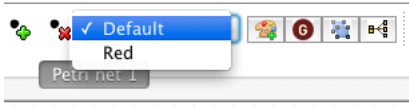
### 3.1 Using PIPE 4's editing features

PIPE 4 supports a variety of features in order to build and edit a Petri net which are labelled in Figure 3.2 and explained below:

- Petri net component creator actions — places, timed and immediate transitions, normal and inhibitor arcs and annotations can be added to the Petri net by selecting one of these actions from the tool bar and clicking on the canvas.
- Token editing actions — PIPE supports coloured GSPN's by allowing multiple tokens to be defined (Figure 3.1). These tokens can be added to places by the use of the add and remove token actions.
- Rate parameter action — PIPE has been extended to support rate parameters as a new feature. A rate parameter is a shared rate that can be referenced by multiple transitions that only needs changing in one place.
- Component editing actions — PIPE supports cut, copy, paste, delete, undo and redo actions to help smooth the build process of a Petri net.
- File editing actions — Petri nets can be loaded from PNML and saved to PNML format. New Petri nets can be created which show up as tabs above the canvas.
- Zoom features — In order to view large and small Petri nets PIPE supports zooming in and out. To help view large Petri nets canvas becomes scrollable when the Petri net no longer fits in the window.



(a) Token Editor that allows the user to create and edit coloured tokens.



(b) Enabled Tokens are displayed in the token drop down on the tool bar for selection. The selected token is the one that will get added to places when using the add and remove token actions.

Figure 3.1: Token editing forms

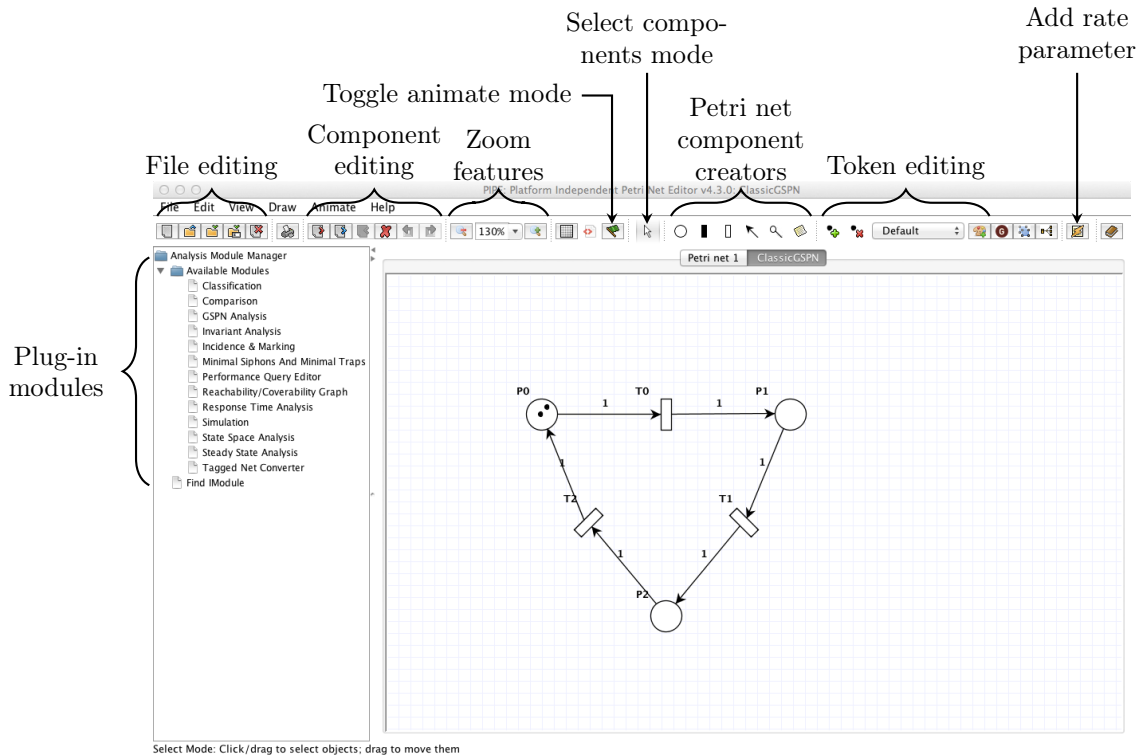


Figure 3.2: A labelled diagram of PIPE 4 in editing mode.

## 3.2 Editing a Petri net

In order to edit existing Petri net components PIPE supports some other useful features described below.

### 3.2.1 Petri net component menus

All Petri net components that are displayed on the canvas have an edit menu that can be accessed by right clicking on the component.

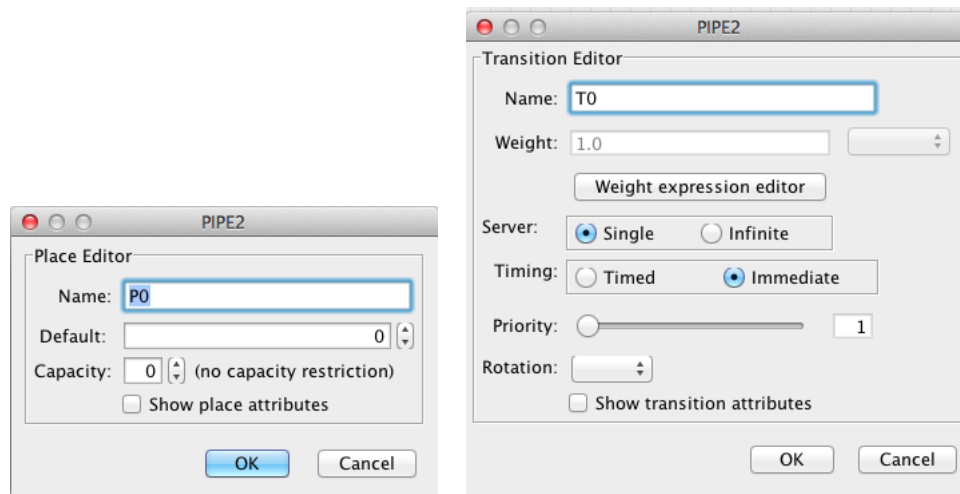
Both places and transitions can be renamed via their menu as seen in Figure 3.3. The menus also allow the user to change individual specified properties of the component, such as the number of tokens in a place, the server semantics of a transition and so on.

### 3.2.2 Functional expressions

In 2013 functional expressions were added to PIPE 4 to give users the ability to create more expressive Petri nets [16]. Functional expressions allow arc weights, transition rates and rate parameters to refer to other Petri net components dynamically when animating.

The supported operations in functional weights are:

- Simple maths, e.g. addition, subtraction, multiplication, division.
- Ceiling and floor mathematical operations.
- References to places. By placing the component id between a hash and two brackets, `#(...)`, the number of tokens in the place can be referenced. For transitions this refers to the total number of tokens in the place; for arc weights it refers to the number of tokens in the place matching the particular coloured token whose weight is being described. It is also possible to refer to the capacity of a place by placing its component id between `cap(...)`.



(a) Place editing menu.

(b) Transition editing menu.

Figure 3.3: PIPE 4 component editors for places and transitions.

### 3.3 Animation

Once a Petri net has been imported or built it may be animated by pressing the green flag icon on the tool bar.

#### 3.3.1 Firing transitions

In animation mode enabled transitions are highlighted in red. PIPE provides three ways to fire them:

- Double click on an enabled transition to fire it.
- Click on the single lightning bolt icon to fire a random transition.
- Click on the multiple lightning bolt icon to fire a specified number of random transitions.

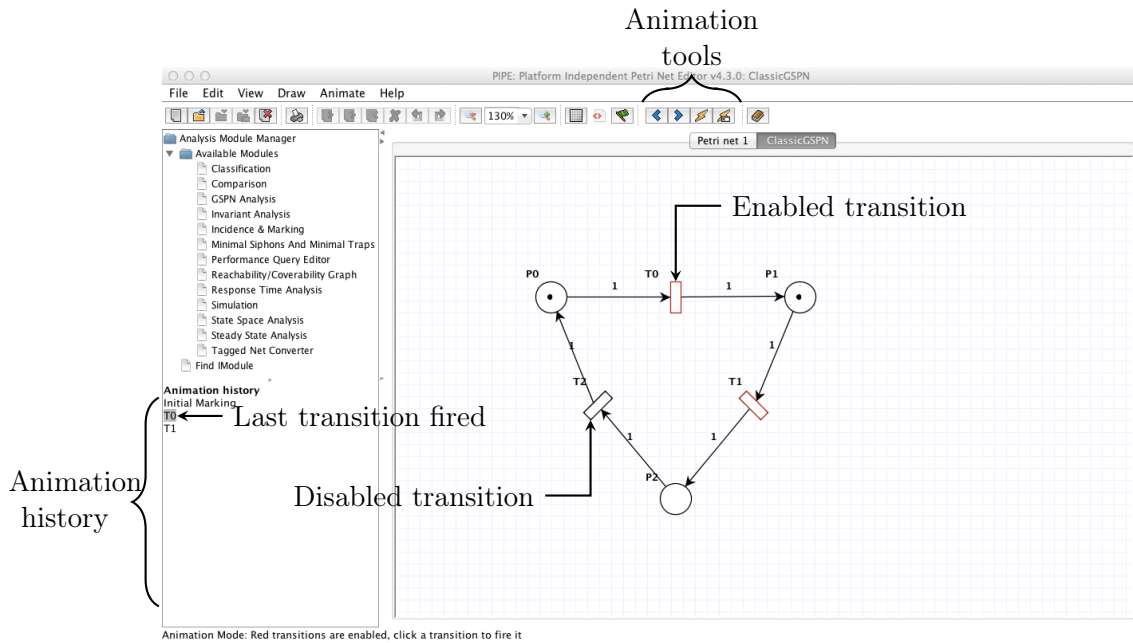


Figure 3.4: Animation mode in PIPE 4. The enabled transitions are shown in red and the animation history can be used to step forwards and backwards throughout a firing sequence.

### 3.3.2 Animation history

The animation history of PIPE can be stepped through via the left and right arrows in the animation tool bar shown in Figure 3.4. The firing sequence history is displayed on the left hand side of the application.

## 3.4 Analysis modules

The user may choose to analyse their Petri net further by using any of the available modules listed on the left hand side of the GUI as seen in Figure 3.2.

PIPE 4 supports the following plug-in analysis modules:

- **Classification** — this classifies a Petri net into one or more of the following types: State Machine, Marked Graph, Free Choice Net, Extended Free Choice Net, Simple

Net and Extended Simple Net.

- **Comparison** — this compares two Petri nets and establishes if they are identical or similar. It provides options for choosing comparison fields such as component ids, place token counts and so on. It displays any differences to the user in its output.
- **GSPN Analysis** — performs steady state analysis on a GSPN providing information about the tangible states, the steady state distribution, the average number of tokens on a place and so on.
- **Invariant Analysis** — calculates P-invariants and T-invariants to determine if the Petri net is bounded (its reachability set is finite) and live.
- **Incidence & Marking** — calculates the forward and backward incidence matrices of the Petri net.
- **Minimal Siphons And Minimal Traps** — finds the non-empty set of places  $P' \subset P$  whose successors are also predecessors:  $P'\bullet \subseteq \bullet P'$  and the non-empty set of places  $P' \subset P$  whose predecessors are also successors:  $\bullet P' \subseteq P'\bullet$  [17].
- **Performance Query Editor** — analyses the Petri net through a graphical mathematical performance query language described in [18].
- **Reachability/Coverability Graph** — generates the reachability or coverability graph of a Petri net.
- **Response Time Analysis** — performs response time analysis queries on the Camelot cluster at Imperial College London [19].
- **Simulation** - calculates the average number of tokens in each place for a given number of firings and replications.
- **State Space Analysis** — determines if the Petri net is bounded, safe, and free of deadlocks.
- **Steady State Analysis** — analyses the reachability set and calculates the steady state of the Petri net
- **Tagged Net Converter** — tags a token and tracks its position as it moves around a Petri net [20].



## Chapter 4

# Analysing the existing PIPE 4 software architecture

It isn't the original scandal that gets people in the most trouble - it's the attempted cover-up.

Tom Petri

This section details the architecture of the code contained in the PIPE v4.3.0 release. In order to reproduce any of the results in this chapter or to view the code which was analysed, the settings and steps taken can be found at <https://github.com/sarahtattersall/PIPEThesisSettings>.

### 4.1 Model–view–controller structure

The model–view–controller (MVC) architecture is a commonly used design pattern for GUIs that aims to decouple the logic of models and their displayed format. It is broken down into three components [21]:

1. **Model** — the application object that contains all of the data represented.
2. **View** — the screen representation of the model data.

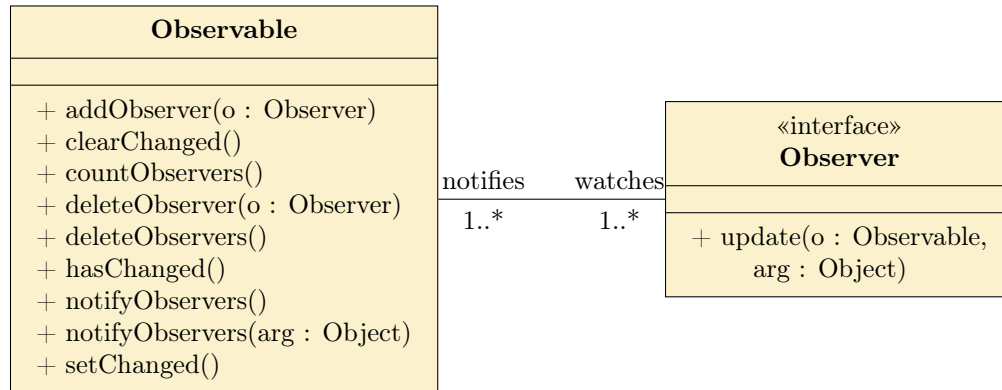


Figure 4.1: UML diagram for the observer design pattern in Java. Unfortunately the Java API for the observer design pattern was implemented as a set of classes rather than interfaces, meaning that to use this pattern you must extend the `Observable` class. The downfall of this is that Java does not support multi-inheritance and so using this pattern only allows you to extend this class. It is therefore considered a broken implementation in Java and it is generally recommended to avoid it [22].

### 3. **Controller** — processes and responds to user events by updating the models.

A MVC architecture decouples the logic of the models from the views through the use of the observer design pattern. In Java this pattern allows classes to implement the `Observer` interface and extend the `Observable` class which can be seen in Figure 4.1. The observer can register itself to observable objects, which in turn notify registered observers when the underlying data structure changes. This ensures that views are up to date with the latest model information and de-couples back-end computational logic from the view display code. Additionally it means that multiple different views for a given model can be created without the need to re-write the back-end representation. Moreover it allows for changes to the model to affect any number of observers without the changed object needing to know anything about them.

Splitting of the view and controller is an important concept too as it allows the developer to change the way that a view responds to user input without actually changing its visual representation. Controllers may be swapped in and out, and in the same way views may also be exchanged whilst keeping the same controller. For example keeping the views and controllers separate could allow for a Swing based implementation and a command-line implementation.

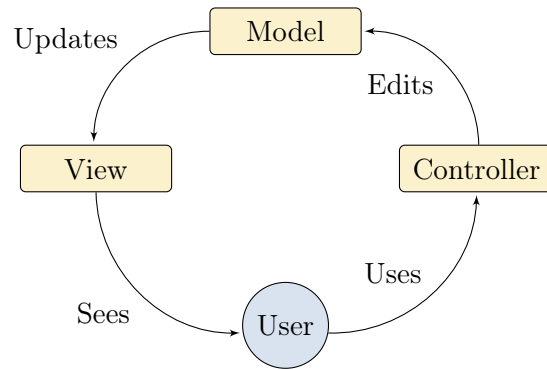


Figure 4.2: The model–view–controller design pattern shown graphically. The user interacts with the controller, which in turn updates the model. This is then observed by the view which updates its graphical display accordingly. This diagram shows that in the MVC design pattern the views should not be interacting with the models directly.

PIPE claims to support the MVC architecture in its documentation [23] but after inspecting the earliest available source code of PIPE2 v0.1 it appears that this architecture never existed [24]. In the code each `PetiNetObject` extends the Swing `JComponent` class. This design choice makes it impossible for other developers to use these classes for their model logic without polluting their codebase with Swing components.

## 4.2 Plug in modules

PIPE supports many different analysis modules for Petri nets as detailed in Section 3.4 and a description of their architecture can be found in this section.

### 4.2.1 The `IModule` interface

In order to support pluggable modules PIPE 4 contains an `IModule` interface that any analysis modules must implement. The current API can be seen in Figure 4.3.

Initially in 2003 when PIPE was developed this API was a little different with the `start` method taking the Petri net model (then called `PNMLData` which housed the Petri net components) as a parameter [23]. Now in PIPE 4 the `PetiNetView` is obtained via a static class

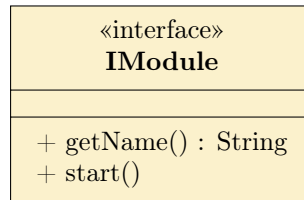


Figure 4.3: The interface required by all PIPE 4 modules. Any class that implements this interface will automatically be detected and loaded into PIPE 4 so that it can be available for analysis. Sadly this API is badly designed because the `start` method does not take a Petri net as a parameter; instead it relies on the `ApplicationSettings` class to provide the Petri net statically.

called `ApplicationSettings`. The purpose of this class is to bind the `PipeApplicationView`, `PipeApplicationModel`, and `PipeApplicationController` at run-time to make them accessible anywhere. Sadly the introduction of this static class makes it impossible to perform Petri net analysis without using the GUI since the Petri net is accessed via the following call:

```
ApplicationSettings.getApplicationView().getCurrentPetriNetView()
```

which tangles the Petri net up in the view code.

The loading of these modules has largely stayed the same since 2003 and uses the Java Reflection API to provide run-time information about classes that are located in the module package [23]. This approach has been taken to avoid explicitly listing modules in a set-up file. Once loaded the classes are stored in a Swing `TreeNode` and their `start` method is invoked using reflection. Whilst dynamically loading the modules is a nice feature, the classes used for the implementation are disjointed and difficult to use.

Once the `start` method of a module has been invoked it is left to display its own GUI and perform its specific analysis on a Petri net. Since many modules have a similar layout the GUI code has been shared where possible [25].

Unfortunately the current implementation of the module system does not allow the modules

to be used independently of the PIPE GUI, something that many people are keen on [26]. Once again the logic for analysing Petri nets is also tied up in the code for displaying the results.

### 4.2.2 The state space exploration algorithm

PIPE 4 supports the generation of a reachability graph via the ‘Reachability/Coverability Graph’ module. The underlying code for the generation of these graphs uses a sequential dynamic-probabilistic hashing algorithm developed by William Knottenbelt in his MSc. thesis [27] and subsequently improved on in his PhD thesis [13]. Algorithm 1 shows how it performs a breadth first search on the state space and uses dynamic storage allocation and probabilistic hashing to yield good memory efficiency.

#### 4.2.2.1 Hashing of states

The idea behind the dynamic-probabilistic hashing algorithm is that once visited, the only information needed about a processed state in the future is that it has already been explored. Therefore after processing a state its full representation can be written to disk for future analysis and a compressed integer representation of it stored in memory. Then when processing a states successors we can quickly ascertain if any have been explored before by converting them into their compressed representation and querying the explored set data structure. If the successor has been seen before it does not need to be added to the exploration queue and only the transition from the current state will be recorded.

In order to implement compressing of a given state, two independent uniformly distributed hash functions should be used known as the primary hash function,  $h_1$ , and the secondary hash function,  $h_2$  [13]. These are used in conjunction with an array in the following manner to compress a state  $s$ :

```

/* initialise  $Q_T$  with initial tangible states */;
if  $s_i \in T$  then
  |  $push(Q_T, s_i)$ ;
  |  $E = \{s_i\}$ ;
else
  |  $push(Q_V, \langle s_i, 1.0 \rangle)$ ;
  | while  $Q_V$  not empty do
  |   |  $pop(Q_V, \langle v, p \rangle)$ ;
  |   | for  $v' \in succ(v)$  do
  |   |   | if  $v' \in T$  then
  |   |   |   |  $push(Q_T, v')$ ;
  |   |   |   |  $E = E \cup \{v'\}$ 
  |   |   | else
  |   |   |   |  $p' = p * prob(v, v')$ ;
  |   |   |   | if  $p' > \epsilon$  then
  |   |   |   |   |  $push(Q_V, \langle v', p' \rangle)$ ;
  |   |   |   | end
  |   |   | end
  |   | end
  | end
end
end
/* perform state space exploration, eliminating vanishing states */;
while  $Q_T$  not empty do
  |  $pop(Q_T, s)$ ;
  | for  $s' \in succ(s)$  do
  |   | if  $s' \in T$  then
  |   |   |  $transition(s, s', rate(s, s'))$ ;
  |   |   | if  $s' \notin E$  then
  |   |   |   |  $push(Q_T, s')$ ;
  |   |   |   |  $E = E \cup \{s'\}$ ;
  |   |   | end
  |   | else
  |   |   |  $push(Q_V, \langle s', rate(s, s') \rangle)$ ;
  |   |   | while  $Q_V$  not empty do
  |   |   |   |  $pop(Q_V, \langle v, p \rangle)$ ;
  |   |   |   | for  $v' \in succ(v)$  do
  |   |   |   |   |  $p' = p * prob(v, v')$ ;
  |   |   |   |   | if  $v' \in T$  then
  |   |   |   |   |   | if  $v' \notin E$  then
  |   |   |   |   |   |   |  $push(Q_T, v')$ ;
  |   |   |   |   |   |   |  $E = E \cup \{v'\}$ 
  |   |   |   |   |   | end
  |   |   |   |   |   | end
  |   |   |   |   |   |  $transition(s, v', p)$ ;
  |   |   |   |   | else
  |   |   |   |   |   | if  $p' > \epsilon$  then
  |   |   |   |   |   |   |  $push(Q_V, \langle v', p' \rangle)$ ;
  |   |   |   |   |   | end
  |   |   |   |   | end
  |   |   |   | end
  |   |   | end
  |   | end
  | end
end
end
end

```

**Algorithm 1:** The dynamic-probabilistic hashing state space exploration algorithm developed in [27] with on the fly vanishing state removal.

1. The primary key  $h_1(s)$  is calculated and determines the row of the array that holds the state. Since the hash could be larger than the size of the array the row is taken to be  $h_1(state) \bmod n$ , where  $n$  is the size of the array.
2. The secondary key  $h_2(s)$  is calculated and compared to the keys stored in the row already.
3. If the secondary key is already present then the state is deemed to be “explored” and will not be added again to the row. Otherwise the secondary key is added to the row.

This is illustrated diagrammatically in Figure 4.4.

In order to determine if a state is contained in the array, and thus already explored, the following scheme is performed:

1. Again the primary key  $h_1(s)$  is used to determine the row of the array that potentially holds the state.
2. The secondary key  $h_2(s)$  is calculated and compared to the keys stored in the row already. If  $h_2(s)$  is equal to any of the values already stored then the state is deemed to be explored.

Note it is possible for two states to have the same primary or secondary hash code however in this algorithm two states,  $s$  and  $s'$ , are considered equal if and only if  $h_1(s) = h_1(s')$  and  $h_2(s) = h_2(s')$ . This greatly reduces the probability that a state is not explored because it reduces collisions in the explored set array. It has been proved in [13] that if the primary and secondary hash functions distribute states randomly and independently the probability of false positives is sufficiently low, allowing this process to be used as a memory reduction technique.

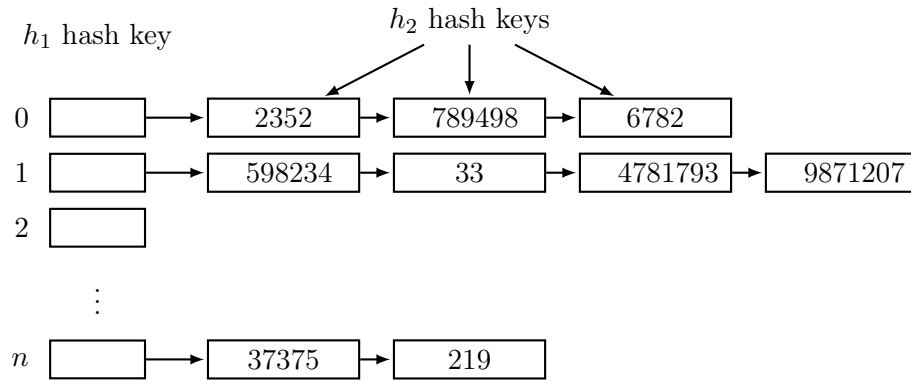


Figure 4.4: A hashing implementation of state compression used whilst exploring the state space. The primary hash key, generated with  $h_1$ , maps to an index in an array containing a linked list of secondary hash keys generated with  $h_2$ . This structure can be queried using primary and secondary hash keys of a state to determine if it has already been seen. States secondary hash keys can also be appended to the relevant linked list to build up a representation of compressed seen states.

#### 4.2.2.2 On the fly vanishing state elimination

Algorithm 1 also performs on the fly vanishing state elimination to stop vanishing states from entering the exploration queue. This algorithm, explained in [13], mathematically calculates the tangible states reachable from a given vanishing state along with their transition rates.

### 4.2.3 The implementation of the state space exploration

#### 4.2.3.1 Exploring the Java code

Although the implementation in PIPE 4 uses the algorithm just described it does so in a convoluted manner. It contains two overridden `generate` methods which both appear to calculate the state space with no indication of what is different between the two. These methods contain a lot of duplicate code between them and do not use many auxiliary



```

hash_key = 0;
for offset ∈ (0 ... no_places] do
  hash_key = 2 * hash_key;
  for place_index ∈ (0... (no_places - offset)] do
    | hash_key = hash_key + state(place_index);
  end
  if hash_key < 0 then
    | hash_key = MAX_INTEGER + hash_key;
  end
end

```

**Algorithm 2:** Primary hash function,  $h_1$ , implemented in PIPE 4 for the explored set data structure. This data structure acts as a memory reduction technique whereby two states,  $s$  and  $s'$ , are considered equal if  $h_1(s) = h_1(s')$  and  $h_2(s) = h_2(s')$ .

```

hash_key = 0;
for offset ∈ (0 ... no_places] do
  hash_key = 2 * hash_key;
  for place_index ∈ (offset... no_places] do
    | hash_key = hash_key + state(place_index);
  end
end

```

**Algorithm 3:** Secondary hash function,  $h_2$ , implemented in PIPE 4 for the explored set data structure. This data structure acts as a memory reduction technique whereby two states,  $s$  and  $s'$ , are considered equal if  $h_1(s) = h_1(s')$  and  $h_2(s) = h_2(s')$ .

methods. This and the fact that the class is 1344 lines long make it very complicated to understand. Furthermore both generate implementations have many unused fields and variables, no useful comments and contain no unit tests.

To implement state compression PIPE 4 uses hand written primary and secondary hash functions that can be seen in Algorithm 2 and Algorithm 3. These functions do not yield a uniform distribution since their implementation is heavily tied to the number of places in the Petri net and the number of tokens in each place. As the number of places remains constant throughout state space exploration, hash codes are likely to fall within a small distribution of the 32-bit integer spectrum.

When running the state space exploration algorithm PIPE 4 reports to the user the time it took to explore the state space and shows an interactive graph that can be seen in Figure 4.5. Sadly when saving the results only the timing information is saved to a HTML file; there is no useful information saved about the state space, for example the number of transitions and states. This means there is no way to reload previously calculated results into the UI; once exited the only option to view the reachability graph again is to re-run the algorithm.

Furthermore PIPE 4's support for unbounded state spaces is implemented by waiting for an `OutOfMemoryException` to be thrown. It then catches this exception and calculates the coverability graph. On modern machines this can take an unacceptable amount of time.

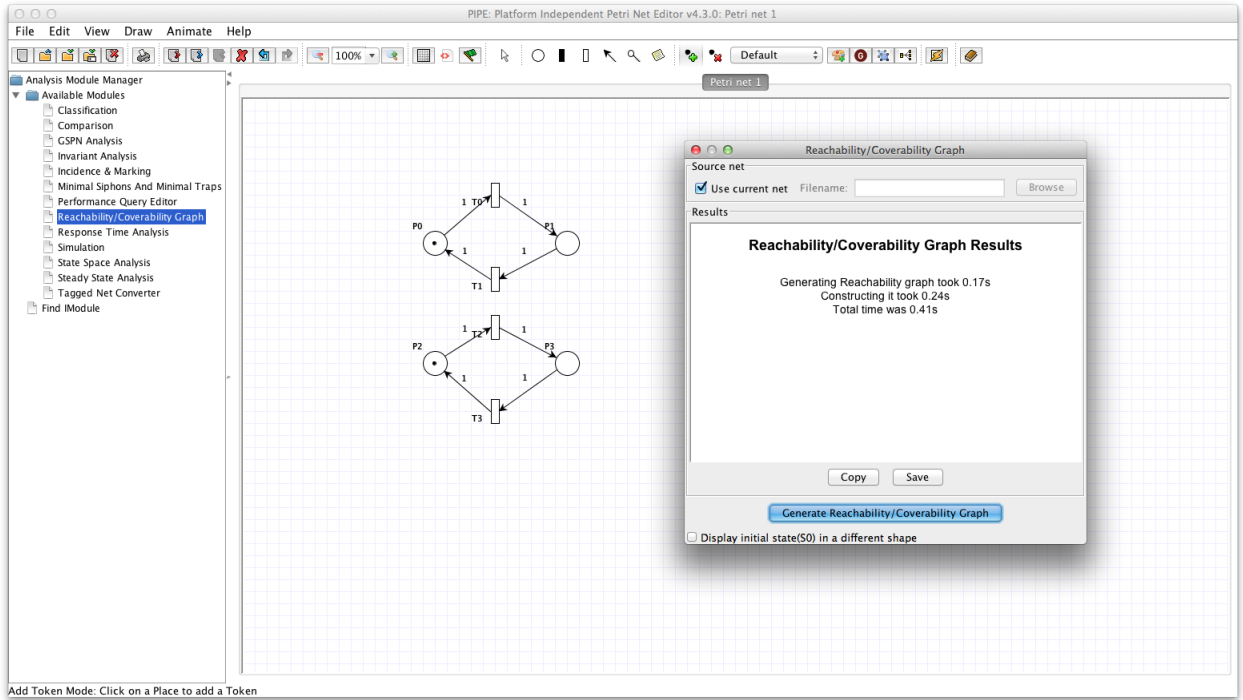
Finally the module cannot be run with coloured Petri nets and so must first convert them into a single token Petri net using an unfolding algorithm [8]. Unfortunately there are bugs in this algorithm, for example when unfolding inhibitor arcs they get converted to normal arcs. This will then yield wrong results when the state space is explored.

#### 4.2.3.2 Speed and performance

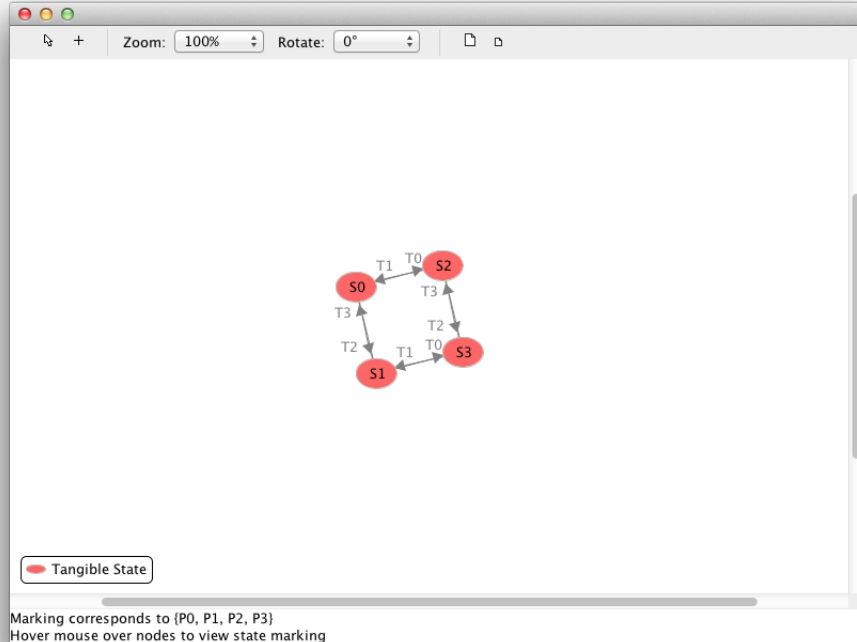
A speed analysis was performed on a 2013 MacBook Air with a dual-core hyper-threaded 2GHz 3rd generation i7 processor and 8GB of RAM. The maximum heap size of PIPE 4 was set to 1GB and the results in Table 4.1 show that the generation of the reachability graph suffers from poor speed performance for all sized state spaces.

#### 4.2.4 Steady state at equilibrium

PIPE 4 can calculate a Petri net's steady state at equilibrium by solving Equation (2.4) sequentially via the Gauss-Seidel method.



(a) Timing results displayed in PIPE 4 to the user once the state space has been explored.



(b) Reachability graph displayed to the user in PIPE 4 once the state space has been explored. The graph is displayed in a new window.

Figure 4.5: PIPE 4 reachability module results for the Petri net depicted in Figure 2.8.

Number of states	Number of transitions	Time (s)
40	156	0.21
100	480	0.36
625	4000	25.12
1350	9450	83.67
4096	28672	728.02
11664	93312	2738.37

Table 4.1: The time taken in seconds to perform the state space exploration for various sized Petri nets in PIPE 4. These results were produced by timing the `StateSpaceExplorer.generate` method using a 2013 MacBook Air with a dual-core hyper-threaded i7 processor and 8GB of RAM. The heap size was set to 1GB. We can see that PIPE 4 quickly becomes infeasibly slow for larger state spaces, taking over 12 minutes to explore a Petri net with 4096 states and over 45 minutes to explore the Petri net with 11664 states.

Not only are there are two classes, `SteadyStateSolver` and `NewSteadyStateSolver`, in PIPE 4 that both solve the steady state via the Gauss-Seidel algorithm, but there are two modules that display the results of the steady state, the GSPN analysis module and the steady state module. This duplication is unnecessary as the steady state module only displays a subset of the GSPN results. Both modules and algorithms suffer from many of the same problems described in Section 4.2.3.1.

An example of the results of the steady state module for the Petri net in Figure 2.8 can be seen in Figure 4.6. This time when saving the results the steady state distribution does appear in a HTML file, but again there is no way to preload this.

Unfortunately the size of a Petri net that can be analysed is directly related to the reachability graph module since the state space and its transitions are required to calculate the steady state at equilibrium. Although the steady state vector can be calculated quickly, users are kept waiting whilst the reachability graph is being generated.

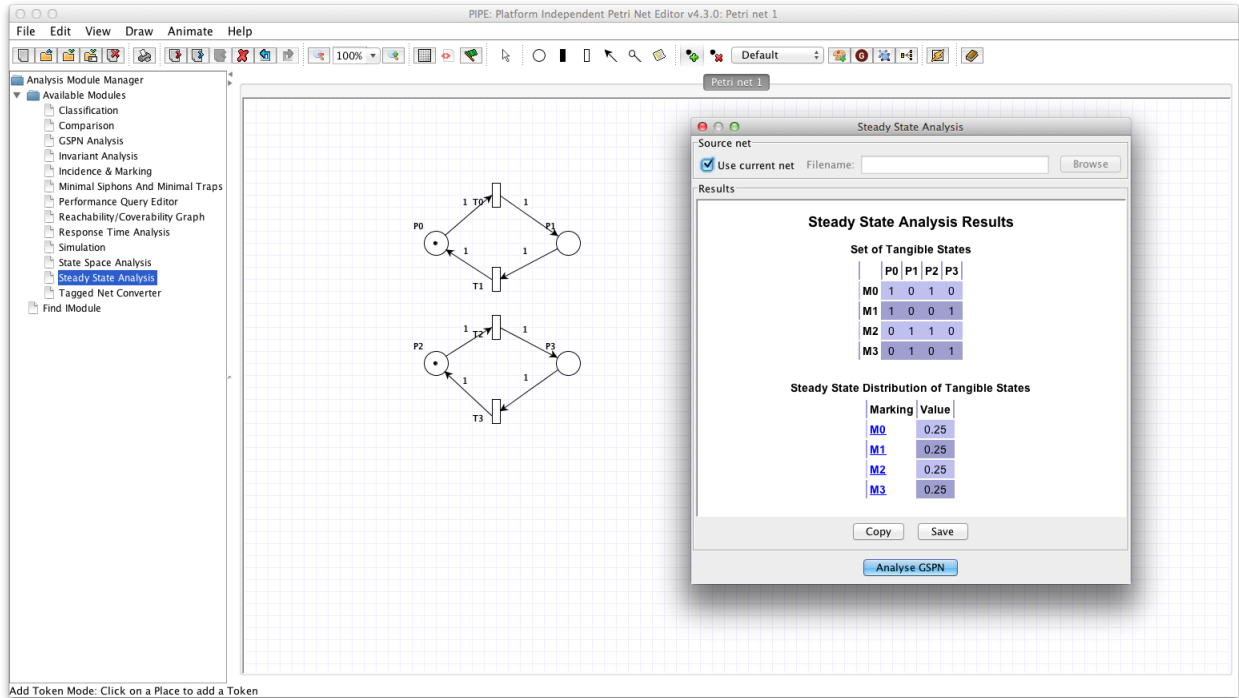


Figure 4.6: PIPE 4 steady state analysis results from running the steady state module on the Petri net depicted in Figure 2.8.

## 4.3 Quantitative code analysis

In this section PIPE 4's codebase has been analysed using static analysis plug-ins for IntelliJ and external standalone tools.

### 4.3.1 Static analysis

Installing and running the QAPlug plug-in for IntelliJ with the FindBugs, Checkstyle, PMD, and Hammurapi plug-ins found a total of 12 904 issues. The metrics chosen for this analysis were hand picked to avoid conflicting messages and to provide information about interesting problems.

We were particularly interested in the ‘God class’ metric because it highlights classes that have too much functionality making them over-complex and poorly designed. Out of PIPE 4’s 587 classes 66 of these were flagged up as ‘God classes’.

These metrics help identify areas of PIPE that need serious improvement and a further selection of the quality issues highlighted by the plug-in can be seen in Figure 4.7.

### 4.3.2 Cyclic dependencies

A cyclic dependency, as depicted in Figure 4.8, arises when a set of packages depend on each other making it impossible to import them independently. Cyclic dependencies are considered bad practice in software engineering because they introduce a tight coupling between the packages making them difficult to re-use or modify [28].

A structural analysis of PIPE 4 was performed by the Stan4J standalone tool. It reports cyclic dependencies as tangles and considers a tangle to be ‘A subgraph with at least two nodes, where each node is reachable from each other. Every cycle lies in a tangle and every tangle consists of just cycles’ [29]. It reports PIPE 4 as 29.17% tangled which can be seen diagrammatically in Figure 4.9.

Ideally PIPE should be acyclic; that is it should have no tangles. This is in practice however quite difficult so PIPE should have as few tangles as possible.

### 4.3.3 Documentation

The code itself lacks any good documentation structure; there are little to no comments and those that do exist are generally not written using Javadoc. The comments present often spell out what the language does rather than what the code is meant to do, why a decision has been made, or what a field represents. Comments on classes tend to only

- **Efficiency: 225**
  - ‘*Performance - Method concatenates strings using + in a loop count: 44*’ — In each iteration, the String is converted to a StringBuffer/StringBuilder, appended to, and converted back to a String. This can lead to a cost quadratic in the number of iterations, as the growing string is recopied in each iteration. Better performance can be obtained by using a StringBuffer (or StringBuilder in Java 1.5) explicitly.
  - ‘*Avoid Array Loops count: 3 - System.arraycopy is more efficient*’ — Instead of copying data between two arrays, use System.arraycopy method.
- **Maintainability: 1803**
  - ‘*Avoid duplicate literals count: 160*’ — Can usually be improved by declaring the String as a constant field.
  - ‘*Bad practice - Comparison of string objects using == or != count: 7*’ — This code compares java.lang.String objects for reference equality using the == or != operators. Unless both strings are either constants in a source file, or have been interned using the String.intern() method, the same string value may be represented by two different String objects. Consider using the equals(Object) method instead.
  - ‘*God class count: 66*’ — The God class rule detects the God Class design flaw using metrics. God classes do too many things, are very big and overly complex. They should be split apart to become more object-orientated.
- **Portability: 47**
  - ‘*Replace Vector With List count: 22*’ — Consider replacing Vector usages with the newer java.util.ArrayList if expensive thread-safe operation is not required.
  - ‘*Integer Instantiation count: 19*’ — Calling new Integer() causes memory allocation. Integer.valueOf() is more memory friendly.
- **Reliability: 1847**
  - ‘*Malicious code vulnerability - May expose internal representation by returning reference to mutable object count: 10*’ — Returning a reference to a mutable object value stored in one of the object’s fields exposes the internal representation of the object. If instances are accessed by untrusted code, and unchecked changes to the mutable object would compromise security or other important properties, you will need to do something different. Returning a new copy of the object is better approach in many situations.
  - ‘*Magic Number count: 1385*’ — Checks for magic numbers.
- **Usability: 8982**
  - ‘*Dodgy - write to static field from instance method count: 49*’ — This instance method writes to a static field. This is tricky to get correct if multiple instances are being manipulated, and generally bad practice.
  - ‘*System Println count: 412*’ — System.(out|err).print is used, consider using a logger.
  - ‘*Code Too Long count: 15*’ — Method is too long.

Figure 4.7: Examples of interesting code quality issues highlighted by the QAPlug plug-in for IntelliJ for the PIPE 4 codebase.

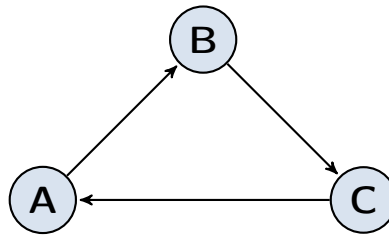


Figure 4.8: An example depicting three packages A, B and C with a cyclic relationship. A imports B, B imports C and C imports A.

document the author(s) of a class and/or method and the year which the changes were made. The comment of a class should instead document what it should be used for and additionally, if needed, provide example usage. Examples of types of comments found in PIPE 4 can be seen in Figure 4.10.

Using the Metrics Reloaded plug-in in IntelliJ for Javadoc we can observe from Figure 4.11 that the pipe package in the codebase has an alarmingly low amount of formal documentation. The amount of useful documentation is actually far less than this since the plug-in is unable to analyse the content of the documentation.

#### 4.3.4 Coding style

It is often considered good practice for classes and methods to be short, composed of axillary methods and to have a single purpose for the following reasons [30]:

- **Readability** — it is much easier to understand what a short piece of code does than try to dig through something that is lengthy. By creating short methods composed of other well named auxiliary methods it aids understanding of something which would be far more complex if written in a single few hundred line implementation.
- **DRY (don't repeat yourself)** — when you wish to make a change to an implementation you only have to change it in one place.



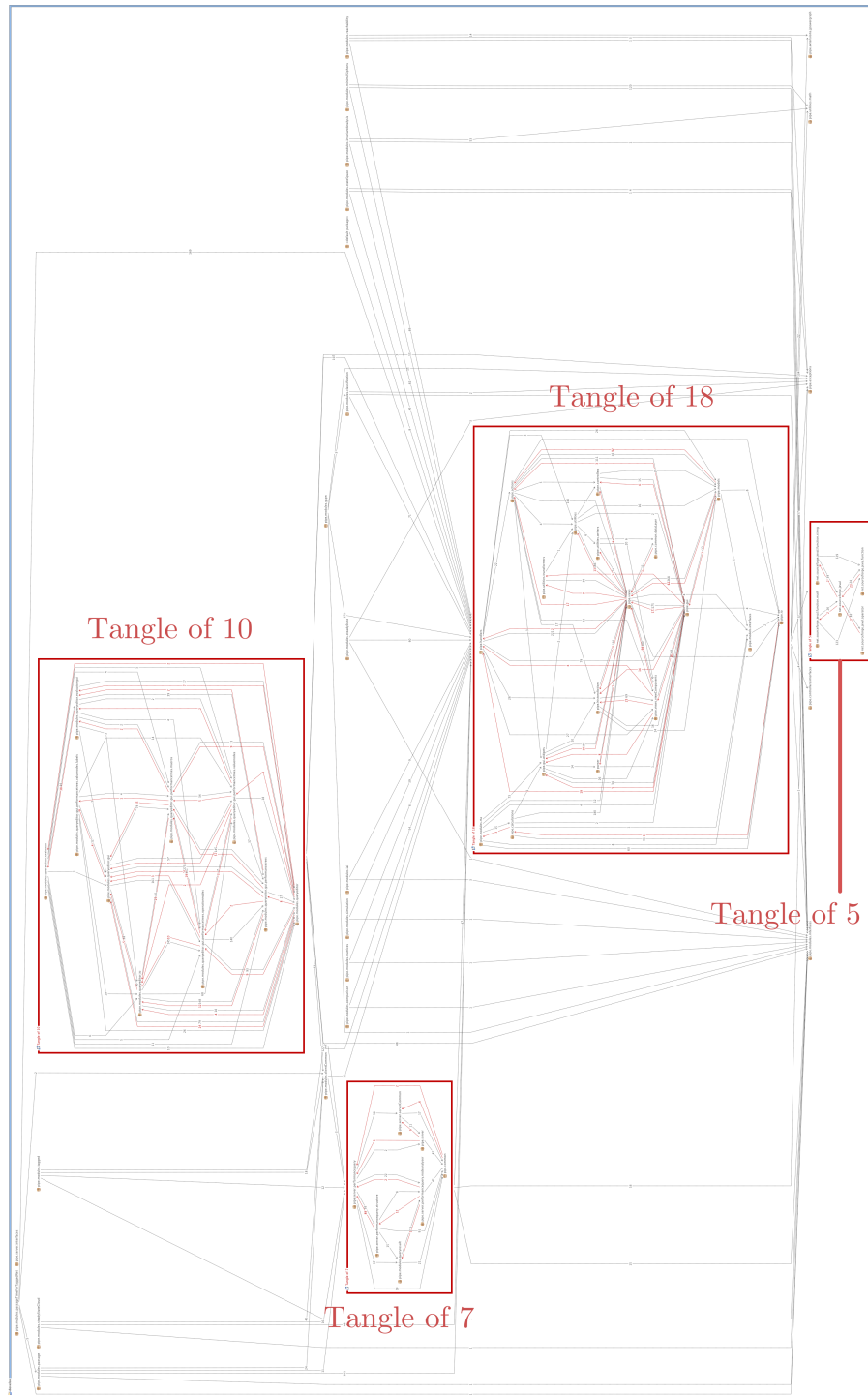


Figure 4.9: Stan4J tangle graph of the entire PIPE 4 codebase. It has 4 distinct tangles which are of sizes 18, 10, 7 and 5.

```
//aquest nom no est disponible...
JOptionPane.showMessageDialog(null, "There is already a place named " +
    newName, "Error", JOptionPane.WARNING_MESSAGE);
```

(a) A comment in PIPE 4 that is in another language, making it impossible to quickly understand.

```
public class PlaceTransitionObjectHandler extends PetriNetObjectHandler
{
    // STATIC ATTRIBUTES AND METHODS
    private static boolean mouseDown = false;
    public static boolean isMouseDown() { return mouseDown; }

    ...

    //constructor passing in all required objects
    PlaceTransitionObjectHandler(Container contentpane, ConnectableView
        obj) {
        ...
    }
}
```

(b) Comments which add nothing to the code and also suffer from spelling mistakes.

Figure 4.10: Examples of poor commenting in PIPE 4.

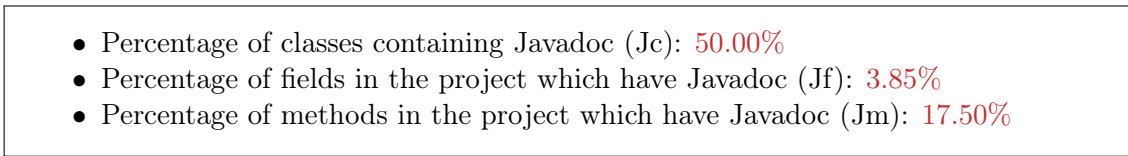


Figure 4.11: Statistics for the amount of Javadoc in the PIPE 4 codebase, taken from the IntelliJ Metrics Reloaded plug-in.

- **Testability** — unit testing is so much easier to do on a small method, and tests themselves end up being more readable because of this.
- **Reduces bugs** — well written and tested code reduces the number of bugs a project will suffer from in the long run.

The statistic plug-in for IntelliJ reveals that PIPE 4’s two largest classes are the `PetriNetView` with 3089 lines of code and `FlanaganMath` which contains 3571 lines of code.

Even the shorter classes in PIPE can often be unclear and difficult to understand and contain bad coding practices. Examples of this can be seen in Figure 4.12. As explained in Section 4.2.3.1 complex algorithms are often written in a single method with little to no auxiliary methods.

```

private String compareTransitions(TransitionView[] source,
    TransitionView[] comparison, boolean compareID, boolean
    compareName, boolean compareRate, boolean comparePriority) {
    ...
    if((!compareID || source[i].getId().equals(comparison[j].getId())) &&
        (!compareName ||
            source[i].getName().equals(comparison[j].getName())) &&
        (!compareMarking ||
            source[i].getCurrentMarkingView().get(0).getCurrentMarking()
            == comparison[j].getCurrentMarkingView().get(0)
            .getCurrentMarking()) &&
        (!compareCapacity || source[i].getCapacity() ==
            comparison[j].getCapacity()))
    {
        s = "Identical";
    }
    ...
}

```

(a) A snippet of the code used in the comparison module in `Comparison.java` to determine equality and similarities between Petri nets. These specific lines of code are particularly difficult to understand and are used to determine the equality of two arcs. This code appears 3 times within the same class.

```

public class AnnotationPanel extends javax.swing.JPanel {
    ...
    private void exit() {
        //Provisional!
        this.getParent().getParent().getParent()
            .getParent().setVisible(false);
    }
    ...
}

```

(b) `AnnotationPanel.java` exit method. The usage of `getParent` makes the code susceptible to bugs from a layout change elsewhere. It is also impossible to tell from the code which Swing component was intended to be closed. The comment adds no clues. An alternative to this should have been to use the `SwingUtilities` class which provides plenty of useful methods for getting parents and root level panes.

```
public File openFile() {
    if (showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
        try {
            return getSelectedFile().getCanonicalFile();
        } catch (IOException e) {
            /* gulp */
        }
    }
    return null;
}
```

(c) Ignoring exceptions is a very bad practice in Java because it hides genuine errors in the code and makes debugging very difficult. Not only is this an example of bad software engineering in `FileBrowser.java` but the comment is inexcusable.

Figure 4.12: Examples of poor quality code in PIPE 4.

## 4.4 Critical aspects of functionality

This section addresses critical issues with particular sections of code in PIPE 4.

### 4.4.1 Global state

Described briefly in Section 4.1 the `ApplicationSettings` class contains a set static of methods for binding and retrieving the `PipeApplicationView`, `PipeApplicationController` and `PipeApplicationModel`.

On instantiation each class must be registered via a call to the corresponding method in the `ApplicationSettings` class giving PIPE 4 a global state. In modern day programming having a global state is generally considered to be harmful to application design [31] as its use causes APIs to lie about their true dependencies [32]. For example a new developer using either of the APIs depicted in Figure 4.3 or Figure 4.13 would have no idea that they require a Petri net to analyse; only on a deeper inspection of the underlying implementation would they find the line:

```
ApplicationSettings.getApplicationView().getCurrentPetriNetView().
```

This design greatly hinders the learning curve of the project. Furthermore this class makes unit-testing very difficult and unreliable because static classes and methods affect the behaviour across every single test.

In PIPE 4 there are 280 uses of the `ApplicationSettings` class.

### 4.4.2 Functional expressions

The implementation of functional weights and rates is a significant area of concern in the PIPE 4 architecture.

<b>ExprEvaluator</b>
+ parseAndEvalExprForTransition(expr : String) + parseAndEvalExpr(expr : String, tokenId : String)

Figure 4.13: PIPE 4 API exposed to users for evaluating functional weights. Since there is no Javadoc associated with this class it is not immediately clear the differences that will occur when analysing the functional expression with each method.

The implementation of functional expressions happens in a class called `ExprEvaluator` whose API can be seen in Figure 4.13. The implementation is an area of concern for the following reasons:

- The Petri net is not explicitly passed into any of the methods. Instead they rely upon the `ApplicationSettings` static class to access the currently displayed `PetriNetView`. This design couples the logic of the Swing GUI to the evaluation of an expression.
- There are two evaluate methods with no clear explanation of what is different about the two of them.
- The method `parseAndEvaluateForTransition` loops through every single Petri net component checking to see if it is an instance of a `PlaceView`. It then attempts to look up these place names in the form `#(<name>)` in the expression. If the name exists it will replace it with the total number of tokens in the place. The string expression is then passed to an external library that can perform basic mathematical operations on strings. For clarity this is shown in Listing 4.1. String find and replace operations like this are extremely costly since the entire string must be manipulated in order to find, insert, and remove items. Looping through every single component is also an unnecessary task and shows a bad design of a Petri net if it is unable to return just the places it contains.
- The method `parseAndEvalExpr` performs a similar process to the one described above.

However instead of replacing the place reference ( $\#(\langle\text{name}\rangle)$ ) by the total number of tokens contained in the place, it replaces it by the number of tokens contained in the place for the specified token id.

- There is no user documentation on how to use functional weights for PIPE 4, the only help given describes the basic mathematical operations that can be used (Figure 4.14). This has led to people asking how to use the functional rate features [33].
- The logic for referencing a places tokens is not very clear. It swaps between the entire number of tokens for a transition and the number of a particular token for an arc. This choice should be deferred to the user as it is currently curbing the expressiveness of functional expressions.
- The transition rate editor displays an error message if the user tries to define a functional rates on an infinite server transition when it is connected to an arc with a functional rate (Figure 4.15). On inspection of the code it is not clear why this design decision has been applied. More worryingly if the user does this operation in reverse order, by setting the transitions rate first and then the arc weight no error is shown. This introduces a level of inconsistency to the logic of functional expressions and causes the analysis of a Petri net to break further down the line.
- It is possible to delete places when an arc weight references them.
- Rate parameters cannot have functional rates, there is no explanation behind this reasoning.

### 4.4.3 Infinite server semantics

Unfortunately the infinite server semantics in PIPE 4 are broken. Figure 4.16 shows the broken behaviour of a transition with infinite server semantics before and after firing.



```

public Double parseAndEvalExprForTransition(String expr) throws
    EvaluationException{
    String lexpr=new String(expr.replaceAll("\\s",""));
    Iterator iterator = _pnmldata.getPetriNetObjects();
    Object pn;
    String name;
    if (iterator != null) {
        while (iterator.hasNext()) {
            pn = iterator.next();

            //we parse the places with their number of tokens
            if (pn instanceof PlaceView) {
                lexpr = findAndReplaceCapacity(lexpr, pn);
                name=((PlaceView) pn).getName().replaceAll("\\s","");
                name = ("#("+name+")");
                if(lexpr.toLowerCase().contains(name.toLowerCase())){
                    int numOfToken=((PlaceView) pn).getTotalMarking();
                    do{
                        lexpr=lexpr.toLowerCase().replace(name.toLowerCase(),
                            numOfToken+"");
                    }while(lexpr.toLowerCase().contains(name.toLowerCase()));
                }
            }
        }
    }
    Evaluator evaluator = new Evaluator();
    String result = null;
    try {
        result = evaluator.evaluate(lexpr);
    } catch (EvaluationException e) {
        throw e;
        //e.printStackTrace();
    }

    Double dresult = Double.parseDouble(result);
    return dresult;
}

```

Listing 4.1: The parsing of a functional rate in the ExprEvaluator class. This code is particularly unclean because it loops through every Petri net object looking for a place and then queries the string to see if the place's id is referenced. Despite the underlying implementation being nonoptimal this code could benefit from the use of foreach loops instead of iterators, keeping declaration and assignment together and the splitting of operations into multiple lines for clarity. The formatting has been left as found in the class.



Figure 4.14: The help displayed by PIPE 4 when creating a functional weight or rate. It does not explain how to reference places at all.

Infinite server semantics only effect the rate at which the transition fires and should effect the number of tokens it takes from a place.

#### 4.4.4 Transition throughput analysis

When analysing the simple GSPN depicted in Figure 4.17 for its performance metrics using the GSPN analysis module we notice that the throughput of the transitions reported by PIPE 4 is incorrect.

The one-step generator matrix  $A$  for this Petri net is:

$$A = \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix},$$

which leads to a steady state distribution of:

$$\pi = [0.5, 0.5].$$

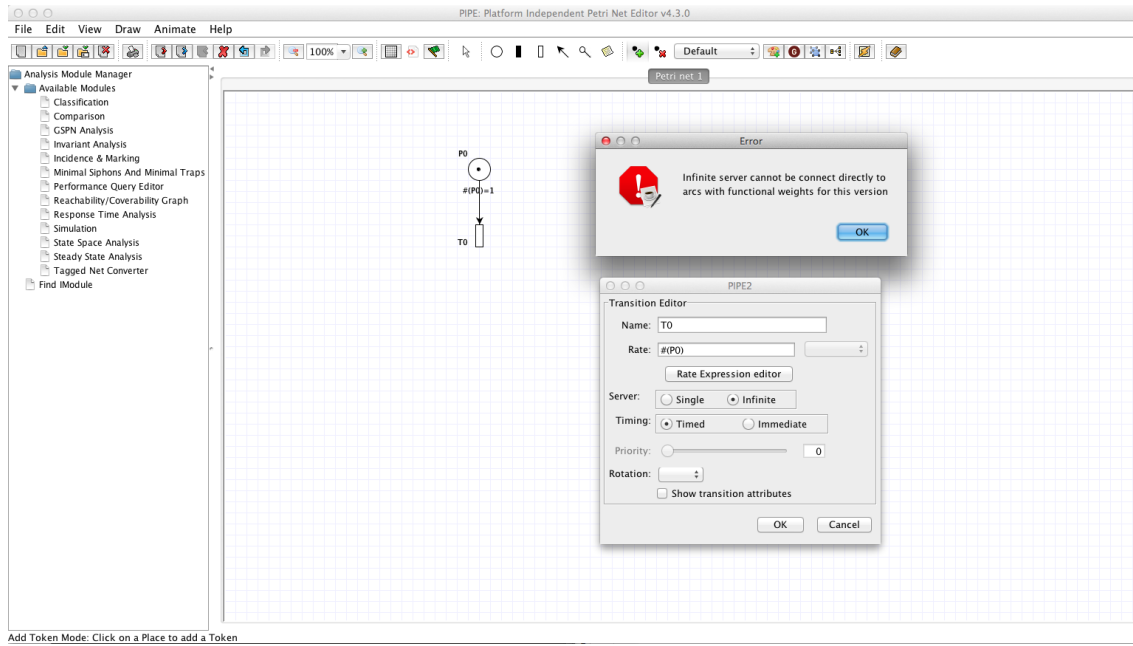


Figure 4.15: PIPE 4 does not support functional rates on infinite server transitions connected to weights with a functional rate. When trying to add a functional rate the error message displayed reads ‘Infinite server cannot connect directly to arcs with functional weights for this version’.

Using Equation (2.8) we calculate the set of states for which each transition is enabled:

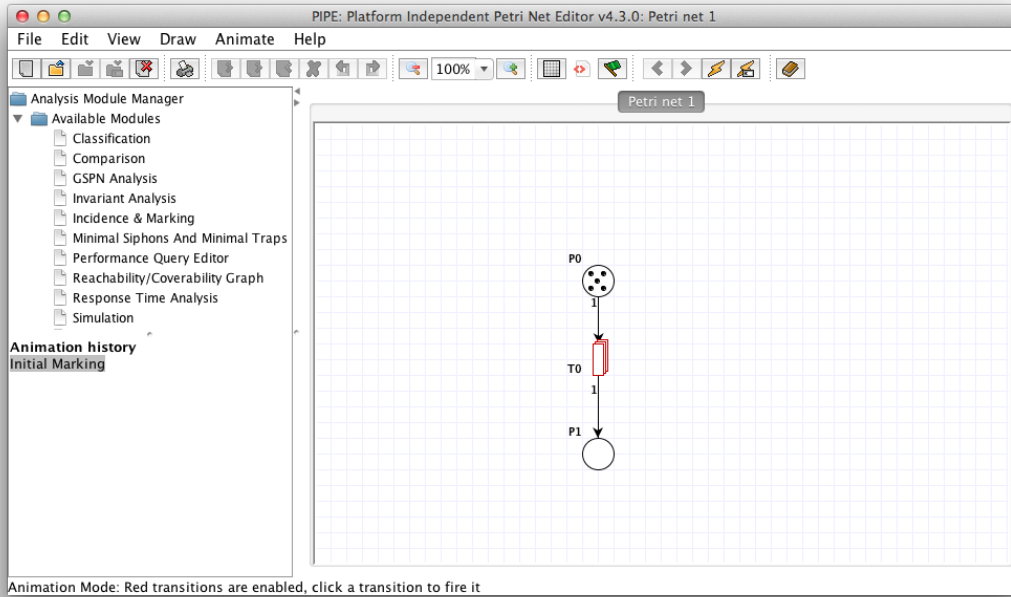
$$EN_{t_1} = \{(1,0)\}$$

$$EN_{t_2} = \{(0,1)\},$$

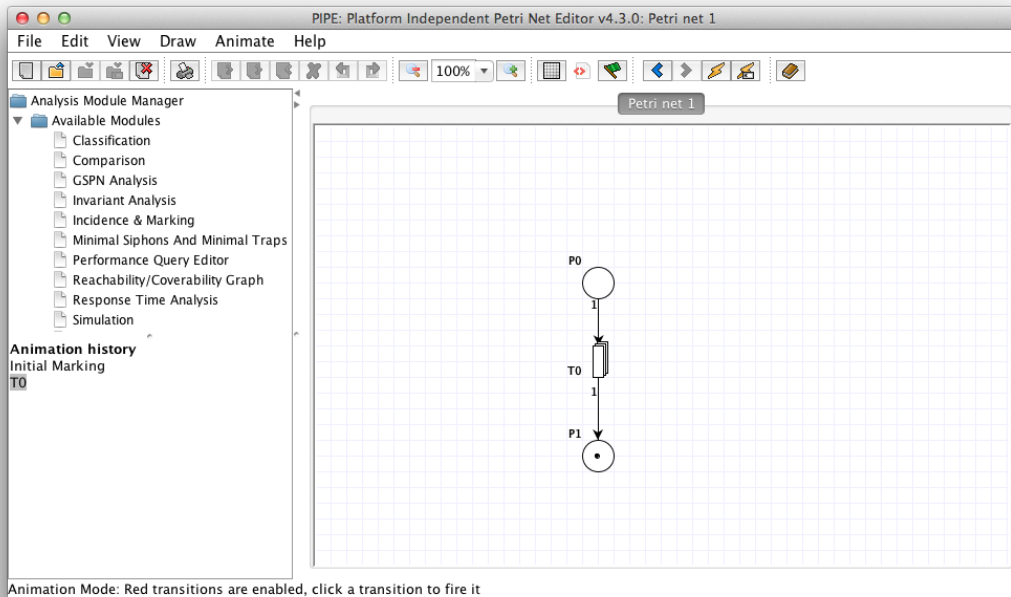
and generate the transition throughput metric of:

$$d = [0.5, 0.5]. \quad (4.1)$$

Comparing this with the results of PIPE 4’s GSPN analysis module (Figure 4.18) we see that the results reported by PIPE 4 are wrong.



(a) Before firing transition T0.



(b) After firing transition T0.

Figure 4.16: A Petri net with a single infinite server transition, T0, connected to places P0 and P1 before and after a single firing. Infinite server semantics should only effect the rate at which a transition fires, not the number of tokens that are produced. In this case firing T0 should have only taken one token from P0, not all 5.

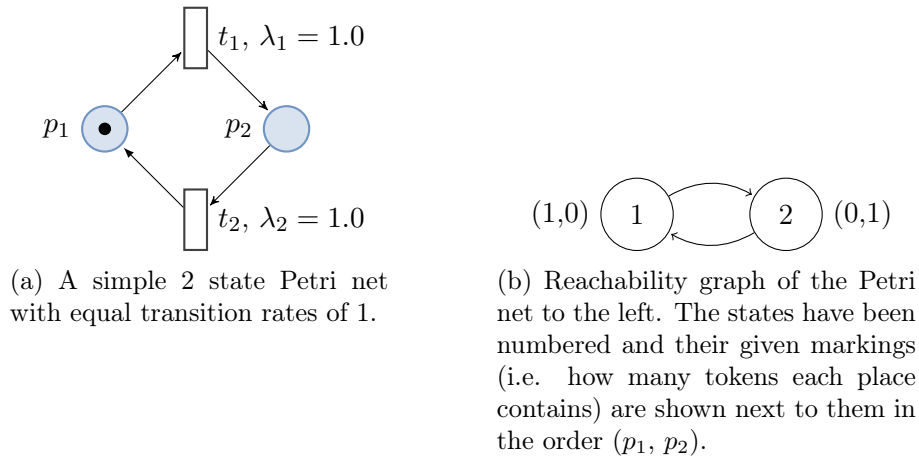


Figure 4.17: An example Petri net and its reachability graph for which PIPE 4 calculates the transition throughput performance metrics wrong.

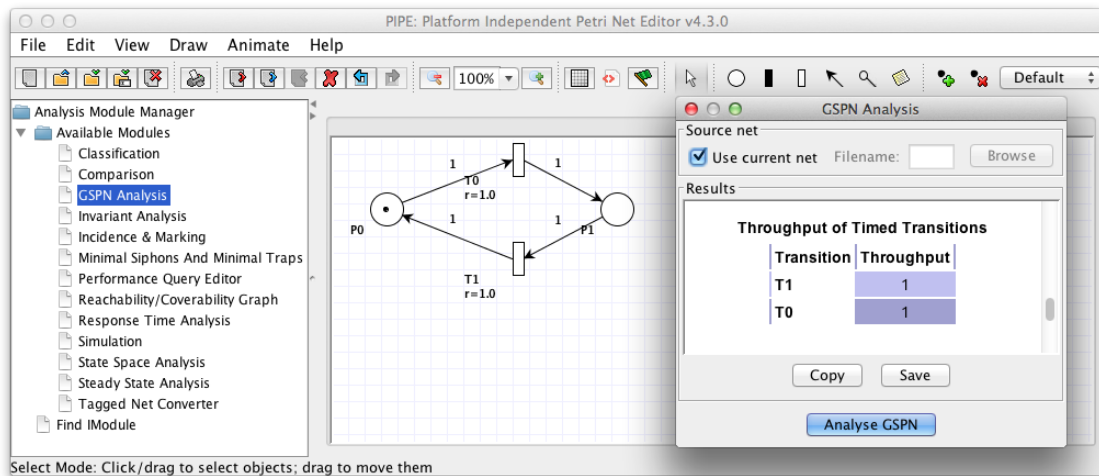


Figure 4.18: The incorrect transition throughput performance metrics reported by the GSPN analysis module in PIPE 4 for the simple 2 state Petri net in Figure 4.17. Equation (4.1) shows the expected transition throughput metric is  $[0.5, 0.5]$ , but the reported result in PIPE 4 is  $[1.0, 1.0]$ .

## Chapter 5

# Streamlining and modernising the build process of PIPE 5

I'm very classic and structural. I love clean lines and interesting, modern details. But I'm all about being streamlined - less is more.

Nina Garcia

### 5.1 Version control

Before this project began PIPE 4's build process had not changed since it was first developed in 2003; it was stored on SourceForge and made use of CVS for its version control. CVS is now generally considered a legacy version control system because its successors improve greatly on its feature set [34].

The main alternatives for to CVS are:

- **Subversion (SVN)** — released in 2000, SVN sports a centralised revision control

model meaning there is a single central copy of the code that programmers commit their changes to. As soon as changes are committed they become available to all developers accessing the codebase.

- **Mercurial** — released in 2005, Mercurial is a distributed version control tool which was designed with careful attention to scalability and performance [35].
- **Git** — like Mercurial, Git is a distributed version control system released also in 2005.

In recent years there has been a large migration towards distributed version control systems such as Git and Mercurial. With these systems making a change to a project involves a user checking out the entire repository on their own system, making their changes to a local copy and then checking them back into the central server. The benefit of this is that developers are able to make more granular changes it also allows them to work offline if needs be. SVN cannot offer this.

Git was chosen for this project instead of Mercurial because we have had most experience with it. Its benefits largely outweigh those of SVN and CVS and unlike Mercurial it can be used in conjunction with GitHub for open-source programming as documented in Section 5.2. Furthermore Git offers a large range of benefits to the user, some of which can be seen in Table 5.2, making it the best tool for the job.

Feature	Git Benefits
Setting up a repository	Git stores its repository information in a <code>.git</code> directory in the top level of the project. CVS requires setting up a central place for storing version control information for different projects known as a <code>CVSRoot</code> . This makes importing existing sources into Git very easy where as with CVS it is more complicated.
Atomic operations	Commits and other operations are not atomic in CVS due to its origins from a set of scripts. If an operation on the repository is interrupted the repository can be left in an inconsistent state. In Git all operations are atomic.
Changesets	Changes in CVS are per file, whilst commits in Git always refer to the whole project. This means in Git it is very easy to revert or undo an entire change. Due to this the <code>git-blame</code> feature can be used to see the author of every line of code for every change.
Naming revisions / version numbers.	Since CVS changes are per file, version numbers reflect how many times a given file has been changed. In Git each version of a project (each commit) has its own unique name given by an SHA-1 id.
Easy branching	Branches in CVS are complicated and difficult to implement for many reasons including the fact that it does not support merge tracking. Git remembers all information required for the merge by itself so creating and merging branches is as easy as a few command line operations. Since branches are so easy to use in Git good branching work flows have been developed which lead to much more collaborative programming.
Renaming	File renames are not supported in CVS and manual renaming can invalidate the history. Git uses a heuristic approach based on the similarity of file contents and the file name itself to allow for renaming.
Amending commits	In distributed version control systems the act of publishing is separate from creating a commit allowing developers to change unpublished parts of the history without inconveniencing others. For example a typo in a commit message or a bug in a commit can easily be changed using <code>git commit --amend</code> . This is not possible in CVS.
Tools	Git offers a vast number of useful programmer tools that CVS does not. For example <code>git bisect</code> can be used to find a commit that introduced a bug, <code>git blame</code> can be used to see who was last responsible for editing a line of code and in what commit, <code>git subtree split</code> can move whole directories out into new projects.

Table 5.2: Benefits of Git version control over CVS, heavily influenced by the Stack Overflow answer in [36].



## 5.2 GitHub

In addition to the benefits of Git, GitHub has emerged as a repository hosting service with a feature set that enhances the development an open-source project. PIPE 5 has been migrated to GitHub because it provides some very useful collaboration features via its web based graphical interface which are detailed below:

- **Landing page** — the GitHub project landing page has lots of useful information for potential users and developers. It shows the number of commits, the dates of the last commits to each file, and displays the users README in Markdown format. The landing page for PIPE can be seen in Figure 5.1.
- **Issue tracking** — issues and enhancements can be created and discussed in a collaborative manner (Figure 5.3). Coloured tags can be added to the issues to help identify their type and urgency (Figure 5.2). Issues can also be referenced and optionally closed in Git commit messages by referring to the issue number; once pushed the commit appears under the issue itself.
- **Forking** — a feature unique to GitHub which offers the ability to copy a project to another developers account. Forking is seen now as one of the best ways to contribute to open-source projects by allowing the user to make changes and optionally submit an interactive ‘pull request’. The owner of the original project can see all lines of code changed (Figure 5.4), and a discussion can be had about the proposed changes. If further work needs doing it can be submitted to the same pull request and only when the project owner is sufficiently happy they can easily merge the changes with the click of a single button.
- **Releases** — GitHub supports releases of projects with an optional pre-release beta identifier. The project owner can choose to release their project when they believe

stable development points have been reached or when significant bug fixes have been made. This allows users to run pre-chosen versions of the code rather than the development branch. The release page for PIPE 5 can be seen in Figure 5.5.

- **GitHub pages** — GitHub provides hosting for project documentation which is discussed in Section 6.5.1.
- **Continuous integration** — GitHub provides support for a continuous integration server using post-commit hooks as discussed in Section 5.3.

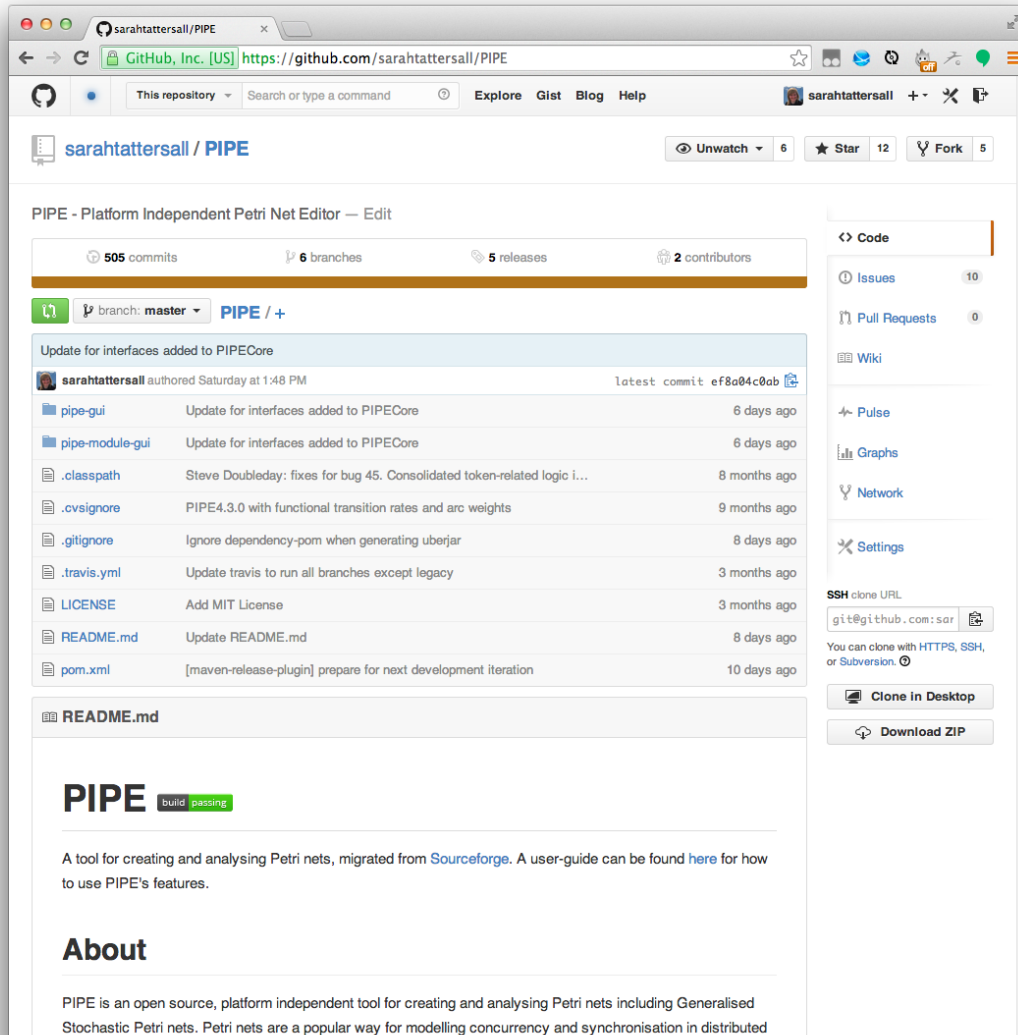


Figure 5.1: The GitHub landing page for PIPE 5. It displays the README written in Markdown format, which contains the build status of the continuous integration server (passing in green) and instructions for building, running and contributing to the project.

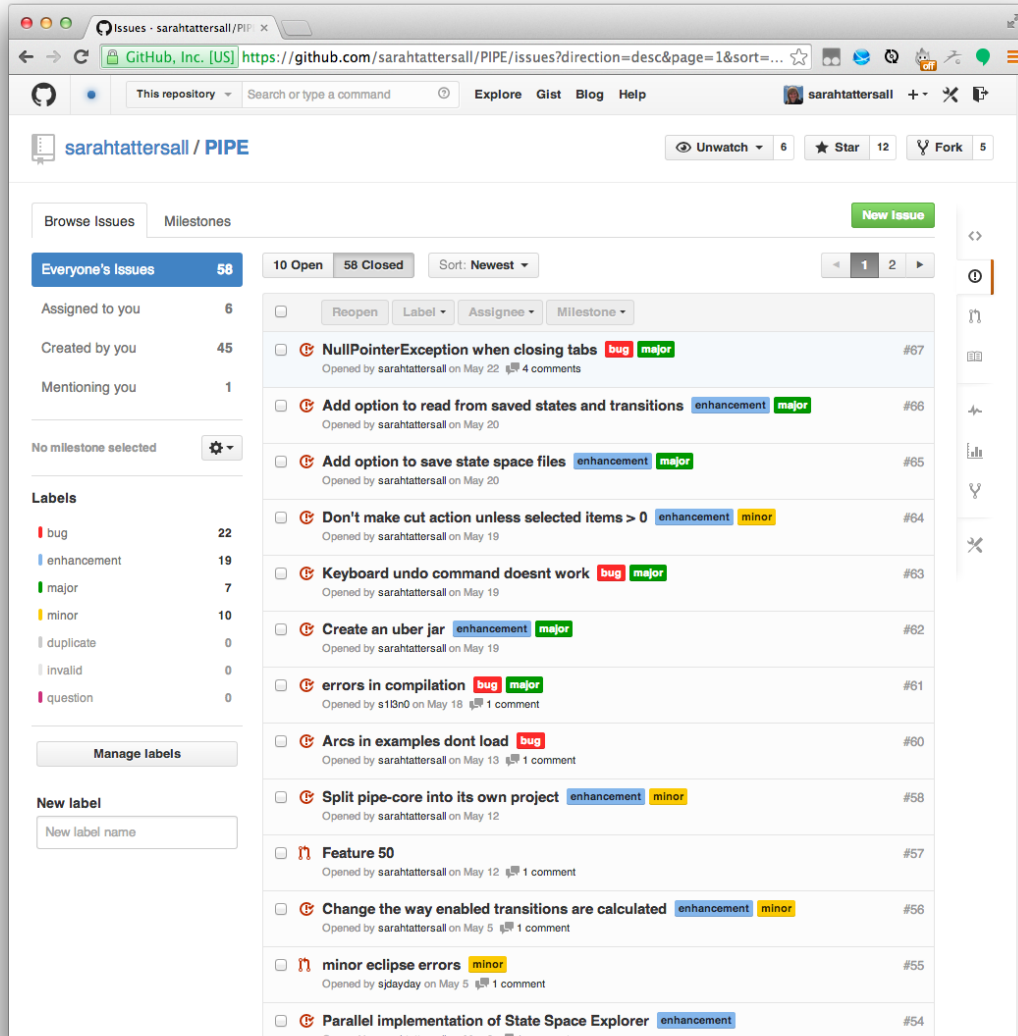


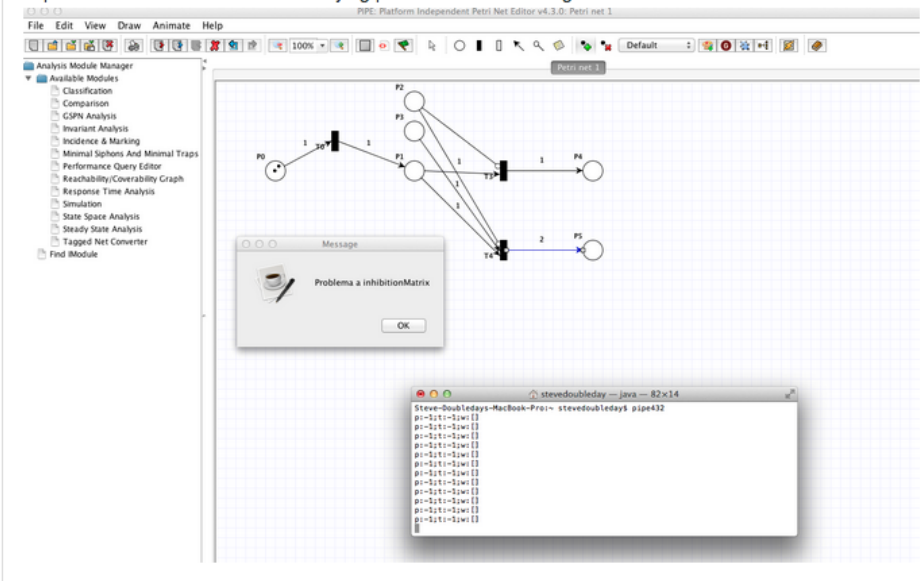
Figure 5.2: GitHub issue page for PIPE 5 showing a selection of issues that have been raised during the development of PIPE 5. The tags used in PIPE 5 can be seen next to the issues and helped us to prioritise our workload.

## ◀ Problem with inhibitory arcs doesn't let user continue #21

**Closed** sdayday opened this issue on Feb 10 · 5 comments

**sdayday** commented on Feb 10 Collaborator

The dialog box for some error condition relating to inhibitory arcs continues to re-appear, making it impossible to fix whatever the underlying problem is. Seen in tag v4.3.2.



**sdayday** commented on Feb 10 Collaborator

I included the window where I invoked PIPE, as each time the Message box re-appears we see something written to stdout/err....might help figure out where we're going wrong.... Steve

```

Steve-Doubladays-MacBook-Pro:~ stevedoubladays pipe432
p1=1:t1:-1:wi {}
p1=1:t1:-1:wi {}
p1=1:t1:-1:wi {}
p1=1:t1:-1:wi {}
p1=1:t1:-1:wi {}
p1=1:t1:-1:wi {}
p1=1:t1:-1:wi {}
p1=1:t1:-1:wi {}
p1=1:t1:-1:wi {}
p1=1:t1:-1:wi {}

```

**sarahattersall** commented on Feb 10 Owner

What were you trying to do with the arc at the time? Since this is v4.3.2 I'll see if it's fixed in the develop branch.

Figure 5.3: A discussion on a GitHub issue of an inhibitor arc bug present in PIPE 4. GitHub allows users to upload screenshots and paste code snippets. Issues and comments are written and displayed in Markdown format for ease of reading.

The screenshot shows a GitHub pull request page for a repository. The title is "Moved Travis Build Icon #28". The status is "Closed", and it indicates that "petehamilton" wants to merge 1 commit into "sarahtattersall:develop" from "petehamilton:patch-1".

Below the title, there are statistics: "Conversation 0", "Commits 1", and "Files changed 1". A progress bar shows 4 red squares, with "+1 -6" next to it.

The main content area shows a comment from "petehamilton" dated Feb 24: "Travis build icon is now at the top of the README". Below the comment is a commit link: "Moved Travis Build Icon" with a green checkmark and the hash "70db3fb".

Below the commit link, a red circle with a white 'X' indicates that "sarahtattersall" closed this pull request on Feb 24.

On the right side, there are sections for "Labels" (None yet), "Milestone" (No milestone), and "Assignee" (No one assigned). There is also a "Notifications" section with an "Unsubscribe" button and a note: "You're receiving notifications because you modified the open/close state." At the bottom right, it says "2 participants" with two profile icons.

At the bottom of the page, there is a comment input area with a "Write" tab, a "Preview" tab, and a "Comment" button. The input area contains the text "Leave a comment" and a "ProTip" that says "Add comments to specific lines under Files changed."

(a) Pull request informational page for moving the Travis CI build status to the top of the README.

The screenshot shows a GitHub pull request titled "Moved Travis Build Icon" by user "petehamilton" on Feb 24. The pull request is for the repository "petehamilton / PIPE", which is a fork of "sarahattersall/PIPE". The pull request details show 1 parent commit (11a74e3) and the current commit (70db3fbbfffb7fcb7a62fbfb6022b66193733aad). The diff shows changes to the "README.md" file, with 1 addition and 6 deletions. The diff highlights the following changes:

```

@@ -1,4 +1,4 @@
1 1  -# PIPE 2 #
1 1  +# PIPE 2 [![Build Status](https://travis-ci.org/sarahattersall/PIPE.png?branch=develop)](https://travis-ci.org/sarahattersall/PIPE)
2 2
3 3  A tool for creating and analysing Petri nets, migrated from http://pipe2.sourceforge.net/about.html
4 4
@@ -46,11 +46,6 @@ The original method for these local libraries required running a Python script t
46 46
47 47  If you know of a better way to do this, please raise it in the issues section.
48 48
49 49  -## Testing ##
50 50  -PIPE is using Travis for it's builds. The current build status is:
51 51  -[![Build Status](https://travis-ci.org/sarahattersall/PIPE.png?branch=develop)](https://travis-ci.org/sarahattersall/PIPE)
52 52  -
53 53  -
54 49  ## Contributing ##
55 50
56 51  Just follow the following recommended process:

```

Below the diff, there is a comment form with a "Write" tab selected, a "Preview" tab, and a "Comment on this line" button. The form contains the text "Leave a comment" and a "Close form" button.

(b) Clicking on the commit hash '70db3fb' on the pull request page depicted in (a) leads to this page. Any developer can browse the changes made and optionally comment on any of the lines.

Figure 5.4: A pull request submitted for PIPE where Peter Hamilton forked the project and moved the build status icon to the top of the README.

The screenshot shows the GitHub releases page for the repository 'sarahtattersall / PIPE'. At the top, it indicates the repository is 'PUBLIC' and shows 6 Unwatch, 12 Star, and 5 Fork actions. The main heading is 'Releases / Tags' with a 'Draft a new release' button. Two pre-releases are listed:

- PIPE 5.0.0 Beta 3** (Pre-release): Released this Tuesday at 9:14am with 8 commits to master since this release. Description: 'This pre-release supports animation but no module interaction yet. It has fixed numerous bugs and if you only wish to simulate animation is preferable to PIPE v4. It includes a stand-alone runnable uber-jar so that installation is not necessary.' Assets: 'PIPE-gui-5.0.0-beta-3.jar', 'Source code (zip)', and 'Source code (tar.gz)'. Edit button is present.
- PIPE 5.0.0 Beta 2** (Pre-release): Released this on Feb 23 with 163 commits to master since this release. Description: 'This pre-release fixes #25. Editing functional transitional rates should now work for any valid expressions on a transition.' Assets: 'Source code (zip)' and 'Source code (tar.gz)'. Edit button is present.

Figure 5.5: A snippet of the release page on PIPE 5’s GitHub repository. Pipe 5.0.0 Beta 3 includes a runnable jar which requires no installation, the user just needs to double click the jar and it opens PIPE 5. This makes the project more accessible to users who do not wish to install the project on their machine.



## 5.3 Travis CI

Originally there was no continuous integration server for PIPE 4 which meant that committed changes that broke the build could go undetected. PIPE 5 has now been set-up with a continuous integration server called Travis CI. Travis CI offers its services for free for open-source projects and can be easily integrated into a project via a GitHub plug-in which uses post-commit hooks to trigger builds on their servers.

Travis CI provides users with a build status icon that they can include in their GitHub README, to show off the level of quality they have associated with their project. PIPE 5's can be seen in Figure 5.1 at the top of the README and immediately shows visitors to the landing page the projects most recent build status. Clicking on the icon takes them to the most current build on Travis CI as seen in Figure 5.6.

Travis CI is also very useful for open-source contributions to a project made in a fork. When submitting a pull request Travis CI will run all tests meaning no broken builds can accidentally get merged.

Furthermore Travis CI can be set-up to send emails when the build has been broken, errors, or is fixed.

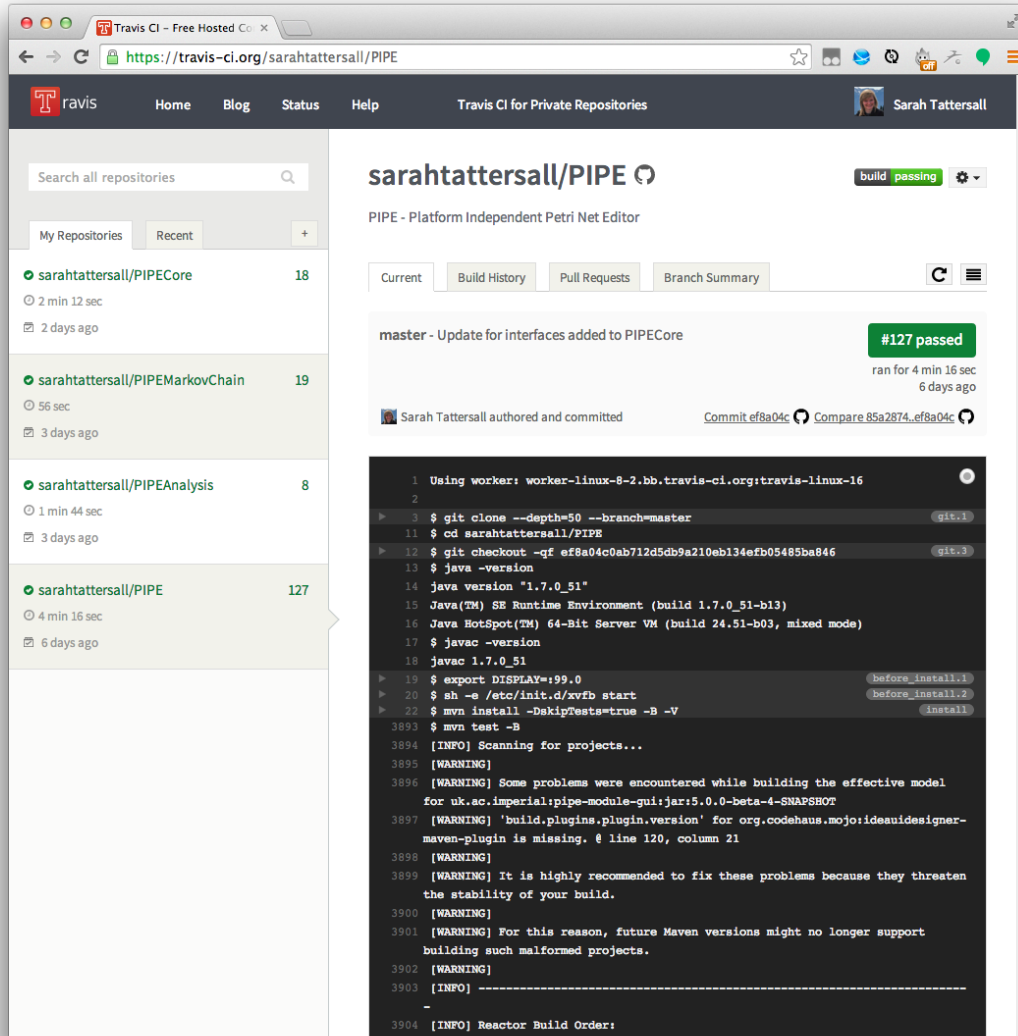


Figure 5.6: A passing build that has run on the Travis CI server. The entire console output is shown in the build. This is particularly helpful for spotting the cause of an error in failing builds.

## 5.4 Automating the build process via Maven

Originally PIPE 4 was built using a collection of user-written scripts that could be fiddly to get to work. To address this Maven was introduced into PIPE 5 to provide a stable cross-platform build process and dependency management system.

The dependency management feature of Maven is particularly useful because it allows for libraries to be automatically downloaded during installation rather than stored in a jar alongside the repository.

In Maven dependencies are declared in the following manner in an XML file called `pom.xml`:

```
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>15.0</version>
</dependency>
```

This structure for declaring dependencies makes it very easy to include other libraries and to upgrade to a newer version.

We made use of Maven build plug-ins to simplify the full release process of PIPE 5 which now involves the following steps:

- Resolving any SNAPSHOT dependencies — these are dependencies that reference unreleased versions of projects. In PIPE 5 the codebase has been split across multiple repositories so these are the dependencies that reference the other PIPE libraries.
- Bumping the version number — this involves bumping any of the major, minor or patch numbers in the pom for the next release.

- Creating a runnable uber-jar — this creates a jar of PIPE that can simply be run by double clicking on it without having to build the project.
- Deploying a project to a Maven repository as an artifact — all of the repositories involved in PIPE 5 are stored as artifacts on GitHub. This stage allows other developers to reference the projects in their dependencies without manually downloading the jars.
- Generate a GitHub release — this includes releasing the uber-jar and sources on GitHub for users to download. It appears on the GitHub project release page.

The command to execute all of the aforementioned steps can be performed in a single line, which significantly eases the build and release process of PIPE 5:

```
mvn release:clean release:prepare release:perform.
```

## Chapter 6

# A new software architecture for PIPE 5

Leap, and the Petri net will appear.

John Burroughs (paraphrased)

As part of PIPE 5's renovation the logic of a Petri net has been separated into its own project which can be accessed at <https://github.com/sarahtattersall/PIPECore>. This separation allows other developers to make use of our Petri net models without the contamination of any Swing GUI code. The rest of this chapter details the new architecture for a Petri net, its components and underlying Markov chain.

### 6.1 Creating a new Petri net model

The first step in developing a new back-end for PIPE was to define a set of interfaces, shown in Figure 6.1, that depict the Petri net's component structure. Interfaces were used to make the code extendable for future use, such as the introduction of multiple server

semantics in a transition. These components can then be stored in a `PetriNet` class which is responsible for housing components and determining if operations are legal within the Petri net. For example if a place is deleted from a Petri net, the Petri net first ensures that no arcs or transitions reference the place in their functional expressions. If they do a `PetriNetComponentException` is thrown. This exception is used for any illegal operations that happen within a Petri net and contains a description of the error in its message.

As well as the Petri net components a set of new classes to represent the underlying Markov chain of the Petri net were created. They are located in their own `PIPEMarkovChain` repository on GitHub and aim to decouple the state of a Petri net from its underlying structure. The core classes in this library can be seen in Figure 6.2. They directly correspond to the Markov chain representation of a Petri net and can be used for simulation and analysis.

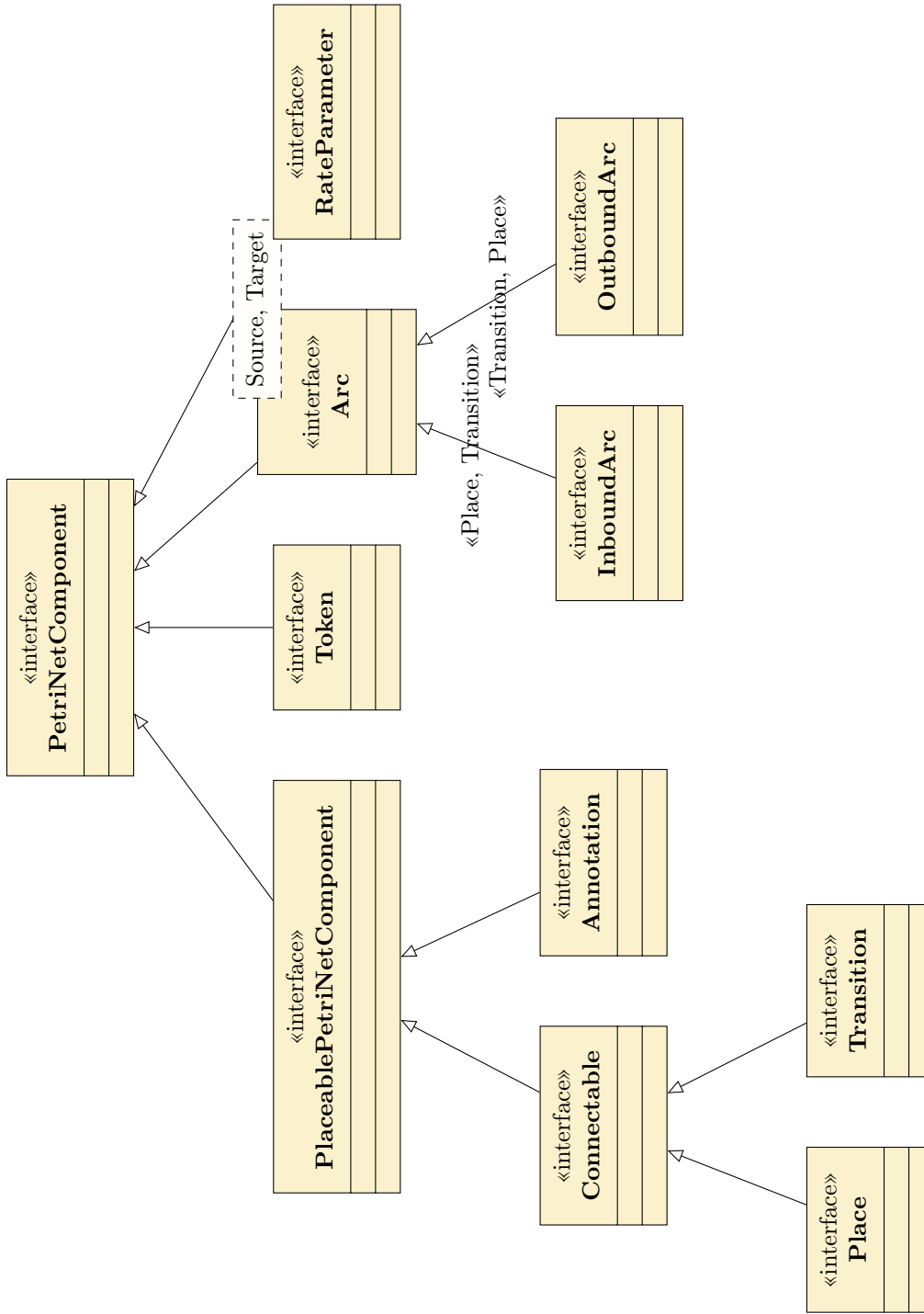


Figure 6.1: UML diagram of the PIPE 5 Petri net components hierarchy. At the top of the hierarchy is the PetriNetComponent which contains methods common to every component. The PlaceablePetriNetComponent interface then defines a set of methods for storing the coordinates of components that can have (x, y) locations on the canvas. The Connectable interface describes a class which an arc can connect to and is subclassed by the Place and Transition interfaces which represent the nodes in the bipartite graph. Furthermore an Arc has been split into its connection type using Java generics and makes operations on different arc types much easier. PIPE 5 implements a set of concrete classes which extend these interfaces.

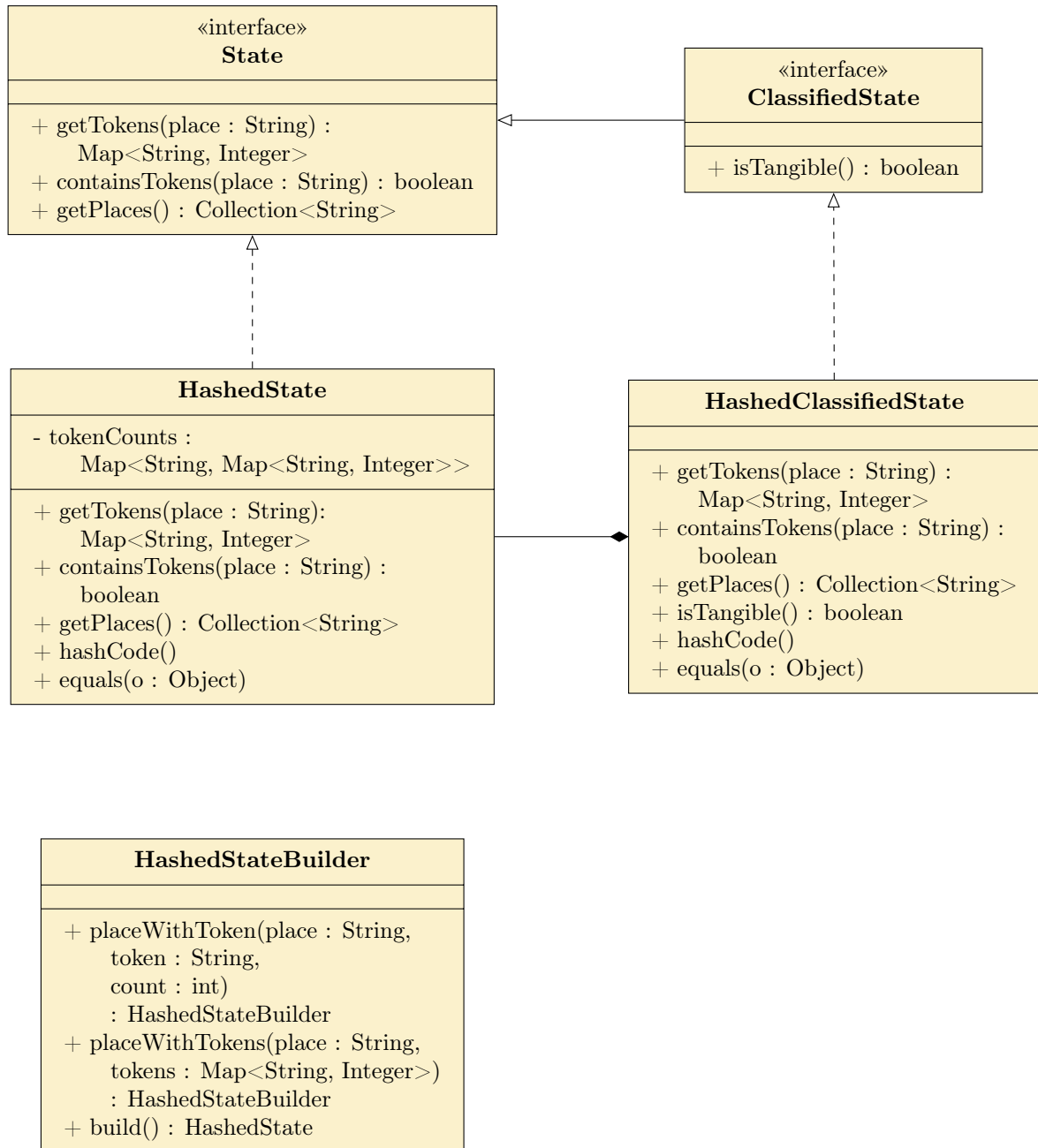


Figure 6.2: UML diagram for the state classes available in the Markov chain library introduced to PIPE 5. The `State` is immutable once built and maps place ids to a map containing token ids and their count. In order to ease the process of building a state the builder design pattern has been used to create the class `HashedStateBuilder` that creates `HashedState` objects.



## 6.2 Petri net component software architecture

The Petri net components exhibit a variety of design-patterns which aid in their long term usability and maintenance.

### 6.2.1 The publish–subscribe design pattern

The publish–subscribe design pattern is used to model a one-to-many dependency between objects. Subscribers and publishers register themselves a topic; when a publisher sends its message to these topics it is then forwarded to any subscribers. These subscribers can then update themselves according to the message they have received [21]. This design pattern avoids tight coupling of objects by removing the dependencies between them and makes the objects more reusable for different purposes. Broadcast communication is very easy with this pattern since a subject can send messages to any number of listeners.

In the case of PIPE 5 this design pattern is perfect for the implementation of a model–view–controller architecture because it removes the coupling between the models and the views. The Petri net components are the publishers and the views the subscribers.

We implemented this design pattern in Java using a `PropertyChangeSupport` class which manages a list of listeners and dispatches `PropertyChangeEvents` to them rather than using the flawed `Observable` class. Change events consist of a message, the old value, and the new value; they represent a change in an objects property. A simplified example of the publish–subscribe architecture can be seen in Figure 6.4. In the case of PIPE 5 the Petri net components act as a publisher and a topic and the Petri net acts as both a publisher firing messages when components are added and deleted but also as a subscriber listening for changes in Petri net components, such as their id, so that it can internally update its data structures. The GUI views act entirely as subscribers listening to their underlying

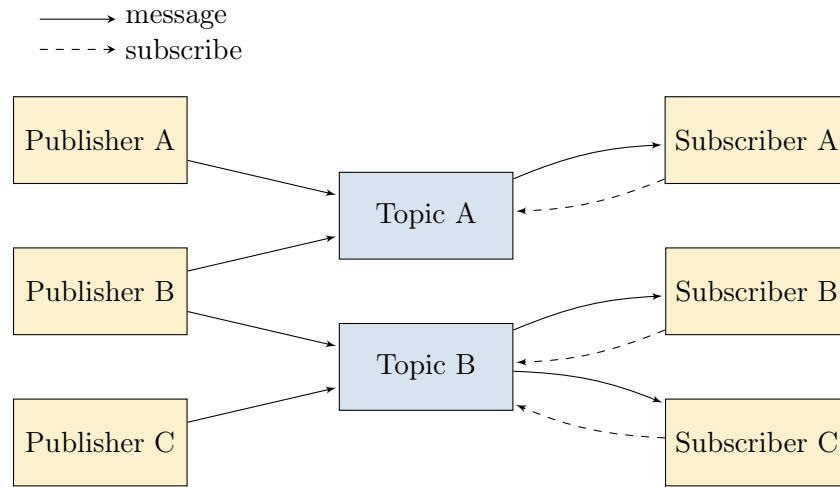


Figure 6.3: The publish–subscribe design pattern architecture. All publishers send messages to any number of registered topics who in turn forward the messages on to any subscribers who have subscribed to them.

model.

## 6.2.2 The acyclic visitor design pattern

The visitor design pattern is used in situations when a set of capabilities need to be added to a composite of objects and encapsulation is not important [37]. The visitor visits each element of a composite and can perform various operations based on the objects state. The class structure for this pattern can be seen in Figure 6.5.

A great use of this design pattern in PIPE 5 is when an API returns a collection of `PetriNetComponent` classes. This pattern can be used to perform various functionalities on the concrete component classes without adding the specific behaviour to them. By having each item accept a visitor we can programatically define the behaviour for each class instead of checking the objects instance type, casting and then calling a method.

There are however some design problems with the visitor pattern described above because

```

public class Transition {

    public static final String RATE_CHANGE_MESSAGE = "RATE_CHANGE";

    private int rate = 1;

    protected final PropertyChangeSupport changeSupport = new
        PropertyChangeSupport(this);

    public void addPropertyChangeListener(PropertyChangeListener
        listener) {
        changeSupport.addPropertyChangeListener(listener);
    }

    public void setRate(int newRate) {
        int oldRate = rate;
        this.rate = newRate;
        changeSupport.firePropertyChange(RATE_CHANGE_MESSAGE, oldRate,
            newRate);
    }
}

```

(a) Transition class that fires a property change message when its rate changes.

```

public class TransitionView {
    public TransitionView(Transition transition) {
        transition.addPropertyChangeListener(new PropertyChangeListener() {
            @Override
            public void propertyChange(PropertyChangeEvent evt) {
                String message = evt.getPropertyName();
                if (message.equals(Transition.RATE_CHANGE_MESSAGE)) {
                    int newRate = (int) evt.getNewValue();
                    displayChangedRate(newRate);
                }
            }
        });
    }
    ...
}

```

(b) A transition view that takes the transition in its constructor and registers a property change listener to it via an anonymous inner class. When the transition fires the anonymous listeners property change method is called and the transition view can update accordingly.

Figure 6.4: A simplified example of how the publish-subscribe design pattern is used in PIPE 5. In this case the Transition fires a message when its rate is changed and the TransitionView is alerted to this and can display the change accordingly.

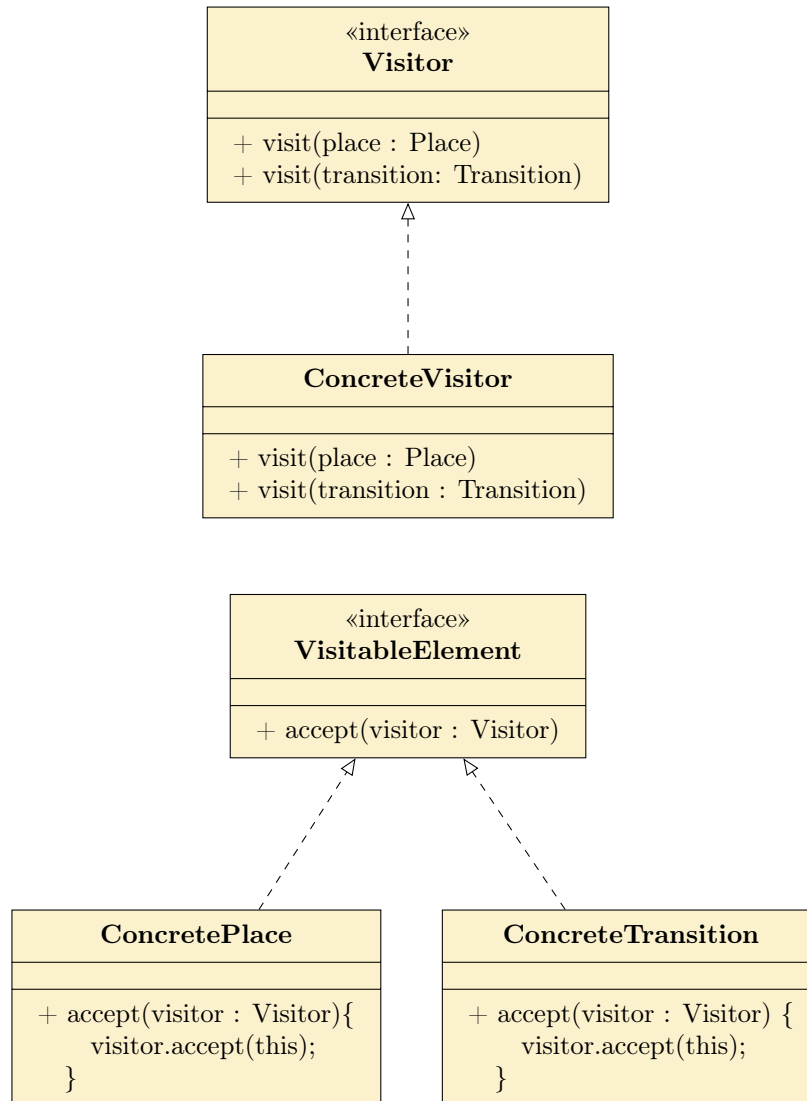


Figure 6.5: An example of the visitor design pattern applied to places and transitions. The `Visitor` interface is accepted by all `VisitableElement` subclasses and then parameter overloading is used to call the correct concrete method in the `ConcreteVisitor`.

it creates a dependency cycle between Java packages [38], which is just the thing we wanted to avoid in Section 4.3.1. Additionally adding new objects, in PIPE 5's case a new Petri net component, means that every visitor class must also be changed to support the new visit method. This can make things difficult for programmers and does not always encourage good coding standards.

Moreover this pattern does not allow users to pick only a handful of objects that actually need visiting. For example when we wish to translate items on the canvas we only want to move those that are actually placeable. The original visitor pattern would require methods for every single Petri net component, many of which would not be used. Without good documentation it may not be immediately clear why the method has not been implemented and so could be mistaken for a bug.

The acyclic visitor design pattern aims to overcome these issues by breaking the dependency cycle between the visitors and the components. It does this by making the base visitor a marker interface, an interface with no methods [39], and then requires one derivative interface to be declared for each type of element. An example is shown in Figure 6.6.

This design pattern has been invaluable in implementing a new core library for PIPE 5 and has been used in the following places:

- cloning a Petri net
- pasting components in a Petri net
- translating components in a Petri net by a specified vector amount
- adding and removing items from a Petri net
- querying if an arc can be attached to a particular source and target

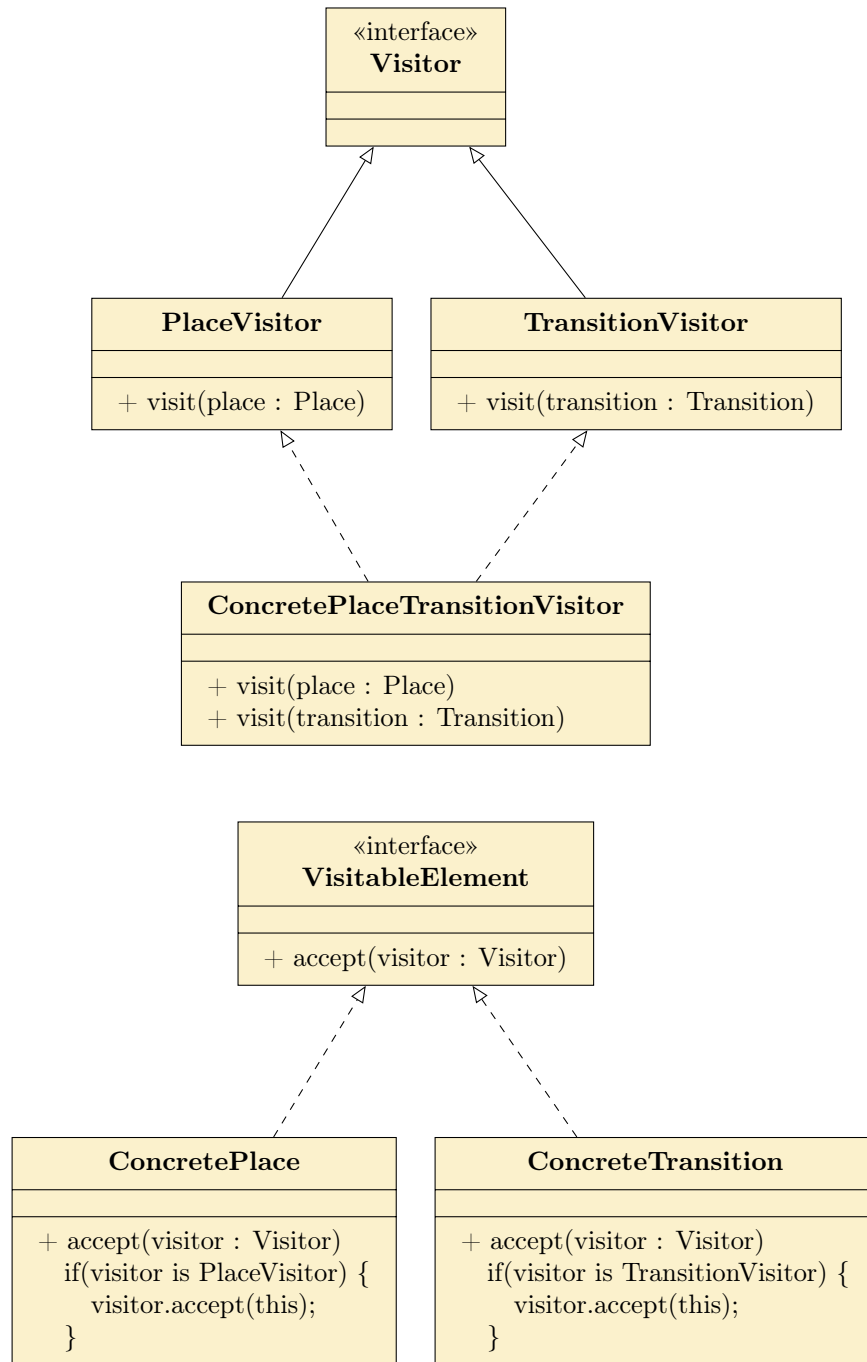


Figure 6.6: An example of the acyclic visitor design pattern applied to places and transitions. It differs from the visitor pattern seen in Figure 6.5 because concrete visitors now explicitly need to declare all component visitor interfaces they wish to visit. This design pattern now requires a cast in the concrete implementations of `VisitableElement` but the benefits of the acyclic visitor pattern outweigh the use of this cast.

### 6.2.3 The builder design pattern

The builder design pattern aims to separate the construction of complex objects from their creation and representation [21]. It does this via constructing an object in a step by step fashion, returning the finished product only in the final ‘build’ step. It removes the need for complex constructors exposed as an API to developers.

An example of the builder pattern in action can be seen in Figure 6.2; instead of passing a complex `HashMap` into the constructor of a `HashedState` we can build it up piece by piece with successive calls to `placeWithToken(...)`. This makes the API more readable and avoids a constructor anti-pattern.

We used the builder design pattern throughout the new PIPE 5 repositories to create complex objects.

### 6.2.4 Domain Specific Language

A Domain Specific Language (DSL), is an easy to use language that has a very specific purpose. For models it acts as a succinct, readable mechanism for expressing how a model is configured and adds a layer of abstraction on top of the model [40]. A good DSL makes it easier to understand what a model does and yields a clear sentence like construction of a model.

A DSL for building Petri nets has been incorporated into the PIPECore library in the aim to make building a Petri net read more like a sentence. The effectiveness of this DSL can be seen in Figure 6.7 which reduces the number of lines required to construct a Petri net by a third and more importantly dramatically increases the readability.

```

1 PetriNet petriNet = new PetriNet();
2 Token defaultToken = new ColoredToken("Default", Color.BLACK);
3 Token redToken = new ColoredToken("Red", Color.RED);
4 Place p0 = new DiscretePlace("P0");
5 p0.setTokenCount("Default", 1);
6 Place p1 = new DiscretePlace("P1");
7 p1.setTokenCount("Red", 1);
8 p1.setTokenCount("Default", 2);
9 Transition t0 = new DiscreteTransition("T0");
10 t0.setTimed(false);
11 Transition t1 = new DiscreteTransition("T1");
12 t1.setTimed(true);
13 t1.setRate(new NormalRate("#(P0, Red)"));
14 Map<String, String> arc1Weights = new HashMap<>();
15 arc1Weights.put("Default", "4");
16 InboundArc arc1 = new InboundNormalArc(p0, t0, arc1Weights);
17 Map<String, String> arc2Weights = new HashMap<>();
18 arc2Weights.put("Default", "#(P1)*2");
19 OutboundArc arc2 = new OutboundNormalArc(t0, p0, arc2Weights);
20 petriNet.add(defaultToken);
21 petriNet.add(redToken);
22 petriNet.add(p0);
23 petriNet.add(p1);
24 petriNet.add(t0);
25 petriNet.add(t1);
26 petriNet.add(arc1);
27 petriNet.add(arc2);

```

(a) Example code showing how to create Petri net components from their constructors in PIPE 5.

```

1 PetriNet petriNet =
    APetriNet.with(AToken.called("Default").withColor(Color.BLACK))
2     .and(AToken.called("Red").withColor(Color.RED))
3     .and(APlace.withId("P0").and(5, "Default").tokens())
4     .and(APlace.withId("P1").and(1, "Red").and(2, "Default").to())
5     .and(AnImmediateTransition.withId("T0"))
6     .and(ATimedTransition.withId("T1").andRate("#(P0, Red)"))
7     .and(ANormalArc.withSource("P0").andTarget("T0").and("4",
    "Default").tokens())
8     .andFinally(ANormalArc.withSource("T0").andTarget("P0")
9     .with("#(P1)*2", "Default").tokens());

```

(b) Example code showing how to create a Petri net with the DSL provided with PIPE 5.

Figure 6.7: Comparison code emphasising that using the domain specific language (DSL) shipped in the PIPECore library not only reduces the lines taken to create a Petri net by a third, but also significantly increases the readability.



## 6.3 A new implementation of functional weights

As described in Section 4.4.2 PIPE 4's support for functional expressions was terribly inefficient and a common source of bugs and user confusion.

PIPE 5 introduces an entirely new, fully extendable context-free grammar for functional expressions. Context-free grammars are the best way to handle functional expressions because they can be parsed using extensive tooling libraries. These libraries turn an expression into a parse tree which represents different nodes of the expression. It is then very easy to pick out and store the ids of referenced components, parse the tree for illegal operations and efficiently process the number of tokens in declared places.

### 6.3.1 ANTLR v4

Parsers work by grouping characters into tokens in a phase known as lexical analysis. These tokens are then analysed and a parse tree is created in the parsing phase. Nodes on the parse tree directly represent the structure of the BNF specified and the application can choose to process the tree however it likes, walking it multiple times if necessary.

ANTLR v4 is a parser generator that can read, process, execute or translate structured text or binary files. It takes a grammar and a formal language description and automatically builds parse trees, tree walkers and error listeners for your language. It uses a new parsing technology called Adaptive LL(\*) or ALL(\*) which performs grammar analysis dynamically at runtime rather than statically before the parser executes. While static analysis must consider all possible input sequences, dynamic analysis need only consider the finite collection of input sequences actually seen and thus avoids the undecidability of static LL(\*) grammar analysis. This new parsing strategy means that for a user, developing a grammar is much easier because ANTLR v4 handles ambiguity for you. For this

Expression	Meaning
$\#(P0)$	The sum of all tokens in the place with id P0
$\#(P0, \text{Default})$	The number of Default tokens in the place with id P0
$\#(P0, \text{Default}) * 10$	The number of Default tokens in the place with id P0 multiplied by 10
$\text{floor}(10.5/3)$	the floor of $10.5/3$ , i.e. 3
$\text{ceil}(\text{cap}(P0) * 2.4)$	The ceiling of the capacity (max number of tokens allowed) in the place with id P0 multiplied by 2.5
$\#(P0, \text{Default})$	The number of Default tokens in the place with id P0
$\#(P1) + \#(P2)$	The sum of the total number of tokens in the places P1 and P2

Table 6.1: Example usage of PIPE 5 functional weights. It aims to resolve the ambiguity between places token counts by giving the user the choice of all tokens within a place or the count of a specific token within a place.

reason ANTLR v4 was chosen for use in PIPE 5 due to its ease of use and its support for understandable BNF languages.

### 6.3.2 Grammar

PIPE 4's support for coloured tokens was somewhat complicated changing its logic depending on whether the expression was used in an arc or a transition. To avoid this ambiguity a new grammar was devised that allows the choice between all tokens in a place or a specific colour count within a place. The BNF for PIPE 5 can be seen in Listing 6.1 and an example of its usage can be observed in Table 6.1.

### 6.3.3 Implementing an API

As mentioned in Section 6.3.1 ANTLR v4 provides some auto-generated visitor interfaces that can be used to implement language applications. We extended one of these to create the `EvalVisitor` which performs the relevant operation for each node type it visits so that

```

grammar RateGrammar;

// PARSER
program : expression;

expression
    : '(' expression ')'           # parenExpression
    | expression op=('*'|'/') expression # multOrDiv
    | expression op=('+'|'-') expression # addOrSubtract
    | 'ceil(' expression ')'       # ceil
    | 'floor(' expression ')'      # floor
    | capacity                     # placeCapacity
    | token_number                 # placeTokens
    | token_color_number           # placeColorTokens
    | INT                          # integer
    | DOUBLE                       # double;
capacity: 'cap(' ID ')';
token_number: '#(' ID ')';
token_color_number: '#(' ID ',' ID ')';

// LEXER
ID      : ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9')*;
INT     : '0'..'9'+;
DOUBLE  : '0'..'9'+ '.' '0'..'9'+;
WS      : [ \t\n\r]+ -> skip ;

```

Listing 6.1: PIPE 5's new BNF grammar for functional expressions, written for ANTLR v4.

the final result of parsing the tree is a single numerical expression. Since the language for functional expressions behaves like a calculator these visited methods can be simple in their functionality. For those visitor methods who implement functional behaviour the relevant information can be obtained from the underlying Petri net model or the Markov chain state.

In order to expose a useful API two top-level classes were defined which can be seen in Figure 6.8. We wrote a `PertriNetWeightParser<Double>` which implements the `FunctionalWeightParser<T extends Number>` and creates the parsed result by doing the following:

1. Generates a parse tree. If for any reason a parse tree cannot be generated a specially written `RateGrammarErrorLister` produces informative error messages.
2. Next the parse tree is walked to get all referenced components. This uses a hand written `ComponentListener` to register the ids of any components seen. If any of the components referenced in the expression do not exist in the Petri net then relevant errors are produced and returned.
3. The tree is then walked again with the `EvalVisitor` to produce the actual result.

Results can then be queried and used as necessary by the API caller.

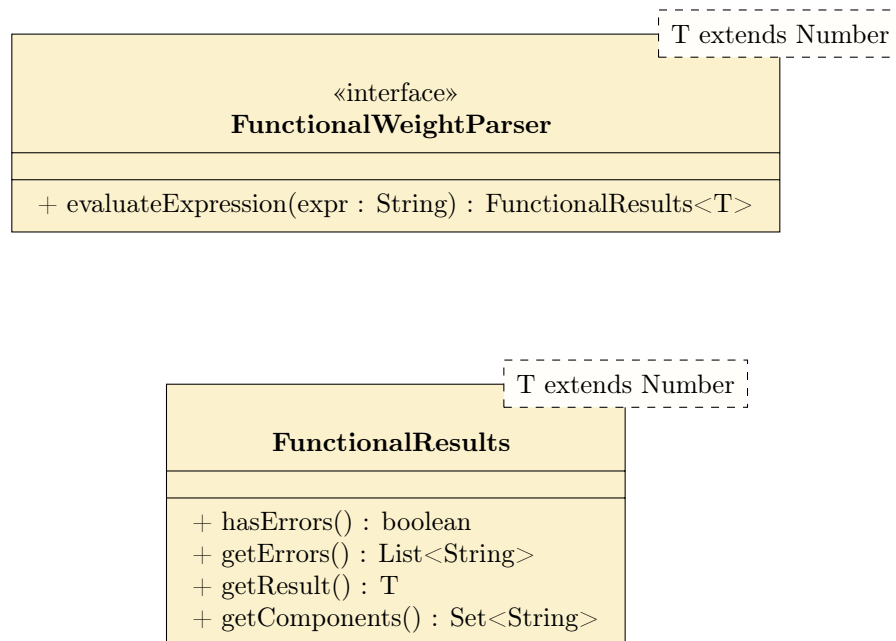


Figure 6.8: API exposed to users for parsing functional expressions. If the `hasErrors()` method of `FunctionalResults` returns true then `getResult()` is invalid and will not return a meaningful result.

## 6.4 Input/output

In order to support ICSI's requirement of a fully usable set of back-end classes the reading and writing of Petri nets needed to be addressed. PIPE 5 supports a new modern way of parsing models into XML via the JAXB library.

### 6.4.1 Parsing XML in Java

In order to use the Petri nets defined in an XML file a tool must first parse this file to produce a model of the data. The two main implementations available in Java to do this are described below.

### 6.4.1.1 JAXP

The Java API for XML Processing (JAXP) is used to process XML data using the Java programming language. It provides a set of low-level API's to parse XML documents. The ones relevant to parsing PXML are:

- **DOM** — the Document Object Model (DOM) API represents the entire XML in a tree structure in memory. This document class can then be parsed manually by the user. It is considered the easiest of the JAXP API's to use, but is the most CPU and memory-inefficient [41].
- **SAX** — the Simple API for XML (SAX) aims to be more memory and time efficient than that of a DOM parser. Instead of loading the entire XML into memory, events are triggered when XML elements are parsed, meaning that only part of the stream is in memory at a given time. In order to achieve this SAX works on the assumption that elements do not depend on previous elements in the tree [42]. To use this API the user must define their own `ContentHandler` whose major event-handling methods are: `startDocument`, `endDocument`, `startElement`, `endElement`, and `characters`. SAX works best when you only want to read the data as it comes in, rather than modify the structure of the data itself. Due to its implementation it can make processing a document more tricky.
- **StAX** — the Streaming API for XML (StAX) is a hybrid between SAX and DOM and is the latest addition to the JAXP libraries. It is event-driven, like SAX, but instead of using a push parser architecture it provides state-dependent processing with the use of a bidirectional pull parser interface that allows the client application to make method calls on the API. This makes StAX as easy to use as DOM but provides a more efficient implementation.

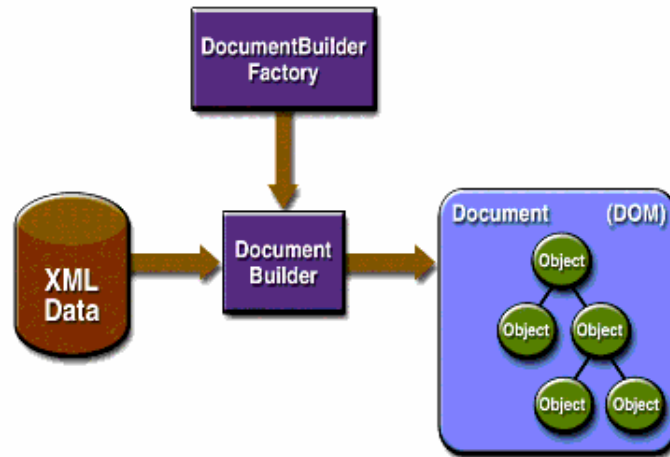


Figure 6.9: JAXP DOM API's for processing XML files [43].

#### 6.4.1.2 JAXB

The Java Architecture for XML Binding (JAXB) is a high-level API allowing Java classes to be marshalled to XML format and unmarshalled into Java objects. In order to achieve this class fields and methods can be marked up via the use of various annotations, for example `@XmlElement` and `@XmlAttribute`. Under the hood JAXB makes use of JAXP, but provides a higher-level implementation of XML parsing.

### 6.4.2 Implementation

After a review of the common XML parsing libraries we chose JAXB over JAXP since its use of annotations is more modern and much easier to use than JAXP.

In order to directly annotate the Petri net components JAXB requires the set of classes to be Java Beans. This means they need a default no-argument constructor and must have getter and setters for all field properties. In PIPE 5 it would be illegal to create a Petri net

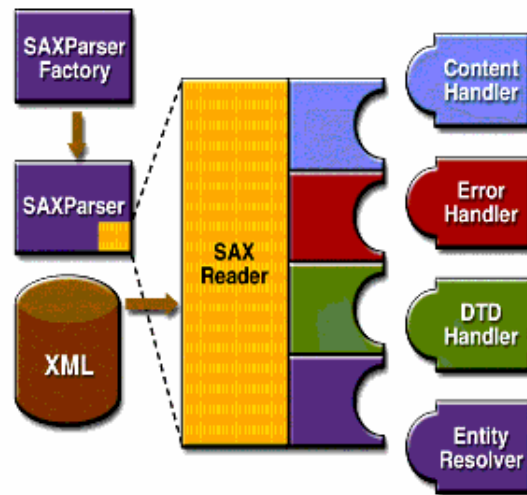


Figure 6.10: JAXP SAX API's for processing XML files [43].

component without an id so instead of allowing a default no-argument constructor a set of adapted components which model the exact structure of the PNML layout have been created. These adapted components contain the `@XmlElement` and `@XmlField` annotations according to the PNML specification. Along with the adapted classes a set of adapters were written to marshal adapted components into their real component classes and vice versa. Since PNML is very verbose, with component attributes being wrapped in several levels of XML tags, this design decision benefited our Petri net component models by allowing them to have a flat attribute structure rather than requiring their fields to be wrapped in objects that merely represent the PNML hierarchy.

Since multiple Petri nets are allowed in a single PNML file, a `PetriNetHolder` was created to contain each Petri net read in from an XML file. This class has the annotation `@XmlRootElement(name = "pnml")` which refers to the top level class of the XML file.

Once the adapters have been linked to class types the actual code to read and write Petri nets can be implemented in 8 lines or less. This is shown in Listing 6.2 and is exposed to



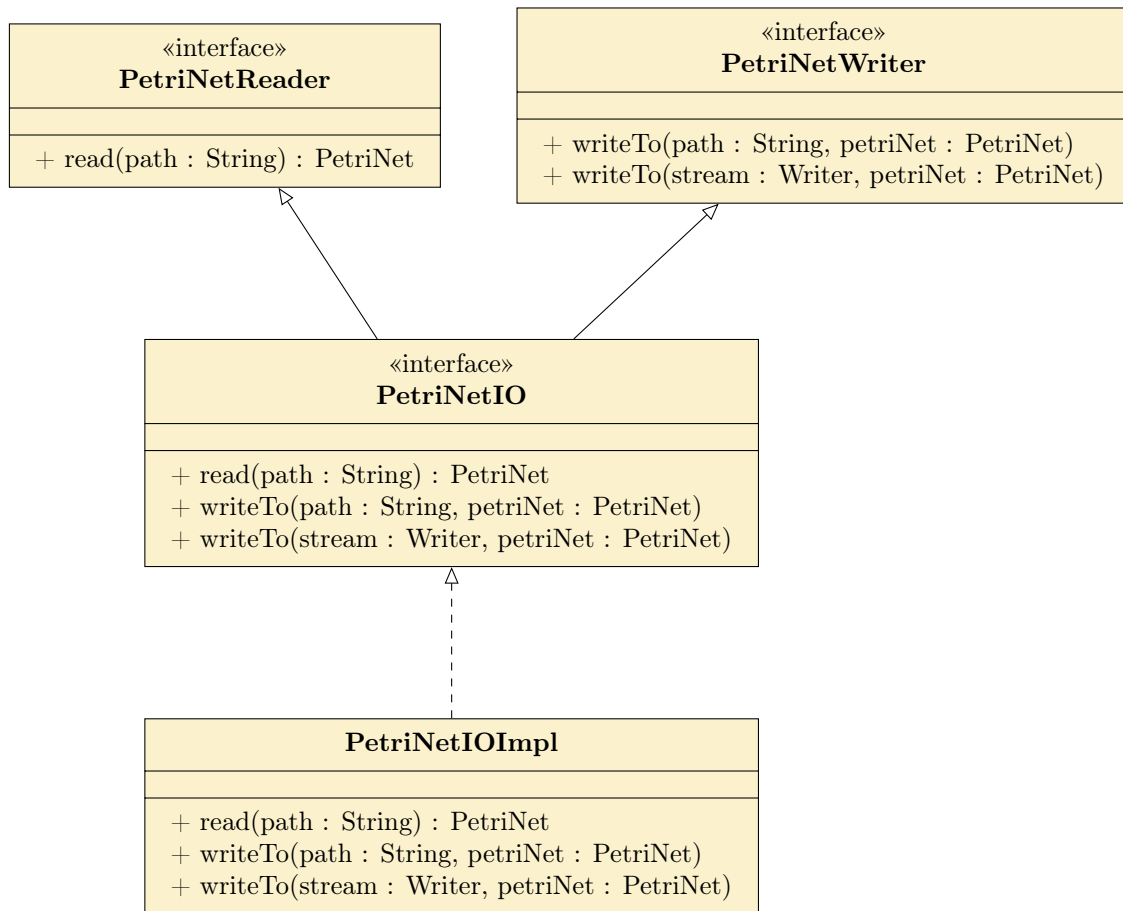


Figure 6.11: The exposed API in PIPE 5 to read and write Petri nets.

the user in the API seen in Figure 6.11.

```

1 public class PetriNetIOImpl implements PetriNetIO {
2     private final JAXBContext context;
3
4     public PetriNetIOImpl() throws JAXBException {
5         context = JAXBContext.newInstance(PetriNetHolder.class);
6     }
7
8     @Override
9     public void writeTo(Writer stream, PetriNet petriNet) throws
10         JAXBException {
11         Marshaller m = context.createMarshaller();
12         m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
13         PetriNetHolder holder = new PetriNetHolder();
14         holder.addNet(petriNet);
15         m.marshal(holder, stream);
16     }
17
18     @Override
19     public PetriNet read(String path) throws JAXBException,
20         FileNotFoundException {
21         Unmarshaller um = initialiseUnmarshaller();
22         PetriNetHolder holder = (PetriNetHolder) um.unmarshal(new
23             FileReader(path));
24         PetriNet petriNet = holder.getNet(0);
25         if (petriNet.getTokens().isEmpty()) {
26             Token token = createDefaultToken();
27             petriNet.addToken(token);
28         }
29         return petriNet;
30     }
31
32     private Token createDefaultToken() {...}
33     private Unmarshaller initialiseUnmarshaller() throws JAXBException
34         {...}
35 }

```

Listing 6.2: An example of the code needed to read and write Petri nets in PIPE 5 using JAXB. The code is contained in a single class and only requires a total of 5 lines to write a Petri net and 8 lines to read a Petri net. PIPE 4's read implementation was embedded in an 84 long line method in the `PetriNetView` and its write implementation involved a 637 line long class called `PNMLWriter`. This new implementation is much cleaner.

## 6.5 Documentation

This section describes how we increased the level of documentation in PIPE 5 to aid both users and developers.

### 6.5.1 GitHub pages

PIPE 5 supports new online documentation that provides a modern alternative to the basic HTML documentation previously embedded into PIPE 4. We hosted the documentation directly from PIPE 5's GitHub repository using its GitHub pages feature. The new documentation can be accessed at [http://sarahtattersall.github.io/PIPE/user\\_guide.html](http://sarahtattersall.github.io/PIPE/user_guide.html). A screenshot of the user guide is displayed in Figure 6.12.

The code for the documentation is located in the `gh-pages` branch of the PIPE repository. Any changes to this are picked up by a Git post-commit hook and immediately published.

The underlying code behind PIPE 5's new user guide is written using the Jekyll blogging framework which produces static sites from Markdown or HTML formatted files. Under the hood Jekyll makes use of the Liquid templating engine to provide reusable code templates and simplify operations within the static files.

We make use of the Jekyll framework to allow for easy additions to the user guide. Each section is stored in its own file under the `_posts` directory and has the following format:

```
---  
layout: post  
title: <title>  
post-id: <edit>  
---
```

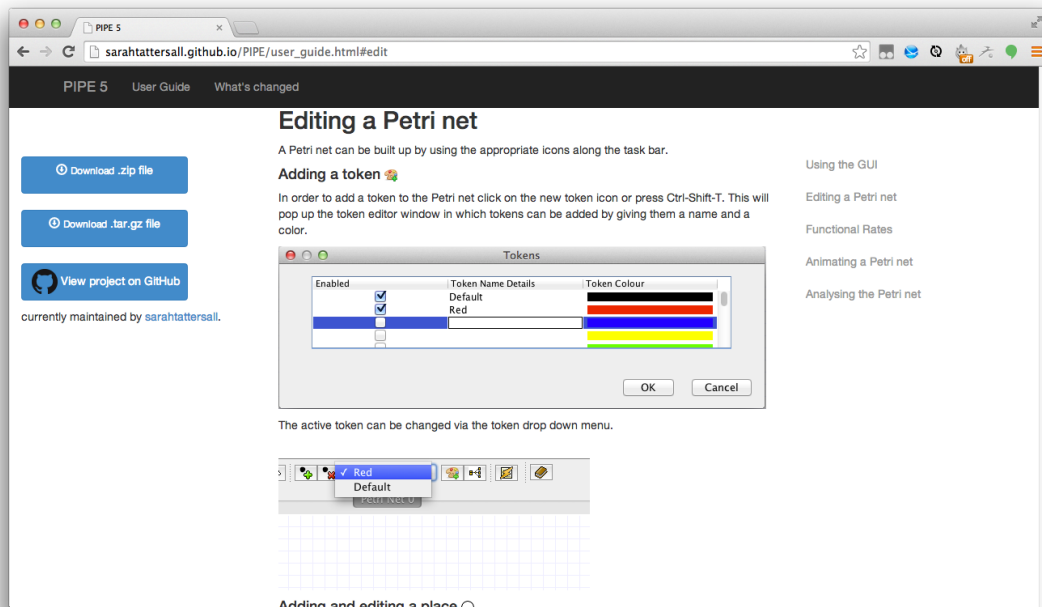


Figure 6.12: A screenshot of the new PIPE 5 user-guide hosted on GitHub pages.

```
<content>
```

All posts are then dynamically incorporated into the user guide using Liquid tags shown below:

```
{% for post in site.posts reversed %}
<div class="row-fluid">
  <h2 id="{{post.post-id}}">
    {{ post.title }}
  </h2>
  <div class="span12">
    {{ post.content }}
  </div>
</div>
{% endfor %}
```

This means that in the future when new features are added to the user guide they are automatically included. This is a major benefit for easy maintenance.

## 6.5.2 Javadoc

Whilst developing PIPE 5, a great deal of care was put into developing good documentation for the written code. Javadoc is the official documentation generator from Oracle for generating API documentation from Java source code and is embedded between `/** ...*/`. Javadoc supports useful annotations such as `@param`, `@return`, `@throws` which give rise to a clear and concise comment structure.

## Chapter 7

# Re-engineering the existing view code

You can't throw money at a Petri net to make it work — it really is all about the quality of the content.

Chris Hardwick (paraphrased)

In order to integrate the new model structure and accompanying classes into the existing view code some significant changes needed to be made. These changes are discussed in the following sections.

### 7.1 Modifying the architecture

#### 7.1.1 Applying a model–view–controller architecture

Since the entire logic of a Petri net in PIPE 4 resided in the views all this code had to be eradicated and replaced with custom listeners to respond to changes in the new Petri

net component models. The listeners have been set up to trigger a change in their view when the models they are registered to fire a change message. Different actions are then performed depending on the message's content. These listeners form part of the publish-subscribe architecture.

We implemented an entire set of Petri net component controllers to combat the number of 'God' classes reported in PIPE 4 by the QAPlug IntelliJ plug-in. By having a set of specialised classes who are responsible for changing only their underlying model, there is no one class that tries to modify all components in the MVC architecture.

We went on to create individual action classes for all GUI actions along the tool bar rather than having a single class that tried to cover several actions. Whilst this increases the class count, it significantly simplifies the purpose of each class, which is arguably better.

### 7.1.2 Eradicating the `ApplicationSettings` class

Eradicating all 280 uses of the `ApplicationSettings` class was a particularly difficult task. We became acutely aware of the poor architectural structure of the views and found that the method calls to the `ApplicationSettings` class had become deeply nested within the call stack.

In order to eventually rid PIPE of the `ApplicationSettings` class a technique called dependency injection was applied to the codebase. This technique stops objects from creating their own dependencies and instead has them receive them through their constructor or as method parameters. One example of this technique was the restoring of the `IModule` interface to its original implementation as described in Section 4.2.1; its `start` method now requires a Petri net as a parameter.

Unfortunately due to the poor design of class interaction it was very difficult to inject some of the objects straight into classes who directly use them. We had to settle for a temporary

work around by storing references of objects in places they did not belong just to be able to make method calls to classes that needed them.

However, once the `ApplicationSettings` class was removed, testing became a lot easier as it allowed for the mocking of injected classes and removed race conditions in the test suite due to a global state. It also avoids the views from becoming even more tangled due to this class.

### 7.1.3 Replacing custom code with third party libraries

In many places PIPE 4 defined custom code for functionality that is defined in the Java core libraries and common third party libraries. We have replaced this custom code with the applicable libraries where possible because commonly used open-source libraries tend to be well tested, correct and have an easy to use API.

As an example PIPE 4 implemented its own `HistoryManager` and `HistoryItem` classes for managing the undo/redo functionality in the GUI. Unfortunately the `HistoryManager` fronted a very awkward API for handling the case of multiple changes requiring a single undo action.

These have now been replaced by the `UndoManager` and `UndoableEdit` classes that have shipped with the Java core libraries since Java 5. These classes have been rigorously designed and tested and provide a much more flexible API for performing undo and redo actions. We re-wrote all of the individual undoable actions and the undo behavior to use these classes.



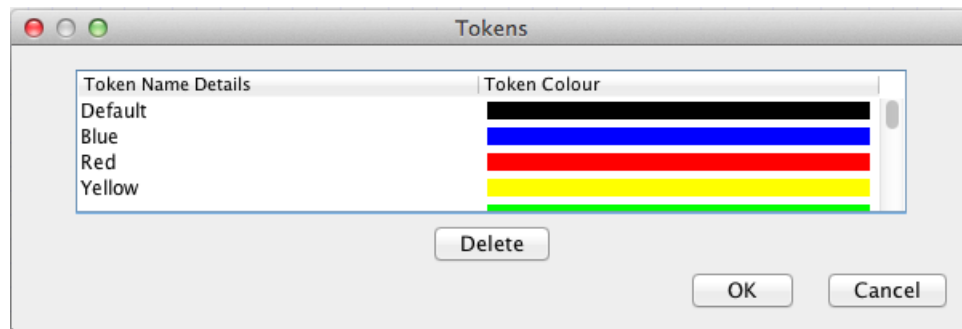


Figure 7.1: The modified PIPE 5 token editor panel. The option to delete a token has been made available. The user can simply highlight the row of the token they wish to delete and then press the delete button.

## 7.2 Changes in displayed components

Some changes were made to the way that Petri net components were displayed in the GUI. They are detailed below.

### 7.2.1 Deleting the tokens

In PIPE 4 tokens could be enabled and disabled but they could never be deleted from a Petri net. This means any tokens created in PIPE would persist in a Petri nets PNML whether they were really needed or not. The only way to remove them would be to manually delete them out of the XML file.

To overcome this problem the enabled field was removed from the tokens logic, and instead a controller method to delete the token from the Petri net was set-up. The changes to the token editor can be seen in Figure 7.1.

If a user tries to delete a token that is contained in a place or referenced in a functional expression then the `PetriNet` class handles this in its `deleteToken` method and throws a `PetriNetComponentException`. This exception is caught by the token editor in the GUI and

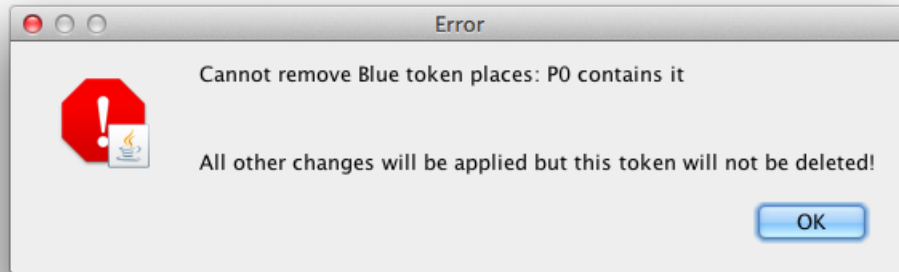


Figure 7.2: PIPE 5 token editor error message when trying to delete a token that is used in the Petri net. In this particular instance a place contained the blue token.

the error message is shown accordingly. Figure 7.2 shows the error message displayed when trying to delete a token that exists in a place.

## 7.2.2 Changing the rate parameters

In PIPE 4 it seemed strange that rate parameters were placed on the canvas of a Petri net in a similar way to annotations as it tended to clutter the layout. In PIPE 5 rate parameters have been migrated to the tool bar and have a new editor that is very similar to the token editor (Figure 7.3).

If a user enters an expression for a rate parameter that is invalid they are notified when trying to exit the editor as shown in Figure 7.4. This means that no invalid rate parameters can be introduced into the Petri net. Furthermore if a rate parameter that is in use is deleted, instead of throwing an error all transitions referring to this parameter have their rate set explicitly to the expression contained inside the rate parameter.

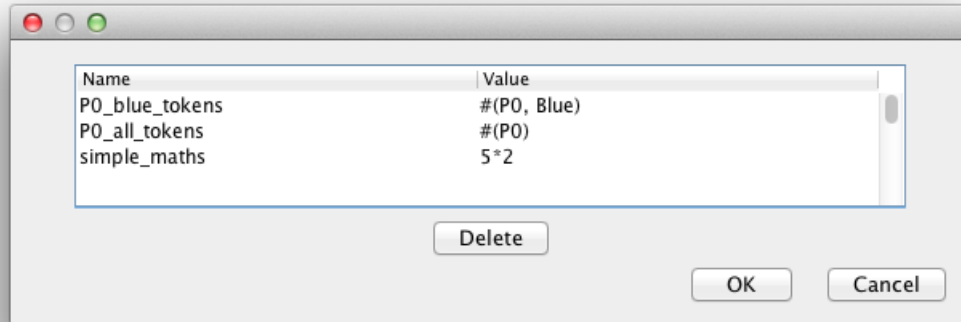


Figure 7.3: PIPE 5 new rate parameter editor.

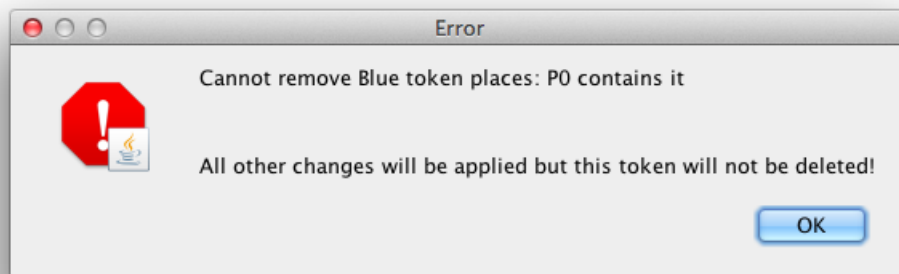


Figure 7.4: PIPE 5 token editor error message when trying to delete a token that is used in the Petri net. In this particular instance a place contained the blue token.

### 7.3 Useful errors

Quite a lot of error messages in PIPE 4 were nondescript and did not alert the user to what they had actually done wrong. PIPE 5 tries to address these changes by incorporating exceptions into the back-end models and catching them in the GUI code.

An example of this can be seen when deleting a place the Petri net model. The Petri net model throws an exception if it is referenced in any functional expressions. The PIPE 5 GUI code then catches this error and displays the message contained in the error which says: “Cannot delete <place> as it is referenced in a functional expression!”

## Chapter 8

# New analysis modules

There was a time when people felt the Petri net was another world, but now people realise it's a tool that we use in this world.

Tim Berners-Lee (paraphrased)

PIPE 5 supports new implementations of the state space exploration and steady state solver modules. These modules have been redesigned to provide a more object-orientated feel to the algorithms, to increase their reliability and to provide faster run-times by exploiting multi-core processors.

### 8.1 State space exploration

The state space exploration algorithm has been redesigned to increase the readability and understandability of the code and to improve confidence in the implementation of the algorithm through unit and integration tests. Two implementations of the algorithm have been developed: the previously described sequential algorithm and a newly developed MapReduce-style parallel algorithm.

With the addition of the new Markov chain state classes, who map a place id to a map of token ids and their count within the place, PIPE 5 can now process coloured Petri nets giving far more flexibility to the analysis of a Petri net.

### 8.1.1 Sequential state space exploration

PIPE 5's sequential implementation follows the same algorithm as described in Section 4.2.2, but has a more object-orientated structure than PIPE 4's implementation. The new API for the exploration analysis can be seen in Figure 8.1.

The job of the `AbstractStateSpaceExplorer` is to provide common methods and fields for manipulating the state space and to perform the initial step of exploring the first state. The iterative step is then delegated to the underlying subclass, which in this case is the `SequentialStateSpaceExplorer`. In order to allow users to optionally include vanishing states in their graph, the `VanishingExplorer` API (Figure 8.2) was introduced. This API encapsulates the logic of whether to process only tangible states or to include vanishing states in the state space too. The logic for generating the successors of a state has also been abstracted out of the algorithm so that the code in the `SequentialStateSpaceExplorer` is highly reusable and extremely understandable. The iterative step of the algorithm is now only a mere 16 lines long, as demonstrated in Listing 8.1, compared to the `StateSpaceGenerator` in PIPE 4 which is 1341 lines long.

In order to deal with a potentially infinite state space in the reachability graph the `ExplorerUtilities` interface was introduced. All concrete classes are responsible for calculating successor states and help to determine if the algorithm can continue for another iteration. There are three implementing classes the user can choose from which provide different functionality:

- `UnboundedExplorerUtilities` — in the simplest form this class returns exactly the

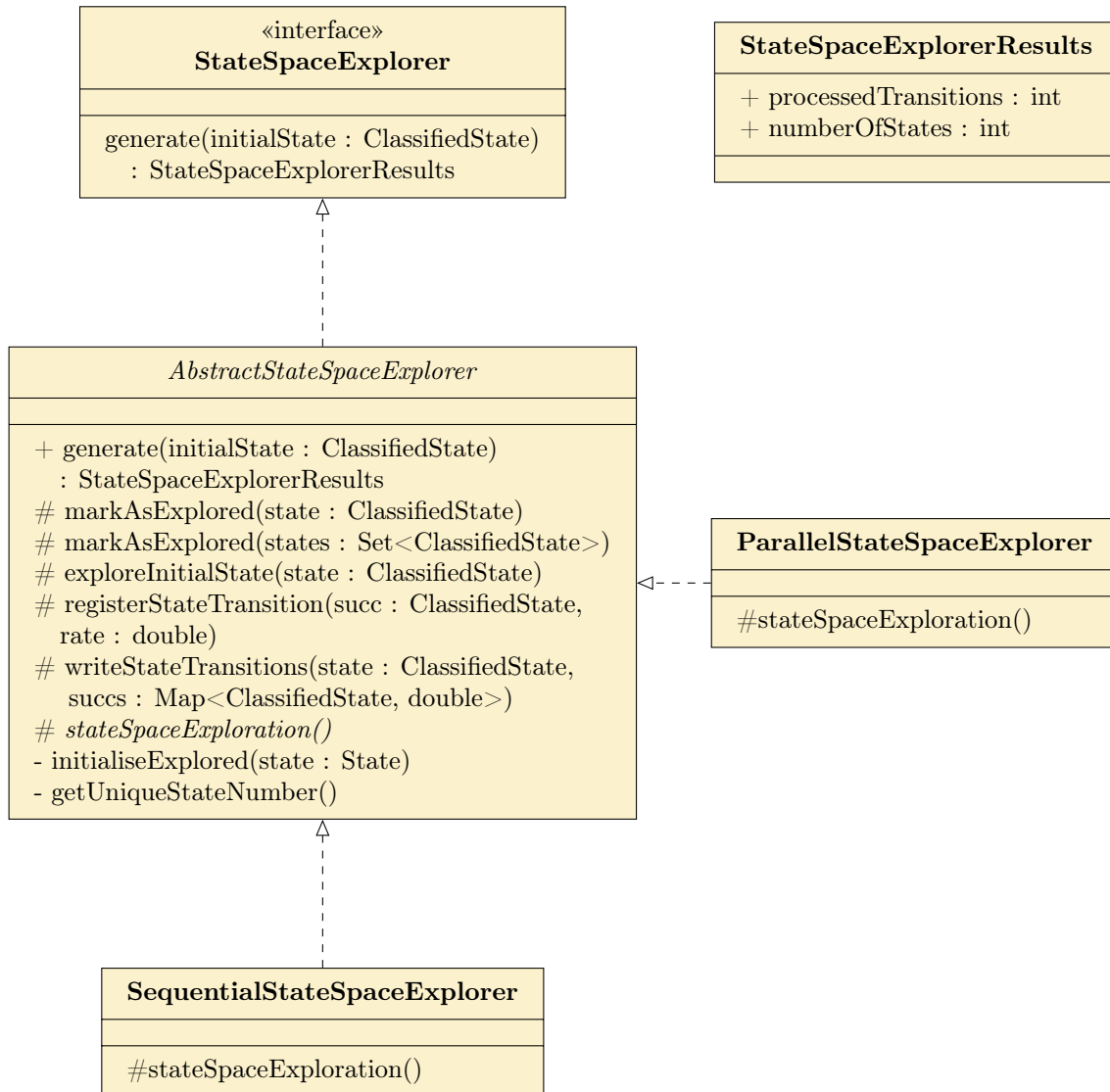


Figure 8.1: The steady state exploration API that is exposed to the developer and the set of classes that implement this API. The `AbstractStateSpaceExplorer` contains as much of the shared code as possible to ensure that subclasses only need implement their algorithm logic.

```

1 public final class SequentialStateSpaceExplorer extends
   AbstractStateSpaceExplorer {
2
3 public SequentialStateSpaceExplorer(ExplorerUtilities
   explorerUtilities, VanishingExplorer vanishingExplorer,
   StateProcessor stateProcessor) {
4     super(explorerUtilities, vanishingExplorer, stateProcessor);
5 }
6
7 @Override
8 protected void stateSpaceExploration() throws TimelessTrapException,
   IOException, InvalidRateException {
9     while (!explorationQueue.isEmpty() &&
   explorerUtilities.canExploreMore(stateCount)) {
10        ClassifiedState state = explorationQueue.poll();
11        successorRates.clear();
12        for (ClassifiedState successor :
   explorerUtilities.getSuccessors(state)) {
13            double rate = explorerUtilities.rate(state, successor);
14            if (successor.isTangible()) {
15                registerStateTransition(successor, rate);
16            } else {
17                Collection<StateRateRecord> explorableStates =
   vanishingExplorer.explore(successor, rate);
18                for (StateRateRecord record : explorableStates) {
19                    registerStateTransition(record.getState(),
   record.getRate());
20                }
21            }
22        }
23        writeStateTransitions(state, successorRates);
24        explorerUtilities.clear();
25    }
26 }
27 }

```

Listing 8.1: Java implementation of the sequential state space exploration breadth first search in the PIPE 5 analysis module. The method performs the iterative loop of processing states in the exploration queue once the initial state has been added. The original analysis module contained over 1000 lines of code for the algorithm, but this implementation achieves it in a meager 16 lines by delegating responsibilities to other classes making the algorithm far more clear. Note Javadoc has been removed from this class for report purposes only and is present in the actual code.



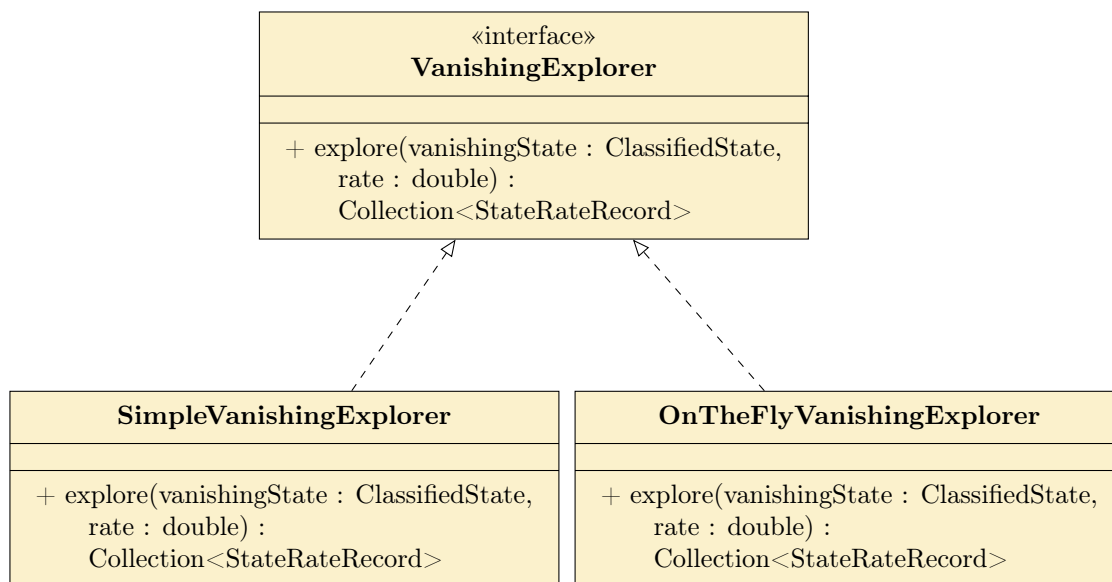


Figure 8.2: Vanishing explorer API exposed to users. The `explore` method is used to return a set of states to explore. The `SimpleVanishingExplorer` simply returns the vanishing state so it is processed in the main algorithm, just like a tangible state in Algorithm 1. The `OnTheFlyVanishingExplorer` on the other hand, removes all vanishing states from the state space by only returning the tangible states reachable from the given vanishing state. In the case of a cycle of vanishing states the tangible states reachable have their rates summed until the overall rate has dropped below a certain value.

successors of a state and also allows the algorithm to run until finished.

- `BoundedExplorerUtilities` — provides the same results as `UnboundedExplorerUtilities`, but contains a limit to the number of states the algorithm is allowed to process. Going over this number will terminate the algorithm.
- `CoverabilityExplorerUtilities` — this modifies the successors of a state by setting those that cover a state to a programmatic equivalent of  $\omega$  so that the algorithm has a finite state space. It does not impose a limit on the number of states explored.

During the algorithm states are given a unique id, different to their double hash, which is implemented by a monotonically increasing function `getUniqueStateNumber()`. This id is

assigned to a state when it is added to the explored set and is an enhancement suggested in [13] to speed up writing the transitions to disk. The technique works by writing the integer id in place of the state and avoids serializing the entire state many times. It gives good results because a single integer is far quicker to write than a whole class.

For example if state 1 has transitions to states 2, 3, and 4 with rates 2.0, 3.0, and 4.0 respectively we can write to disk a binary representation of 1 to (2, 2.0), (3,3.0), (4, 4.0). In order to not loose any information about the state two streams are necessary when performing state space exploration:

- A state mapping stream — this maps the unique integer id of a state to the physical class itself.
- A transition stream — this maps integer states to their successors by writing the state followed by the number of successors followed by pairs of successor and rate.

A set of interfaces and classes for reading and writing states and transitions has been implemented alongside the state classes in the PIPEMarkovChain library.

Finally in order to ensure correctness of our algorithm we implemented integration tests using the Cucumber framework. An example of these can be seen in Listing 8.2.

We chose the Cucumber framework for our integration tests over JUnit because of the added readability provided by the Gherkin language. Cucumber works by allowing integration tests to be written as a series of sentences containing the Given, When, Then and And keywords provided by the Gherkin language. These sentences are handled by developer written `StepDefinition` classes which map the sentences to a method via annotations and regular expressions. When Cucumber executes each scenario it runs the underlying method for each sentence.

```
Feature: state space exploration of tangible states only

@tangibleOnly
Scenario: Parsing a simple Petri net file
  Given I use the Petri net located at /simple.xml
  When I generate the exploration graph sequentially
  Then I expect to see 2 state transitions
  And I expect a record with state
    """
    { 'P0' : { 'Default' : 1 }, 'P1' : { 'Default' : 0 } }
    """
  And successor
    """
    { 'P0' : { 'Default' : 0 }, 'P1' : { 'Default' : 1 } }
    """
  And rate 1.0
  And I expect a record with state
    """
    { 'P0' : { 'Default' : 0 }, 'P1' : { 'Default' : 1 } }
    """
  And successor
    """
    { 'P0' : { 'Default' : 1 }, 'P1' : { 'Default' : 0 } }
    """
  And rate 1.0
```

Listing 8.2: An example Cucumber integration test written in Gherkin for the state space exploration algorithm. We designed the step definitions to flow like an English sentence and found the best way to describe a Markov chain state was by expressing it in JSON.

### 8.1.2 The explored states data structure

In order to give the reachability analysis module a more object-orientated feel the explored states set has been implemented as its own data structure. Its API exposes methods to store a state and its id, to query if a state is contained in the set and to retrieve the id for a given state.

Instead of writing the two hash functions by hand PIPE 5 uses Google Guava's hashing utilities library since it provides a range of benefits Java's hash code implementation lacks [44]:

- Java's implementation of hash codes is constricted to 32 bit integers, Guava removes this constraint by implementing a `HashCode` class to provide support for larger hashes. This immediately provides a larger bit dispersal and reduces the probabilistic chance that two states will suffer from the same primary and secondary hash functions.
- Hand written hash codes tend to have weak collision prevention and no expectation of bit dispersion. Whilst this is acceptable for use in a simple hash table, for PIPE equality of states depends on these hash codes so a better dispersed hash code is probabilistically better.
- Guava's hash code implementations can be swapped in and out for different ones without re-writing any code. In order to hash any given item a funnel must be written which describes how to decompose an object into its primitive field values. This funnel can then be used with any hashing algorithm.

In order to determine which hash functions to use, results of external experiments were taken into consideration [45,46]. Whilst `xxHash` looked the most promising, with the native implementation being the fastest, in practice it gave inconsistent results when hashing the same state and is not currently implemented in the Guava libraries [45]. Following this

an investigation into Google's CityHash algorithm for strings showed that at present there is also no Java implementation due to API issues [47] which ruled it out too. Out of the remaining hashing algorithms in the comparisons `alder32` and `crc32` prove to be the fastest although they come unrecommended as they are not cryptographically secure. In practice we saw some hashing collisions when using `alder32` and `crc32` together for Petri nets over 50 000 states large so we have chosen `crc32` and `murmur_128` for the primary and secondary hash functions respectively. This is an accuracy-speed trade-off.

In order to store the secondary hash and state id information in memory an array of `TreeMap` items was created, mapping the secondary hash to the unique id. The `TreeMap` in Java is essentially a red-black re-balancing tree and was chosen over the linked list implementation explained in Section 4.2.2 because it yields  $O(\log n)$  average and worst case complexity look up times versus  $O(n)$  for a linked list. This is important for two reasons:

1. Whilst an insert into a linked list can be performed in constant time opposed to  $O(\log n)$  for a red-black tree, insertions happen significantly less in the state space exploration than lookups. This means that lookup times are the more important factor.
2. When the array becomes saturated we cannot dynamically expand it due to losing the original state information during hashing. This means that on saturation fast look up times become even more important as the data structure grows in size.

The size of the array is also very important to ensure a good dispersal of data. It can be mathematically shown that the constant used in the hash algorithm and the size of the array should be co-prime [48]. In the case of the complex hashing libraries used in PIPE 5 this constant is unknown so it is safest to use a prime number for the size of the array. Since we take the modulus of the first hash, the larger we make the size of the array the smaller the probability is of a collision assuming our hash algorithms provide a uniform distribution. We must however make a trade-off between memory and collision

probabilities. Originally we tried the number 300 007 but occasionally found states whose primary hash mod the array size and secondary hash were equal. A process of trial and error led us to increase this value to 358 591.

### 8.1.3 A parallel implementation

It follows from the sequential implementation that the breadth first search can be parallelised across many different threads.

At first Hadoop was considered for this problem since their MapReduce algorithm handles graph problems very well. However, after some research, it became apparent that Hadoop is more suited to distributed programming across many machines rather than a single machine using multiple cores since it has a large start up overhead to convert the data to the Hadoop File System format. Instead simple Java threads are more than sufficient for this type of problem.

The thread base algorithm we developed takes inspiration from Hadoop and is in the form of a MapReduce problem. In order to process states in parallel the main thread starts up a specified number of worker threads, normally one for each virtual core, and allows blocks waiting for them to return. Once running each thread performs the sequential breadth first algorithm, keeping track of the state transitions which it has processed. In order to try and avoid threads processing a state more than once we make use of a concurrent non-blocking shared queue. We allow each worker thread to run until it has processed a specified number of states or the exploration queue is empty after which it returns with the data it has collected. The main algorithm then collects the results as threads finish, writes them to disk, stores the explored states in our explored set data structure and starts another iteration off. This continues until the state space has been fully explored.

This algorithm highlights the need for the Markov chain state classes we developed as they

allow analysis to take place on multiple states at a time rather than using the underlying Petri net structure. The processing of a Petri net's underlying structure would have led to race conditions. Instead the abstraction of a Petri net into a state space handles concurrent algorithms very nicely.

#### 8.1.4 Improving the performance

In order to improve performance of the state space exploration algorithm the code was profiled using the YourKit Java profiler which led to the following changes being made:

- YourKit flagged up an unnecessary amount of time was spent calculating a states enabled transitions. The existing implementation of the algorithm often made several calls to this method whilst processing a state. For example when calculating its successors, when classifying the state and so on. Although we could have passed around the set of transitions enabled for a state we decided it was cleaner for the API not to expose these transitions. We therefore applied a technique called memoization, which caches a state and its transitions, in order to reduce the time spent calculating subsequent queries about the same state. Whilst this speed improvement could be seen as counter-acting the memory gains we make using an exploration set, we have implemented a `clear` method in the API so that the cache can be cleared after each iteration.
- Following this YourKit then flagged up that 22% of the algorithms run-time was spent calculating inbound and outbound arcs of a transition, as can be seen in Figure 8.3. On inspection of the Petri net implementation we found that it was iterating through every arc to find any that were connected to the transition specified. To reduce this to a constant look up time two maps have been created to store each transition's id to its inbound and outbound arcs. These are then populated and maintained when arcs are added or removed from the Petri net or when a transition's id changes.

- Another round of profiling showed that the algorithm was spending 19% of its time parsing the functional expressions to determine if an arc can fire. Since many of the expressions in a Petri net are likely to be numerical, we added a call to `Doubles.tryParse(expression)` to avoid walking the tree if the functional expression is entirely numerical. This small change added a 1.5x speedup for our parallel algorithm using 4 virtual cores for numerically weighted Petri nets.
- Finally YourKit showed that a lot of time was spent adding and retrieving items from the hash maps used as caches. On closer inspection we saw most of the time was spent determining the equality of the states. Since the double probabilistic-hashing method has a significantly low false positive rate we decided to use the two hash functions for determining equality of states everywhere, not just in the explored set. Now the states pre-calculate their own primary and secondary hashes on construction and use them in their `hashCode` and `equals` methods. This has the added bonus of speeding up the explored sets data structure up as it no longer needs to calculate a state's hash code on the fly. This change saw a 3x speedup for the algorithm when running on four virtual cores.

We then investigated changing the underlying implementation of the concurrent hash maps used as caches. Dr. Cliff Click invented a non-blocking (concurrent, lock-free) map that claims to scale far better than Java's `ConcurrentHashMap` by using a State-Machine based algorithm [49]. We tried replacing any uses of a `ConcurrentHashMap` with the `NonBlockingMap` and surprisingly saw a slight speed decrease. On profiling the code YourKit indicated that calls to `put` and `get` in the `NonBlockingHashMap` often took longer to run and so we decided to stick with the Java 7 implementation of a concurrent hash map. We assume the reason for this speed decrease is because we clear the caches after each iteration of our main threads loop and thus never reach the types of scale that the non-blocking hash map was designed for.



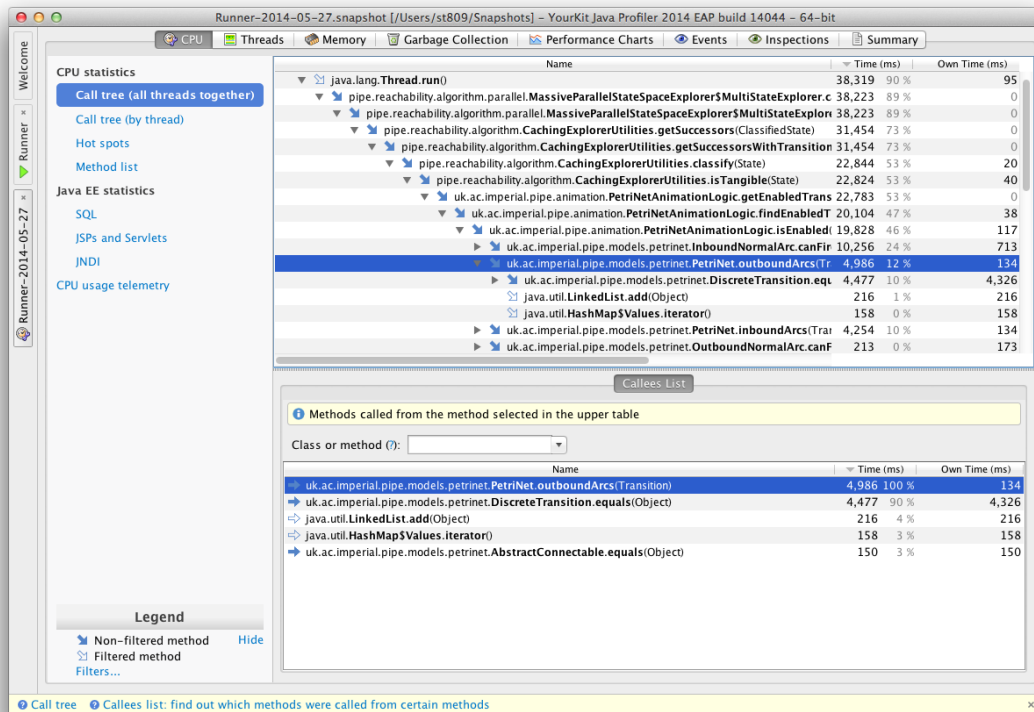


Figure 8.3: A screenshot of YourKit after gathering CPU run-time results. The highlighted line shows that 12% of the algorithms time is spent calculating the outbound arcs for transitions. Similarly four lines down it can be seen that calculating the inbound arcs takes up 10% of the time, yielding a total of 22% time spent calculating a transitions inbound and outbound arcs.

### 8.1.5 PIPE 5 reachability graph UI

The state space exploration module supports a new user interface in PIPE 5 which aims to increase the flexibility of generating reachability and coverability graphs and also tries to address the issues described in Section 4.2.2.

The new UI now provides the option to generate both a reachability and coverability graph of a Petri net. Since the calculations for the coverability graph only work with SPNs, we only allow this graph to be generated for a Petri net containing entirely timed transitions. Furthermore due to the introduction of an approximate maximum state as explained in Section 8.1.1 the algorithm will no longer hang for unbound Petri nets.

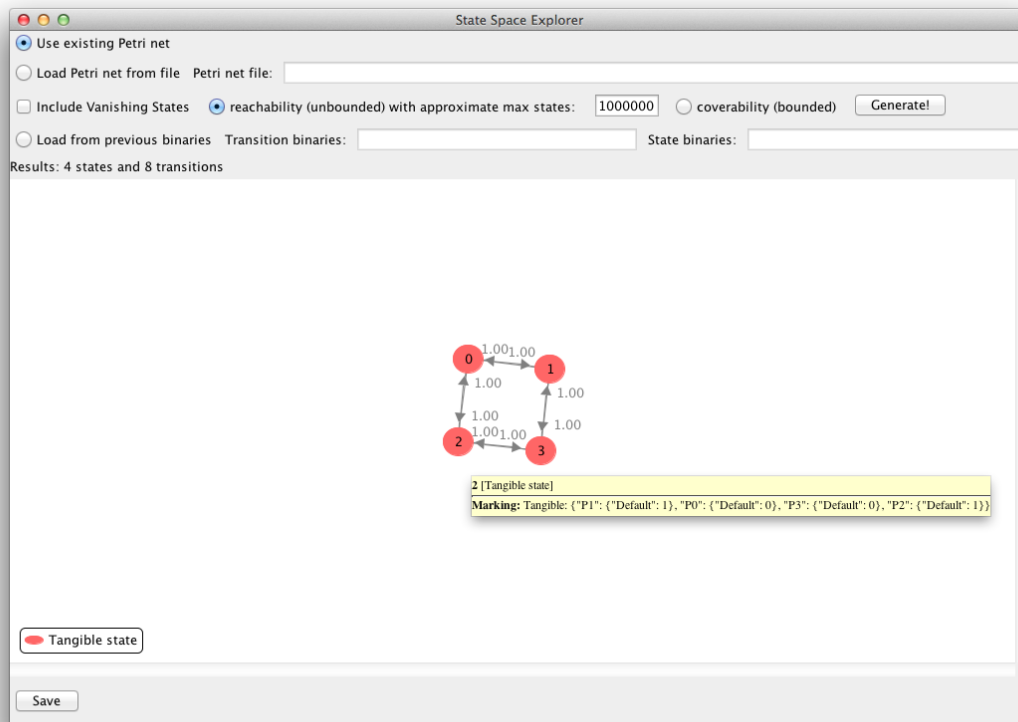


Figure 8.4: The new PIPE 5 state space exploration UI displaying a reachability graph for the Petri net shown in Figure 2.8. The tool uses the JPowerGraph library to display the visual representation of the reachability graph, as in PIPE 4, and as a result of hovering over state 2 is showing its underlying information as a tool tip. This plug-in module aims to give more flexibility to users about where they generate their results from by offering to save and load the state space binaries.

## 8.2 State space exploration module

In this section we describe how we re-wrote the steady state module so that it could make use of our parallel state space explorer algorithm. We also investigate a parallel implementation of the steady state solver.

### 8.2.1 A sequential implementation

A sequential implementation of Gauss-Seidel has been implemented to solve the steady state in the PIPEAnalysis library. The API exposed to users can be seen in Figure 8.6. Here the `AbstractSteadyStateSolver` performs the transpose of matrix  $A$  and calculates the diagonals so that the concrete implementation of the steady state solver need only perform the algorithm of choice.

Cucumber integration tests were implemented for a range of Petri nets with pre-calculated results to ensure correctness of this algorithm some of which can be seen in Listing 8.3.

### 8.2.2 A parallel Jacobi implementation

We first experimented with the Jacobi algorithm for creating a parallel implementation of the steady state solver since refinements to  $x$  can easily be made in parallel. Although convergence rates for a sequential Jacobi algorithm are likely to be lower than a Gauss-Seidel algorithm we wanted to see if a parallel implementation could see any speed benefits.

It is a well known that the Jacobi algorithm is only guaranteed to converge if the matrix is diagonally dominant, however in many cases it will still converge if this is not true. Whilst solving the steady state we found a simple two state Petri net that does not converge using the Jacobi algorithm or Power method and causes a well known billion state solver to hang

```

Feature: steady state using Gauss-Seidel

Scenario: Parsing a simple Petri net file
  Given I use the Petri net located at /simple.xml
  When I calculate the steady state using a sequential solver
  Then I expect a record for
  """
  { 'P0' : { 'Default': 1 }, 'P1' : { 'Default': 0 }}
  """
  And its probability to be 0.5
  And I expect another record for
  """
  { 'P0' : { 'Default' : 0 }, 'P1' : { 'Default' : 1 }}
  """
  And its probability to be 0.5

```

Listing 8.3: An example Cucumber integration test written in Gherkin for the Gauss-Seidel steady state solver.

when using the Power method. The Petri net can be seen in Figure 8.5 and its one-step transition matrix and transpose are:

$$A = \begin{pmatrix} -2 & 2 \\ 1 & -1 \end{pmatrix},$$

$$A^T = \begin{pmatrix} -2 & 1 \\ 2 & -1 \end{pmatrix}.$$

Solving for an initial guess of  $x^0 = [1.0, 1.0]$  and applying the Jacobi algorithm from Equa-

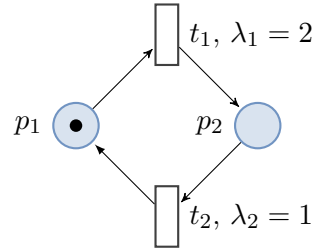


Figure 8.5: A simple two state Petri net whose steady state calculation does not converge when using the Jacobi algorithm or the Power method.

tion (2.6) we find that the solution for  $x$  loops forever between two different values:

$$x_1^1 = 0 - (1/ - 2) = 0.5$$

$$x_2^1 = 0 - (2/ - 1) = 2.0$$

$$x_1^2 = 0 - (2/ - 2) = 1.0$$

$$x_2^2 = 0 - (1/ - 1) = 1.0$$

$$x_1^3 = 0 - (1/ - 2) = 0.5$$

$$x_2^3 = 0 - (2/ - 1) = 2.0$$

$$x_1^4 = 0 - (2/ - 2) = 1.0$$

$$x_2^4 = 0 - (1/ - 1) = 1.0$$

and so on.

To potentially overcome this problem we can use a technique called uniformization to transform a CTMC into an aperiodic DTMC which produces a diagonally dominant stochastic matrix  $Q$  which is guaranteed to converge to the same steady state as  $A$  [6]. We define:

$$Q = A/a + I \tag{8.1}$$

where the rate  $a > \max_i |a_{ii}|$  and ensures that the DTMC produced is aperiodic because it ensures there is at least one single step transition from a state to itself [50]. We can then solve the steady state by re-writing the DTMC balance equation seen in Equation (2.3) as:

$$(I - Q^T)x^T = 0.$$

Unfortunately for the same example as in Figure 8.5 once the the DTMC matrix  $Q$  has been transposed it is still not diagonally dominant and when solving using the Jacobi algorithm it never converges. To overcome this the Power method could be used on  $Q$ , however since it is arguably much slower than the Jacobi and Gauss-Seidel algorithms, we choose to run a sequential Gauss-Seidel implementation on the matrix instead. This leads to a hybrid parallel solver which implements the following logic:

- Convert the CTMC to a DTMC using Equation (8.1).
- Transpose the DTMC matrix  $Q$ .
- Analyse  $(I - Q^T)$  for diagonal dominance.
- If  $(I - Q^T)$  is diagonally dominant then the matrix can be solved via a straight parallel implementation.
- If on the other hand,  $(I - Q^T)$  is not diagonally dominant then apply a hybrid implementation running the Gauss-Seidel sequential solver in a single thread, and the parallel Jacobi solver in the remaining number of threads. Take the first result.

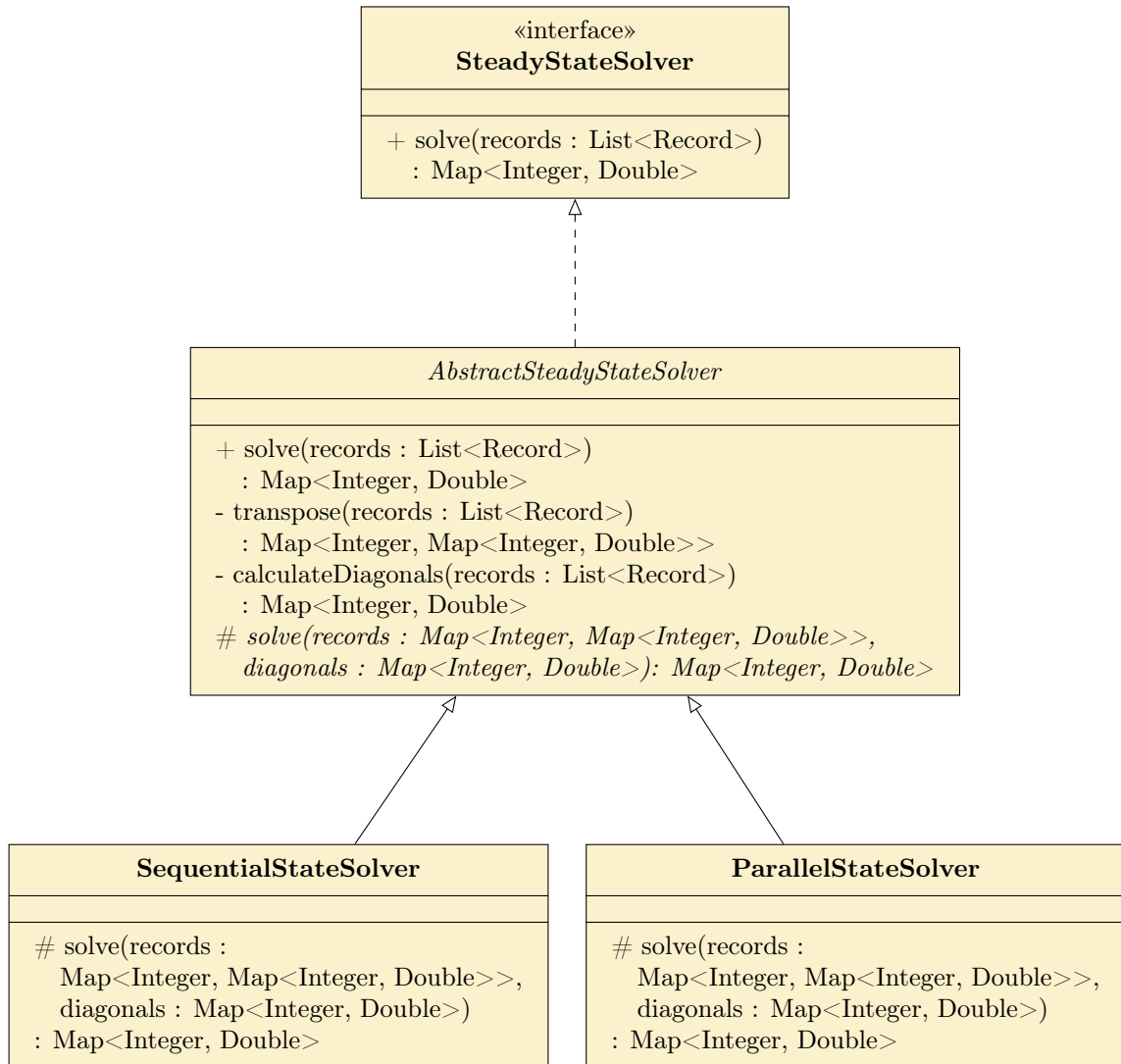


Figure 8.6: Steady state API exposed to users in PIPE 5.



### 8.2.3 A parallel asynchronous Gauss-Seidel implementation

Finally since we saw poor convergence results in practice for our parallel Jacobi algorithm we adapted the algorithm described in [51] to run on a single multi-core processor.

For each iteration of the main Gauss-Seidel algorithm we run separate worker threads that calculate successive improvements on their specific rows of  $x$ . In order to get maximum performance out of each thread we allow  $n$  sub-iterations to run before returning to check convergence. This value of  $n$  can be customised at run-time.

Graph partitioning was not needed to assign a smart distribution of rows of  $x$  to threads since memory is shared on a single processor. This simplified our algorithm significantly, allowing us to assign successive rows of  $x$  to each thread. Moreover we allow inter-thread communication of updated results during sub-iterations by using an `AtomicReferenceArray` to store the values of  $x$ . This is a concurrent thread-safe data structure that blocks at element level allowing a larger level of concurrency than other Java data structures which block at the `add` and `get` level.

When all threads have returned  $x$  is checked for convergence. If it is deemed to have converged we return the normalised value; if not we continue for further iterations.

## 8.3 Performance Analysis

Having solved the steady state at equilibrium for a Petri net we continue by calculating state based metrics for a performance measure  $m$  as described in Equation (2.7). Examples of metrics we calculate using this equation are the average number of tokens in each place and the average transition throughputs.

We ensured the correctness of our algorithms through a series of Cucumber integration

```
Feature: Calculating transition throughput metrics
Scenario: Parsing a simple Petri net file
  Given I use the Petri net located at /simple.xml
  When I calculate the metrics
  Then I expect the transition throughputs to be
  """
      { 'T0' : 0.5, 'T1' : 0.5 }
  """

Feature: Calculating average token metrics
Scenario: Parsing a simple vanishing Petri net file
  Given I use the Petri net located at /simple_vanishing.xml
  When I calculate the metrics
  Then I expect the places to have the following average number of
  tokens
  """
      { '1' : { 'Default' : 0.0 }, '2' : { 'Default' : 0.0 },
        '3' : { 'Default' : 0.0 }, '4' : { 'Default' : 0.0 },
        '5' : { 'Default' : 0.25 }, '6' : { 'Default' : 0.25 },
        '7' : { 'Default' : 0.25 }, '8' : { 'Default' : 0.25 }}
  """
```

Listing 8.4: An example of two Cucumber integration test features written in Gherkin for verifying the average number of tokens in a place and the transition throughputs at equilibrium. Note that the transition throughput test verifies the behaviour of the problem reported in Section 4.4.4.

tests, some of which can be seen in Listing 8.4, which compare the results of our algorithms to hand calculated results for each Petri net.

## Chapter 9

# Evaluation

Now is the era of intellect, information and the Petri net.

Lech Walesa (paraphrased)

To evaluate this project we must first perform a quantitative analysis of the codebase of PIPE 5 against its predecessor, PIPE 4. We then analyse the performance gains of the new Petri net analysis algorithms developed.

### 9.1 Quality metrics

The quality of the codebase can be judged on its cyclic dependencies, results of static analysis tools and improved level of testing. For the purposes of reproduction all repositories have been marked in Git with the tag `evaluation` and results can be reproduced as per the instructions in Section 4.3.1.

Repository	Tangle (%)
PIPEMarkovChain	0
PIPEAnalysis	0
PIPECore	7.10
PIPE	12.47

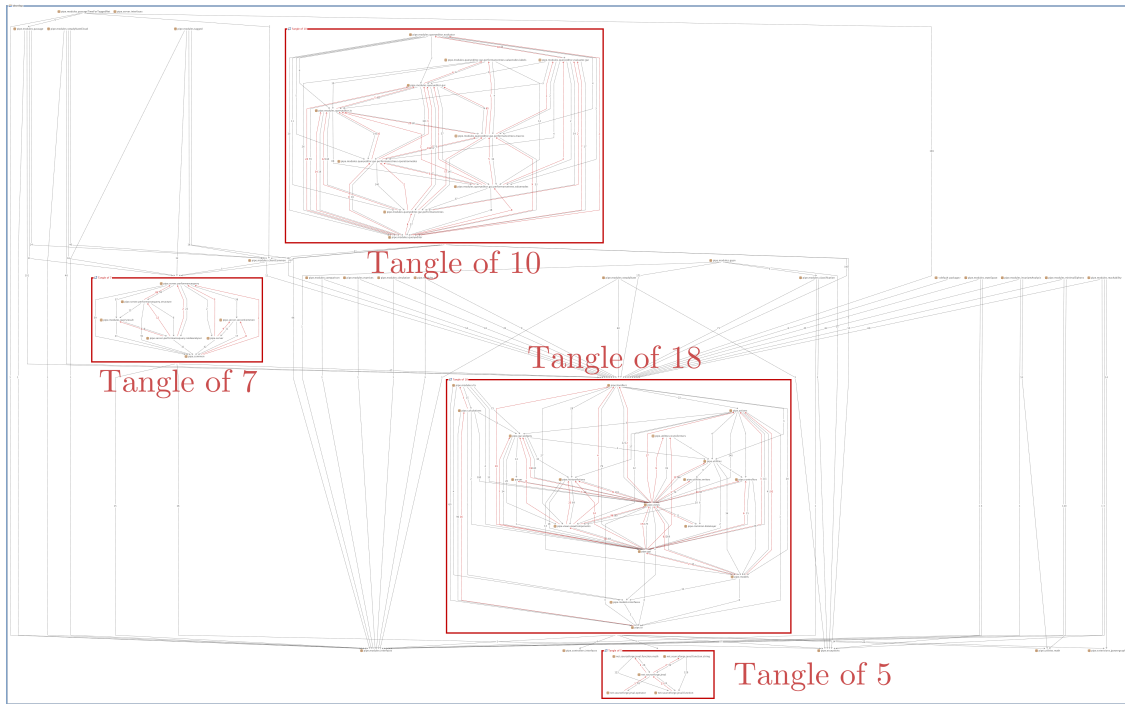
Table 9.1: Break down of overall tangle metric for the separate PIPE 5 repositories as reported by Stan4J. This metric is directly correlated to the number of reported tangles in Figure 9.1 and shows a good reduction from the 29.17% reported for PIPE 4.

### 9.1.1 Reduction in tangles

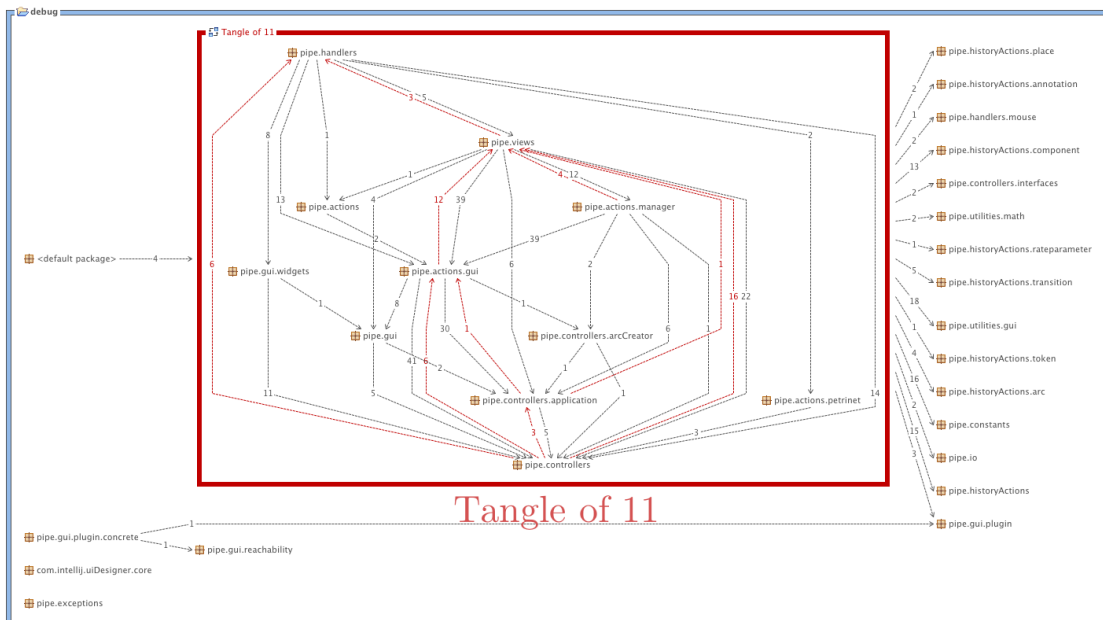
PIPE 5 has significant improvement gains in the structuring of its classes and packages. Stan4J now reports only two tangles of size 11 and 2 opposed to the four tangles in PIPE 4 of size 18, 10, 7 and 5. This reduction in tangles reduces the overall tangle metrics for the repositories as seen in Table 9.1 and the tangle graphs are visualised in Figure 9.1.

The splitting of the code into separate repositories helped significantly with reducing the number of tangles as it yielded smaller sub-projects that were easier to manage. It also helped to avoid an overlap of functionality between the separate libraries, for example having the Petri net logic in a different repository ensures that no GUI code can ever creep in.

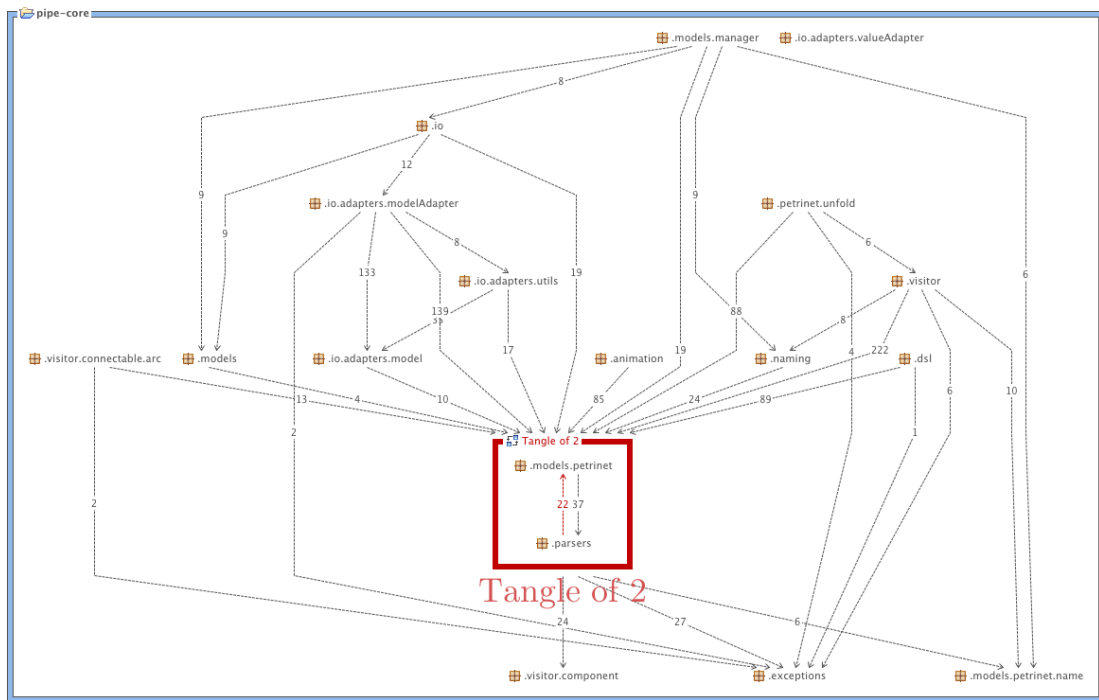
Unfortunately, as previously mentioned, we were not able to get rid of the tangles entirely. The PIPECore repository contains a tangle of size two which is due to the auto-generated ANTLR v4 code in the parsers package referencing Petri net components, and the Petri net components referencing the parsers package. We decided to keep this tangle rather than try to remove it because it keeps the Petri net component API simple. By keeping the parsing logic encapsulated in the Petri net components it allows developers to use the interfaces in their code rather than require direct knowledge of the concrete implementations. For example, when analysing a transition a developer wishes to be able to acquire its rate with a single call to `getRate()`. Since transitions have single server and infinite server semantics



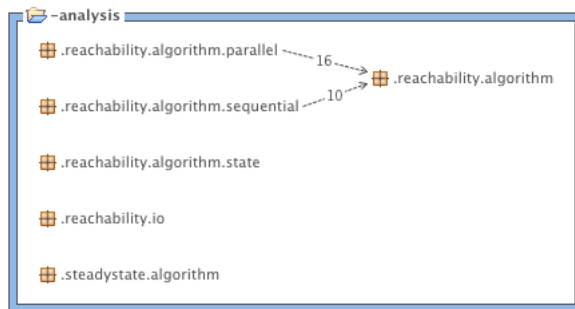
(a) The original PIPE 4 tangle graph reporting four tangles of size 18, 10, 7 and 5.



(b) The tangle graph for the PIPE repository housing the view code in PIPE 5 which contains the Swing GUI code. It contains only a single tangle of size 11. Unfortunately due to time constraints PIPE 5 uses much of the original PIPE 4 architecture which is why we were not able to reduce the tangle count further as many of the existing classes are tightly coupled.



(c) The new PIPE 5 tangle graph for the PIPECore repository reporting a single tangle of size 2. This tangle is between the functional expression parsers and the Petri net models. It has been kept because it improves the quality of the API by allowing Petri net components, namely transitions, to be able to evaluate their functional expressions without extra interaction from the user.



(d) The tangle graph for the new PIPEAnalysis repository containing the code for the steady state solver and the state space exploration. It reports 0 tangles.



(e) The tangle graph for the new PIPEMarkovChain repository containing the classes which represent the underlying Markov chain of a Petri net and the explored sets data structure. It reports 0 tangles.

Figure 9.1: A comparison of the tangles in the PIPE 5 architecture against those in the PIPE 4 architecture. The tangle graphs clearly show a reduction from four tangles of size 18, 10, 7 and 5 down to only two tangles of size 11 and 2.

Repository	Efficiency	Maintainability	Portability	Reliability	Usability	<b>Total</b>
PIPEMarkovChain	0	0	0	0	0	<b>0</b>
PIPEAnalysis	0	0	0	0	0	<b>0</b>
PIPECore	0	19	0	68	294	<b>381</b>
PIPE	13	77	0	426	40	<b>556</b>
<b>Total</b>	<b>13</b>	<b>96</b>	<b>0</b>	<b>494</b>	<b>334</b>	<b>937</b>

Table 9.2: Break down of issues highlighted by the QAPlug analysis plug-in for IntelliJ for each of the new PIPE 5 repositories. The total number of code quality issues for the entire project has been reduced from 12 904 to 937. Of these 937 issues 349 are due to auto-generated code via the ANTLR v4 plug-in.

we would like this logic to remain entirely in the transition itself which equates to an infinite server multiplying its rate by its enabling degree. Due to the nature of functional expressions this design decision requires parsing the transitions functional rate against a given Petri net state in the transition itself. Without this design, developers would need to assess the type of transition and perform the multiplication and parsing themselves.

Furthermore due to time constraints PIPE 5 re-uses the overall architecture of the PIPE 4 GUI which avoided starting again from scratch with the user interface. Sadly the design and usage of these classes together leads to significant tangles due to very tightly coupled classes. Much time was spent trying to reduce the tangle count, however it became clear that a complete re-design of how these classes interact with each other would be necessary to reduce the tangle count even further.

### 9.1.2 Static analysis

By analysing the separate repositories with the same settings as in Section 4.3.1 we can see that the previous count of 12 904 reported issues for PIPE 4 has been considerably reduced to a total of 937 issues. The break down of quality issues across repositories can be seen in Table 9.2.

On analysing these code quality issues we found that 349 of them are due to the ANTLR

v4 auto-generated code meaning that the number of code quality issues for our written code is actually 588.

We have reproduced the figure from Section 4.3.1 which documents the most important quality issues and their new counts in Figure 9.2. We are specifically interested in the reduction of ‘God classes’ from 66 in PIPE 4 to a total of 9 split across the PIPECore and PIPE repositories. This metric is very useful to describe code quality and shows a significant improvement in the architecture of the classes. In the future we hope to decrease this to 0.

Of the remaining issues a total of 473 come from the ‘*Magic Number Count*’ metric. These are entirely due to layout and sizing settings for view components and indicate that whilst a useful metric for non-view code it is perhaps not so relevant for projects containing GUI code.

It was difficult to reduce the number of issues in the PIPE repository further due to the tangled nature of the legacy code and lack of documentation. When trying to re-write certain features the functionality often broke and could not be fixed. A compromise had to be made to leave these areas of the legacy view code as is until further time can be spent on them.

### 9.1.3 An improved test suite

PIPE 5 now supports a more rigorous test suite comprised of 1042 unit and integration tests (the breakdown of these can be seen in Table 9.3) versus the 58 tests that existed in PIPE 4. Not only do the tests provide an added level confidence in the implementation but they also safe guard any future changes or additional features breaking intended functionality.

We found that by applying best practices such as Test Driven Development (TDD) to the development process of PIPE 5 it led to small, modular and easy to use classes as it



- **Efficiency: 13 (225)**
  - ‘Performance - Method concatenates strings using + in a loop count: 0 (44)’  
— In each iteration, the String is converted to a StringBuffer/StringBuilder, appended to, and converted back to a String. This can lead to a cost quadratic in the number of iterations, as the growing string is recopied in each iteration. Better performance can be obtained by using a StringBuffer (or StringBuilder in Java 1.5) explicitly.
  - ‘Avoid Array Loops count: 0 (3) - System.arraycopy is more efficient’ — Instead of copying data between two arrays, use System.arraycopy method.
- **Maintainability: 96 (1803)**
  - ‘Avoid duplicate literals count: 0 (160)’ — Can usually be improved by declaring the String as a constant field.
  - ‘Bad practice - Comparison of string objects using == or != count: 0 (7)’  
— This code compares java.lang.String objects for reference equality using the == or != operators. Unless both strings are either constants in a source file, or have been interned using the String.intern() method, the same string value may be represented by two different String objects. Consider using the equals(Object) method instead.
  - ‘God class count: 9 (66)’ — The God class rule detects the God Class design flaw using metrics. God classes do too many things, are very big and overly complex. They should be split apart to become more object-orientated.
- **Portability: 0 (47)**
  - ‘Replace Vector With List count: 0 (22)’ — Consider replacing Vector usages with the newer java.util.ArrayList if expensive thread-safe operation is not required.
  - ‘Integer Instantiation count: 0 (19)’ — Calling new Integer() causes memory allocation. Integer.valueOf() is more memory friendly.
- **Reliability: 494 (1847)**
  - ‘Malicious code vulnerability - May expose internal representation by returning reference to mutable object count: 0 (10)’ — Returning a reference to a mutable object value stored in one of the object’s fields exposes the internal representation of the object. — If instances are accessed by untrusted code, and unchecked changes to the mutable object would compromise security or other important properties, you will need to do something different. Returning a new copy of the object is better approach in many situations.
  - ‘Magic Number Count: (473) 1385’ — Checks for magic numbers.
- **Usability: 334 (8982)**
  - ‘Dodgy - write to static field from instance method count: 2 (49)’ — This instance method writes to a static field. This is tricky to get correct if multiple instances are being manipulated, and generally bad practice.
  - ‘System Println count: 0 (412)’ — System.(out|err).print is used, consider using a logger.
  - ‘Code Too Long count: 0 (15)’ — Method is too long.

Figure 9.2: A re-evaluation of the most important issues flagged about the PIPE 4 codebase against the number of these issues in PIPE 5. The previous PIPE 4 values are shown in brackets.

Repository	Test count
PIPEMarkovChain	35
PIPEAnalysis	500
PIPECore	302
PIPE	205

Table 9.3: Break down of the number of unit and integration tests split across PIPE 5's repositories. The total count is 1042 which is significantly higher than the 58 that existed in PIPE 4.

encourages good class design through continual testing.

Unfortunately due to time constraints it was not possible to introduce unit tests for the Swing GUI using frameworks such as FEST due to the complex level of coupling between the existing view classes. FEST requires the individual view components be easily runnable as standalone items, something which is just not possible in the current architecture. In order to be able to write a full suite of Swing tests the entire view codebase needs to be re-written. However we did manage to introduce unit tests for the new controllers in the PIPE 5 repository but it still leaves the Swing GUI open to unnoticed bugs which is a significant limitation of the existing test suite.

#### 9.1.4 Documentation

PIPE 5 contains a considerable amount more Javadoc than PIPE 4, with the percentage breakdowns for classes, methods and fields being shown in Table 9.4. This increased level of Javadoc details what methods do, what fields and classes are used for and where appropriate explains design decisions taken and example usage. This Javadoc will make development of the open-source project significantly easier for future contributors.

It was not possible to achieve 100% coverage over the brand new PIPE 5 repositories (PIPEMarkovChain, PIPECore and PIPEAnalysis) because they contain auto-generated code in the form of the hashCode and equals methods and the ANTLR v4 classes in

Repository	Classes (%)	Fields (%)	Methods (%)
PIPE 4	50.00	3.85	12.50
PIPEMarkovChain	100.00	90.48	89.04
PIPEAnalysis	96.43	88.06	88.49
PIPECore	93.13	94.87	89.27
PIPE	89.71	79.05	78.73

Table 9.4: Percentage of Javadoc for classes fields and methods for the new different PIPE repositories and the predecessor (PIPE 4). Overall there is a far higher level of Javadoc for the three brand new repositories (PIPEMarkovChain, PIPECore and PIPEAnalysis) and the Javadoc for the new and reused code in the PIPE repository has also seen an increase.

PIPECore.

Furthermore, beyond its testing capabilities, the introduction of an extensive test suite in the PIPE 5 repositories acts as a form documentation on how classes are composed and their expected usage.

## 9.2 Analysing the new modules

In this section we analyse the speedups gained by our sequential algorithm over its predecessor and perform an analysis of our new parallel algorithms. Unless specified all tests were run using a 2013 MacBook Air with a 2GHz dual-core hyper-threaded 3rd generation i7 processor and 8GB of RAM.

### 9.2.1 Analysing the state space explorer module

We begin our analysis by comparing our new sequential algorithm for the state space exploration with its predecessor in PIPE 4. In Section 4.2.3.2 we saw that PIPE 4 was slow to generate the reachability graph for Petri nets larger than 1000 states. Our new sequential implementation yields increasing speedups compared to PIPE 4 for small state

Number of states	Number of transitions	PIPE 4 (s)	PIPE 5 (s)	Speedup
40	156	0.21	0.17	1.24
100	480	0.36	0.40	0.90
625	4000	25.12	1.35	18.61
1350	9450	83.67	1.75	47.81
4096	28672	728.02	3.82	190.58
11664	93312	2738.37	8.51	321.78

Table 9.5: The time taken in seconds to generate the reachability graph in PIPE 4 compared with the new sequential algorithm in PIPE 5. For very small state spaces we can see that the times are comparable, but for moderate sized Petri nets PIPE 5 is more scalable.

spaces as can be seen in Table 9.5. Interestingly the algorithm for the sequential exploration does not differ from that of PIPE 4, the speedup comes from advanced data structure usage, memoization and other optimisation techniques.

Next we compare our sequential algorithm to our MapReduce-style parallel algorithm running on 4 virtual cores. Since the underlying algorithm performed is very similar we expect to see a linear speedup and indeed the results in Figure 9.3 show approximately this. We allowed each worker thread to process 100, 200, 500 states before returning to have their results reduced and although there is little difference between them, the 100 states per thread run provides the best speedup against the sequential implementation. Using these results we went on to investigate the benefits for larger state spaces and observed that we see similar speedups in Figure 9.4.

Overall these results show a considerate speedup and allow for Petri nets with large state spaces to be analysed realistically on local machines.

## 9.2.2 Scalability

Following good results on a MacBook Air we investigated the scalability of our state space explorer algorithm using a machine with a 3.40GHz quad-core hyper-threaded 3rd

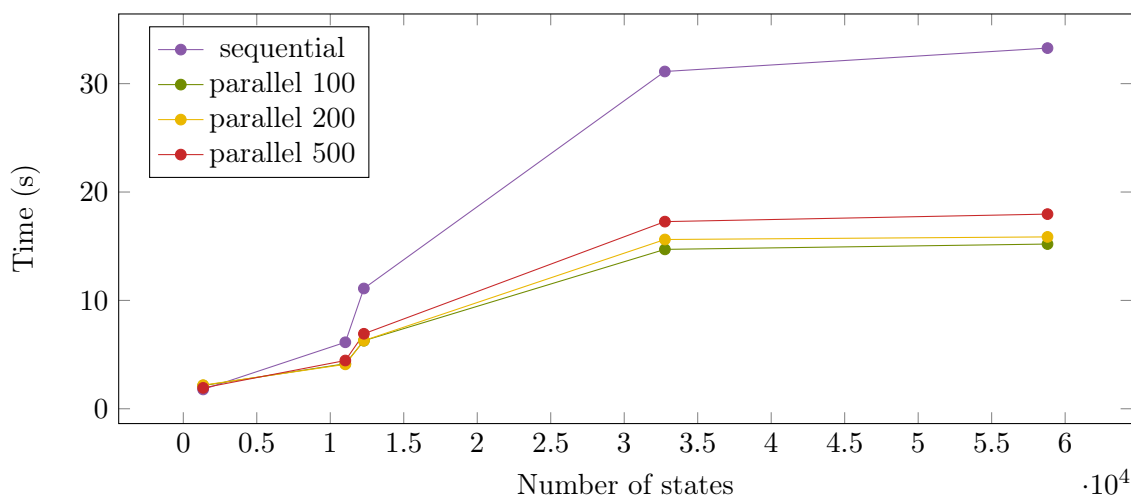


Figure 9.3: A comparison between the run-times of PIPE 5’s sequential state space exploration algorithm and the new MapReduce-style parallel algorithm running on 4 cores. We have allowed the parallel implementation to explore 500, 200, and 100 states before the reduction phase of each iteration. All parallel implementations yield at least a 2x speedup over the sequential algorithm although the 100 states per thread run performs slightly better than the others.

generation i7 processor and 16GB of RAM. This change of architecture allows us to analyse our steady state exploration algorithm using 2, 4, and 8 virtual cores.

In order to deduce useful results we analysed the speedups gained by increasing the number of virtual cores. Figure 9.5 shows that whilst the speedup from 2 to 4 virtual cores is promising at around 60%, we do not get such good results from running the algorithm with 8 virtual cores. On profiling we found that the bottleneck in our algorithm is the creation of a states primary and secondary hash codes, taking 68% of the algorithms run-time. Unfortunately because states are generated in the call to `getSuccessors(state)`, for a successor that has numerous parents, say  $n$  of them, our current algorithm will generate this state  $n$  times. Moreover since there is no implementation of a concurrent queue in Java that does not allow duplicates, we discovered that quite often it is possible to add a state twice to the queue, meaning that its successors get generated twice too. Even though

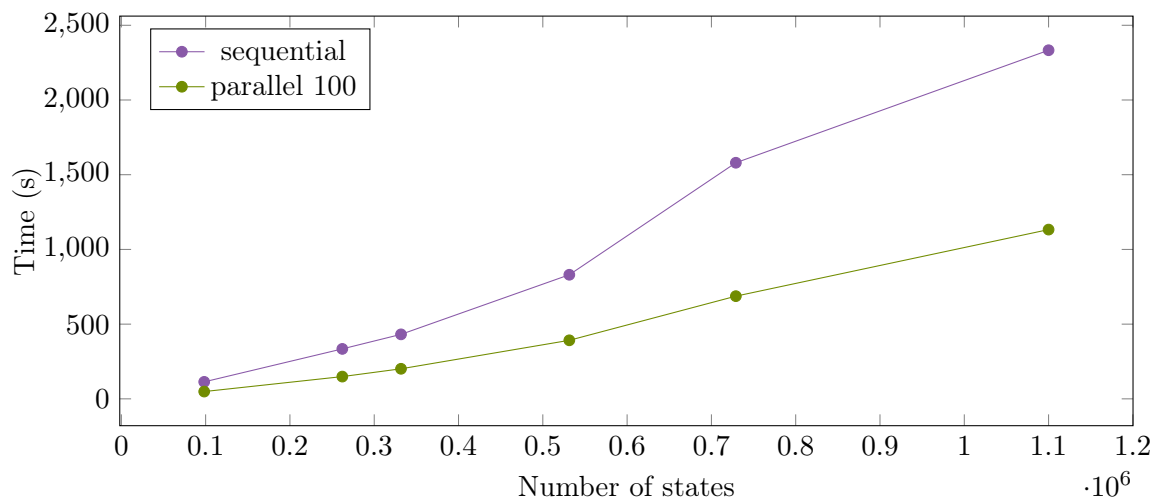
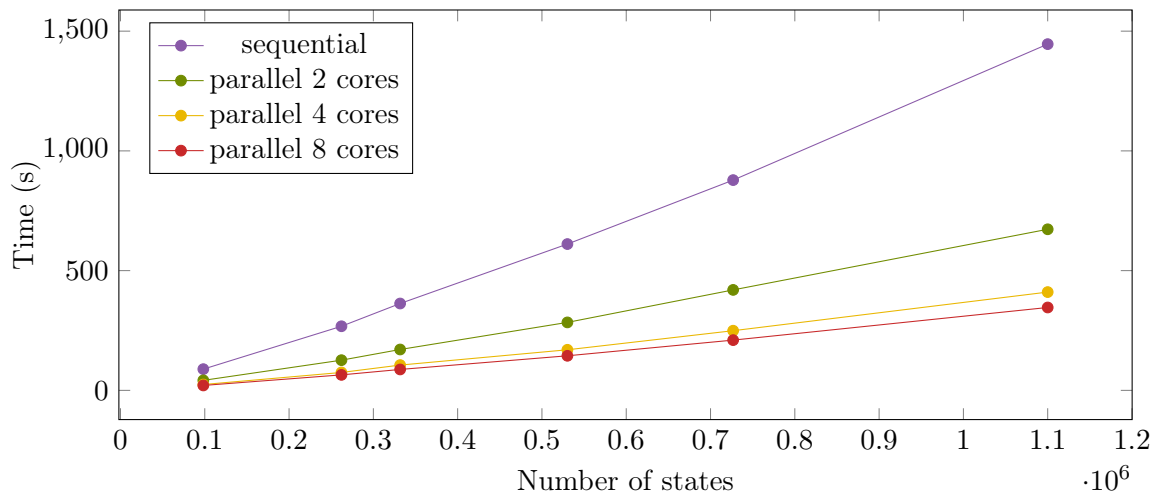


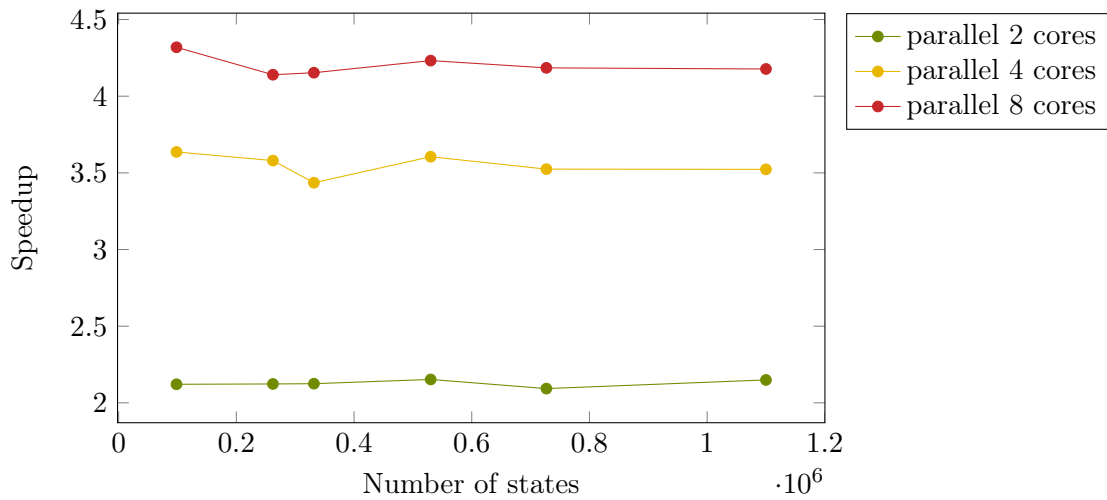
Figure 9.4: A comparison between the run-times of PIPE 5’s sequential state space exploration algorithm and the new MapReduce-style parallel algorithm running on 4 cores for large Petri nets. This shows that the parallel implementation is consistently around twice as fast.

each thread first checks to see if a successor has been seen before adding it to the queue, if two separate worker threads do this check before marking the successor as seen then it is possible to add the successor to the queue twice. On inspection of the number of times `getSuccessors` was called, we saw that when using 8 virtual cores this method’s call count was considerably higher.

Nevertheless in order to put into perspective the speedup gained by our new parallel algorithm using 100 states per thread and 8 virtual-cores we compared the run-times of a 4096 state Petri net with this algorithm and PIPE 4. Whilst PIPE 4 explores the state space in 9.37 minutes, our new algorithm can explore it in 2.65 seconds which amounts to an incredible 211x speedup. Moreover our new algorithm can solve a Petri net with 1 099 999 states in 5.77 minutes which is faster than the time taken to solve the 4096 state Petri net in PIPE 4!



(a) The raw times of the steady state exploration graph. The results gathered are for a sequential algorithm and for a parallel implementation with 100 states per thread running on 2, 4, and 8 virtual cores.



(b) The relevant speedup gained over the sequential algorithm running our parallel algorithm with 100 states per thread and an increasing number of virtual cores. The graph shows that for each increase in the number of virtual cores we see a consistent speedup.

Figure 9.5: A comparison between the run-times and relevant speedup of PIPE 5's sequential state space exploration and its parallel algorithm with 100 states per thread running on 2, 4 and 8 virtual cores on a 3.2GHz quad-core hyper-threaded 3rd generation i7 processor.

### 9.2.3 Steady state solver results

Finally we investigated the speedup that our new set of steady state solvers yield.

Since the state space is required in order to solve the steady state we first observe that capabilities of the steady state solver directly depend upon the results of the state space exploration module. The benefits we gain from our state space explorer algorithm mean we can solve the steady state for much larger Petri nets than the previous PIPE 4 module.

#### 9.2.3.1 Sequential Gauss-Seidel

In order to gather comparable results we compare PIPE 5's Gauss-Seidel implementation against its predecessor in PIPE 4. Table 9.6 below shows that whilst PIPE 5 appears to perform slightly worse in some cases than PIPE 4, the differences are negligible because all times are less than one second. The discrepancies in time between the two algorithms come from PIPE 5's transformation of a CTMC to a DTMC meaning the algorithms solve different matrices. It was however, very difficult to get meaningful results for a comparison against PIPE 4 due to its dependency on its slow state space exploration algorithm making larger matrices very difficult to acquire.

#### 9.2.3.2 Results of the Jacobi implementation

Whilst a parallel Jacobi implementation looked promising against its sequential counterpart, in practice the poor convergence properties of this algorithm render it unable to solve Petri net steady states. For almost all Petri nets we tried the algorithm timed out at a million iterations.



Number of states	Number of transitions	PIPE 4 time (ms)	PIPE 5 time (ms)
40	156	17.44	50.47
100	480	83.91	995.87
225	1200	50.65	121.09
675	4500	66.09	5.21
1000	7200	0.14	10.51
1350	9450	65.12	439.07

Table 9.6: The time in milliseconds taken to solve the steady state with the sequential Gauss-Seidel algorithms in PIPE 4 and PIPE 5. The results shown are for various sized Petri nets with random transition rates between 0-20. We found it difficult to get results for any reasonably sized state spaces because PIPE 4 cannot decouple solving the steady state from exploring the state space. Although PIPE 5 appears to perform slightly worse on average for these small states than PIPE 4, the two algorithms solve for different matrices. PIPE 5 solves a DTMC whilst PIPE 4 solves the raw CTMC and since these matrices are different they will converge at different rates, sometimes being better for the CTMC and sometimes as in the case of the 675 state space being better for the DTMC. Nevertheless PIPE 5's sequential solver still runs in under 1 second so the user will not notice the slight slowdown.

### 9.2.3.3 Results of the asynchronous Gauss-Seidel solver

We ran our asynchronous Gauss-Seidel steady state solver against its sequential counterpart for various sub-iteration values. Figure 9.6 shows an average speedup of up to 3.47x that of the sequential algorithm and a maximum speedup of 5.13x the sequential algorithm for moderate Petri nets.

Overall it appears that whilst the size of the state space is related to convergence times Figure 9.7 shows us that the rates of the timed transitions in the Petri net heavily influence the rate at which the matrix converges and thus the time taken to solve it. [52] also explains that the ordering of the rows of the matrix influence its rate of convergence and that a breadth first search of the state space yields optimal results. It is for these reasons that we do not see linear speed improvements in Figure 9.6.

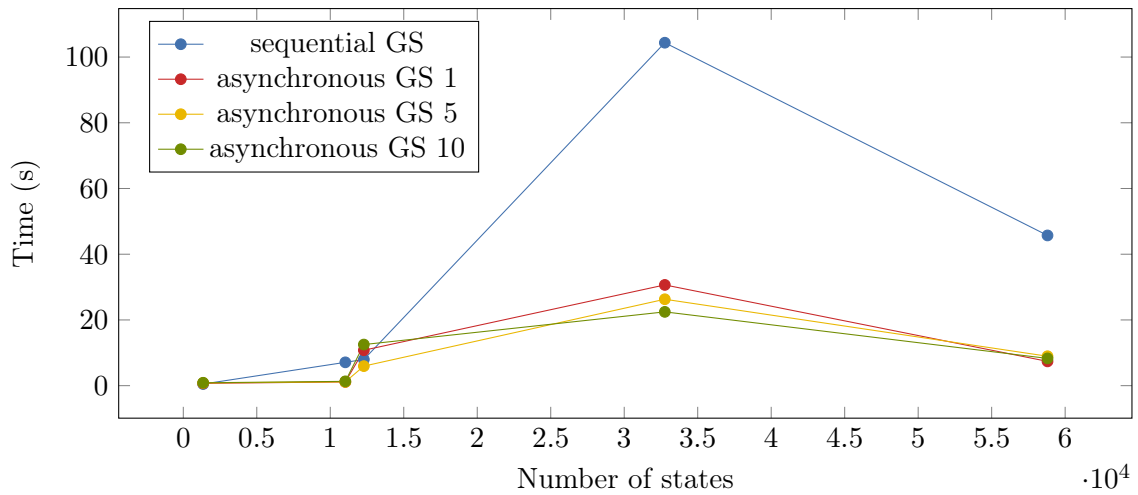


Figure 9.6: A comparison of the time taken in seconds to solve the steady state using our sequential Gauss-Seidel solver and our asynchronous parallel Gauss-Seidel solver running on 4 virtual cores. Our parallel algorithm was set-up to use 1, 5, and 10 sub-iterations per thread. For most cases we see that our asynchronous algorithm with 10 sub-iterations outperforms the sequential algorithm with an average speedup of 3.35x for the results shown.

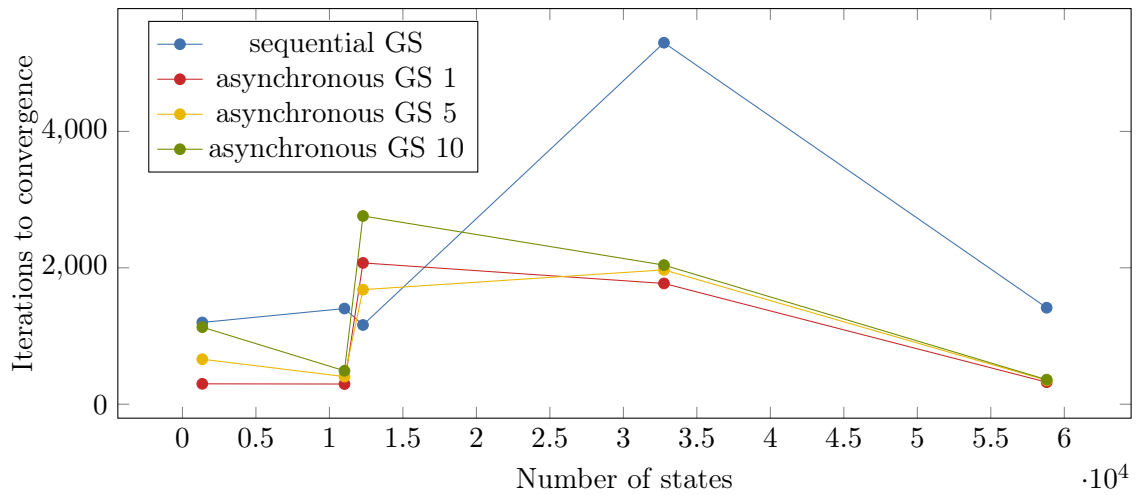


Figure 9.7: A comparison of the number of iterations taken to solve the steady state when using the sequential Gauss-Seidel algorithm and the asynchronous Gauss-Seidel algorithm running on 4 virtual cores with different sub-iteration settings. In order to provide comparable results the results of the asynchronous Gauss-Seidel algorithm were multiplied by their respective sub-iteration count. These results show that the size of the matrix has less to do with the convergence rate than the contents.

### 9.3 Testimonial

We received the following testimonial from Steve Doubleday a researcher at University of California Irvine, who works closely with ICSI [26].

*“PIPE has been re-engineered top to bottom, as part of its most significant upgrade in years. Key improvements are the separation of business logic from the GUI, the addition of a large suite of unit tests, and the re-packaging of the software as Maven artifacts. All of this makes my work significantly easier. The work at ICSI (International Computer Science Institute), and my own dissertation both rely on extensions to the basic PetriNet capabilities, to better enable the creation of neurally-plausible cognitive models of action. The existing ICSI software consists of many modifications to a very old version of PIPE. My intent is to re-build the existing ICSI software on top of the new PIPE artifacts. This will make both current development and future maintenance much cleaner, and will also make it more attractive to other cognitive scientists to re-use the ICSI extensions.”*

### 9.4 Strengths and limitations

The main strengths of the work undertaken in this project include:

- An in-depth analysis of the PIPE 4 architecture — this highlighted to us PIPE 4’s shortcomings and the specific areas in which we would focus our work.
- The design and release of new back-end models of a Petri net and its underlying Markov chain — these made the implementation of our analysis algorithms much easier. In fact without the abstraction of the Markov chain states our parallel algorithms could not have been developed. This work creates a reliable and stable platform off which other researchers can base their work.
- Integration of the Petri net models into the PIPE front-end — this has untangled

much of the view code and makes the classes themselves much simpler.

- A parallel implementation of the state space explorer — this implementation sees speedups of up to 4.31x that of the sequential algorithm when running on 8 virtual cores and can calculate the reachability graph for a million state Petri net in under six minutes.
- Significant improvement in code quality — through new and re-engineered classes we have reduced the number of code quality issues from 12904 to 937, reduced the cyclic complexity from 40 cyclic relationships to 13 and improved the reliability of the codebase through an extensive test suite containing 1042 unit and integration tests.

Although the project has been very successful it is not without its limitations. These are detailed below:

- Scalability of the state space explorer — unfortunately, as documented in Section 9.2.2, our state space explorer did not scale well to 8 virtual cores. In order to improve the performance of this algorithm a concurrent queue which does not allow duplicate entries needs to be investigated and implemented so that no state is processed more than once in the algorithm.
- Convergence times of the parallel steady state solver — whilst our asynchronous Gauss-Seidel implementation was sometimes much faster than a sequential implementation it did not always perform with maximum results. Sometimes the overhead of a thread based system slowed down the overall time spent to solve the system.
- The legacy view code still exists — due to time constraints much of the legacy view code has to be reused. This means that the PIPE repository still contains some tangles and is not as understandable as we hoped.

# Chapter 10

## Conclusion

We're still in the first minutes of the first day of the Petri net revolution.

Scott Cook (paraphrased)

In this chapter we reflect on our achievements, summarise the lessons we have learned and conclude with some examples of future work.

### 10.1 Achievements

During the duration of this project we have performed a complete re-architect of the structure of the Platform Independent Petri net Editor. To overcome the tangled and buggy codebase we decoupled the logic of a Petri net from its graphical representation via the creation of Petri net and Markov chain back-end models. To ease the programmatic creation of Petri nets we provide a DSL API to our users. Not only does this reduce the average number lines taken to create a Petri net by a third, but it has the added benefit of providing a sentence like structure to their creation. Indeed we make use of this DSL throughout our test suite to increase the readability of our tests so that they may act as

another form of documentation.

On top of these models we developed two new Petri net analysis modules for generating the reachability graph and solving and analysing the steady state of a Petri net. We have brought the analysis of large Petri nets back to local machines by exploiting multi-core processors. Using 8 virtual cores our state space explorer algorithm yields speedups of over 200x that of PIPE 4 and can successfully generate of the reachability graph of a Petri net with over a million states in under 6 minutes on a 3.2GHz quad-core hyper-threaded i7 processor.

In order to encourage future development and research in the area of Petri nets we have revamped the development process of PIPE. It now boasts Git version control, GitHub integration, a Maven build system, a test suite comprised of 1042 unit and integration tests and a Travis CI continuous integration server.

Our work in this project has developed a robust, reliable and powerful platform upon which future generations of researchers can use and extend. Indeed scientists at the International Computer Science Institute in Berkeley California are already in the process of customising our work to build and solve neural cognitive models.

## 10.2 Lessons learned

This project has proved challenging on many levels. Notably the use of profiling, good testing and iterative releases proved instrumental in the development process.

Gaining speed improvements for the state space exploration module proved to be a tedious and time consuming process. Changes needed to be made incrementally and analysed in depth. We learned the importance of using a profiler to gain better insight into where bottlenecks occur. Many of the hotspots found with the YourKit profiler were in unexpected

areas, highlighting the necessity of a good profiler for building efficient algorithms.

Whilst a good test suite is commonplace in production level projects, continuous integration servers are still somewhat a novelty. Our Travis CI deployment proved to be an essential safety net, catching unforeseen errors in the installation process and bugs when we occasionally forgot to run the test suite before pushing our changes to GitHub.

Finally we came to understand the importance of releasing a project iteratively to acquire feedback from our users which could not be gained from testing. For example the request for an uber-jar which when double-clicked launches PIPE 5 to avoid the build process of the project.

## 10.3 Future work

To improve further on the work performed during this project the following are suggested as future areas of improvement.

### 10.3.1 Improving the convergence time of the steady state solver

In 1991 Ananda D. Gunawardena proposed a modified version of the Gauss-Seidel algorithm that yields significant convergence speedups by pre-conditioning the matrix  $A$  to solve  $PAx = Pb$  instead of  $Ax = b$ , where  $P$  is a pre-conditioning matrix [53]. An extension of this is an adaptive iterative Gauss-Seidel method that improves the rate of convergence further by modifying the matrix  $P$  [54]. Studying and applying these iterative techniques to increase the convergence rate of the steady state solver would be great benefit to future releases of PIPE 5.



### 10.3.2 Reversed Compound Agent Theorem

Programmatically solving the balance equations to determine a Petri nets steady state at equilibrium is computationally expensive. The Reversed Compound Agent Theorem (RCAT) derives product-form solutions for stochastic models defined as a composition of two or more smaller stochastic models [55]. It provides a way to solve the steady state of a system that satisfies a set of RCAT properties by exploring local state spaces of components rather than the global state space. In her Masters project Rhea Potdar created the first working implementation of RCAT applied to SPNs [56]. This algorithm could be added to the steady state analysis module in PIPE 5.

### 10.3.3 Cloud based modules

In order to speed up the processing of large Petri nets, or to allow for larger state spaces to be explored and analysed cloud-based modules could be developed. Amazon Elastic Compute Cloud (Amazon EC2) provides resizeable compute capacity in the cloud that can increase or decrease capacity in minutes to respond to the demand curve of an application [57].

### 10.3.4 An improved graphical interface

In order to allow for further UI features such as subnet capabilities to be incorporated into PIPE 5 and to overcome the remaining code quality issues the UI architecture needs to be re-designed. It is worth researching JavaFX8 as a new graphical framework since it has replaced Swing as the new client UI [58] and has been designed to be easier to use.

# Bibliography

- [1] Igor Kottenko and Mihail Stepashkin. Analyzing Vulnerabilities and Measuring Security Level at Design and Exploitation Stages of Computer Network Life Cycle. *Lecture Notes in Computer Science*, pages 311–324, 2005.
- [2] V. Varadharajan. Petri net based modelling of information flow security requirements. *[1990] Proceedings. The Computer Security Foundations Workshop III*.
- [3] Hamed Ghasemieh, A.K.I. Remke, and B.R.H.M. Haverkort. Analysis of a sewage treatment facility using hybrid Petri nets. 2013.
- [4] An analysis of existing Petri net tools. <http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html>.
- [5] SourceForge reviews for PIPE. <http://sourceforge.net/projects/pipe2/reviews?source=navbar>.
- [6] G. Bolch. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. Wiley-Interscience Publication. Wiley, 1998.
- [7] Wilfried Brauer and Wolfgang Reisig. Carl Adam Petri and “Petri nets”. *Fundamental Concepts in Computer Science*, 3:129–139, 2009.

- [8] Falko Bause and Pieter S. Kritzinger. *Stochastic Petri Nets*. Vieweg+Teubner Verlag, 2002.
- [9] D Kartson, Gianfranco Balbo, S Donatelli, G Franceschinis, and Giuseppe Conte. *Modelling with generalized stochastic Petri nets*. John Wiley & Sons, Inc., 1994.
- [10] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.
- [11] Erwin Kreyszig. *Advanced engineering mathematics*. John Wiley & Sons, 2010.
- [12] V Sundarapandian. *Numerical Linear Algebra*. PHI Learning Pvt. Ltd., 2008.
- [13] William John Knottenbelt. *Parallel performance analysis of large Markov models*. PhD thesis, Imperial College London (University of London), 2000.
- [14] Michael Weber and Ekkart Kindler. The Petri Net Markup Language. *Lecture Notes in Computer Science*, pages 124–144, 2003.
- [15] PIPE Downloads. <http://sourceforge.net/projects/pipe2/files/stats/timeline?dates=2004-05-05+to+2014-01-14>.
- [16] Yufei Wang. Extending PIPE2 to Support Functional Arc Weights and Transition rates. Master’s thesis, Imperial College London, 2013.
- [17] Faten Nabli, François Fages, Thierry Martinez, and Sylvain Soliman. A Boolean Model for Enumerating Minimal Siphons and Traps in Petri Nets. *Lecture Notes in Computer Science*, pages 798–814, 2012.
- [18] W.J. Knottenbelt, N.J. Dingle, and T. Suto. Performance Trees: A Query Specification Formalism for Quantitative Performance Analysis. *Computational Science, Engineering & Technology Series*, pages 165–198, Apr 2009.

- [19] Harini Kulatunga, Ashok Argent-Katwala, and William Knottenbelt. Cluster Grid based Response-time analysis module for the PIPE Tool. *Fourth International Conference on the Quantitative Evaluation of Systems (QEST 2007)*.
- [20] Nicholas J. Dingle and William J. Knottenbelt. Automated Customer-Centric Performance Analysis of Generalised Stochastic Petri Nets Using Tagged Tokens. *Electronic Notes in Theoretical Computer Science*, 232:75–88, Mar 2009.
- [21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.
- [22] Rekha Kumari. Problems in java.util.Observable. <http://javapracs.blogspot.co.uk/2011/02/problems-in-javautiobservable.html>, 2011.
- [23] James Bloom, Clare Clark, Camilla Clifford, Alex Duncan, Haroun Kahn, and Manos Papantoniou. PIPE Final Report, 2003.
- [24] PIPE2 v0.1 Source Code. <http://sourceforge.net/projects/pipe2/files/Pipe%20Pipe%20v0.1/>, 2004.
- [25] Jan Vlasak. Unifying the PIPE Petri net Editor Project, 2011.
- [26] Steve Doubleday. E-mail communication, 2013-2014.
- [27] William J Knottenbelt. Generalised Markovian analysis of timed transition systems. Master’s thesis, University of Cape Town, 1996.
- [28] Aaronnaught. What’s wrong with circular references? <http://programmers.stackexchange.com/questions/11856/whats-wrong-with-circular-references>, 2010.
- [29] Odysseus Software. STAN Structure Analysis For Java. 2009.
- [30] Martin Blore. Can a function be too short? <http://programmers.stackexchange.com/questions/64449/can-a-function-be-too-short>, 2011.

- [31] Miško Hevery. Root Cause of Singletons. <http://googletesting.blogspot.co.uk/2008/08/root-cause-of-singletons.html>, 2008.
- [32] Miško Hevery. Singletons are Pathological Liars. <http://misko.hevery.com/2008/08/17/singletons-are-pathological-liars/>, 2008.
- [33] MuzShekh. Github PIPE issue 23 - how to use the rate expression editor. <https://github.com/sarahtattersall/PIPE/issues/23>, 2014.
- [34] Matthew Rankin. Who Still Uses CVS? <http://stackoverflow.com/questions/3584820/who-still-uses-cvs-and-why>, 2010.
- [35] Matt Mackall. Towards a better SCM: Revlog and Mercurial. In *Linux Symposium*, page 83, 2006.
- [36] Jakub Narebski. Difference between Git and CVS. <http://stackoverflow.com/questions/802573/difference-between-git-and-cvs>, 2004.
- [37] Elisabeth Freeman. *Head first design patterns*. O'Reilly Media, Inc., 2004.
- [38] Robert C Martin. Acyclic visitor. In *Pattern languages of program design 3*, pages 93–103. Addison-Wesley Longman Publishing Co., Inc., 1997.
- [39] Robert Cecil Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.
- [40] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.
- [41] JAXP Package Overview. <http://docs.oracle.com/javase/tutorial/jaxp/intro/package.html>.
- [42] Java SAX tutorial. <http://docs.oracle.com/javaee/1.4/tutorial/doc/JAXPSAX.html>.
- [43] JAXP Tutorial. <http://docs.oracle.com/javase/tutorial/jaxp>.

- [44] Kurt Kluever. Hashing Explained. <https://code.google.com/p/guava-libraries/wiki/HashingExplained>, 2013.
- [45] Kurt Alfred Kluever. Benchmarking Hash Functions in Guava. <https://plus.google.com/+googleguava/posts/1eDwAr1YRE2>, 2013.
- [46] Adrien Grand. xxhash-benchmark. <http://jpountz.github.io/lz4-java/1.2.0/xxhash-benchmark/>, 2014.
- [47] Issue 889: Request: CityHash as a hasher option. <https://code.google.com/p/guava-libraries/issues/detail?id=889>, 2012.
- [48] Steve Jessop. Why should hash functions use a prime number modulus? <http://stackoverflow.com/questions/1145217/why-should-hash-functions-use-a-prime-number-modulus>, 2013.
- [49] Dr. Cliff Click. Non-blocking hashtable. <http://www.azulsystems.com/blog/cliff/2007-03-26-non-blocking-hashtable>, 2007.
- [50] Nicholas J Dingle, Peter G Harrison, and William J Knottenbelt. Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large Markov models. *Journal of Parallel and Distributed Computing*, 64(8):908–920, Aug 2004.
- [51] Nicholas J Dingle and William J Knottenbelt. Distributed solution of large markov models using asynchronous iterations and graph partitioning. In *Proceedings of the 18th UK Performance Engineering Workshop (UKPEW'02)*, pages 27–34, 2002.
- [52] D.D. Deavours and W.H. Sanders. “on-the-fly” solution techniques for stochastic Petri nets and extensions. In *Proceedings of the Seventh International Workshop on Petri Nets and Performance Models*. IEEE Comput. Soc.

- [53] Ananda D. Gunawardena, S.K. Jain, and Larry Snyder. Modified iterative methods for consistent linear systems. *Linear Algebra and its Applications*, 154-156:123–143, Aug 1991.
- [54] Masataka Usui, Hiroshi Niki, and Toshiyuki Kohno. Adaptive Gauss-Seidel method for linear systems. *International Journal of Computer Mathematics*, 51(1-2):119–125, Jan 1994.
- [55] Peter G. Harrison. Turning back time in Markovian process algebra. *Theoretical Computer Science*, 290(3):1947–1986, Jan 2003.
- [56] Rhea Potdar. Automatic construction of product-form solutions in stochastic networks. Master’s thesis, Imperial College London, June 2013.
- [57] Amazon. Amazon EC2. [http://aws.amazon.com/ec2/?nc1=h\\_12\\_cn](http://aws.amazon.com/ec2/?nc1=h_12_cn).
- [58] Oracle. Is JavaFX replacing Swing as the new client UI library for Java SE? <http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6>, 2014.

# Appendices



## Appendix A

# An example PNML listing

The PNML XML file for the Petri net in Figure A.1 can be seen below.

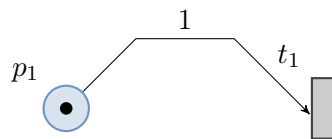


Figure A.1: A place-transition net whose arc has two intermediate points.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<pnml>
  <net>
    <token blue="0" green="0" red="0" enabled="true" id="Default"/>
    <place id="P0">
      <graphics>
        <position y="225.0" x="300.0"/>
      </graphics>
      <name>
        <value>P0</value>
      <graphics>
        <offset y="35.0" x="-5.0"/>
      </graphics>
    </place>
  </net>
</pnml>
```

```

    </name>
    <capacity>
      <value>0</value>
    </capacity>
    <initialMarking>
      <graphics>
        <offset y="0.0" x="0.0"/>
      </graphics>
      <value>Default,1</value>
    </initialMarking>
  </place>
  <transition id="T0">
    <graphics>
      <position y="225.0" x="465.0"/>
    </graphics>
    <name>
      <value>T0</value>
      <graphics>
        <offset y="35.0" x="-5.0"/>
      </graphics>
    </name>
    <infiniteServer>
      <value>>false</value>
    </infiniteServer>
    <timed>
      <value>>false</value>
    </timed>
    <priority>
      <value>1</value>
    </priority>
    <orientation>
      <value>0</value>
    </orientation>
    <rate>
      <value>1</value>
    </rate>
  </transition>
  <arc target="T0" source="P0" id="P0 TO T0">
    <graphics>
      <position y="186.0" x="342.0"/>
      <position y="185.0" x="442.0"/>
    </graphics>
  </arc>

```

```
        </graphics>
        <type value="normal"/>
        <inscription>
            <value>Default,1</value>
        </inscription>
    </arc>
</net>
</pnml>
```

Listing A.1: The PNML of the place-transition net depicted in Figure A.1.