Enhancing the PEPA Workbench with Simulation and Experimentation Facilities

Stathopoulos Fotis

Master of Science
School of Computer Science
Division of Informatics
University of Edinburgh
2001

Abstract

The Performance Evaluation Process Algebra (PEPA) approach to modelling is supported mainly by the PEPA Workbench and the discrete event simulator. The aim of this thesis is the effective integration of these tools which will turn the PEPA Workbench into a powerful multifunctional framework that can be applied to the most of the cases of modelling needs. A necessary addendum to this would be porting both tools sources to Java and restructuring them in order to fit each other. The tool was extended to facilitate the comparison of the accuracy of the two integrated tools. Another novel feature of the new tool is a facility which enables the modeller to experiment with the model solving it again after changing the rates. The need for a graphical presentation of the results is now more substantial due to the increased complexity of output.

Acknowledgements

Many thanks to Stephen Gilmore for his excellent supervision throughout the project

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Stathopoulos Fotis*)

Dedicated to the victims of the tragedy on September 11, 2001.

Table of Contents

1 Introduction **1**

1.1 Overview 1

1.2 Project Aims 2

1.3 Design and Implementation 3

1.4 Outcome 4

2 PEPA Workbench and PEPAroni **5**

2.1 Overview 5

2.2 The PEPA Modelling Language 6

2.3 PEPA Workbench 9

2.4 PEPAroni 10

3 Implementation **13**

3.1 Overview 13

3.2 PEPAroni 13

3.2.1 Translation of Perl scripts 13

3.2.2 Porting of pizza sources to java 16

3.3 PEPA Workbench 24

3.3.1 Motivation for structuring 24

3.3.2 Packaging 24

3.4 Integration 28

4 Additional Features **31**

4.1 Overview 31

4.2 Compare tool 31
4.2.1 Motivation 31
4.2.2 Implementation 32
4.2.3 Usability 33
4.3 Experimentation Facility 34
4.3.1 Motivation 34
4.3.2 Implementation 35
4.3.3 Usability 36
4.4 Overall issues 36

5 The PEPA2 GUI **39**
5.1 Overview 39
5.2 Old Features 39
5.2.1 Load a model 39
5.2.2 Select an option 40
5.2.3 Run the model 40
5.2.4 Solve the model 41
5.2.5 Use of Rate, Walkabout and PSF utilities 41
5.3 Simulator 43
5.4 Comparison 45
5.5 Experimentation 47

6 Results and Evaluation **53**
6.1 Overview 53
6.2 Experimental Details 53
6.2.1 Evaluation of integrated simulator 53
6.2.2 Comparison tool 55
6.2.3 Experimentation Facility 56
6.3 Further Evaluation Issues 57
6.4 Conclusions 57
6.5 Future Directions 58
6.5.1 Intelligent Experimentation Programme 58

6.5.2 Graphical Tree-form Walkabout Facility 58
6.5.3 Handling of memory exhaustion 58

A Test Model **61**

B Gnuplot file created by the comparing tool **63**

C Gnuplot file created by the experimentation tool **67**

Bibliography **69**

List of Figures

3.1 PEPA2 structure 25

5.1 Loading a model 40
5.2 The model has been run and solved with steady state method 42
5.3 Setup screen for the simulation 43
5.4 The simulation is running 44
5.5 Output of simulation (interpreted) 45
5.6 Graphical output of comparing tool 46
5.7 Text output of comparing tool 47
5.8 Experiment setup screen 48
5.9 PSF for the experiment 49
5.10 The experiment is running 50
5.11 Output of 2D experiment 51
5.12 Experiment setup screen, setting one rate variable 51
5.13 Output of 1D experiment 52

6.1 Graphical output of comparing tool for long simulator run 55

List of Tables

6.1 Comparison of PEPA2 simulator and PEPAroni performance 54
6.2 Minimum time amounts required for the simulator to cover the state space 56

Chapter 1

Introduction

1.1 Overview

The performance of a system strongly influences tasks like designing and using it. The general aim of Performance Analysis is to ensure that a system achieves the highest performance given certain constraints. It may be carried out in any stage of the product life-cycle. Modelling and simulation are methods which are commonly used by performance analysts to represent constraints and optimise performance. The nature of the system could be of any type: computer systems, networks, manufacturing, etc. We use simulation models to represent a system in a form which allows us to make predictions about the behaviour of the system. How accurate these predictions are going to be depend on the detail we have invested in the model. In all cases, a model is an abstraction of the system. Once the model has been defined and parameterised it can be used to investigate numerically the system behaviour [1]. Modelling offers flexibility, safety, economy and extended facilities to system design and evaluation.

There are various methods of modelling like queueing networks, stochastic Petri nets and process algebras. Process algebras are high-level model specification languages for low-level stochastic models. The advantages of stochastic process algebras are that they incorporate the attractive features of process algebras and thus bring to the area of performance modelling several attributes which are not offered by the existing formalisms. Perhaps the most important such feature is the compositionality which is inherent in the models and can be exploited during their analysis [2]. Performance Evaluation Process Algebra (PEPA) is a stochastic process algebra which is used for modelling systems composed of concurrently active components which cooperate and share work. It extends classical process algebras by associating a random variable, representing duration, with every action. PEPA allows the modeller to study either behavioural or performance properties.

PEPA is supported by a suite of tools which include the PEPA Workbench and PEPAroni. The PEPA Workbench analyses PEPA descriptions and transforms them into a form suitable for solution. PEPAroni is a discrete-event simulator for PEPA. These tools have been implemented separately although they serve a common goal.

1.2 Project Aims

The core objective of the project was the integration of these two tools in one new, enhanced version of the PEPA Workbench. The first version of the PEPA Workbench was available when the project was received. It was unstructured (no packages) and the basic functionality had been implemented. PEPAroni was also available but it was implemented in Pizza and it also contained some Perl scripts. Pizza is a language which extends Java with additional data structures, the most useful of which are the algebraic data types and the parametric polymorphism, which significantly simplify the implementation of complex programs. Unfortunately, Pizza has stopped progressing and we need to re-implement the Pizza classes as pure Java classes. In addition this transformation will allow us to smoothly integrate the tool with the workbench. Similarly the Perl scripts should be ported to Java.

Apart from the above goal, further objectives were set to be achieved time allowing. One of them was the implementation of a tool which would compare the output of the Workbench and PEPAroni in the new integrated version. This would allow us to evaluate the tools and experiment on their accuracy for various input. Another useful facility would be an Experimental Programme. A series of experiments could be conducted by numerically re-evaluating the state space over a predetermined range of values of the symbolic rates. This could give rapid feedback of the performance metrics of a model, particularly bottlenecks in the system, in a short time frame.

In short the goals of the project are as follows:

- Translation of PEPAroni source to Java
- Re-structuring of PEPA Workbench and PEPAroni
- Integration
- Comparison tool
- Experimentation tool

1.3 Design and Implementation

The design of the Java version of PEPAroni did not begin from scratch, the Pizza version already existed and its source code was available for analysis. Pizza is a strict superset of Java that incorporates three ideas from the academic community: parametric polymorphism, higher-order functions, and algebraic data types. Pizza is defined by translation into Java and compiles into the Java Virtual Machine, requirements which strongly constrain the design space. Nonetheless, Pizza fits smoothly to Java, with only a few rough edges. Most of the methods and class structures found in the Pizza version of PEPAroni can be reused. Pizza was chosen as the implementation language primarily because of the pattern matching offered, the polymorphic definitions being more of a convenience. The translation of algebraic types includes a tag indicating which algebraic constructor is represented. Each "case" declaration introduces a subclass with a constructor that initialises the tag and the fields. The "switch" construct is translated to a "switch" on the tag, and each case initialises any bound variables to the corresponding fields. We have two methods to translate parametric polymorphism: heterogenous and homogeneous translations. A heterogenous translation that produces a specialised copy of code for each type at which it is used, and a homogenous translation uses a single copy of the code with a universal representation [3]. Typically the heterogenous translation yields code that runs faster, while the homogenous translation yields code which is more compact.

The Perl scripts were ported to Java easily as they were of small size. Their functionality was to handle and interpret the text output of the simulation and compute state rates and other statistics.

Following the translation, both tools were structured into homogeneous packages. Every package contains a functionality like the GUI, the lexer, the basic types of the system. In that way the reuse of code was minimised and the integration of the tools was simplified. Furthermore the code become more usable and easier to be extended.

The other facilities actually began with a clean sheet using the outputs of the new PEPA Workbench which will be referred to as PEPA2 throughout this report. The interface analysis showed that the usefulness of the new facilities (Comparison and Experimentation tool) would be dramatically increased if graphical feedback was used. As a consequence of that the output of the tools was formed into files which are able to be plotted by the gnu-plot utility of Linux.

1.4 Outcome

PEPA2 was built with all the stated objectives being met. Testing of the new simulation facility of the system has shown complete conformance of the outputs with those of PEPAroni. This gives some degree of confidence in its correct behaviour in the new environment of PEPA2. A big addendum to the PEPA2 is that the comparing tool provides a mechanism to check that both integrated tools are working properly. The new experimentation facility enhanced PEPA2 in terms of usability and speed. A model can be solved easily multiple times, boosting the performance analysis procedure. PEPA2 will be added to the suite of performance tools available to the PEPA user community.

Chapter 2

PEPA Workbench and PEPAroni

2.1 Overview

In performance modelling we are concerned with the process of building and evaluating models of complex systems efficiently and accurately. The advantages of compositionality and flexibility of stochastic process algebras has led to them being widely used to model increasingly larger and more complex systems.

PEPA (Performance Evaluation Process Algebra) extends classical process algebra with the capacity to assign rates to the activities which are described in an abstract model of a system [4]. Taken together, the information about the rates of performance of activities and the definition of the outcome of performing an activity specify a stochastic process and thus PEPA is said to be a stochastic process algebra. The PEPA language has been used to investigate the behaviour and performance of a diversity of systems.

The PEPA language is supported by a suite of modelling tools which assist in the solution and analysis of PEPA models [5]. The design and development of these tools have been influenced by a variety of factors, including the wishes of other users of the tools to use the language for purposes which were not anticipated by the tool designers. The suite of PEPA tools has adapted and grown to attempt to serve these needs, while continuing to support the language designers themselves. Among the variety of tools two of the most important are considered to be the PEPA Workbench and PEPAroni.

2.2 The PEPA Modelling Language

Process algebras offer several attractive features which are not necessarily available in existing performance modelling paradigms. The most important of these are compositionality, the ability to model a system as the interaction of its subsystems, formality, giving a precise meaning to all terms in the language, and abstraction, the ability to build up complex models from detailed components but disregarding internal behaviour when it is appropriate to do so. Queueing networks offer compositionality but not formality; SPN and GSPN offer formality but not compositionality; neither queueing networks nor Petri nets offer abstraction mechanisms [1].

PEPA extends classical process algebra by associating a random variable, representing duration, with every action. These random variables are assumed to be exponentially distributed and this leads to a clear relationship between the process algebra model and a Markov process.

PEPA models are described as interactions of components. Each component can perform a set of actions: an action a is described as a pair (α, r), where α is the type of the action and $r \in \Re^+$ is the parameter of the exponential distribution governing its duration. Whenever a process P can perform an action, an instance of a given probability distribution is sampled: the resulting number specifies how long it will take to complete the action. A small set of combinators is used to build up complex behaviour from simpler behaviour. The combinators are familiar: sequential composition (prefix), choice, synchronisation (cooperation) and abstraction (hiding). We explain each of them below, in terms of a extremely simple model of a webbased information system [2].

Prefix (.): A component may have purely sequential behaviour, repeatedly undertaking one activity after another and eventually returning to the beginning of its behaviour. A simple example is a web server, which allows one data transfer at a time. Each browser requiring web pages etc. will need to acquire access to the server and only when the transfer is complete will the server be released and available again for acquisition.

$$Server \stackrel{def}{=} (get, \top).(download, \mu).(rel, \top).Server$$

In some cases, as here, the rate of an action is outside the control of this component. Such actions are carried out jointly with another component, with this component playing a passive role. For example, the server is passive with respect to the get action and this is recorded by the distinguished symbol,$\top$ (called “top”).

Choice (+): A choice between two possible behaviours is represented as the sum of the possibilities. For example, if we consider a browser in the information system, displaying the current data may have two possible outcomes: demand for access to data available in the local cache (with probability p_1) or demand for access to data stored at the remote server (with probability $p_2 = 1 - p_1$). These alternatives are represented as shown below:

$$\begin{aligned} Browser &\stackrel{def}{=} (display, p_1\lambda).(cache, m).Browser \\ &+ (display, p_2\lambda).(get, g).(download, \top).(rel, r).Browser \end{aligned}$$

A race condition is assumed to govern the behaviour of simultaneously enabled actions and the continuous nature of the probability distributions ensures that the actions cannot occur simultaneously. Thus a sum will behave as either one summand or the other. When an action has more than one possible outcome, e.g. the display action in the browser, it is represented by a choice of separate actions, one for each possible outcome. The rates of these actions are chosen to reflect their relative probabilities (decomposition principle). Note that in a GSPN we would represent this situation by a single timed transition to represent the display action, which when it fired enabled two immediate transitions with weights p_1 and p_2 to reflect the different possible outcomes.

Cooperation ($\bowtie_L$): We have already anticipated that the browser and the server in the example will be working together within the same system. This will require them to cooperate when the browser needs to download data which is not available locally. In contrast, the local activities of the browser can be carried out independently of the server. Cooperation over given actions is reflected in the cooperation by the cooperation set, $L = \{get, download, rel\}$ in this case. Actions in this set require the simultaneous involvement of both components. The resulting action, a shared action, will have the same type as the two contributory actions and a rate reflecting the rate of the action in the slowest participating component. Note that this means that the rate of a passive action will become the rate of the action it cooperates with. If, for simplicity, we assume that the system consists of just two browsers, the system is represented as the cooperation of the browsers and the server as follows:

$$WEB \stackrel{def}{=} (Browser||Browser) \underset{L}{\bowtie} server \qquad L = \{get, download, rel\}$$

The combinator $||$ is a degenerate form of the cooperation combinator, formed when two components behave completely independently, without any cooperation between them, as in the case of the two browsers. This pure parallel combinator can be thought of as cooperation over the empty set: ($Browser \underset{\emptyset}{\bowtie} Browser$).

Abstraction ($/$): It is often convenient to hide some actions, making them private to the component or components involved. The duration of the actions is unaffected, but their type becomes hidden, appearing instead as the unknown type τ . Components cannot synchronise on τ . For example, as we further develop the model of the information system we may wish to hide the access of a browser to its local cache. This might lead to a new representation of the browser:

$$Browser' \stackrel{def}{=} Browser/\{cache\}$$

and a corresponding new representation of the system:

$$WEB' \stackrel{def}{=} (Browser' \parallel Browser') \underset{L}{\bowtie} Server \qquad L = \{get, download, rel\}$$

Use of the hiding combinator in this way has two implications. Firstly, it ensures that no components added to the model at a later stage can interact, or interfere, with this action of the browser. Secondly, private actions are deemed to have no contribution to the performance measures being calculated and this might subsequently suggest simplifications to the model.

Throughout the simple example above we have used constants (names) such as Server to associate identifiers with behaviours. Using recursive definitions we are able to describe components with infinite behaviours. Representing the components of the system as separate components means that we can easily extend our model. Now we may want to consider a system consisting of more than two browsers which act independently of each other but compete for the use of the web server. This extension may be achieved compositionally by combining more instances of the Browser component already described. For example, in the case of four browsers we have:

$$WEB4 \stackrel{def}{=} (Browser \parallel Browser \parallel Browser \parallel Browser) \underset{L}{\bowtie} Server$$

where, as before, $L = \{get, download, rel\}$

2.3 PEPA Workbench

The PEPA Workbench is a modelling tool for composing and checking descriptions of systems composed of communicating concurrently active components. The PEPA Workbench processes a textual description of a PEPA model, and generates output which can be used to calculate a steady state or transient probability distribution for the model. The Workbench performs some well formedness checks on the model as it attempts to internally generate a derivation graph [1].

The PEPA Workbench exists in several different versions but in this project the Java version was extended and adapted. In this version basic functionality has been implementing and the system can solve models and interface with the Maple, Mathematica or Matlab linear algebra packages for further processing. The PEPA workbench supports a graphical user interface which simplifies tasks like loading and running a model from the user perspective.

The assumption of exponentially distributed random variables creates some weaknesses of the methods implemented in the Workbench. For example a model with non-exponential variables cannot be solved numerically. However, simulation methods can cope with such number distributions.

One goal of this project is to enhance the Workbench with additional facilities like the simulation tool. A necessary addendum to this would be re-structuring the class system as a set of packages. Another goal is to equip the PEPA Workbench with the ability to experiment on the models. This means to check its performance for a range of given rate values. At the end, in order to make the Workbench GUI more friendly and attractive to users advanced features of Java 1.2's Swing API can be used.

2.4 PEPAroni

A stochastic simulation model can be regarded as an algorithmic abstraction of the system, a simulation model gives a representation which when executed reproduces the behaviour of the system. Using the analytic approach of Markov processes we characterised all possible sample paths by the global balance equations. Using simulation we investigate the sample paths directly. In other words we allow the model to trace out a sample path over the state space. Each run of the simulation model will generate another, usually distinct, sample path [6].

There are many reasons why simulation may be preferable to analytic modelling. Markovian modelling relies on many assumptions and abstractions so they may not be appropriate for the system being studied. Simulation models allow us to represent a system at arbitrary levels of detail, although there is always a trade-off between how elaborate the model is and how long it takes to produce a statistically significant run.

In addition, in some cases we are not interested in the steady state behaviour of a system, but in its transient behaviour. Some systems never reach a steady state. Even those that do will usually have a "warm-up" period while the behaviour settles into the regular pattern which characterises steady state. The analytic solutions we have considered earlier in the course ignore this period since the global balance equations only capture the behaviour after steady state has been reached. However, it is not always possible to extract all the information we want about a system from its steady state behaviour. A sample path derived from a simulation model will clearly represent transient behaviour in addition to steady state.

Finally, in most cases solving a model analytically involves constructing and storing the complete state space of the model. For example, for a Markov process with N states solving the global balance equations involves (at least) an N * N matrix (the generator matrix) and a vector with N elements. When N becomes very large this becomes infeasible. In contrast, in a simulation model the state space is generated "on-the-fly" by the model itself during execution so it does not need to be all stored at once [1].

Simulation models are complex computer programs. The discrete event simulator, one of the performance modelling tools available to the PEPA user community named PEPAroni, was developed to investigate situations in which general distributions may successfully replace exponential distributions without affecting the model's steady state solution. It was implemented in Pizza and uses SimJava, a language with built-in simulation features. Its skeletal class structure evolves naturally from the properties of process algebras. Some of this structure is inherited by the Java PWB but the differing objectives of the two applications limits commonality in their implementation.

Chapter 3

Implementation

3.1 Overview

The integration of PEPAroni into the workbench had to be accomplished efficiently and precisely. One of the difficulties that were faced was the low commentary of the code of PEPAroni. This was overcome by experimentation of its functionality. After defining how the system works it was easy enough to start integrating. The integration had several phases explained below. In short, first we reformed both the PEPA workbench and PEPAroni and after the combination was achieved.

3.2 PEPAroni

3.2.1 Translation of Perl scripts

The first step of the PEPAroni re-implementation was the translation of the Perl scripts in the source code. These scripts interpret the simulation output and create statistics. Actually there are two scripts at the original PEPAroni system : *interp.pl* and *stats.pl*. Although Perl offered easier implementation, its existence would consist a drawback for the new system, since it is designed to be implemented in pure Java. So both scripts had to be re-implemented in Java. We will show their functionality through a sample run of the simulator.

Peparoni The simulator creates a series of events that change the model state. The event triggerings and the new states of the model are stored in *sim.out* file in a text format as shown below.

```
event fired - I am activity (a,1.0) after 0.31338588671951706
the simderiv will be (b,2.0).(b,3.0).A1 + (c,4.0).(b,5.0).A1
event fired - I am activity (b,2.0) after 0.11463013247113715
the simderiv will be (b,3.0).A1
event fired - I am activity (b,3.0) after 0.5644617380094357
the simderiv will be A1
event fired - I am activity (b,1.4) after 2.0661694933651393
the simderiv will be A
event fired - I am activity (a,1.0) after 1.4248837843851896
the simderiv will be (b,2.0).(b,3.0).A1 + (c,4.0).(b,5.0).A1
event fired - I am activity (c,4.0) after 0.42176043318706524
the simderiv will be (b,5.0).A1
....
```

interp.pl When the *interp.pl* run on this file it actually interprets it, counting the residence of each state of the model during the simulation. The output can be stored in a file again in text format.

```
 stats:
*simulation time 5.74080131461144
(b,3.0).A1 time 0.564461738009436
A time 1.87521432358691
(b,5.0).A1 time 0.226501370895604
A1 time 2.22969520737206
(b,2.0).(b,3.0).A1 + (c,4.0).(b,5.0).A1 time 0.844928674747427
stats:
*simulation time 10.3863801355944
(b,3.0).A1 time 0.564461738009436
A time 2.98718230438217
```

```
(b,5.0).A1 time 0.655406288965971
A1 time 5.17408344193678
(b,2.0).(b,3.0).A1 + (c,4.0).(b,5.0).A1 time 1.0052463623
...
```

stats.pl If *stats.pl* apply on the output of *interp.pl* it gives back useful statistics of the simulation. These are considered to be the state residence, the variance of state residence and finally, the firings of the events. This output, which represents the simulator output, has the following format for the same example:

```
 -- STATS --
mean of state residence for  A1   is 32.73024%
mean of state residence for  A   is 48.566067%
mean of state residence for  (b,5.0).A1   is 6.057935%
mean of state residence for  (b,3.0).A1   is 5.221594%
mean of state residence for  (b,2.0).(b,3.0).A1 + (c,4.0).(b,5.0).A1
                                          is 7.424087%
for total simulation time 914.8781
{
 event (a,1.0) fired, times =418,
 event (b,1.4) fired, times =418,
 event (b,2.0) fired, times =146,
 event (b,3.0) fired, times =146,
 event (b,5.0) fired, times =272,
 event (c,4.0) fired, times =272}

variance of state residence for  A1   is 0.08679688%
variance of state residence for  A   is 0.052843045%
variance of state residence for  (b,5.0).A1   is 0.002996396%
variance of state residence for  (b,3.0).A1   is 0.42998526%
variance of state residence for  (b,2.0).(b,3.0).A1 +
                            (c,4.0).(b,5.0).A1   is 0.00423423%
```

In the new version of PEPAroni both functionalities are combined in one Java class (*stats.class*). It is set to apply again autonomously on the *sim.out* file because we did not want to lose the initial simulation output. The main implementation issue that is worth mentioning is the use of *TreeMap* datatype which allows holding a set of pairs of values. In our case the state and the resident time was held in every position. While parsing the *sim.out* file, the residence time is reevaluated for every appearance of its adjustment state. Through that structure various statistics could easily be extracted.

3.2.2 Porting of pizza sources to java

PEPAroni had been implemented in Pizza. Pizza simplified the implementation through the algebraic data types and the parametric polymorphism that supports. However, Pizza has stopped progressing and moreover the re-implementation of PEPAroni is essential for the future cooperation with PEPA Workbench [7] .

3.2.2.1 Pizza

Pizza is a strict superset of Java that incorporates three ideas from the academic community :

- parametric polymorphism,
- higher-order functions, and
- algebraic data types.

Pizza attempts to make these ideas accessible by translating them into Java. We mean that both figuratively and literally, because Pizza is defined by translation into Java. The requirement that Pizza translate into Java strongly constrained the design space. Despite this, it turns out that the new features integrate well: Pizza fits smoothly to Java, with relatively few rough edges [3].

Promoting innovation by extending a popular existing language, and defining the new language features by translation into the old, are also not new ideas. They have proved spectacularly successful in the case of C++.

3.2.2.2 Heterogenous and homogenous translations

There are two types of translations: heterogenous translations that produce a specialised copy of the code for each type at which it is used, and homogenous translation that use a single copy of the code with a universal representation. Typically the heterogeneous translation yields code that runs faster, while the homogenous translation yields code that is more compact.

The two translations correspond to two common idioms for writing programs that are generic over a range of types. The translations are surprisingly natural, in that they are close to the programs one would write by hand in these idioms. Since the translations are natural, they help the programmer to develop good intuitions about the operational behaviour of Pizza programs.

Mixtures of heterogenous and homogenous translation are possible, so programmers can trade size for speed by adjusting their compilers rather than by rewriting their programs. The best balance between performance and code size will typically be achieved by using the heterogenous translation for base types and the homogenous translation for reference types.

3.2.2.3 The translation of algebraic data types

Object types and inheritance are complementary to algebraic types and matching. Object types and inheritance make it easy to extend the set of constructors for the type, so long as the set of operations is relatively fixed. Conversely, algebraic types and matching make it easy to add new operations over the type, so long as the set of constructors is relatively fixed. The former might be useful for building a prototype interpreter for a new programming language, where one often wants to add new language constructs, but the set of operations is small and fixed (evaluate, print). The latter might be useful for building an optimising compiler for a mature language, where one often wants to add new passes, but the set of language constructs is fixed.

An object oriented environment can encode pattern matching through the use of one abstract base class and one subclass for each algebraic case being encoded. Pizza provides an easy to use syntax to avoid the complicated abstract and concrete class relationship typically found in object oriented programming. Inclusion of case statements within the Process class as detailed below implements constructors for the five subtypes listed.

```
class Process {
     case Prefix(Activity A, Process P);
     case Sum(Process P1, Process P2);
     case Cooperation(Process P1,ActionSet l,Process P2);
     case Hide(Process P1,ActionSet l);
     case Var(String str)
     ...
     method illustration() {
       switch (this) {
          case Prefix(_): ...
          case Sum(_): ...
          ... }
          }
}
          Example 3.1 Algebraic data types in Pizza
```

The switch statement provided by Java can then be used to determine the constructor used and the behaviour of the program can be changed accordingly. An example of this is shown.

The switch statement actually operates on a tag which is included as a data member in the base class and is assigned according to which of the algebraic constructors is used. This is a very useful technique for accessing instances of a class and is frequently exploited in PEPAroni code.

An algebraic type for process is shown in the above examples. The five case declarations introduce constructors for the algebraic type: Prefix to represent the Prefix process with two fields, Activity A, and Process P; Sum to represent the Sum process with two fields, Process P1, and Process P2; Cooperation to represent Cooperation process with three fields, Process P1, ActionSet l, and Process P2; Hide to represent the Hide process with two fields, Process P1, and ActionSet l; and Var to represent the

Var process with one field, String str. The method illustrates matching against a given process.

The translation from Pizza to Java is shown in Example 3.2. The translation class includes a tag indicating which algebraic constructor is represented. Each case declaration introduces a subclass with a constructor that initialises the tag and the fields. The switch construct is translated to a switch on the tag, and each case initialises any bound variables to the corresponding fields.

```
class Process{
     final int Var_tag = 1;
     final int Prefix_tag = 2;
     final int Sum_tag = 3;
     final int Coop_tag = 4;
     final int Hide_tag = 5;
     int tag;
     method illustration(){
       switch(this.tag){
          case Prefix_tag:...
          case Sum_tag:...
          ...}
     }
}
class Prefix extends Process {
     Activity a;
     Process p;
     Prefix(Activity a, Process p) {
          this.Processtag = Prefix_tag;
          this.a = a; this.p = p;
          }
}
class Sum extends Process {
     Process p1;
```

```
    Process p2;
    Sum(Process p1, Process p2) {
        this.Processtag = Sum_tag;
        this.p1 = p1; this.p2 = p2;
    }
}
```

Example 3.2 Translation of algebraic types into Java

3.2.2.4 The translation of parametric polymorphism

Polymorphism provides a general approach to describing data or algorithms where the structure is independent of the type of data values manipulated. PEPAroni makes extensive use of the polymorphic list structure that the language provides. Pizza was chosen as the implementation language primarily because of the pattern matching offered, the polymorphic properties being more of a convenience. Nevertheless, extensive use is made of the simulated polymorphism in the list structures and priority queue data structures in PEPAroni and this gives greater run time security of the code.

Java does not support polymorphism but Java simulates polymorphism (as in fact do Java arrays). As a trivial example, we consider an algorithm to swap a pair of elements, both of the same type. Pizza code for this task appears in Example 3.3.

```
class pair<elem>{
    elem x;
    elem y;
    pair(elem x, elem y) {
        this.x = x;
        this.y = y
    }
    void swap(){
        elem t = x;
        x = y; y = t;
    }
```

```
}
pair<string> p = new pair(``world!'', ``hello,'');
p.swap();
system.out.print(p.x + p.y);

pair<int> q = new pair(22,64);
q.swap();
system.out.println(q.x - q.y);
```

Example 3.3 Polymorphism in Pizza

The class Pair takes a type parameter elem. A pair has two fields x and y, both of type elem. The constructor Pair takes two elements and initialises the fields. The method swap interchanges the field contents, using a local variable t also of type elem. The test code at the end creates a pair of integers and a pair of strings, and prints

```
Hello, world! 42
```

to the standard output. (Here + is string concatenation.)

We consider two ways in which Java may simulate polymorphism. The first method is to macro expand a new version of the Pair class for each type at which it is instantiated. We call this the heterogenous translation, and it is shown in Example 3.4.

```
class Pair_String {
    String x;
    String y;
    Pair_String (String x, String y) {
        this.x = x;
        this.y = y;
    }
    void swap () {
        String t = x;
```

```
            x = y; y = t;
      }
}
class Pair_int {
      int x;
      int y;
      Pair_int(int x, int y){
            this.x = x;
            this.y = y;
      }
      void swap(){
            int t = x;
            x = y; y = t;
      }
}
Pair_String p = new Pair_String(``world!'', ``Hello,'');
p.swap();
System.out.println(p.x + p.y);

Pair_int q = new Pair_int(22,64);
q.swap();
System.out.println(q.x-q.y);
```

Example 3.4 Heterogenous translation of polymorphism in Pizza

The appearance of parameterised classes Pair_String and Pair_int causes the creation of the expanded classes Pair_String and Pair_int, within which each occurrence of the type variable elem is replaced by the types String and int, respectively. The second method is to replace the type variable elem by the class Object, the top of the class hierarchy. We call this the homogenous translation, and it is shown in Example 3.5.

```
class pair{
```

```
    Object x;
    Object y;
    pair(object x, object y){
        this.x = x;
        this.y = y
    }
    void swap(){
        object t = x;
        x = y; y = t;
        }
}
class Integer{
    int i;
    Integer (int i){
    this.i = i;
    }
    int intvalue(){
    return i;
    }
}
Pair p = new Pair((Object)''world!'',(Object)''Hello,'');
p.swap();
System.out.println((String)p.x + (String)p.y);

Pair q = new Pair((Object)new Integer(22),
(Object)new Integer(64));
q.swap();
System.out.println(((Integer)(q.x)).intValue()-((Integer)(q.y)).intValue());
```

Example 3.5 Translation of Polymorphism into Java

The key to this translation is that a value of any type may be converted into type

Object, and later recovered. Every type in Java is either a reference type or one of the eight base types, such as int. Each base type has a corresponding reference type, such as Integer, the relevant fragment of which appears in Example 3.5.

If v is a variable of reference type, say String, then it is converted into an object o by widening (Object)s, and converted back by narrowing (String)o. If v is a value of base type, say int, then it is converted to an object o by (Object)(new Integer(v)), and converted back by ((Integer)o).intValue(). (In Java, widening may be implicit, but we write the cast (Object) explicitly for clarity.)

Java programmers can and do use idioms similar to the heterogenous and homogenous translations given here. Given the code duplication of the first, and the lengthy conversions of the second, the advantages of direct support for polymorphism are clear.

The Pizza sources of PEPAroni were completely and efficiently ported to Java following the methods explained above.

3.3 PEPA Workbench

3.3.1 Motivation for structuring

When the project received the PEPA Workbench did not have any particular file structure. The source code was set in a pile of Java files. As a result there was big uncertainty about how the system was functioning and how the classes were communicating. The effort of further development of the Workbench would have dramatically increased if the Workbench did not become structured. Furthermore, the unstructured form would have obstructed the integration of PEPAroni. For the above reasons the Workbench source code was split into a set of packages.

3.3.2 Packaging

Depending on their functionality and commonality the Java files were put into packages as shown in the picture below. The new structure make clear the dependencies between the classes.

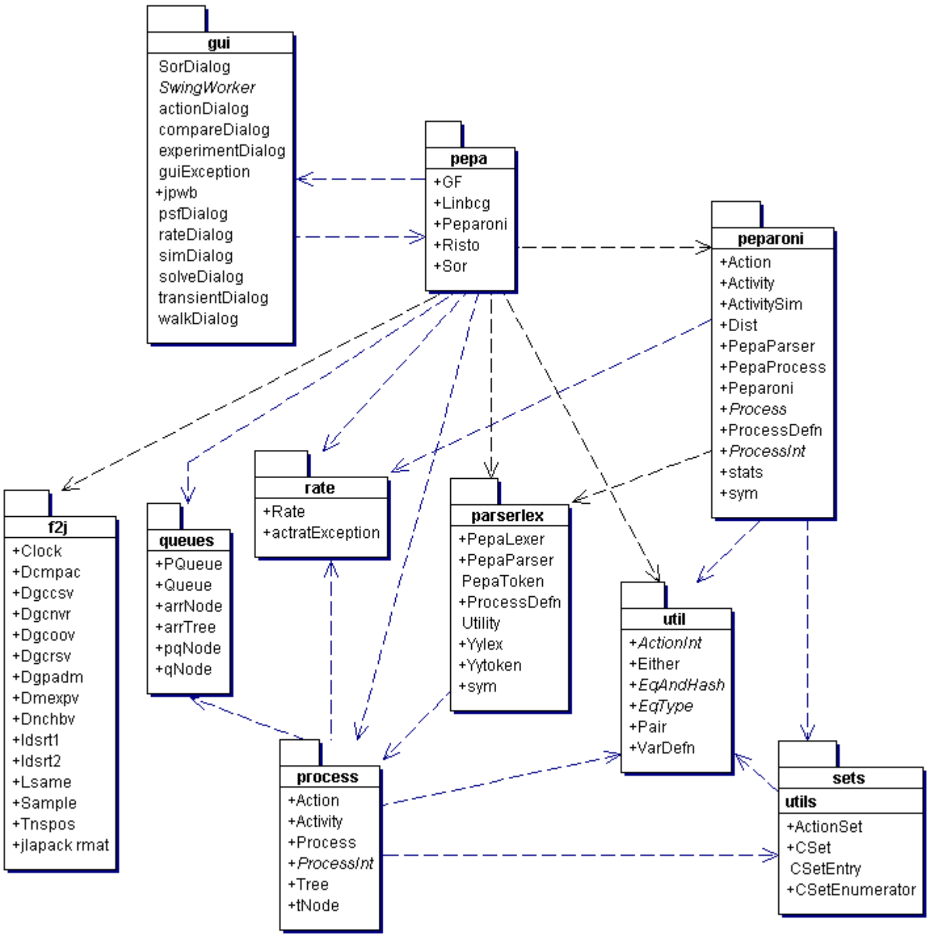

Figure 3.1: PEPA2 structure

3.3.2.1 gui

Contains the main class of the system (*jpwb.class*) and all the GUI utilities. The experimenting and comparing facilities have also been implemented in this package. It communicates mainly with the *pepa* package.

3.3.2.2 parserlex

The parser and the lexer of the Pepa syntax have been placed here. There are linked to each other and they are used from the *peparoni* and the *pepa* packages. The parser is generated by the Yacc-like CUP parser generator.

3.3.2.3 pepa

This contains the engine of the Workbench. *Peparoni.class* is the main class of the package and it is responsible for the derivation of the transition graph and the output of states and transitions to file. It solves the PEPA models using the utilities in *f2j* package and in the new version it activates the simulation and experimentation facilities as well. It does all the input/output relations and leaves the number crunching to the lowest algorithms. This does deal with holding the sparse storage after being input and relating the data back to jpwb to be shown to the user. It communicates with most of the packages and it is considered to be the heart of the system.

3.3.2.4 peparoni

This is the integrated PEPAroni transformed into a package for the new Workbench. It uses the auxiliary data structures and the lexer of PEPA. The parser of PEPAroni did not fit with the PEPA parser, so PEPAroni parser maintained in the new system inside this package.

3.3.2.5 process

The different processes/behaviours that may occur from the description of the model within the PEPA notation are defined here. Five constructors are implemented for the description of the different notation. The Class structures include a hashtable for expanding the component definitions, a sequential process tree to determine the assigned sequential state number of a sequential component, an array of the sequential components to return the sequential component if the state number is known, and a list structure which maintains the static structure of the parsed process. It is used for the solution of a model as it efficiently represents the state space as a structure of a process [8]. It is linked with the *pepa* and *process* packages.

3.3.2.6 queues

It contains the queue data structures supporting the priority queue functionality. The last is needed for ordering of all the instantaneous transitions out of a state avoiding duplications of arcs in the derivation graph. It is used by *pepa* and *process* packages.

3.3.2.7 rate

The *Rate.class* is implemented in this package. It specifies the type of rate for use in the PEPA notation. Notionally rates in PEPA are of two types, specified and passive. It uses five of the combinator rate subclasses to produce this types. This utility is used from both the PEPA Workbench and PEPAroni.

3.3.2.8 sets

It contains data structures that provide a set enumerator for the Workbench. It is used mainly by the *peparoni* package. It also contains the *utils* package that is needed for the implementation of Set and CSet data types.

3.3.2.9 util

Auxiliary data types and methods are defined here for use from *pepa* and *peparoni*.

3.3.2.10 f2j

The routines needed for analysing and solving a model are defined here. These routines are produced by f2java. f2java is a Fortran to Java translator. They consist the essential minimum substitute to BLAS and LAPLACK, linear algebra packages needed to analyse numerically a PEPA model. In that way the Workbench can operate autonomously even if these libraries are not installed. They are used mainly from the Workbench engine, the *pepa* package.

3.3.2.11 debug

Contains a single class for creating debugging information. It is not actually an essential part for the Workbench functionality. It is used in almost all the Java classes of the system.

3.4 Integration

The integration of PEPAroni into the PEPA Workbench was simplified a lot after the modifications mentioned before. The Java version of PEPAroni could combined easily with the structured PEPA Workbench. However, some work still should be done to achieve efficient and reliable combination of the two systems. Duplication of code should be minimised and new modules should be implemented.

The final step before starting integrating was to find the commonalities of the systems. Both take as input the same type of models (in PEPA notation). So, as expected, they were using the same lexer, the same tokens and other utilities. As a consequence of that PEPAroni was directed to use the Workbench lexer. The parsers were different in some points although they were both CUP generated and largely identical to each other. These differences prevented the easy combination of the parsers so each system kept its own. The rate utility had also been implemented identically and it was inserted unchanged in the new system (*rate* package). The same auxiliary types were using in both systems and they maintained in the new one (*util* and *sets* packages). PEPAroni was using some additional set types that were added (*sets.util* package). Finally the debug utility (*debug* package) was kept as it was in the new system.

The integration completed successfully by creating the *peparoni* package in the Workbench system and linking it with the other packages. A new pop-up menu in the Workbench GUI was created that give the ability to set and run the simulation. Additional modules have been added to the simulation software that allow the simulation to run for a specific amount of time or untimed. In order to achieve that efficiently, Java threads were added in the simulation code and the *SwingWorker* class was used. This is an abstract class provided by Sun that allows to perform GUI-related work in a dedicated thread. So while the simulation is running, time can be counted and graphical feedback can be provided. This was essential for the system consistency and usability. If a simulation ran without user's attention it would create great frustration and lack of clarity of the system state, considering that a simulation could be a very long procedure lasting for hours or days.

An implication of the threads implementation is that the user can work with other facilities simultaneously. However, this possibility was a welcome result and not an objective of the integration. Further implementation of the Workbench could support accurately and consistently this facility but this was beyond the project aims.

In the new system the simulation output is still stored in a file and the interpretation is a one step procedure activated from the Workbench menu. The final statistics appear in a pop-up window. The user can save the statistics manually, copying the *stats.out* file. This file has the format mentioned before, reporting the state residence, the variance of state residence and the event firings and the simulation time.

Chapter 4

Additional Features

4.1 Overview

The integration of PEPAroni in the PEPA workbench was followed by the appearance of new necessities for the system. The new system would have limited strength and usability if facilities for analysing and experimenting on the new outputs were not provided. At this first stage of further development two kind of tools seemed to be most valuable. Firstly, a comparing tool for comparing the results of PEPA numerical solution and PEPAroni simulation and secondly, an experimentation tool for solving a model repeatedly over a range of given rate values.

4.2 Compare tool

4.2.1 Motivation

The definition of the problem was quite simple. We should develop a tool which could efficiently show if and how the steady state rates derived from the PEPA Workbench solver match the rates given from the simulator output. The importance of this facility could be extremely significant for complex models with unstable behaviour. For example, comparing the simulation output of a model with naturally non-exponential distributed random variables with that of the PEPA solver assuming exponential distributed variables, we could see if this assumption is reliable and can be used for further analysis of the model. Furthermore, the tool would be advantageous in any case for the designing and the performance analysis procedures of a system. It could even be used as a testing tool for the system in order to validate the two integrated sub-systems. But the most important value of this tool would be that it significantly increases the usability of the integrated simulator as a performance analysis tool. The integration would not have any meaning if a tool for processing the results of the two systems did not exist.

4.2.2 Implementation

The values to be compared are the state rates coming from the steady state solution and the simulation. The reason we have chosen the steady state solution, among the other methods of solving a model, is that this method calculates analytically based on theoretical equations what the simulation is trying to reach, the probability distribution of every state over the state space as time goes to infinite.

The comparison tool initially implemented like a pop-up window containing a text array with the two rate values to be compared as column indexes and the model states as row indexes (each state has two rates derived from the simulation and the steady state solution).

In order to achieve this we had to introduce the *TreeMap* datatype as follows:

```
TreeMap states= new TreeMap();
TreeMap steadystaterates= new TreeMap();
TreeMap simrates= new TreeMap();
...
states.put(id,state);
steadystaterates.put(id,rate);
simrates.put(state,rate);
...
```

The *TreeMap* datatype effectively keeps a pair of values. We have used *states* to keep the PEPA notation of every state together with a state id, *steadystaterates* to keep the rates coming from the solver together with the state ids and *simrates* to keep the simulation rates with the state PEPA notation. In that way the implementation of the comparison was simplified a lot and comparisons can be drawn very easily retrieving the rates and the states of these structures. The simulation rates are taken from the interpreter in the *peparoni* package and the solver rates from *peparoni.class* in the *pepa* package. Some adjustments are made in *jpwb.class* of *gui* package and the last stage of this procedure takes place in the *gui* package (*simDialog.class*) where the rates and the states are passed to the output text area.

Although this first implementation offered accurate and consistent comparisons it could not give an overall idea for the general deviation of rates over the state space. In fact it was useful only for comparing individual state rates. It was necessary to support a quick and flexible tool for generic and overall comparisons of the simulator and the steady state solver. This was achieved by graphically representing the probability distribution of states. Using the same TreeMap data structures a temporary file (*compare.dat* in *models* directory) is created to be plotted (with *gnuplot* utility of Linux). This contains a graph where the x-axis represents the state ids and the y-axis the state rates. A sample format of the file is shown in Appendix B. Two lines are plotted, one for each method. The distance between them reveals if the rates derived from the solver and the simulator are matching generally. Of course in order to run the comparison tool both methods should have been executed for the same model. If the user wants to keep the graph she can manually save the temporary gnuplot file and reload it anytime.

4.2.3 Usability

The completion and testing of the comparing tool proved that PEPAroni has been successfully integrated in PEPA workbench as the probability distributions were closely matching for most of the models.

Further experimentation with the tool revealed some hidden strengths of it. For most of the models there are states that are not reached but only after long simulation run. This can be shown if we operate on a short simulation output. This could be critical for the system that the model represents. On the other hand, some states analytically have rates values much more higher than those derived from the simulation. This demonstrates the fact that the ideally calculations of the solver do not reflect precisely the actual behaviour of the model. Generally speaking, at the plotted graph, the simulation line represents the reality while the solver graph the theory. Obviously a lot of useful conclusions about the performance can be derived from the analysis of the comparison graph. Actually, it could play the role of an evaluation tool for the PEPA modelling and the Workbench. The system designer is provided now with two tools (PEPA and the simulator) which can be directly compared, in terms of outputs, and evaluated.

4.3 Experimentation Facility

4.3.1 Motivation

The workbench is able to solve efficiently a model given its PEPA notation and parametric values. Although this is quite useful we would like to be able to solve a model more than once for different values of the PEPA parameters, that are the rates of the model. This would transform the Workbench into a powerful performance analysis tool as we would be able to optimise the model through setting the appropriate rate values. Of course this could be achieved by hand changing the rates with the Workbench rate utility and solving the model again. But this is an inconvenient and slow way with no actual usefulness. In fact, it is not practical at all for the case of checking the performance of the system for a big range of a particular rate.

Furthermore, performance analysis requires retrieving the global rate of a set of important states. For example, we would like to check that the case of a single token appearing simultaneously with another one is practically impossible ($rate \rightarrow 0$) because this would be dangerous for the system. This case could involve more than one states and for real world systems it could be hundreds or thousands of states. Obviously is quite hard to count by hand the sum of the implicated state rates and rather impossible to experiment by changing the rate variables. This facility had not been implemented for the Workbench although it is considered to be a necessity for a complete modelling tool such as the PEPA Workbench.

4.3.2 Implementation

For the development of the tool a new class was created over the Workbench GUI. The new panel allow the user to setup the experiment choosing one or two rates as variables and selecting the set of states. The ranges of values and the steps should have been set from the user before running the experiment. The step should be positive if the rate is gradually increasing through the experiment or negative if it is decreasing. The final action before running the experiment is to choose the involved state or states. In case that only one state interests the user, this can be selected from a drop-down list. But for any other case, the PEPA State Finder tool (PSF) was linked to the experiment creation tool so it can be run from there. Running the PSF the user can create a set of states that are saved in a file as usual. The tool is quite consistent and reliable as it does not allow invalid values or null state input. When the user finally executes the experiment the associated states are retrieved from the file and their rates are calculated.

The mechanism of the experimentation tool is quite simple. A loop is created where the model is solved (steady state solver) multiply while the specified rates are changing. For that purpose a new function was created in *peparoni* class in *pepa* package. From the output of every loop the rates of the implicated states are added and the total is kept. The final output consists of a series of state rate values each of them related with a rate variable value or a pair of values. When the simulation ends the results are passed to *jpwb.class* which handles the workbench GUI and it is responsible for presenting that to the user. Through *experimentDialog.class* the results are output to the screen in a graphical way.

Again, because the experiment procedure could be long in terms of time and resources the progress bar that is used for the simulation was used again to reduce inconsistency and unsteadiness of the system. Threads were used for renewing the interface and making the progress bar operating feasibly. The same software modules were used here as in the integration of PEPAroni.

Obviously, like the comparing tool, the most efficient and consistent way of representing the outcome is graphically. The use of graphical representation of the experiment was essential for the tool as it would allow the user to create a visual representation of the performance of the system as the rates are changing. The plotted graph could be 2D (one parametric rate) or 3D (two parametric rates). The Linux *gnuplot* utility was used for that purpose. The output is saved in the appropriate format in a temporary file (*experiment.dat* in *models* directory) and the system loads *gnuplot* for this file. The result is a two or three dimensional graph representing the state rates over the rate variables values. If the user wants to keep the graph she can manually save the temporary gnuplot file.

4.3.3 Usability

Experimenting on various models has proved our initial assumption, the tool would be extremely useful for the design and performance analysis of any system. It is able to give rapid feedback of the performance metrics of a model in a short time frame, revealing bottlenecks and other characteristics that are dependant on the rate values. It can be used like a quick testing tool for the model behaviour over different circumstances and in a more technical context, different rate values. The procedure of assigning the appropriate values to the rates has simplified a lot as we can automatically evaluate the performance of the system for a huge amount of rates automatically. The produced graph is a very flexible and positive way to optimise the model for a given target defined by the selected set of states. Moreover the graph can intuitively reveal secret combinations of rate values where the model performs a totally unexpected behaviour. The ability of concurrent experimentation over two variables exploits the workbench and covers a vast part of the experimentation needs for performance analysis.

4.4 Overall issues

The experimentation tool in combination with the comparing tool actually forms a powerful device capable of executing accurate and efficient performance analysis and maximising of model optimisation. This can be achieved by passing intuitively the positive feedback from one tool to the other until we reach a satisfactory level of performance. In order to clarify that we will go through a practical example.

Assume we have to set the speed of some machines working concurrently, in some way in a factory, in order to maximise the overall performance. After defining the

PEPA model we have no clue about the speed we need in each case except probably a recommended value. Experimenting on the model with the tool for a range of speed values around the recommended for every machine we can detect where, or under which circumstances, the performance increases and tends to be maximised. Passing this first set of rates to the simulation and running the comparison tool we can see if this set maximises the performance indeed or if there is any other implication. Moreover, for slight augmentations the experiment tool can be run again for smaller or even tiny steps until we reach the maximum or the satisfactory, at least, level of performance.

In fact, PEPA2 can be already considered as an autonomous workbench for modelling, analysing and optimising systems although it needs and supports further development and integration of new tools.

Chapter 5

The PEPA2 GUI

5.1 Overview

The facilities of the GUI were enhanced dramatically with the integration of PEPAroni and the implementation of the additional tools. The main structure was kept and the development of the new GUI modules took place on the existing interface. We are going to use an example model, *badge.pepa*, to illustrate the PEPA2 GUI and its facilities. We will give a quick description of the functionality which existed when the project was received before continuing with the detailed analysis of the new features.

5.2 Old Features

This is a short reference to the basic functionality of the Workbench. For more information look at [7].

5.2.1 Load a model

A new model may be loaded from the *File* menu. The PEPA model will be parsed automatically. An approval message is shown if it is syntactically correct or a parser error is reported in the status box. Here, we select *badge.pepa* which will be parsed as shown in Figure 5.1.

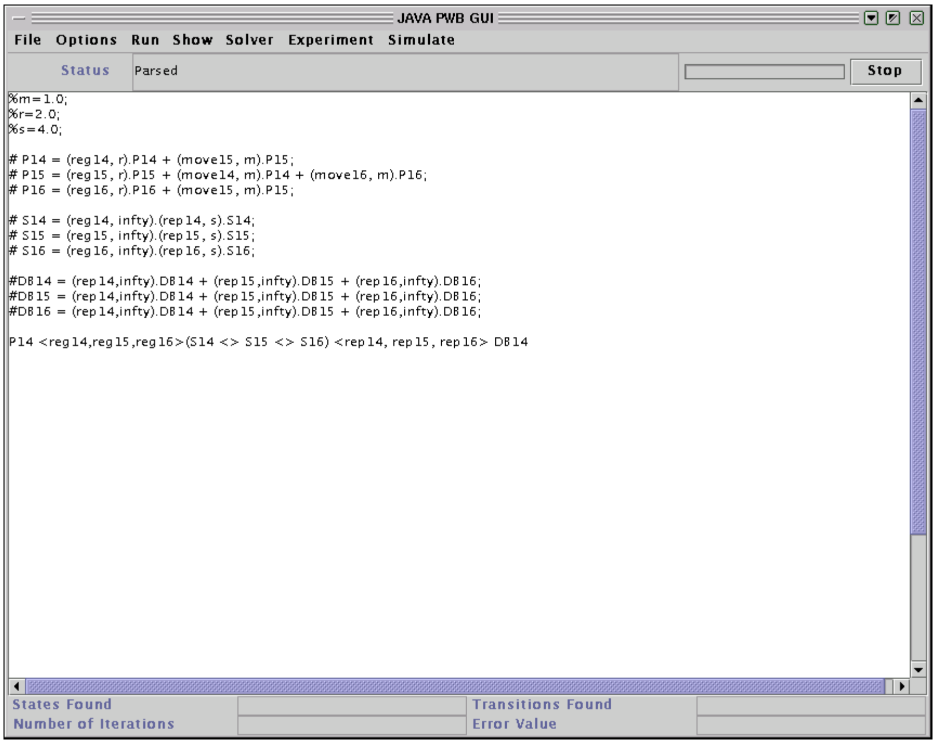

Figure 5.1: Loading a model

5.2.2 Select an option

The Options Menu Bar contains three options. If the user selects Maple or Matlab as the solution tool the Workbench outputs the transitions to file with the appropriate syntax for these high level solution packages. If the Numerical Recipes option is selected the transitions are stored in memory in row indexed format. If the modeller does not specify a solution the Numerical Recipes option is chosen by default. Here, we use the default option.

5.2.3 Run the model

The *Run* menu bar offers five alternatives to the user. There may be no state output, or the output may be to file or the central display area of the main window. The format of these may either be in the byte array representation of a process or in the concrete replacement syntax of the mathematical PEPA notation. This is the most decipherable representation of the Markov process and is recommended to novice users exploring models that do not suffer state space explosion. Large models should always be output to file in the compressed format to minimise file IO. If either of the two compression options is chosen a *model.seq* file is created containing the sequential derivatives of a model and their associated sequential state numbers.

5.2.4 Solve the model

The long-run or stationary solutions are obtained by the linear biconjugate gradient method. Not like steady-state solutions, probabilities at an arbitrary time t can be obtained by transient analysis, which was implemented by using matrix-free Krylov subspace projection methods.

The user may either execute the solution tool or change the parameters used to estimate if an acceptable level of convergence has been obtained. Novice users are advised to leave the default values as these are adequate for ensuring an acceptable level of convergence for even large models. Any exception will be reported in the status box. Here we will obtain only the steady state solution as it is going to be used later in the additional facilities demonstration. Stationary solutions will be output to file badge.steadystate. We can view this file by using the *View* bar as shown in Figure 5.2

Transient or SOR solution again can be obtained from the same menu.

5.2.5 Use of Rate, Walkabout and PSF utilities

Pepa state finder (PSF) is used to find all the possible states derived from the parsing of the model and match a given template sequence. It outputs the matching states in a *.psf* file. The tool is quite consistent as it displays a list of all possible sequential components eliminating typing errors.

The Rate utility allows the modeller to examine rate values and make changes to these values before running the Workbench (undeclared rates are assigned a zero value). Only when Maple is selected as the solution tool is it acceptable for a rate to be processed as being undefined and the transitions are encoded symbolically. That

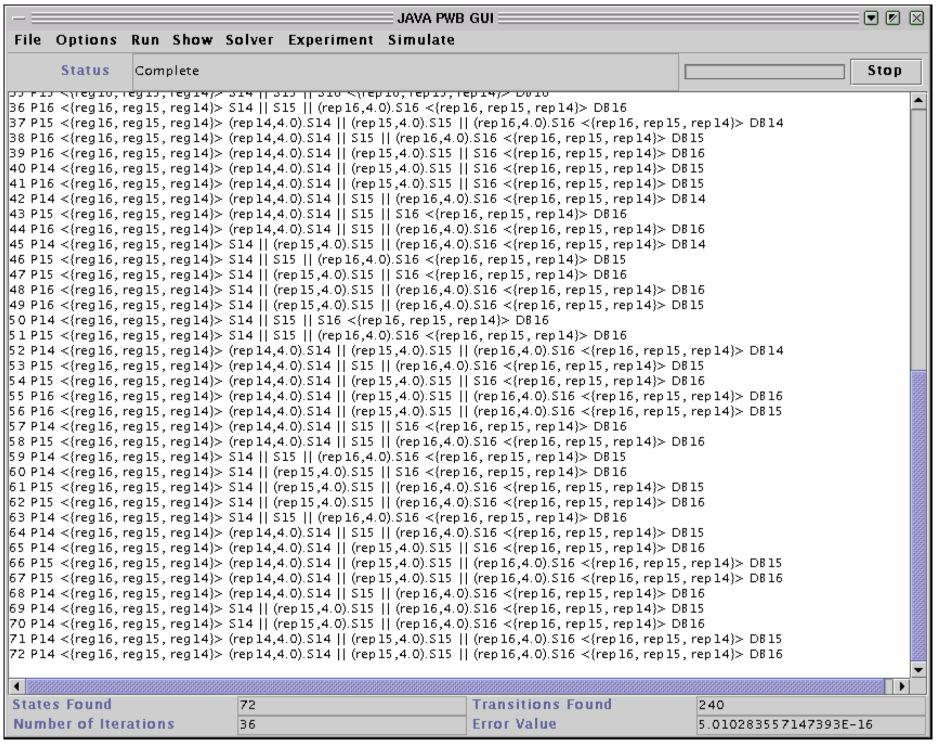

Figure 5.2: The model has been run and solved with steady state method

means that any changes to assigned rate values will not reflect always to the further performance analysis of the model.

The Walkabout facility allows the modeller to walk around the derivation graph of a model for debugging or experimentation reasons.

The above utilities offered by PEPA Workbench are maintained almost unchanged in PEPA2. The few changes are mentioned below

- The PSF utility is used by the Experimentation tool in order to select a set of states. Its functionality remained unchanged.

- The Rate utility implementation was slightly changed because there was big uncertainty whether the rate values have been changed or not. This malfunction was fixed in PEPA2, however the utility still does not change the rate assignments in the model PEPA file. So whenever the file is parsed the rates are initialised again.

5.3 Simulator

The simulation tool can be run anytime after a model has been loaded. We will demonstrate the simulation of *badge.pepa* model. Selecting from the main menu *Simulate* a drop down menu appears. Pressing *run* the simulator setup screen appears as shown in Figure 5.3.

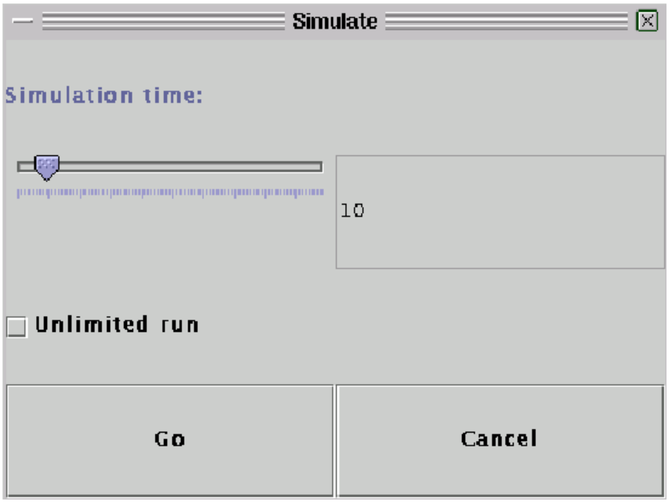

Figure 5.3: Setup screen for the simulation

There are two options: either to set a time limit or to let the simulator run untimed. In any case a *Stop* button is provided to end the simulation.

In the first case the user moves the slider to a desirable time value. The time unit of the slider is not actually a real time unit, it just represents one hundred events in the simulator. Depending on the complexity of the pepa model and the CPU speed of the system where the workbench run we can make a good estimation of the real time that the simulation is going to run. In our case every unit of the slider represents about 15 seconds simulation, so the maximum that the simulator can run through the timed option is 1500 seconds or 25 minutes (the slider maximum value is 100). This period is quite enough to allow the simulator cover the whole derivation graph of a small or medium sized model. Here we run the simulation for 10 time units (about 2.5 minutes).

For long time runs the untimed option should be used. In that case the simulator can stop only by the user (pressing the *Stop* button). This option is very useful for big models or for obtaining long term state rates. In addition, the simulation accuracy is increased when the running time is bigger. However in most of the cases even small simulation runs can give satisfactory approximations of the actual results.

Pressing *Go* the simulator starts and the setup screen disappears. If it is timed the progress bar on the main screen will be activated indicating the percentage of completion of the simulation procedure as shown in Figure 5.4. If it is untimed the progress bar will remain deactivated. In any case the border of *stop* button is blinking throughout the simulation run indicating that the utility is in use.

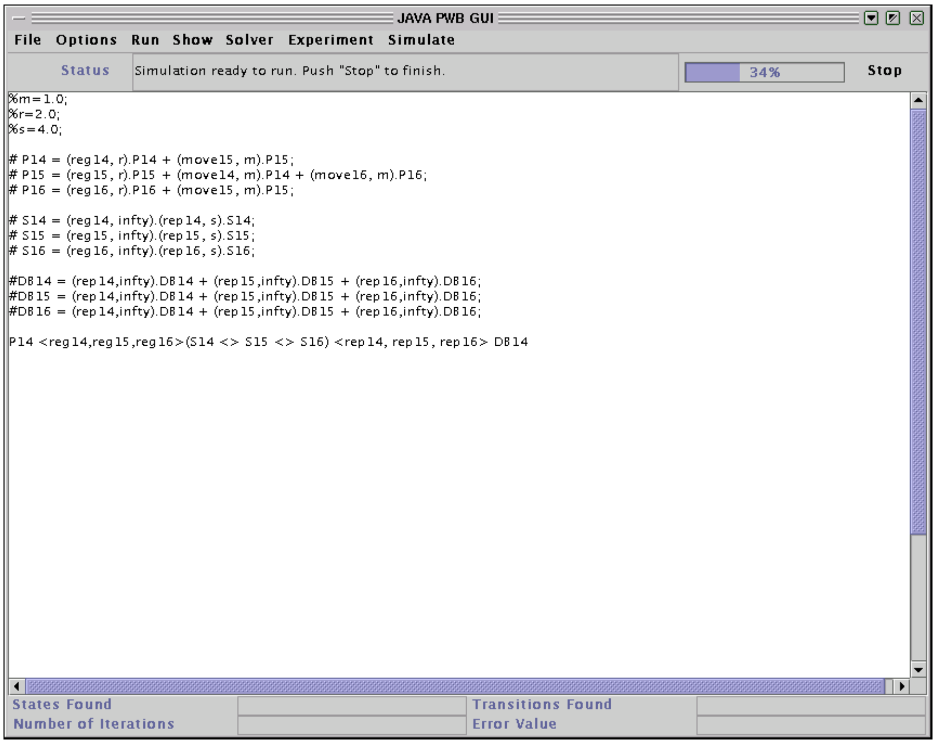

Figure 5.4: The simulation is running

The simulation ends when it reaches 100% at the progress bar or the user presses the *stop* button. The output so far has been saved in *sim.out* file. Next step is to interpret the results. This facility was served by the Perl scripts in PEPAroni, now the *stats.class* executes all the interpretation. Selecting again the *Simulate* menu and the *interpret* option the interpretation takes place. In case the simulator has run for a long time it may take a long time as well but it is much quicker than the simulation. In our case it lasted 5 seconds, that is about 3% of the simulation time. The interpretation results will appear in a new pop-up window as shown in Figure 5.5.

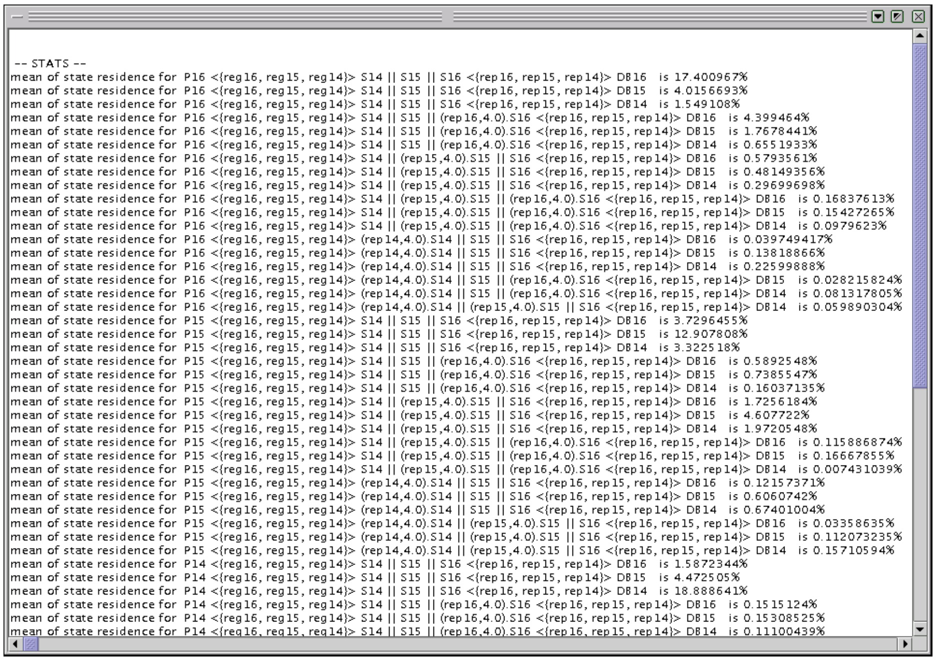

Figure 5.5: Output of simulation (interpreted)

This window contains the means of state residences that have been calculated from the simulator output and it can be kept for future reference. The statistics have also been saved in *stats.out* file.

5.4 Comparison

After the simulator has run we are able to run the comparison tool. The steady state solver should first be applied to the model so we obtain results to be compared with that from the simulator. We run the model with one of the four options, in that case *to screen*, as shown previously in Figure 5.2. The PEPA mathematical notations of 72 states appear in the main screen. Solving of the model follows, resulting in the creation of *badge.steadystate* file with the model state rates in the concrete replacement syntax of the mathematical PEPA notation.

Since the model has been solved and simulated the comparison tool can be applied. From *Simulate* menu choosing *Compare* the comparison facility is executed. This will result in two pop-up windows as shown in Figures 5.6 and 5.7. If the model has not either solved or simulated an empty window will appear indicating that some data is missing. The same will happen if the user has loaded an other model.

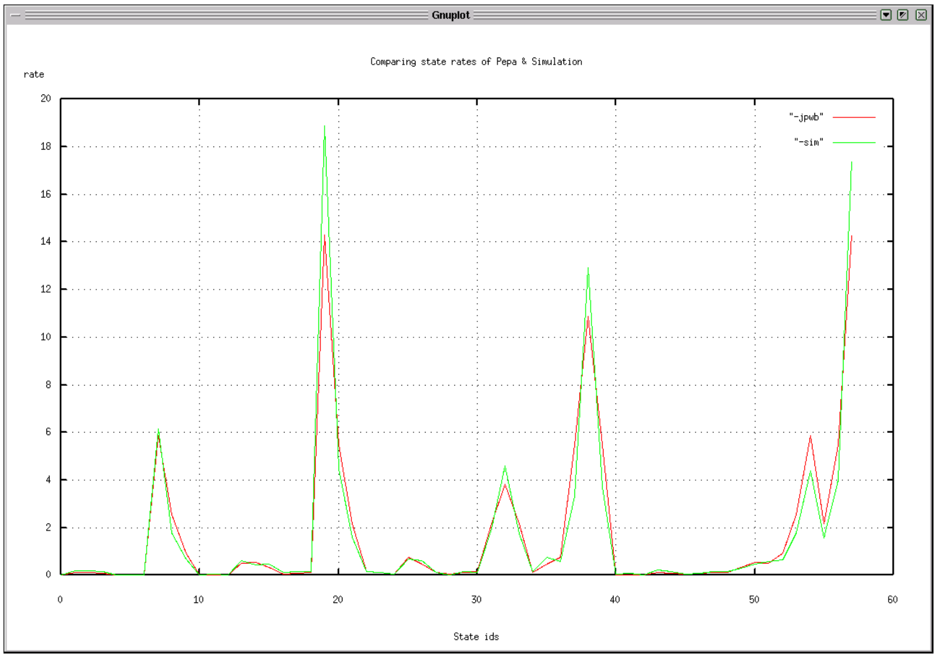

Figure 5.6: Graphical output of comparing tool

In the first window a text array is included. It contains in columns the state PEPA notations extended with the state ids, the corresponding rates calculated by the solver and the rates from the simulator. The user can evaluate the deviation of the rate values for any particular state. A visual and more general picture of the deviation between the two methods can be obtained from the second window. There the state rates are visually represented by two graphs. The distance between them indicates the deviation of the rate values.

Notice that not all the states have been included in the comparing result. Although the model has 72 states only 59 have been compared. This can be explained if we consider that the simulation has not reached all the states of the derivation graph of the model. The states that did not appear in the simulator do not have a rate value to pass

Comparing resulrs

States	jpwb rates	sim rates
0. P14 <{reg16, reg15, reg14}> (rep14,4.0).S14 || (rep15,4.0).S15 || (rep16,4.0).S16 <{rep16, rep15, rep14}> DB15	0.0016242965316 3762 %	0.017011926 %
1. P14 <{reg16, reg15, reg14}> (rep14,4.0).S14 || (rep15,4.0).S15 || S16 <{rep16, rep15, rep14}> DB14	0.130632401080014 %	0.17051363 %
2. P14 <{reg16, reg15, reg14}> (rep14,4.0).S14 || (rep15,4.0).S15 || S16 <{rep16, rep15, rep14}> DB15	0.1332791601069 7063 %	0.1924146 %
3. P14 <{reg16, reg15, reg14}> (rep14,4.0).S14 || (rep15,4.0).S15 || S16 <{rep16, rep15, rep14}> DB16	0.07869776336326527 %	0.15083463 %
4. P14 <{reg16, reg15, reg14}> (rep14,4.0).S14 || S15 || (rep16,4.0).S16 <{rep16, rep15, rep14}> DB14	0.011707910643250493 %	0.0015549916 %
5. P14 <{reg16, reg15, reg14}> (rep14,4.0).S14 || S15 || (rep16,4.0).S16 <{rep16, rep15, rep14}> DB15	0.02181939615216801 %	0.008471759 %
6. P14 <{reg16, reg15, reg14}> (rep14,4.0).S14 || S15 || (rep16,4.0).S16 <{rep16, rep15, rep14}> DB16	0.02405240216898885 %	0.0038208077 %
7. P14 <{reg16, reg15, reg14}> (rep14,4.0).S14 || S15 || S16 <{rep16, rep15, rep14}> DB14	5.871018819963305 %	6.1527286 %
8. P14 <{reg16, reg15, reg14}> (rep14,4.0).S14 || S15 || S16 <{rep16, rep15, rep14}> DB15	2.556383462433251 %	1.7618651 %
9. P14 <{reg16, reg15, reg14}> (rep14,4.0).S14 || S15 || S16 <{rep16, rep15, rep14}> DB16	0.9319462431591902 %	0.6987878 %
10. P14 <{reg16, reg15, reg14}> S14 || (rep15,4.0).S15 || (rep16,4.0).S16 <{rep16, rep15, rep14}> DB14	0.004901906268926452 %	0.027081113 %
11. P14 <{reg16, reg15, reg14}> S14 || (rep15,4.0).S15 || (rep16,4.0).S16 <{rep16, rep15, rep14}> DB15	0.010130834332279447 %	0.043677457 %
12. P14 <{reg16, reg15, reg14}> S14 || (rep15,4.0).S15 || (rep16,4.0).S16 <{rep16, rep15, rep14}> DB16	0.01490893169956406 %	0.010427771 %
13. P14 <{reg16, reg15, reg14}> S14 || (rep15,4.0).S15 || S16 <{rep16, rep15, rep14}> DB14	0.5058466805 1246 %	0.6044774 %
14. P14 <{reg16, reg15, reg14}> S14 || (rep15,4.0).S15 || S16 <{rep16, rep15, rep14}> DB15	0.5440366316538296 %	0.44429573 %
15. P14 <{reg16, reg15, reg14}> S14 || (rep15,4.0).S15 || S16 <{rep16, rep15, rep14}> DB16	0.3271794506556191 %	0.45994687 %
16. P14 <{reg16, reg15, reg14}> S14 || S15 || (rep16,4.0).S16 <{rep16, rep15, rep14}> DB14	0.05119558486078735 %	0.11100439 %
17. P14 <{reg16, reg15, reg14}> S14 || S15 || (rep16,4.0).S16 <{rep16, rep15, rep14}> DB15	0.08646536860840502 %	0.15308525 %
18. P14 <{reg16, reg15, reg14}> S14 || S15 || (rep16,4.0).S16 <{rep16, rep15, rep14}> DB16	0.1067457967266042 1 %	0.1515124 %
19. P14 <{reg16, reg15, reg14}> S14 || S15 || S16 <{rep16, rep15, rep14}> DB14	14.30393676136513 %	18.888641 %
20. P14 <{reg16, reg15, reg14}> S14 || S15 || S16 <{rep16, rep15, rep14}> DB15	5.46299456145475 %	4.472505 %
21. P14 <{reg16, reg15, reg14}> S14 || S15 || S16 <{rep16, rep15, rep14}> DB16	2.1506810608852085 %	1.5872344 %
22. P15 <{reg16, reg15, reg14}> (rep14,4.0).S14 || (rep15,4.0).S15 || S16 <{rep16, rep15, rep14}> DB14	0.16399824869520516 %	0.15710594 %
23. P15 <{reg16, reg15, reg14}> (rep14,4.0).S14 || (rep15,4.0).S15 || S16 <{rep16, rep15, rep14}> DB15	0.11143917765507355 %	0.112073235 %
24. P15 <{reg16, reg15, reg14}> (rep14,4.0).S14 || (rep15,4.0).S15 || S16 <{rep16, rep15, rep14}> DB16	0.03483214800080011 %	0.03358635 %
25. P15 <{reg16, reg15, reg14}> (rep14,4.0).S14 || S15 || S16 <{rep16, rep15, rep14}> DB14	0.7472205770862393 %	0.67401004 %
26. P15 <{reg16, reg15, reg14}> (rep14,4.0).S14 || S15 || S16 <{rep16, rep15, rep14}> DB15	0.48549089105574605 %	0.6060742 %
27. P15 <{reg16, reg15, reg14}> (rep14,4.0).S14 || S15 || S16 <{rep16, rep15, rep14}> DB16	0.12805025816786778 %	0.12157371 %
28. P15 <{reg16, reg15, reg14}> S14 || (rep15,4.0).S15 || (rep16,4.0).S16 <{rep16, rep15, rep14}> DB14	0.034832148000799396 %	0.007431039 %
29. P15 <{reg16, reg15, reg14}> S14 || (rep15,4.0).S15 || (rep16,4.0).S16 <{rep16, rep15, rep14}> DB15	0.11143917765507462 %	0.16667855 %
30. P15 <{reg16, reg15, reg14}> S14 || (rep15,4.0).S15 || (rep16,4.0).S16 <{rep16, rep15, rep14}> DB16	0.16399824869520557 %	0.115886874 %
31. P15 <{reg16, reg15, reg14}> S14 || (rep15,4.0).S15 || S16 <{rep16, rep15, rep14}> DB14	2.1704894653862152 %	1.972054 8 %
32. P15 <{reg16, reg15, reg14}> S14 || (rep15,4.0).S15 || S16 <{rep16, rep15, rep14}> DB15	3.8082564215768153 %	4.607722 %
33. P15 <{reg16, reg15, reg14}> S14 || (rep15,4.0).S15 || S16 <{rep16, rep15, rep14}> DB16	2.1704894653862103 %	1.7256184 %

Figure 5.7: Text output of comparing tool

to the comparison tool and so, they have been excluded from the comparing procedure. Actually the rate values that we get from the solver for these states are the lowest of all and tend to zero.

Analysing the comparing graph it is obvious that the simulator results are very close to the solver ones. The differences would be smaller if the simulation had run for a longer time frame.

5.5 Experimentation

An experiment can be created as soon as a model has been loaded. If the user select the *Experiment* menu and the *create* command the loaded model will be run and solved automatically before the experiment setup screen appears. This action satisfies the prerequisites for the initialisation of the experiment. This means that the rate variables, their initial values and the PEPA states are known from the system. Through the setup window the user can set the desirable experimentation variables as shown in Figure 5.8.

Firstly, the user should choose one or two rate variables to experiment on. Each of

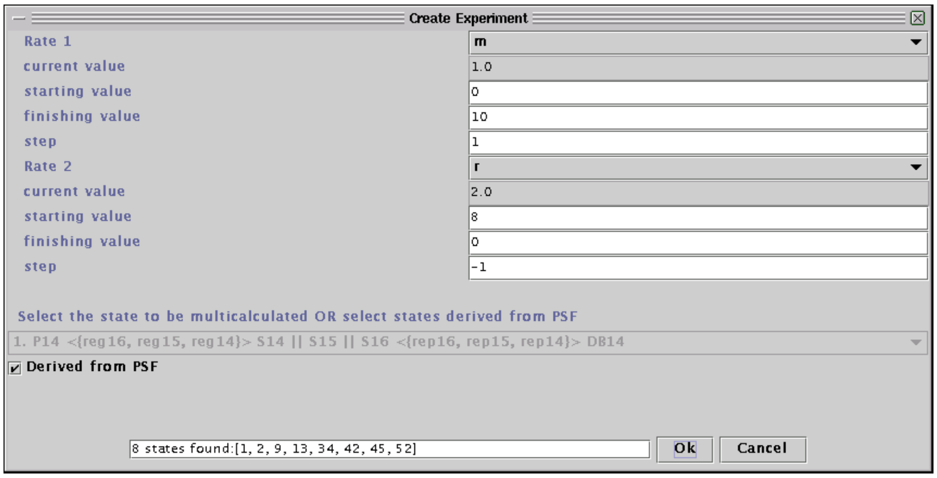

Figure 5.8: Experiment setup screen

the two drop down menus includes all the rates defined in the model. The second one has the additional option *none*. We are going to run twice the experimentation facility in order to show the two-dimensional and three-dimensional output. The first occurs when one rate variable has been chosen and the second if the experiment is run with two.

The user is required to give the range and the step of the chosen variables. In this case we set m to vary from 0 to 10 with step 1 and r to vary from 8 to 0 with step -1. This will create 80 experiment points to be plotted. Notice that the system checks the validity of the input. They should be float numbers and the step should be positive or negative according to the range boundaries. If the check fails an error message appears in the bottom text box and the experiment does not start.

Following the setting of rate variables, the states to be calculated should be selected. There are two options, either to select one individual state or a set of states derived by the PEPA State Finder tool. For the first option the user simply selects the state from a drop down menu. In our case we will choose the states derived from PEPA State Finder for the following sequential representation as shown in Figure 5.9:

$$P14 < reg16, reg15, reg14 > ** || ** || ** < rep16, rep15, rep14 > DB14$$

This will result in a set of 8 states which are going to be shown in the bottom text box of the setup screen as soon as the experiment starts running. The experiment

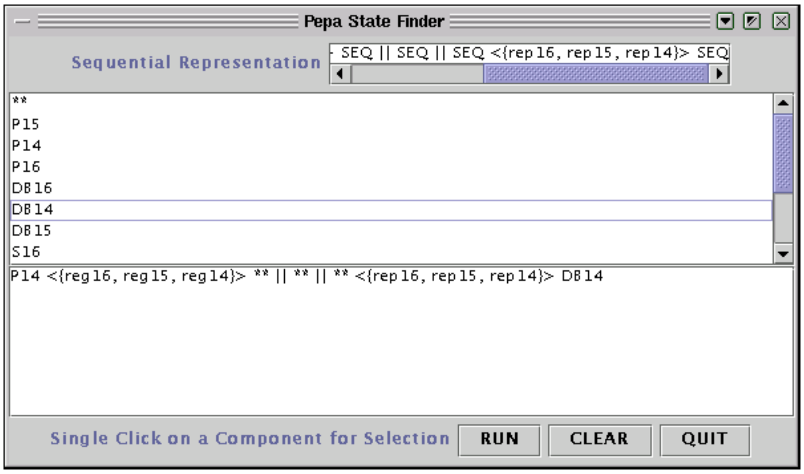

Figure 5.9: PSF for the experiment

cannot start if no states have been selected or output by the PSF tool. Instead an error message will appear.

While running, the progress bar is used again for the experimentation facility to indicate the progress of the procedure as shown in Figure 5.10. In this case the *Stop* button is deactivated.

When the experiment finish the gnuplot file is created and plotted on the screen as shown in Figure 5.11 . It contains a 3D graph showing the variance of the sum of steady rates of the selected states for the range of the rate variables.

We run the experiment again for the same set of states but with only one variable varying this time as shown in Figure 5.12. This time we assign at the second rate the value *none*. Rate s will vary from 0.1 to 10 with step 0.5. This will give us 20 experimentation points in a two-dimensional graph.

The two-dimensional graph we get as output intuitively shows that s rate cannot affect the rate of the chosen states more than a boundary value (Figure 5.13).

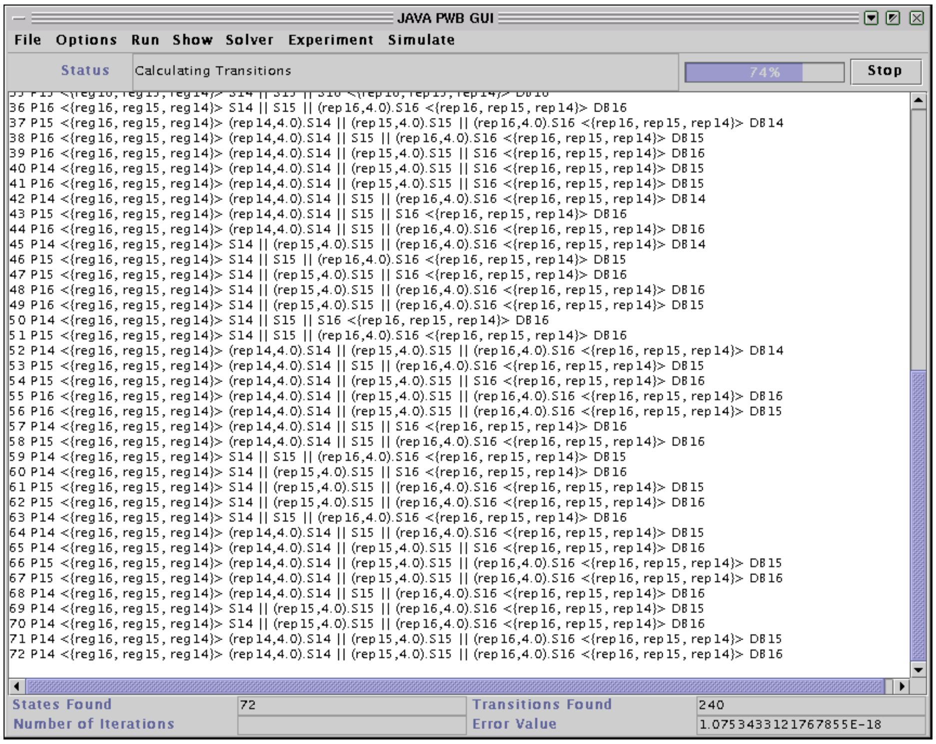

Figure 5.10: The experiment is running

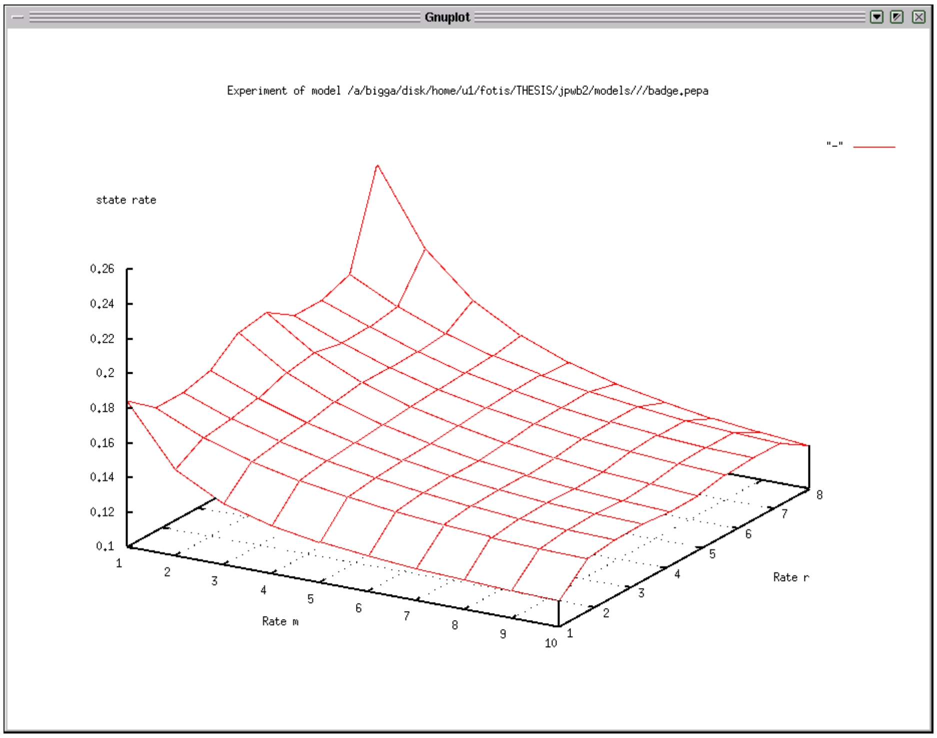

Figure 5.11: Output of 2D experiment

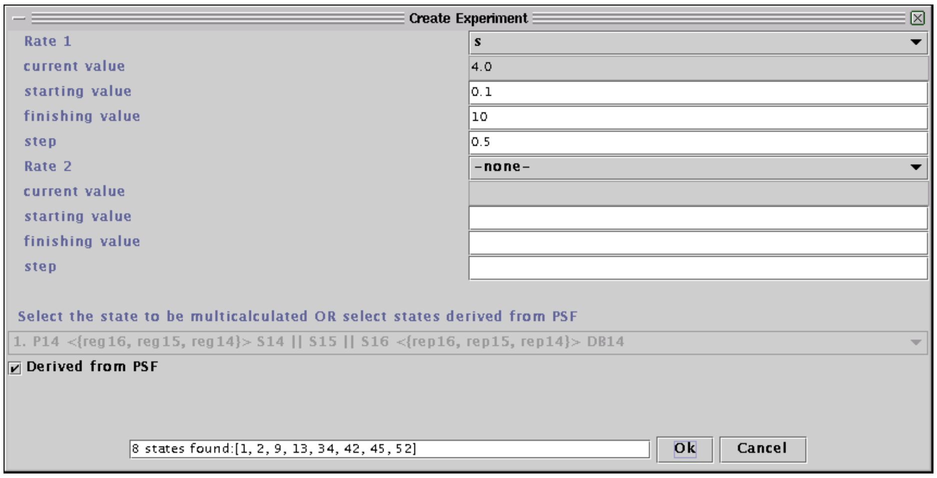

Figure 5.12: Experiment setup screen, setting one rate variable

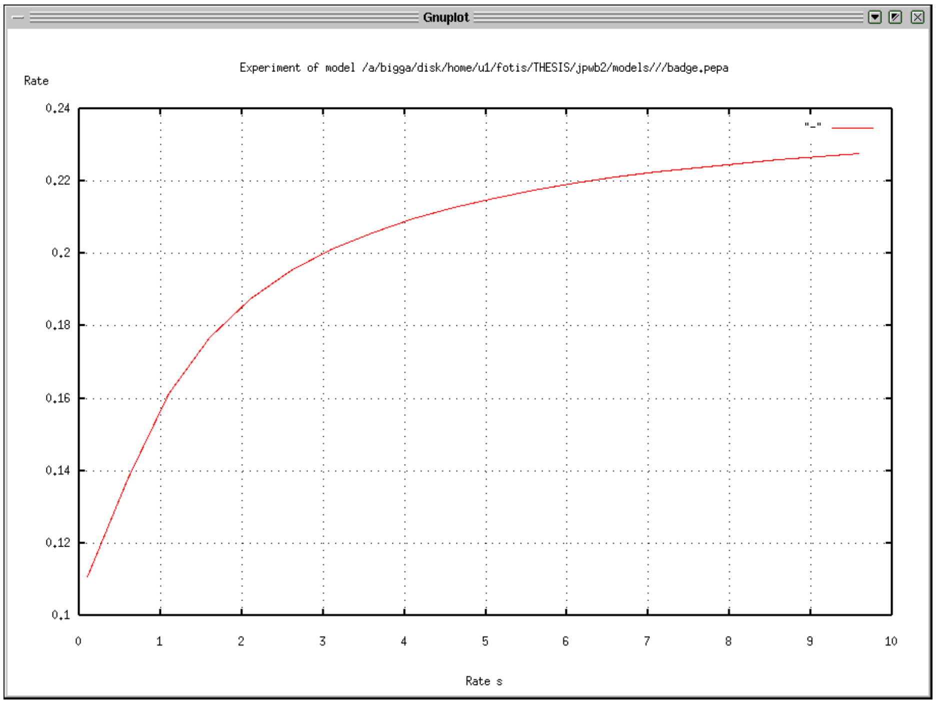

Figure 5.13: Output of 1D experiment

Chapter 6

Results and Evaluation

6.1 Overview

The new PEPA2 Workbench has even more advanced facilities than the previous versions. So, a complete evaluation of PEPA2 would include various phases and types of experimentations. We will focus the evaluation procedure on the new facilities as further examination of the system would require great effort and it is not actually a goal of this thesis.

6.2 Experimental Details

The host workstation for all the experiments was a Linux machine with a 200 MHz processor and 64 Mb of memory. Various PEPA example models were used that can be found in *http://www.dcs.ed.ac.uk/pepa/*.

6.2.1 Evaluation of integrated simulator

In order to assess the simulator in the new environment we run PEPA2 simulation facility simultaneously with PEPAroni. The experiment was repeated for various models for the same time frame (10 minutes) in different machines in order to prevent influencing the performance of each other. The outcome can be evaluated in terms of time and accuracy.

10 min. run	PEPAroni		PEPA2 simulator	
PEPA model	size(kb)	events fired	size(kb)	events fired
test.pepa (9 states)	5.200	50.000	5.000	49.000
badge.pepa (72 states)	4.250	24.800	4.200	24.000
pc-lan4.pepa (128 states)	3.700	18.500	3.600	18.000
pc-lan6.pepa (768 states)	2.700	10.500	2.500	10.000
papmodel.pepa (1557 states)	1.500	3.400	1.300	3.050

Table 6.1: Comparison of PEPA2 simulator and PEPAroni performance

- time

  The time performance of the simulator was slightly decreased after the integration. This is reasonable due to the introduction of threads in the implementation. The graphical work load that added to the simulator caused it to run 3% - 8% slower in untimed mode than the PEPAroni simulator. This is affordable although the percentage is getting much higher, about 15% - 20%, in timed mode. Naturally, the timed mode requires additional computational power which results to the reduction of the resources available to execute the simulation. Table 6.1 shows the detailed outcome of the simulation of different models with the two systems for the same time frame [1].

  Obviously the performance of the simulator depends on the complexity of the model. As we can see from the above table, the number of events in a time frame decreases rapidly as the number of states increases. The size of the output, though, depends on the length of the PEPA notation of the states. A simple model has more events per kb of output than a more complex one.

- accuracy

  The *sim.out* files of the two simulators were exactly the same until the point that both have produced output (the seed of the random number generator remained the same in both implementations). That means that the simulator functionality has not been affected by the integration. The interpretation tool has also given

[1] Workbench simulator in untimed mode

identical statistics, as expected.

6.2.2 Comparison tool

The comparison tool can compare the state rates derived by the simulator and the Workbench solver. The results of the two programs theoretically should be identical. However, we have noticed that this is tightly related to the simulation time. We run the *badge.pepa* model again for much longer time (about 40 minutes) and we get the following output diagram.

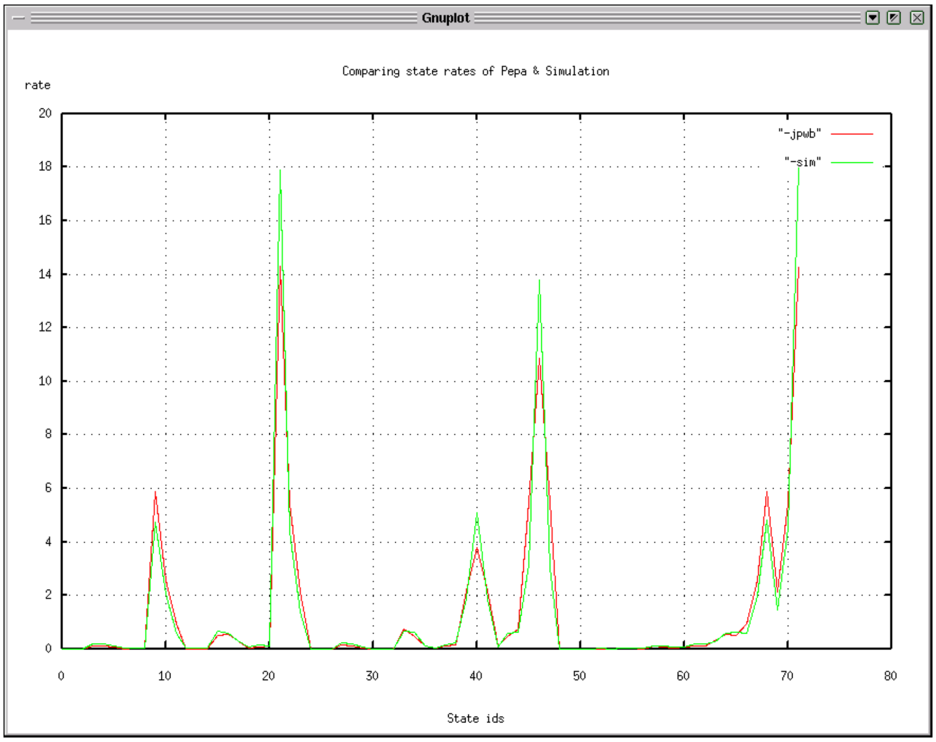

Figure 6.1: Graphical output of comparing tool for long simulator run

The diagram includes all 72 states of the model indicating that all have been appeared at least once during the simulation. Furthermore, the two lines in the diagram are closer to each other than they were in Picture 5.6. The deviation of the two lines is inversely proportional to the simulation time.

Below we give a table indicating for each model the minimum amount of time required for the simulator to access the whole state space. This time seems to be

PEPA model	states	time	events fired
test.pepa	9	5 sec	60
badge.pepa	72	35 min	90.000
pc-lan4.pepa	128	2 hours	340.000
pc-lan6.pepa	768	>20 hours	$\sim 10^7$
papmodel.pepa	1557	>40 hours	$\sim 10^9$

Table 6.2: Minimum time amounts required for the simulator to cover the state space

exponentially analogous to the size of the state space of the model. The importance of this time value is that it consists an indicator of the simulation time required to achieve an acceptable level of accuracy in the simulation outcome.

6.2.3 Experimentation Facility

The performance of the experimentation tool depends on the number of points that has been set to calculate. The implementation of the tool involves solving the model as many times as the points defined during the experiment setup. So for a matrix of 100 rate values the model will be solved 100 times and the time required for that will be $100 * t$ where t the time required to solve the model once. The performance, though, can be improved with a better implementation. Currently, the tool needs to parse the model in order to change the rate values. This obviously can be avoided but was not succeeded in this first development phase.

In terms of accuracy the tool is completely accurate as it uses the existing steady state solver of the Workbench to extract the results.

It is worth mentioning that this facility changes the rate values after the completion of the experiment. This change does not affect the simulator which reads these values from the model PEPA file. If we want to run a comparison again the rates should be adjusted to their initial values.

6.3 Further Evaluation Issues

Despite the careful implementation some of the weaknesses of PEPA workbench remained in the new version. The memory exhaustion problem while working with big models still exists but is very likely to be fixed concerning the structural form of PEPA2. The poor IO performance of Java consist a drawback to the flawless implementation but certain optimisations can be performed to minimise its effects.

Unfortunately, time did not permit extensive testing of the system with more models and under various circumstances. The complete testing of PEPA2 Workbench would be a highly time consuming procedure considering that the simulation and experimentation facilities are now a part of it. However, this first evaluation has proved the well functioning of the system and has indicated the level of confidence and power that it has.

The source code of PEPA2 consists of about 15.000 lines of code (excluding comments and external sources). The system has the GNU public license and so, it can be distributed and used freely. A copy of this document, PEPA2 system and the Java documentation is available at *http://www.dcs.ed.ac.uk/home/fotis/PEPA2*

6.4 Conclusions

The specific aims of this project were outlined in Chapter 1 and were accomplished properly. PEPAroni was efficiently integrated in the Workbench after translating the Pizza source to Java. The conversion required additional programming effort but it greatly supported the integration that followed. The new Workbench was restructured and packages of related classes were constructed. We are confident that they are functioning correctly. Moreover, a possible future upgrade of the Workbench has become an easier task. The comparison tool was implemented successfully and it was found that the results of the simulator and the Workbench solver are similar in most of the cases. The tool proved to be a valuable addendum to the performance analysis. The implementation of the experimentation tool followed, incorporating some additional useful features for the Workbench. In particular, the facility to solve a model many times over a range of rate values. This tool has significantly broaden the Workbench abilities in terms of feedback quality and speed. Graphical facilities were added to the new tools enhancing intuitively the interpretation of the output for further analysis by the user.

6.5 Future Directions

The Java Workbench has seen several revisions since its introduction and the same should be expected on this version. These will be ongoing in nature and do not warrant the tag further work. Some more conceptual ideas for further work are suggested.

6.5.1 Intelligent Experimentation Programme

The experimentation tool constructed during this project supports solving the model many times for a range of rates. It would be very useful to provide the Workbench with the recursive of the above facility. In other words, to let the system find if we can achieve a particular performance (set of state rates) and identify the rate values in that case. This tool would consist a direct performance analysis facility. PEPA2 structure provide the basis for the implementation of such tool using the existed experimentation tool.

6.5.2 Graphical Tree-form Walkabout Facility

The Walkabout facility provided by the PEPA2 Workbench currently outputs the textual representation (PEPA notation) of the states in a single walk around of the derivation graph of a model. The user is unable to observe the whole path from the beginning of execution except the previous and the next step. A tool which can graphically show the whole path would be very useful. This may be achieved using the *JTree* Java class on the Walkabout facility.

6.5.3 Handling of memory exhaustion

The Workbench has limited capabilities in terms of memory. So, very large models may fail to be solved or simulated. This boundary should be extended as far as possible to give the system the ability to handle big, commercial models. Although this might require re-implementation of the system and generally great programming effort, it will greatly broaden the horizon of the Workbench into a new and dynamic area.

Appendix A

Test Model

badge.pepa

```
%m=1.0;
%r=2.0;
%s=4.0;

# P14 = (reg14, r).P14 + (move15, m).P15;
# P15 = (reg15, r).P15 + (move14, m).P14 + (move16, m).P16;
# P16 = (reg16, r).P16 + (move15, m).P15;

# S14 = (reg14, infty).(rep14, s).S14;
# S15 = (reg15, infty).(rep15, s).S15;
# S16 = (reg16, infty).(rep16, s).S16;

#DB14 = (rep14,infty).DB14 + (rep15,infty).DB15 + (rep16,infty).DB16;
#DB15 = (rep14,infty).DB14 + (rep15,infty).DB15 + (rep16,infty).DB16;
#DB16 = (rep14,infty).DB14 + (rep15,infty).DB15 + (rep16,infty).DB16;

P14 <reg14,reg15,reg16>(S14 <> S15 <> S16) <rep14, rep15, rep16> DB14
```

Appendix B

Gnuplot file created by the comparing tool

compare.dat

```
set title "Comparing state rates of Pepa & Simulation"
set auto
set dgrid3d
set ticslevel 0
set grid
set hidden3d
set xlabel "State ids"
set ylabel "rate"
plot "-jpwb" with lines,"-sim" with lines
0 8.041831668841063E-4
1 0.00162429653163762
2 0.0023437255408156454
3 0.130632401080014
4 0.13327916010697063
5 0.07869776336326527
6 0.011707910643250493
7 0.02181939615216801
8 0.02405240216898885
9 5.871018819963305
10 2.556383462433251
11 0.9319462431591902
12 0.004901906268926452
13 0.010130834332279447
14 0.01490893169956406
15 0.50584668051246
16 0.5440366316538296
17 0.3271794506556191
18 0.05119558486078735
19 0.08646536860840502
20 0.10674579672660421
21 14.30393676136513
22 5.46299456145475
23 2.1506810608852085
24 6.505686316490241E-4
25 8.541862467147754E-4
26 6.505686316491517E-4
```

```
27 0.16399824869520516
28 0.11143917765507355
29 0.03483214800080011
30 0.0029800260676866044
31 0.004355007195366702
32 0.002980026067687058
33 0.7472205770862393
34 0.48549089105574605
35 0.12805025816786778
36 0.034832148000799396
37 0.11143917765507462
38 0.16399824869520557
39 2.1704894653862152
40 3.8082564215768153
41 2.1704894653862103
42 0.1280502581678677
43 0.4854908910557469
44 0.7472205770862379
45 5.474416181872441
46 10.880732633076606
47 5.474416181872434
48 0.002343725540830999
49 0.0016242965316359533
50 8.041831668832974E-4
51 0.014908931699564141
52 0.010130834332279487
53 0.00490190626892731
54 0.024052402168989104
55 0.021819396152175066
56 0.011707910643251614
57 0.10674579672660402
58 0.08646536860840477
59 0.051195584860791093
60 0.07869776336327013
61 0.13327916010697066
62 0.13063240108001417
63 0.3271794506556143
64 0.5440366316538296
65 0.505846680512462
66 0.9319462431591902
67 2.55638346243325
68 5.871018819963295
69 2.1506810608852125
70 5.462994561454743
71 14.303936761365124
e
0 0.0018431299
1 0.002809793
2 0.0036162867
3 0.21300934
4 0.18664639
5 0.1299199
6 0.026044909
7 0.039621282
8 0.041253544
9 4.7227507
10 2.0345051
11 0.60069895
12 0.017831882
13 0.021909412
14 0.023874609
15 0.67178744
16 0.57456577
17 0.3325884
```

```
18 0.08360875
19 0.13900995
20 0.10326726
21 17.899105
22 4.3993998
23 1.3668036
24 0.0013462182
25 6.5190956E-4
26 0.0028121236
27 0.23449618
28 0.2110472
29 0.061533734
30 0.00827857
31 0.015873495
32 0.0048552766
33 0.6684587
34 0.61619896
35 0.11177736
36 0.0491439
37 0.16957766
38 0.2204442
39 1.8963927
40 5.110731
41 1.8440758
42 0.09833966
43 0.5987102
44 0.63786185
45 3.1687279
46 13.789974
47 3.0396132
48 0.0041042333
49 0.004279783
50 0.003758882
51 0.014960774
52 0.020064691
53 0.009614775
54 0.033470634
55 0.04202175
56 0.018588554
57 0.13267612
58 0.10820806
59 0.09678818
60 0.09390054
61 0.20791836
62 0.1823376
63 0.2864083
64 0.5753026
65 0.6366971
66 0.6008275
67 1.9717486
68 4.8148518
69 1.4441935
70 4.4861565
71 18.013765
e
```

Appendix C

Gnuplot file created by the experimentation tool

experiment.dat

```
set title "Experiment of model /a/bigga/disk/home/u1/fotis/THESIS/jpwb2/models//badge.pepa"
set auto
set dgrid3d
set ticslevel 0
set grid
set hidden3d
set xlabel "Rate m"
set ylabel "Rate r"
set zlabel "state rate"
splot "-" with lines
1.0 1.0 0.18406126848414142
1.0 2.0 0.20880044247860755
1.0 3.0 0.2210790312205727
1.0 4.0 0.22828555576351556
1.0 5.0 0.2329419752972991
1.0 6.0 0.23614618210174265
1.0 7.0 0.23845200588776794
1.0 8.0 0.2401681007672821
2.0 1.0 0.14986323618829878
2.0 2.0 0.16833338984493965
2.0 3.0 0.17886019933837338
2.0 4.0 0.1855038285732072
2.0 5.0 0.18998361048306642
2.0 6.0 0.19314842628998063
2.0 7.0 0.1954632184679313
2.0 8.0 0.1972024581045488
3.0 1.0 0.1354264527256431
3.0 2.0 0.14888339510498372
3.0 3.0 0.15717956896774557
3.0 4.0 0.162664487016896
3.0 5.0 0.1664731112902291
3.0 6.0 0.16921571118409656
3.0 7.0 0.1712468441327382
3.0 8.0 0.17278493568255693
4.0 1.0 0.12785126608049974
4.0 2.0 0.1379113649190345
4.0 3.0 0.14442641071188075
```

```
4.0 4.0 0.14887190938151032
4.0 5.0 0.15202461687533517
4.0 6.0 0.15432779046372608
4.0 7.0 0.15605026478384415
4.0 8.0 0.15736305697344644
5.0 1.0 0.12335396012958236
5.0 2.0 0.13109319735088448
5.0 3.0 0.13627089433732842
5.0 4.0 0.13988202733013716
5.0 5.0 0.14248203066889184
5.0 6.0 0.14440156119891917
5.0 7.0 0.14584758327005834
5.0 8.0 0.14695496719997395
6.0 1.0 0.12045708839501652
6.0 2.0 0.12656328657282742
6.0 3.0 0.1307397720474564
6.0 4.0 0.13369740775387515
6.0 5.0 0.13584973530611313
6.0 6.0 0.13745058540500646
6.0 7.0 0.138662575529093
6.0 8.0 0.13959361105726617
7.0 1.0 0.11847873357601697
7.0 2.0 0.12340062642750725
7.0 3.0 0.12681863227352344
7.0 4.0 0.12926479306976704
7.0 5.0 0.13105792222379845
7.0 6.0 0.1323981303797454
7.0 7.0 0.13341585006753548
7.0 8.0 0.13419881140768064
8.0 1.0 0.11706655933174821
8.0 2.0 0.12110592919629896
8.0 3.0 0.12394032594474087
8.0 4.0 0.12598319035033648
8.0 5.0 0.1274876601308019
8.0 6.0 0.12861525902414642
8.0 7.0 0.12947262538289467
8.0 8.0 0.13013221431919833
9.0 1.0 0.11602290199029225
9.0 2.0 0.11938889654037765
9.0 3.0 0.12176707144755075
9.0 4.0 0.1234886872262007
9.0 5.0 0.1247597911457943
9.0 6.0 0.1257134643005712
9.0 7.0 0.1264384033198587
9.0 8.0 0.12699532524040594
10.0 1.0 0.11522964586724767
10.0 2.0 0.11807122906865296
10.0 3.0 0.12008739716921579
10.0 4.0 0.12155033442537347
10.0 5.0 0.1226313143042026
10.0 6.0 0.12344194930771502
10.0 7.0 0.1240571456864717
10.0 8.0 0.12452845992820838
```

Bibliography

[1] Stephen Gilmore and Jane Hillston. The PEPA workbench: A tool to support a process algebra-based approach to performance modelling. In *Computer Performance Evaluation*, pages 353–368, 1994.

[2] J. Hillston. A compositional approach to performance modelling, 1994.

[3] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[4] G. Clark, S. Gilmore, J. Hillston, and N. Thomas. Experiences with the pepa performance modelling tools, 1999.

[5] G. Clark, S. Gilmore, and J. Hillston. The PEPA performance modelling tools. In *Proceedings of the Seventh Annual Workshop on Process Algebra and Performance Modelling*.

[6] Graham Clark. The pepa performance modelling tools.

[7] Feng Wan. Interface engineering and transient analysis for the pepa workbench, 2000.

[8] J. Hunter. Re-evaluation of the pepa workbench, 1999.

[9] G. Clark. Techniques for the construction and analysis of algebraic performance models, 1999.

[10] Harvey M. Deitel. Perl : how to program, 1998.

[11] S. Donatelli, J. Hillston, and M. Ribaudo. A comparison of Performance Evaluation Process Algebra and Generalized Stochastic Petri Nets. In *Proc. 6th International Workshop on Petri Nets and Performance Models*, Durham, North Carolina, 1995.

[12] Alan Dix, Janet Finlay, Gregory Abowd, and Russel Beale. *Human Computer Interaction*. Prentice Hall, 2nd edition, 1997.

[13] Bruce Eckel. Thinking in patterns with java. Unpublished, curently available at http://www.mindview.net/Books/TIPatterns/.