University of Edinburgh
Division of Informatics

Re-evaluation of the PEPA Workbench

M.Sc. Project Report

Jonathan Hunter

September 24, 1999

Abstract

The Performance Evaluation Process Algebra[3] (PEPA) approach to modelling is supported by a tool called the PEPA Workbench[2] implemented in the ML programming language. This thesis describes the re-evaluation of the PEPA workbench in Java[4]. The motivation for the project was to produce a mobile implementation, capable of completing the derivation graph of the underlying Markov process and calculating the associated steady state probability distribution. A prototype has been produced with incorporates an easy to use GUI to simplify the modelling task. A novel feature of the tool is a facility which enables the modeller to traverse the derivation graph of the model at will. This is expected to be of great assistance in the development and debugging of new models.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Stephen Gilmore for his excellent supervision throughout the project, and Graham Clark for his infinite patience in explaining the finer points of the PEPA modelling language.

Chapter 1

Introduction

1.1 Overview

Computer systems can be described using formal languages which provide a means of constructing an abstract model of the system in question. The model is used to establish the correct functional behaviour of complex systems by deriving qualitative properties such as freedom from deadlock or livelock. Performance Evaluation Process Algebra (PEPA) is a stochastic process algebra and is an enhancement of the classical process algebras via the inclusion of timing information in the generated models. This allows the modeller to determine performance metrics of a system in addition to its functional behaviour.

The objective of the modelling study is to generate the steady state probability distribution for use as a basis in the calculation of performance measures. A tool called the PEPA Workbench has previously been developed in the ML[6] programming language to support this approach to performance modelling. However, the ML implementation is limited by its inability to solve a model without interfacing with a linear algebra package and by its lack of platform mobility in comparison with a mobile code such as Java.

1.2 Project Aims

When the project was originally conceived it was thought that the Workbench should be reimplemented in Java. The reasons for selecting Java as the implementation medium were:

- Java is able to perform the iterative calculations on array elements that linear algebra entails;
- Java was designed with the intention of producing a safe and secure mobile language suitable for distribution over networks and capable of running on any platform for which a Java interpreter is available. The reimplementation of the Workbench in Java increases its portability;
- Use of Java allows access to the extensive graphical capabilities that the class libraries of the language provide. The facilities of the Abstract Window Toolkit (AWT) are used in creation of a GUI to act as the front end to the Workbench.

More specifically the objectives of the project were to:

- Implement a linear algebra package as a back end to the Workbench to create an integrated software package;

- Rewrite the implementation in a mobile language to give greater platform mobility
- Include an easy to use GUI as a front end to the Workbench
- To carry out the above as efficiently as possible in terms of memory usage to enable the analysis of large PEPA models.

These were the core objectives of the project. Time allowing it was hoped to extend the facilities offered by the GUI. One such example would be the provision of a walkabout function allowing a modeller to step back through the states of a model. This would be especially useful in assisting with the debugging of models with deadlocks, as it facilitates the rapid diagnosis of the entry state or states that lead to the deadlocked state. No such facility is provided by the ML implementation.

1.3 Design and Implementation

The design of the Java version of the Workbench did not begin with a clean sheet, the ML version already existed and its source code was available for analysis. However, the greatest influence on the new Workbench was not the ML code but the applicative properties of the ML language and the Discrete Event Simulator[1](DES) software package, one of the PEPA performance modelling tools. The DES was coded in Pizza[5], a superset of Java, because the language offered the algebraic data types and polymorphism associated with functional programming languages and the use of the extensive Java class libraries and software utilities, in particular Simjava (Simjava is a toolkit for the building working models of complex systems).

Implementing the Workbench in Pizza allowed the reuse of some of the methods and class structures found in the DES, but more significantly, it permitted the reuse of the parser built for this application. The reasons stated for selecting Java in 1.2 apply equally well to Pizza - Pizza is defined by translation into Java. In light of the close relationship between the two languages the Pizza Workbench will be referred to as the Java Workbench throughout this report.

An alternative to the use of the high-level numerical computation packages such as Matlab and Maple is the collection of routines written in the C programming language by Numerical Recipes[7]. The format required to interface with this package is more subtle, as the matrices are encoded in a row indexed sparse storage mode. Output from the ML Workbench is not directly compatible with this form of matrix representation so was not supported by the exported version. The routines supplied by Numerical Recipes are simply a collection of common matrix operations suited to the manipulation of sparse matrices. These operations are ordered and performed iteratively to implement the *Linear Biconjugate Gradient Method*, a more powerful alternative to Gaussian elimination in solving matrix equations.

The GUI was implemented using the facilities of the AWT class library. The GUI acts as the driver for the Workbench code, the user setting parameters that dictate exactly which solution tool is to be used, what performance metrics are to be analysed and what, if any, output is required.

1.4 Outcome

The state space for even modest PEPA descriptions can be large and can cause any tool engineered to compute the transition graph to fail as a result of memory exhaustion. State-space explosion is an inherent disadvantage of any Markovian modelling formalism, and PEPA is no exception. The greatest problem in producing the new Workbench was the minimisation of memory usage to allow as large a model as possible to be analysed. Attempts to increase memory efficiency focused on compressing the representation of the state space of a model. A technique was developed to systematically reduce a state to a `byte` array representation and build a state from its `byte` array representation. This storage mode was instrumental in expanding the range of facilities offered via the GUI.

The Workbench was implemented in the Pizza language with all the stated objectives being met. Testing of the new PWB has shown complete conformance on the number of states and transitions found in comparison with the same models tested on the ML version. This gives confidence in its correct behaviour. Some of the facilities available to the modeller in the Java PWB are not directly available in the ML version and result from the integration of the Workbench into a stand alone package and the compression of the state space. The Java PWB will be added to the suite of performance tools available to the PEPA user community.

Chapter 2

The PEPA Modelling Language

2.1 Overview

The development of modern communication and computer systems is increasingly complex. Traditional process algebras such as CCS and CSP can be used to construct an abstract model of a system to establish its correct functional behaviour. PEPA is an enhancement of classical process algebras in the inclusion of a timing attribute. Probabilistic branching replaces the nondeterministic branching of CCS and CSP, introducing the concept of temporal and spatial uncertainty. This specifies PEPA as a stochastic process algebra capable of supporting a performance evaluation approach to modelling.

2.2 PEPA Terminology

A proposed or actual system is analysed to identify substructures which may have a quantifiable effect on the behaviour of a system taken as a whole. In PEPA these substructures are termed components and it is their composite behaviour that abstracts the system's behaviour. Components may be atomic or may themselves be subcomponents of a larger component. Each component has behaviour which is defined by the activities in which it can engage. Each activity has an action type and an associated duration.

It is assumed that each discrete action within a system has a unique type and there is a countable set A of all possible such types. The action types of a PEPA term correspond to the actions of the system being modelled and may act as a qualatitive description of the systems behaviour. The special action type τ is a reserved word in the PEPA language and is used in situations where a component is carrying out an action that is unknown or is unimportant. Components are not able to synchronise on the silent action type as they are considered private to the component in which they occur.

The duration of an action is a positive real number which is a parameter of the exponential distribution function. The parameter determines the rate of an activity and it is the random nature of the exponential distribution function that specifies PEPA as a stochastic process algebra. The duration may be represented by the $\top$ symbol, indicating that a component cannot engage in an activity in isolation and must synchronise with another component on the common action to perform.

When enabled an activity $a = (\alpha, r)$, will delay for a period determined by its associated distribution, denoted $F(t) = 1 - e^{-t}$. We can think of this as a timer being coupled with every activity and on every occurrence of an activity being enabled this timer is set. If several activities are enabled at once the activity associated with the first timer to complete is assumed to succeed. At the system level the event associated with this activity is assumed to happen. All other activities are aborted.

2.3 The Syntax and Semantics of PEPA

The behaviour of a component is characterised by its activities. A system composed of several components which perform only local processing is unrealistic. PEPA provides a small but powerful set of combinators which allow expressions or terms to be constructed defining the activities which components may undertake and the interactions between them.

The syntax of these terms is defined below and a brief description of each follows.

$$P ::= (\alpha, r).P \mid P + Q \mid Q \underset{L}{\bowtie} R \mid P/L \mid A$$

Prefix $(\alpha, r).P$

This sequential composition is the basic mechanism by which the behaviour of components are constructed. The example given states that the component engages in the activity (α, r). and then behaves as P again. The action type of this activity is α and must be a member of the countable set of discrete actions in the system.The variable r is the exponential distribution parameter that governs the rate at which the action occurs. It may be the case that two or more components cooperate on an action type and one of them may have an unspecified or passive rate. This is represented by the reserved word $\top$ (named Top) and indicates that a component plays no part in determining the duration of the activity.

Choice $P + Q$

The component $P + Q$ represents a system that may behave as either P or Q. Probabilistic branching is determined by enabling all the current activities of P and Q, and selecting the component associated with the activity that completes first. The continuous nature of the probability distribution ensures that a distinct choice is made and the other component is discarded. The system will subsequently behave as P' or Q' respectively, where P' is the component which results from P completing first and Q' is the component which results from Q completing first. The choice operator signifies that components are competing for an assumed underlying resource of the system.

Cooperation $P \underset{L}{\bowtie} Q$

The Cooperation symbol indicates that components must interact on any mutual action type specified in the action set, L, of the combinator. The outcome of such an interaction is termed a shared activity. This has the same action type as the contributing activities and a rate reflecting the rate of the slower participant. In general both components will need to do some work to complete a shared activity, except when one of the activities has an unspecified rate $\top$. This indicates that the cooperation of a component may be required to complete a task but it does no actual work and is akin to a master-slave relationship.

All activities of P and Q which have types which do not occur in L will proceed unaffected and are termed individual activities. This can lead to blocking as one component waits for the other to be ready to participate. The cooperation set, L, can be empty and this implies a pure parallel arrangement without any cooperation between the components. This is usually denoted by the $||$ symbol.

Hiding P/L

The actual behaviour of a component is the same but any action contained within the action set is hidden. These activities can be regarded as unknown (τ) and are assumed to represent the occurrence of internal processing within the component. Cooperation on a hidden action type is not permitted.

Constants $P \stackrel{\text{def}}{=} Q$

Constants allow names to be assigned to components and give the variable on the left hand side of the assignment operator the behaviour of the component on the right hand side.

2.4 Apparent Rates

When two or more enabled activities are competing for the same underlying resource, the race condition ensures that the rate at which an activity α is performed is the sum of the exponential distribution parameters. It may be the case that the enabled activities are described by the same action type. To an external observer of the system the apparent rate of activities of that type will be the same.

2.5 The Underlying Markov Process

2.5.1 Transition Diagram

Application of the operational semantics of the PEPA language allow the behaviour of a given system to be deduced automatically from its language description. The evolution of a process can be thought of in terms of states and transitions. In PEPA states are the syntactic terms of the language and transitions describe the action type and rate at which a change of state occurs. Those seeking a formal description of this notion should consult [3].

A transition diagram called the derivation graph can be associated with every language expression. The language terms form the nodes in the graph and the arcs represent the possible transitions. Qualatitive properties can be deduced from the graph. Nodes from which there is no possible exit transition represent deadlocks in the model and a subset of the nodes in the graph from which no transition is possible indicates a livelock. To form the underlying stochastic (in fact Markov) process each node in the graph is taken to be a state of the stochastic process and the transition rate between states is sum of the rates on the arcs connecting nodes.

2.5.2 Generator Matrices

A Markov process with n states can also be characterised by its $n \times n$ *infinitesimal generator matrix* $\mathbf{Q}$. The entry in the jth column and ith row of the matrix ($j \neq i$) will be q_{ij} where q_{ij} is defined as the instantaneous transition rate between states j and j. In other words it is the sum of the rates labelling the arcs in the derivation graph. The diagonal entries in each row contain the negated sum of the entries in that row. This is a necessary condition for calculation of the steady state probability distribution of the Markov process.

2.5.3 Solving the Markov Process

Steady state analysis is a prerequisite for the calculation of some performance measures, including throughput and utilisation. The conditions for the existence of a steady state probability distribution over the state space S of a Markov process are well known. A stationary or equilibrium probability distribution, Π, exists for every time homogeneous irreducible Markov process whose states are all positive recurrent.

PEPA models are intrinsically time-homogeneous. The activities enabled by components are not temporally dependent. Only models with a bounded number of components are considered thus they are finite. Irreducible implies that all states can be reached from all other states as the process evolves and recurrent implies that the probability of a state being revisited is equal to one.

For finite state PEPA models whose derivation graph is strongly connected, the equilibrium distribution of the model, Π, is found by solving the matrix equation.

$$\Pi\mathbf{Q} = 0$$

where Q is the infinitesimal generator matrix. This collection of equations is not in fact irreducible. Some manipulation of the matrices is necessary. The generator matrix must be transposed to comply with the format required by all the linear algebra packages and one row in the transposed matrix must be replaced by a row of ones to reflect the so-called normalisation condition. The normalisation condition states that the probability of the process being in some state within the state space at any instance in time is one. Finally the solution vector, initially a column vector of zero entries, is replaced by a column vector with a one in the final column, denoted $\mathbf{e}_n$. The resulting matrix equation can be expressed:

$$\mathbf{Q}_n^T \pi = \mathbf{e}_n$$

This equation is in a format suitable for solution by any linear algebra package.

2.6 Performance Metrics

Solving the Markov process is a necessary condition for the calculation of throughput and utilisation. The utilisation of a subset of states is the probability that the model is in that subset of states. Throughput is the product of the rate of an event and the probability that the event is enabled.

Chapter 3

Design Influences

3.1 Overview

The Java PWB was never intended to be a line by line translation of the ML code. Certain aspects of the ML language merited imitation and resulted in the implementation being coded in Pizza, a superset of Java. Another member of the family of PEPA modelling tools, the Discrete Event Simulator, was developed in Pizza and this allowed the reuse of some code. Of greater significance, however, was the reuse of the parser written for this application as this gave a foundation to work from.

3.2 ML

The linguistic constructions of ML have a larger bearing in the design of the Java PWB than the source code itself. Central to the operation of the Workbench is the function that takes a state and returns all the one-step derivatives. A structure is passed to this function and the function then looks into the structure and makes the appropriate binding. This is termed *pattern matching*. The structure in this example is the representation of a component as a datatype. A language called Pizza which is a superset of Java incorporates this feature. Pizza is discussed next.

3.3 Pizza

When the project was originally conceived it was intended that the Workbench be re-evaluated in Java. The existence of Pizza significantly simplifies the implementation. Pizza extends Java with several features, the most useful of which are the algebraic data types and parametric polymorphism offered by the language. Moreover, Pizza is defined by translation into Java; this extra functionality does not compromise the previously stated reasons for choosing Java as the implementation medium.

3.3.1 Algebraic Data Types in Pizza

Object types and inheritance are complementary to algebraic types and pattern 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 data types and pattern matching make it easy to add new operations over the type, so long as the set of constructors is relatively 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.

```
abstract 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)
      ...
   }
```

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.

```
method illustration()
  {
     switch (this)
        {
        case Prefix(_):
        ...
        case Sum(_):
        ...
        ...
        }
  }
```

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 the Workbench code.

3.3.2 Parametric Polymorphism in Pizza

Polymorphism provides a general approach to describing data or algorithms where the structure is independent of the type of data values manipulated. The ML Workbench makes extensive use of the polymorphic list structure that the language provides.

Java does not support polymorphism but Pizza simulates polymorphism (as in fact do Java arrays). This is achieved by replacing the parameterised data type by the class Object. The key to its functioning is the hierarchical class structure of the Java language. A value of any type can be converted to an object and later recovered. Every type in Java is a reference type or one of the eight basic types such as `int`. For the primitive types, wrapper objects can be used to convert their values into a `java.lang.Object`

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 into an object `o` by `(Object)(new Integer(v))`, and converted back by `((Integer)o).intValue()`. In Java widening may be implicit but the cast is written for the purpose of clarity.

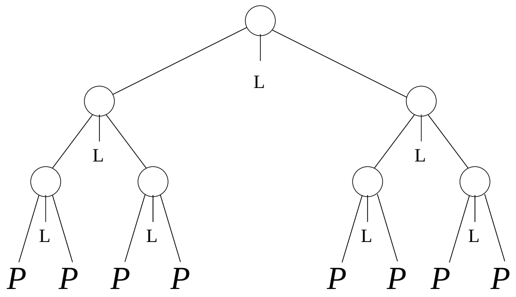

Figure 3.1: The Tree Representation of a Process

In Java, type information is stored implicitly with every object. Java performs run time checking on object coercion by matching object types. In ML rigorous type checking is carried out at compile time. The translation of all parametrised data types in Pizza ensures that run time type coercion is secure, subject to the constraint that the Pizza compiler accepts all the operations on a parametrised data type at compile time.

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 the Workbench and this gives greater run time security of the code.

3.4 The Discrete Event Simulator

A recent theme of stochastic process algebra research has been the introduction of generally distributed activities. The discrete event simulator, one of the performance modelling tools available to the PEPA user community, was developed to investigate situations in which general distributions may successfully replace exponential distributions without affecting the model's steady state solution. The discrete event simulator is the first PEPA performance modelling tool to be implemented in a language other than ML, the chosen language being Pizza.

3.4.1 The Parser

The parser used by the DES was implemented using the Yacc-like CUP parser generator. The mathematical notation of PEPA is not available in an ASCII character set and so, as with the ML Parser, a concrete replacement syntax was devised. Modifications to the parser were limited to fine tuning and the removal of the functionality that allows the exponential distributions of PEPA to be replaced by general distributions.

The parser accepts process definitions and follows the semantic rules of the PEPA language to build a Java representation of a PEPA process. This takes the form of a process tree and it is a reference to the highest level cooperation in the tree that is returned on initialisation of the parser. The tree representation of the process shown below is illustrated in Figure 3.1.

$$((P \underset{L}{\bowtie} P) \underset{L}{\bowtie} (P \underset{L}{\bowtie} P)) \underset{L}{\bowtie} ((P \underset{L}{\bowtie} P) \underset{L}{\bowtie} (P \underset{L}{\bowtie} P))$$

The definition of the component and the members of the action set L are superfluous and are not defined here. Each node in the tree represents the cooperation paradigm and the leaves represent the atomic components. The parenthesis dictate the scope of the cooperation operator and affect the shape of the tree (The PEPA parser associates to the left). Use of the legacy parser obliges all methods in the Java Workbench to work with this form of process representation. The switch operator provided by Pizza is paramount to the efficient recursive traversal of the tree.

3.4.2 Class Structure and Methods

The skeletal class structure of the discrete event simulator 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. A full discussion of the class structure and implementation as used by the Java PWB is deferred until Chapter 5.

3.5 Memory Considerations

In generating the state space of a model there is always the possibility of exhausting the memory capacity of the workstation on which it is running. It would be pointless to create a tool that runs extremely rapidly but is incapable of completing the derivation graph for even modestly sized PEPA descriptions. The design of the Workbench is geared towards the conservation of memory resources often at the expense of the run time.

This problem is accentuated by the need to store the transitions in addition to the state space. Unlike C or C++, Java does not provide a `sizeof` operator (its designers considered this operator superfluous as pointer arithmetic is not possible in Java) so the techniques for estimating memory consumption are unfortunately left to intuition and an estimate of the efficient operation of the Java garbage collector. It is not just memory conservation that is an issue. Unallocated memory must be contiguous, space must be sufficient for the large arrays required by the sparse representation of the generator matrix and for the vectors used in the calculation of the steady state solution. Memory conservation is a recurring theme throughout this report.

Chapter 4

Sparse Storage and Linear Algebra

4.1 Overview

The exported version of the PEPA Workbench can interface with the Maple, Mathematica or Matlab linear algebra packages to produce the steady state solution. The use of these packages is particularly suited to less experienced modellers as they provide an easily interpreted representation of the matrices and vectors which are essential to store the Markov process encoding of the user's model.

An alternative to these packages is a collection of routines written in the C programming language by Numerical Recipes. The output required to interface with this package is more subtle, as the matrices are encoded in a row indexed sparse storage mode. Output from the Workbench is not directly compatible with this form of matrix representation so was not supported by the exported ML version.

The routines supplied by Numerical Recipes are simply a collection of common matrix operations suited to the manipulation of sparse matrices. These operations are ordered to implement the linear biconjugate gradient method which is a more powerful alternative to Gaussian elimination in solving matrix equations of the form.

$$\mathbf{Ax} = \mathbf{b} \tag{1.1}$$

where $\mathbf{A}$ is an $n \times n$ matrix, $\mathbf{x}$ is a column vector of n unknowns, and $\mathbf{b}$ is a column vector of n values.

A previous study had shown that the solution time was reduced by a factor of fifteen when compared with the Matlab package. In light of the superior performance of this solution methodology it was decided at an early stage to incorporate the numerical solutions into the Workbench. In doing so the Java PWB becomes a self standing unit capable of deriving the underlying stochastic process of a model and providing a steady state solution.

4.2 Row Indexed Sparse Storage Mode

Typically the infinitesimal generator matrix for an N-state Markov process will be best represented within a computer as sparse. Sparse in this context can be taken to mean that there are relatively few N non-zero elements. The storage of such a matrix as a two dimensional array, where each of the dimensions is of size N, is highly inefficient in terms of both memory usage and run time due to the large number of zero elements and null calculations respectively.

In row indexed sparse storage mode only the non zero elements are stored in addition to auxiliary information to determine where an element logically belongs. In comparison with other storage schemes which may require as many as five times the number of number of non zero elements row indexed storage requires only about two times the number of non zero elements. Hence, it is particularly suited to this application where memory conservation is a prime concern.

To represent a square matrix A of dimension $N \times N$, a row indexed storage scheme sets up two one-dimensional arrays. These will henceforth be referred to as `ija` and `sa`. The `ija` array is used as an index to find the logical position of any element stored in the `sa` array. The `sa` array contains the value of all elements found in the matrix. The storage rules are:

- The first N locations of`sa` store the diagonal elements of matrix A in order.
- Each of the first N locations of `ija` store the index of the array `sa` that contains the first off-diagonal element of the corresponding row of the matrix.
- Location 1 of `ija` is always equal to $N + 2$
- Location $N + 1$ of `ija` is one greater than the index in `sa` of the last off-diagonal element of the last row (It can be read to determine the number of elements in the `ija` and `sa` arrays). Location $N + 1$ of `sa` is not used and can be set arbitrarily.
- Entries in `sa` at locations $\geq N+2$ contain the off-diagonal elements of A, ordered by row and, within each row, ordered by columns. (This has important implications for the Workbench as it demands that the instantaneous transitions from a state are ordered).
- Entries in `ija` at locations $\geq N+2$ contain the column number of the corresponding element in `sa`.

At first this seems an unorthodox form of storage but an example helps to clarify the logic of the scheme.

Consider a PEPA model with two copies of a component, P_1, executing in a pure parallel synchronisation. P_1 is a simple sequential process which undergoes a *starting* activity with rate r_1 to become P_2 and then performs a *stopping* activity with rate r_2 to become P_1 again.

$$\begin{aligned} P_1 &\stackrel{def}{=} (starting, r_1).P_2 \\ P_2 &\stackrel{def}{=} (stopping, r_2).P_1 \end{aligned}$$

The process has the following generator matrix.

$$\begin{pmatrix} -2r_1 & r_1 & r_1 & 0 \\ r_2 & -r_1 - r_2 & 0 & r_1 \\ r_2 & 0 & -r_1 - r_2 & r_1 \\ 0 & r_2 & r_2 & -2r_2 \end{pmatrix}$$

In row indexed sparse storage form this matrix is represented by the two arrays of length 13, as follows.

Here x is an arbitrary value. Notice, that according to the storage rules, the value of N (namely 4) is `ija[1]`-2, and the length of each array is `ija[ija[1]`-1`]`-1, namely 13. The diagonal elements in row i is `sa[`i`]`, and the off-diagonal elements in that row are are in `sa[`k`]` where k loops from `ija[i]` to `ija[`$i+1$`]`-1, if the upper limit is greater or equal to the lower one.

	1	2	3	4	5	6	7	8	9	10	11	12	13
ija	6	8	10	12	14	2	3	1	4	1	4	2	3
sa	$-2r_1$	$-r_1 - r_2$	$-r_1 - r_2$	$-2r_2$	x	r_1	r_1	r_2	r_1	r_2	r_1	r_2	r_2

Table 4.1: Sparse Representation of a Matrix

The generator matrix of a Markov process can be examined to determine the assigned state numbers of the successor states from any given starting state. For example, the column numbers of the non-zero elements in row three are the assigned state numbers of the successor states to state three. Rows which have a non-zero entry in their third column are states from which a transition to state 3 is possible.

This information is not lost when the matrix is encoded in row indexed form, but it is more difficult to derive. The relevance of this observation is, it is possible for the modeller to follow the direction of the arcs in the derivation graph or to step back against the direction of the arcs. Without knowledge of the state associated with its identifying state number this is of limited use. This topic is discussed in more detail in 5.2.3.

4.3 Biconjugate Gradient Method for a Sparse System

The Conjugate gradient methods provide a general means for solving the $N \times N$ matrix equation 1.1. The attractiveness of these methods is that they reference $\mathbf{A}$ only through its multiplication of a vector, or the multiplication of its transpose and a vector. These operations can be efficiently implemented for a row indexed sparse storage representation of a matrix. The actual method implemented is the biconjugate gradient method. It is suitable for solving linear equations which may or may not be positive definite or symmetric.

The method constructs four sequences of vectors, $\mathbf{r}_k$, $\overline{\mathbf{r}}_k$, $\mathbf{p}_k$, $\overline{\mathbf{p}}_k$, $k = 1, 2....$ The initial values of $\mathbf{r}_1$ and $\overline{\mathbf{r}}_1$ must be provided, and $\mathbf{p}_1 = \mathbf{r}_1$, $\overline{\mathbf{r}}_1 = \overline{\mathbf{p}}_1$. Then the following recurrence is carried out:

$$\begin{aligned}
\alpha_k &= \frac{\overline{\mathbf{r}}_k \cdot \mathbf{r}_k}{\overline{\mathbf{p}}_k \cdot \mathbf{A} \cdot \mathbf{p}_k} \\
\mathbf{r}_{k+1} &= \mathbf{r}_k - \alpha_k \mathbf{A} \cdot \mathbf{p}_k \\
\overline{\mathbf{r}}_{k+1} &= \overline{\mathbf{r}}_k - \alpha_k \mathbf{A}^T \cdot \overline{\mathbf{p}}_k \\
\beta_k &= \frac{\overline{\mathbf{r}}_{k+1} \cdot \mathbf{r}_{k+1}}{\overline{\mathbf{r}}_k \cdot \mathbf{r}_k} \\
\mathbf{p}_{k+1} &= \mathbf{r}_{k+1} - \beta_k \mathbf{p}_k \\
\overline{\mathbf{p}}_{k+1} &= \overline{\mathbf{r}}_k + \beta_k \overline{\mathbf{p}}_k
\end{aligned} \tag{3.2}$$

The sequence of vectors satisfies the *biorthogonality* condition.

$$\overline{\mathbf{r}}_i \cdot \mathbf{r}_j = \mathbf{r}_i \cdot \overline{\mathbf{r}}_j = 0, \; j < i \tag{3.3}$$

and the *biconjugacy* condition

$$\overline{\mathbf{p}}_i \cdot \mathbf{A} \cdot \mathbf{p}_j = \mathbf{p}_i \cdot \mathbf{A}^T \cdot \overline{\mathbf{p}}_j = 0, \; j < i \tag{3.4}$$

There is also a mutual orthogonality.

$$\overline{\mathbf{r}}_i \cdot \mathbf{p}_j = \mathbf{r}_i \cdot \overline{\mathbf{p}}_j = 0, \; j < i \tag{3.5}$$

These properties can be proved by straightforward induction. The recurrence must terminate after

$m \leq N$ steps with $\mathbf{r}_{m+1} = \overline{\mathbf{r}}_{m+1} = 0$, as all the N possible orthogonal search directions have been exhausted.

An initial guess $\mathbf{x}_1$ is made at the solution. The vector $\mathbf{r}_1$ is chosen to be the residual

$$\mathbf{r}_1 = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_1 \tag{3.6}$$

and choose $\overline{\mathbf{r}}_1 = \mathbf{r}_1$. Then obtain the sequence of improved estimates

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha \mathbf{p}_k \tag{3.7}$$

by carrying out the recurrence 3.2. Equation 3.7 guarantees that $\mathbf{r}_{k+1}$ from the recurrence is in fact the residual $\mathbf{b} - \mathbf{A} \cdot \mathbf{x}_{k+1}$ corresponding to $\mathbf{x}_{k+1}$.

It is possible that the recurrence may break down should one of the denominators be equal to zero but in practice this is rare. More importantly, the exact termination in at most N iterations occurs only with exact arithmetic. Roundoff error means that the process should be regarded as a genuinely iterative procedure, to be halted when some appropriate error criterion is met.

The user must enter a tolerance value against which the routine will compare a computed error metric to determine whether the elements in the $\mathbf{x}$ vector are within an acceptable error bound. There are four different error metrics. These are:

- The value $\mid \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \mid / \mid \mathbf{b} \mid$ is less than the given tolerance value
- The value $\mid \widetilde{\mathbf{A}}^{-1} \cdot (\mathbf{A} \cdot \mathbf{x} - \mathbf{b} \mid / \mid \widetilde{\mathbf{A}}^{-1} \cdot \mathbf{b} \mid$ is less than the given tolerance value
- The routine uses its own estimate of the error in $\mathbf{x}$, and requires its magnitude, divided by the magnitude of $\mathbf{x}$ to be less than the tolerance value
- As above except the largest (in absolute value) component of the error and the largest component of $\mathbf{x}$ are used instead of the vector magnitude.

It is left to the user of the Workbench to determine which criterion to apply. However, if none are selected the Workbench uses default values. The default convergence parameter is $\mid \mathbf{A} \cdot \mathbf{x} - \mathbf{b} \mid / \mid \mathbf{b} \mid$. The acceptable tolerance value is set at $1e - 15$ and the number of iterations to 100. These values are set arbitrarily and have no particular significance.

Chapter 5

The Structure of the Java PWB

5.1 Overview

The Workbench divides into three logical sections, the GUI at the front, the engine in the middle responsible for the generation of the derivation graph and the solution tool at the back to perform steady state analysis. The chapter initially describes implementation issues affecting the whole, then proceeds to describe the class structure and principal methods of each of the individual sections. Supporting data structures are detailed at the end.

5.2 Implementation Issues

Essential to the correct operation of the Workbench is a means of storing, identifying and labelling every state in the state space of a model. This introduces several problems.

- A suitable data structure is required to hold all the visited states.
- States as represented by the Workbench must be comparable.
- States must occupy the minimum amount of memory space.
- Labelling of states

Labelling of states is trivial; states are numbered in the order in which they are first visited. However the other issues are more complex and are discussed in turn.

5.2.1 The State Space Data Structure

The structure has to provide an efficient means of inserting a newly discovered state and an efficient means of returning the assigned state number should the state already exist as a visited node in the derivation graph. In the absence of a hypothesis for predicting the size of the state space of a model, without generating the state space, it is necessary to use a dynamic storage structure (The exception to this rule is the special case of a pure parallel arrangement of components). The selected data structure was a binary tree due to its $\log N$ search and insertion characteristics.

The binary tree does not allow a modeller to find the state associated with a given state number so is considered obsolete after all the transitions are stored or output. If the tree is collapsed to an array of references to processes, a method exists to return the state associated with a state number. The large contiguous memory blocks required to solve a Markov process does not permit memory to be stored wasteful on non essential structures. However, if the collapsed tree can be stored in a limited amount of memory, the possibility to walk around the state space as described in 4.2 would exist.

5.2.2 State Representation

Even small PEPA descriptions may have state spaces that consist of thousands or tens of thousands of states. The first version of the Java PWB, on the identification of a newly visited node in the derivation graph, stored the state in the binary tree as a reference to the top node of a process tree as described in 3.4.1. This was found to be intolerably inefficient in terms of memory usage. Blunt analysis can be used to calculate a lower bound on the memory requirements of the Java Virtual Machine(JVM) for each process tree. The tree consists of seven cooperations, and eight sequential processes. Each cooperation consists of two references to the two nodes below and a reference to a cooperation set. In this example the cooperation set is empty but the reference to the empty set still exists. The sequential components or leaves in the tree are also memory references. Assuming that the JVM uses four bytes for a memory reference, each process tree requires at least 116 bytes of memory!

The solution to this problem resides in structure of a process. If the static structure of a process is known any state in the state space can be identified by its sequential components. In terms of the process tree only the leaves change, the nodes stay fixed. Enumeration of the derivatives of the sequential components allows a state to be represented by an array of numerical values. This approach to the compact representation of a state was given more credibility by the realisation that even large models have a relatively small number of sequential components. A 160,000 state model was found to have only 40 sequential derivatives! The discovery that even large PEPA models have a very limited number of sequential components suggested that the array be composed of the basic Java `byte` type, further compressing the required storage space for a state.

5.2.3 State Evaluation

The binary state tree has to evaluate a state to determine its position in the tree relative to the other states. The mapping of a component to an object allows a straightforward evaluation of states. For example an activity extends an action object and includes a rate object as a data member. Comparison of the constituent parts of an activity allows activities to be evaluated relative to one another. The same logic applies to the comparison of processes. The combinators of the PEPA language allow processes to be formed from the interaction of components thus a process can be compared by evaluation of its subcomponents. The comparison of constants and prefixes cannot be undertaken directly so an arbitrary order between these syntactic terms is implemented (rather naively this ordering extended to the full set of language terms in the original design).

The array representation of a process is only intended as a means of state compression. It complicates the operation of finding the underlying Markov chain since it necessitates methods to breakdown a process to its array representation and build a process from its array representation. Conversely, it greatly simplifies the task of relative state evaluation as this is reduced to the comparison of byte arrays.

If one state can be stored as an array of bytes, then the whole state space can be stored as an array of bytes. This form of compaction makes the concept of walking around the state space a realistic proposition. Clearly, there is a reduction in the absolute size of a model that the Workbench. However, as will be later proven this is not significant.

5.3 Class Structure and Methods

5.3.1 GUI

The GUI makes extensive use of the AWT facilities found in the Java class library. The main window of communication extends the Java Frame class and the default border manager is used to arrange the contents appropriately.

The features of the GUI are supported by several pop up boxes. They all follow a similar design pattern, extending the dialog class provided by the AWT and using bespoke collections of panel objects and layout managers to give the intended aesthetic appearance for each window. The boxes are all event driven; they implement the action listener interface and wait for action events on their AWT components. If an event occurs, the actionPerformed method is invoked and this determines successive behaviour.

The GUI is supported by around one thousand lines of Java code, but this is principally aimed at the display of information. This information is sourced from the engine of the Workbench through a series of access functions; the GUI as it name suggests performs little in the way of calculation.

5.3.2 Workbench

The Workbench code is conceptually equivalent to the ML version. It contains mappings of the syntactic terms of PEPA to objects and other associated classes which perform more general duties. These are described in turn.

5.3.2.1 Class Action

The natural mapping of an action type in the PEPA modelling language to an action in an object oriented environment would suggest use of a string representation. To keep memory utilisation to a minimum each discrete action type is actually assigned an actionID which is one of basic Java `short` types. To support this representation two class structures are defined, a hashtable to given an efficient means of finding the actionID associated with the string and a Vector for the alternate conversion.

5.3.2.2 Class Rate

Notionally rates in PEPA are of two types, specified and passive. In maintaining compatibility with Maple it was necessary to implement a symbolic representation of a rate. This was a desirable objective in itself as it allows the state space of a model to be output with rates represented as character literals.

In the early stages of development it was anticipated that the modeller could write a model but leave some or all the rates as undeclared variables. The Workbench would calculate all the transitions in terms of these undeclared rates through the use of the combinator rate classes. If there was a requirement for the Workbench to calculate the sum of an undefined variable and a specified numerical value it would create a rate object of type Sum. This rate object could itself be combined with another rate object to produce a rate tree similar in structure to the process tree described in 3.4.1.

This was implemented through the use of eight rate subclasses using the case constructors provided by Pizza described in Chapter three. These are shown below

```
abstract class Rate
   {
      case Times(Rate r1,Rate r2);
      case Divide(Rate r1, Rate r2);
      case Plus(Rate r1, Rate r2);
      case Minus(Rate r1, Rate r2);
      case Min(Rate r1, Rate r2);

      case Spec(double r1);
      case Unspec(double r1);
      case Strung(String r1);
      ...
   }
```

Five of these are combinator rate classes the first four representing the basic mathematical notions (Times, Divide, Minus and Plus) and the other is a minimum combinator for use in calculating the apparent rate of an activity. The others, Spec, Unspec and Strung represent specified, passive and undeclared respectively.

By deferring evaluation of a rate until the completion of the transition matrix for a model, a modeller would have a means to recalculate the transition values through the simple evaluation of these rate structures. This would have been particularly useful to the modeller wishing to carry out steady state analysis with a variation in the rate input parameters as it would only of involved evaluation of the generator matrix for these new values. However the rate structures produced were found to be memory intensive for models involving even a modest number of cooperating components and support for this particular facility was dropped. The subtypes of the rate class are still implemented to allow symbolic output of rates but are not stored in the transition matrix.

5.3.2.3 Class Activity

This class extends Action and has an object of type rate as a class member. Its behaviour is determined by these two entities.

5.3.2.4 Class Process

Five constructors are implemented using the convenient syntax provided by the Pizza language. Class structures include a hashtable for expanding component definitions, a sequential process tree to determine the assigned sequential state number of a sequential component, an array of 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. The most significant methods are highlighted.

derivatives

This method recurses down a process tree until the sequential components are reached. The one step derivatives of these component and the enabled activities are returned to the nodes immediately above. The semantics of PEPA for cooperation (or hiding) are applied and the resulting one step derivatives and associated activities from this branch of the process tree are passed up to the next level. At the highest level a list of derivatives and associated activities for the process referenced on calling the method is returned. These represent all the instantaneous transitions available from the originating process and the rate and action type of these transitions.

apparentRate

```
protected void P2A(byte array[])
   {
      switch (this)
         {
         case Coop(Process P1, CSet<Action> l, Process P2):
            P1.P2A(array);
            P2.P2A(array);
            break;

         case Hide (Process P1, CSet<Action>l):
            P1.P2A(array);
            break;

         case (_):
            array[pCount++] = seqTree.retInt(this);
            break;
         }
   }
```

Figure 5.1: The Process to Array Algorithm (P2A)

The apparent rate method follows a similar pattern to the derivatives method. The call stack of the method grows until the atomic components at the leaves of the tree are reached. As the recursion unwinds the apparent rates are calculated for each level and passed up to the nodes above. The rules for the calculation of apparent rates are simple. If the components are competing for a resource, then their exponential distribution parameters are summed. If they are cooperating, then it is the slowest component that dictates the rate of the activity and the lesser of the two parameters is chosen.

sequential derivatives

This method enumerates the process class hashtable and calls the derivatives method to build a sequential process tree containing all the sequential derivatives. The purpose of the tree is analogous to that of the main process tree, to return the assigned state number of a process. On completion of the processing of the tree an array is constructed to perform the inverse operation.

P2A and A2P

These function are responsible for the systematic reduction of a process to its byte array representation and the building of a process tree from its array representation respectively. The first method recurses through a process tree identifying the sequential components and inserting the sequential state number of these components into a byte array. The latter uses a flattened representation of the parsed process and a byte array to create a depth first rebuild of a process. Both methods use the type matching of Pizza for the identification of sequential components. The P2A algorithm is representative of the recursive methods found in this class and is presented in Figure 5.1.

flattenParse

The flatten parse method operates in the similar style to the above. It takes the parsed process and stores the static combinators in a list structure for use in the reconstruction of processes from their array representation.

5.3.2.5 Class Driver

The driver class is the engine of the Workbench. It includes the routines responsible for the derivation of the transition graph and the output of states and transitions to file. The GUI communicates only with the driver class; the operations of the Workbench are invoked via this class to maintain data encapsulation and support a logical package structure. When a modeller selects the run option in the GUI, the Driver class flattens the parsed process and calls the sequential derivatives function previously described. Depending on the flag options set, either the NR(Numerical Routines) method or the MAMA (Matlab - Maple) method is then called.

MAMA

The parsed process is passed as an argument to the method. It is reduced to its array representation and pushed on the back of a list containing the unexplored states (this list dynamically grows and contracts as the state space is explored). The following iteration is then performed. An array is popped of the unexplored queue and reconstructed into the Java representation of a process before calling the derivatives method. The derivatives method returns all the successor states and associated activities. One by one the successor states are reduced to their array representation and passed to the tree structure to obtain an assigned state number. Newly visited states are pushed on the back of the unexplored queue and heap space is reallocated for the next array. The elements in the unexplored queue reference the same memory addresses as the nodes in the state tree. This prevents memory seepage. There should only ever be one copy of the array representation of a state in memory at any instance (except when the tree is being collapsed). The iteration stops after the unexplored list is emptied. States and transitions are written out to file as the iteration progresses.

NR

When this method is selected the transitions and state space are stored in memory. The only known method of calculating the number of transitions in a PEPA model is through the application of the operational semantics to the language terms. This introduces a problem. Dynamic structures such as linked lists are applicable to the storage of elements where the number of elements to be stored is not known but they are inefficient in comparison with arrays for the storage of large numbers of elements.

The NR method has two distinct phases. During the first phase the state space tree is built and an over estimate of the size of the transition set is calculated. In the second phase space is allocated in the memory for the rates and these are stored. The iterative process in these phases is similar to that outlined for the MAMA method above. The apparent rate method is disabled during the first phase to improve the run time.

This is the clearest example of a trade off between the execution time of the Workbench and maintaining the maximum amount of contiguous memory space for the provision of a solution. The run time could be significantly improved by allocating the space for the transition matrix in discrete blocks. Each time a block is filled it is assigned to an existing variable and space reallocated for the next discrete block. This seems an excellent solution but has the side effect of littering the memory space of the workstation. As outlined previously, memory is considered a more precious resource than time.

5.3.2.6 Class LBGM (Linear Biconjugate Gradient Method)

An object of the class is instantiated with the sparse representation of the generator matrix. There is one general routine implemented as a public method and called with the convergence criteria outlined in 4.3 as arguments. The access modifiers of all other methods within the class are private; these methods represent the common matrix operations which are performed iteratively. Whereas the the routines supplied by numerical solutions operate on vector structures the Java version uses arrays. The vector structure defined by numerical solutions is analogous to an array in Java; it is a structure containing an array with a size data member.

5.3.2.7 Class SGM (Static General Methods)

This class contains a host of general routines for array manipulation and should never be instantiated. The generator matrix once encoded in the row indexed format must be transposed and manipulated to incorporate the normalisation condition outlined in 2.5.3. The transposition of a sparse matrix representation is non trivial and a collection of routines is supplied by Numerical Recipes for the task. However, the Workbench does not at this point hold the matrix in the pure format outlined in 4.1. The off-diagonal elements need not be calculated until the matrix is actually transposed and so are not stored. This is a significant memory conservation; the `sa` array is furnished in double precision; a one million state model would use 8Mb of memory pointlessly.

There are two main methods which interpret the transition matrix as generated by the Workbench and transpose it into the form required for generating a solution. The functionality of the SGM class does not end here though. Once the normalisation condition has been included in the generator matrix it may be passed to the LBGM class (so steady state analysis can be carried out) but it no longer accurately reflects the transitions of the derivation graph. This is resolved by methods to write over and write back the final row of the transition matrix. These are invoked every time the modeller uses the walkabout facility described in 6.6.2.

The methods that implement the walkabout functionality are also defined in this class. Finding the the non-zero elements in a row is straightforward. The elements belonging to a row can be obtained by referencing the `ija` array for the chosen row number and finding all the associated column entries. Walkabout is instigated only after the matrix has been transposed (there being no advantage to be gained from maintaining the data in its pretransposed format). The set of numbers returned represents the nodes reached on traversing the derivation graph against the direction of the connecting arcs.

The walk forward method is more complex. The first step is to loop through the `ija` array for all index values greater than N, (where N is the number of transitions in the Markov model) looking for matches. Once the index values of the matching elements are obtained, the associated row numbers must be determined. The algorithm that performs this operation uses the generic priority queue and list structures and is highlighted in Figure 5.2

5.4 Auxiliary Data Structures

5.4.1 The Process and Array Trees

Both the array and process trees are conventional and need little explanation. The array tree implements an additional method to collapse the tree for use by the walkabout and PEPA state finder utilities. Both structures are supported by a templated node class.

5.4.2 The List Structure

The parametrised data types of Pizza are used to implement a generic singly linked list. The functionality of the list includes a shallow copy method and a concatenation method, in addition to push and pop methods. The shallow copy method is used in the one step derivatives function to cycle through action sets in the search for mutual action types. It eliminates the need to copy the whole list every time a check is warranted. It is left to method scoping and and the in built garbage collection provided by Java to clear unreferenced objects.

```
public static int[] walkforward(int[] ija,int matDim, int startstate)
{
    PQueue<int> locQueue1 = new PQueue();
    Queue<int> locQueue2 = new Queue();

    // PHASE 1 - FIND MATCHING COLUMN ENTRIES
    for(int loop = matDim +1; loop < ija.length; loop++)
        if (ija[loop] == startstate)
            locQueue1.pqInsertFF(loop,loop);

    // PHASE 2 - FIND ASSOCIATED ROW NUMBERS
    int start = 0;
    while (!locQueue1.isEmpty())
        {
            int match = locQueue1.pqPop();
            for(int loop = start; loop < matDim; loop++)
                {
                    if ((ija[loop+1] > match) && (ija[loop] <= match))
                        {
                            locQueue2.pushQueue(loop + 1);
                            start = loop;
                            break;
                        }
                }
        }
    int[] array = new int[locQueue2.retNOE()];
    for (int loop = 0; loop < array.length; loop++)
        array[loop] = locQueue2.popQueue();
    return array;
}
```

Figure 5.2: The Walk Forward Algorithm

5.4.3 The Priority Queue

For correct operation of the common matrix routines all the elements in each row of the generator matrix must be sorted in ascending column order. This has important implications for the Workbench. It imposes an ordering on all the instantaneous transitions out of a state. The arbitrary ordering of derivatives for each state necessitates use of an auxiliary data structure to sort them. A particular difficulty is the possible existence of more than one instantaneous transition out of a state leading to the same state. At the level of the derivation graph consider two nodes with more than one arc connecting them. The rate at which the transition out of the source state to the state in question is performed is the sum of the exponential distribution parameters.

A data structure to support ordering of elements is a priority queue. Such data structures are common and a generic implementation could have been used. However it was decided to produce a structure giving the precise functionality demanded by the Workbench, in particular to resolve the example illustrated above. More than one instantaneous transition out of a state to the same state corresponds to an insertion in the queue of two elements with the same priority. The solution was to implement the insert method as returning a boolean. A false value signals that an element of equal priority already exists and the new element has not been inserted, whilst a true value signals the element was inserted correctly. If a false flag is signalled the element, or rate, with the repeated priority value is removed from the queue and summed and the new combined rate of the transition is inserted.

When a node in the derivation graph is visited for the first time in the exploration of the state space the state count is incremented by one and it is assigned this number. Since this state must have a higher priority than all other elements it is pushed on to the back of the queue.

Chapter 6

The GUI and its facilities

6.1 Overview

The minimum expectation of the GUI was to provide an easy to use interface between the structured code of the Workbench and the user. It has been previously stated that the memory resources of a workstation are taxed not only by the need to store all the states as per the ML Workbench but also by the requirement to store the transitions. This burden is turned to an advantage in terms of the facilities that the GUI is able to provide. The facilities of the GUI are discussed individually and are compared directly (where there is an equivalent ML facility against which to do so) with the legacy code.

6.2 The GUI Window

On instantiation of the Workbench the GUI window as shown in Figure 6.1. The window contents include a menu bar at the top for setting parameters in the Workbench code, a large text area for examining models and the output of results, and several text areas at the base for displaying certain informative metrics. Two of these relate to the number of states and transitions in a model. The other two indicate the number of iterations performed during the calculation of the steady state solution and the rounding error of the convergence. The number of states and transitions in a model are updated intermittently as the derivation graph is explored.

A menu bar is located at the top of the window. It presents the user with a variety of options for the instantiation of the main Workbench code. The option menus are self contained and are explained in detail in the following sections. The status box informs the user of any generated exceptions and general changes in the Workbench status.

6.3 The File Menu Bar

The file menu is the simplest of the menus, containing only two options. The user may either quit the Workbench or load a model from file. If a model is selected to be loaded from file it will automatically be parsed. A parse error is reported in the status box for syntactically incorrect models.

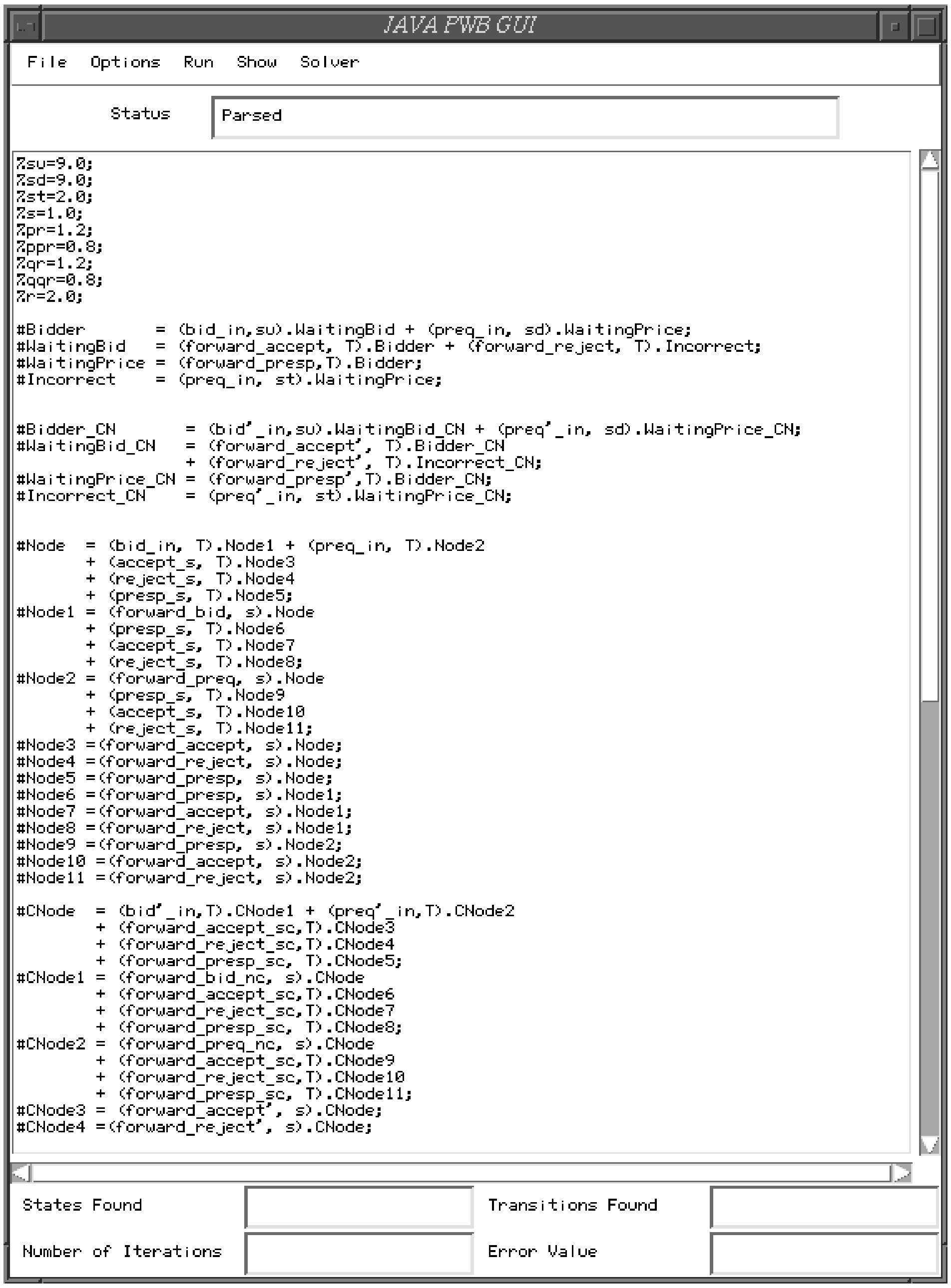

Figure 6.1: The Java PWB Graphical User Interface

6.4 The Options Menu Bar

This 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 the row indexed format detailed in 4.1. If the modeller does not specify a solution the Numerical Recipes option is chosen by default.

6.5 The Output Menu Bar

The ML version of the Workbench outputs the states of a model to file in a hashed lexigraphical representation. The Java PWB offers four alternatives to the user. 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 *modelname*.seq file is created containing the sequential derivatives of a model and their associated sequential state numbers.

6.6 The Utility Menu Bar

The utilities offered by the Java PWB are discussed in turn.

6.6.1 The PEPA State Finder

This feature is only available to a modeller if the Numerical Recipes option is selected in the options menu bar. The reasoning for this is that the Java PSF, unlike the ML version performs state recognition on data held in memory by comparing their byte array representations.

If the PSF option is selected a window is displayed listing all the possible sequential components as shown in Figure 6.2. There is a reference box at the top of the window containing the parallel composition of the parsed process. For the purposes of clarity all the sequential components in the parallel composition are replaced by the letters SEQ. Double clicking on any sequential component in the list results in it being displayed in the lower text area juxtaposed to the appropriate static combinator(s). Should it be the outermost sequential component then a new line is started in anticipation of the next series against which a match is desired.

This approach eliminates typing errors as the list of sequential components provided by the utility is sourced from the parsing of the model. As with the ML version the utility avoids the pitfall of finding the same state twice. The user may select a wild card in lieu of any particular sequential component. All components are then considered legitimate matches.

The PSF can be run after or before steady state distribution vector has been evaluated. Should the PSF be initialised after steady state analysis is complete it will automatically calculate the total probability of the model being in the subset of states identified. The results of the utility are output to file in the same directory as the model. The filename is the same as that of the model and the identifier tag is .psf.

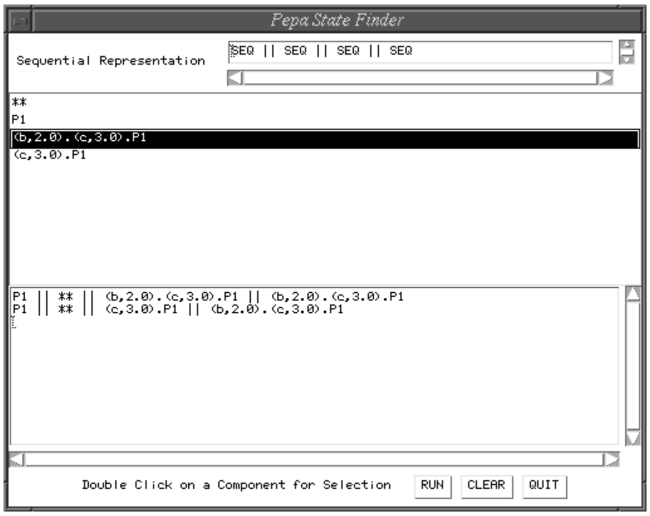

Figure 6.2: The PEPA State Finder Utility

6.6.2 The Walkabout Utility

Complex models are time consuming to develop and due to their complexity difficult to debug. In a strongly connected ergodic model it is possible for a process to visit any state in the state space regardless of the starting state. If these conditions do not hold models may contain a livelock, where a subset of the state space is continuously cycled or even a deadlock from which no instantaneous transition is available. To assist in the debugging of models the Java Workbench offers a 'walkabout' function. The walkabout facility allows the modeller to walk around the derivation graph of a model in the direction of the arcs connecting the nodes, termed walk forward, or against the direction of the arcs, termed walk backward.

The AWT classes are used to produce a utility that is both easy to use and understand. Instantiation of the walkabout option creates the window, pictured in Figure 6.3. At the top left can be seen two buttons for the selection of the walk direction. The text area next to these buttons displays the reference state. A list of states and their assigned state numbers is displayed in the main visual area of the window. The reference state is the node in the derivation graph that the states in the list are sourced from. If the forward direction is selected these represent the successor states of the reference state. If, however, the backward direction is selected the listed states represent states from which a transition to the reference state is possible. To update the reference state the user simply double clicks on any of the listed states and the window is refreshed.

The value of this facility is obvious. The default reference state is the starting state for the evolution of a process, but can be initialised to any state which is known to be deadlocked. By selecting the walkback option the modeller can examine the states which lead to the deadlock resulting in faster model fault diagnosis. There is no ML equivalent to this utility; it results directly from the holding of all transitions in the derivation graph in memory and the collapsing of the state tree.

6.6.3 The Throughput Utility

The Throughput Utility returns all the action types that are part of the finite action set that constitute the model. A window appears and a list is drawn displaying each member of the finite action set. An example can be seen in Figure 6.4. To select an action for throughput analysis the user double clicks on that item in the list. Calculation of the throughput vector necessitates

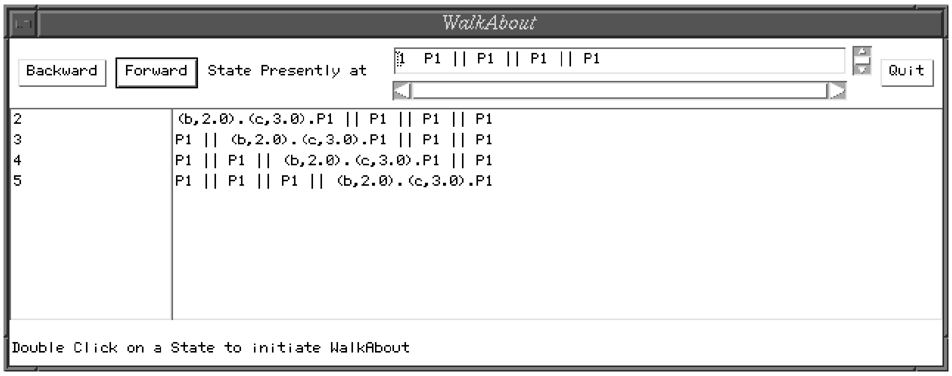

Figure 6.3: The Walkabout Utility

Figure 6.4: The Throughput Utility

running the code to produce the infinitesimal generator matrix. The need to spare memory does not permit the storage of activities, only rates. In other words the action type must be selected before the state space is generated otherwise it is ignored. Throughput is reset every time a new model is parsed.

6.6.4 The Rate Utility

The text file containing a model may declare rates symbolically and typically these are assigned values at the top of the file. The modeller may introduce symbolic rates into a file but not assign any values. 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. Failure to assign a numeric value to a rate will cause the Workbench to generate an invalid rate exception if either of the other options is selected. This is reported in the status box of the GUI. An example rate window is shown in Figure 6.5. Its operation is trivial and is not described here. It should be noted that any changes to assigned rate values are not reflected in the text file containing the model.

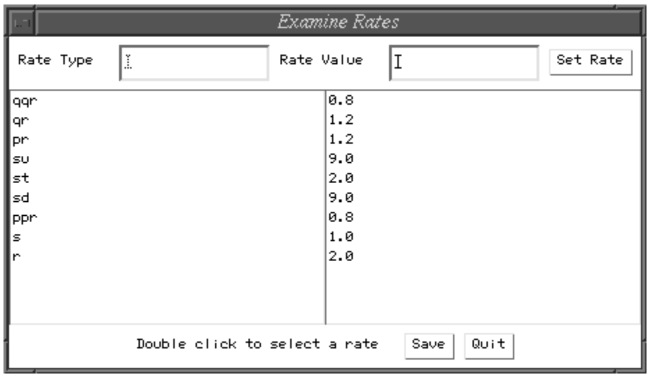

Figure 6.5: The Rate Utility

6.7 The Solve Menu Bar

This contains two different options. 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 4.3 as these are adequate for ensuring an acceptable level of convergence for even large models. If a modeller attempts to run the solution tool and the matrix is not encoded in the appropriate format a message is printed out to this effect in the status box.

Chapter 7

Results and Evaluation

7.1 Overview

The ML Workbench is only capable of writing the states and transitions to file with the appropriate syntax of the high level solution tool chosen. In contrast, the Java version optionally permits the transitions to be stored in memory. Some comparison of the two Workbenches is not unreasonable, but it should be remembered that the Java PWB is a complete package. It provides a greater range of features to the modeller and is a stand alone device which does not need to interface with any mathematical software tool. This makes a critical examination more difficult. Back to back testing was performed to evaluate the run times of the two tools and to estimate the upper limit of state space for which each is able to complete the derivation graph.

7.2 Experimental Details

The host workstation for all the experiments was a Linux machine with a 400 MHz processor and 128Mb of memory. Thirteen different models were used in the evaluation of the two Workbenches. All have a similar structure, a common memory unit cooperating with a series of parallel processors. The processors are capable of some internal processing but must cooperate with the memory unit to perform external processing. Three different types of processor are defined. The only variation in the models was in the number of these components. The smallest model consists of only one copy of each component, for a total of three and the largest contains five of each, for a total of fifteen. The largest model, named tomp555.pepa is listed in Appendix A.

The ML Workbench was ran once for each model, the Java version twice. On the first occasion the transitions were output to file and on the second occasion stored in memory. The back end of the Workbench cannot generate a solution in terms of symbolic rates so these were assigned values in the second set of Java experiments. The first of the right hand columns relates to the ML version, the second to the Java version writing transitions to file and the third to the Java version, but this time maintaining the transitions data internally. This is equivalent to complying with the Maple and Numerical Recipes solution tools respectively.

[1]Failed to allocate memory for transposed transition data
[2]Failed to allocate memory for transition data

Model number	Processes	States	Transitions	Timings		
				ML	**Java 1**	**Java 2**
1	1 P_1, 1 P_2, 1 P_3	72	156	0.078	0.108	0.173
2	2 P_1, 1 P_2, 1 P_3	176	480	0.155	0.384	0.423
3	2 P_1, 2 P_2, 1 P_3	416	1360	0.395	1.076	1.139
4	2 P_1, 2 P_2, 2 P_3	960	3648	1.060	3.106	2.867
5	3 P_1, 2 P_2, 2 P_3	2176	9408	3.017	8.351	7.624
6	3 P_1, 3 P_2, 2 P_3	4864	23552	8.150	22.704	19.428
7	3 P_1, 3 P_2, 3 P_3	10752	57600	21.619	51.794	49.519
8	4 P_1, 3 P_2, 3 P_3	23552	138240	57.769	129.393	121.713
9	4 P_1, 4 P_2, 3 P_3	51200	326656	149.313	326.540	301.46
10	4 P_1, 4 P_2, 4 P_3	110592	761856	381.228	788.942	791.700
11	5 P_1, 4 P_2, 4 P_3	237568	1757184	1026.672	2122.190	1879.841
12	5 P_1, 5 P_2, 4 P_3	507904	4014080	-	5205.772	1848.290 [1]
13	5 P_1, 5 P_2, 5 P_3	1081344	9093121	-	-	4675.273 [2]

Table 7.1: Run Time Comparisons

7.3 Results

The outcome of the experiments is shown in Table 7.1. An important aspect of the results was the complete conformance between the two tools in the reported number of states and transition of each model. That the ML version runs more rapidly is no surprise, it was compiled to native code whereas the Java version was interpreted. The speed advantage of the ML Workbench is greatest for the smallest models (as a percentage), but as the number of states rises this is gradually reduced to approximately a factor of two.

The ML Workbench successfully completed the derivation graph for model number 11 but thereafter the tool exhausted the memory resources of the Linux machine. The Java Workbench succeeded in calculating the steady state solution for model number 11. It also managed to generate the state space and output the transitions to file for the 12th model but was unable to provide a steady state solution.

Writing only the states to file allowed the Java tool to complete the derivation graph for the largest model with over one million states, providing definitive proof of its superior memory efficiency. However, maintaining the data internally caused the Java tool to fail whilst trying to allocate space for the transposition of the sparse representation of the generator matrix for the largest two models.

A hidden detail of the results is the poor IO performance of the Java language. When the Java version maintains the transition data internally it explores the state space twice but calculates the transition set only one. On the first cycle the Workbench builds the state space tree and performs a conservative over estimate of the transition set to allow the allocation of a large contiguous block of memory for the rate data (An alternative to this method was outlined in 5.3.2.5 but implementation of this technique reduces the absolute size of a model that the Workbench is able to solve). Certain optimisations are performed to speed up the run time of the first state space cycle, including the non-calculation of apparent rates. The point is, the execution of the code which outputs the transitions to file takes as much time as the second cycle of the derivation graph.

7.4 Other Comparison Metrics

Unfortunately, time did not permit temporal testing of the linear biconjugate gradient algorithm integrated into the Workbench in comparison with the functionally identical C routines supplied by Numerical Recipes. It was observed that the run time for the linear biconjugate gradient method is modest in comparison with the time expended generating the state space.

The source code for the two tools consists of 1200 lines of code for the ML Workbench and 4500 lines for the Java tool. This can be explained by the more succinct specification of the ML language and the greater complexity of the Java tool. It has no particular merit as a means of relative evaluation.

7.5 Conclusions

The specific objectives of this project were outlined in Chapter 1 and were stipulated as:

- Implement a linear algebra package as a back end to the Workbench to create an integrated software package;
- Rewrite the implementation in a mobile language to give greater platform mobility
- Include an easy to use GUI as a front end to the Workbench
- To carry out the above as efficiently as possible in terms of memory usage to enable the analysis of large PEPA models.

The linear algebra package was integrated as intended and is believed to be functioning correctly. The viability of the Java as the implementation medium has been demonstrated. It is a mobile code and can, in theory, be ran on any machine incorporating the JVM. It has been definitively proved that the Java version is more memory efficient than the ML, in the generation of the state space for larger models. Finally the Java version incorporates some extra useful features, in particular the unique walkabout utility described in Chapter 6 which it is hoped will greatly facilitate the task of writing models in the PEPA language.

7.6 Future Directions

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

7.6.1 Support for an Experimental Programme

The transition matrix could calculate the transitions in the state space symbolically and hold the data in the transition matrix as references to generalised rate objects, as outlined in 5.3.2.2. A sensible upper bound would have to be set on the number of states in the state space to prevent memory exhaustion. 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 would give rapid feedback of the performance metrics of a model, particularly bottlenecks in the system, in a short time frame.

7.6.2 Recognition of Equivalence

The compositionality of the PEPA modelling language can give rise to symmetries within models which may be exploited to allow a system to be represented by a compacted model with the same essential characteristics. For a simple example consider the case of two components running in parallel as detailed below.

$$P1 \mid\mid P2 \qquad\qquad\qquad P2 \mid\mid P1$$

The two states are to the external observer equivalent. This analysis can be applied on a larger scale. It is proposed that an algorithm be incorporated into the Java PWB to recognise this equivalence.

7.6.3 Parallelising the PEPA language

A more radical break from the past would be an attempt to parallelise the Workbench. Consider the parallel composition detailed below.

$$P \mid\mid Q \mid\mid R$$

Let us assume that the model is strongly connected and ergodic, and P has say, 16 sequential components. These components are tagged cyclically between, perhaps, 4 processors and each is then responsible for dealing with the parallel compositions for which P has their associated tag value. For example, if a processor calculates a derivative of the parallel composition and the tag value of the sequential component of P for that derivative belongs to another processor then it is sent to that processor. There would be clearly be memory and communication difficulties with this approach, but it would greatly broaden the horizon of the PEPA modelling language into a new and dynamic area.

Bibliography

[1] G. Clark, S. Gilmore, J. Hillston, and N. Thomas. Experiences with the PEPA Performance Modelling Tools. *IEE Proceedings—Software*, 146(1):11–19, February 1999. Special issue of papers from the Fourteenth UK Performance Engineering Workshop.

[2] S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In G. Haring and G. Kotsis, editors, *Proceedings of 7th Conf. on Mod. Techniques and Tools for Computer Perf. Eval.*, volume 794 of *LNCS*, pages 353–368, 1994.

[3] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[4] Kathy Walrath Mary Campione. *The Java Tutorial Second Edition*. Addison Wesley, 1998.

[5] M Odersky and P Wadler. Pizza into java: Translating theory into practice. In *Proceedings 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[6] R.Harper R. Millner, M.Tofte and D. MacQueen. *The Defintion of Standard ML: Revised 1997*. The MIT Press, 1997.

[7] W.T. Vetterling W.H. Press, S.A. Teukolsky and B.F. Flannery. Numerical Recipes in C: The Art of Scientific Computing. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

Appendix A

Test Model

tomp555.pepa

```
%lamba1=1.0;
%lamba2=2.0;
%lamba3=3.0;
%mu1=4.0;
%mu2=5.0;
%mu3=6.0;
%r=7.0;
%g=8.0;

#P1 = (think, lambda1).P1';
#P1' = (get1, g).P1'';
#P1'' = (use, mu1).P1''';
#P1''' = (rel, r).P1;


#P2 = (think, lambda2).P2';
#P2' = (get2, g).P2'';
#P2'' = (use, mu2).P2''';
#P2''' = (rel, r).P2;

#P3 = (think, lambda3).P3';
#P3' = (get3, g).P3'';
#P3'' = (use, mu3).P3''';
#P3''' = (rel, r).P3;

#Mem1 = (get2, infty).Mem12' + (get3, infty).Mem13';
#Mem12' = (use, infty).Mem12'';
#Mem12'' = (rel, infty).Mem2 ;
#Mem13' = (use, infty).Mem13'';
#Mem13'' = (rel, infty).Mem3;


#Mem2 = (get1, infty).Mem21' + (get3, infty).Mem23';
#Mem21' = (use, infty).Mem21'';
#Mem21'' = (rel, infty).Mem1 ;
#Mem23' = (use, infty).Mem23'';
```

```
#Mem23'' = (rel, infty).Mem3;

#Mem3 = (get1, infty).Mem31' + (get2, infty).Mem32';
#Mem31' = (use, infty).Mem31'';
#Mem31'' = (rel, infty).Mem1 ;
#Mem32' = (use, infty).Mem32'';
#Mem32'' = (rel, infty).Mem2;

(P1 <> P1 <> P1 <> P1 <> P1 <> P2 <> P2 <> P2 <> P2 <> P2 <>
                P3 <> P3 <> P3 <> P3 <> P3)  <get1, get2, get3, use, rel> Mem1
```