Interface Engineering and Transient Analysis for the PEPA Workbench

Feng Wan

Master of Science
Department of Computer Science
University of Edinburgh
September 15, 2000

Abstract

The PEPA Workbench is a modelling tool for composing and checking descriptions of systems composed of communicating concurrently active components. It implements the PEPA language and provides solution and analysis facilities for PEPA models. The PEPA Workbench exists in several different versions and this project has extended and adapted a version which re-implemented the original Standard ML version in Java and Pizza, a language which extends Java with additional data structures. One goal of this project is to enhance the user interface of the Java version by making use of Java 1.2's Swing API. A necessary addendum to this would be re-implementing the Pizza classes as pure Java classes. Another goal is to extend the PEPA Workbench with facilities to compute transient solutions which solve probability distributions at an arbitrary time t.

Acknowledgements

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

Table of Contents

Chapter 1

Introduction

1.1 Overview

Modelling and simulation are methods which are commonly used by performance analysts to represent constraints and optimise performance. The performance of a system influences its design, procurement and subsequent use. Performance analysis may be carried out at any stage of that life-cycle. In general, the problem is to achieve the highest possible performance given the constraints on the system. We use simulation models to mean something which represents a system in a form which allows us to make predictions about the behaviour of the system. How accurate these predictions are will depend on the detail we invest in the model.

Like queueing networks and stochastic Petri nets, and their variants, process algebras can be regarded as 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. Performance Evaluation Process Algebra (PEPA) is a stochastic process algebra and 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.

The PEPA Workbench is a modelling tool for composing and checking descriptions of systems composed of communicating concurrently active components. It implements the PEPA language and provides solution and analysis facilities for PEPA models. The PEPA Workbench exists in several different versions but this project extended and adapted a version which re-implemented the original Standard ML version in Java and Pizza, a language which extends Java with additional data structures. One goal of this project is to enhance the user interface of the Java version by making use of Java 1.2's Swing API. A necessary addendum to this would be re-implementing the Pizza classes as pure Java classes. Another goal is to equip the PEPA Workbench with the ability to compute transient solutions which solve probability distributions at an arbitrary time t.

1.2 Project goals

The Pizza version of PEPA Workbench was available when the project was received. Pizza is a language which extends Java with additional data structures, the most useful of which are the algebraic data types and parametric polymorphism offered by the language and significantly simplifies the implementation of complex programs. However, and unfortunately, Pizza has stopped progressing and we need to re-implement the Pizza classes as pure Java classes.

The steady state probability distribution provides a basis in the calculation of performance measures and is available in the Pizza version. However, steady solutions cannot solve the probability distribution at an arbitrary time t, which makes transient solutions a useful addendum to the PEPA Workbench.

The facilities of the Abstract Window Toolkit (AWT) were used in the creation of a GUI to act as the front end to the Pizza Workbench. Java 1.2's Swing API is now available and makes the GUI more friendly to users, which is another goal of this project.

Thus the goals of the project are as follows:

- Re-implementing the Pizza classes as pure Java classes.
- Add transient solutions to the PEPA workbench.
- Update the GUI using the facilities of Java 1.2's Swing API.

1.3 Design and Implementation

The design of the Java version of the Workbench did not begin with a clean sheet, 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 the PEPA Workbench can be reused. Pizza was chosen as the implementation language primarily because of the pattern matching offered, the polymorphic properties 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. Typically the heterogenous translation yields code that runs faster, while the homogenous translation yields code is more compact.

The Pizza version of the PEPA Workbench solved steady state distribution, which exists only if the underlying Markov process is finite, time homogeneous and irreducible. In order to obtain probability distributions at an arbitrary time t, transient solutions should be a necessary addendum to the Workbench. A software package called Expokit [4] was available to compute matrix exponentials. More precisely, it computes either a small matrix exponential in full, the action of a large sparse matrix exponential on an operand vector, or the solution of a system of linear ODEs with constant inhomogeneity. The backbone of the sparse routines consists of matrix-free Krylov subspace projection methods (Arnoldi and Lanczos processes), and that is why the toolkit is capable of coping with sparse matrices of large dimension. The computation of matrix exponentials is a numerical issue of critical importance in the area of Markov chains and furthermore, the computed solution is subject to probabilistic constraints. The format required to interface with this package is subtle, as the matrices are encoded in a Compressed Row Storage (CRS) mode. Output from the Pizza Workbench is not directly compatible with this form of matrix representation.

The Pizza version GUI was implemented using the facilities of the AWT class library and was found to be a convenience to users. The JDK 1.2's Swing API is now available and provides more facilities than the AWT class library as well as being more friendly to users. Also there are several addendum to the GUI according to the feedback from users.

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 Java language with all the stated objectives being met. Testing of the new Java Workbench has shown complete conformance on the number of states and transitions found in comparison with the same models tested on the Pizza version. This gives some degree of confidence in its correct behaviour. A big addendum to the Pizza Workbench is that the Java version provides transient solutions which are able to compute the probability distributions at an arbitrary time t. Testing of the transient solutions on the new Java Workbench also has shown complete conformance with the same models tested using steady solutions and the transient solutions run more rapidly. Some of the facilities available to the modeller in the Java PWB were not available in the Pizza version.

Chapter 2

PEPA and Markov Process

2.1 Overview

Performance Evaluation Process Algebra (PEPA) is a stochastic process algebra and 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.

2.2 The PEPA 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 integration 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.

PEPA models are described as interactions of *components*. Each component can perform a set of actions: the 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 web-based information system.

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 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 required 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 ($\underset{L}{\bowtie}$): 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 carried out independently of the server. Cooperation over again 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:

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

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:

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

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 complete 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:

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

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

2.3 The Markov Process

A Markov process is the underlying process of Performance Evaluation Process Algebra (PEPA). Our primary objective with respect to a Markovian model will be to calculate the probability distribution of the random variable $X(t)$ over the state space S, as $t \to \top$. This is termed the *steady state probability distribution.* From this probability distribution we will derive performance measures based on subsets of states where some condition holds. Markov processes are widely used because it is well-understood how to solve them.

The memoryless property means that once we have arrived at a particular state the future behaviour is always the same regardless of how we arrived in the state. The solution of Markov processes is closely related to their representation by matrices. Finding the average behaviour of the model then corresponds to solving a simple matrix equation.

Like any stochastic process, a Markov process is characterised by a random variable $X(t)$, indexed over time t, and state space S such that $X(t) \in S$ for all t. The dynamic behaviour of the system being modelled is represented by the transitions between these states, and times spent in states, *sojourn times*. In general, these holding times represent the periods while processing is occurring in the system and, the transitions represent events in the system.

If a state $i \in S$ is entered at time t and the next state transition takes place at time $t + T$, T is the sojourn time in state i. By the Markov property, at any time point τ, the distribution of the time to the next change of state is independent of the time of the previous change of state. In other words, sojourn times are memoryless. Since the only probability distribution function which has this property is the exponential distribution this tells us that the sojourn time in any state of a Markov process is an exponentially distributed random variable. Hence at time τ, the probability that there is a state transition in the interval $(\tau, \tau + dt)$ is $q_i dt + o(dt)$, where q_i is the parameter of the exponential distribution of the sojourn time in state i.

Suppose that when a transition out of state i occurs, the new state is j with probability p_{ij}. By the Markov property, this must only depend on i and j. Thus we have, for $i \neq j$, $i, j \in S$,

$$Pr(X(t + dt) = j | X(t) = i) = q_{ij} dt + o(dt)$$

where the $q_{ij} = q_i p_{ij}$, by the decomposition property. The q_{ij} are called the *instantaneous transition rates*, and if the average time delay before a transition from i to j is exponentially distributed with parameters τ, then $q_{ij} = \tau$.

In practice, we are usually given the q_{ij} because we will known the parameter of the distribution governing the time delay associated with the event which is represented by the state transition.

From the q_{ij} we can derive the *exit rate*, q_{ij}, and the transition probabilities, p_{ij}, as follows. The exit rate is the rate at which the system leaves state i, i.e. it is the parameter of the exponential distribution governing the sojourn time. This will be the minimum of the delays until any of the possible transitions occurs. Thus, by the superposition property it is the sum of the individual transition rates, i.e. $q_i = \sum_{j \in S} q_{ij}$. The transition probability p_{ij} is the probability, given that a transition out of state i occurs, that it is the transition to state j. By the definition of conditional probability, this is $p_{ij} = q_{ij}/q_i$.

State transition diagrams: For small Markov processes the simplest way to represent the process is often in terms of its state transition diagram. In such a diagram we represent each state of the process as a node in a graph. The arcs in the graph represent possible transitions between states of the process. The arcs are labelled by the *transition rates* between states. Since every transition is assumed to be governed by an exponential distribution, the rate of the transition will be the parameter of the distribution. Irreducibility implies that the state transition diagram must be strongly connected.

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 row of the matrix ($j \neq i$) will be q_{ij}, the instantaneous transition rate. In other words, it will be the sum of the parameters labelling arcs connecting nodes i and j in the state transition diagram. The diagonal elements are chosen to ensure that the sum of the elements in every row is zero, i.e. $q_{ii} = -\sum_{j \in S, j \neq i} q_{ij}$

See [2] for steady state solutions and see chapter 4 for the details of transient solutions.

Chapter 3

Re-implementation

3.1 Overview

The Pizza version of PEPA Workbench was available when the project was received. Pizza is a language which extends Java with additional data structures, the most useful of which are the algebraic data types and parametric polymorphism offered by the language and significantly simplifies the implementation. However, and unfortunately, Pizza has stopped progressing and we need to re-implement the Pizza classes as pure Java classes.

3.2 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.

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++.

3.3 Heterogenous and homogenous translations

The definers of the Pizza language like translations so much that they had two of them: a heterogenous translation that produces a specialised copy of code for each type at which it is used, and a homogenous translation that uses 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.

3.4 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.

Example 1.1 Algebraic types in Pizza

```
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(_):
      ...
      ...
    }
  }
}
```

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 the Workbench 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 1.2. 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.

Example 1.2 Translation of algebraic types into Java

```
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;
  }
}
```

3.5 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. The ML Workbench 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 the Workbench 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 2.1.

```
Example 2.1 Polymorphism in Pizza

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);
```

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 2.2.

```
Example 2.2 Heterogenous translation of polymorphism in Pizza

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);
```

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 occurance 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 transation, and it is shown in Example 2.3.

```
Example 2.3 Translation of Polymorphism into Java

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());
```

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 2.3.

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.

Chapter 4

Transient Solutions

4.1 Overview

Not like long-run or stationary analysis, probabilities at an arbitrary time t can be obtained by transient analysis. The matrix exponentials computation package Expokit is a collection of routines written in the Fortran programming language. The package is capable of computing transient states of Markov chains. The backbone of the routines consists of matrix-free Krylov subspace projection methods(Arnoldi and Lanczos processes), and that is why the toolkit is capable of coping with sparse matrices of large dimension. Output from the Pizza Workbench is not directly compatible with this form of matrix representation. The detection of the steady-state is made possible by the *happy breakdown*. *Happy breakdown* is the point after which the probabilities become steady.

Problem Definition Continous-time Markov Chains (CTMC) are widely used in the analysis of the behaviour of many physical systems. Given a system whose evolution can be modelled by an irreducible and homogeneous CTMC, $X = \{X(t), t \geq 0\}$ having a finite discrete state space $\Omega = \{1, 2, ..., N\}$, the transient analysis involves the computation of the state probability vector $\pi(t) = (\pi_1(t), \pi_2(t), ..., \pi_N(t))$ where $\pi_i = Prob\{X(t) = i\}$ is the probability that the Markov process X is in state i at time t. Interesting measures (availability, reliability, performability) of the behaviour of the system being modelled can then be evaluated as weighted sums of state probabilities. Let $Q = [q_{ij}]$ be the infinitesimal generator of X. Q is also called the transition rate matrix and $q_{ij} \geq 0, i \neq j$, is the rate of transition from state i to state j and $q_{ii} = -\sum_{i \neq j} q_{ij}$

The evolution of the state probability vector $\pi(t)$ can be mathematically represented by the Chapman-Kolmogorov system of differential equations.

For convenience of notation we set

$A = Q^T, v = \pi^T(0), w(t) = \pi^T(t)$

and hence, the differential system under concern may be written as

$$\begin{cases} \frac{dw(t)}{dt} = Aw(t), t \in [0, T] \\ w(0) = v, \text{initial state probability vector} \end{cases} \tag{4.1}$$

and has the analytic solution

$$w(t) = e^{tA}v. \tag{4.2}$$

Largeness, stiffness and accuracy are the three major difficulties related to the practical computation of (4.2).

Intrinsic Properties By construction A is essentially nonnegative and in addition there are a number of inherent characteristics of the Markov model. We give below a list of those which are relevant for our presentation.

1. $\alpha^T A = 0$ where $\alpha = (1, 1, ..., 1)^T$.
2. $0 \neq w_i(t) \neq 1$ and $||w(t)||_1 = \alpha^T w(t) = 1$.
3. 0 is a simple eigenvalue of A and for non-null eigenvalue λ of A, $\Re_e(\lambda) < 0$. Hence the differential system (4.1) is stable and has a steady state solution.

It follows from 3 that if ω denotes the eigenvector of A associated with the eigenvalue 0, i.e., $A\omega = 0$ then, we have $\omega = \lim_{t\to\top} e^{tA}v$. the vector ω is known as the stationary probability vector, i.e., ω is the probability that the Markov chain will be at state i at statistical equilibrium.

Notation Throughout this chapter the integer m is some prescribed constant which should not exceed N, the order of the principal matrix A. We set $\rho = ||A||_2$.

The organisation of the chapter is as follows. In section 4.2 we consider the motivations of a time-stepping integrator for the problem. Section 4.3 presents the basic Krylov subspace approximation method and some extensions. Section 4.4 discusses the preservation of probability constraints and Section 4.5 considers numerical issues.

4.2 Integration Scheme

One way to solve (4.1) consists of computing the matrix exponential e^{tA}. Unfortunately this can be difficult. Moreover, for large sparse matrices, the exponential becomes full and little can be done to prevent fill-in. Among current methods, the Taylor expansions have been at the basis of most solution techniques used in Markov chains analysers. This is the case for the Standard Uniformisation (SU) method which is probably the most widely used. Another approach is referred to as the Uniformized Power(UP) method which provides a new mechanism for the control of the error was found to be an improvement over the SU method in time complexity for a class of stiff Markov processes. But it requires the full matrix and it not intended for large systems. It is more promising to tackle the computation of the matrix exponential times a vector rather than the matrix exponential in isolation. For our concern, a consideration is that, system managers are generally interested in discrete observations

$$w(t_i)_{0 \neq i \neq s+1}, 0 = t_0 < t_1 < ... < t_s < t_{s+1} = T \tag{4.3}$$

where the observation times $t_{i0 \neq i \neq s+1}$ may or may not be uniformly spaced and may or may not be known in advance. If we take $w_i = w(t_i)$ and $\tau_i = ti + 1 - t_i$ then, we consider the integration scheme

$$\begin{cases} w_0 = v \\ w_{i+1} = e^{\tau_i A} w_i, i = 0, 1, ..., s \end{cases} \tag{4.4}$$

so that the kernel of the problem remains the operation $e^{\tau A}v$. The accuracy of the matrix exponential depends heavily on the matrix norm and for an arbitrarily large value of t, a straightforward computation of $e^{tA}v$ is usually unsatisfactory. However if we use a step-by-step procedure, the computation at every step can be monitored to be of acceptable accuracy. In this way we make possible the integration process as in a regular single-step ODE solver.

4.3 Krylov Approximation

In this section, to simplify the notation, we first drop the time variable and we will later re-insert it in the proposed schemes.

4.3.1 Arnoldi-based Schemes

We begin by examining the basic Krylov subspace approach to computing $\pi(t_{i+1}) = \pi(t_i)e^Q(t_{i+1} - t_i)$. For ease of notation, we shall set $A = Q^T$, $v^T = \pi(t_i)$,

$w^T = \pi(t_{i+1})$, and $\tau_i = t_{i+1} - t_i$ and consider the computation of $w = e^A v$. Inserting the time constant τ_i into the computation to get $e^{A\tau_i}v$ poses no additional problem.

Our objective is the computation of an approximation of the form $p_{m-1}(A)v$ to the matrix exponential operation $e^A v$, where $p_{m-1}(A)$ is a polynomial of degree $m-1$ in A. Notice that $p_{m-1}(A)$ is a linear combination of the vectors $v, Av, ..., A^{m-1}v$, i.e., it is an element of the Krylov subspace

$$K_m(A, v) = span\{v, Av, ..., A^{m-1}v\}$$

and so our problem may be posed as that of finding the element of $K_m(A, v)$ that best approximates $w = e^A v$. If "best" is taken to mean in the least squares sense, then the best approximation ϖ to w in the Krylov subspace satisfies

$$||\varpi - w||_2 = \min_{x \in K_m(A,v)} ||x - w||_2 = \min_{y \in \Re^m} ||V_m y - e^A v||_2,$$

where $V_m = [v_1, v_2, ..., v_m]$ is a set of basis vectors for $K_m(A, v)$. Since this is a full-rank least squares problem, we essentially solve a system of linear equations and obtain $\varpi = e^A v$. If we choose $v_1 = v/\beta$, where $\beta = ||v||_2$, then $v = \beta V_m e_1$ and we have

$$e^A v \approx \varpi = V_m V_m^{-1} V_m^{-T} V_m^T e^A v = V_m[(V_m^T V_m)^{-1} V_m^T e^A v] = V_m[(V_m^T V_m)^{-1} V_m^T e^A V_m]\beta e_1 \tag{4.5}$$

Let us at this point turn our attention to the computation of basis vectors V_m. One convenient means of doing so is the Arnoldi procedure as follows:

1. Set $v_1 = v/||v||_2$.
2. For j = 1,2,...,m do
   - Compute $z = Av_j$.
   - For i = 1,2,...,j do
     o Compute $h_{ij} = v_i^T z$
     o Compute $z = z - h_{ij} v_i$.
   - Compute $h_{j+1,j} = ||z||_2$ and $v_{j+1} = z/h_{j+1,j}$.

Figure 4.1: Arnoldi Process

Since this algorithm implements a modified Gram-Schmidt orthogonalisation procedure, the computed basis V_m is orthonormal. The $m \times m$ upper Hessenberg matrix H_m formed from the coefficients h_{ij} satisfies the relationship

$$AV_m = V_m H_m + h_{m+1,m v_{m+1}} e_m^T, \tag{4.6}$$

which, from the orthogonality of V_m, yields

$$H_m = V_m^T A V_m.$$

In words, H_m represents the projection of the linear transformation A onto the subspace $K_m(A, v)$ with respect to the basis V_m.

Returning to our approximation ϖ in equation (4.5), the orthonormality of the basis implies that

$$\varpi = \beta V_m (V_m^T e^A V_m) e_1.$$

However, this does not as yet appear to be very helpful, since it still requires e^A; the purpose of the Krylov subspace approach, namely to project the exponential of a large matrix approximately onto a small Krylov subspace, has yet to be achieved. This purpose is accomplished if we approximate $V_m^T e^A V_m$ by $e^{V_m^T A V_m}$, that is, if we replace the restriction of the exponential operator e^A onto the subspace $K_m(A, v)$ with respect to V_m This gives the approximation

$$\varpi \approx \varpi = \beta V_m e^{H_m} e_1, \tag{4.7}$$

which still involves the evaluation of the exponential of a matrix, but this time of small dimension (m instead of n) and of a particular structure (upper Hessenberg).

Since $V_m^T (A\tau) V_m = H_m \tau$, and the Krylov subspaces associated with A and $A\tau$ are identical, we have

$$e^{A\tau} v \approx \beta V_m (V_m^T e^A V_m) e_1.$$

for an arbitrary scalar τ. This justifies our earlier assertion that the time constant can be incorporated without difficulty. We shall therefore, for the sake of simplicity, continue to omit it. In fact, our only concerns with the parameter τ arise in the computation of $e^{H_m \tau}$, for which we have already discussed appropriate algorithms, and in the size of $||A\tau||$.

The alert reader at this point is perhaps becoming increasingly concerned by the apparent piling of approximation on approximation. Indeed, we make a first approximation in extracting ϖ from a subspace of size m (instead of size n), a second in forming ϖ by replacing $V_m^T e^A V_m$ with $e^{V_m^T A V_m}$, and a third in recommending rational Pade approximants for the computation of e^{H_m}. However, we have

$$||e^A v - \beta V_m e^{H_m e1}||_2 \leq 2\beta \frac{\rho^m e^\rho}{m!},$$

where $\rho = ||A||_2$ and $\beta = ||v||_2$. Therefore, the approximations give fairly accurate results even for small-dimension subspaces, as long as $||A||_2$ is not large. For example, choosing a time step τ to give $||A\tau||_2 < .5$ and using a subspace of size 12 should give more than 10 decimal places accuracy.

To summarise, therefore, the basic Krylov subspace approach is to

- Use the Arnoldi process to construct the orthonormal basis V_m and the upper Hessenberg matrix H_m,
- Use rational Pade approximates to compute $e^{H_m\tau}$,
- Form an approximate solution as $\beta V_m e^{H_m\tau} e_1$

The only interaction with the (usually large and sparse) matrix $A(= Q^T)$ is the matrix vector multiply in the first line of step 2 of the Arnoldi process. Thus compact storage schemes may be conveniently implemented. However, to develop an industrial-strength production code, it is necessary to go beyond this basic implementation. A number of important questions still need to be answered, particularly those relating to error control, step size, and choice of size of subspace.

4.3.2 Corrected Schemes and Error Bounds

Notice that the Arnoldi process generates $m+1$ vectors, $v_1, v_2, ..., v_{m+1}$, but that only m of these are actually used. We now show how this additional vector may be used to obtain an improved approximation at very little extra cost. Consider the function

$$\phi(z) = \frac{e^z - 1}{z}$$

and observe that

$$e^A v = v + (e^A - I)v = v + A\phi(A)v.$$

Since $\phi(A)v \approx V_m V_m^T \phi(A) V_m \beta e_1$, we approximate $\phi(A)v$ by

$$\phi(A)v \approx V_m \phi(H_m) \beta e_1.$$

Let us define

$$s_m = \phi(A)v_1 - V_m\phi(H_e)e_1.$$

Then, since $v = v_1\beta$, $s_m\beta$ is the error in this approximation of $\phi(A)v$. We have

$$e^A v_1 = v_1 + A\phi(A)v_1 = v_1 + A[V_m\phi(H_m)e_1 + s_m].$$

Substituting for AV_m from equation (4.5), we get

$$\begin{aligned} e^A v_1 &= v_1 + (V_m H_m + h_{m+1,m}v_{m+1}e_m^T\phi(H_m)e_1 + As_m \\ &= V_m[e_1 + H_m\phi(H_m)e_1] + h_{m+1,m}e_m^T\phi(H_m e_1 v_{m+1} + As_m) \end{aligned} \tag{4.8}$$

from which it immediately follows that

$$e^A v = \beta e^A v_1 = \beta V_m e^{H_m}e_1 + \beta h_{m+1,m}e_m^T\phi(H_m)e_1 v_{m+1} + \beta As_m.$$

The first term on the right-hand side is the approximation obtained by the straightforward Krylov subspace method, equation (4.7). If instead, we use the approximation

$$e^A v \approx \beta[V_m e^{H_m}e_1 + h_{m+1,m}e_m^T\phi(H_m)e_1 v_{m+1}]$$

we get a corrected scheme, and the error in this scheme is given by ϕA_{s_m}. Setting

$$\widehat{\mathbf{H}}_{\mathbf{m+1}} = \begin{pmatrix} H_m & 0 \\ h_{m+1,m}e_m^T & 0 \end{pmatrix}$$

the corrected scheme becomes equivalent to the approximation

$$e^A v \approx \beta V_{m+1}e^{\widehat{H}_{m+1}e_1} \tag{4.9}$$

and has a computational cost close to that of the original. We have the following a priori error bound:

$$||e^A v - \beta V_{m+1}e^{\widehat{H}_{m+1}e_1}||_2 \le 2\beta\frac{\rho^{m+1}e^\rho}{(m+1)!}$$

Continuing in this fashion, k-corrected schemes may be developed. It is shown in [1] that the following expansion holds:

$$
\begin{aligned}
e^A v &= \beta V_m e^{H_m} e_1 + \beta h_{m+1,m} \sum_{j=1}^{k} e_m^T \phi_j(H_m) e_1 A^{j-1} v_{m+1} + \beta A_k s_m^k \\
&= \beta V_m e^{H_m} e_1 + \beta h_{m+1,m} \sum_{j=1}^{\top} e_m^T \phi_j(H_m) e_1 A^{j-1} v_{m+1} \qquad (4.10)
\end{aligned}
$$

where $s_m^k = \phi_k(A)v_1 - V_m \phi_k(H_m)e_1$ and the functions $\phi_k(z)$ are defined recursively as

$$\phi_0(z) = e^z \phi_k(z) = \frac{\phi_{k-1}(z) - \phi_{k-1}(0)}{z}; k \geq 1.$$

Note that $\phi_1(z) = \phi(z)$. The error in the k-corrected scheme satisfies

$$||\beta A^k s_m^K||_2 \leq 2\beta \frac{\rho^{m+k} e^\rho}{(m+k)!}.$$

4.4 Markov Model Constraints

Until now we have scarcely mentioned the requirements of the Markov model on the computed approximations. Indeed there is no mathematical statement which insures that the computed solution vector is still a probability vector, i.e., satisfying property 2. What we can assert is that the corrected scheme (4.9) satisfies

$$\alpha^T e^A v = 1. \qquad (4.11)$$

This follows immediately from property 1 and fact that

$$e^A v = p_m(A) v$$

where $p_m(z)$ interpolates e^z on $\lambda(H_m) \cup 0$ in the Hermite sense and thus $p_m(z) = 1 + z q_{m-1}(z)$ with $q_{m-1}(z)$ interpolating $\phi_1(z)$ on $\lambda(H_m)$ in the Hermite sense. There is no similar result for the basic scheme. Therefore we will exclusively use the corrected approximation (possibly of a higher order than 1).

The remaining consideration is to ensure that the components of $e^A v$ are all non-negatives. Unfortunately this is not always guaranteed. It is shown in [2] that it is always possible to select a step-size small enough such that the approximation vector of the integration process (4.4) is nonnegative.

In practice, if we end up with a nonpositive approximation vector, the components under concern are, in magnitude, less than the error tolerance for otherwise, the approximate solution would have been rejected. Therefore a simple strategy to overcome any difficulty is to set the negative components to zero and afterwards, perform a normalisation. Another strategy is of course to reduce the step-size.

4.5 Numerical Issues

4.5.1 Accuracy and Roundoff Errors Analysis

We will now derive an upper bound for the global truncation error and an upper bound for roundoff errors when we use an integration scheme of the form (4.4). Given a vector w_{i-1} and a step-size τ_{i-1}

- w_i denotes the exact local solution, i.e., the value of $e^{\tau_{i-1}A}w_{i-1}$.
- $\widetilde{w}_i$ denotes the exact solution of the approximation method, i.e. the value of the proposed Krylov method in exact arithmatic.
- $barw_i$ is the computed value of $\widetilde{w}_i$.

We start with the following preliminary result [2].

- Let A be any essentially nonnegative matrix satisfying property 1 and let $\tau \geq 0$. For any $w \in \Re^N$ such that $w \geq 0$ we have $||e^{\tau A}w||_1 = ||w||_1$. More generally for any $w \in \Re^N$ we have $||e^{\tau A}w||_1 \leq ||w||_1$ and therefore $||e^{\tau A}||_1 = 1$.
- Given a starting approximation $\widetilde{w}_0$ of w_0, let $\widetilde{\varepsilon_0} = w_0 - \widetilde{w}_0$ and for $j = 1, ..., j$, let $\widetilde{\varepsilon_j}$ be the local truncation error vectors, i.e., $\widetilde{\varepsilon_j} = e^{\tau_{j-1}A}\widetilde{w}_{j-1} - \widetilde{w}_j$ then, at step i

$$||w_i - \widetilde{w}_i||_1 = ||\sum_{j=0}^{i} e^{t_i - t_j} A_{\widetilde{\varepsilon_j}}||_1 \tag{4.12}$$

$$\leq \sum_{j=0}^{i} ||\widetilde{\varepsilon}_j||_1. \tag{4.13}$$

Similarly, if we set $\bar{\varepsilon}_0 = w_0 - \bar{\varepsilon}_0$ and if we define, for $j = 1, ..., i$, the sequence $\widetilde{\varepsilon_j} = e^{\tau_{j-1}A}\widetilde{w}_{j-1} - \widetilde{w}_j$, i.e., more specifically, $\bar{\varepsilon}_j$ measures the deviation between the actual exact solution and the actual computed approximation then, we also obtain

$$||w_i - \bar{w}_i||_1 = ||\sum_{j=0}^{i} e^{t_i - t_j} A_{\bar{\varepsilon}_j} \tag{4.14}$$

$$\leq \sum_{j=0}^{i} ||\widetilde{\varepsilon}_j||_1. \tag{4.15}$$

We see that there is no amplification on the errors. Thus the integration process is confidently stable. We point out that those results are valid for any integration scheme used for the transient solution of a CTMC.

4.5.2 Step-size Control

In this section we give some practical guidelines on how to handle the step-size τ and m, the size of the Krylov basis. In fact at each step, for a prescribed tolerance there exists optimal values $\tau_{min} \leq \tau_{opt} \leq \tau_{max}$ and $m_{min} \leq m_{opt} \leq m_{max}$ which minimise rounding errors and execution cost with respect to the whole integration interval. τ_{min} and τ_{max} (resp. m_{min} and m_{max}) are smallest and biggest allowable stepsize (resp. size of the basis). Unfortunately those optimal values cannot be obtained in a straightforward manner and we must be satisfied with some heuristics as is usual in ODE solvers. We base the control on the measure

$$\epsilon_i = ||\bar{\varepsilon}_i||/\tau_{i-1}, \text{error per unit step(EPUS)}$$

where $||\bar{\varepsilon}_i||$ has the same meaning as in (4.15) and is to be approximated by err. If tol denotes the prescribed tolerance to be achieved, the first step-size is chosen to satisfy the theoretical bound and within the integration process, the step-size is selected by means of the formula

$$\tau_i = \gamma(\frac{tol}{\epsilon_i})^{1/r}\tau_{i-1} \tag{4.16}$$

rounded to 2 significants digits to prevent numerical noise. The value of r is $m-1$ or m if the error estimate err comes from err1 or err2 respectively. The scalar γ is a safety factor which is intended to reduce the risk of rejection of the step. A step is rejected if $\epsilon_{i+1} > \delta tol$. Classical values of γ and δ are 0.9 and 1.2 respectively. The accumulated global error is upper bounded for a fixed integration time regardless of the number of steps used:

$$\sum_j ||\bar{\varepsilon_j}|| \leq \delta T tol$$

and according to (4.12) we infer that

$$||w_i - \bar{w_i}|| \leq \delta T tol \tag{4.17}$$

When a step is rejected the step-size may be reduced by using formula (4.16) once more but there is also the opportunity of adjusting the dimension of the Krylov subspace by way of the relation

$$m_i = m_{i-1} + \lceil \frac{log(tol/\epsilon_i)}{log\tau_{i-1}} \rceil. \tag{4.18}$$

Let us note that (4.18) allows us to increase or to decrease the size of the basis.

Chapter 5

The Software Structure and Sparse Storage

5.1 Overview

The Java PEPA Workbench did not change the structure of the Pizza PWB when re-implemented the Pizza language in Java, thus [3] is a good reference for the structure of the Java PWB. Here, distinct attention is assigned to transient solutions, which is obtained from the package Expokit. The format required to interface with this package is subtle, as the matrices are encoded in a Compressed Row Storage(CRS) mode.

5.2 Sparse Storage

The Compressed Row Storage (CRS) format puts the subsequent non-zeros of the matrix rows in contiguous memory locations. Assuming we have a non-symmetric sparse matrix , we create vectors: one for floating-point numbers (val), and the other two for integers (col_ind, row_ptr). The val vector stores the values of the nonzero elements of the matrix A, as they are traversed in a row-wise fashion. The col_ind vector stores the column indexes of the elements in the val vector. That is, if $val(k) = a_{i,j}$ then $col\_ind(k) = j$. The row_ptr vector stores the locations in the val vector that start a row, that is, if $val(k) = a_{i,j}$ then $row\_ptr(i) \leq k < row\_ptr(i+1)$. By convention, we define $row\_ptr(n+1) = nnz + 1$, where nnz is the number of non-zeros in the matrix A. The storage savings for this approach is significant. Instead of storing n^2 elements, we need only $2nnz + n + 1$ storage locations.

As an example, consider the nonsysmetric matrix A defined by

$$
\mathbf{A} = \begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix} \tag{5.1}
$$

The CRS format for this matrix is then specified by the arrays $\{val, col\_ind, row\_ptr\}$ given below

val	10	-2	3	9	3	7	8	7	3...9	13	4	2	-1
col_ind	1	5	1	2	6	2	3	4	1...5	6	2	5	6

row_ptr	1	3	6	9	13	17	20

If the matrix A is symmetric, we need only store the upper (or lower) triangular portion of the matrix. The trade-off is a more complicated algorithm with a somewhat different pattern of data access.

5.3 The Software Structure

The coding of transient solutions was done in Java by translating part of package Expokit (from Fortran 77) while using building blocks from the two well-known scientific libraries the Basic Linear Algeba Subroutines (BLAS) and the Linear Algebra Package (LAPACK). The BLAS and LAPACK packages have already been translated into Java (see http://www.cs.utk.edu/f2j/). As a precaution, necessary steps have been taken to ensure portability of the software. We include an essential minimal substitute to BLAS and LAPLACK. In this way the Java PWB is self-contained and can still operate even if these libraries that are not yet installed in the environment of the user.

Dmexpv.java computes $w = exp(tA)v$; t can be positive or negative. The action of the matrix exponential operator on the operand vector is evaluated directly, i.e., $exp(tA)$ is not computed in isolation before being applied to v. The variants dealing with symmetric/Hermitian matrices use the cheaper Lanczos process instead of the Arnoldi process. A check is done within this program to ensure that the resulting vector w is a probability vector.
IMPORTANT: The well-known transition rate matrix Q of a Markov chain satisfies $Q\alpha = 0$ where $\alpha = (1, ..., 1)^T$; Dmexpv uses the matrix-vector product $y = Ax = Q^T x$, i.e., the transpose of Q times a vector. Failure to remember this leads to wrong results.

```
                    Header of Dmexpv.java
-----Purpose----------------------------------------------------------|

---  DMEXPV computes w = exp(t*A)*v - Customised for MARKOV CHAINS.

     It does not compute the matrix exponential in isolation but
     instead, it computes directly the action of the exponential
     operator on the operand vector. This way of doing so allows
     for addressing large sparse problems.

     The method used is based on Krylov subspace projection
     techniques and the matrix under consideration interacts only
     via the external routine 'matvec' performing the matrix-vector
     product (matrix-free method).

     This is a customised version for Markov Chains. This means that a
     check is done within this code to ensure that the resulting vector
     w is a probability vector, i.e., w must have all its components
     in [0,1], with sum equal to 1. This check is done at some expense
     and the user may try DGEXPV which is cheaper since it ignores
     probability constraints.

     IMPORTANT: The check assumes that the transition rate matrix Q
                satisfies Qe = 0, where e=(1,...,1)'. Don't use DMEXPV
                if this condition does not hold. Use DGEXPV instead.
                DMEXPV/DGEXPV require the matrix-vector product
                y = A*x = Q'*x, i.e, the TRANSPOSE of Q times a vector.
                Failure to remember this leads to wrong results.

-----Arguments--------------------------------------------------------|

     n      : (input) order of the principal matrix A.

     m      : (input) maximum size for the Krylov basis.

     t      : (input) time at wich the solution is needed (can be < 0).

     v(n)   : (input) given operand vector.

     w(n)   : (output) computed approximation of exp(t*A)*v.

     tol    : (input/output) the requested acurracy tolerance on w.
              If on input tol=0.0d0 or tol is too small (tol.le.eps)
              the internal value sqrt(eps) is used, and tol is set to
              sqrt(eps) on output ('eps' denotes the machine epsilon).
              ('Happy breakdown' is assumed if h(j+1,j) .le. anorm*tol)

     anorm  : (input) an approximation of some norm of A.
   wsp(lwsp): (workspace) lwsp .ge. n*(m+1)+n+(m+2)^2+4*(m+2)^2+ideg+1
                                    +---------+-------+---------------+
              (actually, ideg=6)        V        H     wsp for PADE

   iwsp(liwsp): (workspace) liwsp .ge. m+2

     matvec : external subroutine for matrix-vector multiplication.
              synopsis: matvec( x, y )
                        double precision x(*), y(*)
              computes: y(1:n) <- A*x(1:n)
                        where A is the principal matrix.

              IMPORTANT: DMEXPV requires the product y = Ax = Q'x, i.e.
              the TRANSPOSE of the transition rate matrix.

     itrace : (input) running mode. 0=silent, 1=print step-by-step info

     iflag  : (output) exit flag.
              <0 - bad input arguments
               0 - no problem
               1 - maximum number of steps reached without convergence
```

```
               2 - requested tolerance was too high

-----Accounts on the computation-------------------------------------|
     Upon exit, an interested user may retrieve accounts on the
     computations. They are located in the workspace arrays wsp and
     iwsp as indicated below:

     location  mnemonic                 description
     -----------------------------------------------------------------|
     iwsp(1) = nmult, number of matrix-vector multiplications used
     iwsp(2) = nexph, number of Hessenberg matrix exponential evaluated
     iwsp(3) = nscale, number of repeated squaring involved in Pade
     iwsp(4) = nstep, number of integration steps used up to completion
     iwsp(5) = nreject, number of rejected step-sizes
     iwsp(6) = ibrkflag, set to 1 if 'happy breakdown' and 0 otherwise
     iwsp(7) = mbrkdwn, if 'happy brkdown', basis-size when it occured
     -----------------------------------------------------------------|
     wsp(1)  = step_min, minimum step-size used during integration
     wsp(2)  = step_max, maximum step-size used during integration
     wsp(3)  = x_round, maximum among all roundoff errors (lower bound)
     wsp(4)  = s_round, sum of roundoff errors (lower bound)
     wsp(5)  = x_error, maximum among all local truncation errors
     wsp(6)  = s_error, global sum of local truncation errors
     wsp(7)  = tbrkdwn, if 'happy breakdown', time when it occured
     wsp(8)  = t_now, integration domain successfully covered
     wsp(9)  = hump, i.e., max||exp(sA)||, s in [0,t] (or [t,0] if t<0)
     wsp(10) = ||w||/||v||, scaled norm of the solution w.
     -----------------------------------------------------------------|
     The 'hump' is a measure of the conditioning of the problem. The
     matrix exponential is well-conditioned if hump = 1, whereas it is
     poorly-conditioned if hump >> 1. However the solution can still be
     relatively fairly accurate even when the hump is large (the hump
     is an upper bound), especially when the hump and the scaled norm
     of w [this is also computed and returned in wsp(10)] are of the
     same order of magnitude (further details in reference below).
     Markov chains are usually well-conditioned problems.

----------------------------------------------------------------------------
```

Dgpadm.java computes the matrix exponential exp(tH) in full where H is a relatively small matrix. The underlying method is the irreducible rational Pade approximation to the exponential function combined with scaling-and-squaring. This procedure can be used in its own right to compute the exponential of a small matrix in full. Computational savings are made in the symmetric/Hermitian variants.

Dnchbv.java computes $exp(tH)y$ where H is a relatively small matrix using the partial fraction expansion of the uniform rational Chebyshev approximation of type (14,14) to the exponential function over the negative real axis. The calculation of each fraction is done using Gaussian elimination with pivoting. The Chebyshev method is intended to compute the direct action of the matrix exponential on a vector when H negative definite.

Dcmpac.java compacts the array ix and sorts ixx and xx (This is a gateway routine for DGCNVR) ...

Debug.java Class for producing debugging information, and so on.

Dgccsv.java computes y = A*x. A is assumed to be under the Compress Column Storage (CCS) format.

Dgcnvr.java converts a sparse storage format into another sparse storage format. The matrix can be rectangular.

```
                    Header of Dgcnvr.java

*-----Arguments------------------------------------------------------|

     from   : (input, character*3) the storage format holding the
              matrix on input. Accepted values are:
              'COO' : COOrdinates
              'CRS' : Compressed Row Storage
              'CCS' : Compressed Column Storage.

     to     : (input, character*3) the storage format holding the
              matrix on output. Same accepted values as above.

     diag   : (input, character*1) specifies whether or not the
              entire diagonal should be lodged, i.e., including
              null diagonal entries. (This may be required by
              some applications that make use of the locations
              of diagonal elements.)
              diag = 'N', no specific attention is given to diagonal
                          elements.
              diag = 'D', the entire diagonal is lodged, including
                          null elements on the diagonal.
              if from=to & diag='D', null diagonal entries are
              explicitly inserted in the actual matrix.

     nrow   : (input) number of rows in the matrix.

     ncol   : (input) number of columns in the matrix.

     nz     : (input/output) number of non-zeros elements.
              If diag='D' and null diagonal entries are inserted, then
              nz is updated on exit and contains the effective number
              of entries stored. In what follows, nz' (read nz prime)
              denotes the updated value of nz, nz <= nz' <= nz+n.
              If diag='N' then nz'=nz.

     ia(*) : (input/output) of declared length .ge. nz'.
             On input,
                if from='CRS', ia(1:nrow+1) contains pointers for the
                beginning of each row.
                if from='COO', or 'CCS', ia(1:nz) contains row indices.
             On output,
                if to='CRS', ia(1:nrow+1) contains pointers for the
                beginning of each row.
                if to='COO' or 'CCS', ia(1:nz') contains row indices
                in increasing order in each column.

     ja(*) : (input/output) of declared length .ge. nz'.
             On input,
                if from='CRS', ja(1:ncol+1) contains pointers for the
                beginning of each column.
                if from='COO' or 'CCS', ja(1:nz) contains col. indices.
             On output,
                if to='CRS', ja(1:ncol+1) contains pointers for the
                 beginning of each column.
                if to='COO' or 'CCS', ja(1:nz') contains column indices
                in increasing order in each row.

     a(*)  : (input/output) On input, a(1:nz) fits the input format
             and on output, a(1:nz') fits the output format.
```

```
iwsp(*) : (workspace) of declared length .ge. max(nrow,ncol).
-----------------------------------------------------------------------------
```

Dgcoov.java Computes y = A*x. A is passed via class file jlapack_rmat.java. A is assumed here to be under the COOrdinates storage format.
NOTE: This is an accessory to Expokit and it is not intended to be complete. It is supplied primarily to ensure an unconstrained distribution and portability of the package. The matrix-vector multiplication routines supplied here fit the non symmetric storage and for a symmetric matrix, the entire (not half) matrix is required. If the sparsity pattern is known a priori, it is recommended to use the most advantageous format and to devise the most advantageous matrix-vector multiplication routine.

Dgcrsv.java Computes y = A*x. A is passed via class file jlapack_rmat.java. A is assumed to be under the Compress Row Storage (CRS) format.

Idsrt1.java indirect sort – sort ix and carry xx along adapted from a SLAP (Sparse Linear Algebra Package) code.

Idsrt2.java indirect sort: sort ix and carry ixx and xx along adapted from a SLAP (Sparse Linear Algebra Package) code.

Sample.java Driver file for transient solutions.

Risto.java converts the Row Indexed Sparse Storage Mode [2] into Compressed Row Sparse Storage Mode. As described in 5.2, we use three arrays *val*, *col_ind* and *row_ptr* to specify a matrix in CRS mode. Array *val* stores the non-zero elements in the matrix, while array *col_ind* and array *row_ptr* provide the information for the position of each element in the matrix. As in RIS mode we need two arrays ija and *sa* to specify a matrix, with array ija stores all the non-zero elements and array *sa* specifies each element's position. This class file uses a loop to convert the RIS mode into CRS mode.

Chapter 6

The GUI and its facilities

6.1 Overview

The facilities of the GUI were slightly changed when we implemented it by using the JDK1.2's Swing API. [2] is still a good reference for the GUI and its facilities. Here, we will use an example model, tomp111.pepa, to illustrate the usage of the Java PWB.

6.2 Load a model

The file menu contains 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 be parsed automatically. A parser error is reported in the status box for syntactically incorrect models.

- Click on the file menu and click on the **New Model** option, an Open Dialog should be shown in Figure 6.1.
- Select a model file that wish to be analysed and click on the **open** button. Here, we select *tomp111.pepa* which will be parsed as shown in Figure 6.2.

6.3 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 detailed in [2]. If the modeller does not specify a solution the Numerical Recipes option is chosen by default. Here, we use the default option.

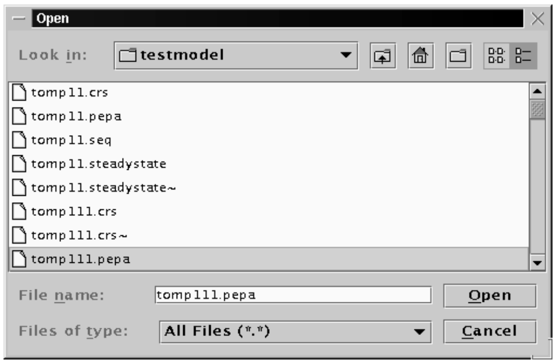

Figure 6.1: Open a model file

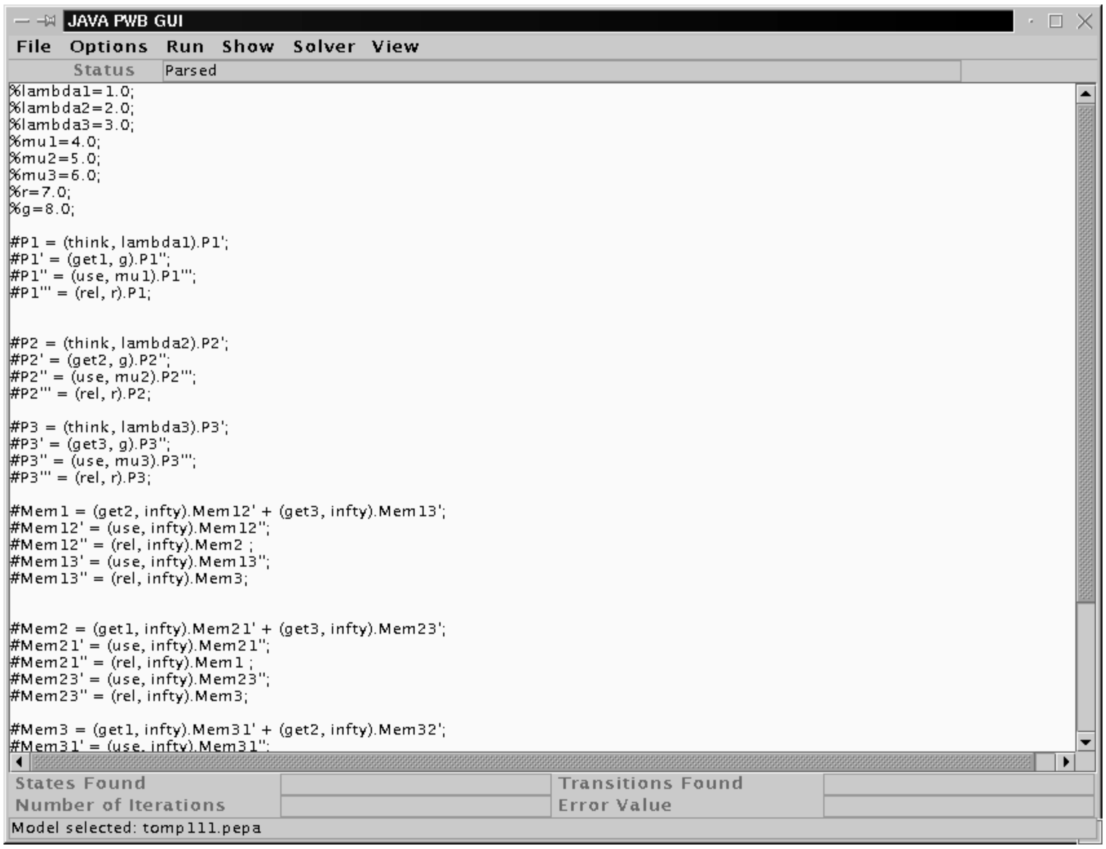

Figure 6.2: A model is parsed correctly

6.4 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 *modelname.seq* file is created containing the sequential derivatives of a model and their associated sequential state numbers.
CAUTION: The model must be run before analysis.

- Click on the Run Menu Bar, click on the **No States Output** option. The model will be run and no state output.
- Or click on the Run Menu Bar, click on the **To File - Compressed Format** option. The model will be run and states will be output to file *tomp111.states* in the compressed format.
- Or click on the Run Menu Bar, click on the **To Screen - Compressed Format** option. The model will be run and states will be output to the screen in the compressed format as shown in Figure 6.3.
- Or click on the Run Menu Bar, click on the **To File** option. The model will be run and states will output to file *tomp111.states*.
- Or click on the Run Menu Bar, click on the **To Screen** option. The model will be run and states will output to screen as shown in Figure 6.4.

6.5 Obtain steady and transient solutions

The long-run or stationary solutions are obtained by implemented the linear bi-conjugate gradient method [2]. Not like steady-state solutions, probabilities at an arbitrary time t can be obtained by transient analysis, which was implemented by using the matrix-free Krylov subspace projection methods(Arnoldi and Lanczos processes).

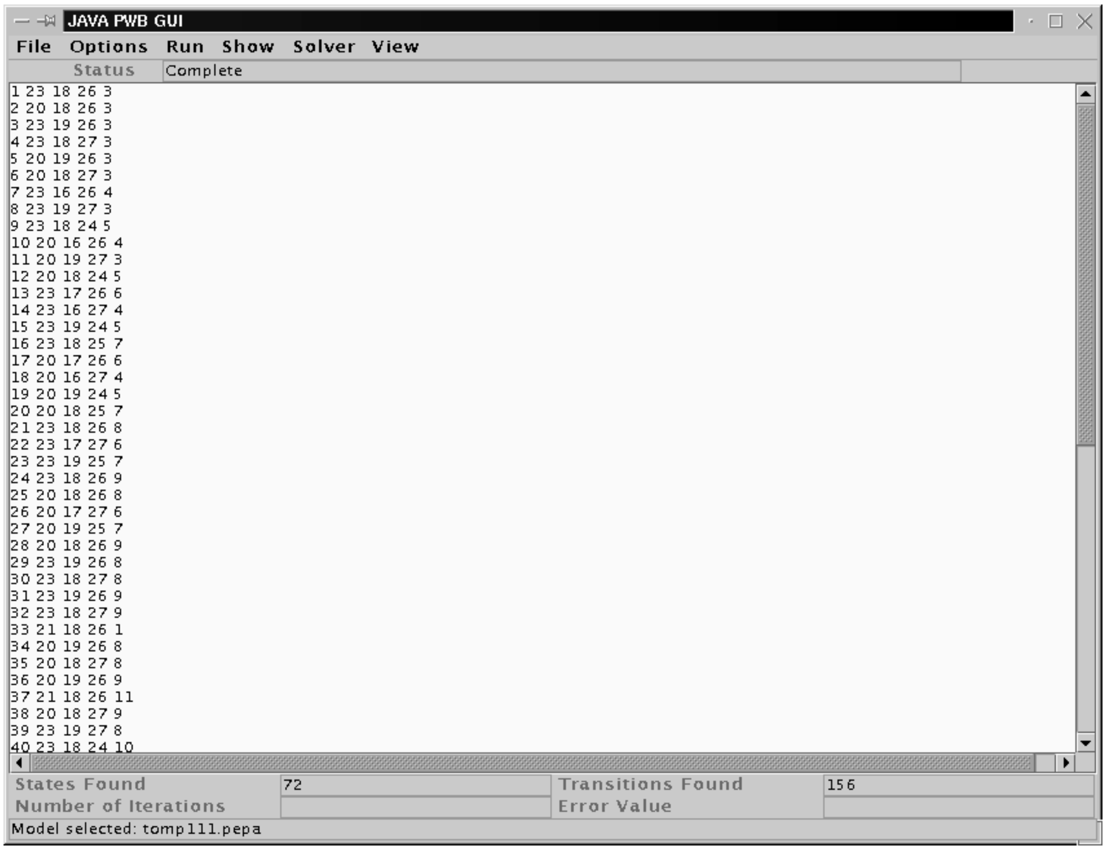

Figure 6.3: To Screen - Compressed Format

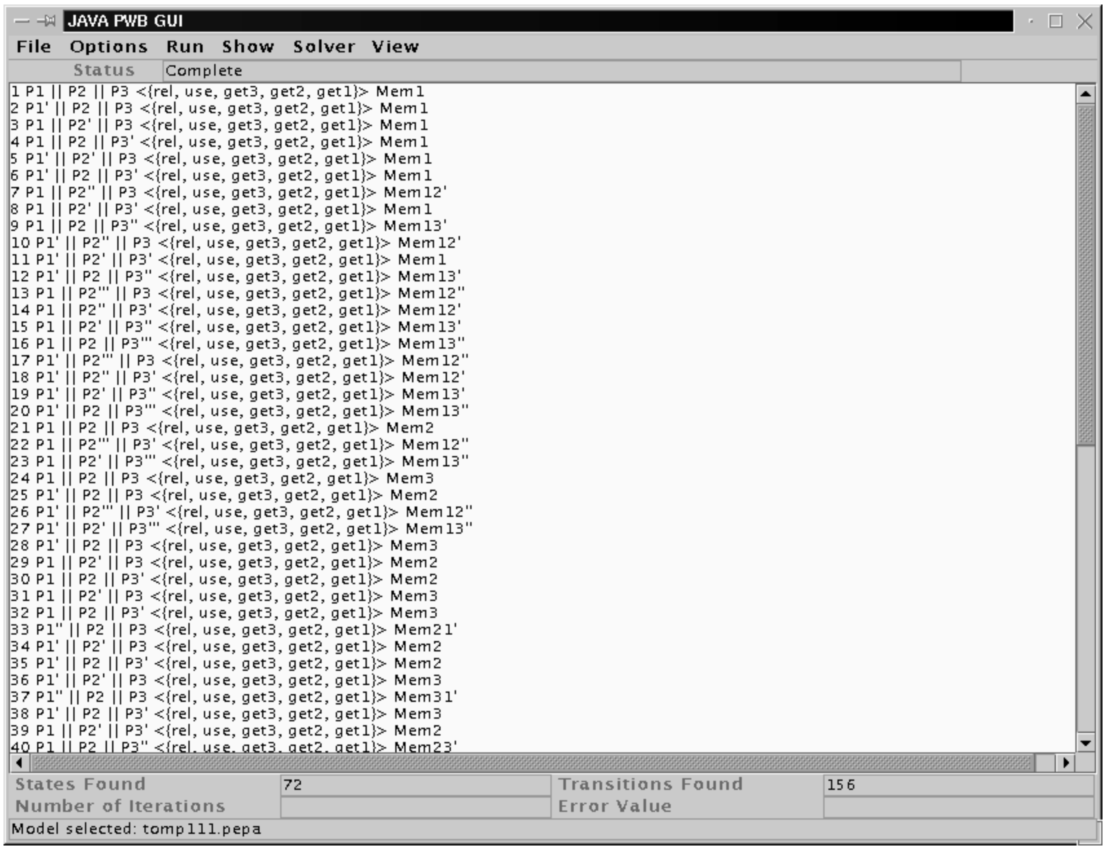

Figure 6.4: To Screen

6.5.1 Steady solutions

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.

- Click on the Solver Bar, and click on the **SolveSteadySolution** option. Stationary solutions will be obtained and the results will be output to file *tomp111.steadystate*. We can view the *tomp111.steadystate* file by using the View Bar as shown in Figure 6.6.

- Click on the Solver Bar, and click on the **Set SolveSteadySolution Parameters** option. A dialog will be shown as in Figure 6.5. User can change the parameters and click the **set** button to set them, then click the **SolveSteadySolution** option to obtain the Stationary solutions using the new values. And the results will be output to file *tomp111.steadystate* also.

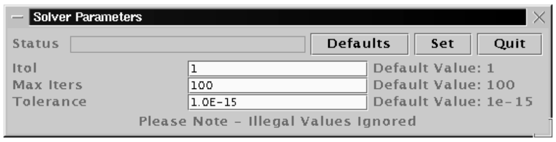

Figure 6.5: Set SolveSteadySolution Parameters

6.5.2 Transient solutions

Again, the user may either execute the solution tool or change the parameters used to estimate if an acceptable level of convergence has been obtained. We can change the time t to obtain the transient solutions at an arbitrary time t. Any exception will be reported in the status box.
CAUTION: The Krylov basis size must be smaller than the number of states of the model being analysed.

- Click on the Solver Bar, and click on the **SolveTransientSolution** option. Transient solutions will be obtained and the results will be output to file *tomp111.transientstate*. We can view the *tomp111.transientstate* file by using the View Bar as shown in Figure 6.8.

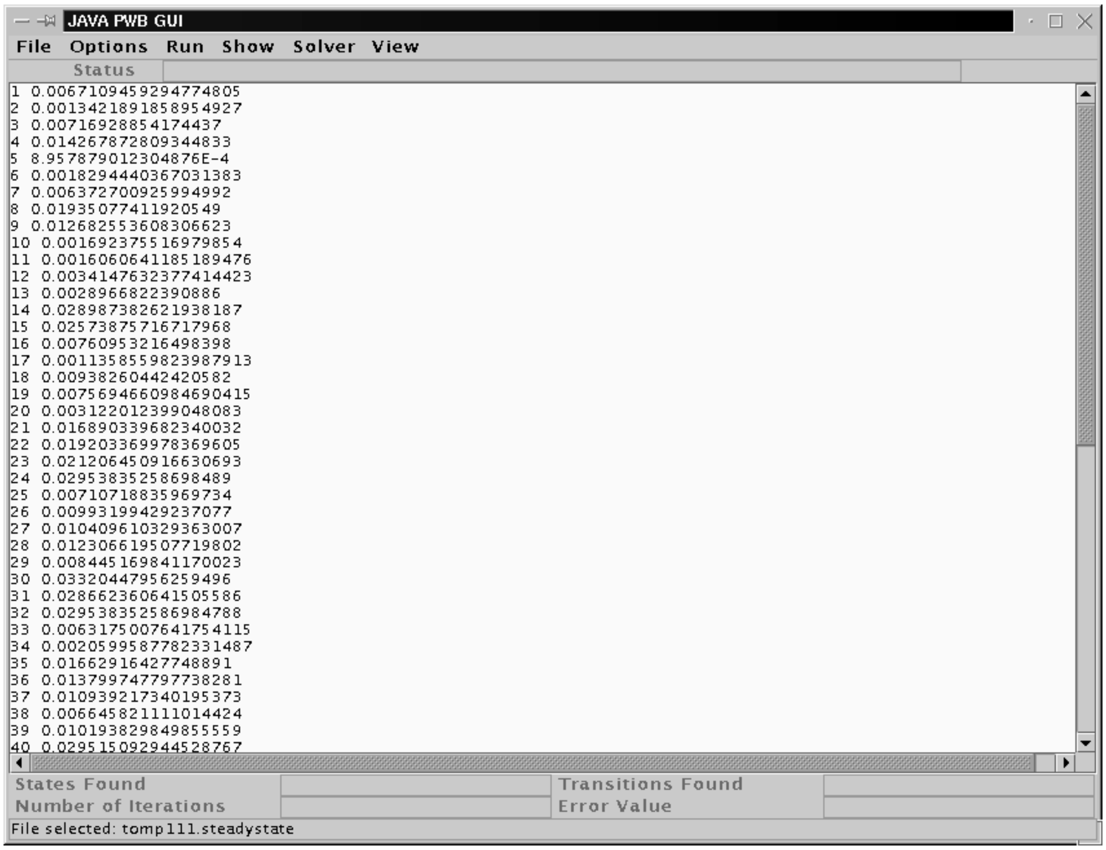

Figure 6.6: View file tomp111.steadystate

- Click on the Solver Bar, and click on the **Set SolveSteadySolution Parameters** option. A dialog will be shown as in Figure 6.7. User can change the parameters and click the **set** button to set them, then click the **SolveSteadySolution** option to obtain the Stationary solutions using the new values. And the results will be output to file *tomp111.transientstate* also.

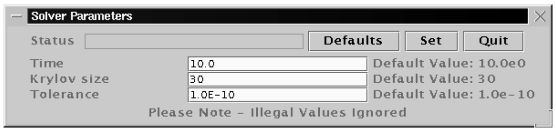

Figure 6.7: Set SolveTransientSolution Parameters

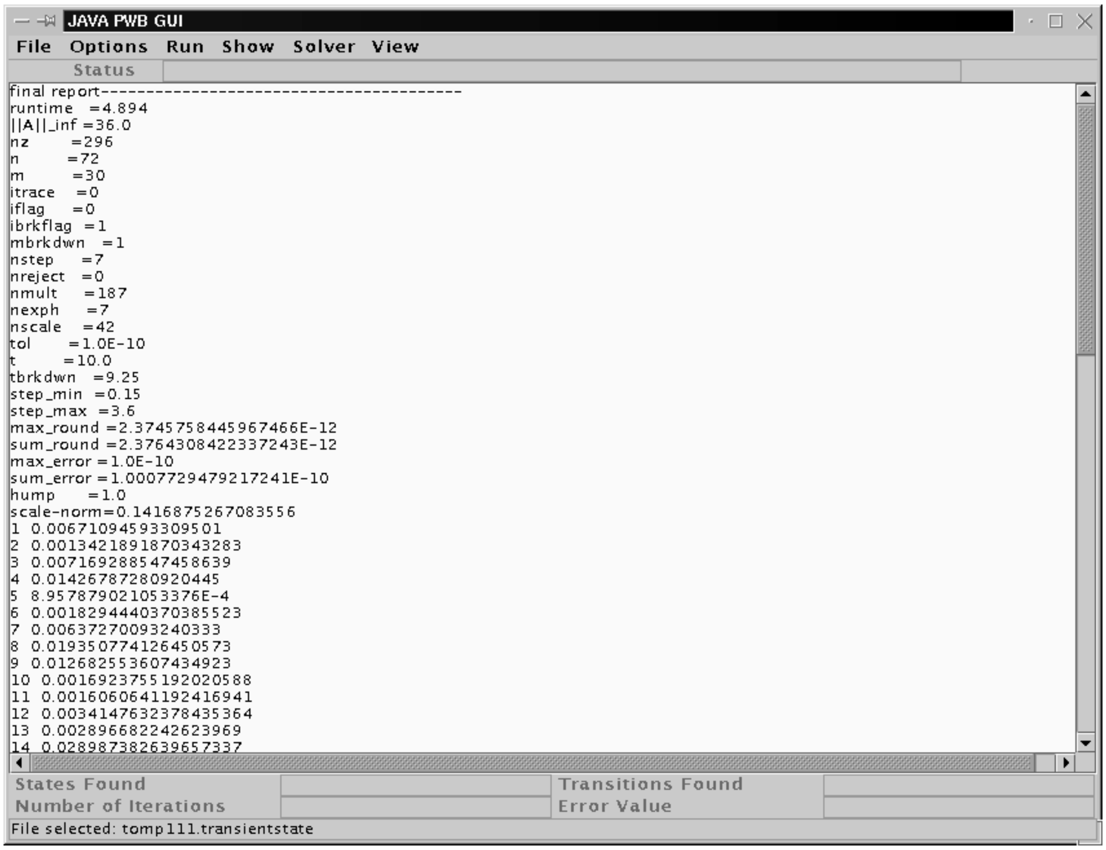

Figure 6.8: View file tomp111.transientstate

6.6 Use the utility

The utilities offered by the Java PWB are almost the same as the Pizza PWB. See [2] for the usage of the facilities. The figures are shown in turn as follows.

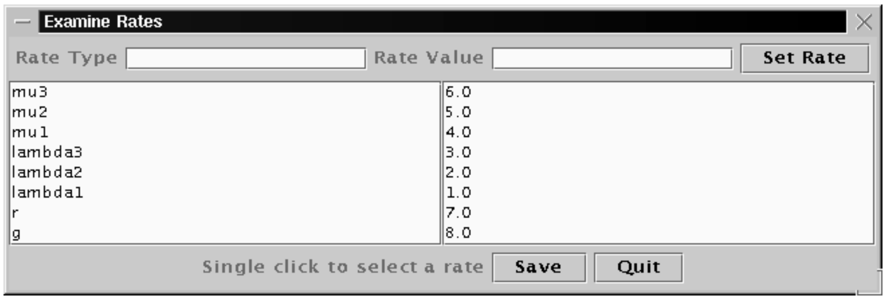

Figure 6.9: Examine Rates

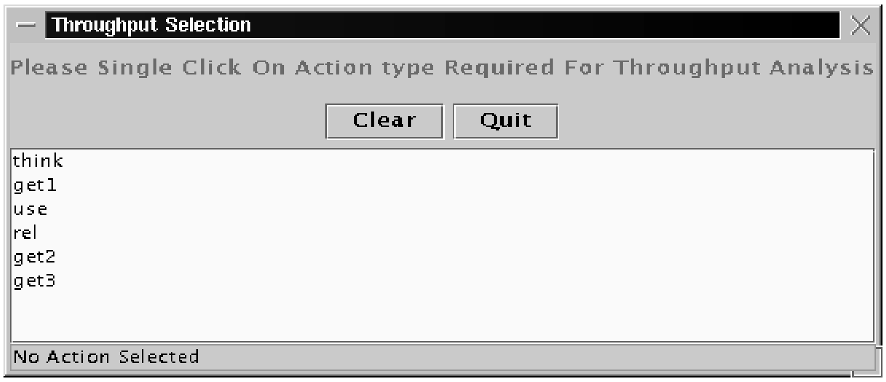

Figure 6.10: Throughput Selection

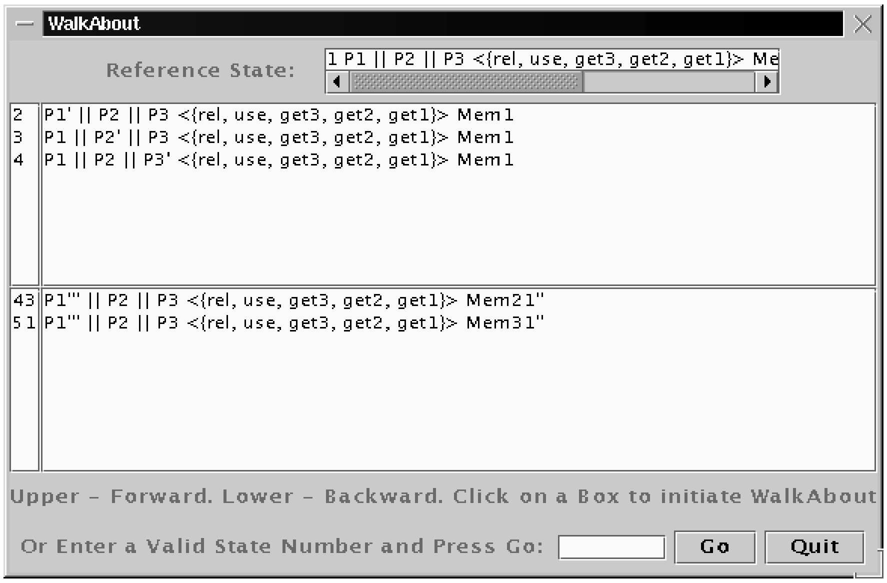

Figure 6.11: Walkabout

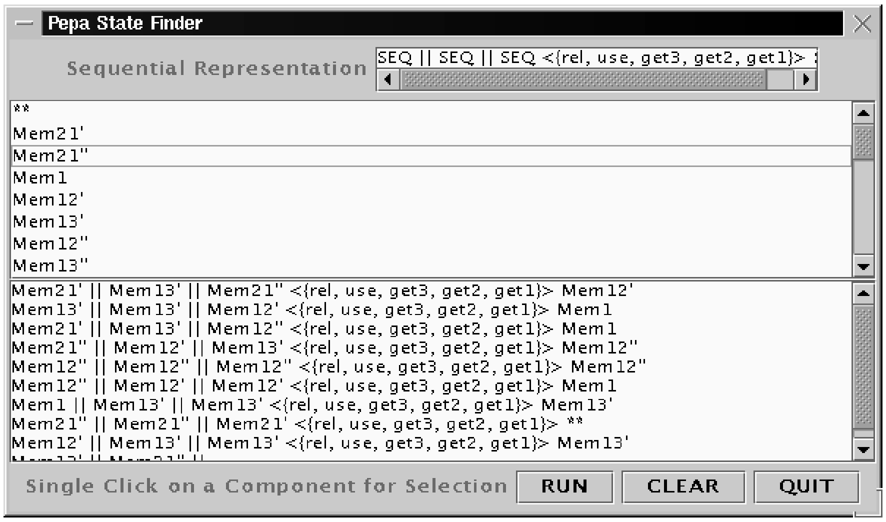

Figure 6.12: PEPA State Finder

Chapter 7

Results and Evalution

7.1 Overview

The Pizza PEPA Workbench is only capable of calculating the steady state probability distribution while the Java Workbench is also capable of obtaining the transient solutions. There is no necessity to compare the stationary analysis between the two Workbenches since Pizza compiles into the Java Virtual Machine. But some comparison of the steady solutions and transient solutions is not unreasonable. Transient solutions provide the probability distribution at an arbitrary time t, which would be very useful to the modeller some time. Back to back testing was performed to evaluate the run times of the two solutions. For the upper limit of state space see [2] for more details.

7.2 Experimental Details

The host workstation for all the experiments was a Linux machine with a Intel Celeron-MMX CPU at 400 MHz and 128 Mb of memory. A model of the TOMP machine protocol was used in the evalution of the two solutions [7]. This model can be scaled very simply by varying the number of copies of the processes which are competing for memory access. Thirteen different models were used in the evalution of the two solutions and all have a similar structure. The largest model, named tomp555.pepa is listed in Appendix A.

We tested the timings for output of transitions to the screen, steay solutions and transient solutions. The first of the right hand columns relates to the output of transitions, the second to the steady solutions and the third to the transient solutions.

Model	Processes	States	Trans	Output	Steady	Transient
1	1 P_1, 1 P_2, 1 P_3	72	156	0.173	0.058	6.951
2	2 P_1, 1 P_2, 1 P_3	176	480	0.423	0.104	7.293
3	2 P_1, 2 P_2, 1 P_3	416	1360	1.139	0.355	8.636
4	2 P_1, 2 P_2, 2 P_3	960	3648	2.867	0.877	14.647
5	3 P_1, 2 P_2, 2 P_3	2176	9408	7.624	2.136	20.805
6	3 P_1, 3 P_2, 2 P_3	4864	23552	19.428	5.347	41.835
7	3 P_1, 3 P_2, 3 P_3	10752	57600	49.519	11.807	87.958
8	4 P_1, 3 P_2, 3 P_3	23552	138240	121.713	27.092	212.170
9	4 P_1, 4 P_2, 3 P_3	51200	326656	301.46	65.927	436.334
10	4 P_1, 4 P_2, 4 P_3	110592	761856	791.700	134.398	903.847
11	5 P_1, 4 P_2, 4 P_3	237568	1757184	1879.841	283.733	-
12	5 P_1, 5 P_2, 4 P_3	507904	4014080	1848.290	-	-
13	5 P_1, 5 P_2, 5 P_3	1081344	9093121	4675.273	-	-

Table 7.1: Run Time Comparisons

7.3 Results

The outcome of the experiments is shown in Table 7.1. A hidden detail of the results is the poor IO performance of the Java language. Note the timings for the output of transitions is based on [2].

The Java Workbench succeeded in calculating the steady state solution for model number 11, while it was only able to obtain transient solution for model until number 10. This is because of the different sparse matrix storage modes used and the CRS mode in particular needs more memory than RIS. It also managed to generate the state space and output the transitions to the screen for 13th model.

7.4 Conclusions

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

- Rewrite the implementation in Java language.
- Add transient solutions to the PEPA workbench.
- Update the GUI using the facilities of Java1.2's Swing API.

The matrix exponentials computation package Expokit which was written in Fortran was re-implemented in Java first in order to add the transient solutions. It was found that the Java version provided more precision than the Fortran version because of the programming language property. Finally the Java version incorporates some extra useful features, like it is able to view a file on the Workbench described in Chapter 6.

7.5 Future Directions

Revisions of the Java Workbench will be ongoing in nature and do not warrant the tag further work. Some more conceptual ideas for further work are suggested.

7.5.1 Support for an Experimental Programme

A sensible upper bound would have to be set on the number of states in the state space to prevent memory exhaustion. This would give rapid feedback of the performance metrics of a model, particularly bottlenecks in the system, in a short time frame.

7.5.2 Show the Action Name

The walkabout facility provided by the Java Workbench 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. It would be useful to show the action names which cause the transitions from one state to another.

Bibliography

[1] William J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[2] J. Hunter. Re-evaluation of the PEPA Workbench. Master's thesis, School of Computer Science, The University of Edinburgh, September 1999.

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

[4] Roger B. Sidje. Expokit: A Software Package for Computing Matrix Exponentials. In *ACM Transactions on Mathematical Software*, Vol. 24, No. 1, March 1998, Pages 130-156.

[5] 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, *Proceedngs of 7th Conf. on Mod. Techniques and Tools for Computer Perf. Eval.*, volume 794 of LNCS, pages 353-368, 1994.

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

[7] Stephen Gilmore, Jane Hillston and Marina Ribaudo. An Efficient Algorithm for Aggregating PEPA Models. *IEEE Transactions on Software Engineering.* Available at www.dcs.ed.ac.uk/pepa.

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

Appendix A

Test Model

tomp555.pepa

```
%lambda1=1.0;
%lambda2=2.0;
%lambda3=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
```