Rapid Prototyping of High-Performance Concurrent Java Applications

K.J.R. Powell

Master of Science
School of Computer Science
School of Informatics
University of Edinburgh
2002

(Graduation date:December 2002)

Abstract

The design and implementation of concurrent applications is more challenging than that of sequential applications. The aim of this project is to address this challenge by producing an application which can generate skeleton Java systems from a high-level PEPA modelling language description. By automating the process of translating the design into a Java skeleton system, the result will maintain the performance and behavioural characteristics of the model, providing a sound framework for completing the concurrent application. This method accelerates the process of initial implementation whilst ensuring the system characterisics remain true to the high-level model.

Acknowledgements

Many thanks to my supervisor Stephen Gilmore for his excellent guidance and ideas. I'd also like to express my gratitude to L.L. for his assistance above and beyond the call of duty in the field of editing.

Declaration

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

(*K.J.R. Powell*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Principal Goal

The design and implementation of distributed and concurrent systems has proved problematical, as the interaction of multiple executing processes can lead to unexpected and unwanted behaviour, such as deadlock, live-lock and starvation. Creation of these systems is a far more error-prone process than the creation of sequential systems. The objective of this project is to develop a methodology and supporting tools that automate much of the process in order to remove some of the pitfalls associated with the design of higher performance concurrent and distributed systems.

One common technique is to use modelling languages to aid the design process as they allow a particular design to be tested and refined so as to deliver maximum performance whilst eliminating dangers such as deadlock. However, because there is usually no clear parallel between modelling language constructs and the facilities provided by programming languages, there is the danger that the design of the system may veer away from the design suggested by the model once actual implementation of the system begins. As a result, the possibility of reintroducing unwanted behaviour into the system arises.

Automating the process of the model's translation to a 'skeleton' implementation means the desirable performance and behavioural properties of the model are maintained. This working skeleton would provide a sound framework for completing the implementation of the system whilst ensuring the system characterisics remain true to the high-level model.

1.2 PEPA and Java

Performance Evaluation Process Algebra (PEPA) is a stochastic process algebra that uses probabilistic branching to model a system. It includes timing attributes for all system activities and uses race conditions to determine state transitions. This allows PEPA to evaluate performance as well as testing for correct functional behaviour in a model. It can model parallel composition and synchronisation behaviour. These features mean it is well suited as a high level design aid for concurrent and distributed systems.

Java is the target language because of its flexibility, its widespread adoption, its mobility between platforms and its rich set of standard libraries. Additionally, it has native support for multiple threading and remote method invocation. As an object oriented language, it allows us to take advantage of inheritance, making it simpler to run a PEPA model as a Java process whilst hiding all the underlying mechanics from the user.

Should the techniques developed prove valuable, it would be possible to re-use the methodology with similar PEPA-equivalent APIs for other programming languages. The Translator's output could then be modified to cater for these new APIs.

1.3 Project Objective

The aim of this project is to produce two things:

1. an *Application Programming Interface (API)* for running PEPA systems in Java, and

2. a *Translator* which takes as input any PEPA model and produces a set of Java classes that use the API to run with the same functional and performance characteristics as the PEPA models they are based on.

1. Design the system in PEPA;
2. Evaluate and refine the resulting model with the PEPA Workbench;
3. Translate the model to its PEPA2Java equivalent using the PEPA2Java Translator;
4. Confirm the Java skeleton is behaving correctly by running it with the PEPA2Java Simulator package enabled;
5. Flesh out the skeleton by implementing actual functionality in the Actions and replacing the probabilistic branching with deterministic behaviour;
6. Use the Simulator package to ensure the system is still running as it should— if it is not, return to stage 5 to fix the problem;
7. To complete implementation, switch from using the Simulator package to using the Barebones package.

Figure 1.1: A Process for the Rapid Prototyping of High-Performance Java Concurrent/ Distributed Applications

The *API* should be flexible and easy to use, whilst providing a clear parallel to the PEPA syntax so that manual modification, translation, and extension remain feasible. Additionally, it should not impose an excessive burden in terms of running cost.

The *Translator* should be able to successfully translate any PEPA model into clear, efficient Java code. As an added convenience it should be integrated into the Java PEPA Workbench.

These tools could then be used in the process described in figure 1.1 to facilitate an improved methodology for the design and implementation of concurrent and distributed systems.

Chapter 2

Performance Evaluation Process Algebra

The following two sections are based heavily on material from Jane Hillston's "A Compositional Approach to Performance Modelling" [Hillston, 1996]. It should give the reader a brief overview of PEPA and why it is useful as a modelling language and design tool.

The second part of the chapter is a brief overview of some of the existing PEPA tools that are relevant to this project.

2.1 Overview of PEPA and its role in design

Process Algebras are valuable in system design because a language like PEPA offers both qualitative (i.e. testing for deadlock) and quantitative (performance measurement) output from the same system description. Additionally, the hierarchical composition of PEPA models is familiar to system designers, whilst the *Component* and *Activity* constructs are similar to the object and method constructs of object-oriented languages.

A stochastic representation of the system may be generated from the derivation graph of the PEPA model. When activity rates are based on randomly-sampled exponential distributions, this stochastic model is equivalent to a continuous time Markov process, meaning the steady-state equilibrium (if one exists) can be found. In terms of performance analysis, this corresponds to the system's long-term "settled" behaviour, showing all the states that it will visit— deadlock is detected if the system gets 'stuck' in one state. Moreover, because we add timing in terms of rates, the mean residence time in each of these states is also calculated, enabling the discovery of any potential bottlenecks in the system.

A system is modelled in PEPA by splitting it into its *component* parts. The interaction of these components determines the behaviour of the system as a whole. The components may be atomic or define the behaviour of larger collections of components. Each component's behaviour is defined by the *activities* in which it may participate and how it determines which of those activities to execute. A compositional approach allows the system to be analysed at varying granularities depending on the degree to which components are broken into sub-components. The macro behaviour of the system will remain the same. The components represent the state of the system, and the activities represent the transitions.

Activities perform *actions* either individually or as part of a cooperating set. Each activity has an associated rate. The *rate* at which an action is carried out is determined by the rates of the participating activities. A rate may be *specified* as a sample from an exponential distribution or left unspecified and denoted $\top$ (TOP). $\top$ means the component is a passive participant in the Action. Passive participants rely on actively partipating components to execute the action.

Exponentially-distributed rates give the process algebra its probabilistic behaviour, making it suitable for performance modelling, and also allows activities to be *memoryless*. This memoryless property means that the time to the next event is independent from the time of the last event occurring. Therefore it can be used to model activities which *continue* their execution each time they are resumed after interruption as well as activities which *restart* their execution from scratch after each interruption.

Race conditions are used to determine the branching in the system— if multiple activities participate in an action, the one to finish first determines what happens next. All other activities are aborted. Similarly, if a component has a choice of activities, all are started. The first activity to complete determines the component's behaviour.

The random sampling means *any* activity may be the first to finish but the probability of an activity completing is given by the ratio of its rate over the sum of the rates of all the activities participating in the action. In the long term, this probability will determine the number of completed runs for each activity.

Unlike sequential systems, distributed and concurrent systems have multiple, simultaneously active, interacting processes. PEPA represents this cooperation of components in its *composition line*, defining synchronisation sets which specify which components must cooperate on which actions.

2.2 The Syntax of PEPA

The syntax of PEPA is given as:

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

The ASCII equivalents (used in this paper) are given in table 2.1. The functionality of the above constructs is given below.

Prefix: $(\alpha, r).\mathcal{P}$

Prefix is used to specify the behaviour of components. In the above example, the component carries out activity (α,r) which participates in action α at rate r. The component then behaves as component $\mathcal{P}$.

Choice: $\mathcal{P} + \mathcal{Q}$

A choice in PEPA means that the system may behave as either $\mathcal{P}$ or $\mathcal{Q}$. The current activities of $\mathcal{P}$ and $\mathcal{Q}$ are both enabled and whichever activity completes first determines whether the system will next behave as $\mathcal{P}'$ or $\mathcal{Q}'$.

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

The cooperation set L determines the interaction between components $\mathcal{P}$ and $\mathcal{Q}$. Specifically the set is made up of all those actions on which $\mathcal{P}$ and $\mathcal{Q}$ must *synchronise*— the

Prefix	(a, r).P
Choice	P + Q
Hiding	P / Q
Constant	#A = P
Synchronisation	P <L> Q
	(P <> Q or P || Q for parallel composition)

Table 2.1: ASCII equivalents of PEPA constructs

actions require the simultaneous participation of both components. *Shared actions* are made up of the activities of components $\mathcal{P}$ and $\mathcal{Q}$ which participate in the action.

Shared actions will complete at a rate determined by the slowest participant. Passive activities (i.e. those with rate $\top$) may be required participants but do not affect the action's rate.

Activities present in components $\mathcal{P}$ and $\mathcal{Q}$ whose actions do not appear is set L are termed *individual* activities and do not require synchronisation.

Parallel composition (where there is no cooperation between components) is denoted $\mathcal{P} \underset{\emptyset}{\bowtie} \mathcal{Q}$ or $\mathcal{P}||\mathcal{Q}$. In this case, both components run concurrently and independently.

Hiding: $\mathcal{P}/\mathcal{L}$

The component $\mathcal{P}/\mathcal{L}$ behaves as P except that any activities in set L are *hidden*— they participate in special action type τ, behaving as an individual activity. They are not externally observable and may not synchronise with other components.

Constant: $\mathcal{A} \stackrel{def}{=} \mathcal{P}$

Constants are used to define the behaviour of components. In the example above, we say that constant A is defined by the behaviour of component P. In this way, names are assigned to components so they can be referred to more easily elsewhere: for example in the system composition line or defining the behaviour that some other component will take on at the end of a prefix construct.

2.3 Existing Tools and Related Work

2.3.1 Java PEPA Workbench

The PEPA Workbench [Gilmore and Hillston, 1994, Hunter, 1999] is a tool used for the analysis of PEPA models. It can compute the underlying Markov process and the transition matrix of a model.

As detailed in the previous section, the steady-state can be calculated, giving relative frequencies of the system states. This information is written to a text file which can be analysed to discover the model's behavioural and performance characteristics.

Additionally, the "walkabout" tool can be used to step through a model manually to detect live-lock (i.e. when a model is stuck in a small loop, limited to a small subset of all the states) or to discover where a model is deadlocking.

The Workbench also includes a "simulation" tool [Fotis, 2001] which allows a model to be run as a SimJava simulation. Also provided is a facility allowing the simulation results to be compared to the steady-state solution.

As it stands, the Workbench is a tool for evaluating and exploring a model. By adding a translation function, it will enhance the tool's capabilities as a design aid, extending its use into the early implementation stages of a system.

2.3.2 Pepa2Ada

The Pepa2Ada translator [Gilmore et al., 1996] is a tool for producing PEPA-equivalent Ada programs. Although the paper describing the process was very useful in providing pointers and the possible pitfalls, Java is a very different language from Ada and so the approach taken was also different. For example, the problems that needed to be addressed regarding *internal choice* [Gilmore and Hillston, 1996] were not of relevance to this project. Also, using Java's inheritance means most of the mechanics of synchronisation and choosing can be hidden and should make it easier to extend an automatically generated program.

Chapter 3

Design

3.1 Overview and Design Decisions

The project plan is made up of six stages, of which implementation was always going to be the most time-consuming and challenging:

1. Orientation and Exploration– study related work, investigate the issues surrounding the project and identify possible designs.
2. Design– select and finalise a design then review it with the supervisor of the project.
3. Implementation– bring the design to realisation.
4. Review and Refinement– scrutinise the implementation and make any necessary adjustments and corrections.
5. Extension and Improvement– as far as time permits, take the project beyond its original scope by considering possible additions and enhancements.
6. Documentation– write the project report and review/ complete the commenting of the code.

In the event, difficulties and unforeseen obstacles relating to the implementation resulted in this phase absorbing a disproportionately large amount of the time and effort invested. The difficulties encountered are described in the next chapter. They arose mainly from the challenge of creating efficient and correct equivalents of PEPA constructs in Java.

Because this report has been produced at the end of the project, this design chapter describes the whole process of *ongoing design*. The design ideas were as much affected by the process of implementation and its changing needs as the implementation was determined by the initial ideas and designs. There is no clear split between design and implementation stages, so this chapter records some of the ideas (i.e. the concept of Scripts) that developed only as implementation progressed.

The implementation has been carried out in Java and in line with object-oriented design the different parts of the implementation were modularised as far as possible. Three things have been produced: the API (section 3.2), the Barebones and Simulator packages implementing the API (section 3.3) and the Translator (section 3.4).

3.1.1 Concurrent and Distributed Systems: Practicalities of Initial Implementation

The original title of the project was "Rapid Prototyping of High-Performance Distributed Java Applications". However, in order to focus on the core problem of ensuring a close fit of PEPA2Java to PEPA, an early decision was made to build a concurrent implementation of the API. The difference between a concurrent system and a distributed system is not large, especially if there is an API such as Java's Remote Method Invocation available. In essence, the main differences of a concurrent system are the more efficient communication between components and the sharing of a common memory. To reflect the shift of emphasis, the project has been re-titled "Rapid Prototyping of High-Performance Concurrent Java Applications".

3.1.2 Inherent Limitations: Deterministic and Probabilistic Behaviour

One clear limitation of translating from a PEPA model to an actual implementation is that of branching. In a model, the choice of which branch to execute is simplified. In the case of PEPA, it is determined by a race condition of randomly sampled, exponentially distributed rates. However, in an implemented system, branching is decided by the system's past choices— namely, it is deterministic.

For example, consider a system which is made up of a server and three clients. The server serves one client at a time, participating in a shared action, such as sending the client some requested data. In a PEPA system, which client is served at any one time is determined randomly. However, in a real system, client's requests will be triggered by some other action, such as a user's interactions.

The generated skeleton's behaviour will be probabilistic, rather than deterministic. This is not a problem as over the long term a client's requests may be adequately modelled by a random distribution. Therefore, performance measures will still be useful as long as rates are carefully chosen. However, the skeleton will be useful only as a starting point in implementation: one of the first things which will be done in implementation is to replace the probabilistic choice branching with deterministic branching.

In section 3.3.1 below, the determination of branching is discussed further.

3.1.3 Design Decision: PEPA Hiding

In consultation with the project supervisor, it was realised that whilst the *Hiding* construct may be valuable for the process of abstract model design and evaluation, its importance in an actual system implementation is not obvious.

Hiding is used to simplify complex models, so that some of the details of the underlying complexity may be hidden. The process of top-down implementation works in the opposite direction— adding the underlying complexity abstracted away in modelling languages (for example, PEPA's replacement of actions with delays and its use of probabilistic branching).

As PEPA2Java is a tool for speeding implementation, there is no need to include support for *Hiding*— the tool exists to aid in the process of adding the underlying detail, not to remove it. Accordingly, there is no equivalent for *Hiding* constructs in the API and the Translator will fail to translate any model including them.

3.2 The PEPA2Java API

3.2.1 Package independence

The heart of the project is the API for executing PEPA-equivalent commands in a Java program, which provides a clear, straight-forward manner to do this.

One of the benefits of the API is that it is general and flexible enough to work with different packages, each providing different functionality. For example, in the next section the two packages implemented for this project are introduced. The API-implementing packages can be used interchangeably to provide different functionality for executing a PEPA model.

In the same vein, if would be possible to create a PEPA2Java package which implements Remote Method Invocation. Then, any translated model could be run as either a simulation, a concurrent system or a distributed system. It should be possible to re-use a great deal of the current code if this were attempted. The algorithms for synchronisation and the form of the objects would remain the same, though some of the synchronisation code would need to be modified to reduce the number of method calls and sharing of data structures. There are more details on the actual API in the next two chapters.

3.2.2 Specified Objects and Methods

The final API defines *Component* and *Action* objects. Inheritance is used to *extend* these objects and make use of their methods to will provide PEPA-like functionality. *Components* also contain nested classes called *Scripts* which define their behaviour. Each *Component* is run as a separate Java *Thread* in the system, executing its scripts. Each script returns a reference to the next script which should be run.

In addition, a *Rate* class and *Activity* class are defined. The *Rate* class provides methods for accessing an exponentially distributed random number generator. The *Activity* class provides methods for a *Component* to interface with global and individual *Actions* (also defined as separated executing Threads).

Finally, a *PepaSystem* class is defined that creates the *Components* and their *Scripts*, the *Actions* and their *synchronisation sets*, initialises all these objects and starts the whole system running.

3.3 The Barebones and the Simulator Packages

These two packages are the concurrent system implementations of the PEPA2Java API. Their functionality (and their code) is for the most part identical, except that the Simulator package includes (as the name suggests) the ability to display the state of the system. This is useful for debugging purposes and discovering what is happening in the system. In order to keep the two packages working identically they are generated from the same source code, except that in the case of the Barebones package the debugging statements are disabled and the Simulator window class is removed. They therefore run identically but the Barebones package requires less overhead as there is no Simulator window to update.

All fields and methods are *protected* where appropriate to prevent the models from being able to disrupt the correct functioning of the underlying system.

3.3.1 Simulating the Race Condition

The race condition (see section 2.1) can be viewed as a form of speculative execution of multiple branches. There are two forms of branching in PEPA. Choice constructs are the obvious form. The more subtle form is the case where multiple Components are all vying for the chance to cooperate on some shared Action where not all may cooperate at the same time. An example of this is two clients trying to synchronise with one server. A model of this seen will be seen in section 4.4.1.

Using race conditions means that in cases of branching, all branches are speculatively executed and the first one to finish is the winner. The winning branch then dictates the ensuing behaviour of the Component, whilst all other executions are aborted. Using speculative execution in the form of race condition is *not* an option when producing skeletons that are to become full implementations (in which each Component represents a single *Thread* of computation). Speculative branching would mean the forking of execution and aborting of the actions pre-completion, which would make the skeleton impossible to “flesh out”. Instead, simulating the race condition, as is done in these implementations, allows some certainty:

- once committed to an Action, a Component will not do anything else until the Action has completed;
- once started, an Action must complete;
- if a branch has been chosen, the other branches will definitely not be chosen, and will therefore not be able to affect the system.

These certainties are not of value in evaluating a model but they are necessary for creating a full implementation. Speculative execution would mean Actions might not complete and this would lead to the system being left in an inconsistent state, were it actually performing commands. Apart from this point, it would mean spawning a new thread for each branch and then culling the losing branching, leading to very inefficient runtime characteristics.

3.3.2 Concurrency Control

In accordance with tried-and-tested design patterns [Lea, 1997], concurrency control is separated from the functionality through the creation of separate utility classes. This is especially important where inheritance is used— the Component objects of the system are extended and these extensions may use the object's *synchronising* object monitor lock. When using inherited methods, unexpected behaviour may arise when the different layers of code lock or wait on the object monitor whilst it is also being used separately in another layer.

3.4 The PEPA2Java Translator

The API and its implementations handle running a PEPA-like model as a Java system, but it is the job of the Translator to create the classes that make up the implementation of a particular model. It is designed to be very easy to use and creates clear code which may be compiled and run automatically (by creating "Makefiles").

3.4.1 Lexer and Parser

The process of lexical analysis is one of pattern-matching input text to recognise specified tokens such as keywords and operators, and returning them to a tool such as a parser for further processing.

A parser is a program or a component of a program that analyses the grammatical structure of an input, with respect to a given formal grammar [Hidders, 2001]. It takes its input from the lexical analyser and uses a specified grammar to break the PEPA model into its constituent definitions and commands.

The lexer of the PEPA Workbench can be re-used as-is. However, as the Translator will need to create different types of objects than the Workbench (because of the its different functionality), the parser needed to be modified . The PEPA Workbench uses a Look-Ahead Left-Right (LALR) parser, generated by the YACC-like Java CUP tool[Appel, 1998, Hudson, 1999]. This helpful utility enables quick generation of Java LALR parsers. It was possible to reuse the PEPA Workbench's CUP grammar specification, specifying new actions for each rule the parser comes across. CUP was then used to generate a new parser suited to the project's needs.

3.4.2 The Translator's Data Structures

After parsing, the Translator has all the information from the model available to it. However, this is not yet in the correct form for easy generation of a skeleton. Therefore, the Translator first scans the various data structures it has built during parsing to determine how many components there are, whether any of them are multiple instances of the same type, how many actions there are, which ones are shared, which ones are private, and the synchronisation sets. It collects this information and stores it in new data structures of a "friendlier" form, making it more easily available to the processes generating Java code.

On the whole, this approach needs more memory but not excessive amounts. Producing a correct translation is of higher priority than minimising memory usage. In any case, once the Translator exits on completion, the memory will be made available by the garbage collector should it be needed for running a model.

3.4.3 Integration into the PEPA Workbench

Integration into the PEPA Workbench is straightforward— a matter of adding a menu item, which opens a Translator dialogue window. From this window, options such as translation type, debug level and simulation speed can be set. Buttons in the window allow the model to be translated, compiled and run. The model translated is the one that is currently being worked on in the Workbench. Keeping the Translator in a separate package and using a separate "pop-up" window means that it will be easy to integrate in different versions of the Workbench, or removed if necessary.

3.5 Evaluation

The evaluation has been carried out on the models produced by the Translator to see whether they compile and run correctly, and also to see whether their performance is the same as predicted by the Workbench.

Evaluation has been carried out on the numerous existing PEPA models already distributed as part of the Workbench. Some additional models which test areas not covered by the existing models were also created.

Though time has not allowed, it would be worthwhile attempting to take a simple model of a practical application through the full process shown in figure 1.1. One such example might be to take a very simple Client-Server model where the Server controls a resource such as a printer, and turn it into a working Java system. This exercise could highlight any problems with this idealised process of design and implementation.

Chapter 4

Prototype A: Specification & Implementation of a Flawed API

4.1 Overview

The process of implementation is one of discovery, of hacked solutions and of re-design. Designing the perfect system is impossible and this implementation is no exception. Various ideas were tried and evaluated. Some ideas survived right through to the final version but most came and went. The most dramatic stage was when, after Prototype A and its Translator were fully completed, it was discovered to have major shortcomings. Almost all the code was scrapped and Prototype B was started.

As disheartening as starting from scratch was, the time and effort that went into Prototype A (described in this chapter) was certainly not wasted. Prototype B is much stronger for the process and eventually became the finalised PEPA2Java package. Chapter 5 describes the functionality of the completed version. However, it is also worthwhile recounting the evolution of the implementations and the ideas that didn't work, as well as the ones that did.

4.2 First Steps: Proof of Concept

The first step to building the API and its implementation is to take a simple PEPA example and build a working Java model of it. Through this process, ideas are tested. A practical working system helps to give substance to the concepts and highlight those areas where the ideas do not work.

4.2.1 A Simple Client-Proxy-Server System

The model chosen for the Proof of Concept is a simple system made up of three components: a client that makes requests for information, a server that is the authoritative provider of the requested information, and a proxy that serves the client but may also need to query the server when it can't fulfil a request itself. The ASCII-formatted PEPA model is:

```
%a = 1.0;
%b = 2.0;
%c = 3.0;

#Client = (cReq, a).(cRep, T).Client;

#Proxy = (cReq, T).Proxy';
#Proxy' = (cRep, b).Proxy + (pReq, a).(pRep, T).(cRep,b).Proxy;

#Server = (pReq, T).(pRep, c).Server;

Client <cReq, cRep> Proxy <pReq, pRep> Server
```

This model provides a simple example which nevertheless contains all the PEPA constructs most important to the project.

4.2.2 Activities as Method Calls

In the *Proof of Concept* version, method calls are used by the passive participant of an activity to call a method of the actively participating Component. This works because all shared Actions in the model contain one active and one passive participant. This is also the case with most of the example PEPA models distributed with the PEPA Workbench. Using method calls, each request has an implicit reply so the "request" and "reply" Activities can be merged. Each passive participant needs a reference to the active participant, so it can call, for example, `Proxy.request(info)`.

There is only one thread of execution in such a system, which follows the method calls through the system. For example, the client calls the proxy's request method. This request method either returns a reply straight to the client (i.e. if it can fulfil the request), or calls the server's request method (i.e. if the proxy can not fulfil the request). This choice is determined by a randomly generated number. When the proxy does need to call the server's request method, the server will reply. Finally the proxy passes back the reply to the client, completing one loop of the system.

The method call concept fits very nicely with traditional programming ideas but is quite different from the way a PEPA model functions. In a PEPA model, each Component is executing separately and shared activities are places in the system where multiple threads synchronise on a single action. This is also more akin to an actual client-server relationship where both computers are executing independently. The server is idle until a request comes in. It processes the incoming request, and then becomes idle again. The original concept will work only for those PEPA models which follow the passive-active activity cooperation scheme and where one thread of execution is sufficient to capture all system behaviour. This is clearly unacceptable.

4.3 Prototype A

Given these rather severe limitations, a new concept sprung to mind. The new concept allowed the system to behave much more like a genuine PEPA model and became Prototype A.

4.3.1 The Component and Activity Classes

Instead of having **one** thread of execution which moves between Components, each Component object should be a **separately** executing Thread object. Each Thread should loop through its Activities. Shared Activities, needing synchronisation, should pause until all Component members of the synchronisation set are in the right state and ready to cooperate on the Activity. A semaphore mechanism, which is decremented each time a Component commits to the Activity, holds them and only lets the Activity run when the counter hits zero. It continues holding the Components until the Activity completes, then releases them to continue their separate executions.

Activities are a separate class from Components and Components receive references to the Activities they participate in from a central Reference object. One of the active participants in an Activity starts it when all participants are in the correct state.

Component.run and Component.loop

The *run* method of the Thread superclass will repeatedly call *this.loop()* as long as the system is running. The *loop* method cycles through this Component's instructions.

Components in this prototype equate not to the Components as defined in the composition line of the PEPA model but rather to the definition lines of Components in the model. Furthermore, a separate Component class is created for each derivative. The PEPA composition line is used to determine which Threads begin started and which ones do not. Take, for example, the simple model:

```
#A  = (a, r).A′;
#A′ = (b, s).A;


A
```

The translation of this model would yield two Component objects, *A* and *A'*. At the beginning of the simulation *A* would be set running. The last instruction of its *loop* method (after it had completed Activity (a,r)) would execute the *this.call(A')* method call. Respectively, Component *A'* would issue *this.call(A)* as its last instruction.

This is the first problem with Prototype A: the method call stack will continue to grow as these *loop* methods never complete but instead keep calling each other in a cycle.

Component.init and *Component.addMeTo*

The *init* method is an abstract method which needs to be defined for each particular Component. It gets and sets the references to other Components and Activities it needs and also calls the *addMeTo* method to add itself to particular Activities it participates in, as either a passive or an active participant.

Component.call

A *call* method is defined for Component objects, which takes as an argument an Activity to join. This *locks* the Component into running that Activity (and also sets the *ready* flag). Individual activities are always able to execute immediately but shared ones require synchronisation between multiple Components. Shared Activities contain a SynchSet object which holds a *Ready* and a *Locked* flag for each Component that participates in the Activity. Only once all the *Locked* flags are set have the required participants joined and the Activity may start.

Alternatively, the *call* method can take a Component as its argument. In this case, it starts that Component by calling *Component.loop*.

Component.choice

Component objects have a *choice* method defined that takes as argument two Activities and returns either 1 or 2 as the decision on which Activity to run. It is used in a *switch* block in the Component's *loop* method to choose which branch a model should take, as illustrated below:

```
switch(this.choice(activity1, activity2)) {
case 1:
    this.call(activity1);
    break;
```

```
case 2:
    this.call(activity2);
    break;
}
```

On calling *choice* the *ready* flag is set in the Activity's synchronisation set object, indicating that this Component may choose to join this Activity. The execution of this Component is then paused briefly to allow other Components to join this Activity or evaluate their own *Choices* with this Component marked as *Ready* for the two Activities.

When it awakes, the *Choice* method then checks whether either Activity is able to execute— i.e. all *Ready* flags are set. If so, that Activity may be chosen, depending on whether the other Activity is ready or not. Activities are prioritised for choosing, with Shared Active as highest priority, then Shared Passive, and Individual as lowest. If both Activities are ready to go, and also of the same priority, the faster one (the one with the smaller sleep-time) is chosen.

If neither Activity is ready to run, the Component cycles through the pause/ recheck routine until one of the Activities becomes available to run.

Activity.run

When all synchronising Components have joined an Activity, it is run. It pauses for a period determined by its rate and then exits.

Activity problems

A problem with the Activity class is that there is no separation of Activities and Actions as there is in PEPA. In this implementation, an Activity is behaving more like an Action. In PEPA, Actions are global, whereas Activities specify for a single Component which Action it is partaking in and at what rate. In this implementation, all Components are referencing the same global Activity and submit the rate at which they wish to participate.

4.3.2 The SynchSet Class

This class represents the synchronisation set for a particular Action (or Activity in this implementation), specifying which Components must join before it can run. Each Component can declare that it is *Ready* to join, or that it has *Locked*. A *Ready* Component may or may not join— it is notifying the Activity that it is choosing between this Activity and another.

Locking is a notification that the Component has committed to this Activity and is waiting for the Activity to execute. Once committed, the Component must halt its execution until the the Activity has run.

The SynchSet class is the fatal flaw in Prototype A because it failed to take into account the facility of PEPA to contain more complex synchronisation sets (See 4.4.1).

4.3.3 The Rate Class

Exponential rate distributions were first built directly into Activity objects and were used to determine how long the delay was when the Activity ran. In order to separate functionality, Rate objects were created. These Rate objects were initialised with the distribution's rate and used to calculate a sleep-time:

```
sleep-time = ExpDist(rate).sample() * baseSleep;
```

To generate useful delays, the base sleep time was set to 1000. When an Activity was run, `sleep(sleep-time)` was called on the executing Thread, which paused it for so many milliseconds.

The *ExponentialDistribution* class used generates properly balanced distributions and is included in a GNU General Public License package. The package originates from the Probability/ Statistics Object Library of the Department of Mathematical Sciences, University of Alabama in Huntsville [Siegrist and Duehring, 2001].

4.3.4 The Ref Class

The Ref class contains Vector objects that hold references for all the system's Component and Activity objects. It defines two methods: *getActivity* and *getComponent* which take as argument the "id string" of the Activity or Component (usually just the name as given in the PEPA model) and returns a reference to the object representing that construct in the system or, if no match can be found, throws an Exception.

References are added automatically to the appropriate vectors of the Ref object by the Component and Activity constructors— calling `new Component(''CompA'')` will automatically add a reference to the newly created object to Ref's Components Vector. Other Components may then obtain the reference to Component A by calling `Ref.getComponent(''CompA'')`.

4.3.5 The Simulator Class

This class creates all the Activity and Component objects, starts those Components mentioned in the composition line and handles all Simulator calls to do with updating the Simulator Window. It defines abstract methods *createActivities* and *createComponents*, which is the place where a model implementation should issue the appropriate Component and Activity creation statements. This is necessary because the Simulator needs to perform background commands on the Activities and Components before it can start the system, such as calling `Component.init` on all Components and `Component.start` on all Components which are set to begin on start-up.

In addition, the Simulator adds the various Threads of the system to appropriate ThreadGroup objects, so that it can perform actions such as *pausing* the execution of the system or force Activities to awake from long *sleep* delays.

4.3.6 Model Implementations

The Client-Proxy-Server model (given in Appendix A) is implemented using the Prototype A package and executes as desired. The Java code (given in Appendix B) is relatively straightforward— the parallels between the PEPA constructs and the Java constructs are clear.

The PC-LAN 4 model has also been manually translated to run the Prototype A package. With Prototype A seemingly capable of running models correctly and a few manual translations implemented, the API and its implementations were frozen to begin work on creating an automatic translator.

4.3.7 Translator

As chapter 6 is dedicated to describing the implementation of the Translator and much of the final Translator is very similar to the first Translator, this section will only give a brief overview of the first incarnation.

The first translator is quite simple in its functioning, as the initial API takes a rather rudimentary view of the way PEPA works. It parses the various data structures created by the parser but does not need to do a lot of maniplation to create its output. For example, each definition line in the PEPA model becomes a Component; each Component in the composition line has a flag set so that it is started by the Simulator when the system is started; and for each Action in the composition line, all Components found in the set to the left and right of it call the `addMeTo(thisActivity)` method.

Although the code of the CPS model (Appendix B) was manually translated, the first Translator produces exactly the same code.

With both Prototypes A and B, the Java code needed to define a PEPA system is purposely kept as straightforward as possible and is also as similar as possible to the PEPA model code. This is to make manual translation, extension and comprehension of the code as simple as possible but has the added side-effect of making an automatic Translator's work easier. Unfortunately, however, the first API was *too* simple to run many PEPA models.

4.4 Problems with Prototype A

Two months into the project there was a fully working PEPA2Java package and Translator. At this stage, testing began by using the Translator to generate Java equivalents of many of the PEPA models found in the Java Workbench package. However, whilst some models were successfully translated, and ran as desired, there were many cases where this was not the case.

It was now clear that this version of the PEPA2Java API was fundamentally flawed in the sense that it did not provide PEPA-equivalent constructs, a key condition for meeting the objective of the project. Considerable effort was spent to try and correct the problems but eventually these endeavors were abandoned in favour of taking a new approach, as outlined in the next chapter. The new approach would maintain the parts of Prototype A that work well, whilst replacing those that do not.

Prototype A's API differed to the semantics of PEPA in three ways:

4.4.1 Synchronisation Sets

The biggest problem with Prototype A was its implementation of synchronization sets. Each Activity had one synchronization set and this set contained all the Components that cooperate on the Activity. However, the implementation was overly simple: it allowed only for one set per Activity and thus limited the number and type of PEPA models that could be translated.

In Prototype A, an Activity may not run until all the members of its synchronization set have committed to it. Consider, for example, the model:

```
%a = 1.0;
%b = 2.0;

#Server = (req, T).(rep, b).Server;

#Client = (req, a).(rep, T).Client;


Server <req, rep> (Client || Client)
```

This system cannot be modelled correctly in Prototype A. In order to implement a system such as this one, Activity *req* would need to be split into Activity req_1 and req_2. The two Activity objects would have slightly different synchronization sets, each containing the *Server* and one of the two *Client* Components.

This type of hack may work but adding extra Activities is undesirable as it breaks the parallel between the Java Skeleton and the PEPA model. Also, there are some models where this fix would cause the model to act differently. Finally, it results in the creation of extra objects, which negatively affects running efficiency. The SynchSet problem is the main reason why Prototype A was scrapped.

4.4.2 Activities and Actions

In Prototype A, Activities performed a combination of the function that Actions and Activities perform in PEPA. There was no Action class at all because the Activity class was a hybrid of the two constructs. There was no clear separation of an Activity, which is local to a Component, and an Action, which is a global construct that may or may not be joined by multiple Activities.

This difference did not cause any real problems when it came to running a model as a Java system but was counterintuitive to the user. Therefore, Prototype B should specify separate Activity and Action classes.

4.4.3 The Form of Components

Consider the simple PEPA model:

```
A  = (a, 2.0).A1;
A1 = (b, 1.0).A;


A
```

In a true PEPA interpretation of the model, the separate Components of the system are defined in the final composition line. Each variable in the composition line equates to a separately executing process and the Actions they cooperate on are held in the synchronization sets between them. Therefore `A` is the only Component in the system above. Component behaviour is defined above in the process definitions, so `A` can exhibit both the behaviour defined by `A` and the behaviour defined by `A1`.

However, in Prototype A, each definition is taken as a separate Component, capable of being executed as a separate Thread. Those Components that are referred to in the composition line are started on system start-up but are otherwise no different to other Components. So, in a Prototype A translation of the above system, Component `A` executes an Activity `a`, then calls `A1` to execute. Similarly, `A1` executes `b` and then calls `A`. For Prototype A, the composition line is used only to determine that the Thread object holding `A` should be started on system start-up.

Again, the system functions as one would expect a PEPA system to function. However, there is a lack of equivalence between a PEPA Component and a PEPA2Java Component, potentially causing confusion. Also, this means that more Component classes are created by the Translator than necessary and Thread objects are created which are never started. These problems lead to less efficient run-time performance once all the objects are created.

4.4.4 The Method-Call Stack

This final problem with Prototype A is not a problem with the translation of PEPA semantics but an implementation problem. The way Components call other Components without ever exiting their `loop` method leads to a forever growing method-call stack.

Consider the model introduced above in Section 4.4.3. The final action of both Component `A` and `A1` is to call the other. Due to the cycle, neither of the `loop` method calls will ever return, so the stack will continuing growing until it eventually overflows. This is a problem with the definition of Components and the implementation mechanism used to mimic PEPA Component behaviour.

4.4.5 Conclusion

Any one of these problems might have been solved but taken together they indicated a serious rift between PEPA constructs and their PEPA2Java equivalents. Specifically, the composition line, being the core definition of a PEPA system, was not the core definition of a PEPA2Java system. Also, as Prototype A had started life as a proof-of-concept, the code was becoming more and more convoluted. To solve these discrepancies and end up with a more elegant implementation, the slate was wiped clean and Prototype B was created.

Chapter 5

Prototype B: Specification & Implementation of the PEPA2Java API

5.1 Overview

In developing Prototype B, the flawed ideas and the shortcomings of its predecessor were left on the drawing board but the lessons learnt and the sound principles established provided a solid base from which to move forward. Prototype B is both a versatile and powerful tool. It is the blueprint for the final PEPA2Java API specification and has evolved into the Barebones and Simulator packages which implement that API.

5.2 The PEPA System

Central to the implementation is the PepaSystem class. This class performs essentially the same function as the Simulator class did in Prototype A. It defines three abstract methods which must be defined by implementing models — namely `createComponents`, `createActions` and `initSynchs`. Examining Appendix C shows an example of the Client-Proxy-Server PEPA model (given in Appendix A) implemented using Prototype B. It demonstrates the form and use of the PepaSystem-extending Sim class that implements the required abstract methods. The PepaSystem object will create and initialise all the model objects, and start the model running. If the Simulator package is used, it will also control the GUI and the debugging messages.

The PepaSystem object also maintains three vectors, holding references to all the Components, Activities and Actions that make up the system. Two methods, `comp(id)` and `action(id)` provide similar functionality to the Ref object's functionality (see section 4.3.4) in Prototype A. These vectors and methods are used by the SimWindow class.

The preferred manner for Components to get references to the other global objects is to access them through static references which should be defined in the PepaSystem-implementing class for each model. This is the form of the output as created by the Translator. However, the API is flexible enough to allow the user to assign references in whichever manner she finds convenient.

5.3 Forcing Lock-step: The Waiter and Lock classes

As in Prototype A, the most important part of the implementation deals with ensuring that the Java skeleton of a model runs as the PEPA model dictates it should. However, there is no native support for the concept of synchronising multiple threads of execution for cooperation on an action. Therefore, much of the challenge lies in getting Java to behave in this way. The Action class, specified in section 5.6, ensures the proper synchronisation of multiple threads on cooperating sections. To accomplish this, the Action class, and also the Component class, make use of two simple concurrency control helper classes (see section 3.3.2), the Lock class and the Waiter class.

The Waiter class is a very simple extension to `java.lang.Object`. It uses the object monitor each Java object has and provides wrapper methods (`w` and `n`) around its `wait` and `notifyAll` methods. These wrapper methods are *synchronised* because the caller must own the object monitor before it can call `wait` or `notifyAll`. In addition, it catches any thrown `InterruptedException` and exits the method, performing no further action. The Waiter class exists to avoid any user-defined extending subclass inadvertently locking or releasing the object monitor and interfering with the functioning of the PEPA2Java implementation. It also allows as many Waiter objects to be created as needed and allows these objects to be *private* to an object or shared within the package as a *protected* object.

These objects are used throughout the implementation wherever an object must wait on a condition being true. It is used, for example in choice checking when a Component is waiting on any branch to become available for running, or when an Action is waiting for all cooperating Components to join. This replaces the unnecessary and inefficient constant re-checking of conditions— checking only occurs after an influencing condition has changed.

The Lock class is more complex and is intended to replace the object monitor when multiple locks are required to be held simultaneously. It is specifically built for use in the `choice` method. It defines two public functions, `request` and `release` which, respectively, request the lock and release it. Requesting a lock holds the caller until the lock is available. The method locks the object again, and returns. Releasing the lock allows the next waiting requester to acquire the lock. Note, however, that there is no guarantee of fairness: the lock will go to one of those waiting, but not necessarily the one that has been waiting longest. Because locks are held only very briefly, inside the `Component.choice` method, this is not a problem.

This functionality on its own adds no new behaviour but the class specifies several *protected* methods that do. The `Lock[] getLockers(Activity[])` method takes a multi-set of Activity objects and returns a globally sorted set (i.e. no duplicates) of the Lock objects of the Actions referred to by those Activities (each Action contains a Lock object). The static methods `request(Lock[])` and `release(Lock[])` are similar to their non-static cousins, except that they lock an array of Lock objects, rather than a single one. Because the Locks arrays are returned in a globally sorted order (by `Hashcode`), two threads simultaneously requesting Lock sets with shared members will never cause deadlock.

5.4 Simulating Race Conditions

To address the schism between PEPA models and working systems, the race condition (see section 3.3.1) is simulated in the following manner in the Simulator and Barebones packages that make up Prototype B:

1. Rate samples are decided a priori— the branch with the smallest sleep-time will be chosen. This is the case for both types of branching— choosing and multiple Components vying to join an Action (see section 3.3.1).

2. Losing branches are resampled to simulate speculative execution— this prevents branches that get a high sleep-time from losing again and again because of one "unlucky" sample.

3. The winning branch will pause for its sleep-time, resample and then execute its next behaviour.

4. Before a branch can be chosen, it must be ready to execute— all the members of the synchronisation set must be prepared to participate. Otherwise, another branch will be chosen. If no branches are ready, the system will execute the first branch that becomes ready. However, choosing commits the system to execute that branch— it cannot be interrupted by a faster branch that becomes ready later.

5. Because the first ready branch wins, the choice method necessarily needs a pause to allow participants of shared Actions to join. Otherwise, individual Actions would unfairly dominate choice branching as they are *always* ready. The length of this pause is defined as a constant value in the Component class, at present set to 100 milliseconds.

The first three mechanisms allow the PEPA2Java system to mimic a PEPA system perfectly but the last two are not ideal— they cause some models to execute incorrectly. Chapter 7 gives more detail.

5.5 Components and Scripts

The Component class extends the Java Thread class— each is started on system startup. Each Component represents one of the Components as defined in the composition line of the PEPA model. A Component class on its own defines no behaviour. For this, the CompScript interface is used.

The CompScript interface is similar in idea to the Runnable interface. It defines a single method, `actions`. The method takes no argument but, crucially, it must return a CompScript object reference.

Each Component has a reference to a CompScript object, which is initially set by the `Component.setStartScript` method. The Component `run` method calls the current CompScript's `actions` method. This executes the behaviour of the Component as it is defined for one particular definition, for example by executing Activities or making branching choices. The final action of the `CompScript.actions` method is to return a reference to a CompScript object, possibly itself. The `Component.run` method will then run the `actions` method of the newly assigned CompScript.

This implementation solves two problems of Prototype A. First, it resolves the potential for Method Call Stack overflow, because each method returns a CompScript reference and exits. Second, Components in this implementation may take on many varying behaviours as defined by scripts, but their synchronisation sets and their representation remain constant. Thus Prototype B Components are equivalent to Components as defined in the PEPA composition line, and the Component's CompScripts are equivalent to a Component's sequential process definitions in a PEPA model.

CompScript-implementing classes are defined as inner classes of their Component, which means they have access to all the references of their holding Component, including the references to any other CompScript classes it may define.

The new translation mentioned in Section 5.2 of the Client-Proxy-Server example from the previous chapter demonstrates the new form of Components and their Scripts in Prototype B. This translation is given in full in Appendix C.

Finally, the other public method in Component class is the `choice` method, which is detailed in section 5.9.

5.6 Actions

Each shared Action of the model is represented by an Action object, which implements the `Runnable` interface. It also holds the root to a synchronisation tree (section 5.8) which is used to determine whether the Action is ready to run or not. An Action steps through five stages:

1. **Wait for** `noRunners`: The first command of a running Action object is to hold until all Component members of its synchronisation set have their `runner` flag set to false. This indicates that the Component believes the previous run of the Action to be completed, which is a prerequisite of the next run. When all Components know that the last run has completed, the next step can take place. This is necessary because if an Action is very short, there is the possibility that in the multi-threaded concurrent execution environment, one or more of the the last chosen Component threads may not have executed since the last run. If the next run were to be allowed to start before all Components were aware that the last had ended, deadlock might occur— while one of the Components might still be waiting for the already executed Action to run, those Components aware that the last Action already ran might be waiting for the next Action to occur. Neither Action could then ever execute because neither would have a complete synchronisation set. Therefore, the Action will pause between runs until all Components have acknowledged its execution.

2. **Wait for** `allLocked`: The next step in the execution of this Action is to pause until the `allLocked` method returns true. This happens only when all the Component members of any particular subset of the synchronisation set have joined the Action and committed to its running. The determination of the subsets is detailed in section 5.8.

3. **Set the Runners**: Now the Action is ready to be run, it notifies the members of a particular cooperation set that they have been chosen to participate in this run of the Action. The Action passes these members a reference to its `holder` Thread. They set their `runner` flags to true, and `join` this Thread, meaning they will cooperate in the Action. The `join` method is a Thread object method which causes the calling Thread to wait for the target Thread to die before it can resume execution.

4. **Start the Action**: The Action calls its `action` method, which will cause the Thread to sleep for a number of milliseconds set by its determining rate. (Deter-

mining rates are described in section 5.10.1.) Alternatively, in user implementations, the sleep action will be replaced by the actual functionality required.

5. **Reset**: The `reset` method is the final command of this run of Action. It creates a new Thread object and assigns the Action's `holder` reference to it. The Action object is passed to the Thread as a *Runnable* argument. The Action's Rate is reset to TOP, so that any Rates submitted by Activities will dominate. The final action of the method is to `start` the newly created Thread. This old holder Thread will then die, releasing the joined Runners, which will immediately set their `runner` flags back to false, signalling they are aware that the current run of this Action has completed.

Note that on system start-up, the PepaSystem object (section 5.2) will call the `reset` method of all Actions, which will start them.

5.7 Activities

An Activity is joined when a Component calls its `join` or `joinAt` method. These are the only public methods of the Activity class.

Conceptually, there is a subtle difference between Activities as specified in PEPA and PEPA2Java. Namely, Activities in Prototype B are objects held by their executing Component. Every Component participating in an Action holds one Activity per Action. This means that if a Component runs multiple Activities at different rates, but the Activities are participating in the same Action, they will be represented by a single Activity object in the Component. The only difference is that the Component will *join* that Activity at different Rates at different times. This is done by changing the Rate reference the Activity holds.

The `join` method commits the Component to running the Activity, at whichever Rate the Activity is currently set to run at. When the Action is about to run, it calculates its determining Rate from all the participating Activities. The mechanism for deciding the determining rate of the Action is described in section 5.10.1. The `joinAt` method is identical to the `join` method except that it also allows the caller to specify the Rate at which this Activity should be performed. This second method is the recommended implementation of the PEPA *prefix* construct, and is the method always used in Translator generated code. It is equivalent to a `setRate` call, immediately followed by a call to `join`. Otherwise, the Activity will use whichever Rate it is currently assigned.

There are two sub-classes of the Activity class, IndivActiv and SharedActiv. IndivActiv is used for executing individual Activities— those without synchronization. SharedActiv is used to execute Activities which require synchronization between multiple Components.

5.7.1 The SharedActiv class

The SharedActiv class is quite complex. Synchronization is handled by creating Synchronization Set trees from `SSNode` objects. As is clear from figure 5.1, the Activity class and its subclasses inherit SSNode class functionality. These objects are detailed below in section 5.8. The most complex methods of the SharedActiv and the SSNode classes deal with choice branching, and the manner this is accomplished is detailed in section 5.9.

An individual Activity may execute an Action immediately on calling `joinAt`, whereas a shared Activity will most likely need to wait for additional Components to join. Also, even if a Component does join an Action, it may or may not be chosen as a participant if there are multiple possible combinations of cooperators for an Action. Using an example from a previous chapter, consider the composition line:

```
(Client || Client) <serve> Server
```

In this case, the `Server` Component will always participate whenever the `serve` Action is run. However, only one of the two `Client` Components may join each run. Therefore, if both `Client` Components are waiting for `serve`, one must be chosen at random to be a *Runner* whilst the other will need to wait for a future run of the Action. As outlined above, the Action will designate the *Runners* when it is ready to execute.

The `setRunners` method traverses the synchronisation set tree and picks the “fastest” subset which is ready to go. The rates of the losing subsets are resampled to ensure they aren’t continually passed over.

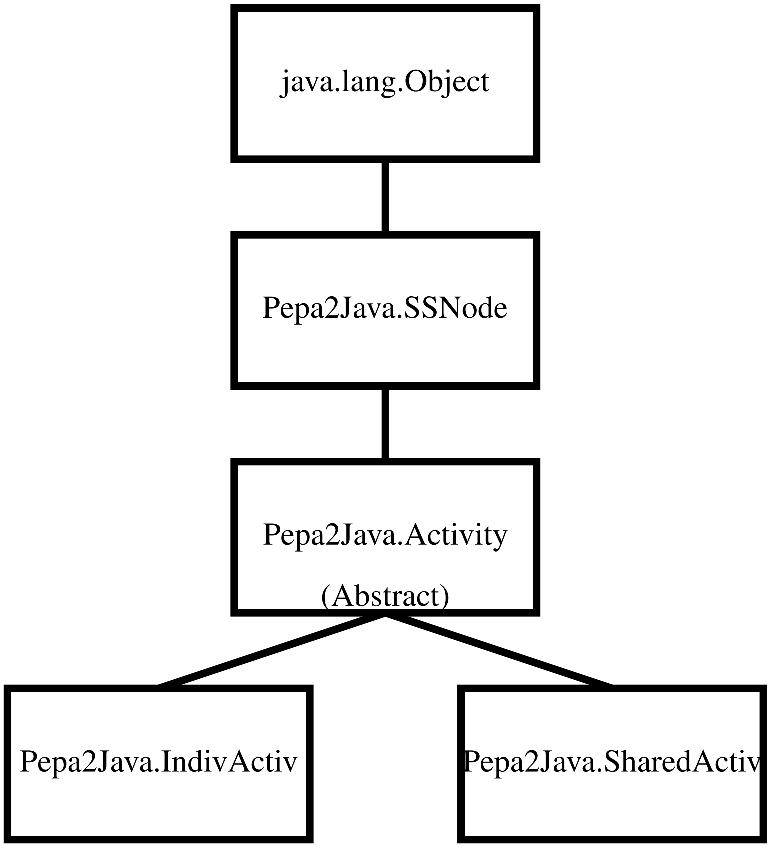

Figure 5.1: The SSNode/ Activity class hierarchy

5.7.2 The IndivActiv class

In essence, the IndivActiv class exists to increase efficiency of execution, as individual Activities may always run immediately on joining, whereas shared ones may need to pause until other Components are ready. Therefore, a lot of the "check and wait" methods needed to synchronize multiple objects, such as `isReady` and choice locking, are not needed in the case of individual activities, and return immediately without performing any commands. This allows the rest of the framework to treat individual and shared activities identically.

5.8 Synchronization Trees and the SSNode class

Synchronization set trees connect Actions to their participating Activities. An Action is constantly waiting to run. Each time an Activity *joins* it, the Action will re-check the status of its synchronization tree to see whether it may run. This waiting is accomplished with the usual wait/notify Waiter methods (see section 5.3). The tree is made up of SSNode objects. There are three types of SSNode objects, two used as the branches and one as the leaves— *AND* nodes and *OR* nodes are the branches, and Activity objects are the leaves. Each branch node has a left child and a right child. Though the leaves of the tree are specified as Activity objects, only SharedActiv objects are used as part of a sychronization tree because individual Activities require no synchronization.

Consider the Composition line from section 5.7.1 again:

```
(Client || Client) <serve> Server
```

From this line, we can see that the synchronization set has two *subsets* of cooperating *Runners*. Both sets include the `Server` Component, and either one or the other of the two `Client` Components.

This would yield the synchronization tree specified in figure 5.2. The most important function of the synchronization tree is to determine whether the Action may run or not. This is checked each time the Action object calls `isLocked` on the root node of the tree. The method is recursively called on the nodes. `AND` nodes return

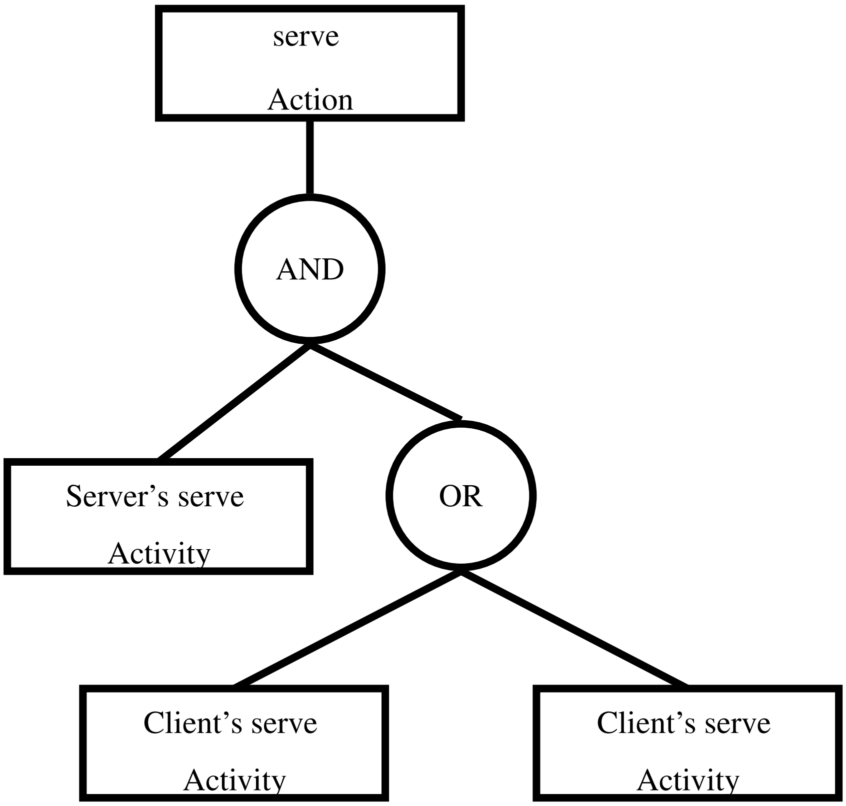

Figure 5.2: A Simple Synchronization Tree

true iff both the left and the right child return true, whereas `OR` nodes return true iff either (or both) of the children return true. The tree construct is equivalent to the composition line because the `serve` action requires the `Server` Component and either of the two `Client` Components to cooperate before it can execute. Using these trees, an arbitrarily complex synchronization set can be represented.

When the Action runs, it calls the `setRunners(actionThread)` method on the root of the tree. This passes the reference to the running Thread that will execute the Action, on which the Runners should call `actionThread.join`. (N.B.: the naming of the `Activity.join` method is not ideal, as it could easily be confused with the `Thread.join` method, which is another reason to instead use `Activity.joinAt`.) Also, the `setRunners` method *returns* the determining Rate of the subset it has chosen, which will specify the pause length simulating the Action running.

Calling `setRunners` on an `AND` node will recursively call the method on both the node's children, whereas calling it on an `OR` node will cause it to be recursively called on only one of the node's children.

Which child is chosen in an `OR` node is determined firstly by whether either or both the children return true for `isLocked`. If both are locked, then the faster branch will be returned. If they are the same speed (i.e. because they both run at rate $\top$), then the choice is decided by a weighted random function: the chance of `setRunners` being called on the left branch is equal to the proportion of leaves in the left subtree over the total number of leaves in the two subtrees. If only one of the two subtrees returns true for `isLocked`, then `setRunners` is called on the locked subtree.

If there were three `Client` Components (see figure 5.3), they all operated at rate $\top$, and only one was needed to join the Action, then the first `OR` node would return the right child (`Client1`'s node) one-third of the time, and the left child (another `OR` node) two-thirds of the time. The lower `OR` node would contain the final two `Client` nodes and would return them with equal probability.

This method is quite fair, except that `Client2` and `Client3` have a slight advantage over `Client1`. Imagine `Client1` and `Client2` are committed to join, but `Client3` is not. In this case, `Client2` will be chosen with a two-thirds probability, and `Client1` with a one-third probability, whereas it should be even. To fix this, the weight used should be the number of children in the *locked* state, rather than just the total number of children. However, to fix this for larger mixed `AND` and `OR` subtrees would confuse the matter further, and require more complex algorithms. In any case, in most models, the synchronization set is quite simple, and the Action runs as soon as the first subset returns true for `isLocked`. If not, the fastest branch usually decides matters. Nevertheless, this is an area which could be looked into, to see if a solution exists which still runs efficiently but chooses completely fairly. One final note is that the members which are ready but are *not* set as Runners have their Rates resampled to avoid them being passed unfairly over and over again.

The other defined methods are used in the evaluation of choices and are therefore described in section 5.9.

5.9 The *choice* Method

The mechanism for evaluating choice constructs fairly, together with the mechanisms for ensuring lock-step (section 5.3), is the most complex part of the implementation. The Component class defines a method `choice`, that takes as argument an array of Activity objects and an array of their respective Rate objects. It returns an integer, which is the position in the array of the chosen Activity. The `choice` method should be used in a switch statement, just as it was in Prototype A (see section 4.3.1 for the switch syntax).

However, this implementation of choice is quite different from that in Prototype A. There is no prioritization of choices as there was in the first implementation. Rather, this implementation sticks more closely to the original PEPA mechanism— the fastest branch which is ready to run is the one that is chosen. Determining the rate of individual Activities is simple: there is only one Rate object. However, in shared Activities, the Rate of the Action is determined by the slowest actively participating Activity. In section 5.10.2, the mechanism for finding the correct determining Rate for an Action in a choice block is described. Only those Actions which are ready to run are considered. In order to discover which Actions are ready to run without the Component first committing to join any of them, a two-phase process is carried out.

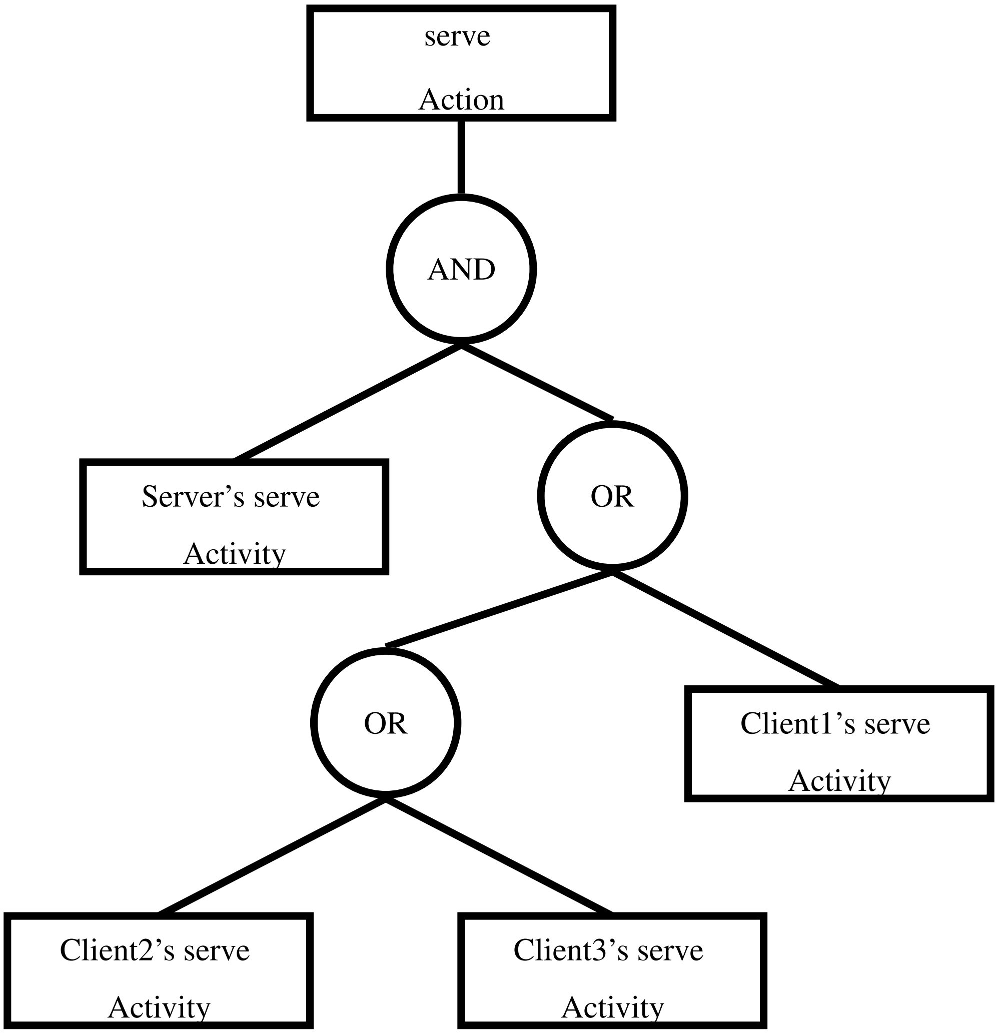

Figure 5.3: Synchronization Tree of $(Client1||Client2||Client3) < serve > Server$

5.9.1 Registering Interest

The first phase is a "registering of interest" in all the Activities which could possibly be chosen (i.e. all those in the array). This involves setting the `ready` flag on the Activity to true, indicating to other Components that this Activity *may* by chosen. These other Components evaluating `choices` of their own make use of this when deciding which of their branches may run. This way two or more simultaneously executing `choice` methods may choose branches where there is not yet a fully-ready cooperation set (i.e. not all Activities in the set are committed to the Action). Without some ready-but-not-locked provision, there would be starvation for some branches. For example, consider the example:

```
A = (m, T).A + (n, T).A
B = (m, 1.0).B + (i, 1).B
C = (n, 1.0).C

C <n> A <m> B
```

After registering interest in the set of Activities, this Component *pauses* for "choice-pause" (a constant set in the Component class). Any other `choice` methods that had previously yielded may discover that there are now additional Activities that may potentially run that will complete the set of an Action they themselves are considering choosing. They may then choose a branch that was previously believed to be unready for running.

Without the "choice-pause" and some provision for discovering potential choices, `B` would always choose individual Activity `i`, and `A` would always cooperate with `C` on `n`. This is incorrect behaviour, as we can see from the Rates that Action `m` is the fastest of the three Actions. The yielding increases the probability of two choice blocks to coming together at the same time and choosing Action `m`.

5.9.2 Locking and Deciding

After pausing, once the Thread resumes execution, it will attempt to gain the Locks of all the Actions it is considering joining. As described in section 5.3, the Locks are requested in a globally sorted order, meaning deadlock cannot occur. Without a global ordering, for example, Component A holding Lock 1, and requesting Lock 2, will wait forever for Component B to release its holding of Lock 2 because B may be waiting forever to acquire Lock 1 from A. This circular deadlock will not occur if Lock 1 is *always* requested before Lock 2 by all Components.

Once the Thread acquires the Locks of all the Actions it is considering, it will create a null reference which will eventually point to the fastest Activity it finds and a reference (named *best*) to the fastest Rate it has found. These are initially null and TOP, respectively. It then performs the following steps on each Activity:

1. Set the Rate of the Activity to be an `almostClone` (same Rate, different sample) of the one found in the corresponding position in the Rate array. This is necessary for cases such as `A = (x,1) + (x, 2)`, where there are different Rates for identical Activities in the same choice. Also, the `almostClone` method means two Activities can share the same Rate object outside the `choice` method without having identical sleep-times within it.

2. Use the `getPriority` method to get the determining Rate of this Activity's Action.

- For individual Activities, this returns its Rate.

- For shared Activities, if the Action the Activity participates in is *ready* to run, the method calls `getRate(myPath)` on the synchronization tree (see section 5.10.2) to return the determining Rate. If the Activity is not ready, the method returns *null*, signifying this Activity is unavailable for choosing. An Activity is *ready* to run if the *subset* of cooperating *Runners* that contains this Activity are either *Locked* (joined), or *Ready* (as set by another `choice` method).

3. If the determining Rate returned from `getPriority` is faster than that referred to by *best*, this Activity is the current *winner*. Note that in the case of equal sleeptimes, one is chosen at random.

After checking all Activities, if a *winner* is found, those Activities that are not chosen have their *ready* flags set back to false, all locks are released, and the position of the *winner* in the Activities array will be returned as the result of the `choice` method. Alternatively, if after checking all the Activities there are no ready Actions, all locks are released, the Thread will *pause* again, and the process (beginning with the locking) will repeat, until a choice is returned.

If a winner is chosen, all the losers' Rates are resampled to avoid a branch being passed over unfairly again and again. Notice that this includes the resampling of *all* the branch's cooperators— resampling a $\top$ rate has no effect and if that passive Activity's partner has sampled a very big sleep-time, that branch may be skipped many times.

The requesting and releasing of locks is necessary to avoid deadlock when there is more than one Component considering joining the same Action at the same time. For example, consider the following:

```
A = (m, 1) + (n, 1)
B = (m, T) + (n, T)
```

At the beginning, both `A` and `B` register their interest in perhaps joining their Activities `m` and `n`. If they are allowed to continue without first acquiring the locks to the Actions `m` and `n`, their concurrent execution might mean `A` chooses `m` whilst `B` chooses `n`. This is possible because all the *ready* flags are still set to true after one Component has chosen, whilst the other is still choosing. Locking ensures a *ready* flag is only ever set to true when it should be, by controlling access to the critical section.

5.10 Rates and Sleep-time

The Rate object is least changed between Prototype A and Prototype B. Each Rate object contains an *ExponentialDistribution* object [Siegrist and Duehring, 2001], and is used to calculate the period that an Action runs for, as well as determining race conditions in branching. TOP rates have the `specified` flag set to false, and return `Long.MIN_VALUE` as their sleep-time. Specified Rates return a sleep-time which is a random sample from an Exponential Distribution. Each time an Activity runs it generates a new sample by calling `Rate.next`. Losing branches are also resampled to simulate a race condition.

In addition, there is a `final static` unspecified Rate named `UNSPEC` which is defined for convenience and to avoid the unnecessary creation of multiple TOP Rate objects.

The static `slowerRate` method takes two Rates as parameters and returns the one with the longer sleep-time. Similarly, the `compare` method is used to compare *this* Rate to another. If this Rate is faster it returns 1, if it is slower it returns -1 and if the sleep-times are identical or they are both $\top$ it returns 0.

5.10.1 Determining Rates of an Action

An Action's Rate is the determining rate, meaning that when the Action eventually runs, it will use this Rate to determine the sleep-time. The determining rate is also used to decide the branching in choice constructs, where the fastest branch is chosen. The determining rate is defined as the slowest rate of all *actively* participating Activities. Participating Activities are those members of whichever subset of the synchronization set is currently set as the Action's *Runners*.

5.10.2 Determining Rates in *choice*

In choice constructs, the participating components have not been set yet— therefore the determining Rate is not yet known. In this case, the determining Rate is retrieved by the `getRate(myPath)` method. This method returns the determining Rate of the Components included in the synchronization tree containing all the required participants of the *subset which includes the Activity the choice method is currently considering*. For example, in the three client/ one server model, a call to `getRate(pathToClient1)` would return the slower of the two Rates of `Client1`'s `serve` Activity and `Server`'s `serve` Activity, but would not consider the `serve` Activities' Rates for Client2 or Client3.

5.11 The Simulator Package Extensions

All the details given so far describe the working of both the Barebones and the Simulator implementations. However, the Simulator implementation has added functionality in the form of debugging output and the SimWindow.

These extra functions are embedded into the source code of the Barebones implementation, but are marked by special comment tags. These tags allows the extra commands to be "commented out" automatically with a simple script. These tagged commands call the methods of the SimWindow and Debug class to do things like notify the SimWindow of state changes to each individual PEPA object, or send debug messages that give specific information on what the objects are doing.

It was important to be able to use one source for both implementations, as during development so much of the code was being constantly fixed, or scrapped and replaced. Maintaining two sets of code that were meant to operate identically in all ways except the debugging information and the GUI updates would have proven impossible.

5.11.1 Debugging

If run with the Simulator package included, every object in the PEPA2Java model will contain an instance of a Debug object. This provides two core methods.

The first method is `debug.println`, which takes a String and an integer as arguments. This method will echo the message to screen and to the SimWindow, providing it is not filtered. The filter works by only printing those messages with a integer less than or equal to the current Debug level set. There are `final static` integers set in the Debug class naming the various granularities of debug messaging for convenience (i.e. `NONE`, `CALLS`, `ALL`, etc.)

The second method is `debug.setState`, which takes as argument an integer representing the state of the object. These integers, like the debugging level, reference a set of predefined `final static` integers defined in the Debug class, and allow the state to be set to things like `Debug.JOINED`. This information is used to keep the SimWindow updated.

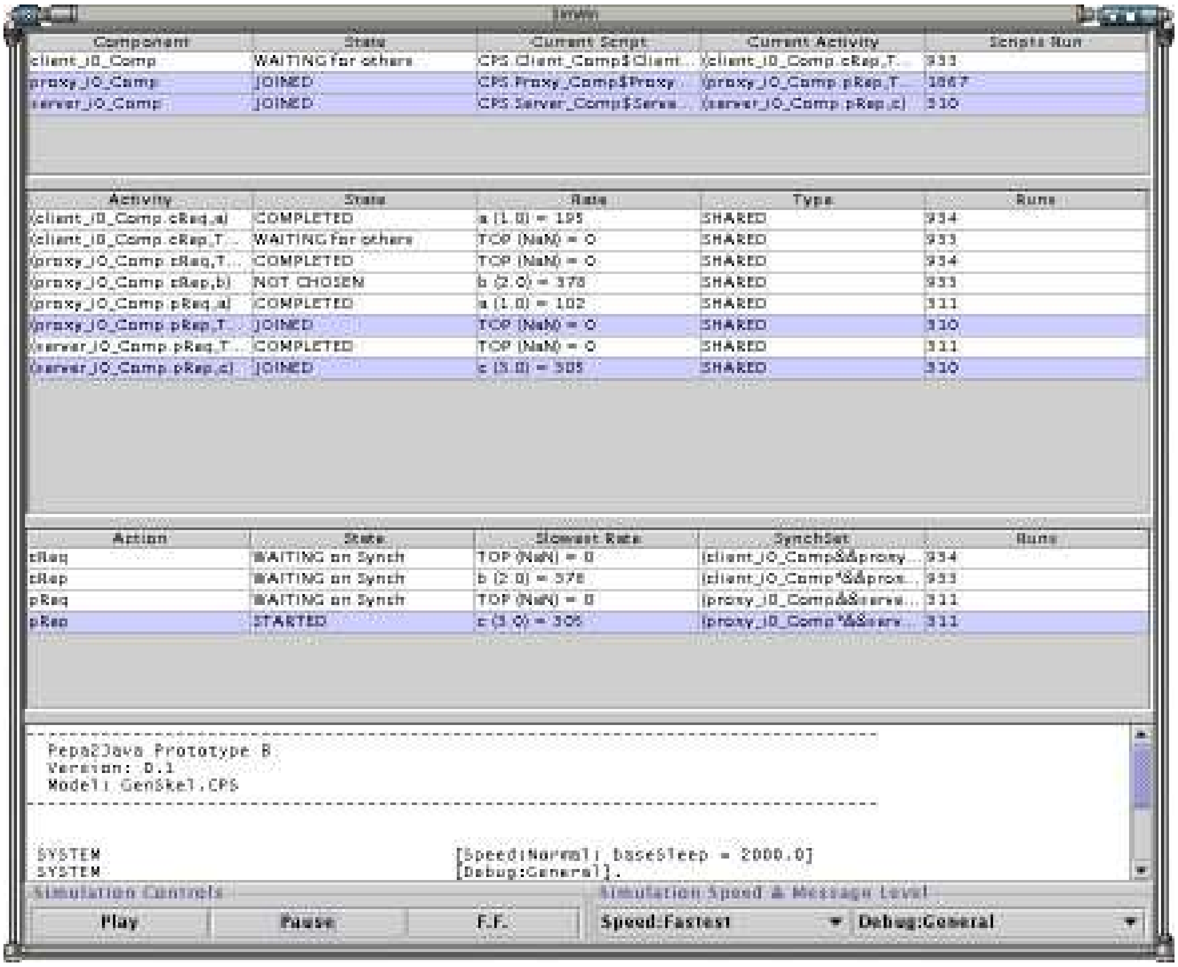

Figure 5.4: The SimWindow: Running the Client-Proxy-Server model

5.11.2 The SimWindow

The SimWindow is a GUI intended to allow the user to understand the state of the running model, pause and resume simulation, interrupt sleeping Actions, and adjust the execution speed and the level of debug messaging.

All the elements in the window, including the window itself can be resized to allow the user to monitor the parts she is interested in even in particularly large models. As can be seen in figure 5.4, the initial space allocated to the various tables is proportional to the number of rows in each of them, so that if possible, all elements may be monitored at the same time.

Consulting figure 5.4, there are three main parts to the SimWindow, the state tables, the debug message panel and the control panel. There are three state tables, each displaying information on the three main different types of PEPA objects of the model.

5.11.2.1 The System State Tables

The top table displays information on the Components running in the system, giving the state they are currently in, the name of the script they are running, the current Activity (if any) they are participating in, and the number of scripts they have run. Possible states are things such as "Waiting for others", "Joined" and "Choosing". To draw attention to the active elements, Components that have joined an action are highlighted.

The second table displays information on the Activities in the system, giving their state, their currently assigned Rate, their type (shared or individual) and the number of times they have run. Activities which are in the state "Joined" (those that have been selected as *Runners*) are highlighted.

The bottom table displays information on the Actions in the system, again giving their state, but also their determining Rate, the state of their synchronization tree, and the number of times they have been run. The full synchronization tree is given, and those nodes which are *ready* or *locked* are marked with an asterisk. The two most common states for an Action are "Waiting on Synch" and "Started". Those Actions which have been started are highlighted.

5.11.2.2 The Message Window

In the middle portion of the SimWindow lies the message window. Here, all system and object debug messages are displayed, along with the name and type of the object from which the message originates. By default, only system status messages are displayed, but the level of detail can be modified below in the control panel, to the point where a great deal of information can be discovered on exactly what is happening not only in the PEPA model but also what the API's methods are doing below the surface. Set to the least discriminating level, for example, lock requests and releases, the status of all Activities in a `choice` call and the eventual outcome, and even Rate comparison messages are available. This is useful if the model is behaving incorrectly or unexpectedly.

5.11.2.3 The Control Panel

The controls available are *play* and *pause* for resuming and pausing the running of the simulation, and *interrupt* for prematurely ending long-running Actions. The *interrupt* command will wake any *sleeping* Thread. Because the `Thread.suspend` and `Thread.resume` methods are depracated since the release of JDK1.3, it is possible that the Simulator package may display warnings on compilation. This is because careless use of these methods can lead to deadlock. However, in this package the use of these methods is the best way to control the simulator's behaviour and is perfectly safe.

Additionally, there are two pull-down lists where simulation speed and the level of debug messaging can be specified. There are five speeds to choose from which affect the multiplier used to determine the sleep-time of Actions as calculated by Rate objects. The level of debugging too can be set to five levels of detail, ranging from no feedback to very detailed feedback.

5.11.3 Simple Analysis

Finally, when the simulation is halted (by closing the window), another window (fig 5.5) will pop-up, displaying some basic statistics to allow analysis of the model. Unlike solving for steady-state in Markov chains, this analysis is much more simple. Because the model runs concurrently, and is therefore made up of multiple Threads of execution, rather than the percentage spent in each state, the percentages spent running each Action are given. Each time an Action runs, it adds the length of time it will sleep to the global sleep-time total, as well as its own internal sleep-time total. Then, at the end of the simulation, it calculates how much time, as a percent of the system total, it accounted for in the global running time. Note this does not include the time taken waiting for synchronization or internal system actions (such as acquiring locks, status checking, etc.). Also displayed are the mean sleep time of each run and the number of times each Action (shared and internal) was executed. This facility is meant only to provide basic feedback to help the user determine whether the model is working

Figure 5.5: The SimResults Window: Client-Proxy-Server model results

correctly or not, and to see whether the PEPA2Java results are similar to the PEPA Workbench results.

Chapter 6

Implementation of the Translator

This chapter starts by describing how the Translator works and concludes with how to go about using it.

6.1 Overview of the Translation Process

Figure 6.1 shows the four steps involved in translating a PEPA model into its PEPA2Java equivalent. For the lexing and parsing stages, it was possible to build upon previous PEPA-related work and also make use of standard lexer- and parser-generators. The latter stages— analysing the parser output and extracting the necessary information to build the data structures for the final Java code output— were the most challenging part of creating the Translator.

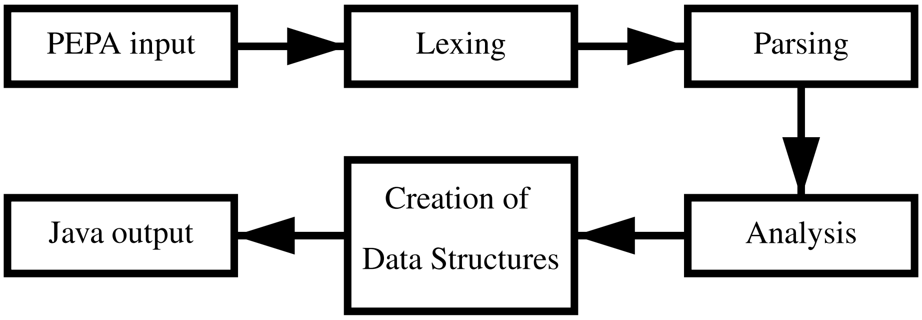

Figure 6.1: The Translation Process

6.2 The Translator Grammar

The PEPA grammar that the Workbench accepts, in EBNF notation [Gilmore, 2001], is given in figure 6.2. This same grammar is used in the Translator's parser. However, the Workbench does not accept the full grammar and neither does the Translator. Rather, it accepts a subset of the grammar, known as guarded PEPA. Specifically:

- As justified in section 3.1.3, there is no provision for *Hiding* in PEPA2Java.
- The composition line may contain only *Cooperation* constructs and Component identifiers.
- The composition line is also the *only* place that *Cooperation* constructs may be placed.
- *Prefixes*, additional *Choices* or grouping structures holding either type of construct are the only valid branches of choice constructs. This is because the `choice` method chooses by comparing the determining Rate of the Action of the Activities of each branch. *Choices* may have any number of branches and may also be nested as long as there is always be an Activity to evaluate for each branch. For example,

  `#A = (a,1).A + (b,2).B + (c,3).((d,4).D + (e,5).E)`

  is a valid construct, but

  `#A = B + (c, 1.0).C`

  is not, regardless of what B defines.

6.3 Lexing

Developing a lexer and parser for the PEPA input was straightforward because the PEPA2Java Translator accepts the same input syntax as the PEPA Workbench (see figure 6.2).

```
      program   ::==   declaration + composition
  declaration   ::==   #id = seq_component ;
seq_component   ::==   seq_component / { [idseq] }          [hiding]
                  |    seq_component + seq_component        [choice]
                  |    ( id, rate) . seq_component          [prefix]
                  |    id                                   [variable]
                  |    ( seq_component )                    [grouping]
  composition   ::==   seq_component < [idseq] > seq_component
                  |    seq_component < [idseq] > composition
                  |    ( composition )
        idseq   ::==   id
                  |    id, idseq
         rate   ::==   id
                  |    int
                  |    infty
           id   ::==   alphanumeric sequence
          int   ::==   unsigned numeric sequence
```

Figure 6.2: The Java PEPA Workbench's PEPA Grammar

Graham Clark created a lexer for the PEPAroni tool [Clark et al., 1999] using a JLex-like tool, which allows the quick generation of Java lexical analysers. The lexer was also incorporated into the PEPA Workbench and is now again used in the Translator, as the scanner for the parser.

6.4 Parsing

Unlike the lexer, the parser could not be used as it was. The previous PEPAroni work was of use, however. The parsers used in PEPAroni, the PEPA Workbench, and this Translator are all generated by the Java CUP program from the same grammar. Whilst the three parsers use the same grammar, what they do with the input and the objects they create are different. For example, the Translator needs to create a completely different type of object and perform different analysis than the Workbench does when it comes across a Rate definition, an Action or the constructs in the composition line.

The CUP program takes as input a grammar specification with embedded commands for each rule, and generates the Java parser accordingly. The definition of the grammar can be quite tricky, so being able to re-use the grammar whilst defining different actions sped its implementation.

The parser creates five types of object:

ProcObj Process objects are linked to each other in a tree-like structure and represent the PEPA constructs *Prefix*, *Choice*, *Hiding* and *Cooperation*.

CompScript A Component Script object is created for each process definition line. It consists of an identifier and the process tree (consisting of ProcObj objects). These definitions are used to specify a Component's behaviour, and are named to reflect the fact that they will be translated into the CompScript classes in the final Java output.

RateObj Rates are created from rate declarations (e.g α=1.0) and within *prefix* constructs, where they may be given as unspecified ($\top$, `infty`), as a numerical value (e.g. 1.0), or refer to a previously declared Rate (e.g. α).

ActivObj When a *Prefix* is found, an ActivObj is created which holds two references— one to a RateObj representing the rate at which the Activity runs, and the other to an ActionObj which represents the global Action this Activity participates in. If the name of the Action does not correspond to an already-defined ActionObj, a new one with that identifier is created.

ActionObj Unlike Rates or Activities, the set of Actions may not contain duplicate identities. Each Action is a global entity, even when they are executed without synchronisation. Therefore, when the parser comes across an Action, if the Action has already been defined, a reference to the existing Action is returned. If the Action is not already defined, a new ActionObj is created.

ActionSet ActionSet objects are created within *Cooperation* constructs and define which Actions Components must synchronise on. They can also be used in *Hiding* constructs, but PEPA2Java accept these constructs (see section 3.1.3).

These objects are either created as components of the ProcObj trees, or are put into static vectors so that they can be easily referenced in the analysis stage. Note that no Component objects exist yet— the parser returns as it's final result a ProcObj, which corresponds to the model's composition line. This composition line will be analysed to create the Components and the synchronisation set trees.

6.5 Analysis Algorithms

Once the parsing has been completed, the input has been transformed into a collection of Java objects holding the model's details. Several steps need to be performed to manipulate this data into the correct form for outputting a PEPA2Java skeleton.

6.5.1 SSNode

One of the most challenging problems is that of generating the PEPA2Java commands for building correct synchronisation trees from the composition line. The composition line returned by the parser is of a tree form. For example, take the the composition

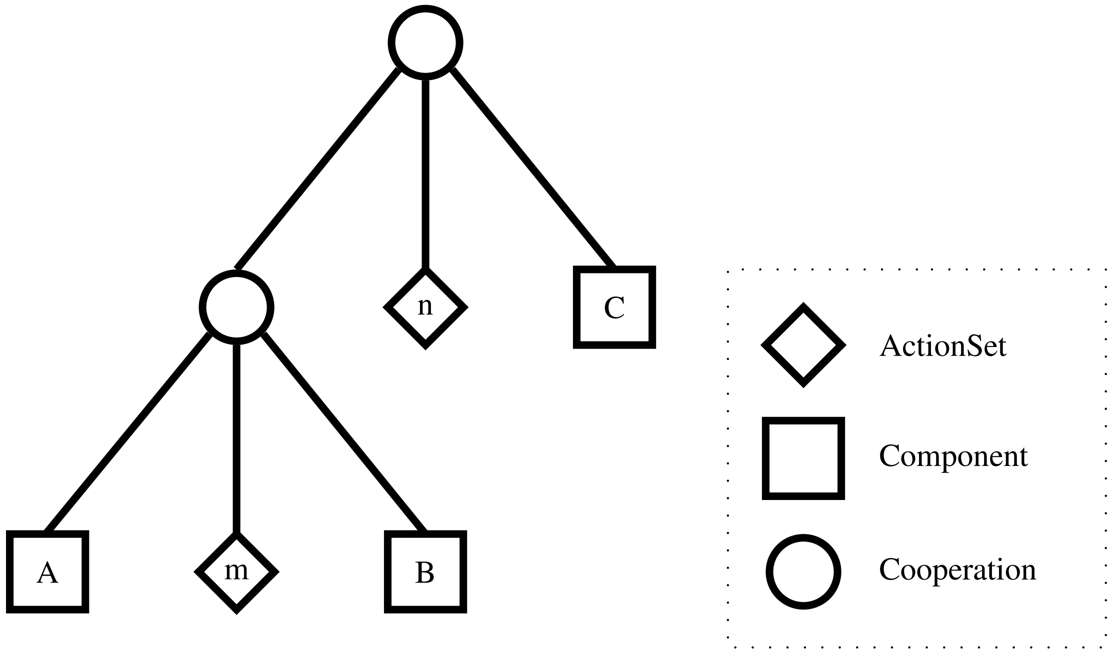

Figure 6.3: The data structure created from $A < m > B < n > C$

line:

`A <m> B <n> C`

The parser will return the tree as shown in figure 6.3. For PEPA2Java, a `setSynch` method call must be made by each Action, to create the synchronisation tree. To generate this call, the algorithm goes through two stages— create a full synchronisation set tree, and then prune the tree. The four steps for generating the full synchronisation tree from the composition line tree are (for each Action):

1. Set a SharedAction flag to false.

2. Start at the top of the composition line tree (fig. 6.3).

3. If the current node is a *Cooperation* node:

- If the current Action is in this node's ActionSet, return a new AndNode in our synchronisation tree. Also, set the SharedAction flag to true.

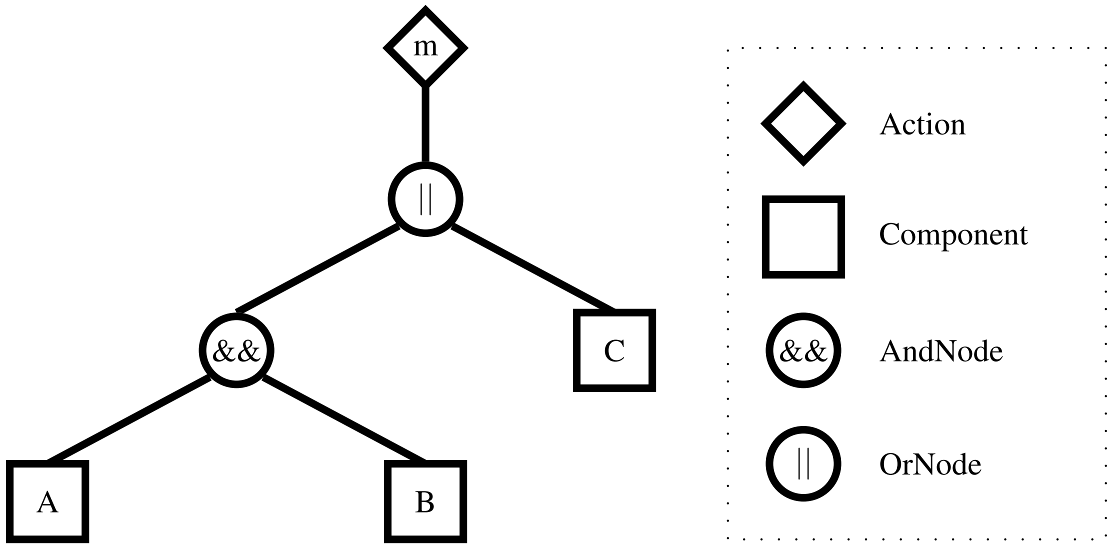

Figure 6.4: Action M's Unpruned Synchronization Tree for $A < m > B < n > C$

- If the current Action is not in this node's ActionSet, return a new OrNode in our synchronization tree.
- Set the left and right child to be the return result of a recursive descent into those nodes.

4. If the current node is a Component identifier, return the reference to it.

For Action m, this would return the unpruned tree shown in figure 6.4. The next step is to prune the tree to contain only Components that cooperate on this Action. If the SharedAction flag is still set to false, this is an individual Action which requires no cooperation— the synchronization tree is scrapped. Otherwise, we set the Action's synchronization tree to be the result of the method `unpruned_root.prune`, which will return a pruned version of the tree. The three steps in the pruning algorithm are:

1. Call the `contains(thisAction)` method on the left child node. This method returns true if this sub-tree contains a Component that contains an Activity that participates in this Action.

- If the method returns false, return the result of `right_node.prune`— this node, and it's left sub-tree have been pruned. The ends the algorithm for this node.

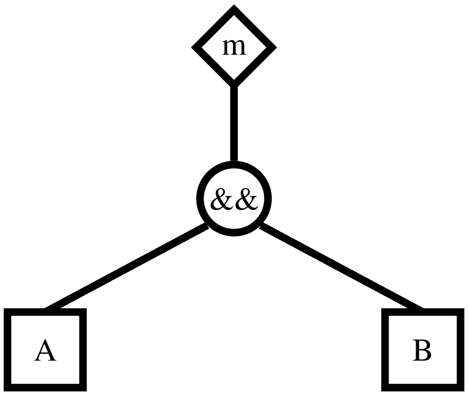

Figure 6.5: Action M's Pruned Synchronization Tree for $A < m > B < n > C$

2. Call the `contains(thisAction)` method on the right child node.

- If the method returns false, return the result of `left_node.prune`.
- If no result has been returned, then both children must have returned true for `contains`. The left node is set to the result of `left.prune` and set the right node to `right.prune`.

3. Finally, return `this` as the result of `this.prune`— this node has not been pruned.

Action M's resulting pruned tree is shown in figure 6.5. Because the right child of the OrNode did not contain any Activities participating in Action M, but its left subtree did, the left subtree was returned as the result of calling `prune` on the root. The OrNode and its right subtree were dropped. The left subtree survived intact because both its children contained Components that participate in Action M. Performing these two algorithms on all the Actions in the model will create the synchronization set trees needed to define the PEPA2Java API's abstract `initSynchs` method in the PepaSystem object.

6.5.2 Creating Components and Scripts

Apart from generating the synchronization set trees, the composition line is also used to create the Components of the system. Each Component identifier found in the composition line will be made into a running instance of a Component-extending class. There may be multiple instances of a single class as is the case in this composition line from the model example given before:

```
(Client || Client) <serve> Server
```

In this example the multiple instances of `Client` will be given names such as `client_i0_Comp` and `client_i1_Comp`, and will be of the `Client_Comp` class.

For the initial behaviour of the Components, equality of identifier is searched for between the Component definition in the composition line and the sequential process definitions. When the sequential process definitions are turned into CompScript objects, the `setStartScript` method is used to set the reference to the initial script that Component should execute.

The parser returns a tree made of ProcObj objects for each sequential definition. Turning the tree into an `actions` method of a CompScript object is difficult, especially as correct nesting is important not only for proper execution but also for comprehensibility. Good tabbing (e.g. indentation of nested blocks) is also equally important for comprehension by the user.

First, the Component's behaviour trees are traversed to gather a list of all the Rates, Activities and CompScripts this Component uses. These will be defined as fields in the Component class.

Next, to generate the CompScript output classes, the `CompScript.createScript` method is called on each CompScript object. This method will recursively traverse the `ProcObj` tree generated by the parser and translate it into PEPA2Java commands. A small example should help to illustrate. Consider the input model:

```
#A = (m, 1.0).A + (n, 2.0).A + (o, 3.0).((x, 0.5).A + (y, 0.5).A);

A
```

This results in the CompScript `A_Script` given in figure 6.6. Handling references to sequential process definitions is simple—they are translated into `return` statements that return the name of the CompScript object which defines the process behaviour.

This exits the current `actions` method and causes the parent Component to call the `actions` method of the newly returned CompScript.

For *Prefix* constructs, the result is also quite simple— the appropriate `joinAt(Rate)` method call is inserted.

Processing a *Choice* construct is more difficult, because any number of branches is allowed. However, since each *Choice* ProcObj has only two branches, some account must be kept of whether each encounter with another *Choice* ProcObj is part of a new choice block, or adding another branch to an existing choice block. Therefore, as well as receiving a reference to the current ProcObj, a reference to the previous ProcObj is also passed.

Additionally, a ChoiceData object is created for each new choice block to hold information on the Activity of each branch and its corresponding Rate. This is used to create the `ch_Act` and `ch_Rate` arrays defined in each CompScript. The arrays are passed to the `choice` method at the top of each choice block. The ChoiceData object is also used to guarantee a unique name is given to each choice block. It also holds the `case` number of the current branch and registers tabbing information for formatting the output.

A ChoiceData object is passed along with the current ProcObj and the previous ProcObj as arguments to the `createScript` method, which recursively descends the tree and eventually returns a String which will be written to disk as the final CompScript object.

6.6 Form of the Output

To see an example of the type of output the Translator creates, consult Appendix C. A file holding the Component class and its nested CompScript classes will be created for each of the Components. Also, a PepaSystem-extending class will be created in a file, by default named Sim (see Appendix D for a full list of Translator defaults). These files define the behaviour of the system. Important to note is that all these files will *import* one of the two PEPA2Java API implementing packages. Therefore, it is important to make sure the API package is in the *classpath* for compiling and running.

```
public class A_Script implements CompScript {
   Activity [] ch_Act_0 = {m_Activ, n_Activ, o_Activ};
   Rate [] ch_Rate_0 = {new Rate(1.0), new Rate(2.0), new Rate(3.0)};
   Activity [] ch_Act_1 = {x_Activ, y_Activ};
   Rate [] ch_Rate_1 = {new Rate(0.5), new Rate(0.5)};
    public CompScript actions () {
        switch(choice(ch_Act_0, ch_Rate_0)) {
        case 0:
            m_Activ.joinAt(new Rate(1.0));
            return a_Script;
        case 1:
            n_Activ.joinAt(new Rate(2.0));
            return a_Script;
        case 2:
            o_Activ.joinAt(new Rate(3.0));
            switch(choice(ch_Act_1, ch_Rate_1)) {
            case 0:
               x_Activ.joinAt(new Rate(0.5));
               return a_Script;
            case 1:
               y_Activ.joinAt(new Rate(0.5));
               return a_Script;
            }
            throw new Error ("Problem with choice: no valid case
returned!");
        }
        throw new Error ("Problem with choice: no valid case returned!");
    }
}
```

Figure 6.6: CompScript generated from: $A = (m,1.0).A + (n,2.0).A + (o,3.0).((x,0.5).A + (y,0.5).A)$

6.6.1 Generating Valid Class, Object and Package Names

To generate names for the different classes, objects, files and directories the Translator takes the identifiers from the PEPA model and performs some manipulation. Naturally, all the names to be used in the Java output must be legal— a Java identifier may *begin* with any alphabetic character, a currency symbol (such as "`$`") or an underscore ("_"). The rest of the identifier must also be made up of these characters but may also include numeric characters.

Therefore the first step is to pass the PEPA identifier to the `legalize` method. This method takes as input any string and returns a legal Java identifier by removing illegal characters or replacing them with legal ones. If the name cannot be legalized, an error is thrown informing the user to modify the identifier in the PEPA input. A character that is often seen in PEPA models is "`'`", denoting derivative behaviours. Also, some of the models include the dash symbol ("`-`"). These characters may not be used in Java identifiers (or as file/ directory names, for that matter) and are therefore replaced by the underscore character. The result may be used as both legal Java and file system identifiers.

In line with common coding practice, class identifiers begin with an upper-case letter, whilst object identifiers begin with a lower-case letter. Additionally, objects and classes are suffixed with an abbreviation indicating their type. These suffixes are set as `final static` strings in the `Main` class of the Translator. Other defaults (Appendix D) control the name of the PepaSystem-extending class, the default output directory and the names of the implementing packages. The full list of the Translator's default constants is given in .

Finally, because multiple instances of the same Component may exist in a system, each Component instance has the infix "`_iX`" inserted between the identifier and the type suffix, where `X` is incremented for each new instance. This scheme ensures that all objects are easily identifiable. The suffix clearly identifies the object class, regardless of the context in which it is used. Most importantly, however, this scheme also ensures that the unintentional clashing or multiple-declaration of identifiers is minimised. The package name for the output is derived from the *legalized* version of the PEPA model's filename.

6.7 Using the Translator

The Translator can be interfaced via either the command line or via the P2J Dialog window (see section 6.7.3). Either of these methods gathers the necessary information to begin translation and spawns a new Translator. Due to the use of some static data structures, only one Translator can be started at any one time. The following information is gathered from user interaction with either the dialog window or the command line invocation and passed to the Translator:

Filename The path to, and the name of, the PEPA model file that should be translated. The filename is also used to generate the Package name (see section 6.6.1);

Output Path The path to the directory in which the new model should be created. In accordance with Java package naming rules, a subdirectory will be created with the name of the package. Note that whilst the subdirectory will be created, the output path must be an already existing directory. Otherwise, the translation will fail.

Output type A flag is passed that is used to determine which API-implementing package the model should run with. If the flag is set to true, the Simulator package is imported, otherwise it is the Barebones package which will be imported. The import statements may of course always be changed manually later.

Overwrite flag If this flag is set to true, any files in the output directory with the same name as newly created files will be overwritten. If set to false, the Translator will exit without overwriting the existing files. The default value is false. Note that the Makefile entry will be created/ appended regardless of this flag.

Write files flag If this flag is set to false, the Translator will perform as usual, except that no files will be written to disk. This is used mainly for debugging and accordingly is not very useful unless the debug level is set high.

Simulator Speed and Debugging Level The initial speed and debugging level at which to run the Simulator. These have no effect on the Barebones package.

Verbosity Additionally, the debug level of the Translator may be set on initialisation.

6.7.1 Automatically Generated Makefiles

In order to make the generated PEPA2Java model package easier to compile and run, a `Makefile` is placed in the output path directory (usually `models/p2jgens`), if there is not already one there. If a duplicate entry is found in the already existing Makefile, it will be replaced. The *classpath*, the directory path to the Workbench and the commands for running the Java compiler and runtime are held as default constants in the Main class of the translator (see Appendix D).

This `Makefile` contains a `target` entry for each model generated. In order to compile and run any model found in a package subdirectory of the one holding the `Makefile`, the command "*make run_ModelName*" can be issued from the command line. This is useful for running Barebones models because these models cannot be run from the PEPA Workbench— as they contain no GUI, there would be no way to halt a running model. Additionally, the last model translated is also the default for the `Makefile`, so typing "*make*" is sufficient.

6.7.2 Running the Translator from the command line

The Translator may be initialised from the command line, in case the only required argument is the name of the input file. The Translator should be run by issuing the command:

```
java Pepa2Java.translator.Main -in <PEPA input file>
```

Ensure the *CLASSPATH* environmental variable is defined to include the two jar files in `pepa/Pepa2Java/lib`. Alternatively, the *classpath* may be passed directly to the java interpreter with the `-classpath` argument.

The optional command line arguments are:

```
-in <Pepa Input File> : File to translate (Required)
-out <output path>    : A <packagename> subdirectory will
                        be created here.
-skel                 : Output Skeleton System, not Simulation.
-ow                   : If set, overwrite files in the output
```

```
                              directory.
  -debug <0-6>              : set Debug Message level 0(none) - 6(all)
  -dry                      : Dry run - write only to screen.
  -simDebug <0-5>           : Simulation debug level
  -simSpeed <0-4>           : Sim. speed: 0 = slowest, 4 = fastest
```

These are used to set the Translator's parameters as described in Section 6.7. Any messages will be printed to the screen.

6.7.3 Integration with the PEPA Workbench: The P2J Dialog Window

An easier way to control the Translator is by accessing it through the PEPA Workbench. Selecting *Translate model* from the *Pepa2Java* will open the PEPA2Java Translator Dialog window, as shown in figure 6.7.

The Translator Dialog window is passed whichever PEPA model file is currently open in the PEPA Workbench— the current file is displayed at the top of the Dialog window. The window allows the user to specify the output directory, whether to use the Simulator or the Barebones package, the initial Simulation speed and Debug level. The text area displays any messages from the translation process. At the bottom of the screen, the user may select to overwrite existing files and to get extra information about the Translation process as it is carried out. Finally, action is invoked by the two buttons at the bottom of the window. Selecting the first will cause the Translator to process the model and output the PEPA2Java equivalent. Selecting the second will compile and run the resulting model. Naturally, the model cannot be compiled or run before it has been translated. On translation, a dialog window will pop up to inform the user if translation was successful or, if not, the cause of the failure.

Though the Dialog window allows a more user-friendly interface with the Translator, it comes at a cost. Because calling the Java Compiler from within the runtime environment is not supported in most versions of the Java Developer Kit, the compile and run button instead causes the runtime to execute the "*make run_ModelName*" command as a new system process. There are two problems with this approach— firstly, the

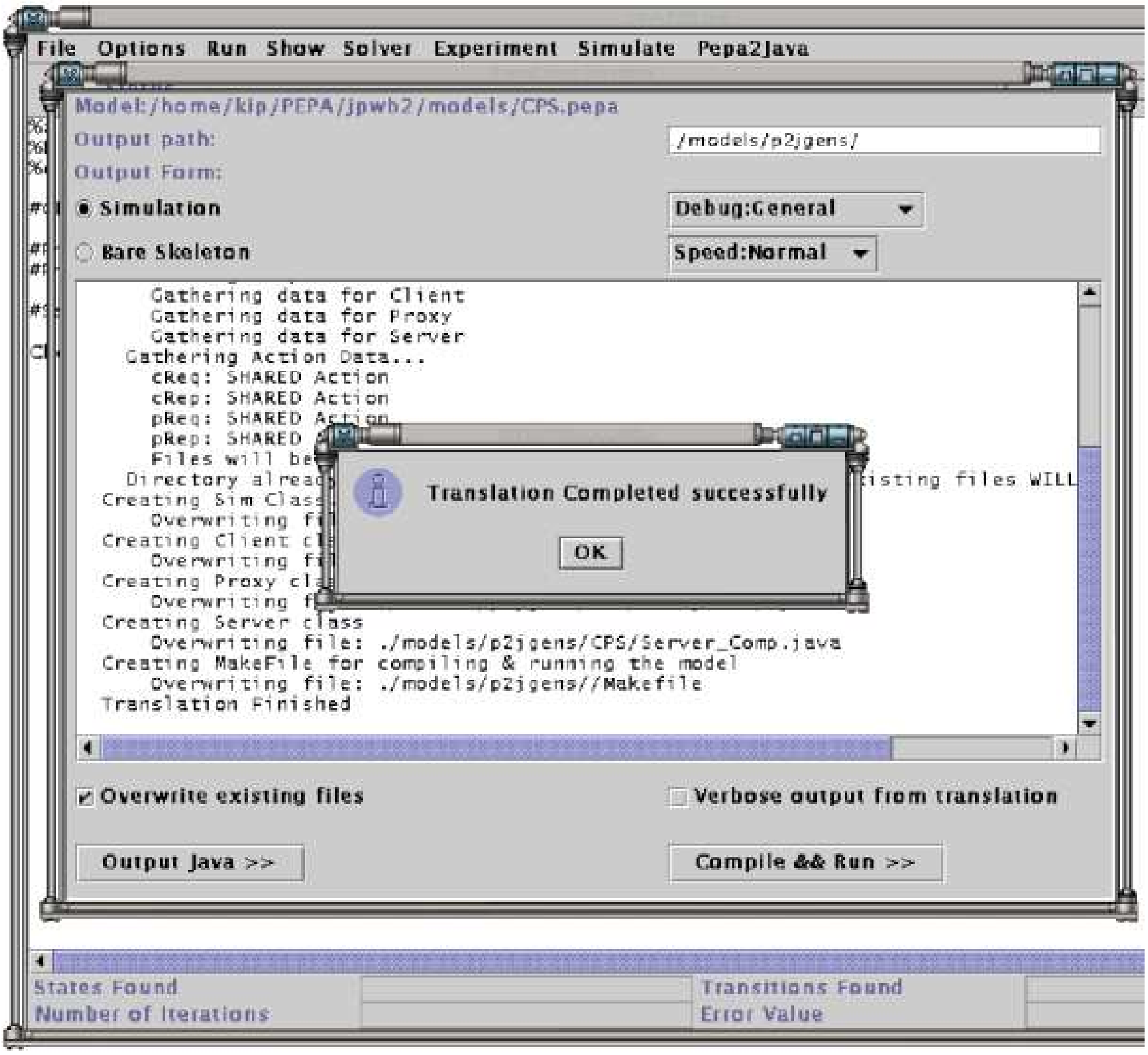

Figure 6.7: The PEPA2Java Translator Dialog Window

compile and run command can therefore *only* be used in systems with *make* installed. Therefore, the Translator cannot automatically compile and run the model in some less developer-friendly operating systems, such as Microsoft Windows. Secondly, and more importantly, any error messages from the compilation or running process are lost.

If the model does not run after pressing the compile and run button, or it is acting strangely or freezing, run the "*make*" command from the command line in the output directly to see any error messages.

Chapter 7

Evaluation

7.1 Quantitative Comparison to the PEPA Workbench

One goal of testing is to discover whether or not a PEPA2Java translation maintains the same running properties as the PEPA model's *steady-state* solution as calculated by the PEPA Workbench.

It is not evident how to compare the Action *sleep-times* provided by the Simulator's *Simple Analysis* to the *steady-state* solution analysis of the Workbench. This is because the *steady-state* solution gives the proportion of time the system is in each state. For example, for a basic two-component model:

```
%m=4;
%n=5;

#P = (read,m).(write,n).P;

P || P
```

the PEPA Workbench calculates the various states of the system to be:

```
1 P || P
2 (write,5.0).P || P
3 P || (write,5.0).P
```

```
4 (write,5.0).P || (write,5.0).P
```

and gives the *steady-state* solution as:

```
1  0.30864197530864207
2  0.24691358024691354
3  0.24691358024691354
4  0.19753086419753074
```

where the first number refers to the state as calculated above, and the second gives the mean of the residence in that state— the proportion of time the system is in that state. For example, the above system is in state 4 (where both *P* Components are independently executing Activity *write*) 19.75% of the time. State 1 is the system state when both Components are executing the *read* Activity. Note that each state is a "global" snapshot, describing the behaviour of every Component in the system at that point.

It can be very difficult to interpret the result of the *steady-state* analysis of even moderately-sized models, as PEPA is prone to "state-space explosion." For example, by adding an extra *P* Component to the system, which is wholly independent to the other two *doubles* the number of states from four to eight. If an intermediary Activity is added (i.e. read, *analyse*, write), the number of states jumps from eight to 27. The ability to get such exact results is very useful but if one considers that the *PAPM* model has six components but 1557 states and 6155 possible transitions between states, it becomes clear that analysing such in-depth results for more complex systems can be very tricky.

The results returned from the PEPA2Java Simulator are quite different. It does not return the type of global snapshot that the Workbench does— hence the name, *Simple Analysis*. It returns the behaviour of the Actions— the parts which will perform the actual work of the finished system. Some sample results of it running the same model are:

```
Total sleeptime: 170.871 seconds
Actions:
```

```
Indiv. Activities:
(p_i0_Comp.read,m)   sleepTime: 47.432 secs (27.7589527 %)
                     Runs: 91(AVG 0.5212308 secs/run)
(p_i0_Comp.write,n)  sleepTime: 38.78  secs (22.6954837 %)
                     Runs: 90(AVG 0.4308889 secs/run)
(p_i1_Comp.read,m)   sleepTime: 44.374 secs (25.9692985 %)
                     Runs: 101(AVG 0.4393465 secs/run)
(p_i1_Comp.write,n)  sleepTime: 40.285 secs (23.5762651 %)
                     Runs: 101(AVG 0.3988614 secs/run)
```

We can see from the results how much of the total Action running time this particular Action accounted for, how many times it ran and the average duration of the runs. For example, we can see that the second instance of *P* (`p_i1_Comp`) running Action *write* accounted for approximately 23.58% of the total running time. It does not give information regarding which Activity the other Component (`p_i1_Comp`) was running at the same time.

Providing the same complete state analysis that the Workbench provides would be useful and is therefore a suggested extension to the Simulator package (section 7.5).

There is an alternative to using the *steady-state* results of the PEPA Workbench. The Workbench's *simulation* menu allows a model to be run as a simulation (using Edinburgh's SimJava package). The results from the *simulation* include a summary of how many times each Action ran. So although it is not possible to quantitatively test whether the PEPA2Java model is behaving as the *steady-state* solution predicts the model should, it is possible to test whether the PEPA2Java system runs as the Workbench *simulation* does.

However, this may not be as useful as it first seems. Firstly, the results of the *simulation* give the number of times each Action has been run, but does not, in the case of individual Actions, distinguish between the different Components. The results for one run are:

```
event (read,4.0) fired, times =1505,
event (write,5.0) fired, times =1506
```

The results for this model are hardly surprising as each *read* must be followed by a *write*. More useful would be the residence times for each Action but unfortunately these are not given. Instead, the mean residence times are given for the states as defined for the *steady-state* solution. The Workbench allows the two to be easily compared.

It might still be possible to use the simulator's results as a "bridge" between the PEPA2Java results and the *steady-state* solution. If:

- the Workbench's *simulation* results (for the number of times each Action is run) are equivalent to the PEPA2Java *Simple Analysis* results, and;
- the Workbench's *simulation* results for state residence means are equivalent to the Workbench's *steady-state* solution's calculated state residence means,

then it might be possible to argue that the PEPA2Java system is behaving as the *steady-state* solution predicts the PEPA model should, by using the Workbench's *simulation* results as a common comparison measure.

However, this tenuous "transitive" comparison of results falls at the first hurdle. No statistical test is needed to realise that the results (given in Table 7.1) comparing the mean state residences of the *simulation* and the *steady-state* solution for the simple two-component model (given at the beginning of this section) are different. There is no chance that these two are equivalent. The results in Table 7.1 are for a relatively short run of the *simulation* (<5 minutes) but the model is also very basic. It is stated [Fotis, 2001] that for very long simulation runs (>20 hours), even large state space models are completely sampled and that the closeness of fit of the *simulation* and *steady-state* results is directly proportional to the simulation length. Unfortunately it is at this time wholly inpractical to run the simulation for such long periods of time to establish what would be at best a tenuous link between the PEPA2Java results and *steady-state* results.

Therefore, comparing the PEPA2Java system to the Workbench's *simulation* results still does not get us closer to the real goal— discovering whether or not a PEPA2Java translation maintains the same running properties as the PEPA model's *steady-state* solution.

This goal is not achievable with the current PEPA2Java Simulator implementation.

In order to do this, the Simulator package would need to be extended to provide mean residences in each of the states as defined by the Workbench (section 7.5.1).

That said, it *has* been possible to appraise the important aspects of the performance of the API and to identify areas for extension and improvement.These are described below.

7.2 Qualitative Evaluation

Even if it is not possible to test for statistical equivalence between the PEPA2Java output and the *steady-state* solution to a model, qualitative evaluation is still feasible.

To achieve this, the Translator will be used to translate the majority (just under 50) of the models included with the PEPA Workbench distribution to see if it can successfully create a PEPA2Java equivalent. If a model can be successfully translated, the resulting Java code will be compiled and run, using the Simulator package.

The *SimWindow*, debugging messages and *Simple Analysis* window results can then be examined to judge approximately how the Simulator's execution compared to the expected behaviour. The expected behaviour is found through the manual examination of the PEPA model and the *steady-state* solution found by the PEPA Workbench. The clearest sign of a problem are Actions that are being chosen much more frequently than others, or which are rarely/ never chosen. In either case, the model would need to be examined to see whether this is the correct behaviour.

Special attention will be given to the larger models, such as the *Big* and *Workcell2* models as well as the *PAPM* and *TOMP* series of models. Also, any models that fail to translate, compile or run correctly will be more closely examined to discover the

State	Workbench Steady-state	Workbench Simulation
(write,5.0).P ||(write,5.0).P	19.753086419753075 %	28.693142 %
P ||(write,5.0).P	24.691358024691354 %	12.711846 %
(write,5.0).P ||P	24.691358024691354 %	12.228244 %
P ||P	30.864197530864207 %	46.366653 %

Table 7.1: Comparing the Workbench's *steady-state* and *simulation* results

reason for the problems.

7.3 Results of Model Tests

The Translator managed to produce compilable and runnable equivalents for **every** valid input model tested, which is an excellent result. Three of the models ("test19", "test21" and "tools") were not evaluated because they make use of non-exponential distributions. They specify *Normal* or *Uniform* distributions. Although the PEPA2Java API does not currently support these alternative distribitions, it is a trivial matter to add them (see section 7.5.2).

Secondly, one of the other models, "test4c" performs "rate maths" in a rate definition. The parser does not allow this. Because there is already support for "rate math" when it occurs directly in a prefix construction, modifying the CUP parser-generator's grammar-defining file to allow this should be straightforward. However, as this construct occurs in only one of the models, it is not a high priority.

In short, the PEPA2Java Translator can produce compilable and runnable code from any model that the Workbench can read, with one exception ("test7", mentioned below).

Whilst the models can be successfully transformed, compiled and run, some of them do not execute as the Workbench predicts they should. The full list of results can be found in Appendix F. The following models are the ones which caused problems. In increasing order of severity, they are:

Deadlock Running this model results in deadlock but (the clue is in the name) this is the correct behaviour.

Maple This model refers to the rates α, β, γ and δ but does not define them anywhere. The model was completed by specifying all these rates to be equal to 1.0. This version was saved as `maple-mod.pepa` and now works as it should in both the Workbench and as a PEPA2Java system.

test9, test10, test17 These models deadlock when running because they are **invalid**. Specifically, they define and try to start Actions which do not contain any active participants.

test7 The composition line of this model refers to Actions that are not used anywhere in the model. The Translator creates code to initialise these Actions but compilation fails because the Actions are not defined anywhere.

PC-LAN4 The PC-LAN 4 system demonstrates the second short-coming of the choice algorithm, more serious than the first. This problem is related to the race condition problems of "Big". In this case, under certain circumstances, a branch may *never* be chosen. As the models are so similar, this same starvation occurs in PC-LAN 6. One potential solution is to modify the model itself to fix the problem— a modified version of the PC-LAN 4 model named "p2j_PC-LAN4-mod.pepa" was created and runs without problems. The problem is explained in full in the next section.

Big The "Big" model results in deadlock.The problem concerns the mechanisms used to replace race conditions and determine choices. It is also described in the next section.

The PEPA2Java translations of all the other models performed as expected.

7.4 Evaluation of Results and Suggestions for Improvement

Whilst the PEPA2Java API-implementing packages do a fine job at running almost all the PEPA models, some problems have been found. These are caused by one aspect of the implementation— simulation of the race condition as described in section 5.4.

The choice algorithm was a very difficult one to develop and it is still not perfect. It involves complex checks for readiness of branches, locking and unlocking of large numbers of objects, wait/ notify mechanisms and prioritizing of branches. Despite the careful design of the algorithm, there remain two flaws, described below.

7.4.1 Drawbacks to the "choice-pause" Mechanism

The choice-pause mechanism described in section 5.4 leads to undesirable variation in execution patterns. For example, the ability of shared Actions to run when competing against individual actions depends on two factors: the speed of the simulation and the length of the choice-pause. The shorter the choice-pause and the slower the simulation, the less chance there is of all the members of a synchronisation set arriving in time (before the choice-pause expires) to be considered as a "ready" branch. As it is always "ready", individual Actions have an unfair advantage, regardless of the branch rates.

For example, with the model:

```
#A = (m, infty).A + (n, infty).A;
#B = (m, 1.0).B + (i, 1.0).B;
#C = (n, 1.0).C;

A <m, n> (B <> C)
```

we can see the effects the value of choice-pause and the speed of simulation have on defining behaviour by examining table 7.2. The behaviour of the system is variable depending on the values chosen. This underlines the difficulties surrounding choice when true race conditions are not used.

In the example above, this implementation's particular method for simulating race conditions means behaviour is not wholly dependent on the model but also on other parameters (specifically, the value of "choice-pause"). This is undesirable but is not fatal except in the case of the two PC-LAN models.

	Sim.Speed, Choice-pause	
	Normal, 100ms	Fastest, 500ms
Action m	4.268 %	25.369 %
Action n	45.122 %	38.635 %
Action i	50.61 %	36.074%

Table 7.2: Runs per Action as percentage of all Runs

The PC-LAN models execute incorrectly using PEPA2Java. In these models, there are 4 or 6 PCs (respectively) being served by a token-ring LAN. Access to the LAN is passed around the ring of 4 or 6 PCs. If the current PC has a network request, it participates in the `serve` Action, otherwise it participates in the `walkon` Action. In either case, once the Action is complete, access to the LAN is passed on to the next PC. The model is given as:

```
%lambda=2;
%omega=3;
%mu=1;

#PC10 = (arrive,lambda).PC11 + (walkon2,infty).PC10;
#PC11 = (serve1,infty).PC10;

#PC20 = (arrive,lambda).PC21 + (walkon3,infty).PC20;
#PC21 = (serve2,infty).PC20;

#PC30 = (arrive,lambda).PC31 + (walkon4,infty).PC30;
#PC31 = (serve3,infty).PC30;

#PC40 = (arrive,lambda).PC41 + (walkon1,infty).PC40;
#PC41 = (serve4,infty).PC40;

#S1 = (walkon2,omega).S2 + (serve1,mu).(walk2,omega).S2;
#S2 = (walkon3,omega).S3 + (serve2,mu).(walk3,omega).S3;
#S3 = (walkon4,omega).S4 + (serve3,mu).(walk4,omega).S4;
#S4 = (walkon1,omega).S1 + (serve4,mu).(walk1,omega).S1;

(PC10 <> PC20 <> PC30 <> PC40)
        <walkon1,walkon2,walkon3,walkon4,
                         serve1,serve2,serve3,serve4> S1
```

The first PC starts in state `PC10`. A race condition between the internal `arrive`

Action and the shared `walkon2` Action governs which behaviour the PC exhibits. The `arrive` Action is used to model other processes on the PC issuing LAN requests. On completion of the `arrive` Action, the Component will take the `PC11` derivative behaviour, indicating it requires network access. Alternatively, if the `walkon` Action completes, this indicates access to the LAN became available to this PC when there were no outstanding requests. Therefore, it moved on the next PC in the ring.

Using a race conditions as PEPA does means that the `arrive` Action can be started whilst the the `walkon2` Action must await synchronisation with the Server Component (i.e. must wait for it to enter its `S1` state). However, once the Server becomes available, it participates in `walkon2`. If that Action completes before the `arrive` Action does, the Component will abort the `arrive` action.

The problem with translating this model into the PEPA2Java equivalent is that only "ready" branches may be chosen, as outlined in points four and five in section 5.4. The `arrive` Action is always ready, whilst the `walkon2` Action requires synchronisation. In most models, the "choice-pause" mechanism provides a chance for synchronisers to arrive but in this model the Server needs to make three other choices (with the three other PCs) before it returns to the first state. This means it needs to await three "choice-pause" expiries before returning. By that time, the "choice-pause" of this branch will of course have expired. Therefore, the `walkon` branch will never be executed, always losing to the individual Action. This is a major drawback of using the "choice-pause" mechanism in place of an interruptable race condition.

It is possible to modify the program so that it behaves in the same manner but does not require the use of a race condition. Namely, adding a separate process ("PR") to model the arrival of network requests means the Component is only committed to the "serve" Action when the "arrive" Action has completed. It does not "lock-out" the possibility of executing the "walk-on" Action. The modified model is given in Appendix E.

Admittedly, this has made the model more complex but it will now work with a system, such as PEPA2Java, where Actions may not be aborted. Note that this also means that it is not necessary to isolate the "serve" Action in a separate derivative as was the case before (using "PC10" and "PC11")— once the "serve" Action has been chosen it will not be usurped by the “walkon” Action.

7.4.2 Drawbacks to Choice Committing

This second problem is the more serious of the two and also arises from the simulation of the race conditions that govern branching in PEPA. As detailed in previous sections (3.3.1 and 5.4), tentative execution of all branches is not an option in a skeleton implementation of a model. Instead, the program must somehow choose one branch to execute but behave in the same way as the PEPA model does. In the choice mechanism there is currently a two stage algorithm of expressing readiness: pausing for “choice-pause” and then choosing the fastest “ready” branch. However, in the“Big” model, when three Components come together at one choice at the same time, deadlock may result. The conditions leading to this are quite complicated but once understood, the problem is clear.

The “choice” method utilises locking of Actions when evaluating branches to avoid the deadlock. Consider the example:

```
#A = (m, 1.0).A + (n, 1.0).A;

#B = (m, infty).B + (n, infty).B;

A <m, n> B
```

Without locking, both “A” and “B” could register interest in both Actions “m” and “n” at the same time. Without locking, both would believe “m” and “n” “ready”. “A” might choose “m” and “B” might choose “n”, resulting in deadlock. Instead, “A” will lock the two Actions and commit to one, then withdrawing its “ready” flag from the other before unlocking. “B” would then receive the locks but see only one of the Actions as “ready” and therefore choose the same one as “A”, avoiding deadlock.

The problem with the “Big” model is that there are parts of the model where three Components come together in a choice block. The participation of one (the “Node”

Component) is required by both of the other Components (the “Bidder” Components). However, the Action is exclusive— only one of the two “Bidder” Components may join. Occasionally, all three Components arive at the choice almost simultaneously.

They register their “readiness” to choose between the multiple branches. Seeing that the “Node” Component is “ready” to join either branch, “Bidder1” might commit to the first branch. “Bidder2”, also seeing that “Node” is prepared to join either branch, might then register to join the second branch’s Action. Say then that the “Node” Component chooses to participate with “Bidder2” on the second branch. The Action will run, and in this particular model, “Bidder2” now goes into another part of the model, interacting with other Components.

The “Node” then continues running before entering a derivative state where it *must* run again the Action it just ran with “Bidder2”, this time without a choice. The model designer anticipated that “Bidder1” would still be choosing between the two branches and could then cooperate with “Node”. However, in this implementation, the “Bidder1” has already committed to executing the other branch and is stuck there— it can’t change branches. At this point, the model’s execution ends in deadlock.

7.4.3 Fixing the *choice* method

The two problems above can be solved. Introducing true race conditions is one way but is undesirable for the reasons given previously.

To address the first problem, there should be a finer prioritizing of branches. At the moment, after pausing for some time, the fastest “ready” branch wins. Instead, the model should pause only if there are no *shared* Action branches ready to go. In this case, it could pause for a time to allow other Components the opportunity to join the Action before deciding. The value of “choice-pause” is also fixed to a value which gave good results for the models. However, perhaps the pause could be related to the length of the slowest branch, or be drawn as a random sample from an exponential distribution. None of these suggestions are ideal but they would improve the situation.

The second problem, being the more serious, can thankfully also be solved. Unfortunately, it requires extra steps and therefore introduces a higher overhead to running. The solution lies in a more sophisticated commital algorithm, similar to the “two-phase commit" protocol seen in distributed systems. Using this would avoid "Bidder1" committing without guarantee that "Node" is also going to commit.

This would necessitate an extra stage in the choice algorithm between the "ready" to run and the "committed" to run stages. Each Component would advertise the branch it had chosen by removing the "ready" to run flags from the other branches and setting a "chosen" flag on its choice. It would then keep monitoring the other choices until the full set of runners had "committed" or "chosen" the Action it had "chosen". Only at this point would it change from "chosen" to "commit".

If, before committing, whilst monitoring the other branches it noticed another branch was "ready" to run, it could change its choice. Critically though (to avoid any other Components getting stuck in a deadlocked choice) this Component would *only* be allowed to change its choice if no other members in its "chosen" branch had changed from "chosen" to "committed".

Being able to change would avoid the deadlock in "Big". It would not affect the running characteristics except for the extra processing stages: the Component did not "commit" to running the Action but only "chose" it. The difference is that the "chosen" Action would not be able to run until all members "committed". Finally, the requesting and releasing of locks using the current globally-ordered mechanism would ensure that switching choices at any point would not cause deadlock.

This solution seems quite complex and would certainly be overkill for most models which use only basic choosing. However, if the PEPA modelling system were to be used to design genuine concurrent or distributed systems, such a mechanism in PEPA2Java would be needed to avoid possible deadlock.

7.5 Suggestion for Extension

7.5.1 Full Stated-based Analysis

In addition to the *Simple Analysis* already included in the Simulator package, the addition of a method for calculating the mean residence times of the states as done in the Workbench would be very helpful, as it would allow statistical equivalence testing between the two. One way of doing this, for example, would be to parse the state-table structure the Workbench writes to disk and parse it to build an table of all the system states in the Simulator. Then, each time a Component performs an action that means a global state transition, such as executing an Activity, it could submit the details of the change, along with a global timestamp taken from the system clock, to a static queue-like mechanism.

Once the execution of the system has completed, the queue could be processed, beginning from the initial system state (which is known) and look-up each transition in the state-table. A similar lookup mechanism, the PEPA State Finder, already exists in the Workbench. The lookup would give the state index numbers and using this together with the timestamp would be enough to calculate the mean residence time of each state because the queue contains *all* the Component transitions.

7.5.2 Minor Improvements/ Extensions

Simple Analysis

If the user changes the speed whilst a model is running with the Simulator, the Simple Analysis at the end of the model run is invalid. Because the Simple Analysis tallies sleep-times, changing the multiplier used for calculating those sleep-times from the exponential distribibuted random samples will skew the results. There are two possible solutions:

1. reset the sleep-counters and the Action/ Activity run-counters each time the speed is changed, or

2. tally the sample rather than the sleep-times, and calculate the totals at the end using a single multiplier.

The second is the more elegant of the two solutions.

Alternative Distributions

The Java CUP grammar already specifies a syntax for creating Rates with either Uniform or Normal distribution. This part of the grammar is commented out for the PEPA workbench, however, presumably because these distributions lack the properties allowing memoryless analysis (where resumed and restarted activities may be treated in the same fashion).

However, should these distributions be required, modification of the PEPA2Java system would be simple. The package containing the *ExponentialDistribution* class [Siegrist and Duehring, 2001] also contains *Uniform-* and *NormalDistribution* classes that may be created and used in exactly the same way (because they share a common parent which defines the sample methods, etc.) Therefore, it would only be a matter of "plugging" them in to the Rate class and would require a minimum of further alteration.

Supplying Action Scripts

It should be possible to pass an object that implements *Runnable* to an *Action* object in the same way that can be done with *Thread* objects. This would define the *Action's* behaviour in the actual implementation, over-riding the default *sleep* behaviour when run.

Avoiding the Creation of extra Rate Objects

Currently the Translator equates the PEPA phrase: `(a,1.0)` with the PEPA2Java equivalent: `a_Activ.joinAt(new Rate(1.0).` This means a new object is created each time the Activity is called. To avoid this inefficiency, the Translator could define a Rate object (say, called `Rate_1_0`) for the Component calling this Activity, so the Rate could be re-used and resampled, rather than recreated each time.

Echo "stdin" and "sterr" to the P2J Dialog Window

Code to catch the messages and errors from running an external process is already included in the P2J Dialog Window class but at present does not work correctly. Being able to see the messages (especially error messages) during compilation and running of models from within the Workbench would be very useful.

7.6 Future Work

It would be an interesting follow-up exercise (as outlined in section 3.5) to use the tools produced to work through the design and implementation of an actual system, however simple, following the plan stipulated in figure 1.1.

Chapter 8

Conclusion

The principal goal of the project was to devise a method for the "Rapid Prototyping of High-Performance Concurrent Java Applications." This was accomplished by automating the creation of "skeleton" Java implementations from designs specified in the high-level modelling language, PEPA.

Automating the process means undesirable behaviour such as deadlock will not be introduced. Also, the performance characteristics of the model, tweaked to deliver the best results in the Workbench, will be maintained into implementation. In order to fulfill the goal of the project, two objectives were set.

The first objective was to produce and implement an API for creating and using PEPA-equivalent constructs in Java. The PEPA2Java API has been specified and is implemented by the Barebones and Simulator packages. These can run any PEPA model as a concurrent system.

The second objective was to build an application that could translate PEPA models into Java. The PEPA2Java Translator produces running "skeleton" implementations which use the commands specified in the PEPA2Java API to run as PEPA-equivalent systems.

The evaluation of the finished tools found that all models were able to be translated successfully into PEPA2Java-compatible Java packages. Compiling and running these systems with the Simulator package found that in all but two cases, the resulting systems performed as expected. The two special cases arose as a consequence of the necessity of replacing genuine race conditions with simulated race conditions. Suggestions made to extend and improve the Barebones and Simulator packages would, if put into effect, enable the implementations to run these final two models correctly. They will also provide the facility for statistical comparison of the PEPA2Java execution of a model to the steady-state solution found in the PEPA Workbench in order to see whether they are truly equivalent.

By automating the process of creating the initial Java implementation from a PEPA model, the result will maintain the performance and behavioural characteristics of the model, providing a sound framework for completing the concurrent application.

Not only do these tools ensure the system characterisics remain true to the high-level model, they remove uncertainty and the potential of introducing human error, whilst also greatly speeding the process of prototyping and implementation.

Appendix A

The Client-Proxy-Server PEPA Model

Note that all the working models used in this paper have been added to the *models* directory and may be recognized by their $p2j\_$ prefix.

```
%a = 1.0;
%b = 2.0;
%c = 3.0;

#Client = (cReq, a).(cRep, T).Client;

#Proxy = (cReq, T).Proxy';
#Proxy' = (cRep, b).Proxy + (pReq, a).(pRep, T).(cRep,b).Proxy;

#Server = (pReq, T).(pRep, c).Server;

Client <cReq, cRep> Proxy <pReq, pRep> Server
```

Appendix B

Prototype A Example: the Client-Proxy-Server System

```
/*********************************/
/********* Client.java ***********/
/*********************************/

package csp;

import prototypeA.*;


/**
  * <p>Title: Client-Proxy-Server Prototype 2</p>
  * <p>Description: </p>
  * <p>Copyright: Copyright (c) 2002</p>
  * <p>Company: Edinburgh University</p>
  * @author K. J .R. Powell
  * @version 1.0
  */
```

```
public class Client extends Component {
        Activity proxyRequest, proxyReply;

    public Client() {
        super("Client", true);
    }
    protected void init () {
        proxyRequest = Ref.getActivity("proxyRequest");
        proxyReply = Ref.getActivity("proxyReply");
        try {
                addMeTo(proxyReply, false);
                addMeTo(proxyRequest, false);
        } catch (AlreadyInitializedException aie) {
                aie.printErrMsg();
        }
    }

    public void loop() {
        call(proxyRequest);
        call(proxyReply);
    }
}

/********************************/
/********* Proxy.java ************/
/********************************/

package csp;

import prototypeA.*;
```

```
/**
 * <p>Title: Client-Proxy-Server Prototype 2</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Edinburgh University</p>
 * @author K. J .R. Powell
 * @version 1.0
 */

public class Proxy extends Component {
  Activity proxyRequest, proxyReply, serverRequest, serverReply;

  public Proxy() {
    super("Proxy", true);
  }
  protected void init () {
    proxyRequest = Ref.getActivity("proxyRequest");
    proxyReply = Ref.getActivity("proxyReply");
    serverRequest = Ref.getActivity("serverRequest");
    serverReply = Ref.getActivity("serverReply");
    try {
        addMeTo(proxyRequest, true);
        addMeTo(proxyReply, true);
        addMeTo(serverRequest, false);
        addMeTo(serverReply, false);
    } catch (AlreadyInitializedException aie) {
        aie.printErrMsg();
    }
  }
  public void loop() {
```

```
    switch (choice(proxyRequest, proxyRequest)) {
      case 1:
        call(proxyRequest);
        call(proxyReply);
        break;
      case 2:
        call(proxyRequest);
        call(serverRequest);
        call(serverReply);
        call(proxyReply);
        break;
    }
  }
}

/********************************/
/********* Server.java ***********/
/********************************/

package csp;

import prototypeA.*;


/**
 * <p>Title: Client-Proxy-Server Prototype 2</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Edinburgh University</p>
 * @author K. J .R. Powell
 * @version 1.0
```

```
 */

public class Server extends Component {
   Activity serverRequest, serverReply;

   public Server() {
      super("Server", true);
   }
   public void init(){
      serverRequest = Ref.getActivity("serverRequest");
      serverReply = Ref.getActivity("serverReply");
      try {
            addMeTo(serverRequest, true);
            addMeTo(serverReply, true);
      } catch (AlreadyInitializedException aie) {
            aie.printErrMsg();
      }
   }
   public void loop() {
      call(serverRequest);
      call(serverReply);
   }
}

/********************************/
/********** Sim.java ************/
/********************************/

package csp;

import prototypeA.*;
```

```
/**
 * <p>Title: Client-Proxy-Server Prototype 2</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: Edinburgh University</p>
 * @author K. J .R. Powell
 * @version 1.0
 */

public class Sim extends Simulator {
  public static final double rP = 1,
                             rS = 1;
// Server server;
// Proxy proxy;
// Client client;

  public Sim(int debugLevel) {
    super("Client-Proxy-Server System", debugLevel);
  }

  public void createComponents() {
    new Client();
    new Proxy();
    new Server();
  }

  public void createActivities() {
    new Activity("proxyRequest", Sim.rP);
    new Activity("proxyReply", Sim.rP);
```

```
        new Activity("serverRequest", Sim.rS);
        new Activity("serverReply", Sim.rS);
    }

    public static void main(String[] args) {
        Sim sim1 = new Sim(Simulator.DEBUG_ALL);
// Sim sim1 = new Sim(30, 5);
    }
}
```

Appendix C

Prototype B Example: the Client-Proxy-Server System

```
/************************************/
/********** Client_Comp.java ********/
/************************************/

package CPS;

import pepa.Pepa2Java.Simulator.*;

/** Skeleton generated by Pepa2Java Translator v.0.1
  * Built: 29 August 2002 12:00:18 CEST
  * Source: /home/kip/PEPA/jpwb2/models/CPS.pepa
  */

public class Client_Comp extends Component {
    Rate a_Rate = new Rate("a", 1.0);
    Rate TOP_Rate = new Rate();
    Activity cReq_Activ = new SharedActiv(Sim.cReq_Act, this);
    Activity cRep_Activ = new SharedActiv(Sim.cRep_Act, this);
```

```
    Client_Script client_Script = new Client_Script();
    public Client_Comp(String name) {
        super(name);
    }
    public class Client_Script implements CompScript {
        public CompScript actions () {
            cReq_Activ.joinAt(a_Rate);
            cRep_Activ.joinAt(TOP_Rate);
            return client_Script;
        }
    }
}


/***********************************/
/********** Proxy_Comp.java *********/
/***********************************/

package CPS;

import pepa.Pepa2Java.Simulator.*;

/** Skeleton generated by Pepa2Java Translator v.0.1
 * Built: 29 August 2002 12:00:18 CEST
 * Source: /home/kip/PEPA/jpwb2/models/CPS.pepa
 */

public class Proxy_Comp extends Component {
    Rate TOP_Rate = new Rate();
    Rate b_Rate = new Rate("b", 2.0);
    Rate a_Rate = new Rate("a", 1.0);
```

```
Activity cReq_Activ = new SharedActiv(Sim.cReq_Act, this);
Activity cRep_Activ = new SharedActiv(Sim.cRep_Act, this);
Activity pReq_Activ = new SharedActiv(Sim.pReq_Act, this);
Activity pRep_Activ = new SharedActiv(Sim.pRep_Act, this);
Proxy_Script proxy_Script = new Proxy_Script();
Proxy__Script proxy__Script = new Proxy__Script();
public Proxy_Comp(String name) {
    super(name);
}
public class Proxy_Script implements CompScript {
    public CompScript actions () {
        cReq_Activ.joinAt(TOP_Rate);
        return proxy__Script;
    }
}
public class Proxy__Script implements CompScript {

    Activity [] ch_Act_0 = {cRep_Activ, pReq_Activ};
    Rate [] ch_Rate_0 = {b_Rate, a_Rate};

    public CompScript actions () {
        switch(choice(ch_Act_0, ch_Rate_0)) {
        case 0:
            cRep_Activ.joinAt(b_Rate);
            return proxy_Script;
        case 1:
            pReq_Activ.joinAt(a_Rate);
            pRep_Activ.joinAt(TOP_Rate);
            cRep_Activ.joinAt(b_Rate);
            return proxy_Script;
        }
```

```
                throw new Error ("Problem with choice: no valid case returned!");
            }
        }
}


/***********************************/
/********** Server_Comp.java ********/
/***********************************/

package CPS;

import pepa.Pepa2Java.Simulator.*;

/** Skeleton generated by Pepa2Java Translator v.0.1
 * Built: 29 August 2002 12:00:18 CEST
 * Source: /home/kip/PEPA/jpwb2/models/CPS.pepa
 */

public class Server_Comp extends Component {
    Rate TOP_Rate = new Rate();
    Rate c_Rate = new Rate("c", 3.0);
    Activity pReq_Activ = new SharedActiv(Sim.pReq_Act, this);
    Activity pRep_Activ = new SharedActiv(Sim.pRep_Act, this);
    Server_Script server_Script = new Server_Script();
    public Server_Comp(String name) {
        super(name);
    }
    public class Server_Script implements CompScript {
        public CompScript actions () {
            pReq_Activ.joinAt(TOP_Rate);
```

```
            pRep_Activ.joinAt(c_Rate);
            return server_Script;
        }
    }
}


/************************************/
/************ Sim.java **************/
/************************************/

package CPS;

import pepa.Pepa2Java.Simulator.*;

/** Skeleton generated by Pepa2Java Translator v.0.1
 * Built: 29 August 2002 12:00:18 CEST
 * Source: /home/kip/PEPA/jpwb2/models/CPS.pepa
 */

public class Sim extends PepaSystem {

    /* Shared Actions */
    static Action cReq_Act;
    static Action cRep_Act;
    static Action pReq_Act;
    static Action pRep_Act;

    /* Components */
    static Client_Comp client_i0_Comp;
    static Proxy_Comp proxy_i0_Comp;
```

```
static Server_Comp server_i0_Comp;

public Sim() {
    super("GenSkel.CPS", 2, 1);
}

public void createActions() {
    cReq_Act = new Action("cReq");
    cRep_Act = new Action("cRep");
    pReq_Act = new Action("pReq");
    pRep_Act = new Action("pRep");
}

public void initSynchs() {
    cReq_Act.setSynch(SSNode.AND(Sim.client_i0_Comp.cReq_Activ,
                                 Sim.proxy_i0_Comp.cReq_Activ));
    cRep_Act.setSynch(SSNode.AND(Sim.client_i0_Comp.cRep_Activ,
                                 Sim.proxy_i0_Comp.cRep_Activ));
    pReq_Act.setSynch(SSNode.AND(Sim.proxy_i0_Comp.pReq_Activ,
                                 Sim.server_i0_Comp.pReq_Activ));
    pRep_Act.setSynch(SSNode.AND(Sim.proxy_i0_Comp.pRep_Activ,
                                 Sim.server_i0_Comp.pRep_Activ));
}

public void createComps() {
    client_i0_Comp = new Client_Comp("client_i0_Comp");
    client_i0_Comp.setStartScript(client_i0_Comp.client_Script);
    proxy_i0_Comp = new Proxy_Comp("proxy_i0_Comp");
    proxy_i0_Comp.setStartScript(proxy_i0_Comp.proxy_Script);
    server_i0_Comp = new Server_Comp("server_i0_Comp");
    server_i0_Comp.setStartScript(server_i0_Comp.server_Script);
```

```
    }

    public static void main(String[] args) {
        Sim sim = new Sim();
    }
}
```

Appendix D

Default Constants Defined in the Translator

```
API-implementing packages:
    Simulator package                    "pepa.Pepa2Java.Simulator.*"
    Barebones package                    "pepa.Pepa2Java.Bare.*"

For model output:
    PepaSystem-extending Class:          "Sim"
    Default output path                  "./models/p2jgens/"
    Write output to disk:                true
    Overwrite existing files:            false

Class and Object identifiers:
    Action suffix:                       "_Act"
    Activity suffix:                     "_Activ"
    Component suffix:                    "_Comp"
    CompScript suffix:                   "_Script"
    Rate suffix:                         "_Rate"
```

```
Choice Switch-block variables:
    Choice Activity array prefix:       "ch_Act_"
    Choice Rate array prefix:           "ch_Rate_"

For creating and updating Makefiles:
    Default "makefile" name             "Makefile"
    Java compiler command               "javac"
    Java runtime command                "java"
    Workbench $HOME                     "../../";
    (from model subdirectory)
    Classpath:
        "$HOME:$HOME/pepa/Pepa2Java/lib/distributions.jar:."
```

Appendix E

The Modified PC LAN 4 Model

```
%lambda=0.1;
%omega=3;
%mu=1;
%sigma=10;


#PR1  = (arrive, lambda).(request1, sigma).PR1;
#PC10 = (request1, infty).(serve1,infty).PC10 +
(walkon2,infty).PC10;

#PR2  = (arrive, lambda).(request2, sigma).PR2;
#PC20 = (request2, infty).(serve2,infty).PC20 +
(walkon3,infty).PC20;

#PR3  = (arrive, lambda).(request3, sigma).PR3;
#PC30 = (request3, infty).(serve3,infty).PC30 +
(walkon4,infty).PC30;

#PR4  = (arrive, lambda).(request4, sigma).PR4;
#PC40 = (request4, infty).(serve4,infty).PC40 +
(walkon1,infty).PC40;
```

```
#S1 = (walkon2,omega).S2 + (serve1,mu).(walk2,omega).S2;
#S2 = (walkon3,omega).S3 + (serve2,mu).(walk3,omega).S3;
#S3 = (walkon4,omega).S4 + (serve3,mu).(walk4,omega).S4;
#S4 = (walkon1,omega).S1 + (serve4,mu).(walk1,omega).S1;

((PC10 <request1> PR1) <>
 (PC20 <request2> PR2) <>
 (PC30 <request3> PR3) <>
 (PC40 <request4> PR4))
        <walkon1,walkon2,walkon3,walkon4,
                         serve1,serve2,serve3,serve4> S1
```

Appendix F

Model Evaluation Results

The following results are taken from translating, compiling and running the models in the `jpwb2/models` directory.

```
Badge.pepa:
******************************
********* SIM RESULTS ********
******************************
GenSkel.badge
Total sleeptime: 141.798 seconds
Actions:
reg14  sleepTime: 10.669 secs   (7.5240836 %)    Runs: 77(AVG 0.1385584 secs/run)
reg15  sleepTime: 17.235 secs   (12.1546143 %)   Runs: 96(AVG 0.1795312 secs/run)
reg16  sleepTime: 8.579 secs   (6.0501559 %)    Runs: 37(AVG 0.2318649 secs/run)
rep14  sleepTime: 6.709 secs   (4.7313784 %)    Runs: 77(AVG 0.0871299 secs/run)
rep15  sleepTime: 7.569 secs   (5.337875 %)    Runs: 96(AVG 0.0788438 secs/run)
rep16  sleepTime: 3.284 secs   (2.3159706 %)    Runs: 37(AVG 0.0887568 secs/run)
Indiv. Activities:
(p14_i0_Comp.move15,m)  sleepTime: 42.094 secs   (29.6858912 %)   Runs: 104(AVG 0.40475 secs/run)
(p14_i0_Comp.move14,m)  sleepTime: 24.135 secs   (17.0206914 %)   Runs: 61(AVG 0.3956557 secs/run)
(p14_i0_Comp.move16,m)  sleepTime: 21.524 secs   (15.1793396 %)   Runs: 43(AVG 0.5005581 secs/run)

Big.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.big
Total sleeptime: 199.154 seconds
Actions:
bid_in  sleepTime: 5.11 secs   (2.5658536 %)  Runs: 19(AVG 0.2689474 secs/run)
preq_in  sleepTime: 6.965 secs   (3.4972936 %)  Runs: 22(AVG 0.3165909 secs/run)
forward_accept  sleepTime: 19.142 secs   (9.6116573 %)  Runs: 13(AVG 1.4724615 secs/run)
```

```
forward_reject  sleepTime: 8.542 secs   (4.2891431 %)  Runs: 4(AVG 2.1355 secs/run)
forward_presp  sleepTime: 17.274 secs   (8.6736897 %)  Runs: 20(AVG 0.8637 secs/run)
bid'_in  sleepTime: 0.0 secs   (0.0 %)  Runs: 0(AVG NaN secs/run)
preq'_in  sleepTime: 0.214 secs   (0.1074545 %)  Runs: 1(AVG 0.214 secs/run)
forward_accept'  sleepTime: 0.0 secs   (0.0 %)  Runs: 0(AVG NaN secs/run)
forward_reject'  sleepTime: 0.0 secs   (0.0 %)  Runs: 0(AVG NaN secs/run)
forward_presp'  sleepTime: 0.0 secs   (0.0 %)  Runs: 0(AVG NaN secs/run)
accept_s  sleepTime: 14.305 secs   (7.1828836 %)  Runs: 8(AVG 1.788125 secs/run)
reject_s  sleepTime: 5.563 secs   (2.7933157 %)  Runs: 4(AVG 1.39075 secs/run)
presp_s  sleepTime: 22.077 secs   (11.0853912 %)  Runs: 16(AVG 1.3798125 secs/run)
forward_bid  sleepTime: 27.16 secs   (13.6376874 %)  Runs: 12(AVG 2.2633333 secs/run)
forward_preq  sleepTime: 31.837 secs   (15.9861213 %)  Runs: 16(AVG 1.9898125 secs/run)
forward_accept_sc  sleepTime: 0.0 secs   (0.0 %)  Runs: 0(AVG NaN secs/run)
forward_reject_sc  sleepTime: 0.0 secs   (0.0 %)  Runs: 0(AVG NaN secs/run)
forward_presp_sc  sleepTime: 0.0 secs   (0.0 %)  Runs: 0(AVG NaN secs/run)
forward_bid_nc  sleepTime: 0.0 secs   (0.0 %)  Runs: 0(AVG NaN secs/run)
forward_preq_nc  sleepTime: 0.0 secs   (0.0 %)  Runs: 0(AVG NaN secs/run)
accept_sc  sleepTime: 9.665 secs   (4.8530283 %)  Runs: 6(AVG 1.6108333 secs/run)
reject_sc  sleepTime: 8.198 secs   (4.1164124 %)  Runs: 1(AVG 8.198 secs/run)
presp_sc  sleepTime: 4.84 secs   (2.4302801 %)  Runs: 4(AVG 1.21 secs/run)
forward_bid_cs  sleepTime: 16.241 secs   (8.1549956 %)  Runs: 7(AVG 2.3201429 secs/run)
forward_preq_cs  sleepTime: 2.021 secs   (1.0147926 %)  Runs: 4(AVG 0.50525 secs/run)Indiv. Activities:

bug1.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.canon1
Total sleeptime: 235.49 seconds
Actions:
a  sleepTime: 92.18 secs   (39.1439127 %)  Runs: 45(AVG 2.0484444 secs/run)
Indiv. Activities:
(a_i0_Comp.b,r)  sleepTime: 46.83 secs   (19.8861947 %)  Runs: 45(AVG 1.0406667 secs/run)
(a_i0_Comp.c,s)  sleepTime: 25.261 secs   (10.7269948 %)  Runs: 44(AVG 0.5741136 secs/run)
(b_i0_Comp.f,m)  sleepTime: 35.625 secs   (15.1280309 %)  Runs: 22(AVG 1.6193182 secs/run)
(b_i0_Comp.g,m)  sleepTime: 35.594 secs   (15.1148669 %)  Runs: 23(AVG 1.5475652 secs/run)

bug2.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.bug2
Total sleeptime: 127.904 seconds
Actions:
a  sleepTime: 46.985 secs   (36.7345822 %)  Runs: 25(AVG 1.8794 secs/run)
Indiv. Activities:
(a_i0_Comp.b,r)  sleepTime: 25.381 secs   (19.8437891 %)  Runs: 25(AVG 1.01524 secs/run)
(a_i0_Comp.c,s)  sleepTime: 13.133 secs   (10.2678571 %)  Runs: 24(AVG 0.5472083 secs/run)
```

```
(b_i0_Comp.f,m)  sleepTime: 27.026 secs   (21.1299099 %)  Runs: 18(AVG 1.5014444 secs/run)
(b_i0_Comp.g,m)  sleepTime: 15.379 secs   (12.0238616 %)  Runs: 7(AVG 2.197 secs/run)

canon1.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.canon1
Total sleeptime: 151.717 seconds
Actions:
Indiv. Activities:
(a_i0_Comp.a,r)  sleepTime: 151.717 secs   (100.0 %)  Runs: 62(AVG 2.4470484 secs/run)

canon2.pep
******************************
********* SIM RESULTS ********
******************************
GenSkel.canon2
Total sleeptime: 504.129 seconds
Actions:
Indiv. Activities:
(a_i0_Comp.a,m)  sleepTime: 137.316 secs   (27.2382664 %)  Runs: 290(AVG 0.4735034 secs/run)
(a_i0_Comp.b,n)  sleepTime: 116.791 secs   (23.1668878 %)  Runs: 290(AVG 0.4027276 secs/run)
(b_i0_Comp.c,s)  sleepTime: 131.918 secs   (26.1675087 %)  Runs: 422(AVG 0.3126019 secs/run)
(b_i0_Comp.d,t)  sleepTime: 118.104 secs   (23.4273371 %)  Runs: 422(AVG 0.2798673 secs/run)

canon3.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.canon3
Total sleeptime: 389.188 seconds
Actions:
b  sleepTime: 92.057 secs   (23.653607 %)  Runs: 218(AVG 0.4222798 secs/run)
Indiv. Activities:
(a_i0_Comp.a,m)  sleepTime: 104.435 secs   (26.834075 %)  Runs: 219(AVG 0.4768721 secs/run)
(b_i0_Comp.c,s)  sleepTime: 106.461 secs   (27.3546461 %)  Runs: 306(AVG 0.3479118 secs/run)
(b_i0_Comp.d,t)  sleepTime: 86.235 secs   (22.1576719 %)  Runs: 306(AVG 0.2818137 secs/run)

canon4.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.canon4
Total sleeptime: 118.249 seconds
Actions:
a  sleepTime: 46.773 secs   (39.5546685 %)  Runs: 34(AVG 1.3756765 secs/run)
b  sleepTime: 37.541 secs   (31.7474144 %)  Runs: 17(AVG 2.2082941 secs/run)
```

```
c  sleepTime: 33.935 secs   (28.6979171 %)  Runs: 17(AVG 1.9961765 secs/run)
Indiv. Activities:

deadlock.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.deadlock
Total sleeptime: 1.771 seconds
Actions:
a  sleepTime: 0.4 secs   (22.5861095 %)  Runs: 1(AVG 0.4 secs/run)
b  sleepTime: 1.371 secs   (77.4138905 %)  Runs: 2(AVG 0.6855 secs/run)
Indiv. Activities:


*************************
*** java*.pepa -- skipped
*************************


maple-mod.pepa : K.P. added rates (all 1.0)as they were referred to
but undefined.
******************************
********* SIM RESULTS ********
******************************
GenSkel.maple_mod
Total sleeptime: 148.965 seconds
Actions:
a  sleepTime: 78.08 secs   (52.4149968 %)  Runs: 44(AVG 1.7745455 secs/run)
b  sleepTime: 26.949 secs   (18.0908267 %)  Runs: 22(AVG 1.2249545 secs/run)
c  sleepTime: 43.936 secs   (29.4941765 %)  Runs: 21(AVG 2.0921905 secs/run)
Indiv. Activities:

p2j_bigchoice.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.p2j_bigchoice
Total sleeptime: 166.673 seconds
Actions:
Indiv. Activities:
(a_i0_Comp.m,1.0)  sleepTime: 28.88 secs   (17.3273416 %)  Runs: 13(AVG 2.2215385 secs/run)
(a_i0_Comp.n,2.0)  sleepTime: 19.539 secs   (11.7229545 %)  Runs: 18(AVG 1.0855 secs/run)
(a_i0_Comp.o,3.0)  sleepTime: 14.671 secs   (8.8022655 %)  Runs: 23(AVG 0.6378696 secs/run)
(a_i0_Comp.x,0.5)  sleepTime: 74.183 secs   (44.5081087 %)  Runs: 13(AVG 5.7063846 secs/run)
(a_i0_Comp.y,0.5)  sleepTime: 29.4 secs   (17.6393297 %)  Runs: 10(AVG 2.94 secs/run)

p2j_choiceyielding.pepa -- two sets of results
```

```
first with choicetime = 100 and speed = normal.
******************************
********* SIM RESULTS ********
******************************
GenSkel.p2j_choiceyielding
Total sleeptime: 641.919 seconds1414
Actions:
m  sleepTime: 20.155 secs   (3.1398042 %)  Runs: 14(AVG 1.4396429 secs/run)
n  sleepTime: 302.401 secs   (47.1089031 %)  Runs: 148(AVG 2.04325 secs/run)
Indiv. Activities:
(b_i0_Comp.i,1.0)  sleepTime: 319.363 secs   (49.7512926 %)  Runs: 166(AVG 1.9238735 secs/run)

p2j_choiceyielding.pepa -- two sets of results
second with choicetime = 500 and speed = fastest.
******************************
********* SIM RESULTS ********
******************************
GenSkel.p2j_choiceyielding
Total sleeptime: 339.335 seconds
Actions:
m  sleepTime: 79.142 secs   (23.3226752 %)  Runs: 327(AVG 0.2420245 secs/run)
n  sleepTime: 114.75 secs   (33.8161404 %)  Runs: 498(AVG 0.2304217 secs/run)
Indiv. Activities:
(b_i0_Comp.i,1.0)  sleepTime: 145.443 secs   (42.8611844 %)  Runs: 465(AVG 0.3127806 secs/run)


p2j_PC-LAN4-mod.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.p2j_PC_LAN4_mod
Total sleeptime: 944.303 seconds
Actions:
request1  sleepTime: 0.717 secs   (0.075929 %)  Runs: 8(AVG 0.089625 secs/run)
serve1  sleepTime: 15.272 secs   (1.6172775 %)  Runs: 8(AVG 1.909 secs/run)
walkon2  sleepTime: 20.564 secs   (2.1776908 %)  Runs: 33(AVG 0.6231515 secs/run)
request2  sleepTime: 0.355 secs   (0.0375939 %)  Runs: 3(AVG 0.1183333 secs/run)
serve2  sleepTime: 2.374 secs   (0.2514024 %)  Runs: 3(AVG 0.7913333 secs/run)
walkon3  sleepTime: 24.473 secs   (2.591647 %)  Runs: 38(AVG 0.6440263 secs/run)
request3  sleepTime: 3.022 secs   (0.3200244 %)  Runs: 10(AVG 0.3022 secs/run)
serve3  sleepTime: 13.605 secs   (1.4407452 %)  Runs: 10(AVG 1.3605 secs/run)
walkon4  sleepTime: 20.915 secs   (2.2148611 %)  Runs: 30(AVG 0.6971667 secs/run)
request4  sleepTime: 1.943 secs   (0.2057602 %)  Runs: 12(AVG 0.1619167 secs/run)
serve4  sleepTime: 38.393 secs   (4.0657501 %)  Runs: 12(AVG 3.1994167 secs/run)
walkon1  sleepTime: 20.094 secs   (2.1279187 %)  Runs: 28(AVG 0.7176429 secs/run)
Indiv. Activities:
(pR1_i0_Comp.arrive,lambda)  sleepTime: 199.215 secs   (21.0965125 %)  Runs: 9(AVG 22.135 secs/run)
(pR2_i0_Comp.arrive,lambda)  sleepTime: 200.32 secs   (21.21353 %)  Runs: 4(AVG 50.08 secs/run)
```

```
(pR3_i0_Comp.arrive,lambda)  sleepTime: 187.686 secs   (19.875612 %)  Runs: 11(AVG 17.0623636 secs/run)
(pR4_i0_Comp.arrive,lambda)  sleepTime: 177.873 secs   (18.8364328 %)  Runs: 13(AVG 13.6825385 secs/run)
(s1_i0_Comp.walk2,omega)  sleepTime: 6.881 secs   (0.7286856 %)  Runs: 8(AVG 0.860125 secs/run)
(s1_i0_Comp.walk3,omega)  sleepTime: 1.653 secs   (0.1750497 %)  Runs: 3(AVG 0.551 secs/run)
(s1_i0_Comp.walk4,omega)  sleepTime: 4.798 secs   (0.5080996 %)  Runs: 10(AVG 0.4798 secs/run)
(s1_i0_Comp.walk1,omega)  sleepTime: 4.15 secs   (0.4394776 %)  Runs: 12(AVG 0.3458333 secs/run)

p2j_Server_3Client.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.p2j_Server_3Client
Total sleeptime: 1517.454 seconds
Actions:
req  sleepTime: 692.002 secs   (45.6028321 %)  Runs: 829(AVG 0.8347431 secs/run)
rep  sleepTime: 825.452 secs   (54.3971679 %)  Runs: 828(AVG 0.9969227 secs/run)
Indiv. Activities:

papmmodel.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.papmmodel
Total sleeptime: 2976.565 seconds
Actions:
reqlock11  sleepTime: 11.459 secs   (0.384974 %)  Runs: 45(AVG 0.2546444 secs/run)
writelock11  sleepTime: 9.242 secs   (0.3104921 %)  Runs: 44(AVG 0.2100455 secs/run)
reqlock12  sleepTime: 4.381 secs   (0.1471831 %)  Runs: 19(AVG 0.2305789 secs/run)
writelock12  sleepTime: 3.549 secs   (0.1192314 %)  Runs: 19(AVG 0.1867895 secs/run)
reqlock13  sleepTime: 3.252 secs   (0.1092535 %)  Runs: 16(AVG 0.20325 secs/run)
writelock13  sleepTime: 5.471 secs   (0.1838025 %)  Runs: 16(AVG 0.3419375 secs/run)
reqlock21  sleepTime: 11.011 secs   (0.369923 %)  Runs: 39(AVG 0.2823333 secs/run)
writelock21  sleepTime: 6.476 secs   (0.2175662 %)  Runs: 39(AVG 0.1660513 secs/run)
reqlock22  sleepTime: 8.239 secs   (0.2767956 %)  Runs: 37(AVG 0.2226757 secs/run)
writelock22  sleepTime: 12.037 secs   (0.4043923 %)  Runs: 37(AVG 0.3253243 secs/run)
reqlock23  sleepTime: 5.444 secs   (0.1828954 %)  Runs: 19(AVG 0.2865263 secs/run)
writelock23  sleepTime: 4.731 secs   (0.1589416 %)  Runs: 19(AVG 0.249 secs/run)
reqlock31  sleepTime: 2.056 secs   (0.0690729 %)  Runs: 8(AVG 0.257 secs/run)
writelock31  sleepTime: 2.49 secs   (0.0836535 %)  Runs: 8(AVG 0.31125 secs/run)
reqlock32  sleepTime: 2.873 secs   (0.0965207 %)  Runs: 9(AVG 0.3192222 secs/run)
writelock32  sleepTime: 2.401 secs   (0.0806634 %)  Runs: 9(AVG 0.2667778 secs/run)
reqlock33  sleepTime: 1.625 secs   (0.0545931 %)  Runs: 9(AVG 0.1805556 secs/run)
writelock33  sleepTime: 2.145 secs   (0.0720629 %)  Runs: 9(AVG 0.2383333 secs/run)
Indiv. Activities:
(txnc1_i0_Comp.think1,t1)  sleepTime: 822.015 secs   (27.6162288 %)  Runs: 72(AVG 11.416875 secs/run)
(txnc1_i0_Comp.fork,w1xr11)  sleepTime: 72.565 secs   (2.4378772 %)  Runs: 151(AVG 0.4805629 secs/run)
(txnc1_i0_Comp.finish,a11xonemq)  sleepTime: 27.5 secs   (0.9238837 %)  Runs: 150(AVG 0.1833333 secs/run)
(txnc1_i0_Comp.reqlock1r,rl1r)  sleepTime: 18.025 secs   (0.6055638 %)  Runs: 71(AVG 0.2538732 secs/run)
```

```
(txnc1_i0_Comp.readlock,r1)  sleepTime: 18.709 secs   (0.6285433 %)  Runs: 71(AVG 0.263507 secs/run)
(txnc2_i0_Comp.think2,t2)  sleepTime: 469.267 secs   (15.7653873 %)  Runs: 118(AVG 3.976839 secs/run)
(txnc2_i0_Comp.fork,w2xrr2)  sleepTime: 196.637 secs   (6.6061719 %)  Runs: 380(AVG 0.5174658 secs/run)
(txnc2_i0_Comp.finish,a2rxq)  sleepTime: 70.859 secs   (2.3805628 %)  Runs: 379(AVG 0.1869631 secs/run)
(txnc2_i0_Comp.reqlock2r,rl2r)  sleepTime: 69.625 secs   (2.3391056 %)  Runs: 285(AVG 0.2442982 secs/run)
(txnc2_i0_Comp.readlock,r2)  sleepTime: 73.355 secs   (2.4644179 %)  Runs: 285(AVG 0.257386 secs/run)
(txnc3_i0_Comp.think3,t3)  sleepTime: 887.557 secs   (29.8181629 %)  Runs: 21(AVG 42.264619 secs/run)
(txnc3_i0_Comp.fork,w3xr31)  sleepTime: 94.145 secs   (3.162874 %)  Runs: 104(AVG 0.9052404 secs/run)
(txnc3_i0_Comp.finish,a31xonemq)  sleepTime: 20.961 secs   (0.704201 %)  Runs: 104(AVG 0.2015481 secs/run)
(txnc3_i0_Comp.reqlock3r,rl3r)  sleepTime: 17.848 secs   (0.5996173 %)  Runs: 78(AVG 0.2288205 secs/run)
(txnc3_i0_Comp.readlock,r3)  sleepTime: 18.615 secs   (0.6253853 %)  Runs: 78(AVG 0.2386538 secs/run)

papmmodelsmall.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.papmmodelsmall
Total sleeptime: 2400.003 seconds
Actions:
reqlock11  sleepTime: 29.18 secs   (1.2158318 %)  Runs: 112(AVG 0.2605357 secs/run)
writelock11  sleepTime: 29.209 secs   (1.2170401 %)  Runs: 112(AVG 0.2607946 secs/run)
reqlock21  sleepTime: 53.016 secs   (2.2089972 %)  Runs: 217(AVG 0.2443134 secs/run)
writelock21  sleepTime: 51.7 secs   (2.154164 %)  Runs: 217(AVG 0.2382488 secs/run)
reqlock31  sleepTime: 14.884 secs   (0.6201659 %)  Runs: 63(AVG 0.236254 secs/run)
writelock31  sleepTime: 16.613 secs   (0.6922075 %)  Runs: 63(AVG 0.2636984 secs/run)
Indiv. Activities:
(txnc1_i0_Comp.think1,t1)  sleepTime: 676.594 secs   (28.1913814 %)  Runs: 65(AVG 10.4091385 secs/run)
(txnc1_i0_Comp.fork,w1xr11)  sleepTime: 48.949 secs   (2.0395391 %)  Runs: 112(AVG 0.4370446 secs/run)
(txnc1_i0_Comp.finish,a11xonemq)  sleepTime: 22.601 secs   (0.9417072 %)  Runs: 112(AVG 0.2017946 secs/run)
(txnc2_i0_Comp.think2,t2)  sleepTime: 272.293 secs   (11.3455275 %)  Runs: 80(AVG 3.4036625 secs/run)
(txnc2_i0_Comp.fork,w2xr21)  sleepTime: 347.694 secs   (14.4872319 %)  Runs: 217(AVG 1.6022765 secs/run)
(txnc2_i0_Comp.finish,a21xonemq)  sleepTime: 44.083 secs   (1.8367894 %)  Runs: 217(AVG 0.2031475 secs/run)
(txnc3_i0_Comp.think3,t3)  sleepTime: 580.623 secs   (24.1925948 %)  Runs: 9(AVG 64.5136667 secs/run)
(txnc3_i0_Comp.fork,w3xr31)  sleepTime: 199.65 secs   (8.3187396 %)  Runs: 63(AVG 3.1690476 secs/run)
(txnc3_i0_Comp.finish,a31xonemq)  sleepTime: 12.914 secs   (0.5380827 %)  Runs: 63(AVG 0.2049841 secs/run)

papmmodel-smaller-exp.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.papmmodel_smaller_exp
Total sleeptime: 1675.943 seconds
Actions:
reqlock11  sleepTime: 11.882 secs   (0.708974 %)  Runs: 53(AVG 0.2241887 secs/run)
writelock11  sleepTime: 13.319 secs   (0.7947168 %)  Runs: 53(AVG 0.2513019 secs/run)
reqlock12  sleepTime: 4.657 secs   (0.2778734 %)  Runs: 23(AVG 0.2024783 secs/run)
writelock12  sleepTime: 8.524 secs   (0.5086092 %)  Runs: 23(AVG 0.3706087 secs/run)
reqlock21  sleepTime: 8.211 secs   (0.4899331 %)  Runs: 30(AVG 0.2737 secs/run)
```

```
writelock21  sleepTime: 6.898 secs   (0.4115892 %)  Runs: 30(AVG 0.2299333 secs/run)
reqlock22  sleepTime: 6.437 secs   (0.3840823 %)  Runs: 29(AVG 0.2219655 secs/run)
writelock22  sleepTime: 8.242 secs   (0.4917828 %)  Runs: 29(AVG 0.2842069 secs/run)
Indiv. Activities:
(txnc1_i0_Comp.think1,t1)  sleepTime: 671.494 secs   (40.0666371 %)  Runs: 69(AVG 9.7317971 secs/run)
(txnc1_i0_Comp.fork,w1xrr1)  sleepTime: 74.486 secs   (4.4444232 %)  Runs: 168(AVG 0.443369 secs/run)
(txnc1_i0_Comp.finish,a1rxonemq)  sleepTime: 31.2 secs   (1.8616385 %)  Runs: 168(AVG 0.1857143 secs/run)
(txnc1_i0_Comp.reqlock1r,rl1r)  sleepTime: 22.553 secs   (1.3456902 %)  Runs: 92(AVG 0.2451413 secs/run)
(txnc1_i0_Comp.readlock,r1)  sleepTime: 20.308 secs   (1.2117357 %)  Runs: 92(AVG 0.2207391 secs/run)
(txnc2_i0_Comp.think2,t2)  sleepTime: 445.238 secs   (26.5664166 %)  Runs: 101(AVG 4.408297 secs/run)
(txnc2_i0_Comp.fork,w2xrr2)  sleepTime: 167.083 secs   (9.9694918 %)  Runs: 312(AVG 0.5355224 secs/run)
(txnc2_i0_Comp.finish,a2rxonemq)  sleepTime: 59.238 secs   (3.5346071 %)  Runs: 312(AVG 0.1898654 secs/run)
(txnc2_i0_Comp.reqlock2r,rl2r)  sleepTime: 54.705 secs   (3.2641325 %)  Runs: 253(AVG 0.2162253 secs/run)
(txnc2_i0_Comp.readlock,r2)  sleepTime: 61.468 secs   (3.6676665 %)  Runs: 253(AVG 0.2429565 secs/run)

tomp11.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.tomp11
Total sleeptime: 1110.276 seconds
Actions:
get1  sleepTime: 56.881 secs   (5.1231406 %)  Runs: 217(AVG 0.2621244 secs/run)
use  sleepTime: 557.138 secs   (50.1801354 %)  Runs: 435(AVG 1.280777 secs/run)
rel  sleepTime: 126.014 secs   (11.3497905 %)  Runs: 434(AVG 0.2903548 secs/run)
get2  sleepTime: 59.103 secs   (5.323271 %)  Runs: 218(AVG 0.2711147 secs/run)
Indiv. Activities:
(p1_i0_Comp.think,lambda1)  sleepTime: 93.792 secs   (8.4476292 %)  Runs: 218(AVG 0.4302385 secs/run)
(p2_i0_Comp.think,lambda2)  sleepTime: 217.348 secs   (19.5760333 %)  Runs: 218(AVG 0.9970092 secs/run)

tomp111.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.tomp111
Total sleeptime: 1597.932 seconds
Actions:
get1  sleepTime: 54.598 secs   (3.4167912 %)  Runs: 202(AVG 0.2702871 secs/run)
use  sleepTime: 316.467 secs   (19.8047852 %)  Runs: 786(AVG 0.4026298 secs/run)
rel  sleepTime: 227.13 secs   (14.2139966 %)  Runs: 786(AVG 0.2889695 secs/run)
get2  sleepTime: 68.465 secs   (4.2846003 %)  Runs: 284(AVG 0.2410739 secs/run)
get3  sleepTime: 74.908 secs   (4.687809 %)  Runs: 300(AVG 0.2496933 secs/run)
Indiv. Activities:
(p1_i0_Comp.think,lambda1)  sleepTime: 393.418 secs   (24.6204469 %)  Runs: 203(AVG 1.9380197 secs/run)
(p2_i0_Comp.think,lambda2)  sleepTime: 255.933 secs   (16.0165138 %)  Runs: 285(AVG 0.8980105 secs/run)
(p3_i0_Comp.think,lambda3)  sleepTime: 207.013 secs   (12.9550569 %)  Runs: 301(AVG 0.6877508 secs/run)

tomp12.pepa
```

```
******************************
********* SIM RESULTS ********
******************************
GenSkel.tomp12
Total sleeptime: 1080.811 seconds
Actions:
get1  sleepTime: 55.259 secs   (5.1127348 %)  Runs: 218(AVG 0.2534817 secs/run)
use  sleepTime: 551.965 secs   (51.0695209 %)  Runs: 437(AVG 1.2630778 secs/run)
rel  sleepTime: 122.218 secs   (11.30799 %)  Runs: 436(AVG 0.2803165 secs/run)
get2  sleepTime: 52.528 secs   (4.8600542 %)  Runs: 219(AVG 0.2398539 secs/run)
Indiv. Activities:
(p1_i0_Comp.think,lambda1)  sleepTime: 2.902 secs   (0.2685021 %)  Runs: 9(AVG 0.3224444 secs/run)
(p1_i1_Comp.think,lambda1)  sleepTime: 74.876 secs   (6.9277607 %)  Runs: 211(AVG 0.3548626 secs/run)
(p2_i0_Comp.think,lambda2)  sleepTime: 221.063 secs   (20.4534373 %)  Runs: 219(AVG 1.0094201 secs/run)

tomp221.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.tomp221
Total sleeptime: 2954.024 seconds
Actions:
get1  sleepTime: 92.134 secs   (3.118932 %)  Runs: 386(AVG 0.2386891 secs/run)
use  sleepTime: 566.574 secs   (19.1797358 %)  Runs: 1409(AVG 0.4021107 secs/run)
rel  sleepTime: 396.82 secs   (13.4332016 %)  Runs: 1408(AVG 0.2818324 secs/run)
get2  sleepTime: 119.724 secs   (4.0529122 %)  Runs: 511(AVG 0.2342935 secs/run)
get3  sleepTime: 129.078 secs   (4.369565 %)  Runs: 512(AVG 0.2521055 secs/run)
Indiv. Activities:
(p1_i0_Comp.think,lambda1)  sleepTime: 271.142 secs   (9.1787338 %)  Runs: 123(AVG 2.2044065 secs/run)
(p1_i1_Comp.think,lambda1)  sleepTime: 511.888 secs   (17.3284983 %)  Runs: 265(AVG 1.9316528 secs/run)
(p2_i0_Comp.think,lambda2)  sleepTime: 114.281 secs   (3.8686551 %)  Runs: 129(AVG 0.8858992 secs/run)
(p2_i1_Comp.think,lambda2)  sleepTime: 411.184 secs   (13.9194536 %)  Runs: 384(AVG 1.0707917 secs/run)
(p3_i0_Comp.think,lambda3)  sleepTime: 341.199 secs   (11.5503124 %)  Runs: 512(AVG 0.6664043 secs/run)


tomp333.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.tomp333
Total sleeptime: 1846.95 seconds
Actions:
get1  sleepTime: 68.134 secs   (3.6890008 %)  Runs: 324(AVG 0.2102901 secs/run)
use  sleepTime: 355.151 secs   (19.2290533 %)  Runs: 886(AVG 0.4008476 secs/run)
rel  sleepTime: 248.436 secs   (13.4511492 %)  Runs: 885(AVG 0.2807186 secs/run)
get2  sleepTime: 27.177 secs   (1.4714529 %)  Runs: 121(AVG 0.2246033 secs/run)
get3  sleepTime: 116.976 secs   (6.3334687 %)  Runs: 441(AVG 0.2652517 secs/run)
Indiv. Activities:
```

```
(p1_i0_Comp.think,lambda1)  sleepTime: 54.973 secs   (2.9764206 %)  Runs: 34(AVG 1.6168529 secs/run)
(p1_i1_Comp.think,lambda1)  sleepTime: 216.597 secs   (11.7272801 %)  Runs: 120(AVG 1.804975 secs/run)
(p1_i2_Comp.think,lambda1)  sleepTime: 362.521 secs   (19.6280896 %)  Runs: 172(AVG 2.1076802 secs/run)
(p2_i0_Comp.think,lambda2)  sleepTime: 1.196 secs   (0.0647554 %)  Runs: 2(AVG 0.598 secs/run)
(p2_i1_Comp.think,lambda2)  sleepTime: 28.217 secs   (1.527762 %)  Runs: 29(AVG 0.973 secs/run)
(p2_i2_Comp.think,lambda2)  sleepTime: 81.416 secs   (4.4081323 %)  Runs: 93(AVG 0.8754409 secs/run)
(p3_i0_Comp.think,lambda3)  sleepTime: 2.21 secs   (0.1196567 %)  Runs: 5(AVG 0.442 secs/run)
(p3_i1_Comp.think,lambda3)  sleepTime: 44.028 secs   (2.383822 %)  Runs: 71(AVG 0.6201127 secs/run)
(p3_i2_Comp.think,lambda3)  sleepTime: 239.918 secs   (12.9899564 %)  Runs: 368(AVG 0.6519511 secs/run)

tomp433.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.tomp433
Total sleeptime: 1892.025 seconds
Actions:
get1  sleepTime: 69.039 secs   (3.6489476 %)  Runs: 302(AVG 0.228606 secs/run)
use  sleepTime: 346.353 secs   (18.305942 %)  Runs: 888(AVG 0.3900372 secs/run)
rel  sleepTime: 247.468 secs   (13.0795312 %)  Runs: 888(AVG 0.2786802 secs/run)
get2  sleepTime: 53.165 secs   (2.8099523 %)  Runs: 221(AVG 0.2405656 secs/run)
get3  sleepTime: 88.314 secs   (4.6676973 %)  Runs: 366(AVG 0.2412951 secs/run)
Indiv. Activities:
(p1_i0_Comp.think,lambda1)  sleepTime: 17.787 secs   (0.9401039 %)  Runs: 10(AVG 1.7787 secs/run)
(p1_i1_Comp.think,lambda1)  sleepTime: 83.844 secs   (4.4314425 %)  Runs: 39(AVG 2.1498462 secs/run)
(p1_i2_Comp.think,lambda1)  sleepTime: 187.636 secs   (9.9172051 %)  Runs: 100(AVG 1.87636 secs/run)
(p1_i3_Comp.think,lambda1)  sleepTime: 338.937 secs   (17.9139811 %)  Runs: 157(AVG 2.1588344 secs/run)
(p2_i0_Comp.think,lambda2)  sleepTime: 8.988 secs   (0.4750466 %)  Runs: 5(AVG 1.7976 secs/run)
(p2_i1_Comp.think,lambda2)  sleepTime: 54.511 secs   (2.881093 %)  Runs: 55(AVG 0.9911091 secs/run)
(p2_i2_Comp.think,lambda2)  sleepTime: 154.33 secs   (8.156869 %)  Runs: 163(AVG 0.9468098 secs/run)
(p3_i0_Comp.think,lambda3)  sleepTime: 9.232 secs   (0.4879428 %)  Runs: 5(AVG 1.8464 secs/run)
(p3_i1_Comp.think,lambda3)  sleepTime: 35.921 secs   (1.8985479 %)  Runs: 63(AVG 0.5701746 secs/run)
(p3_i2_Comp.think,lambda3)  sleepTime: 196.5 secs   (10.3856979 %)  Runs: 301(AVG 0.6528239 secs/run)

tomp444.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.tomp444
Total sleeptime: 347.86 seconds
Actions:
get1  sleepTime: 2.964 secs   (0.8520669 %)  Runs: 343(AVG 0.0086414 secs/run)
use  sleepTime: 66.025 secs   (18.9803369 %)  Runs: 1006(AVG 0.0656312 secs/run)
rel  sleepTime: 52.026 secs   (14.9560168 %)  Runs: 1006(AVG 0.0517157 secs/run)
get2  sleepTime: 2.982 secs   (0.8572414 %)  Runs: 335(AVG 0.0089015 secs/run)
get3  sleepTime: 2.739 secs   (0.7873857 %)  Runs: 328(AVG 0.0083506 secs/run)
Indiv. Activities:
(p1_i0_Comp.think,lambda1)  sleepTime: 26.965 secs   (7.7516817 %)  Runs: 74(AVG 0.3643919 secs/run)
```

```
(p1_i1_Comp.think,lambda1)  sleepTime: 37.571 secs   (10.8006094 %)  Runs: 101(AVG 0.3719901 secs/run)
(p1_i2_Comp.think,lambda1)  sleepTime: 31.7 secs   (9.1128615 %)  Runs: 90(AVG 0.3522222 secs/run)
(p1_i3_Comp.think,lambda1)  sleepTime: 29.097 secs   (8.364572 %)  Runs: 82(AVG 0.3548415 secs/run)
(p2_i0_Comp.think,lambda2)  sleepTime: 16.341 secs   (4.6975795 %)  Runs: 85(AVG 0.1922471 secs/run)
(p2_i1_Comp.think,lambda2)  sleepTime: 13.688 secs   (3.9349163 %)  Runs: 86(AVG 0.1591628 secs/run)
(p2_i2_Comp.think,lambda2)  sleepTime: 13.659 secs   (3.9265797 %)  Runs: 80(AVG 0.1707375 secs/run)
(p2_i3_Comp.think,lambda2)  sleepTime: 15.573 secs   (4.476801 %)  Runs: 88(AVG 0.1769659 secs/run)
(p3_i0_Comp.think,lambda3)  sleepTime: 8.847 secs   (2.5432645 %)  Runs: 86(AVG 0.1028721 secs/run)
(p3_i1_Comp.think,lambda3)  sleepTime: 8.832 secs   (2.5389525 %)  Runs: 84(AVG 0.1051429 secs/run)
(p3_i2_Comp.think,lambda3)  sleepTime: 10.113 secs   (2.907204 %)  Runs: 83(AVG 0.1218434 secs/run)
(p3_i3_Comp.think,lambda3)  sleepTime: 8.738 secs   (2.5119301 %)  Runs: 79(AVG 0.1106076 secs/run)


workcell2.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.workcell2
Total sleeptime: 6740.892 seconds
Actions:
ready_to_pick  sleepTime: 90.013 secs   (1.3353277 %)  Runs: 306(AVG 0.2941601 secs/run)
unload_blank  sleepTime: 93.325 secs   (1.3844607 %)  Runs: 306(AVG 0.3049837 secs/run)
DBelt_ready  sleepTime: 91.985 secs   (1.364582 %)  Runs: 306(AVG 0.3006046 secs/run)
load_blank  sleepTime: 93.565 secs   (1.3880211 %)  Runs: 304(AVG 0.3077796 secs/run)
ready_to_put  sleepTime: 94.057 secs   (1.3953198 %)  Runs: 308(AVG 0.3053799 secs/run)
blank_ready  sleepTime: 91.491 secs   (1.3572536 %)  Runs: 304(AVG 0.3009572 secs/run)
Indiv. Activities:
(robot_i0_Comp.pick_posn_arm_1,A1_T1)  sleepTime: 333.815 secs   (4.9520894 %)  Runs: 153(AVG 2.1817974 secs/run)
(robot_i0_Comp.safe_posn_arm_1,A1_T2)  sleepTime: 288.383 secs   (4.2781133 %)  Runs: 305(AVG 0.945518 secs/run)
(robot_i0_Comp.at_press_arm_2,R_2)  sleepTime: 71.418 secs   (1.059474 %)  Runs: 153(AVG 0.4667843 secs/run)
(robot_i0_Comp.pick_posn_arm_2,A2_T1)  sleepTime: 331.113 secs   (4.9120057 %)  Runs: 153(AVG 2.1641373 secs/run)
(robot_i0_Comp.at_belt_arm_2,R_3)  sleepTime: 50.024 secs   (0.7420976 %)  Runs: 153(AVG 0.3269542 secs/run)
(robot_i0_Comp.put_posn_arm_2,A2_T2)  sleepTime: 157.448 secs   (2.3357146 %)  Runs: 153(AVG 1.0290719 secs/run)
(robot_i0_Comp.safe_posn_arm_2,A2_T3)  sleepTime: 89.599 secs   (1.3291861 %)  Runs: 153(AVG 0.5856144 secs/run)
(robot_i0_Comp.at_press_arm_1,R_4)  sleepTime: 50.143 secs   (0.743863 %)  Runs: 152(AVG 0.3298882 secs/run)
(robot_i0_Comp.put_posn_arm_1,A1_T2)  sleepTime: 149.609 secs   (2.2194244 %)  Runs: 152(AVG 0.9842697 secs/run)
(robot_i0_Comp.at_table_arm_1,R_1)  sleepTime: 100.453 secs   (1.4902034 %)  Runs: 152(AVG 0.660875 secs/run)
(belt_i0_Comp.B_sensor_On,F_T)  sleepTime: 327.529 secs   (4.8588377 %)  Runs: 155(AVG 2.1130903 secs/run)
(belt_i0_Comp.B_sensor_Off,F_B)  sleepTime: 328.627 secs   (4.8751263 %)  Runs: 154(AVG 2.1339416 secs/run)
(table_i0_Comp.move_table,T_T)  sleepTime: 620.198 secs   (9.2005331 %)  Runs: 308(AVG 2.0136299 secs/run)
(press_i0_Comp.move_press,P_b2m)  sleepTime: 587.978 secs   (8.7225548 %)  Runs: 305(AVG 1.9277967 secs/run)
(dBelt_1_i0_Comp.delay,D_B)  sleepTime: 261.216 secs   (3.8750955 %)  Runs: 153(AVG 1.7072941 secs/run)
(dBelt_1_i0_Comp.D_sensor_On,D_T)  sleepTime: 355.034 secs   (5.2668697 %)  Runs: 153(AVG 2.3204837 secs/run)
(dBelt_1_i0_Comp.D_sensor_Off,D_B)  sleepTime: 347.618 secs   (5.1568546 %)  Runs: 152(AVG 2.2869605 secs/run)
(crane_i0_Comp.mag_u,C_V)  sleepTime: 603.12 secs   (8.9471838 %)  Runs: 305(AVG 1.9774426 secs/run)
(crane_i0_Comp.over_belt2,C_H)  sleepTime: 305.073 secs   (4.5257067 %)  Runs: 153(AVG 1.9939412 secs/run)
(crane_i0_Comp.mag_d,C_V)  sleepTime: 537.962 secs   (7.9805759 %)  Runs: 304(AVG 1.7696118 secs/run)
(crane_i0_Comp.over_belt1,C_H)  sleepTime: 246.025 secs   (3.6497395 %)  Runs: 152(AVG 1.6185855 secs/run)
```

```
(crane_i0_Comp.can_accept,xi)  sleepTime: 44.071 secs   (0.6537859 %)  Runs: 152(AVG 0.2899408 secs/run)


test.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.test
Total sleeptime: 652.918 seconds
Actions:
a  sleepTime: 261.614 secs   (40.0684313 %)  Runs: 114(AVG 2.2948596 secs/run)
Indiv. Activities:
(a_i0_Comp.b,a2)  sleepTime: 120.681 secs   (18.4833318 %)  Runs: 114(AVG 1.0586053 secs/run)
(a_i0_Comp.c,a4)  sleepTime: 61.918 secs   (9.4832736 %)  Runs: 114(AVG 0.5431404 secs/run)
(b_i0_Comp.f,a1)  sleepTime: 108.981 secs   (16.6913763 %)  Runs: 60(AVG 1.81635 secs/run)
(b_i0_Comp.g,a1)  sleepTime: 99.724 secs   (15.2735872 %)  Runs: 54(AVG 1.8467407 secs/run)

test1.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.test1
Total sleeptime: 175.816 seconds
Actions:
Indiv. Activities:
(a_i0_Comp.a,r)  sleepTime: 67.926 secs   (38.634709 %)  Runs: 37(AVG 1.8358378 secs/run)
(a_i0_Comp.b,u)  sleepTime: 76.544 secs   (43.5364244 %)  Runs: 98(AVG 0.7810612 secs/run)
(a_i0_Comp.c,t)  sleepTime: 31.346 secs   (17.8288665 %)  Runs: 12(AVG 2.6121667 secs/run)

test2.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.test2
Total sleeptime: 1990.241 seconds
Actions:
Indiv. Activities:
(a_i0_Comp.a,r)  sleepTime: 1319.121 secs   (66.2794606 %)  Runs: 678(AVG 1.9456062 secs/run)
(a_i0_Comp.b,s)  sleepTime: 671.12 secs   (33.7205394 %)  Runs: 678(AVG 0.9898525 secs/run)

test4.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.test4
Total sleeptime: 1990.039 seconds
Actions:
Indiv. Activities:
```

```
(a_i0_Comp.a,u)  sleepTime: 1990.039 secs   (100.0 %)  Runs: 4206(AVG 0.4731429 secs/run)

test5.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.test5
Total sleeptime: 1894.211 seconds
Actions:
Indiv. Activities:
(a_i0_Comp.a,a1)  sleepTime: 1894.211 secs   (100.0 %)  Runs: 2615(AVG 0.7243637 secs/run)

test5c.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.test5c
Total sleeptime: 2037.121 seconds
Actions:
Indiv. Activities:
(a_i0_Comp.a,1.143)  sleepTime: 2037.121 secs   (100.0 %)  Runs: 2181(AVG 0.9340307 secs/run)

test6.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.test6
Total sleeptime: 719.956 seconds
Actions:
a  sleepTime: 503.17 secs   (69.8889932 %)  Runs: 202(AVG 2.4909406 secs/run)
Indiv. Activities:
(a_i0_Comp.b,3.0)  sleepTime: 131.189 secs   (18.221808 %)  Runs: 201(AVG 0.6526816 secs/run)
(b_i0_Comp.c,4.0)  sleepTime: 85.597 secs   (11.8891988 %)  Runs: 201(AVG 0.4258557 secs/run)

test8.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.test8
Total sleeptime: 661.429 seconds
Actions:
reqlock11  sleepTime: 10.939 secs   (1.6538434 %)  Runs: 44(AVG 0.2486136 secs/run)
writelock11  sleepTime: 10.585 secs   (1.6003229 %)  Runs: 44(AVG 0.2405682 secs/run)
Indiv. Activities:
(txnc1_i0_Comp.think1,t1)  sleepTime: 540.541 secs   (81.7232084 %)  Runs: 54(AVG 10.0100185 secs/run)
(txnc1_i0_Comp.fork,w1xr11)  sleepTime: 43.172 secs   (6.52708 %)  Runs: 113(AVG 0.3820531 secs/run)
(txnc1_i0_Comp.finish,a11xonemq)  sleepTime: 21.696 secs   (3.2801707 %)  Runs: 113(AVG 0.192 secs/run)
(txnc1_i0_Comp.reqlock1r,rl1r)  sleepTime: 17.11 secs   (2.5868234 %)  Runs: 69(AVG 0.247971 secs/run)
```

```
(txnc1_i0_Comp.readlock,r1)  sleepTime: 17.386 secs   (2.6285512 %)  Runs: 69(AVG 0.251971 secs/run)

test11.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.test11
Total sleeptime: 304.026 seconds
Actions:
reqlock11  sleepTime: 7.82 secs   (2.5721484 %)  Runs: 37(AVG 0.2113514 secs/run)
writelock11  sleepTime: 7.415 secs   (2.4389361 %)  Runs: 37(AVG 0.2004054 secs/run)
reqlock12  sleepTime: 5.156 secs   (1.6959076 %)  Runs: 23(AVG 0.2241739 secs/run)
writelock12  sleepTime: 4.284 secs   (1.40909 %)  Runs: 22(AVG 0.1947273 secs/run)
Indiv. Activities:
(txnc1_i0_Comp.think1,t1)  sleepTime: 234.115 secs   (77.0049272 %)  Runs: 25(AVG 9.3646 secs/run)
(txnc1_i0_Comp.fork,w1xr12)  sleepTime: 35.648 secs   (11.725313 %)  Runs: 60(AVG 0.5941333 secs/run)
(txnc1_i0_Comp.finish,a11xq)  sleepTime: 9.588 secs   (3.1536776 %)  Runs: 59(AVG 0.1625085 secs/run)


test12.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.test12
Total sleeptime: 205.491 seconds
Actions:
reqlock11  sleepTime: 18.661 secs   (9.0811763 %)  Runs: 84(AVG 0.2221548 secs/run)
writelock11  sleepTime: 23.261 secs   (11.3197172 %)  Runs: 84(AVG 0.2769167 secs/run)
reqlock12  sleepTime: 20.171 secs   (9.8160017 %)  Runs: 59(AVG 0.3418814 secs/run)
writelock12  sleepTime: 15.362 secs   (7.4757532 %)  Runs: 58(AVG 0.2648621 secs/run)
Indiv. Activities:
(txn1_i0_Comp.fork,w1xr12)  sleepTime: 102.332 secs   (49.7987746 %)  Runs: 143(AVG 0.7156084 secs/run)
(txn1_i0_Comp.finish,a11xq)  sleepTime: 25.704 secs   (12.508577 %)  Runs: 142(AVG 0.1810141 secs/run)

test13.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.test13
Total sleeptime: 87.586 seconds
Actions:
reqlock11  sleepTime: 14.896 secs   (17.0072843 %)  Runs: 70(AVG 0.2128 secs/run)
writelock11  sleepTime: 19.7 secs   (22.4921791 %)  Runs: 70(AVG 0.2814286 secs/run)
reqlock12  sleepTime: 16.36 secs   (18.6787843 %)  Runs: 72(AVG 0.2272222 secs/run)
writelock12  sleepTime: 17.533 secs   (20.0180394 %)  Runs: 72(AVG 0.2435139 secs/run)
Indiv. Activities:
(txn1_i0_Comp.finish,a11xq)  sleepTime: 19.097 secs   (21.8037129 %)  Runs: 142(AVG 0.1344859 secs/run)
```

```
test14.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.test14
Total sleeptime: 642.681 seconds
Actions:
reqlock11  sleepTime: 88.077 secs   (13.7046217 %)  Runs: 348(AVG 0.2530948 secs/run)
writelock11  sleepTime: 87.669 secs   (13.6411377 %)  Runs: 348(AVG 0.2519224 secs/run)
reqlock12  sleepTime: 57.617 secs   (8.9651009 %)  Runs: 218(AVG 0.2642982 secs/run)
writelock12  sleepTime: 53.147 secs   (8.269577 %)  Runs: 217(AVG 0.2449171 secs/run)
Indiv. Activities:
(txn1_i0_Comp.fork,w1xr12)  sleepTime: 356.171 secs   (55.4195627 %)  Runs: 566(AVG 0.6292774 secs/run)

test15.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.test15
Total sleeptime: 791.742 seconds
Actions:
r11  sleepTime: 68.367 secs   (8.6350099 %)  Runs: 104(AVG 0.657375 secs/run)
w11  sleepTime: 43.683 secs   (5.5173276 %)  Runs: 104(AVG 0.4200288 secs/run)
r12  sleepTime: 131.339 secs   (16.5886109 %)  Runs: 252(AVG 0.5211865 secs/run)
w12  sleepTime: 80.032 secs   (10.1083434 %)  Runs: 252(AVG 0.3175873 secs/run)
Indiv. Activities:
(txn1_i0_Comp.f,a2)  sleepTime: 468.321 secs   (59.1507082 %)  Runs: 357(AVG 1.3118235 secs/run)

test16.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.test16
Total sleeptime: 1167.068 seconds
Actions:
r11  sleepTime: 363.436 secs   (31.1409447 %)  Runs: 529(AVG 0.6870246 secs/run)
w11  sleepTime: 209.64 secs   (17.9629636 %)  Runs: 528(AVG 0.3970455 secs/run)
r12  sleepTime: 156.115 secs   (13.3766841 %)  Runs: 343(AVG 0.4551458 secs/run)
w12  sleepTime: 105.045 secs   (9.0007609 %)  Runs: 343(AVG 0.3062536 secs/run)
Indiv. Activities:
(txn1_i0_Comp.f,a2)  sleepTime: 332.832 secs   (28.5186467 %)  Runs: 343(AVG 0.9703557 secs/run)

test18.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.test18
Total sleeptime: 1298.439 seconds
```

```
Actions:
r11  sleepTime: 280.892 secs   (21.6330532 %)  Runs: 427(AVG 0.6578267 secs/run)
w11  sleepTime: 149.964 secs   (11.5495607 %)  Runs: 426(AVG 0.3520282 secs/run)
Indiv. Activities:
(txn1_i0_Comp.f,a1)  sleepTime: 867.583 secs   (66.8173861 %)  Runs: 427(AVG 2.0318103 secs/run)

test22.pepa
******************************
********* SIM RESULTS ********
******************************
GenSkel.test22
Total sleeptime: 1193.328 seconds
Actions:
Indiv. Activities:
(a_i0_Comp.a,m)  sleepTime: 375.132 secs   (31.435783 %)  Runs: 48(AVG 7.81525 secs/run)
(a_i0_Comp.b,n)  sleepTime: 818.196 secs   (68.564217 %)  Runs: 47(AVG 17.4084255 secs/run)
```

Bibliography

[Appel, 1998] Appel, A. W. (1998). *Modern Compiler Implementation in Java*. Cambridge University Press.

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

[Fotis, 2001] Fotis, S. (2001). Enhancing the PEPA Workbench with simulation and and experimentation features. Master's thesis, School of Computer Science, Division of Informatics, The University of Edinburgh.

[Gilmore, 2001] Gilmore, S. (2001). *The PEPA Workbench: User's Manual*. LFCS, University of Edinburgh.

[Gilmore and Hillston, 1994] Gilmore, S. and Hillston, J. (1994). The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, number 794 in Lecture Notes in Computer Science, pages 353–368, Vienna. Springer-Verlag.

[Gilmore and Hillston, 1996] Gilmore, S. and Hillston, J. (1996). Refining internal choice in PEPA models. pages 49–64, Department of Computer Science, The University of Edinburgh.

[Gilmore et al., 1996] Gilmore, S., Hillston, J., and Holton, D. (1996). From SPA models to programs. pages 179–198. Dipartimento di Informatica, Università di Torino, CLUT.

[Hidders, 2001] Hidders, J. (2001). Wikipedia definition: Parser. http://www.wikipedia.org/wiki/Parser.

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

[Hudson, 1999] Hudson, S. E. (1999). *LALR Parser Generator for Java: CUP User's Manual*. Graphics Visualization and Usability Center, Georgia Institue of Technology, http://www.cs.princeton.edu/ appel/modern/java/CUP/manual.html.

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

[Lea, 1997] Lea, D. (1997). *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley.

[Siegrist and Duehring, 2001] Siegrist, K. and Duehring, D. (2001). The probability/ statistics object library: Mathematical distributions java package (edu.uah.math.distributions). The Department of Mathematical Sciences, The University of Alabama in Huntsville (http://www.math.uah.edu/psol/). GNU General Public License.