University of Edinburgh
Division of Informatics

Finite State Machine Visualiser
for Discrete Event Simulation
A Visual Debugger for the PEPA Workbench

4th Year Project Interim Report
Computer Science

Antony Rinaldi

March 2, 2005

Abstract: The PEPA workbench[5] is a tool developed within the University of Edinburgh to allow the study of performance models written in the Performance Evaluation Process Algebra (PEPA). The workbench contains a rudimentary text-based debugger. This type of model is naturally suited to visualisation due to its state-based nature. Through this visualisation it should be easier to comprehend this large amount of information. This project aims to develop a useful visual single-step debugger for the PEPA workbench enabling easy visualisation of discrete event based models.

Acknowledgements

I would like to thank Eric McKenzie for originally proposing this project and for guiding me through to its completion!

I would also like to thank my friends and family for their support through many late nights of programming and LaTeX

Additional thanks go to Stephen Gilmore and Valentin Haenel for their input along the way.

Contents

1. Introduction

Simulation, in all of its varied guises, has one base premise; through construction and observation of a model the behaviour of some system (mechanical, economic etc...) under certain conditions can be understood and/or predicted. Discrete Event Simulation (DES) specifically sets out to model a dynamic system through abstraction of "events" within the system.

DES contains two essential concepts: Entities and Logical Relationships. Entities represent the "physical" attributes of the system and could be a robotic arm or conveyor in a manufacturing line. Entities can be permanent as with the previous example or temporary such as an item being manufactured passing down the line. Indeed it is often the case in DES that models consist of temporary entities passing through or being processed by permanent ones.

Logical Relationships embody the links between system entities and are analogous to relations within a Relational Database System. As with Relational Databases it is not the complexity of the relationships themselves (e.g. conveyor C starts when a part is waiting) but rather the number and distribution of such relationships throughout the system that create the over-all complexity of the model.

Traditionally a DES model is initialised with some given starting parameters and left to run until termination. The results are then analysed and any interesting information extracted. In industrial usage this is often adequate and existing DES systems exist and function well within this space. However such techniques do have their drawbacks, for instance specific unusual behaviours of the system may be missed as they are eventually averaged out of results produced. It can also be difficult to intuitively know whether a model is correct or not especially when only presented with a text-based representation.

1.1 Why PEPA?

There are a number of ways of representing DES with varying degrees of popularity and flexibility. However there existed some basic criteria that an ideal representation (language) for this project should fulfil:

- It should be flexible.
- Currently in use to keep work relevant.
- It should preferably have an existing tool-set to aid in testing / evaluation.

- It should have at least a small group of users within the University of Edinburgh for expert advice and opinion.
- It should preferably be simple!

Stochastic Process Algebra (SPA) can be both simple and flexible however one SPA in particular demonstrates the other properties required, Performance Evaluation Process Algebra (PEPA). PEPA as described in [6] is an ideal candidate language; it was created in the University of Edinburgh, has a wide user-base both within and outwith the university and has a well developed bank of software already in place.

SPAs in general are known to be an extremely good modelling paradigm with wide applicability due to their properties of:

- Compositionality; systems are built out of small units which are combined to form larger units and so on until an entire model is formed.
- Parsimony; the entire algebra consists of very little special syntax.
- Expressiveness; the algebra is very flexible and can be turned to any manner of application.
- The ability to derive performance measures automatically.

However despite this apparent appropriateness to the task they lack wide-spread use. This is due, in no small part, to the lack of general experience in using such tools. The existence of software to derive models from specifications of systems (such as UML into PEPA) has however led to a few taking up SPA as a modelling solution.

The PEPA Model in Figure 1.1 is a very simplistic model of the interactions between a set of computers joined by a token-ring network. It consists of 4 identical Computers (PC1x, PC4x) and a representation of the token (Sx). The model embodies the system thus: At a given point the token is held by one of the 4 computers. If this computer has data to send it transmits it and the token moves. If the computer has no data to send the token simply moves on. Additionally each computer may become ready to send at any point.

1.2 Previous PEPA Visualisation Work

There has been previous work into the visualisation of PEPA models, most notably by Thomas, Munro, King and Pooley [10]. The target of this work was the modelling of software systems. This drew focus to the ability of PEPA models

```
// A PC LAN with 4 clients
// 128 States

%lambda = 2.0;
%omega = 2.0;
%mu = 2.0;

#PC10 = (arrive1,lambda).PC11 + (walkon2,infty).PC10;
#PC11 = (serve1,infty).PC10;

#PC20 = (arrive2,lambda).PC21 + (walkon3,infty).PC20;
#PC21 = (serve2,infty).PC20;

#PC30 = (arrive3,lambda).PC31 + (walkon4,infty).PC30;
#PC31 = (serve3,infty).PC30;

#PC40 = (arrive4,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
```

Figure 1.1: A simple PEPA model (PC-LAN4.pepa)

and SPA models in general to be derived from UML models, a natural design tool of the software engineer.

The work proceeded along the lines of attempting to create the visualisation of the model in a way most suited to Software Engineers. Ideally this would present the model in a manner that Software Engineers were used to. The result of this work, a prototype web-browser based navigation tool that produced output similar to a profiler used in programming.

This PEPA "profiler" produced an image representing the state-space embodied by the system and demonstrated how a system moved from one state to another. This is very like the way traditional profilers display data. It included various possible aids to the user such as emphasis on heavily used states, of course this could equate to a bottleneck in the real system.

This visualisation method can create certain problems, most markedly that of an exponential state-space explosion. Even relatively simple models can have a large number of states e.g the PC-LAN example above produces 128 states and 384 transitions and a similar example extended to 6 clients contained 768 states and over 3000 transitions. In TMKP [10] the visualisation of such large systems on a screen was tackled by techniques such as the aggregation of nodes or simply allowing the whole graph to be scrollable. The other major draw-back of this system is the specificity of its audience; the likeness of the system to a profiler is of course useful to Software Engineers but to others could itself be a stumbling block.

What this system lacks is the ability for the user to easily see how different system components are interacting to perform a task. Obviously such information can be extrapolated from state information provided in the profiler but to the uninitiated it is certainly less than apparent. It is in this area that this project aims to proceed, the premise being that through direct visualisation of and interaction with the model the user could gain a better understanding of how the model components are functioning. In essence this system should aid someone who has not necessarily designed the model in question to understand and work with it.

1.3 PEPA Workbench

As discussed, given PEPA's status within the University and the field as a whole it is a natural choice for representing the discrete event models. Conveniently the main tool used to work with PEPA models, the PEPA workbench was also developed in Edinburgh University. It contains all the important language parsing and lexical analysis elements along with mathematical evaluation systems that are useful when working with models. After consulting with various users of the

PEPA Workbench concerning visualisation based on PEPA it became apparent that an interesting addition to the PEPA workbench could be made that would also fulfil the goal of a visualisation system for DES.

Currently the PEPA workbench contains, as part of its toolset, a text-based single-step debugger with forwards and backwards stepping, it does fulfil its task well. However the amount of information can easily become overwhelming, due mainly to the length of the text required to represent a state of the system.

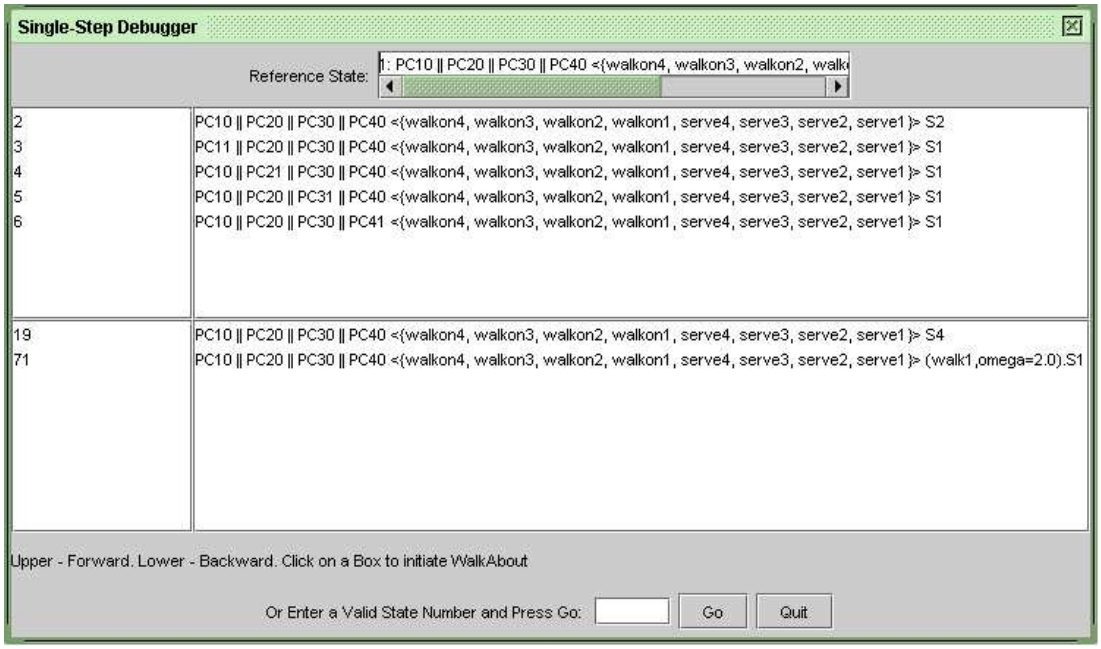


Figure 1.2: Text Based Single Step Debugger displaying state 1 of PC-LAN4.pepa

This lack of clarity stems mainly from the fact that the text debugger represents the entire state of the system including all cooperation sets. Although this method does convey a lot of information it misses out some important details. For instance it is difficult to tell which actions are enabled without close examination of the model code and the output from the debugger. This can be laborious and can lead to confusion especially when the number of connected components and cooperations becomes large. It seems that a more natural solution is to allow the user to choose which action is performed at any point and then view the results.

The representation of this information visually could potentially reduce confusion especially with less experienced users. It would also allow users to follow the progress of a simulation in a more natural way than is currently available in the text-based debugger.

1.4 Motivation

The concept of a visual debugger for the PEPA workbench provides some interesting possibilities in terms of functionality above and beyond a simple visualisation tool. The most obvious additional feature required is that of interactivity; some way of allowing the user to step through a possible chain of actions in the system and view the results in real-time.

It must also be noted that the PEPA workbench (the Java[8] edition in particular) was designed with education in mind. Considering this it has very few features to aid a new user in understanding how the models themselves are functioning, especially in instances where the user is working with models created by others. Indeed after initial discussion with newcomers to PEPA it was generally felt that the existing tools were of little or no use.

These challenges along with the opportunity to produce something new and useful (from personal experience) were major factors in the decision to take this project in a far more specific direction than was originally intended i.e. producing a Visual Single-Step Debugger for the PEPA Workbench.

1.5 Goals

The overall goal of this project is to produce a Visual Single-Step Debugger for the Java Edition of the PEPA Workbench. There are also a number of high level goals to be met:

- The tool should be able to represent PEPA models in an intuitive manner. This is somewhat subjective so this will be established through discussion with users of the system.
- The tool should allow interaction with the model.
- The tool should be easy to use (see above).
- The tool should function with at least the standard examples supplied with the PEPA workbench.
- PEPA models should require no modification to work with the tool.

There are also a number of lower level "features" as requested by users of the PEPA Workbench.

- The tool should make a best-try layout attempt but also allow the layout to be changed manually
- The tool should allow visual layouts to be saved and loaded.

- The tool should allow some sort of simulation to be run on the model.
- The tool should contain some notion of history.

2. Design

Throughout the design phase of the project two main factors had to be continually reassessed: The usefulness of the tool in terms of a debugger and the usefulness of the tool in terms of a visualiser.

2.1 Visualising PEPA

A PEPA model (in the simplest sense) is made up of a series of components interacting to perform activities. A component represents an active unit of the system such as a single PC in a LAN; the activities capture the actions of those units e.g. a PC on a LAN becoming ready to send data. In PEPA an activity is denoted by a pair (a, r) where a denotes the Action type of the activity and r denotes the activity rate. However single components themselves are not particularly interesting and are of little use. The compositionality of PEPA allows us to create more complicated and interesting connected components out of single components and activities. We can also allow components to cooperate with other non-connected ones in order to represent interactions of units in our systems; a PC on a LAN must cooperate with the token in order to transmit data. The following conjunctives are available for use in creating connected components.

(.) prefix (sequence); one activity follows another.

$$P_0 \stackrel{def}{=} (a, t).(b, u).P_1$$

(+) choice, a component has alternate possible sets of activities to perform.

$$Q_0 \stackrel{def}{=} (a, t).Q_1 + (b, u).Q_2$$

(<>) synchronisation (cooperation), occurring over a set of Action types, cooperation requires the synchronised performance of that Action in each participating component. An empty Action set can be seen as parallel and independent behaviour.

$$P_0 \underset{\{a,b\}}{\bowtie} Q_0$$

Deciding how to visualise each of these conditions is the most delicate, and important element of this project, ideally different visualisations will be possible however time is an ever-present limit on how much can be done.

As stated earlier, the aim here is to allow direct visualisation of the model, to this end there must be some visual way to represent the various entities and

conjunctives within the model. If a basic graph is selected as the starting point for the system (this is natural as a PEPA model can be viewed as a description of a Finite State Machine) parallels can be drawn between graphing elements and the PEPA entities. Initially an individual component can be taken as a node in the graph. A component will perform some sequence of prefixed activities (possibly none) before entering another component (possibly itself), therefore a sequence of actions can form an edge between components. For clarity activities can be emphasized as minor nodes along the path of the edge. A component with a choice is simply a component with multiple outward-bound edges. These properties display a connected component as a single connected graph or FSM.

This possible multiplicity of edges introduces the fact that simple straight edges from node to node are inadequate. Not only can edges flow in both directions between nodes but multiple edges may exist in the same direction i.e. a node A can have multiple direct paths to node B. In either of these cases simple straight edges would result in a total loss of clarity: Action nodes from different edges would all be visible along the same apparent edge. Also with only straight edges it is of course impossible to display a simple loop!

As a result there are three edge types: Straight (single), Curved (multiple) and Loop. The type of a given edge is decided by a simple comparison of the start and end points along with a counting of the number of connecting edges. Multiple edges from node A to node B can be easily displayed by drawing each edge with an ever increasing radius.

Now that a concept of the representation of the graph is in mind consideration must be given to how the state of the system is represented and also how interactions with the graph are performed. The manner in which the state of the system is displayed rests on the properties that an element may have and also on what possible attributes Java2D provides to portray this property.

As covered certain system elements have been assigned to graph elements due to their natural similarity. Figure 2.1 shows system elements and variables that the user may wish to have displayed, and also shows the analogous graph elements and their available attributes.

Obviously not all properties / dimensions need be used to visualise the system effectively, however the selection of variables / attributes could be a major determining factor in clarity and therefore usefulness of the visualisation. In order to aid in the clarity of the visualisation it would seem prudent to use a given graph attribute to convey only one piece of information / variable at a time.

Firstly, given the fact that both components and activities are considered as nodes it should be possible to easily differentiate one from the other. Each node should also be differentiable from other nodes of the same type. The natural way for the types to be differentiated is probably the size, if for no other reason

System Element	Variable	Graph Element
Component	Status : current, inactive	Node
Activity	Status : active, next, inactive, blocked	Minor Node
Choice	Rate (or probability)	Edge

Graph Element	Attribute
Node	Size, Shape, Colour, Alpha
Edge	Thickness, Colour

Figure 2.1: System Elements / Graphing Elements

than an activity is being considered a "minor" node. Given this difference in status and also the fact that the graphs in general contain far more activities than components making the node representation of the activity smaller than that of the component is an obvious step to aid clarity. Nodes of different types could also be given a different shape however this would add little or nothing to difference already expressed by the size. In terms of being differentiable from nodes of the same type they already have some form of uniqueness due to their topological position. However to help with the identification of specific nodes each one will be labelled with the corresponding component name or action type.

Following this trend both the components and activities share a common variable: status, it therefore makes sense to portray this in a common fashion between the two. What attribute is used to convey status must be able to be immediately recognisable by the user taking no real thought to perceive This is useful due to the fact that unlike the other properties of a node, the status changes often and in the case of using the simulation engine probably quite rapidly. It can therefore only be colour as shape when changing rapidly is far less clear and when considering the states; inactive, current, next and blocked there exist some natural colour choices. As stated in [1] there exist some standards when relating colour to the concept of progress that should be adhered to. Next and blocked map simply to green and red (as with traffic lights), current and inactive on the other hand have no truly natural mapping. Inactive could be considered unimportant and almost anonymous so colouring it as the background could work well, only showing its outline to the user. Current is important as it must stand out from the other colours, it also seems sensible to choose a neutral colour so blue is selected.

In terms of the choice / edge aspect we must convey the fact that for a set of edges from a node the one with higher rate is more likely to be traversed. We can view this as a "capacity" for the edge, naturally a larger capacity requires a larger edge to accommodate it so the thickness of the lines representing the edges could be varied to show this. Edges also have direction, this will be shown by solid arrow-heads at the end of each edge.

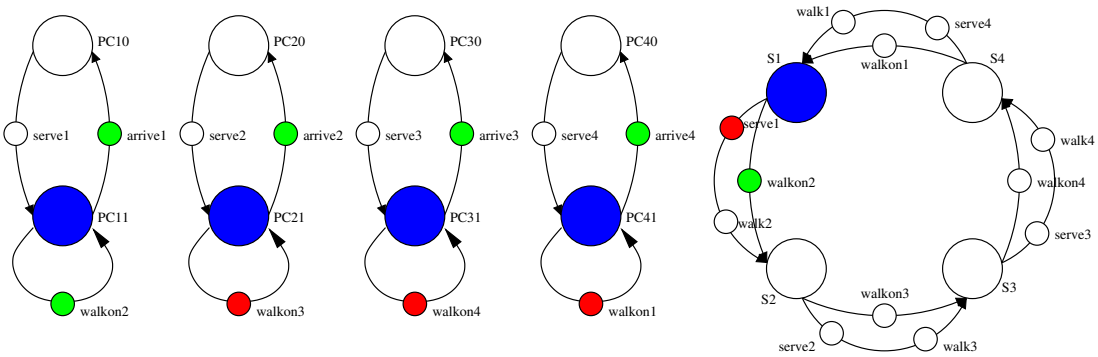


Figure 2.2: Concept visualisation of PC-LAN4.pepa

2.2 2D or 3D (graphics)

Now that consideration has been given to what form the visualisation should take, the method by which this representation is given to the user must also be considered. In the above example a simple two-dimensional graph has been used as it clearly demonstrates the layout of the nodes in the system. Of course it is possible for PEPA models to be far more complex and contain a high number of interconnected elements. In such instances it is obviously preferable to produce graphs in 3D as it is extremely unlikely to be able to produce a graph that cannot be arranged in such a way as to have no (or at least minimal) crossing edges.

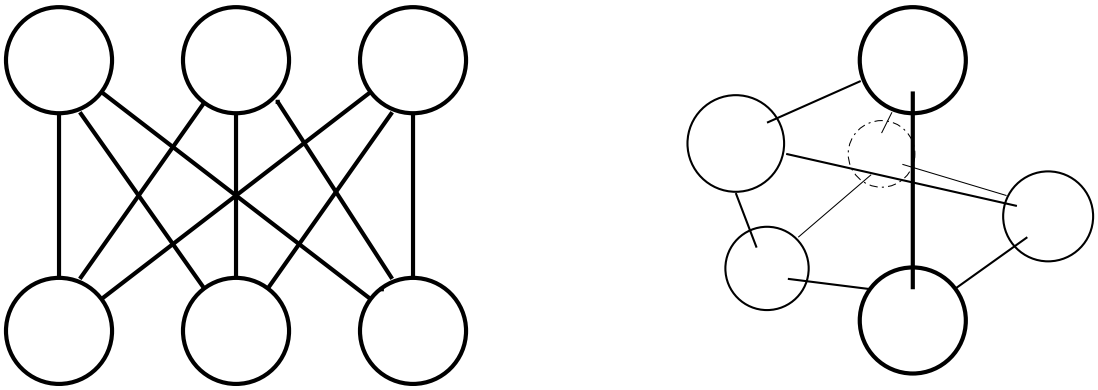


Figure 2.3: Demonstration of crossing edges in 2D and 3D

Perhaps the ideal is a switchable 2D / 3D display system, only using 3D when clarity is lost in the 2D visualisation or at the user's request. It is this system that would be preferred for the PEPA visualisation tool, however such a system would be extremely time-consuming to implement and really would only be an extension to this project. Also given the fact that PEPA models are in general fairly loosely connected a 2D representation only will be used for this project.

However as work proceeds the possibility of extension to 3D will be kept in mind, preliminary work may also be made if time allows.

2.2.1 Graphics Library

There are a large number of packages available when producing visualisation software and as such there was some consideration required when choosing the package to be used. As with the choice of target representation language a number of criteria would determine the library best suited to the task at hand. Firstly the package, whatever its form should be platform independent and come with Java bindings as this is the language of the current PEPA workbench. The package should be simple to use and produce easily readable code. The package should be as light-weight as possible and not require any additional software (that cannot be included with the PEPA workbench) to be run.

OpenGL[4] was considered as it is fairly simple to use but is not entirely simple to integrate into Java (Java 3D can be bound to it) and it is also principally used for 3D work. VTK[7] (Visualisation Toolkit) was also considered and its pipelined approach to visualisation did seem interesting and given work could make some very useful visualisations of PEPA, however it is far from light-weight and is not very simple to set-up on Windows-based systems.

Finally Java2D[8] (AWT / Swing) was selected, this allows the code for the visualisation tool to be pure Java, using no libraries outwith the standard JVM run-time libraries. This obviously means it is extremely lightweight and also a 2D-only API but given the desire for future extension to 3D its similarity to Java3D[9] will allow as natural a progression as possible to made. Another important advantage of the Java2D is the ability to add interaction with the user in a very simple way.

2.2.2 Layout method

The greatest proportion of the graph rendering problem is deciding on the physical positioning of the nodes within the graph without the need for human intervention. This problem falls to the graph layout algorithm, many such algorithms have been created over time, varying widely in their styles and results. This large variation is mainly attributable to the subjective nature of exactly what a "good" graph layout is. In the case of connected graphs such as are being visualised here, the main concern is having a layout that looks "even", i.e. with few (ideally no) crossing edges and connected nodes not being an exaggerated distance apart.

Peter Eades' 1984 paper [2] on force-directed placement for generalised graph

layout would seem of particular interest in this case. The heuristic therein models a graph as a physical system with nodes as "rings" and edges as "springs". Thus the edges pull connected nodes together logarithmically (as opposed to using Hooke's law) and rings repel each other. Over repeated iterations of the algorithm these opposed forces will even out and the system will enter a state of equilibrium, in this state the resulting graph can be generally taken to be aesthetically pleasing and will generally display few or no crossed edges. As such the algorithm will generally expose symmetry and geometric structure very well, it is these properties that make it attractive for laying out the model graphs as such patterns can be useful in immediately identifying flawed models.

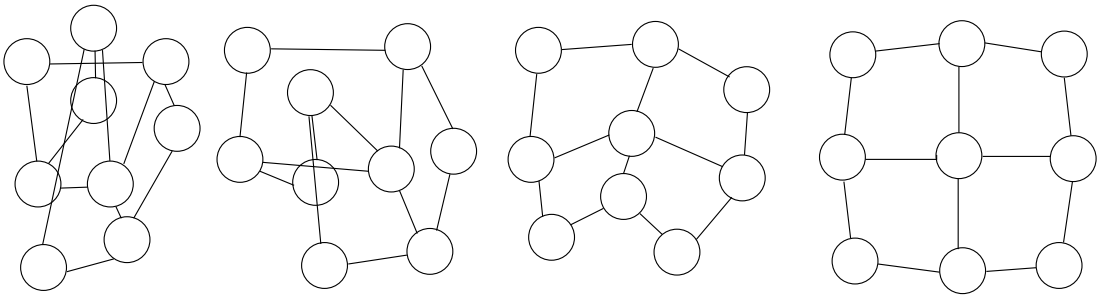


Figure 2.4: Example of iterations of a Force Directed Placement algorithm

Eades' original algorithm can be easily modified to display required characteristics and to curb undesired behaviour, it can also be simply extended into 3D space maintaining the possibility of expansion in this direction with little modification to the layout portion of the tool.

2.3 Debugger Functionality

So far only the visual representation of the model has been discussed and the functional debugger aspect has only been briefly touched upon. This tool is required to allow navigation of the state space through interaction with the graph(s) of the model. As mentioned earlier this should happen through the user clicking on possible "next" nodes, the system is then moved on into the state represented by that node's activation. Due to the cooperative nature of PEPA models it is possible to move into a certain state through many different cooperating nodes simultaneously.

It must also be considered that Graphs produced could be quite large and should still be viewable. This requires some way to navigate around the graph, as if a graph consists of many nodes it would be difficult to fit on screen. This requires that the graph be navigable through use of zoom / magnification tools as well as some sort of directional navigation through scroll bars for example.

A specific feature requested by the user is that of customising the layout, so the nodes must be able to be moved by hand and have all edges etc... update accordingly. These moves should of course not be subject to the restrictions of the force-directed placement used initially. These changes should be able to be saved for use at a later date. After discussion with the users it was decided that this save file should be in XML so as to be human-readable and editable, this also allows for the graphs to be easily processed using an external layout tool if required for extremely complex graphs.

Another feature requested by the user is the ability to run a simulation upon the model. This should take the form of an emulated user stepping through the graphs in weighted-random fashion. The weighting itself is derived from the PEPA concept of rate when concerning activities. Obviously an activity with a higher rate is more likely to happen at any given moment. This method of selecting which action to execute is discussed in Hillston [6]. The simulation should be able to produce traces to file so that the user may follow exactly what happened, to see how a system reached dead-lock for instance.

There are of course many other functions that are desirable in a debugger of any sort such as custom instrumentation of objects and break-points. However such functions in this instance are only useful given the automated simulator portion of the project and as such are a much lower priority over all and could easily be implemented at a later date.

3. Implementation

3.1 Visualisation

The visualisation of the PEPA model is to be produced using Java2D, it is implemented through a series of custom components. The rendering of the graph itself is performed through the passing of a common Graphics object that can then have required painting applied by successive objects.

$$\begin{aligned} VisualSingleStepDebugger->GraphManager2D \; &-> \; Edges \\ &-> \; ProcessNode \\ &-> \; ActionNode \end{aligned}$$

Figure 3.1: The flow of the Graphics object though

3.1.1 Constructing the Graph

Before anything can be drawn or any interaction can take place an internal representation of the graph must be constructed based on the information concerning the model already possessed by the PEPA workbench. This information is stored in the Process object and its subclasses (Figure 3.2)These classes are combined to form a tree representation of each component in the system (Figure 3.3). Using this information it is possible to construct a visual component, a ProcessNode (component) and all if its associated outward-bound Edges (activity chains). This process is performed recursively within each ProcessNode object once upon initialisation of the visual debugger. The algorithm used for this processing is shown in Figure 3.4. The results of the application of this algorithm to the example in Figure 3.3 are shown in Figure 3.5.

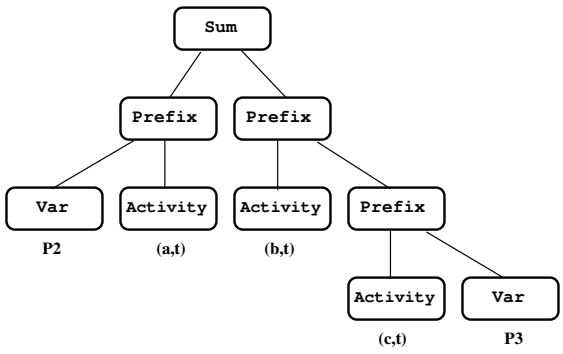
Once all of the ProcessNodes, Edges and associated ActionNodes have been created it is then required that Graphs be built out of them. This would be a simple task, however in PEPA the resultant graphs can be non-connected so it is not just a matter of picking a node and traversing edges to discover the members of a given graph. The solution used does follow a broadly similar principle however before joining a graph the connected neighbours of a node n are checked to see if the connected node nc already has a graph if so n joins this graph instead of a new one.

Process Subclasses	Function	Contains
Var	The Var subclass represents a variable name. For a given component this is the other component at the end of a series of activities.	1 String: the name of the component referenced.
Prefix	The Prefix represents a sequential activity.	1 Process: the rest of the chain. 1 Activity: this activity
Sum	The Sum represents a simple choice between two possible paths	2 Process: representing the string of activities on etiher side of the choice.
Coop	The Coop represents the set action types over-which a pair of components cooperate.	2 Process: Either of type Coop or Var. 1 QSet: Containing all action types involved in this cooperation.

Figure 3.2: Existing sub-classes of Process

$$P \stackrel{def}{=} (a,t).P2 \\ + \ (b,t).(c,t).P3$$

(a) PEPA Model Statement



(b) Equivalent Process tree

Figure 3.3: Example of PEPA model and Process tree

```
ProcessNode.extractProcess( Process p ){
   if(p is a Var){
      create a new Edge between this ProcessNode and p;
      return
   }
   if(p is a Sum){
      extractProcess(p left operand);
      extractProcess(p right operand);
      return
   }
   if(p is a Prefix){
      extractProcess(Process following p)
      add Action represented by p to last Edge created
      return
   }
   else{
      return
   }
}
```

Figure 3.4: Algorithm used for Process extraction

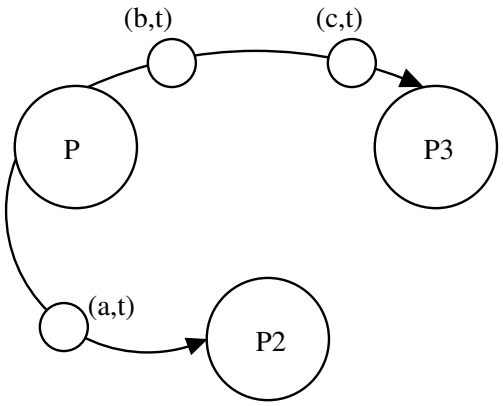


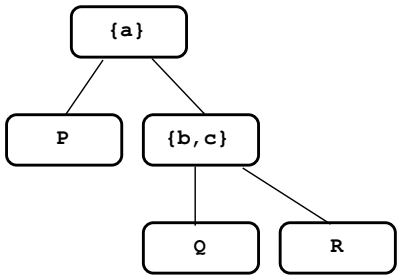
Figure 3.5: Results of algorithm in Figure 3.4 run on Figure 3.3(a)

```
Graph.createGraphs(){
   for each ProcessNode p{
      if p has no graph{
         g = new Graph
         p.findGraph(g)
         if g has no members discard it
      }
   }
}

ProcessNode.findGraph(g){

   graph = g

   for each ProcessNode pn with an Edge to this{
      if(pn.graph != null){
         graph = pn.graph
         break
      }
   }

   graph.addNode(this)

   for each ProcessNode pn with an Edge to this{
      if(pn.graph = null){
         pn.findGraph(g)
      }
   }
}
```

Figure 3.6: Algorithm used find Graphs

$$P \underset{\mathcal{K}}{\bowtie} (Q \underset{\mathcal{L}}{\bowtie} R)$$
where $\mathcal{K} = \{a\}$ and $\mathcal{L} = \{b, c\}$

(a) PEPA System Equation



(b) Equivalent Coop tree

Figure 3.7: Example of PEPA system equation and Coop tree

Once the graph structures have been read in, the final step in creating the internal representation of the graphs is to create a set of relationships to represent the cooperation pairs and sets. These details are taken from the system equation that creates the PEPA model from the connected components; this equation is of the form discussed previously for PEPA cooperation statements. The representation of this within the PEPA workbench is again a tree-structure, this time consisting of only Process.Coop objects. This tree is interpreted in a very similar way to the extractProcess function above; in this case each iteration returns the set of graphs it used, the root returning all the nodes of the graph.

3.1.1.1 Edges

The three edge types; straight, curves and loops were defined in section 2.1. An area of significant investigation was that of how best to represent these given the tools available within the Java2D framework.

Straight edges were of course simple to implement as Line objects could be used. These edges also allow for simple placement of ActionNodes which are placed along the edge proportionally.

Curved edges caused somewhat more of a problem. Java2D does include very flexible Curve2D objects representing various classes of curve. Unfortunately it is not possible to iterate along these curves with any in-built functions and this is required to place ActionNodes along the path of the edge.

There exists in Java2D a GeneralPath object which allows a path to be made up of numerous sub-curves, lines and points. The GeneralPath allows for the iteration along its sub-paths but only one segment as a time i.e. for sub-curves $c1$, $c2$ and $c3$ it is only possible to iterate the control points of each of these curves. What this implies is that to place n ActionNodes along the path of a

curve the GeneralPath must contain $n + 1$ sub-paths. Creating these sub-paths is reasonably simple, however to create them to appear as one complete smooth curve is more complicated. This in turn leads to a circular argument due to the required location of the start and end points of the sub-curves. These points must be placed in an evenly spaced manner along the main curve, in other words these are the very points that this solution hopes to find!

The solution comes through using another Java2D class. the Ellipse2D. Using this it is possible to define an ellipse and create a path from any arc along its edge. It is possible to create a evenly distributed set of $n + 1$ arcs by producing these at $180/n + 1$ deg each. This principle is shown in Figure 3.8.

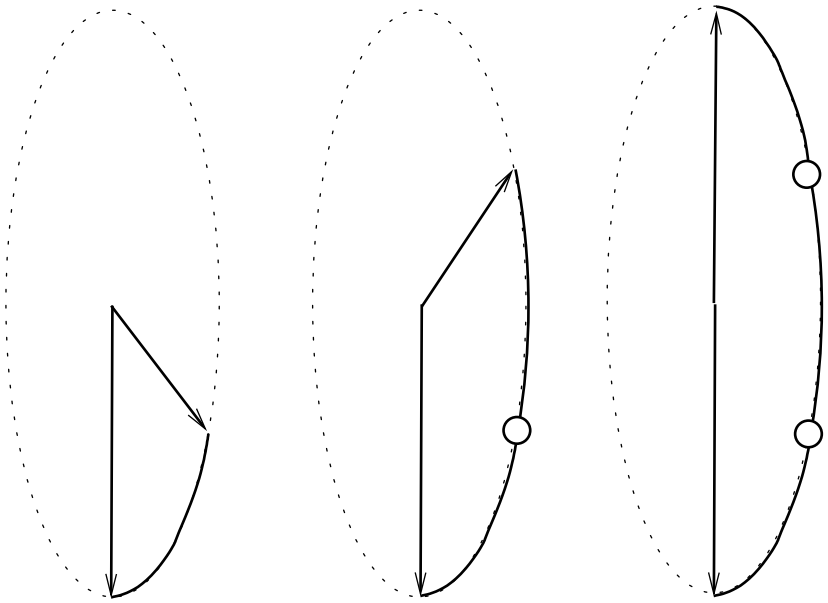


Figure 3.8: Demonstration of drawing curve sections using ellipse arcs

Loops are produced in a broadly similar fashion to the curves described above however in this case the curves are a complete circle.

It should be noted at this stage in the implementation that the design called for variable edge widths to represent the probability of that edge being taken. It was soon discovered that the range of edges realistically available (not too narrow or too wide) was very small. This problem coupled with others such as the difficulty in dealing with "infinite" rates or the changes in effective rate due to cooperation caused this idea to be left-out of this prototype system.

3.1.2 Layout

The layout algorithm implemented is almost exactly that of Peter Eades 1984 force-directed placement algorithm displayed in Figure 3.9. This algorithm was designed with this static-placement scenario in mind.

The repellent force between any two vertices is given by:

$$Repulse(d) = C3/d^2$$

The attractive force between any two nodes connected by an edge is:

$$Attract(d) = C1 \times \ln(d/C2)$$

Where d is the distance between vertices
1.place vertices of graph G in random locations
2.repeat M times:

a)calculate force on each vertex $\vec{F}$ using above functions
b)move vertex by $C4 \times \vec{F}$

Constant	Recommended value	Value used
C1	2.0	2.0
C2	1.0	1.0
C3	1.0	12500.0
C4	0.1	0.1
M	100	500

Figure 3.9: Eades' Force Directed Placement Algorith and constants

This algorithm is designed to be applied iteratively as many times as required to produce what is considered a "good" layout. As can be seen in the table above only values of $C3$ and M are varied in this implementation of the algorithm. The change to $C3$ comes about as without it the layouts are far too small in the coordinate system used, and appear as all nodes of the graph resting on top of one another. The value shown above was found through simple trial and error and seemed to give good graph sizes relative to the initial window size. The iterative constant, M was altered as with the recommended 100 graphs appeared to still be in a state of flux and would not settle for another 200 to 300 iterations, 500 was chosen as a safe number as the graph should generally have settled in a minimum by this point. In some instances of more complicated graphs this state may be only a local minimum in terms of the energy of the graph. Even in such a local minima the layout produced, in virtually all instances, is still a "good" layout in terms of being geometrically even and pleasing to the human eye. However it may contain more crossing edges than the optimum obtainable solution.

Some thought was given as to how exactly the forces should be applied to PEPA graphs given the two classes of node (Process (component) and Action (activity)).

Action nodes as discussed are placed along graph edges to draw emphasis to the current activity state of that edge. It therefore seems natural that these nodes should have no influence upon the position of the edge which they lie along. Thus the Process nodes act as the vertices of the graph and are used for FDP calculations, the Action nodes are then placed evenly along their respective edges.

Although the algorithm described works well on most graphs, problems occur when applying it to multiple disconnected graphs as are generally generated by a PEPA file. The main problem faced is that of graphs "flying" off to infinity with no attractive forces to keep them in proximity. This can be solved by simply adding the required attractive (and repulsive) forces between graphs, however this will generally lay the graphs out in a circular fashion, this is contrary to the expected layout which would follow the PEPA system equation. The solution deemed most suitable in this case is to perform layout on a per-graph basis then place each graph on the screen side-by-side in the order determined by the PEPA system equation.

3.2 Interaction

The goal of this project is to not only create a visualisation system for PEPA models but also allows these models to be interacted with, fulfilling the role of a debugger of sorts. This interaction is handled through the Java interaction system utilising EventListeners etc The active component is the GraphManager2D, this provides an interface to the visualisation both directly through the user clicking on the graphs and to the other debugger components such as buttons and text fields.

3.2.1 Interpreting State

Probably the most complicated part of the project as a whole is interpreting the state of the system at any point in time. The state control portion of the tool (the StateInterpreter) is built upon the foundation laid by the PEPA Workbenchs existing Single Step Debugger. The aim of the StateInterpreter is to examine the strings representing the current state and possible next states of the system and colour the graph accordingly , as well as assigning too each "next" node the number of the state it represents.

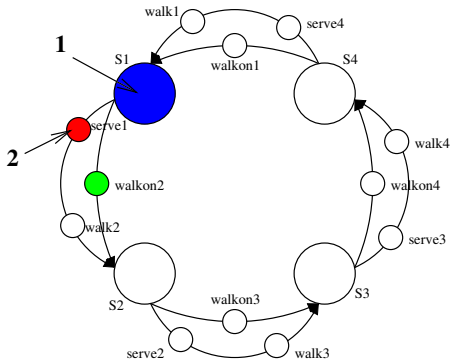
To perform these tasks thought must first be given to what state is represented by a given node. The two types of node represent state in a different way: For a ProcessNode the state can be simply read off as the name of that state. For ActionNodes however the state represented can be found by repeatedly prefixing

the states of the ActionNodes following it on its edge until a ProcessNode is encountered.

The states for the two labeled nodes are:

1 = S2
2 = (walk2,omega=2.0).S2
  = (serve1,mu=2.0).(walk2,omega=2.0).S2

(a) Node states



(b) Equivalent Coop tree

Figure 3.10: An Example of State Text for two nodes.

Now it is possible to start to interpret the state text available to the visualiser. The first step is to decide upon the nodes representing the current state of the system. These can be identified by comparing the state text for each node (Figure 3.10) to the list of states found in the system equation describing the current state. Each nodes state text that is found therein is marked as current. There are two additional factors to be considered for correctness: If a ProcessNode is current and it is connected by a no-action transition to another ProcessNode (e.g. $P = (a, t).C + Q$) then the other ProcessNode (Q) is also marked current. Also an ActionNode can only be marked current if its previous state was "next".

The next step is to highlight possible next nodes. This is performed in a similar manner to that of current states however in this case there are multiple states to be considered. Initially we can narrow down the nodes to be examined as only those that are one-step from each node now marked current. For a current ProcessNode these are the first ActionNode on each edge, and for a current ActionNode this is the node immediately following it. Now each node in the one-step set must be checked to see if the state it represents appears in any of the possible next system states. If so it is marked as next and has the corresponding state number assigned to it (this is only provisional) if it does not it is marked as blocked.

The following step is the most complicated part of the StateInterpreter system. It is also the most important as through this step the cooperations of the components of the model are displayed. This step starts with a list of every currently enabled node, this list is then iterated over with the following algorithm:

The first part of the algorithm determines first if this node is being cooperated on

(if not it is left enabled). If the node is cooperated on then every graph cooperating on the node must also have an instance of the same type of action enabled. This is a deep search as just checking graphs that cooperate directly with ANs graph could lead to unwanted additional iteration of the main loop. The second part of the loop makes sure that all cooperating actions are activated no matter which instance is selected by the user. This means that if a set of cooperating nodes reach this stage of the algorithm there exists a possible next state in which all cooperating actions have been performed. This part of the algorithm functions by checking each possible next system state against the individual states of all cooperating nodes; as soon as one is not included that system state string is rejected. The first acceptable state is retained and its number assigned to all cooperating ActionNodes. Upon the next repaint the highlighted nodes (next, current and blocked) will now correctly represent the state of the system.

3.2.2 Simulation Running

In order to achieve one of the goals requested by the user a basic simulation system is to be implemented in addition to the manual single-step debugger. In essence this simulator will perform the same actions as the user would, however in a weighted-random fashion.

Set of all enabled actions E
Set of all types in E is T
Set of all activities of type t is $A_t \subset E$
r_t min rate of all activities in A_t

r_t is the determining rate of action type t

R is the set of all r_t

let T_∞ be set of all t such that $s_t = \infty$

if $|T_\infty| = 0$ then
probability of given type t is $r_t \times \frac{1}{\sum_{x \in T_\infty} r_x}$
else

$$\text{probability of } t = \begin{cases} \frac{1}{|T_\infty|} & \text{if } r_t = \infty \\ 0 & \text{otherwise} \end{cases}$$

The selection made is based on random doubles between 0 and 1 produced by the Java Random object.

The SimulatronRunner allows the user to interact with the graph whilst it is

running. This allows the user to potentially force the model into unlikely states that may be rarely (or never) entered by the simulation itself.

The simulator can be run for a fixed number of iterations or indefinitely and also paused and resumed at will by the user. Varying degrees of feedback can also be produced from simply outputting the state string for every state entered to giving details of the probabilistic choice made. This trace output can either be placed in a file or the standard output as selected by the user.

```
Top: for each AN in enabledANs
   if AN in set of shared actions{
      let coopG be the set of all graphs cooperating on AN
      let coopAN be all enabled instances of AN in coopG
      let statesStrings be all possible next-state strings

      for each Graph G in coopG{
         if( G contains no active instance of AN){
            set all coopAN to blocked
            remove all coopAN from enabledAN
            break Top:
         }
      }

      ST: for each state in stateStrings{
         if state contains AN.state{
            for each cAN in coopAN{
               if not (state contains cAN.state){
                  break ST:
               }
            }

            AN.stateNumber = state.stateNumber

            for each cAN in coopAN{
               coopAN.stateNumber = state.stateNumber
            }
         }
      }
   }
}
```

Figure 3.11: Final part of state-extraction process to determine states due to cooperation

4. Evaluation

Completing the implementation of any system has no bearing upon how successfully it fulfilled its intended purpose. In fact this applies to visualisation more than most areas due to its subjective nature.

The eventual visualisation operates as follows:

Setup:

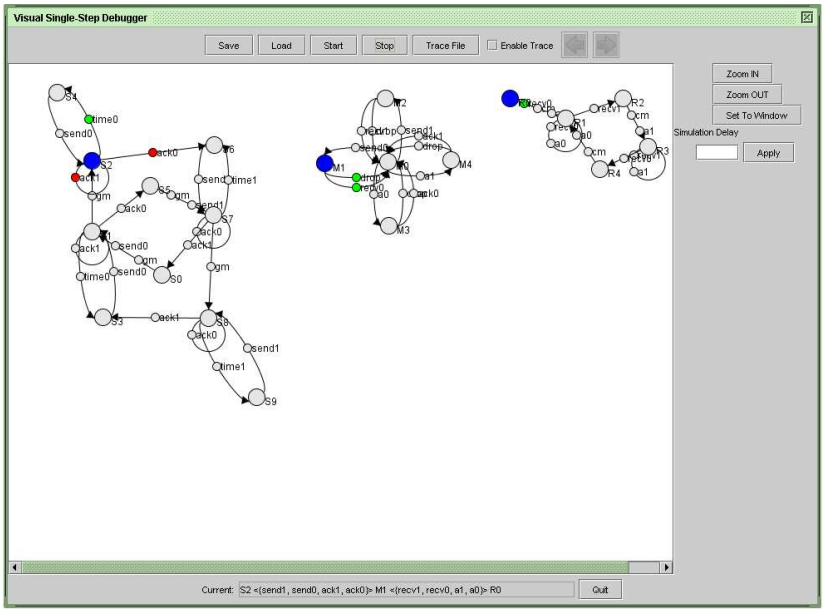
- The visual single step debugger (VSSD) can be started once a valid PEPA file is loaded into the workbench and its state-space derived.
- The VSSD extrapolates the various pieces of information required from the PEPA Workbench. It performs an initial layout of the graphs based on Force Directed Placement. Each graph is then placed on screen based on their position in the system equation.
- The state of the system is then extracted, colouring nodes appropriately (blue = current, green = next, red = blocked, grey = inactive).

Interaction:

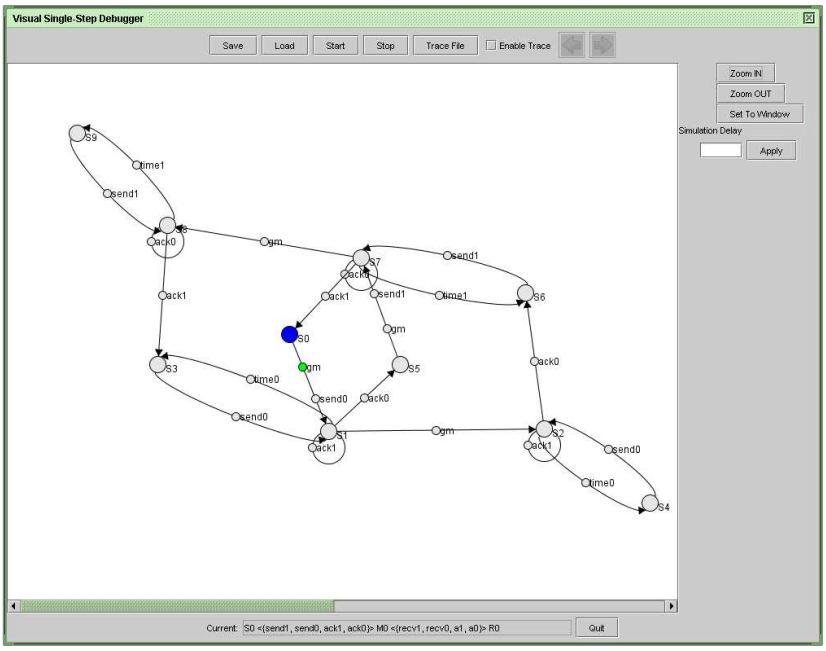
- The user may move any single major node through clicking and dragging with the right mouse button. Similarly the whole graph can be moved with the middle mouse button. The user may view the current or next state represented by any node by holding down the left mouse button over the target node. The system may be moved on by clicking any next node with the left mouse button.
- The user can step backwards and forwards through the history of actions taken by using the arrow buttons. This functions like a Web Browser history that most users will be familiar with.
- The current layout of the system may be saved at any time by clicking the "Save" button, the file is then written to an XML file selected by the user. This file can then be loaded by using the "Load" button.
- Should the layout be too small or too large the user may zoom in / out and resize the visualisation to fit the window by using the appropriate buttons (Zoom IN, Zoom Out, Set To Window)
- A trace can be written to a file by selecting "Trace File" this trace can then be enabled or disabled through the "Enable Trace" check-box.
- A simulation can be run by clicking "Start" this starts the pseudo-random SimulationRunner. The simulation can be stopped at any time by click-

ing "Stop" ("Start" will then resume the simulation). The speed of the simulation can be altered by changing the delay between iterations. This is entered in milliseconds in the "Simulation Delay" box and then can be applied. The simulation rate can be changed at any time whether the simulation is running or not.

Following are some visualisations of example PEPA files supplied with the PEPA Workbench.

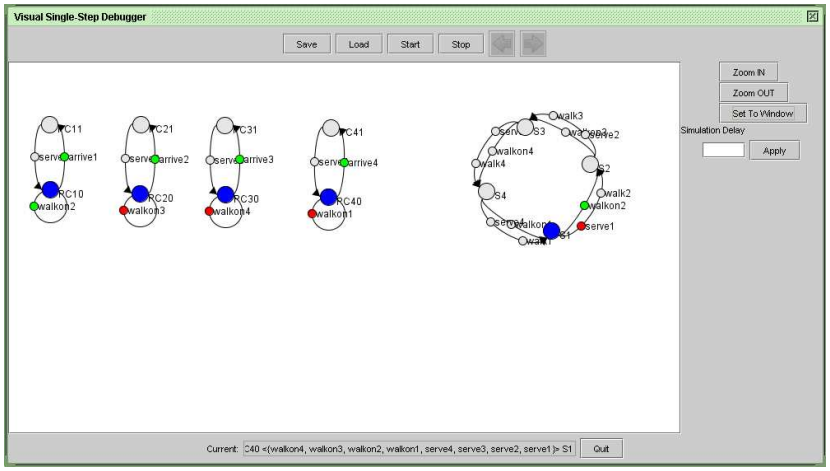


(a) Entire model

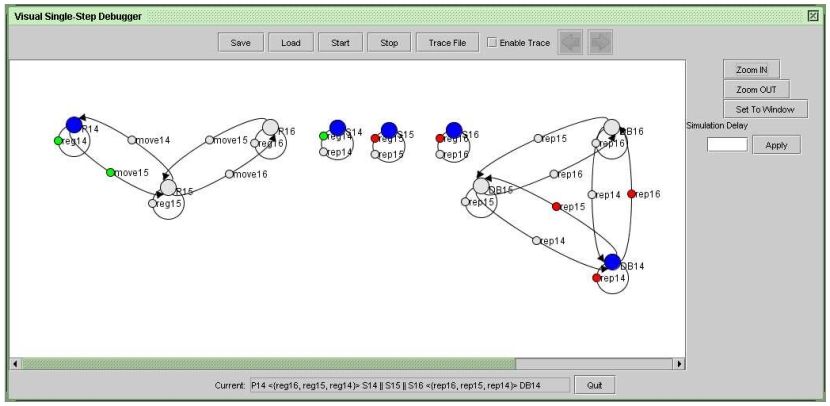


(b) Automated layout of "Sender" in Alternating Bit

Figure 4.1: Example visualisation of AlternatingBit.pepa



(a) PC-LAN4.pepa



(b) ActiveBadge.pepa

Figure 4.2: Other example visualisations

4.1 Performance

Even though much of the intensive calculation in the visualisation system takes place only once it is still desirable for it to be as efficient as possible.

4.1.1 Graph Loading

The process of loading the graphs as described previously (Section 3.1.1) consists of four distinct phases. These phases are shown below along with the order of their implementing algorithms runtime:

Component names are read in and appropriate ProcessNodes made.
One ProcesNode is created for each component so runtime is $O(N)$ where N is the number of components in the system.

Each ProcessNode extracts its definition from the Process object describing that component.
Figure 3.4: The time to extract a given edge is $O(|AN|)$ where $|AN|$ is the number of ActionNodes on the edge. This is done $|n|$ times per node whre $|n|$ is the degree of the node. This is done for every node giving a runtime of $O(N \times |n| \times |AN|)$. Given this it is possible to give a lower bound of $o(N)$. In general the run time will be bounded $O(N^2)$ as unless there are a huge number of activities a fully-connected graph would take $O(c \times N^2)$ where c is the number of edges from a given node A to B.

The Graphs are extrapolated from the ProcessNode information.
Figure 3.6: This algorithm is designed in such a way that each node is only considered once. This results in a runtime of $O(N)$.

Cooperations are taken from the Process.Coop object.
This algorithm is a simple tree iterator. The size of this tree is not determined by the number of components defined but by the number of components in the system equation. This results in a runtime of $O(|SE|)$ where $|SE|$ is the number of components in the system equation.

Firgure 4.3 shows a graph produd by running the visualisation tool on a series of models (PCLAN2, PCLAN4, PCLAN6, Alternating Bit and Active Badge) all of which are supplied with the PEPA Workbench. As the results show the processes of reading in names and finding graphs is linear in N. Also as stated above the extraction of Process details is greater than $O(N)$ however less than $O(N^2)$. It can also be seen that as expected the extraction of cooperations is not bound to N at all.

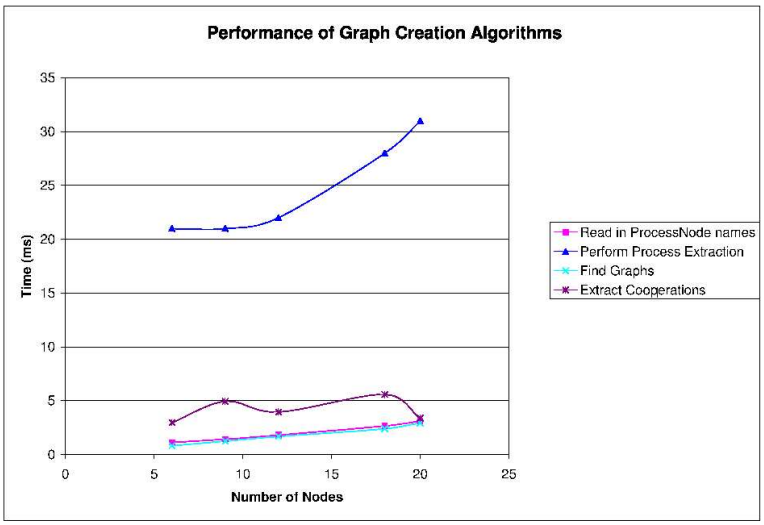


Figure 4.3: Time taken to create various graphs from Process data

4.1.2 Layout

The Force Directed Placement algorithm used (as described in Figure 3.9) is known to run in $O(N^2)$ where N is the number of major (Process) nodes in the system, however this does not give the whole story. A major determining factor in the algorithms runtime is the number of edges in the graph. Highly connected graphs will take considerably longer to layout than sparsely connected ones in real terms, although running $O(N^2)$. This can be seen in Figure 4.4.

These results were taken using cj-technologys JProfiler 2.3 [3]. It was discovered that between 30 40% of the runtime result in Figure 4.4 was attributable to the method call required to retrieve the centre-point of each node. This was somewhat surprising and would certainly be an avenue for investigation in the future should it be decided that the current Layout algorithm was unacceptably slow.

4.1.3 State Interpretation

The running time of the state extraction process is more difficult to generalise however it again can be split into several stages:

Decide on current nodes.
The implemented algorithm performs this in $O(N_{all})$ time, in this case N_{all} is all nodes, major and minor in the system.

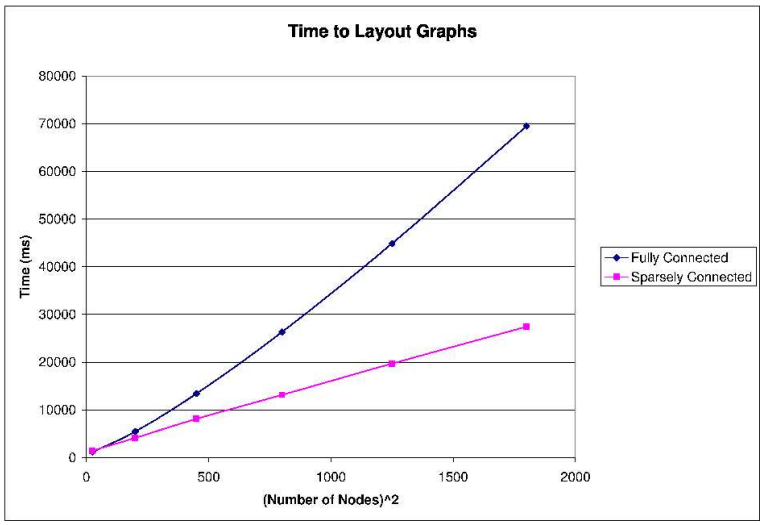


Figure 4.4: Time taken for Force Directed Placement of various graphs

Decide on possible next nodes.
The algorithm to check if a node is possibly next is implemented in $O(s \times |N_{+1}|)$ where N^{+1} is the one-step-set (as described in Section 3.2.1) of all current nodes. N_{+1} is bounded by the number of ActionNodes in the system as only an ActionNode can be next. The factor s is introduced as potentially all nodes must be checked of all possible next states.

Disable remaining nodes.
This operation takes $|N_{+1}|$ checks and (possible) assignments and thus is in performed in $O(|N_{+1}|)$ time.

Check nodes for cooperation restrictions.
For this portion of the algorithm the worst case is that every next node cooperates with every other next node. In this case the runtime is $O((|N_{+1}|)^2)$ however this can be greatly reduced as when a node a node and its set of cooperating nodes are finalised they are dropped from the list to be checked resulting in the above case completing in $O(|N_{+1}|)$ time! As it stands the runtime is bounded by $o(|N_{+1}|)$ and $O(|(N_{+1})^2|)$.

In practice it is the final section that determines the overall run time of the algorithm and not just because it has the dominating O run time. In practical terms it is the complexity of the cooperations that the component is involved in will ultimately determine how long extracting the state of the system takes. It can also be considered that the number of possible next nodes in a system is rarely large so it is unlikely that this portion of the program would become a

stumbling block.

4.2 Goals Met

At the outset of this project several key goals were introduced to decide whether or not it was a success. Users were consulted to gather their reactions to the implemented system. Their reaction to how successfully these goals were achieved follows:

The tool should be able to display PEPA models in an intuitive manner.
The PEPA models are essentially displayed as Finite State Machines, given the construction of a PEPA model this seemed like the most natural choice. Indeed after consulting with users of the system it transpired that this was the way many preferred to draw models when constructing them. Also it was found that being able to move components around and zoom / navigate the visualisation space was a well-liked feature. However it was indicated that the ability to hide or "minimise" specific components would be very desirable. In fact as with any program there are many such improvements that could be made these are discussed in section 4.4.

The tool should be easy to use.
Despite the rather limited user interface on this prototype, feedback in this area was generally good with reservations about button placement etc being the worst of the comments.

The tool should allow interaction with the model.
As discussed in various preceding sections the model is fully interactive. The state-related interaction was well-liked by all asked. In terms of layout-interaction there were two main features found lacking: It is not possible to select a sub-group of nodes form a graph to be moved at once. Also when moving the graph it is placed (significantly) below and right of the pointer rather than where the pointer is released. This was a problem that had been missed in testing due to the test system lacking a middle mouse button and was somewhat unexpected. The problem was quickly resolved however.

The tool should function with at least the standard examples supplied with the PEPA workbench.
This it does flawlessly, all can be fully visualised and simulated without problem. This allows new users to start using the PEPA Workbench "out-of-the-box" to learn about the example models included and see how PEPA models function.

PEPA models should require no modification to work with the tool.
This is, for the most part true. However there are two syntactical parts of PEPA

that require the program to be modified to visualise properly. Both of these problems can be adjusted through substitution and relabeling without altering them semantically. Again these problems and their solutions (linguistic and programmatic) are described in section 4.4.

In addition to the feedback detailed above, the project was discussed at various stages with the originators of the PEPA Workbench. The response was highly enthusiastic with a desire to include the Visual Debugger in a future release of the Workbench being expressed. To this end it was decided that work on the Visual Debugger will continue following the end of this project. This work will entail modifications detailed in the following sections and should result in an extremely useful tool to aid users of the PEPA Workbench.

4.3 Limitations

As with the development of any system when creating software it is impossible to create a system that works perfectly under every set of circumstances. This is especially true of prototype or proof-of-concept systems. Indeed this is the case for the PEPA Visual Debugger. The two (known) limitations have to do with some syntactic elements of PEPA.

4.3.1 Prefixed Choice

The first limitation is to do with a choice occurring within a series of prefixed activities as shown in Figure 4.5.

$$P \stackrel{def}{=} (a,t).((b,t).P1 + (c,t).P2)$$

Figure 4.5: Example of a prefixed choice.

When a form such as this is encountered by the ExtractProcess Algorithm (Figure 3.4) it is completely mis-represented and can result in some very unusual visualisations. However thanks to the compositionality of PEPA a syntactic solution does exist:

$$\begin{aligned} P &\stackrel{def}{=} (a,t).P_b \\ P_b &\stackrel{def}{=} (b,t).P1 + (c,t).P2 \end{aligned}$$

Figure 4.6: Example of an algebraically correct solution to Figure 4.5

Although the above is a correct solution in PEPA terms it is a great inconvenience to user to have to remove anything of this form from their PEPA models. Doing

this could render the files themselves less readable and require the construction of models in a less intuitive manner. There are however two possible programmatic solutions.

One possible solution and perhaps the simplest is to perform the above transformation automatically as the model is being interpreted by the visualiser. This would allow the visualisation to be at least one correct adaptation of the model. This introduces a non-determinism of sorts into the visualisation and is not desirable as the visualisation produced should not contradict what the user expects. This solution could also be considered a "hack" as it does reflect a truly correct interpretation of the given model.

The more desirable solution is to correctly engineer a true representation of prefixed (or sequential) choice. This should take the form of a new class of node combining properties of both ProcessNode and ActionNode. This new class (ChoiceActionNode) would display the characteristics of an ActionNode such as representing an activity pair and also in the way it represented state (see Section 3.2.1). The ChoiceActionNode would also display characteristics of a ProcessNode such as have multiple out-ward bound edges and would also be involved in layout calculations. The creation of the ChoiceActionNode is the preferred solution and should be implemented in any future work on the Visual Debugger.

4.3.2 Multiple Component Instances

The second limitation is concerned with the ability for a modeller to use multiple instance of a connected component within the system equation.

$$\begin{aligned} P &\stackrel{def}{=} (a,t).P \\ Q &\stackrel{def}{=} (a,\infty).Q \\ (P \bowtie P \bowtie P) &\underset{\{a\}}{\bowtie} Q \end{aligned}$$

Figure 4.7: Example of multiple composition of the same componenet.

In this case the visualisation created will only contain one instance of the connected component P. This is of course incorrect, again contradicting what was expected by the user.

Once again thanks to the compositionality of PEPA a (algebraically equivalent) solution exists that requires some modification of the code representing the model. In this case multiple copies of the repeated component could be made e.g. $P||P||P$ would become $P||P10||P20$. This of course goes against the parsimonious nature of PEPA and would require possibly a lot of extra work by the user. Proceeding in this way also carries the penalties to readability and intuitiveness discussed above.

$$
\begin{aligned}
P &\stackrel{def}{=} (a,t).P \\
P0 &\stackrel{def}{=} (a,t).P0 \\
P1 &\stackrel{def}{=} (a,t).P1 \\
Q &\stackrel{def}{=} (a,\infty).Q \\
(P &\bowtie P0 \bowtie P1) \underset{\{a\}}{\bowtie} Q
\end{aligned}
$$

Figure 4.8: Example of solution to Figure 4.7

The most realistic programmatic solution to this problem requires modification of the way a connected component is represented within the visualiser. Specifically it would require further abstraction between the name of a node and its identifier within the visualisation system. It is in some ways similar to the manual method described above however would be performed seamlessly to the user. The method proposed proceeds thus: For every repeated connected component create a clone of the original connected component changing the internal identifier of each sub-component in some common way e.g. a common suffix as described above. The same translations would also have to be performed to every system state examined by the state interpreter so the state $P1||P||P1 \underset{\{a\}}{\bowtie} Q$ becomes $P1||P01||P102 \underset{\{a\}}{\bowtie} Q$. This run-time translation is simple and correct due to the fact that state strings do not change order. If this translation is performed transparently to the StateInterpreter it is possible to leave it and systems relying on it unchanged. Each translation need only be performed once so could be done statically or on-demand as needed. This solution is easily implementable and would have been used in this system had time allowed.

4.4 Future Improvements

Almost all software tools have room for expansion; there are always new technologies that could be integrated and additional user requirements that could be implemented. The PEPA visual debugger is such a tool, currently only in the prototype stage there are many features that could be added to make it a more useful tool.

4.4.1 Short Term

In the short term there are several features that would enhance both the debugging and visualisation capabilities of the tool.

Unified File Format
One simple but useful extension of the current system would be to combine

layout information and a PEPA model in a single file. In this way the creator of a model can edit the code as they wish and then setup what they consider an appropriate visual layout. All of this information could then be packaged in a single XML file (for example) and given to others working with the model. Once loaded into the PEPA Workbench the visualiser would not be required to perform layout, significantly reducing start up time for the tool when working with larger models.

Further Debug Options
Many more instrumentation possibilities exist than are currently implemented. These include methods such as breakpoints: A running simulation could stop as soon a certain state is entered or node becomes activated. Also being able to view the properties of a given node may be useful such as; the number of times it has be current, next or blocked or perhaps the amount of time spent in a given state. This could in turn lead to watches being introduced where certain node or graph properties can be kept track of in some centralised view or even the trace file.

Edges Revisited
As discussed in the implementation section the concept of variable edge widths had to be abandoned in this version of the visualiser. In the future this could be revisited as this may yet be a useful method to visualise more about the system. The problem of limited usable line thicknesses could be overcome if the transition is made to another graphical toolset. It may also be desirable to display he same property in a different manner such as dashed lines or even line colour. The other problems described would have to be overcome or possibly ignored should they be deemed unimportant (froma a usability standpoint) by users of the tool.

4.4.2 3D

As discussed previously the transition of the model into a 3D graphical space has always been of interest. Due to time constraints this was not possible in this project however several interesting 3D paradigms have been considered as possible future 3D visualisations. Any work in this area would probably be best carried out in Java3D as it would allow very simple integration with the existing functionality.

There is a "simple" method which would be to make a direct translation from 2D to 3D. In such a translation nodes would become spheres and edges would be cylinders tipped with cones (to form a kind of 3D arrow). This would be simple enough as all layout techniques are still valid; the vectors used simply become one element longer.

Even though this is a simple mapping of graph elements it would present several

key benefits to the visualisation such as crossing edges only becoming view dependent. It would also potentially allow the user to visualise much larger graphs in a given screen space.

This change would of course have some draw backs such as the visualisation becoming slightly less intuitive as it is unlikely the user themselves would visualise a system in 3D space. It also complicates the navigation; requiring rotation and zoom to follow activity as nodes closer to the viewer may occlude other relevant nodes. There could also be problems associated with editing of the layout due to the interaction with a 2D view of a 3D space.

Some of these problems could be minimised if the transition to 3D is given a little more thought.

4.4.2.1 Multi-Planar Component Separation

The favoured extension to the simple 3D case has been dubbed Multi-Planar Component Separation (MPCS). The idea in the MPCS visualisation is to divide the 2D graphs (which are of course displayed on a single plane) onto separate Virtual Planes (VP). Each VP would be independently movable in the 3D space. This would allow cooperation to be expressly visualised by links and possible colouring of each plane. Especially complicated connected components could also take up more than one VP lessening the annoyance of crossing edges.

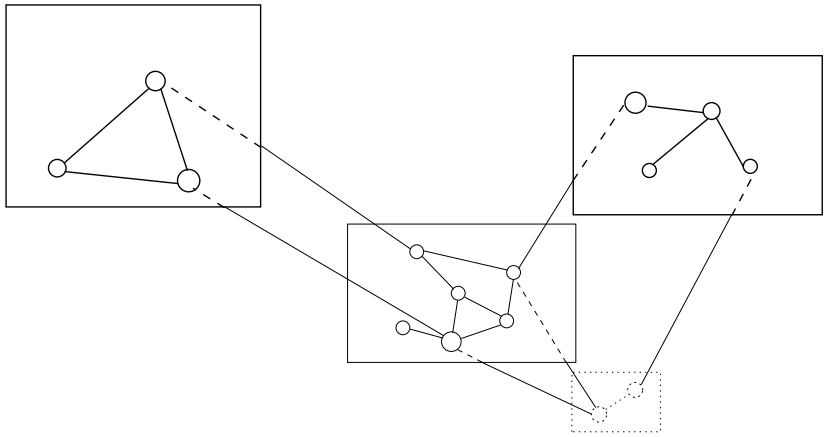


Figure 4.9: Conceptual diagram of an MPCS visualisation

This method could also be used as a form of aggregation as particular sub-graphs could be packaged into a single node then expanded onto another plane to allow close inspection of its behaviour. This feature along with the ability to "push" components that are of little interest into the background could allow a user to easily direct their attention to the specific areas of interest.

This method is attractive as it can be seen to be adding useful functionality to the tool rather than adding 3D for its own sake.

4.4.3 Visual Editing

Once the user has a tool to visualise their models it is likely that they themselves will start to visualise models in a similar fashion. It also seems natural to assume that when a user is creating models they are unlikely to do so purely in code. During talks with users it was apparent that it was a common practice to sketch out ideas before coding them.

In the current state of the system it is a simple procedure to turn the graphs produced into a string representation that matches the original PEPA file. Considering this and the users preference to sketching models rather than simply coding them, the idea of visual model editing seems like a logical next step for the system.

Although not entirely straight forward the implementation of such a system is by no means unrealistic. There are certain obvious modifications to the user interface such as a palette of components and the ability to draw edges etc Once the user is also able to modify the system equation the only other substantial feature required is the ability to merge separate connected components in some sort of sensible manner. Most likely this would be done by prompting the user to select the new representative component to be placed in the system equation.

Unfortunately due to the fact that the current simulator works on top of the existing single step debugger any modification to the models would require re-derivation to be run. This reliance on the derived state space is another possible area for improvement. Through development of the SimulationRunner it became apparent that it was not necessarily required as the simulation could explore the state space on the fly. This would greatly streamline the process of visual editing.

This addition to the PEPA workbench could potentially revolutionise the way PEPA models are worked with and used. It would especially open the door to prospective users who have no experience of Stochastic Process Algebra. This is important as some find SPA intimidating but could still use its power and flexibility for their own purposes.

5. Conclusion

The proposed goal for this project was to create a visualisation system for finite state autonoma representing a discrete event simulation. This was gradually changed to building a Visual Debugger for the PEPA Workbench through the apparent need for such a system and the authors desire to build something useful. It should be noted that in the end these goals did coincide as almost any DES can be represented in PEPA.

The projects goals can be split broadly into making a PEPA visualiser and making an interactive debugger. The first of these goals has been achieved as even taking into account the limitations discussed in 4.3 every model is still visualisable. The second goal has been achieved to a point where it functions at least the functionality of the existing text-based debugger plus some additional features (such as tracing).

As it stands the Visual Debugger is already a useful tool especially as a tool for those less familiar with PEPA. It may even enable those more expert to experiment with their models in new ways. However as mentioned the development of the tool is by no means as an end. Given the possible improvements stated and the authors own intention to continue this work the PEPA Visual Single Step Debugger looks set become a powerful tool for the analysis and manipulation of PEPA models.

Bibliography

[1] Dix, Finlay, Abowd, and Beale. *Human-Computer Interaction*. Pearson Prentice-Hall, third edition, 2004.

[2] Peter Eades. A heuristic for graph drawing. In *Congressus Numerantium*, 42, pages 149–160, 1984.

[3] ej technologies. Jprofiler. http://www.ej-technologies.com/products/jprofiler/overview.html.

[4] Silicon Graphics. Opengl graphics library. http://www.opengl.org.

[5] Valentin Haenel. The java edition of the pepa workbench 0.9.4 tobasco. http://homepages.inf.ed.ac.uk/s9905941/jPEPA/.

[6] J. Hillston. A compositional approach to performance modelling, 1996.

[7] Kitware. Visualisation toolkit VTK. http://www.kitware.com/vtk.

[8] Sun Microsystems. Java. http://java.sun.com.

[9] Sun Microsystems. Java3d. http://java.sun.com/products/java-media/3D/.

[10] N. Thomas, M. Munro, P. King, and R. Pooley. Visualisation for model comprehension. In *Proceedings of the Seventeenth UK Performance Engineering Workshop*, pages 47–58. University of Leeds, July 2000.

Appendix A. PEPA Files

A.1 PC-LAN2.pepa

```
// A PC LAN with 2 clients
// 16 States

%lambda = 2.0;
%omega = 2.0;
%mu = 2.0;

#PC10 = (arrive1,lambda).PC11 + (walkon2,infty).PC10;
#PC11 = (serve1,infty).PC10;

#PC20 = (arrive2,lambda).PC21 + (walkon1,infty).PC20;
#PC21 = (serve2,infty).PC20;

#S1 = (walkon2,omega).S2 + (serve1,mu).(walk2,omega).S2;
#S2 = (walkon1,omega).S1 + (serve2,mu).(walk1,omega).S1;

(PC10 <> PC20 ) <walkon1,walkon2, serve1,serve2> S1
```

A.2 PC-LAN6.pepa

```
// A PC LAN with 6 clients
// 768 States

%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 + (walkon5,infty).PC40;
#PC41 = (serve4,infty).PC40;

#PC50 = (arrive,lambda).PC51 + (walkon6,infty).PC50;
#PC51 = (serve5,infty).PC50;

#PC60 = (arrive,lambda).PC61 + (walkon1,infty).PC60;
#PC61 = (serve6,infty).PC60;

#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 = (walkon5,omega).S5 + (serve4,mu).(walk5,omega).S5;
#S5 = (walkon6,omega).S6 + (serve5,mu).(walk6,omega).S6;
#S6 = (walkon1,omega).S1 + (serve6,mu).(walk1,omega).S1;

(PC10 <> PC20 <> PC30 <> PC40 <> PC50 <> PC60)
        <walkon1,walkon2,walkon3,walkon4,walkon5,walkon6,
                 serve1,serve2,serve3,serve4,serve5,serve6> S1
```

A.3 AlternatingBit.pepa

```
// The Alternating Bit Protocol modelled by James Edwards in
// the paper "Process Algebras for Protocol Validation and Analysis",
// Proceedings of PREP 2001
// should have 94 states

// an example file for use with the Java Edition of the PEPA Workbench 0.9 and beyond

def = 1.0;
retry=1.0;
loss = 1.0;

S0 = (gm, def).(send0, def).S1;
S1 = (gm, def).S2 + (time0, retry).S3 + (ack0, infty).S5 + (ack1, infty).S1;
S2 = (time0, retry).S4 + (ack0, infty).S6 + (ack1, infty).S2;
S3 = (send0, def).S1;
S4 = (send0, def).S2;
S5 = (gm, def).(send1, def).S7;
S6 = (send1, def).S7;
S7 = (gm, def).S8 + (time1, retry).S6 + (ack0, infty).S7 + (ack1, infty).S0;
S8 = (time1, retry).S9 + (ack0, infty).S8 + (ack1, infty).S3;
S9 = (send1, def).S8;

R0 = (recv0, infty).(cm, def).(a0, def).R1;
R1 = (recv0, infty).(a0, def).R1 + (recv1, infty).R2;
R2 = (cm, def).(a1, def).R3;
R3 = (recv1, infty).(a1, def).R3 + (recv0, infty).R4;
R4 = (cm, def).(a0, def).R1;

M0 = (send0, infty).M1 + (send1, infty).M2 + (a0, infty).M3 + (a1, infty).M4;
M1 = (drop, loss).M0 + (recv0, def).M0;
M2 = (drop, loss).M0 + (recv1, def).M0;
M3 = (drop, loss).M0 + (ack0, def).M0;
M4 = (drop, loss).M0 + (ack1, def).M0;

 (S0 <send0, send1, ack0, ack1> M0) <recv0, recv1, a0, a1> R0
```

A.4 ActiveBadge.pepa

```
// A small model (72 states) of a location tracking system based on
// active badges made by Stephen Gilmore, Jane Hillston and
// Graham Clark.  The paper ‘‘Specifying performance measures for
// PEPA’’ appeared in the proceedings of Fifth International AMAST
// Workshop on Real-Time and Probabilistic Systems, Bamberg 1999.
// The proceedings were published as Springer-Verlag LNCS 1601.

// an example file for use with the Java Edition of the PEPA Workbench 0.9 and beyo

r = 1.0;
m=1.0;
s=1.0;

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

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

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

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