Process Algebras for Protocol Validation and Analysis

James Edwards[*]
edwards@cs.bris.ac.uk
http://www.cs.bris.ac.uk/∼edwards/

January 31, 2001

Abstract

Process algebras are a set of mathematically rigous languages with well defined operational semantics that allow for proofs of systems and propositions, either in terms of drawing an isomorphism between two systems such that they can be said to be equivalent, or in terms of deriving a system from its base terms. Process algebras CCS and PEPA are presented as modelling languages for concurrent systems and protocols, and are given a formal definition. PEPA is presented with a worked example of the alternating bit protocol, and it is demonstrated how equivalence relations may be drawn to prove functional and behavioural properties of this protocol, and how mathematically sound performance analysis can be applied using tools associated with this language. The methodology presented here is a generic one, and could easily be applied to other systems and may be considered a framework for modelling, proving and evaluating performance of other network protocols, as well as any concurrent and cooperating system.

1 Introduction

This paper presents a formal methodology for describing and testing communications protocols. It gives an example of the alternating bit protocol and works through the stages of modelling it using a process algebra. It then demonstrates how it is analysed to yield protocol correctness and performance information. Its purpose is to raise awareness of formal methods as a tool for supporting protocol design, verification and analysis.

This paper is divided into the following sections :: Section 2 gives a brief introduction to process algebras, including the background theory and sample applications. Section 3 gives a brief introduction to the alternating protocol, and shows how it can be modelled using a process algebra. Section 4 describes the analytical processes that shows that the alternating bit protocol is correct, and gives some insight on how performance data can be derived from the model. Section 5 gives some conclusions that may be drawn from this and some general remarks.

2 Process Algebras

2.1 Definitions

What are process algebras ? They are best described as a set of formalisms for modelling systems that allow for mechanical mathematical reasoning with respect to a set of desired properties, be it equivalence, absence of deadlocking or some safety property. It is usually the case that process algebras are used to model concurrent systems and communicating systems.

Although I speak of a set of formalisms, they all share some features[1] ::

[*]Department of Computer Science, Woodland Road, Bristol BS8 1UB

[1]In the same way that different flavours of UNIX all share certain features

- A formal language (an algebra), that provides the constructs for system descriptions
- Equivalence relations, describing how to equate systems at one or more levels
- A set of axioms, describing the ground terms which allow for the proving of validity of complex systems

Some process algebras introduce their own additional features. Two process algebras will be discussed here - CCS (The Calculus of Communicating Systems), and PEPA (Performance Evaluating Process Algebra). Many process algebras exist which are not described here, including CSP, LOTOS, TIPP, IMC and many others, but generally speaking, the principles are the same throughout.

2.2 Calculus of Communicating Systems

The Calculus of Communicating Systems (hereafter referred to as CCS) was one of the first process algebras, and was created by Milner [2]. CCS agents are built up from the algebras most simple element, actions, which are defined as an atomic[2] step of execution. Some of these actions refer to internal computational steps, some to interactions with other agents.

2.2.1 Actions

To give a definition of the set of actions, let there be a [countably] infinite set of port labels L ::

$Set\ of\ Labels\ \{L\}$

Now let the set of actions be the union between this set L, and the internal computation action τ. Note that for communication and synchronisation, there is also the addition to the alphabet of complimentary port labels, $\{\overline{\alpha}\ \mid \alpha \in L\}$ where these labels exist in L ::

$Set\ of\ Actions\ \ Act\ =\ L\ \cup\ \{\overline{a}\mid\ \ a \in L\}\ \cup\{\tau\}$

These two form a pair of input and output ports – a channel. Because CCS is a binary synchronous modelling language, there can only be one input and one output port with the same name that are involved in the same synchronisation (that is, there can be more than one input and output ports with the same name, but only one of each can be allowed to be involved in a single synchronisation). Other process algebras allow for more than two agents to be involved in a synchronisation, CSP [1] for example. Note that the internal computation actions, τ actions, are not visible to the outside world. From this it is derivable that the set of visible actions is equal to the set of actions less $\{\tau\}$.

$Set\ of\ Visible\ Actions\ \ Act \setminus \{\tau\}$

2.2.2 Agents

Agents are composed from the sets of actions that have just defined through a set of operators whose definitions follow. Formally ::

$A \overset{def}{=} nil \mid\mid a.A \mid\mid A + A \mid\mid A \mid A \mid\mid A \setminus \{x\} \mid\mid A\backslash_{[f]}$

Where each of the operators is of type $Agent \times Agent \rightarrow Agent$, with the exception of the . operator, which is of type $Action \times Agent \rightarrow Agent$. The distinguished symbol nil with represents inaction (i.e. this agent is unable to perform any action).

[2]i.e. indivisible

2.2.3 Operators

As well as sets of actions, there are additional symbols that form the operators for the language. Operators permit operations such as composition of agents to form more complex systems, choice between two or more agents and the like. The main six operators are ::

- . This symbol allows for prefixing. This allows more than one action to form a behavioural definition of an agent, i.e. $a.Rest$ describes an agent that can do some action a, and then behaves like $Rest$, where rest is an agent. Note also that . is of type $Action \times Agents \rightarrow Agent$. You can think of it as a constructor which builds an agent out of some existing agent, and an action.
- $+$ Choice. This symbol allows for a non-deterministic choice to be made between two or more agents.
- $|$ Composition. This symbol denotes the composition of two or more agents to form a more complex system.
- $A \setminus X$ Restriction. This operator hides all symbols (including complements, where they exist) in X from the agent definition A, where X is a subset of the labels ($X \subseteq L$).
- $A\backslash_{[new/old]}$ Relabelling. Effectively replaces all occurrences of old with new in the agent A. This is where old and new are labels. Relabelling is a function of type $action \rightarrow action$.
- $\stackrel{def}{=}$ Definition. Used to define whole agents or processes. e.g. $A \stackrel{def}{=} action_1.agent_1 + action_2.agent_2$.

These operators form the glue that allows for the construction of agent and system definitions. The convention used here (for clarity) is that actions will start with a lower case letter, and actions will start with an upper case letter.

2.2.4 Semantics

Unsurprisingly, each of the operators described have associated operational semantic rules. These rules describe how a system can behave. The rule triples are of the form ::

$$\frac{premises}{conclusions}$$

So long all the premises are satisfied (above the line) then it is valid to draw the given conclusions (below the line). These rules show how the rules look like, and how they transform a system definition in a step-by-step fashion.

The nil agent has no premises and no conclusions. It is capable of performing no action. Because no state transformations are available, no conclusions may be drawn from it, and there are no conditions for drawing no conclusions !

Prefixing has a rule which looks like ::

$$\frac{}{a.Rest \stackrel{a}{\rightarrow} Rest}$$

Which says that an agent looking like $a.Rest$ may, upon performing action a, behave like $Rest$.

The rule for choice is ::

$$\frac{A_j \stackrel{a}{\rightarrow} A_j^{'}}{\sum_{i \in I} A_i \stackrel{a}{\rightarrow} A_j^{'}} \qquad (j \in I \text{ where } I \text{ is an indexing set})$$

Which says that from a set of choices defined in some indexing set I, a choice is made between them by performing some action. Note that one implication of this rule is that if more than one choice happens to start with the same action, the choice between them is made non-deterministically, and all other choices are discarded. That is, by performing one action you are committed to that choice, and there is no back-tracking. Note also that you could think of the nil operator as a choice with an empty indexing set.

The rules for composition are a little more complicated, but only because there are three of them ::

$$\frac{A \xrightarrow{a} A'}{A \mid B \xrightarrow{a} A' \mid B}$$

$$\frac{B \xrightarrow{b} B'}{A \mid B \xrightarrow{b} A \mid B'}$$

$$\frac{A \xrightarrow{c} A' \qquad B \xrightarrow{\bar{c}} B'}{A \mid B \xrightarrow{\tau} A' \mid B'}$$

The first rule says that given a choice between A and B performing some action, if A has an action that it may perform, it may do so. The second rule says the same for B. This may happen irrespective of whether other agents have valid actions. The third rule says that if two agents A and B are synchronising or communicating via a mutual channel (in this case c), then they must do so atomically, with both agents performing an action synchronously in one indivisible step (τ). It is this rule that restricts CCS to binary synchronisations, since $\bar{\tau}$ is not defined.

The restriction rule is ::

$$\frac{A \xrightarrow{a} A'}{A \setminus L \xrightarrow{a} A' \setminus L} \qquad \text{where } a \notin L$$

This rule only allows actions (including complementary actions) not in the restricting set L to be visibally performed. This internalises actions existing in L such that they are not visible to the environment. This also has the effect that the environment may not interact with actions appearing in the set L.

Relabelling also has one rule ::

$$\frac{A \xrightarrow{a} A'}{A[f] \xrightarrow{a} A'[f]} \qquad \text{where } f \text{ is a relabelling function of form } new \setminus old \text{ and of type } action \rightarrow action$$

The implication of this rule is that the same set of actions are available, yielding the same quantum of possible transitions, but the transitions are now known by the new label as defined in f.

The last operator to be semantically described is the definition operator ::

$$\frac{A \xrightarrow{a} A'}{B \xrightarrow{a} A'} \qquad \text{where } B \overset{def}{=} A$$

All this means is an agent (or process) B behaves like it's definition body A, so long as B is defined in the proper form, $B \overset{def}{=} A$. This concludes the operational semantics for CCS.

2.2.5 Equivalence Relations

Equivalence relations are useful when trying to prove two systems to be equivalent. They are reflexive, transitive and symmetric relations such that if it is required that for any context two systems behave in the same way, then these are congruences. Because of this, any two congruent systems may substitute each other without changing the operational behaviour of the system.

There are three main equivalence relations defined for CCS (and most process algebras) - strong bisimulation relations ($\sim$), weak bisimulation relations ($\approx$) and observational congruence relations ($\approx^c$).

The basic idea of a strong bisimulation relation is that two systems should be able to pass through equivalent states, such that one can simulate the others' behaviour at all times. The relation is defined as ::

Relation $R \subseteq E \times E$ where $(P, Q) \in R$ and the following hold $\forall$ $actions\ a \in Act$:

1. If $P \xrightarrow{a} P'$ then $\exists\, Q',\ Q \xrightarrow{a} Q'$ and $(P', Q') \in R$ (i.e. $P' \sim Q'$)
2. If $Q \xrightarrow{a} Q'$ then $\exists\, P',\ P \xrightarrow{a} P'$ and $(P', Q') \in R$

Where E refers to a set of agent expressions

Note that the second rule is symmetric to the first. What this says about the relation is that there must be a mapping between states such that each state can be related to one state in the other system.

The problem with strong bisimulation as defined above is that it is too inflexible with respect to internal computations (τ actions), in so far as one τ in one system that is not in the other will make the systems inequivalent. To solve this, another bisimulation relation is introduced - a weak bisimulation relation. The definition is similar, although subtly different ::

Relation $R \subseteq E \times E$ where $(P, Q) \in R$ and the following hold $\forall$ $actions\ a \in Act$:
1. If $P \xrightarrow{a} P'$ then $\exists\, Q',\ Q \overset{\hat{a}}{\Rightarrow} Q'$ and $(P', Q') \in R$ (i.e. $P' \approx Q'$)
2. If $Q \xrightarrow{a} Q'$ then $\exists\, P',\ P \overset{\hat{a}}{\Rightarrow} P'$ and $(P', Q') \in R$

Where E refers to a set of agent expressions

The difference is that the ˆ (hat) operator allows for any number of τs to be removed before or after some action a. Similarly note that instead of a $\rightarrow$, there is now a $\Rightarrow$. $\Rightarrow$ allows the insertion of any number of τ actions before or after some action a. $\Rightarrow$ if different in so far as $\rightarrow$ needs to precisely specify the τ actions, whereas $\Rightarrow$ needs to at least specify the τ actions. Note that a $\Rightarrow$ with a hat specifies nothing about the τ actions at all, since they at least need to be specified and additional τs may be prefixed, suffixed and removed, according to needs.

There is a problem with weak bisimular relations - they do not form a congruence relation. Strongly bisimular relations do, however, but they are too strict. The reason that weak bisimulation relations are not congruences are owing to a subtle feature of combining choice and initial τ actions. This means that even though they are weakly bimsimular, substituting one with the other will not result in equivalent behaviour. Consider ::

$A \overset{def}{=} \tau.a.nil + b.nil$
$B \overset{def}{=} a.nil + b.nil$

The τ action from A is matched by an empty transition from B. However, by doing this transition in A, the system is already committed to performing $a.nil$, which is not the same as having a choice between $a.nil$ and $b.nil$. Without the choice between two actions, $\tau.a.nil$ is behaviourally equivalent to $a.nil$, but because of the semantics of the choice operator, introducing another element to the indexing set forces behavioural inequality. Therefore, it has been shown that weak bisimulation relations are not congruence relations. To solve this, we introduce our third type of bisimulation relation, observational congruence. This is defined as ::

Relation $R \subseteq E \times E$ where $(P, Q) \in R$ and the following hold $\forall$ $actions\ a \in Act$:
1. If $P \xrightarrow{a} P'$ then $\exists\, Q',\ Q \overset{a}{\Rightarrow} Q'$ and $P' \approx Q'$
2. If $Q \xrightarrow{a} Q'$ then $\exists\, P',\ P \overset{a}{\Rightarrow} P'$ and $P' \approx Q'$

Where E refers to a set of agent expressions

Which prevents initial τ actions being matched to an empty transition. In our example, the $\xrightarrow{\tau}$ transition would have to be matched by a $\overset{\tau}{\Rightarrow}$ transition (i.e. at least one τ). This is the useful mid-point between strong and weak bisimulation relations that is a congruence.

Forming these relations is an iterative process, starting with the two systems initial states, and for each transition that each system can make, relating the resulting state to another state in the other system. If such a relation can be found then the two systems can be said to be (strongly or weakly) bisimularly equivalent, or observationally congruent.

2.2.6 Examples

For an example, let us model an unreliable communication medium. It accepts a message, and tries to send it. This message can be internally discarded, or it can be given to the output port which performs the sending. From there it returns to it's starting state to perform another action. Suppose that the model is ::

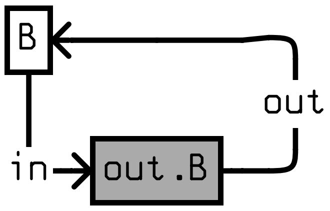

Figure 1: State diagram for our reliable medium B

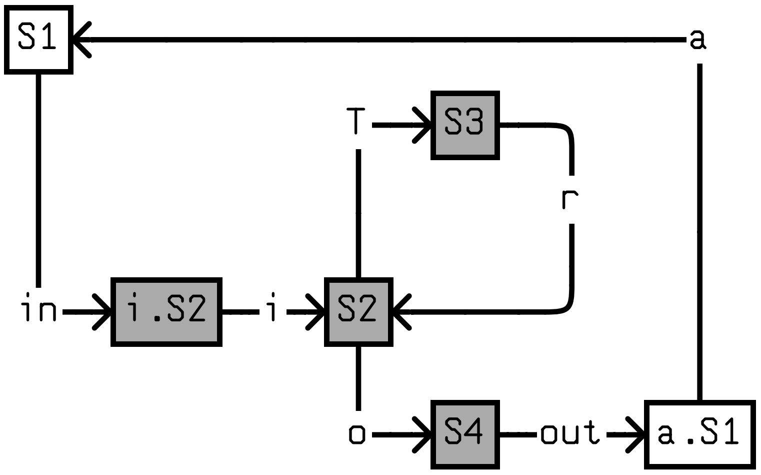

Figure 2: State diagram for our unreliable medium S1

$$\begin{array}{lcl}
S_1 & \overset{def}{=} & in.i.S_2 \\
S_2 & \overset{def}{=} & o.S4 + \tau.S_3 \\
S_3 & \overset{def}{=} & r.S_2 \\
S_4 & \overset{def}{=} & \overline{out}.a.S_1
\end{array}$$

From the inference rules, one can deduce the fact that $S_1 \overset{in}{\rightarrow} i.S_2$, and the like. Once the internal actions have been abstracted from (i.e. hidden), one can prove that this system is observationally congruent to a reliable communication medium defined, defined below. Ideally, one would be able to form an observational congruence relation between $S_1 \backslash_{[i,o,r,a]}$ and B. Note that for clarity, each of the internalised actions are subscripted with their original action.

$$B \overset{def}{=} in.\overline{out}.B$$

The state diagrams for these two systems are given in figures 1 and 2. First start by equating B and S_1, since these are the starting states (i.e. add them to the relation) ::

$$R = \{(B, S_1)\}$$

B can make a transition, but S_1 is unable to do the same transition at this time. Because this is not a strongly bisimularity relation, we are able to perform any number of non-leading τ actions, of which S_1 has one available (since the i action is now an internal τ action). Therefore, add this to the relation ::

$$R = \{(B, S_1), (\overline{out}.B, \tau_i.S_2)\}$$

Similarly, add the next logical relation to the set, since there is only one possible action to be performed from both elements in the set that are not already present (note, we consider all elements in the set because this is a transitive relation) ::

$$R = \{(B, S_1), (\overline{out}.B, \tau_i.S_2), (\overline{out}.B, S_2)\}$$

From here, there are two internal actions to choose from. Therefore, select one and add it. For arguments sake, we will choose the internal action leading to S_3 ::

$$R = \{(B, S_1), (\overline{out}.B, \tau_i.S_2), (\overline{out}.B, S_2), (\overline{out}.B, S_3)\}$$

Again, one is seemingly left with two choices. This is not the case, as the internalised r action loops back to S_2, and this pair is already in the relation. Therefore, there is only one action that can be performed from all elements in the relation, which is the hidden o action taking us to S_4 ::

$$R = \{(B, S_1), (\overline{out}.B, \tau_i.S_2), (\overline{out}.B, S_2), (\overline{out}.B, S_3), (\overline{out}.B, S_4)\}$$

Again, there is only one choice to be made, but this time the other system is able to agree on the action. Hence this is added to the relation ::

$$R = \{(B, S_1), (\overline{out}.B, \tau_i.S_2), (\overline{out}.B, S_2), (\overline{out}.B, S_3), (\overline{out}.B, S_4), (B, \tau_a.S_1)\}$$

From here, there are no more actions that will lead to a new state, since performing the internalised a action will lead back to a state already in our relation. Notice that all states in both systems are accounted for, one could say that the two are observationally congruent. Figures 1 and 2 show this mapping pictorially; the shaded state in figure 1 maps to the shaded states in figure 2, and likewise for the non-shaded states.

2.3 Performance Evaluating Process Algebra

Performance Evaluating Process Algebra (hereafter known as PEPA), is similar to CCS in many respects. It shares a similar syntax, and includes many of the same operators. The only minor difference is the absence of relabelling. The operational semantics are also very similar.

However, PEPA has one important addition, rates for the actions. PEPA is one of many process algebras falling into the class of *Stochastic Process Algebras*. This means that each action has an stochastic delay associated with it that provides some information relating to how quickly the action is performed. PEPA also permits multi-way synchronisation (i.e. it is not bound to the binary model of CCS).

The noticeable syntactical differences include a different symbol for composition. Instead of $|$, we now have $\bowtie$, subscribed with a set of interacting ports (not shown here). Also, each action that was of the form a now looks like (a, R), where R is the rate for action a.

2.3.1 Rates

The basic idea of a rate is to specify the stochastic timing properties of an action. What this translates into is a probabilistic varying delay associated with each action.

In PEPA, this is modelled as a negative exponential distribution, the curve for which is shown in figure 3. If you take the cumulative form of this, as time increases, the probability also increases. The rate of this increase is given by the factor λ. Smaller numbers, therefore, cause a slower growth in probability, while larger numbers cause the probability growth to be faster. By controlling the value of λ, you can control how long an action will (probabilistically) take to happen.

This is the basis of PEPA and the foundation upon which the performance analysis is built.

2.3.2 Performance Evaluation

Without getting into too much technical detail, we can use this rate information to calculate performance for a modelled system (that is, how long the system stays in each state and performing each action in terms of mean time). This usually takes the form of putting a model through a tool such as PWB [9], and feeding this output into a mathematical

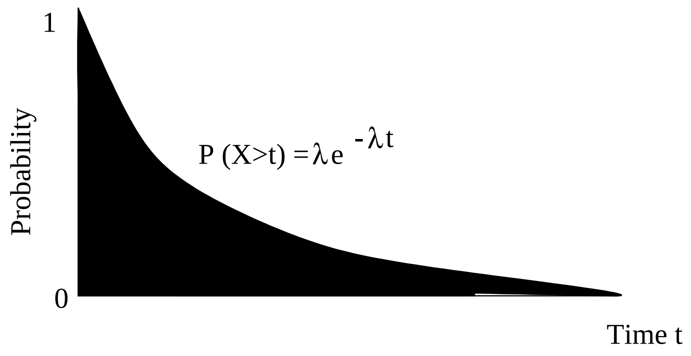

Figure 3: Negative Exponential Distribution Curve. Cumulative form shows that as time increases the event x becomes more probable.

tool like Maple. Such tools can solve a steady-state probabilistic solution, from which some analysis can be derived. A small worked example will be presented in the next section.

3 The Alternating Bit Protocol

3.1 About the Protocol

The alternating bit protocol (hereafter ABP) is a well known protocol for communication between two parties over an unreliable lossy medium (e.g. Ethernet). Briefly, each message that is sent over the medium has a single bit added, either a 1 or a 0. This bit can be used to determine if this message is a duplicate (perhaps re-sent after an acknowledgement was not received after a pre-determined amount of time). By the same token, this also allows the sender to determine if an acknowledgement just received is a duplicate (imagine the case where one message was sent, but the acknowledgement was delayed. The sender re-sends the message, and another acknowledgement was sent. Acknowledgement one arrives, so the next message is sent. If the delayed acknowledgement is received, the sender may otherwise assume that the message it just sent was received).

In this protocol, the loss of whole messages is possible. Corruption is possible over Ethernet, but this does not form part of the protocol (the receiver can always drop a corrupted message if it is determined to be corrupted by some higher level protocol), and neither does reordering of messages. This should be reflected in the model.

3.2 Modelling the ABP

One of the features of most process algebras is the facility to compose systems from specifications of individual agents. As we will see, this is a useful feature. Within the ABP there are three distinct agents to consider - the sender, the receiver and the unreliable medium. Therefore, the modelling task should consider each of these in turn and then put them all together to form the whole protocol.

In order to facilitate the non-deterministic choice for the medium, we will model this protocol using PEPA.

3.2.1 The Sender

It is presumed that the sending part of the protocol will sit between some higher level application (e.g. sendmail, netscape, ...) and the physical medium. Therefore, in real life, the first thing that this part of the protocol would have to deal with is waiting for a message to be given to it for transmission. However, to simplify things, we will simulate this with a probabilistic delay. Let us assume that a model of the sender looks like ::

$$\begin{array}{lll}
S_0 & \overset{def}{=} & (gm, def).(send_0, def).S_1 \\
S_1 & \overset{def}{=} & (gm, def).S_2 + (time_0, retry).S_3 + (ack_0, \top).S_5 + (ack_1, \top).S_1 \\
S_2 & \overset{def}{=} & (time_0, retry).S_4 + (ack_0, \top).S_6 + (ack_1, \top).S_2 \\
S_3 & \overset{def}{=} & (send_0, def).S_1 \\
S_4 & \overset{def}{=} & (send_0, def).S_2 \\
S_5 & \overset{def}{=} & (gm, def).(send_1, def).S_7 \\
S_6 & \overset{def}{=} & (send_1, def).S_7 \\
S_7 & \overset{def}{=} & (gm, def).S_8 + (time_1, retry).S_6 + (ack_0, \top).S_7 + (ack_1, \top).S_0 \\
S_8 & \overset{def}{=} & (time_1, retry).S_9 + (ack_0, \top).S_8 + (ack_1, \top).S_3 \\
S_9 & \overset{def}{=} & (send_1, def).S_8
\end{array}$$

The notation used here has not yet been fully explained. $\top$ is the reserved rate for a synchronising action, analogous to the input ports in a CCS model. They are not allowed to internally decide to happen by themselves. def and $retry$ are variables for rates that will be decided upon later; but they will be real numbers for the λs when the performance analysis is done.

S_0 is the starting state, where we are waiting for some message to arrive for transmission. Once this happens, we send it and move to S_1.

S_1 is the state where the sender is waiting for an acknowledgement of $send_0$. The sender may receive another message for transmission, in which case, the system moves to S_2 and effectively queue it for sending later (there cannot be two transmissions in flight at the same time). The acknowledgement may time out, in which case the system moves to S_3, which re-sends it and comes back to S_1. The sending agent may get an acknowledgement for $send_1$, which the sender ignores it and loops back to S_1. When the sender receive the acknowledgement that it is waiting for, the system moves to S_5.

S_2 is the state which does exactly the same as S_1 does, except with another message queued for sending, and making it impossible for another message to be added to the queue. S_3 and S_4 have already been explained, and could be in-lined to the definitions of S_1 and S_2, and are only separate for clarity.

S_5 is the state where the system has sent and acknowledged $send_0$, and are waiting for another message to arrive. This is analogous to S_0. S_6 is the case where the system has sent and acknowledged the transmission of $send_0$, but has another message to send. Therefore, it is sent, and the system moves to S_7. This is similar to S_3.

S_7 is like S_1, but it is defined for $\{time_1, ack_0, ack_1\}$ in a slightly different (but symmetric) fashion. In the same way, S_8 is symmetric to S_2, and S_9 is similar to S_4. This concludes the modelling of the sending agent.

3.2.2 The Receiver

The receiving agent, as you would expect, is similar to the sending agent. It is simplified in so far as there are no timeouts. Rather than accepting messages for transmission, it simply pushes the messages up-stream to the higher level application (e.g. sendmail, apache, ...). Let's look at the specification ::

$$\begin{array}{lll}
R_0 & \overset{def}{=} & (recv_0, \top).(cm, def).(a_0, def).R_1 \\
R_1 & \overset{def}{=} & (recv_0, \top).(a_0, def).R_1 + (recv_1, \top).R_2 \\
R_2 & \overset{def}{=} & (cm, def).(a_1, def).R_3 \\
R_3 & \overset{def}{=} & (recv_1, \top).(a_1, def).R_3 + (recv_0, \top).R_4 \\
R_4 & \overset{def}{=} & (cm, def).(a_0, def).R_1
\end{array}$$

R_0 is the starting state. In this state, the system is waiting to receive $recv0$. Upon reception, it consumes it and sends an acknowledgement. From here, the system moves to R_1.

R_1 either receives the same message again, in which case, the assumption is made that the acknowledgement was not received, and another is sent. The system then loops back to R_1. If the system receives $recv_1$, then it goes to R_2.

R_2 consumes the message, acknowledges it, and goes to R_3. R_3 is symmetric to R_1, as R_4 is to R_3. This finishes the receiving agent specification.

3.2.3 The Medium

The unreliable medium is the last agent that requires definition. This agent will form the connection between the sending and the receiving agents. One can infer from the definitions of these agents that this agent must have ports labelled $\{send_0, send_1, ack_0, ack_1, a_0, a_1, recv_0, recv_1\}$. Also remember that this agent has to non-deterministically loose messages. Suppose that the specification is ::

$$\begin{array}{lcl}
M_0 & \overset{def}{=} & (send_0, \top).M_1 + (send_1, \top).M_2 + (a_0, \top).M_3 + (a_1, \top).M_4 \\
M_1 & \overset{def}{=} & (drop, loss).M_0 + (recv_0, def).M_0 \\
M_2 & \overset{def}{=} & (drop, loss).M_0 + (recv_1, def).M_0 \\
M_3 & \overset{def}{=} & (drop, loss).M_0 + (ack_0, def).M_0 \\
M_4 & \overset{def}{=} & (drop, loss).M_0 + (ack_1, def).M_0
\end{array}$$

Here, the agent is defined as a transport conduit for four channels, $\{send_0, send_1, a_0, a_1\}$. Upon receiving a message on any of these four channels (in state M_0), it moves to another state where it non-deterministically chooses to drop the message and return to M_0, or it routes it to the appropriate output ($\{recv_0, recv_1, ack_0, ack_1\}$ respectively), and then loops back to M_0.

Obviously, there is a race condition between *drop* and any of the outputs to see whether a message is sent or discarded. It is this which makes the medium unreliable.

3.2.4 Putting It Together

Having defined the agents involved in the ABP, all that remains (for the modelling task) is to bring them together in order to form the larger protocol. This is done with two compositions ::

$$\begin{array}{l}
(S0 \underset{A}{\bowtie} M0) \underset{B}{\bowtie} R0 \\
\quad A \overset{def}{=} \{send_0, send_1, ack_0, ack_1\} \\
\quad B \overset{def}{=} \{recv_0, recv_1, a_0, a_1\}
\end{array}$$

This defines the ABP to be a composition of S and M, interacting on channels $\{send_0, send_1, ack_0, ack_1\}$, and a composition of M and R interacting on channels $\{recv_0, recv_1, a_0, a_1\}$.

For clarity, the whole specification is given in appendix A.

4 Model Analysis

Tools exist to aid the analysis of PEPA models. Examples include PEPA tools [7], PWB [9] and PSF [8], as well as mathematical tools such as Maple [4] and Matlab [6]. However, to examine some properties, some things still have to be done by hand (at the present).

4.1 Automated Analysis

Generally, automated analysis can show us whether a system is free from deadlock, livelock, active-active synchronisations (this is specific to PEPA), and can also show us information relating to the performance of the system (this is specific to stochastic process algebras).

Putting the model through the PWB tells us ::

```
PEPA Workbench Version 0.72 [22-12-1998]
Filename: abp.pepa
Compiling the model
Generating the derivation graph
The model has 94 states
The model has 196 transitions
Writing the hash table file to abp.hash
Exiting PEPA Workbench.
```

This shows that there are no-states which have no possible actions (i.e. there aren't any trivially deadlocking states) and that there are no active-active synchronisations, no zero-rate actions, and no infinite-rate actions. It also gives an indication of the state space in hand, 94 and 196 transitions in this case.

One could take the hash table that it outputs to disk and put it into a mathematical tool like Maple. This can be used to solve steady state solutions to the underlying Markov chain, and can be used to yield some performance indications (this will be shown in the following two sections). The entire Maple session script, with output removed, is given in appendix B.

4.2 State Based Analysis

This type of analysis is still fairly automated; plotting the solutions to the Markov chains (Markov chains form the underlying mathematical structure of a PEPA model) with varying rates for dropping a message ($loss$) and timeouts ($retry$) makes a graph which is given in figure 4. What this graph shows is how much (as a percentage of total time) each state exists for, relative to the variable stochastic rates. It is clear that there is at least one state (toward the top of the graph) that is existing for a lot of the time. This individual state is plotted in figure 5. Tracking down this state using Maple reveals it to be state 23 (this comes from printing out the array of steady state probabilities and noticing that it is state number 23 that has the [relatively] high number around 0.4), which is $S_4 \mid M_0 \mid R_1$ (this comes from looking up state 23 in the hash file). This is the state where the sender is resending after a timeout ($time_0$), the medium is in its starting state, and the receiver is waiting for the next message having already dealt with message 0.

But since the protocol is symmetric with respect to the alternating bit, it stands to reason that there should be another state like this; the same case for the re-sending of message 1. Investigation proves this to be the case, and state 46 (traced through the inverse of the aforementioned method – pick the symmetric state and get it's number, then look in the steady state solution printout) has exactly the same plot.

What this suggests is that each message is being sent many times. For the steady state solution where ::

$$\{def = 1,\ loss = 10,\ retry = 10\}$$

The cumulative time is just under 80%. Therefore, it follows that the system is being throttled somewhere.

Since def is only defined to be 1, it is already know that the probability increases at a slow rate relative to $loss$ and $retry$ which have a rate of 10. Therefore, the system could be re-sending messages before there is adequate chance for it to be delivered and acknowledged. The loss rate also plays an important factor. Looking at the plot, one can see that as the loss rate decreases, the system spends less time in these states (this happens for the timeout rate as well). It follows from this that a more balanced (and efficient) system would have longer timeouts, and a faster transmission medium. Of course, if possible, a less lossy medium would also be desirable, but if there existed a lossless medium then the protocol would no-longer have any use !

More generally, this kind of analysis is useful for identifying states that dominate the system, in terms of time. From here one is able to interpret it, and decide how the system could be altered to improve efficiency. It is also possible to identify the slowest component and for a finite time, determine how many cycles (in this case, how many message-sending cycles) can be performed (although this requires a clear definition of the rates). Other things can be done with

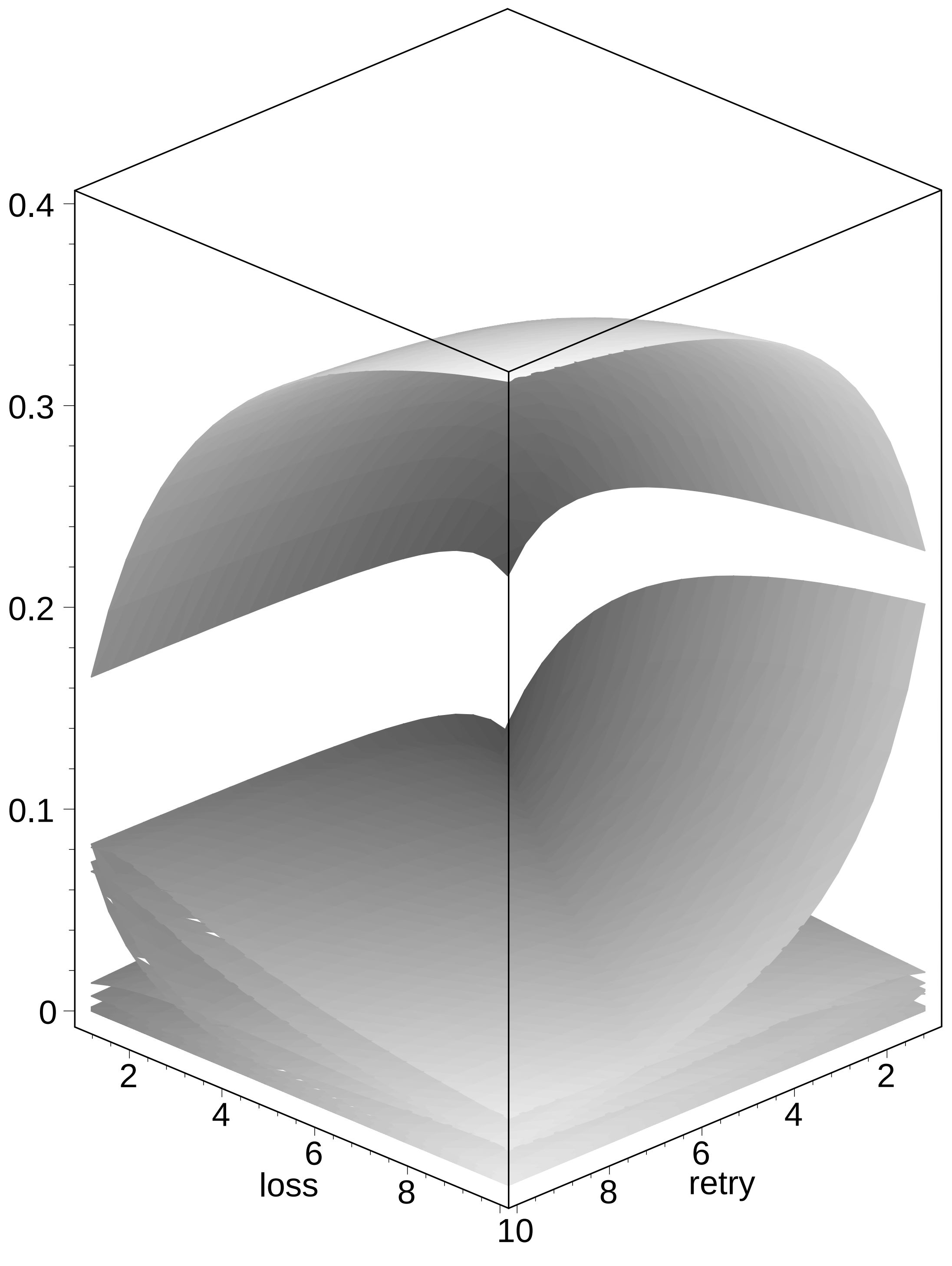

Figure 4: All States Plotted

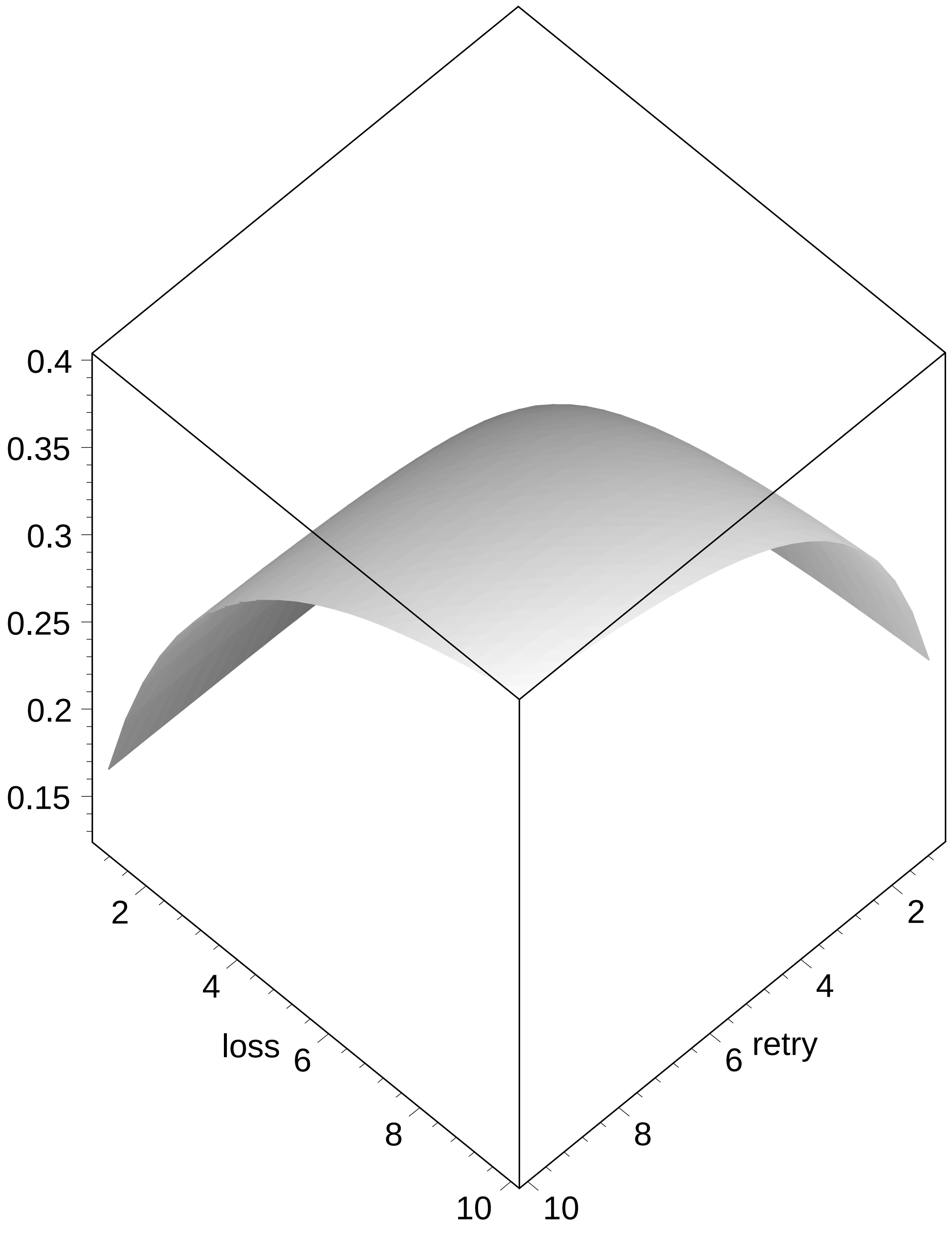

Figure 5: State 23 (and 46)

a clear definition of the rates; from a set of mean times (these are linearly proportional to the rates), cycle times and the like may be computed.

4.3 Action Based Analysis

There are other things that can be done with a clear definition of the rates, when combined with the steady state probabilities - one can determine how many times a desired action is performed, which is very useful. Looking at all states, if all the states which may perform the action in question are filtered out, then you have a vector of states. If for each state in this vector you associate a number which is the rate multiplied by the steady state probability, and sum this vector of numbers, then you have a throughput quantum. This tells you how much time the system spends performing some action (as opposed to spending time in states) in terms of some overall time. For example, if for the alternating bit protocol if it is desired to know how many messages one could send in one hour with our given rates, one would compute and sum the vector, and would end up with a throughput quantum of $.005914$. If rates are defined in terms of seconds, then ::

$$3600 * .005914 = 21.94$$

messages that the system can send per hour. Of course, these rates may not be entirely reflective of a real life system and they were picked at random. However, this shows how one would compute actual throughput if you had properly defined rates.

The reason that this is valid is because it considers the whole systems' cycle time in terms of a cumulative λ, and from all states the probability of doing the action in question is computable. Therefore the sum of these is reflective of the time that the system spends performing this action. It is analogous of multiplying the total time of interest (say one hour) by the amount of time spent in a state (on average), and dividing by the time taken to perform the action you're interested in. This is the case where there is a unique state that can perform the action; where there is more than one it is cumulative; where there is a choice at that state, it is a weighted average (remember, the λ values are linearly proportional to average times).

4.4 Equivalence Checking

Equivalence is one thing that may be checked against, but there exists no convenient tool to automate this analysis to date (although some are under development). Model checkers are the most automated tools to date, only requiring guidance at times where the choice between two rules or axioms is ambiguous. Note that at these times, it is usually the case that only somebody fully conversant with the tool and language in question will be able to steer the model checker in the right direction. Equivalence checking can be used to check that two systems behave the same, such that a more complex system can be shown to be behaviourally equivalent to a simpler system that describes the operational behaviour that one interested in.

Suppose that one wanted to prove that the ABP is behaviourally equivalent to a system that takes a message and passes it out again, a buffered sending agent for example. Suppose the definition is ::

$$\boxed{\begin{array}{lcl} B_1 & \overset{def}{=} & (gm, a).B_2 \\ B_2 & \overset{def}{=} & (gm, a).B_3 + (cm, b).B_1 \\ B_3 & \overset{def}{=} & (cm, b).B_2 \end{array}}$$

To prove that this system is observationally congruent to the ABP is a lengthly process, since the ABP has 94 states. However, the abbreviated final relation is given in figure 6 (a translation table is provided in appendix C). B1, B2 and B3 relate to states in our buffered sender. A1..A94 relate to states in the ABP (these states are outputted by the PWB).

Note that as with the example given in section 2.2.6, most of the ABP's actions (the set of labels not appearing in B) were internalised. However, this result validates the protocol, since it behaves like a lossless order-maintaining buffered sender (with a buffer size of 1), e.g. ::

$$B \approx^c (S_0 \underset{A}{\bowtie} M_0 \underset{B}{\bowtie} R_0)\backslash_{\{send_0, send_1, recv_0, recv_1, a_0, a_1, ack_0, ack_1, drop, time_0, time_1\}}$$

$$
\begin{aligned}
R = \{&(B_1,A_1),(B_1,A_2),(B_2,A_3),(B_3,A_4),(B_2,A_5),(B_2,A_6),(B_2,A_7),(B_3,A_8),(B_3,A_9),(B_3,A_{10}),\\
&(B_3,A_{11}),(B_3,A_{12}),(B_3,A_{13}),(B_3,A_{14}),(B_3,A_{15}),(B_3,A_{16}),(B_3,A_{17}),(B_3,A_{18}),(B_3,A_{19}),(B_3,A_{20}),\\
&(B_3,A_{21}),(B_2,A_{22}),(B_3,A_{23}),(B_3,A_{24}),(B_3,A_{25}),(B_2,A_{26}),(B_3,A_{27}),(B_2,A_{28}),(B_2,A_{29}),(B_2,A_{30}),\\
&(B_3,A_{31}),(B_3,A_{32}),(B_3,A_{33}),(B_3,A_{34}),(B_3,A_{35}),(B_3,A_{36}),(B_3,A_{37}),(B_3,A_{38}),(B_3,A_{39}),(B_3,A_{40}),\\
&(B_3,A_{41}),(B_3,A_{42}),(B_3,A_{43}),(B_3,A_{44}),(B_2,A_{45}),(B_3,A_{46}),(B_3,A_{47}),(B_3,A_{48}),(B_2,A_{49}),(B_3,A_{50}),\\
&(B_2,A_{51}),(B_2,A_{52}),(B_2,A_{53}),(B_3,A_{54}),(B_3,A_{55}),(B_3,A_{56}),(B_3,A_{57}),(B_3,A_{58}),(B_3,A_{59}),(B_3,A_{60}),\\
&(B_2,A_{61}),(B_2,A_{62}),(B_2,A_{63}),(B_2,A_{64}),(B_2,A_{65}),(B_2,A_{66}),(B_2,A_{67}),(B_2,A_{68}),(B_2,A_{69}),(B_2,A_{70}),\\
&(B_1,A_{71}),(B_2,A_{72}),(B_2,A_{73}),(B_2,A_{74}),(B_1,A_{75}),(B_2,A_{76}),(B_2,A_{77}),(B_2,A_{78}),(B_2,A_{79}),(B_2,A_{80}),\\
&(B_2,A_{81}),(B_2,A_{82}),(B_2,A_{83}),(B_2,A_{84}),(B_2,A_{85}),(B_1,A_{86}),(B_2,A_{87}),(B_2,A_{88}),(B_2,A_{89}),(B_1,A_{90}),\\
&(B_2,A_{91}),(B_2,A_{92}),(B_2,A_{93}),(B_2,A_{94})\}
\end{aligned}
$$

Figure 6: Observational Congruence Relation for the ABP versus a reliable buffered medium

$$
\begin{aligned}
A &= \{send_0, send_1, ack_0, ack_1\}\\
B &= \{recv_0, recv_1, a_0, a_1\}
\end{aligned}
$$

5 Conclusions and Remarks

This paper has introduced process algebras with a concrete definition of CCS and the stochastic process algebra PEPA. It has demonstrated how models of systems can be shown to be behaviourally correct with respect to some simpler system which specifies the desired behaviour. It has also illustrated how performance analysis may be completed on a stochastic model, both in terms of state permanence and action throughput.

All of this was presented and then applied to a model of the alternating bit protocol, which has shown to be behaviourally equivalent to a simple buffered sending agent, and with our selected rates, shown to have a mean throughput of about 21.94 messages per hour.

However, there are some caveats. Although a model can be proved to be correct with respect to another model (that could be considered to be a behavioural specification), this has no meaning whatsoever if the model that is being tested for correctness is not correct relative to the protocol or process itself; and there is no way of proving that a model is representative of the thing that is being modelled. This is why the maximum rewards from formal methods can be yielded when they are employed from the specification and design stage. For hardware, languages such as VHDL can be used to “formally” describe circuits, and the behaviour can be verified before it gets to the silicon stage. This has obvious advantages since applying formal methods may cost money, but not as much as recalling millions of defective chips and cutting new templates !

With respect to performance analysis, timings must be considered very carefully. Not all systems may be expressible with a negative exponential distribution, although negative exponential distribution is a tenant of queueing theory. Also, there is some skill required in interpreting the results, and just as experience can aid you to model systems, experience is also necessary to interpret the subsequent analysis.

This shows that formal methods do have a worthwhile function for validating properties of protocols, and could easily show an incorrect protocol, defective digital circuit or deadlockable system to be just that.

A ABP Entire PEPA Specification

$$
\begin{array}{lcl}
S_0 & \overset{def}{=} & (gm, def).(send_0, def).S_1 \\
S_1 & \overset{def}{=} & (gm, def).S_2 + (time_0, retry).S_3 + (ack_0, \top).S_5 + (ack_1, \top).S_1 \\
S_2 & \overset{def}{=} & (time_0, retry).S_4 + (ack_0, \top).S_6 + (ack_1, \top).S_2 \\
S_3 & \overset{def}{=} & (send_0, def).S_1 \\
S_4 & \overset{def}{=} & (send_0, def).S_2 \\
S_5 & \overset{def}{=} & (gm, def).(send_1, def).S_7 \\
S_6 & \overset{def}{=} & (send_1, def).S_7 \\
S_7 & \overset{def}{=} & (gm, def).S_8 + (time_1, retry).S_6 + (ack_0, \top).S_7 + (ack_1, \top).S_0 \\
S_8 & \overset{def}{=} & (time_1, retry).S_9 + (ack_0, \top).S_8 + (ack_1, \top).S_3 \\
S_9 & \overset{def}{=} & (send_1, def).S_8 \\
R_0 & \overset{def}{=} & (recv_0, \top).(cm, def).(a_0, def).R_1 \\
R_1 & \overset{def}{=} & (recv_0, \top).(a_0, def).R_1 + (recv_1, \top).R_2 \\
R_2 & \overset{def}{=} & (cm, def).(a_1, def).R_3 \\
R_3 & \overset{def}{=} & (recv_1, \top).(a_1, def).R_3 + (recv_0, \top).R_4 \\
R_4 & \overset{def}{=} & (cm, def).(a_0, def).R_1 \\
M_0 & \overset{def}{=} & (send_0, \top).M_1 + (send_1, \top).M_2 + (a_0, \top).M_3 + (a_1, \top).M_4 \\
M_1 & \overset{def}{=} & (drop, loss).M_0 + (recv_0, def).M_0 \\
M_2 & \overset{def}{=} & (drop, loss).M_0 + (recv_1, def).M_0 \\
M_3 & \overset{def}{=} & (drop, loss).M_0 + (ack_0, def).M_0 \\
M_4 & \overset{def}{=} & (drop, loss).M_0 + (ack_1, def).M_0
\end{array}
$$

$$
(S0 \underset{A}{\bowtie} M0) \underset{B}{\bowtie} R0
$$

$$
\begin{array}{lcl}
A & \overset{def}{=} & \{send_0, send_1, ack_0, ack_1\} \\
B & \overset{def}{=} & \{recv_0, recv_1, a_0, a_1\}
\end{array}
$$

B Maple Session Script

Please note that the output has been removed from this script. This script is included so that readers can gain some insight relating to the operations necessary to produce the given graphs and figures.

```
> restart:
> with(linalg): with(plots):
> Q:=array(sparse,1..94,1..94):
> read ‘/home/staff/edwards/unix/publications/pepa/abp.maple‘:
> b:=array(sparse,1..94):
> for i to 94 do
>   Q[i,94]:=1.00
> od:
> b[94]:=1:
> QT:=transpose(Q):
> XX:=linsolve(QT,b):
> XXX:={}:
> for ii to 94 by 1 do
>   XXX:={op(ii,XX),XXX}
> od:
> op(op(XX)):
> XXX:=convert(XX,list):
> def:=1.00:
> plot3d({op(XXX)},retry=1..10,loss=1..10,axes=boxed,style=PATCHNOGRID);
> for S from 1 to 94 by 1 do
> plot3d(op(S,{op(XXX)}),loss=1..10,retry=1..10,axes=BOXED,style=PATCHNOGRID)
> od;
>
```

If you’re trying to execute this, note that the path to the Maple file needs to be changed for your local set-up.

C Equivalence Translation Table

A_1	$S_0 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_0$
A_2	$(send_0, 1.0).S_1 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_0$
A_3	$S_1 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > R_0$
A_4	$S_1 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > (cm, 1.0).(a_0, 1.0).R_1$
A_5	$S_2 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > R_0$
A_6	$S_3 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > R_0$
A_7	$S_1 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_0$
A_8	$S_2 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > (cm, 1.0).(a_0, 1.0).R_1$
A_9	$S_3 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > (cm, 1.0).(a_0, 1.0).R_1$
A_{10}	$S_1 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > (a_0, 1.0).R_1$
A_{11}	$S_4 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > R_0$
A_{12}	$S_2 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_0$
A_{13}	$S_3 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_0$
A_{14}	$S_4 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > (cm, 1.0).(a_0, 1.0).R_1$
A_{15}	$S_2 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > (a_0, 1.0).R_1$
A_{16}	$S_1 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > (cm, 1.0).(a_0, 1.0).R_1$
A_{17}	$S_3 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > (a_0, 1.0).R_1$
A_{18}	$S_1 < send_1, send_0, ack_1, ack_0 > M_3 < recv_1, recv_0, a_1, a_0 > R_1$
A_{19}	$S_4 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_0$
A_{20}	$S_2 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > (cm, 1.0).(a_0, 1.0).R_1$
A_{21}	$S_4 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > (a_0, 1.0).R_1$
A_{22}	$S_2 < send_1, send_0, ack_1, ack_0 > M_3 < recv_1, recv_0, a_1, a_0 > R_1$
A_{23}	$S_3 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > (cm, 1.0).(a_0, 1.0).R_1$
A_{24}	$S_1 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > (a_0, 1.0).R_1$
A_{25}	$S_3 < send_1, send_0, ack_1, ack_0 > M_3 < recv_1, recv_0, a_1, a_0 > R_1$
A_{26}	$S_5 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_1$
A_{27}	$S_1 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_1$
A_{28}	$S_4 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > (cm, 1.0).(a_0, 1.0).R_1$
A_{29}	$S_2 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > (a_0, 1.0).R_1$
A_{30}	$S_4 < send_1, send_0, ack_1, ack_0 > M_3 < recv_1, recv_0, a_1, a_0 > R_1$
A_{31}	$S_6 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_1$
A_{32}	$S_2 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_1$
A_{33}	$S_3 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > (a_0, 1.0).R_1$
A_{34}	$S_3 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_1$
A_{35}	$(send_1, 1.0).S_7 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_1$
A_{36}	$S_4 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > (a_0, 1.0).R_1$
A_{37}	$S_4 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_1$
A_{38}	$S_7 < send_1, send_0, ack_1, ack_0 > M_2 < recv_1, recv_0, a_1, a_0 > R_1$
A_{39}	$S_1 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > R_1$
A_{40}	$S_2 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > R_1$
A_{41}	$S_7 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_2$
A_{42}	$S_8 < send_1, send_0, ack_1, ack_0 > M_2 < recv_1, recv_0, a_1, a_0 > R_1$
A_{43}	$S_6 < send_1, send_0, ack_1, ack_0 > M_2 < recv_1, recv_0, a_1, a_0 > R_1$
A_{44}	$S_7 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_1$
A_{45}	$S_3 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > R_1$
A_{46}	$S_4 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > R_1$
A_{47}	$S_8 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_2$
A_{48}	$S_6 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_2$

A_{49}	$S_7 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > (a_1, 1.0).R_3$
A_{50}	$S_9 < send_1, send_0, ack_1, ack_0 > M_2 < recv_1, recv_0, a_1, a_0 > R_1$
A_{51}	$S_8 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_1$
A_{52}	$S_9 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_2$
A_{53}	$S_8 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > (a_1, 1.0).R_3$
A_{54}	$S_7 < send_1, send_0, ack_1, ack_0 > M_2 < recv_1, recv_0, a_1, a_0 > R_2$
A_{55}	$S_6 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > (a_1, 1.0).R_3$
A_{56}	$S_7 < send_1, send_0, ack_1, ack_0 > M_4 < recv_1, recv_0, a_1, a_0 > R_3$
A_{57}	$S_9 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_1$
A_{58}	$S_8 < send_1, send_0, ack_1, ack_0 > M_2 < recv_1, recv_0, a_1, a_0 > R_2$
A_{59}	$S_9 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > (a_1, 1.0).R_3$
A_{60}	$S_8 < send_1, send_0, ack_1, ack_0 > M_4 < recv_1, recv_0, a_1, a_0 > R_3$
A_{61}	$S_6 < send_1, send_0, ack_1, ack_0 > M_2 < recv_1, recv_0, a_1, a_0 > R_2$
A_{62}	$S_7 < send_1, send_0, ack_1, ack_0 > M_2 < recv_1, recv_0, a_1, a_0 > (a_1, 1.0).R_3$
A_{63}	$S_6 < send_1, send_0, ack_1, ack_0 > M_4 < recv_1, recv_0, a_1, a_0 > R_3$
A_{64}	$S_0 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_3$
A_{65}	$S_7 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_3$
A_{66}	$S_9 < send_1, send_0, ack_1, ack_0 > M_2 < recv_1, recv_0, a_1, a_0 > R_2$
A_{67}	$S_8 < send_1, send_0, ack_1, ack_0 > M_2 < recv_1, recv_0, a_1, a_0 > (a_1, 1.0).R_3$
A_{68}	$S_9 < send_1, send_0, ack_1, ack_0 > M_4 < recv_1, recv_0, a_1, a_0 > R_3$
A_{69}	$S_3 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_3$
A_{70}	$S_8 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_3$
A_{71}	$S_6 < send_1, send_0, ack_1, ack_0 > M_2 < recv_1, recv_0, a_1, a_0 > (a_1, 1.0).R_3$
A_{72}	$S_6 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_3$
A_{73}	$(send_0, 1.0).S_1 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_3$
A_{74}	$S_9 < send_1, send_0, ack_1, ack_0 > M_2 < recv_1, recv_0, a_1, a_0 > (a_1, 1.0).R_3$
A_{75}	$S_9 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_3$
A_{76}	$S_1 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > R_3$
A_{77}	$S_7 < send_1, send_0, ack_1, ack_0 > M_2 < recv_1, recv_0, a_1, a_0 > R_3$
A_{78}	$S_8 < send_1, send_0, ack_1, ack_0 > M_2 < recv_1, recv_0, a_1, a_0 > R_3$
A_{79}	$S_1 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_4$
A_{80}	$S_2 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > R_3$
A_{81}	$S_3 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > R_3$
A_{82}	$S_1 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_3$
A_{83}	$S_6 < send_1, send_0, ack_1, ack_0 > M_2 < recv_1, recv_0, a_1, a_0 > R_3$
A_{84}	$S_9 < send_1, send_0, ack_1, ack_0 > M_2 < recv_1, recv_0, a_1, a_0 > R_3$
A_{85}	$S_2 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_4$
A_{86}	$S_3 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_4$
A_{87}	$S_4 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > R_3$
A_{88}	$S_2 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_3$
A_{89}	$S_4 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_4$
A_{90}	$S_1 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > R_4$
A_{91}	$S_4 < send_1, send_0, ack_1, ack_0 > M_0 < recv_1, recv_0, a_1, a_0 > R_3$
A_{92}	$S_2 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > R_4$
A_{93}	$S_3 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > R_4$
A_{94}	$S_4 < send_1, send_0, ack_1, ack_0 > M_1 < recv_1, recv_0, a_1, a_0 > R_4$

D Acknowledgements

This author would like to thank the many people that have contributed to the authoring of this paper, and those who proof read it and gave useful feedback. This author would also like to thank them for their candor and patience. These people (in no particular order) include Kerstin Eder, Friedger Mueffke, Ian Edwards, Nigel Smart and Ian Holyer.

References

[1] C.A.R Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[2] Robin Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science, 1989.

[3] Jane Hilston. *Compositional Markovian Modelling Using A Process Algebra*. May 1994.

[4] Waterloo Software. *Maple*. http://www.maplesoft.com/

[5] Holger Hermanns. *Markovian Process Algebra*. In the proceedings of the 1st Euro Summerschool on Trends in Computer Science (Formal Methods and Performance Analysis special issue), July 2000.

[6] Mathworks Software. *Matlab*. http://www.mathworks.com/products/matlab/

[7] Graham Clark, Stephen Gilmore and Jane Hillston. *The PEPA Performance Modelling Tools*. Technical Report, May 1999.

[8] Stephen Gilmore and Graham Clark. *PEPA State Finder*. A PEPA tool for enumerating state spaces. See http://www.dcs.ed.ac.uk/pepa/tools.html for more details.

[9] Jane Hilston and Stephen Gilmore. *The PEPA Workbench : A Tool to Support a Process Algebra-based Approach to Performance Modelling*.

[10] Rance Cleaveland and Scott Smolka. *Process Algebra*. April 1999.

[11] Marco Bernado et al. *A Stochastic Process Algebra Model for the Analysis of the Alternating Bit Protocol*. In the proceedings of the 11th International Symposium on Computer and Information Sciences, 1996.

[12] Graham Clark and Jane Hilston. *Towards Automatic Derivation of Performance Measures from PEPA Models*. In the proceedings of UKPEW, 1996.