# Refining internal choice in PEPA models

Stephen Gilmore* Jane Hillston*

6th August 1996

**Abstract**

In a previous paper [GHH96] the authors presented a program development technique for stochastic process algebra models which was centred on the translation from a model into an abstract program skeleton which presents the structural and behavioural information from the model in programming language notation. The intention of this work, which we continue here, is to support the methodical development of concurrent programs from stochastic process algebra specifications of their behaviour and performance.

## 1 Introduction

When deriving a concurrent program from a stochastic process algebra model significant care must be taken to achieve the correct treatment of choices in the model. Choices in a stochastic process algebra model represent abstract branching in the progress of the system; this branching is resolved by race conditions which determine the branch to be taken. Choices in the corresponding programming language representation are branching statements such as **case** statements or, more subtly, non-deterministic branching caused by the resolution of requests by concurrently active tasks to synchronise and exchange information. This latter form of choice has been called *implicit choice*. Three kinds of activity are found in a stochastic process algebra model: *shared active*, *shared passive* and *individual*. In general, a choice in a stochastic process algebra model could involve activities of all three kinds

*Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh, King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, Scotland, U.K.

although a particularly problematic case is one with a choice between activities of the same kind, namely shared active. The difficulty in this case is that some of the activities are the same named activity although they might be performed at different rates or lead to different derivatives. We term this choice between two different forms of the same activity or two different outcomes, an *internal choice*. The problems associated with internal choice did not arise in our earlier paper because we placed a restriction upon choices to ensure that they were between different activities and our translation to a programming language representation crucially depended upon this. We present here one approach to solving the problems associated with internal choice and explain its significance for the class of stochastic process algebra models to which our translation can be applied.

### Structure of this paper

Our problem is related to the combined use of two formal notations, the PEPA stochastic process algebra [Hil96b] and the Ada programming language [Bar96]. We begin by first summarising the relevant features of these notations. We then explain the translation process which develops an Ada program in the form of an abstract program skeleton from a PEPA specification. We then go on to explain the problem of internal choice and present an example where it arises in practice when modelling a simple system comprising a process and an unreliable resource. We explain how our problem can be solved by an extension of our set of rules used in the translation process and show that the solution is an acceptable one. The additional rules appear in Section 5 and the existing rule set appears in the appendix. Section 8 concludes the work and outlines the future directions into which we intend to extend this work.

## 2 PEPA

The basic elements of PEPA are *components* and *activities*, corresponding to states and transitions in a Markov process. Each activity has an *action type* (or simply *type*). Activities which are private to the component in which they occur are represented by the distinguished action type, $\tau$. The duration of each activity is represented by the parameter of the associated exponential distribution: the *activity rate* (or simply *rate*) of the activity. This parameter may be any positive real number, or the distinguished symbol $\top$ (read as *unspecified*). Thus each activity, $a$, is a pair $(\alpha, r)$ where $\alpha$ is the action type and $r$ is the activity rate.

$$
\begin{array}{rcll}
\mathcal{P} & ::= & \mathcal{DS};\ \mathcal{C} & \text{declarations; component} \\
\mathcal{DS} & ::= & & \text{declaration sequence} \\
 & & \mathcal{D} & \\
 & | & \mathcal{D};\ \mathcal{DS} & \\
\mathcal{D} & ::= & \mathcal{I} \stackrel{\mathit{def}}{=} \mathcal{S} & \text{identifier declaration} \\
\mathcal{S} & ::= & & \text{sequential components} \\
 & & \mathcal{I} & \text{identifier use} \\
 & | & (\mathcal{A}, \mathcal{R}).\mathcal{S} & \text{prefix} \\
 & | & \mathcal{S} + \mathcal{S} & \text{choice} \\
\mathcal{C} & ::= & & \text{concurrent components} \\
 & & \mathcal{C}/L & \text{hiding, } L \text{ an identifier set} \\
 & | & \mathcal{S} \underset{L}{\bowtie} \mathcal{S} & \text{co-operation pair} \\
 & | & \mathcal{S} \underset{L}{\bowtie} \mathcal{C} & \text{general co-operation} \\
\mathcal{A} & ::= & \tau & \text{internal, silent activity} \\
 & | & \mathit{indiv} & \text{individual activity} \\
 & | & \mathit{shared} & \text{shared activity} \\
\mathcal{R} & ::= & \top & \text{unspecified rate} \\
 & | & \mathbb{R} & \text{real number rate} \\
\mathcal{I} & ::= & \text{identifier} & \text{alphanumeric sequence} \\
\mathit{indiv} & ::= & \alpha, \beta, \gamma, \ldots & \\
\mathit{shared} & ::= & \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}, \ldots &
\end{array}
$$

Figure 1: Grammar for the PEPA subset

PEPA provides a small set of combinators. These allow expressions, or terms, to be constructed defining the behaviour of components, via the activities they undertake and the interactions between them. The grammar for terms in PEPA is given in Figure 1. The components in a co-operation proceed independently with any activities whose types do not occur in the *co-operation set* $L$ (these are *individual activities*). However, activities with action types in the set $L$ require the simultaneous involvement of both components (these are *shared activities*). These activities are only enabled in a synchronisation between $P$ and $Q$ on $L$ when they are enabled in both $P$ and $Q$. The notation $P \parallel Q$ is used when the synchronisation set is empty.

If an activity has an unspecified rate in a component, the component is *passive* with respect to that action type. This means that the component does not influence the rate at which any shared activity occurs. The co-operation combinator associates to the left but brackets may also be used.

A *race condition* governs the dynamic behaviour of a model whenever more than one activity is enabled. This has the effect of replacing the non-deterministic branching of classical process algebra with probabilistic branching. The probability that a particular activity completes is given by the ratio of the activity rate to the sum of the activity rates of all the enabled activities. Any other activities which were simultaneously enabled will be *interrupted* or *aborted*. The memoryless property of the exponential distribution makes it unnecessary to record the remaining lifetime in either case.

The semantics of PEPA, presented in structured operational semantics style, are given in [Hil96b]. The underlying labelled multi-transition system also characterises the Markov process represented by the model. The states of the system are *derivatives*. The *derivative set* of a component is defined recursively (see [Hil96b] for details).

The *derivation graph* is a graph in which syntactic terms form the nodes, and arcs represent the possible transitions between them: the operational rules define the form of this graph. Since the relation used in the semantic definition is a multi-relation, the graph is a multigraph. The derivation graph describes the possible behaviour of any PEPA component and provides a useful way to reason about a model. It is also the basis of the construction of the underlying Markov process: a state is associated with each node of the derivation graph, and the transitions between states are derived from the arcs of the graph. The *transition rate* between two components $C_i$ and $C_j$, denoted $q(C_i, C_j)$, is the sum of the activity rates labelling arcs connecting node $C_i$ to node $C_j$. This use of the derivation graph is analogous to the use of the reachability graph in stochastic extensions of Petri nets such as GSPNs [ACB84].

# 3 Ada

Ada is an imperative programming language which extends Pascal-like languages with facilities for large-scale programming and real-time and concurrent programming. The core language which is used for programming in the small has blocks such as procedures and functions and commands such as assignment, case and conditional statements and a general loop construct which allows an exit from any point within the body. Support for programming in the large is provided in the form of the **package**, which collects together related definitions of procedures and data types, allowing the programmer to define an interface which will hide some of these definitions while allowing others to be accessed by other program units such as procedures and other packages.

In Ada separate entities within a program may be implemented as **task**s. Multiple instances of a task are created by defining a **task type** and declaring task variables of this type. However they have been created, tasks run, at least conceptually, on separate processors and are independent of each other as far as resource contention is concerned. Within a task, distinct sections of sequential behaviour may be packaged into *entries*: entries in tasks are parameterised sequences of commands and thus are analogous to procedures in packages. There are two ways in which tasks interact with each other:

- directly, by message passing in a synchronisation known as a *rendezvous*; or
- indirectly, via shared data which is accessible to both tasks.

Clearly the rendezvous most closely matches cooperation in PEPA. In a rendezvous one task makes a call to an entry in another task. The shared activities of the two tasks are described within an **accept** statement. The task which will be performing the work is at liberty to decide when to accept an entry request and when to refuse. If an entry request is being refused, the caller may decide to dispense with its call and perform other work, perhaps even if it is only to call on another task instead. The language provides support for this kind of queueing and reneging via the **select** statement. A select statement is sometimes used together with a **delay** statement.

# 4 Translating PEPA to Ada

Much of the published work on stochastic process algebras has shown that they can be used to model existing computer systems, not limited to only computer software [Hol95, GHHR96]. When developing a novel software-based system we believe that it is wise for the system developers to expend effort on checking their designs at an early stage both for behavioural suitability and for suitable performance. Our preferred approach to this is to first construct and analyse a stochastic process algebra model.

In moving from a stochastic process algebra model to an efficiently executing program in a programming language we expect that at times we shall have to consider the relationships between different versions of the system specification and between different intermediate designs for the system software. When we are comparing specifications, we have an assembly of different algebraic relations [Hil96b] which can be deployed either manually or mechanically. We appreciate that it is perhaps unrealistic to expect to manipulate large software components within complex systems in the same fashion. Thus when we compare the relative performance of versions of the system software we will automate this process as much as is possible. Thus we will typically be comparing versions of the system software which are instrumented with additional software monitoring.

In order to be able to make a small modification to the specification and to assess the impact of this modification on the associated program it is necessary to automate also some part of program development. We have chosen to automate the implementation of concurrently executing components at the level of an abstract Ada program. The abstract program has the same static task structure and dynamic patterns of synchronisation and communication as the finished system but represents computation only by delays which reflect the expected time to execute the relevant operations. An example which illustrates the form of abstract program which is produced by our translation process appears in Figure 2. Notice that the buffer task which is generated does not include the variable declaration, assignments and parameter passing which would appear in the finished implementation. This software development work is to be performed separately later. Instrumentation which is automatically added to the generated Ada program has been omitted from the figure.

## 5 Internal choice

We now present an example where internal choice is used in a stochastic process algebra model of a simple concurrent system. A similar example is considered in [Hil96a].

Our system is composed of a professor using a printer. The printer is unreliable and will occasionally jam with probability $p$. When the printer jams it remains unavailable for use until it is repaired by the professor's secretary. More generally, the model represents the use of a resource—in our case, the printer—accessed in the second phase of a two-phase process—in our case, the professor. Generally the 'secretary' could be a 'repairman' which is perhaps a daemon process running with low priority. The two-phase process and the repair process never interact directly*. The stochastic

*This is somewhat unrealistic in the case of the professor and her secretary but let us simply say that we are not modelling the other interaction here. Nor, of course, are we modelling the other important duties which the secretary will perform in the course of his working day.

$$
\begin{aligned}
Buffer &\stackrel{def}{=} (Put, r).(Get, r).Buffer \\
Producer &\stackrel{def}{=} (Put, \top).Producer \\
Consumer &\stackrel{def}{=} (Get, \top).Consumer \\
Producer &\underset{\{Put\}}{\bowtie} Buffer \underset{\{Get\}}{\bowtie} Consumer
\end{aligned}
$$

⋆

```
task body Buffer is
begin
  loop
    accept Put do
      delay (1.0/r);
    end Put;
    accept Get do
      delay (1.0/r);
    end Get;
  end loop;
end Buffer;
```

```
task body Producer is
begin
  loop
    Buffer.Put;
  end loop;
end Producer;

task body Consumer is
begin
  loop
    Buffer.Get;
  end loop;
end Consumer;
```

⋆

```
task body Buffer is
  V: Item;
begin
  loop
    accept Put(X: in Item) do
      V := X;
    end Put;
    accept Get(X: out Item) do
      X := V;
    end Get;
  end loop;
end Buffer;
```

Figure 2: Sample PEPA input, intermediate Ada output and final implementation of the *Buffer* task

$$
\begin{array}{lcl}
\mathit{Professor} & \stackrel{\mathit{def}}{=} & (\mathit{write}, r_1).(\mathit{print}, \top).\mathit{Professor} \\[1ex]
\mathit{Printer} & \stackrel{\mathit{def}}{=} & (\mathit{print}, (1-p) \times r_2).\mathit{Printer} + \\
 & & (\mathit{print}, p \times r_2).\mathit{Jammed\_Printer} \\
\mathit{Jammed\_Printer} & \stackrel{\mathit{def}}{=} & (\mathit{unjam}, \top).\mathit{Printer} \\[1ex]
\mathit{Secretary} & \stackrel{\mathit{def}}{=} & (\mathit{unjam}, r_3).\mathit{Secretary} \\[1ex]
\mathit{System} & \stackrel{\mathit{def}}{=} & (\mathit{Professor} \parallel \mathit{Secretary}) \underset{S}{\bowtie} \mathit{Printer} \\[1ex]
 & & \text{where } S = \{\ \mathit{print}, \mathit{unjam}\ \}
\end{array}
$$

Figure 3: A system with an unreliable resource

process algebra model of the system is shown in Figure 3.

The occurrence of an internal choice in this model is within the *Printer* component. When the professor uses the printer she does so without knowing whether or not this use will cause it to jam and require repair. From the perspective of the professor this seems to be an internal choice made by the printer after it had been begun printing.

Our previous translation could not process this example because it represented such choices using an Ada **select** statement with the initial activities of the various alternatives in the selection. These alternatives must be distinct and in consequence our translation cannot provide a programming language representation for any stochastic process algebra models with internal choices. To address this problem we exploit the fact that our programming language representation separates rate information from activity types. There are two reasons why this was done:

1. the same activity may be performed at different rates at different occasions perhaps representing a more costly version of an operation which invokes garbage collection or re-balances an internal data structure;

2. there is an silent activity which has a rate but does not have a name, corresponding to a $\tau$ activity in the stochastic process algebra model.

For these reasons activity names are represented in our translation by named statement blocks such as procedures or entries in tasks and rates are represented using **delay** statements.

## 5.1 A simple rule for internal choice

It might seem that the problem of handling internal choices in stochastic process algebra models can be solved simply by bringing together two of the rules from Appendix A. The first is the rule for $(\boldsymbol{\alpha}, r).S$ in Section A.2 and the second is the rule for $\sum_{i=1}^{m}(\alpha_i, t_i).T_i$ in Section A.3. However, such a simple construction would not provide us with a sound rule and we would be forced to add a condition on its applicability.

$$\frac{R_i \leadsto \widehat{R_i} \qquad \widehat{R_i}\ \mathsf{named}}{\begin{array}{lll} \sum_{i=1}^{k}(\boldsymbol{\alpha}, r_i).R_i & \leadsto & \textbf{accept}\ \boldsymbol{\alpha}\ \textbf{do} \\ & & \quad \textbf{declare}\ \mathsf{A} : \mathsf{rate\_list}(1..k) := (r_1, \ldots, r_k); \\ & & \quad \textbf{begin} \\ & & \qquad \textbf{case}\ \mathsf{psrf}\ (\mathsf{A})\ \textbf{is} \\ & & \qquad\quad [\textbf{when}\ i \Rightarrow \textbf{delay}\ (1.0/r_i);\ \widehat{R_i}]_{i=1}^{k} \\ & & \qquad\quad \textbf{when others} \Rightarrow \textbf{null}; \\ & & \qquad \textbf{end case}; \\ & & \quad \textbf{end}; \\ & & \textbf{end}\ \boldsymbol{\alpha}; \end{array}}$$

The declaration at the start of the **accept** statement introduces an array of dimension $k$ of real number values. The function psrf when given an array containing rates $r_1$ to $r_k$ computes a pseudo-random integer value in the range 1 to $k$ with the correct expectation relative to these rates. The default clause at the end of the **case** statement [**when others** ...] is required because the Ada language does not allow cases to be omitted. However, the psrf function will always return a number in the allowed range and thus every execution of the $\boldsymbol{\alpha}$ activity will have an associated delay and the case of the **null** [do nothing] statement will never be selected.

The condition upon the rule, $\widehat{R_i}$ named, requires all of the derivatives to be identifiers in order that they have a simple translation which does not involve further inter-task communication. It is not possible to weaken this condition, say to include $\tau$ actions or to allow a choice between named derivatives because both of these have associated delays which would be conflated with the delay associated with $\boldsymbol{\alpha}$ in the rule above. It would however be possible to include in the grammar for the PEPA language the restriction that every prefix activity must be followed by a named derivative. We have chosen not to do this because it would be an unnecessary burden in the many cases where it is not needed. We believe that it is preferable to deal with this problem in the place where it arises through the device of an auxiliary function, the named predicate.

## 5.2 An example which does not respect naming

Consider the following simple system which is composed of two mutually communicating tasks.

$$\begin{array}{lcl} P & \stackrel{def}{=} & (\boldsymbol{\alpha}, r).(\boldsymbol{\beta}, \top).P + (\boldsymbol{\alpha}, s).(\boldsymbol{\beta}, \top).P \\ Q & \stackrel{def}{=} & (\boldsymbol{\alpha}, \top).(\boldsymbol{\beta}, s).Q \\ Sys & \stackrel{def}{=} & P \underset{\{\boldsymbol{\alpha},\boldsymbol{\beta}\}}{\bowtie} Q \end{array}$$

The component $P$ has an internal choice. The activity $\boldsymbol{\alpha}$ will either be performed with rate $r$ or with rate $s$ and the calling task cannot determine beforehand which rate will apply for any given call instance. The two occurrences of $(\boldsymbol{\beta}, \top).P$ within the component $P$ are not named and we wish to use this example to illustrate why the naming condition which we have added to the previous rule is needed.

Following the application of rules without respecting the naming condition would lead us to generate a programming language representation which would deadlock. This is caused because the task $P$ accepts an $\boldsymbol{\alpha}$ entry request from $Q$ and within that, calls on task $Q$ to perform activity $\boldsymbol{\beta}$. At this time $Q$ is of course suspended by the $\boldsymbol{\alpha}$ call to $P$. This is a deadlock but the stochastic process algebra model had no deadlocks. Such a translation is not acceptable.

The following simple re-phrasing of the model will make this example suitable for use with the rule which we have already seen.

$$\begin{array}{lcl} P & \stackrel{def}{=} & (\boldsymbol{\alpha}, r).P' + (\boldsymbol{\alpha}, s).P' \\ P' & \stackrel{def}{=} & (\boldsymbol{\beta}, \top).P \\ Q & \stackrel{def}{=} & (\boldsymbol{\alpha}, \top).(\boldsymbol{\beta}, s).Q \\ Sys & \stackrel{def}{=} & P \underset{\{\boldsymbol{\alpha},\boldsymbol{\beta}\}}{\bowtie} Q \end{array}$$

Evolving to the derivative named $P'$ is achieved by an assignment to the local variable which records the present derivative and thus does not involve any further inter-task communication. However, the re-working of a model will not always be so facile and we are motivated to produce a generally applicable rule.

## 5.3 An improved rule for internal choice

The problems which arose with the previous rule could be attributed to the more liberal use of statements within an **accept**. Relaxing our previous convention of only permitting **delay** statements within an **accept** led to the situation where a non-deadlocking model was translated into a deadlocking program. We dispense with these problems by avoiding inter-task communication within an **accept**.

$$\frac{R_i \rightsquigarrow \widehat{R_i}}{\sum_{i=1}^{k}(\boldsymbol{\alpha}, r_i).R_i \;\rightsquigarrow\; \begin{array}{l}\textbf{accept } \boldsymbol{\alpha} \textbf{ do} \\ \quad \textbf{declare } \mathsf{A} : \mathsf{rate\_list}(1..k) := (r_1, \ldots, r_k); \\ \quad \textbf{begin } \mathsf{choice} := \mathsf{psrf}\ (\mathsf{A}); \\ \quad \textbf{end}; \\ \quad \textbf{case } \mathsf{choice} \textbf{ is} \\ \qquad [\textbf{when } i \Rightarrow \textbf{delay } (1.0/r_i);]_{i=1}^{k} \\ \qquad \textbf{when others} \Rightarrow \textbf{null}; \\ \quad \textbf{end case}; \\ \textbf{end } \boldsymbol{\alpha}; \\ \textbf{case } \mathsf{choice} \textbf{ is} \\ \quad [\textbf{when } i \Rightarrow \widehat{R_i}]_{i=1}^{k} \\ \quad \textbf{when others} \Rightarrow \textbf{null}; \\ \textbf{end case};\end{array}}$$

A minor disadvantage of this translation is the length of the Ada statement sequence which is produced. A second, and more serious disadvantage is the use of the additional variable, choice. The correctness of the translation relies crucially upon the choice variable retaining its value between the first inspection in the **case** statement within the **accept** statement and the second inspection afterwards. This is clearly unproblematic in the abstract program because the body of the first **case** statement contains only **delay** statements and a **null** statement. However, as this abstract program is successively refined towards the final system implementation, these statements will be replaced with assignments and uses of other Ada language constructs and care must be taken to ensure that the internal choice which was made within the body of the **accept** statement is respected in the statements which follow it. For this reason, the previous rule should always be used in preference to the general one in any cases where it is applicable.

# 6 The translated model

Both of the rules for internal choice which we have shown have been added to the PEPA to Ada translator which we have previously implemented. In Figure 4 we show the translation of the *Printer* component from our model which illustrated the unreliable resource problem. The rate constants R2 and R3 have the values $(1 - p) \times r_2$ and $p \times r_2$ respectively.

```
task body Printer is
  type states is (Jammed_Printer, Printer);
  state: states := Printer;
begin
  loop
    case state is
      when Jammed_Printer =>
        Secretary.unjam;
        state := Printer;
      when Printer =>
        -- an internal choice
        accept print do
          declare
            A: rate_list(1..2) := (R2, R3);
          begin
            case psrf (A) is
              when 1 => delay (1.0/R2); state := Printer;
              when 2 => delay (1.0/R3); state := Jammed_Printer;
              when others => null;
            end case;
          end;
        end print;
    end case;
  end loop;
end Printer;
```

Figure 4: The translation produced for the *Printer* component

One significant change in the patterns of communication which arises because we bring together separate **accept** statements into a single one is that a **select** statement which might otherwise have seemed to be necessary is shown to be unnecessary and consequently omitted. This identification and removal of unnecessary statements brings attendant benefits which simplify the processes of reasoning about and checking the communication within the model. We are keen to implement simplifications at this level which bring with them a significant reduction in the complexity of the generated programming language representation of the stochastic process algebra model. We have not implemented some small optimisations which might make the generated program more difficult to understand and relate back to its specification. An example of the latter kind of optimisations would be the removal of the redundant state := Printer command in the **when** Printer $\Rightarrow \ldots$ clause in the outer **case** statement in Figure 4. Omitting this command brings no attendant benefit in terms of reduction in complexity or increase in ease of understanding and so we are not motivated to do this.

# 7 The translation tool

Our translator has been implemented in the Standard ML [MTH90] programming language as an extension to the PEPA Workbench [GH94]. Although Standard ML is an imperative programming language we have used only its functional subset. The tool accepts as input a PEPA model and performs static analysis to determine which derivatives should become Ada tasks and which become states which these tasks adopt. Interfaces and scope information is calculated in this phase. Using the information gathered, the dynamic phase then implements the rules shown earlier and those in Appendix A.

# 8 Conclusions and future work

The identification of internal choice as a modelling construction of interest which would appear in many stochastic process algebra models of computer systems directed us to give special attention to occurrences of internal choice in models. The representation of this construction in programming language notation is sensitive to the encoding in the stochastic process algebra model of the operations which follow the internal choice. We identified a syntactic condition which distinguished between two frequently occurring sub-cases and provided two rules which could be applied, in the case of the first when the condition was satisfied and the case of the second when it was not. We extended our existing translator tool to include these two rules.

We encountered difficulties in this part of our work which we had not faced previously because we introduced a form of Ada **accept** statement which could express multi-party synchronisation instead of the two party synchronisation which we had previously employed. We noted that this facility, if used incautiously, could give rise to the derivation of a deadlocking program from a non-deadlocking model. We added a condition of use to the problematic rule in order to prevent difficulties of this kind. However, we will be compelled to return to this problem again in the future because we wish to allow the multi-party synchronisation which is provided in the PEPA language and this would seem to force us to add some facility for use of nested Ada **accept** statements in our generated abstract programs.

# References


[ACB84] M. Ajmone Marsan, G. Conte, and G. Balbo. A Class of Generalised Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93–122, May 1984.

[Bar96] J. Barnes. *Programming in Ada 95.* Addison-Wesley, 1996.

[GH94] S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, number 794 in Lecture Notes in Computer Science, pages 353–368, Vienna, May 1994. Springer-Verlag.

[GHH96] S. Gilmore, J. Hillston, and D.R.W. Holton. From SPA models to programs. In Marina Ribaudo, editor, *Proceedings of the Fourth Annual Workshop on Process Algebra and Performance Modelling.* Universitá di Torino, July 1996.

[GHHR96] S. Gilmore, J. Hillston, D.R.W. Holton, and M. Rettelbach. Specifications in Stochastic Process Algebra for a Robot Control Problem. *International Journal of Production Research*, 34(4):1065–1080, 1996.

[Hil96a] J. Hillston. A Class of PEPA Models Exhibiting Product Form Solution over Submodels. Technical report, Department of Computer Science, The University of Edinburgh, 1996.

[Hil96b] J. Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, 1996.

[Hol95] D.R.W. Holton. A PEPA specification of an industrial production cell. In S. Gilmore and J. Hillston, editors, *Proceedings of the Third International Workshop on Process Algebras and Performance Modelling*, pages 542–551. Special Issue of *The Computer Journal*, 38(7), December 1995.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

# A Summary of the translation

## A.1 Translation of task bodies

$$\frac{\mathsf{ds}\,P = \{\ P\ \} \qquad P \stackrel{\mathit{def}}{=} R \qquad R \rightsquigarrow \widehat{R}}{P \rightsquigarrow \textbf{loop}\ \widehat{R}\ \textbf{end loop};}$$

$$\frac{\mathsf{ds}\,P = \{\ P_i \mid 1 \le i \le n\ \} \qquad P_i \stackrel{\mathit{def}}{=} R_i \qquad R_i \rightsquigarrow \widehat{R_i}}{\begin{array}{lll} P & \rightsquigarrow & \textbf{loop} \\ & & \quad \textbf{case}\ \mathsf{state}\ \textbf{is} \\ & & \quad\quad [\textbf{when}\ P_i \Rightarrow \widehat{R_i}]_{i=1}^{n} \\ & & \quad \textbf{end case}; \\ & & \textbf{end loop}; \end{array}}$$

$P, P_i$ identifiers; $R, R_i$ expressions; $\widehat{R}, \widehat{R_i}$ statements.

## A.2 Translation of simple terms

$$\frac{}{P \rightsquigarrow \mathsf{state} := P;} \qquad \frac{S \rightsquigarrow \widehat{S}}{(\alpha, r).S \rightsquigarrow \alpha(r);\ \widehat{S}}$$

$$\frac{S \rightsquigarrow \widehat{S}}{(\tau, r).S \rightsquigarrow \textbf{delay}\ (1.0/r);\ \widehat{S}} \qquad \frac{S \rightsquigarrow \widehat{S}}{\begin{array}{lll} (\boldsymbol{\alpha}, r).S & \rightsquigarrow & \textbf{accept}\ \boldsymbol{\alpha}\ \textbf{do} \\ & & \quad \textbf{delay}\ (1.0/r); \\ & & \textbf{end}\ \boldsymbol{\alpha};\ \widehat{S} \end{array}}$$

$$\frac{S \rightsquigarrow \widehat{S} \qquad \mathsf{recip}\,\boldsymbol{\alpha} = \{\ P\ \}}{(\boldsymbol{\alpha}, \top).S \rightsquigarrow P.\boldsymbol{\alpha};\ \widehat{S}} \qquad \frac{S \rightsquigarrow \widehat{S} \qquad \mathsf{recip}\,\boldsymbol{\alpha} = \{\ P_i \mid 1 \le i \le n\ \}}{\begin{array}{lll} (\boldsymbol{\alpha}, \top).S & \rightsquigarrow & \textbf{loop} \\ & & \quad \textbf{select} \\ & & \quad\quad P_1.\boldsymbol{\alpha};\ \textbf{exit}; \\ & & \quad \textbf{or} \cdots \textbf{or} \\ & & \quad\quad P_n.\boldsymbol{\alpha};\ \textbf{exit}; \\ & & \quad \textbf{else} \\ & & \quad\quad \textbf{null}; \\ & & \quad \textbf{end select}; \\ & & \textbf{end loop};\ \widehat{S} \end{array}}$$

$P, P_i$ identifiers; $S$ an expression; $\widehat{S}$ a statement sequence.

## A.3 Translation of choices

$$\frac{R_i \rightsquigarrow \widehat{R_i}}{\begin{array}{lll}\sum_{i=1}^{k}(\boldsymbol{\alpha}_i, r_i).R_i & \rightsquigarrow & \textsf{select} \\ & & \quad \textsf{accept}\ \boldsymbol{\alpha}_1\ \textsf{do}\ \ \textsf{delay}(1.0/r_1);\ \textsf{end}\ \boldsymbol{\alpha}_1;\ \widehat{R_1} \\ & & \textsf{or}\ \cdots\ \textsf{or} \\ & & \quad \textsf{accept}\ \boldsymbol{\alpha}_k\ \textsf{do}\ \ \textsf{delay}(1.0/r_k);\ \textsf{end}\ \boldsymbol{\alpha}_k;\ \widehat{R_k} \\ & & \textsf{end select};\end{array}}$$

$$\frac{S_i \rightsquigarrow \widehat{S_i} \qquad \textsf{recip}\,\boldsymbol{\alpha}_i = \vec{P_i}}{\begin{array}{lll}\sum_{i=1}^{l}(\boldsymbol{\alpha}_i, \top).S_i & \rightsquigarrow & \textsf{loop} \\ & & \quad \textsf{select} \\ & & \qquad P_{1_1}.\boldsymbol{\alpha}_1;\ \widehat{S_1}\ \textsf{exit};\ \textsf{or}\ P_{1_2}.\boldsymbol{\alpha}_1;\ \widehat{S_1}\ \textsf{exit};\ \textsf{or}\ \cdots \\ & & \quad \textsf{or}\ \cdots\ \textsf{or} \\ & & \qquad P_{l_1}.\boldsymbol{\alpha}_l;\ \widehat{S_l}\ \textsf{exit};\ \textsf{or}\ P_{l_2}.\boldsymbol{\alpha}_l;\ \widehat{S_l}\ \textsf{exit};\ \textsf{or}\ \cdots \\ & & \quad \textsf{end select}; \\ & & \textsf{end loop};\end{array}}$$

$$\frac{T_i \rightsquigarrow \widehat{T_i}}{\begin{array}{lll}\sum_{i=1}^{m}(\alpha_i, t_i).T_i & \rightsquigarrow & \textsf{declare A : rate\_list}(1..m) := (t_1, \ldots, t_m); \\ & & \textsf{begin} \\ & & \quad \textsf{case psrf (A) is} \\ & & \qquad [\textsf{when}\ i \Rightarrow \alpha_i\,(t_i);\ \widehat{T_i}]_{i=1}^{m} \\ & & \qquad \textsf{when others} \Rightarrow \textsf{null}; \\ & & \quad \textsf{end case}; \\ & & \textsf{end};\end{array}}$$

$\boldsymbol{\alpha}_i$ distinct; $R_i, S_i, T_i$ expressions; $\widehat{R_i}, \widehat{S_i}, \widehat{T_i}$ statements

The symbol $\rightsquigarrow$ is used to signify translation. We often write $P \rightsquigarrow \widehat{P}$, reserving the hat symbol as a decoration for terms which have been produced by translation. Occasionally terms are passed from the stochastic process algebra model to the programming language representation unchanged. When this occurs the terms are always only identifiers.

Several utility functions are used within these rules. Definitions of these appear in the paper [GHH96] but we give brief explanations here for the sake of completeness. The function ds computes the derivative set of a component. These are the derivatives which the component can evolve to through a series of activities. The function recip computes the recipients of an activity as determined by analysis of scope and interface information.