From SPA models to programs

Stephen Gilmore*, Jane Hillston* and Robert Holton†

* Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh, Edinburgh EH9 3JZ, Scotland, UK.

† Department of Computing, University of Bradford, Bradford BD7 1DP, England, UK.

Abstract

Stochastic process algebra models allow desired performance characteristics to be assessed during the design of complex software systems. We present a software development method which aims to ensure that these performance characteristics are attained by an implementation. At the heart of the development method is a translation from a process algebra model into a program skeleton.

1 Introduction

Stochastic process algebras (SPAs) are used to investigate the performance of systems which are composed of concurrently active communicating components. In this paper we consider the use of SPA models as vehicles for analysing and assessing alternative candidate designs for a software system which is to be constructed. Our contribution here is to propose a systematic method of advancing the construction of the system software once an initial SPA model has been selected as a suitable starting point. The aim of our method is to ensure that the performance investigation which has been conducted on that model will serve as a legitimate investigation of the software system under construction. It is our belief that this method is best viewed as specific to the application domain of control systems where sub-tasks have been distributed across loosely-coupled processors. We hold this view because such systems typically contain non-stop processes with periodic behaviour which is tightly constrained to be completed within a fixed time frame and these features make them well suited to our method. Examples of this type of system include automated production and manufacturing systems; controllers for automated bank teller machines; or components in an operating system kernel.

The method which we present is a generic one which could be applied to any of the family of recently developed stochastic process algebras which include PEPA [1], TIPP [2] and EMPA [3]. Similarly, a variety of programming languages could be used as the system implementation language. These include Ada [4], Java [5], occam [6] and parallel variants of C. For the purpose of illustrating the method we have chosen to use PEPA and Ada in this paper.

In our program development method PEPA is used for three purposes:

1. designing the concurrent structure of the system at the level of large-scale tasks;
2. developing patterns of independent behaviour and synchronisation requirements; and
3. setting the level of performance that must be attained.

Supplementary formal notations are used to craft the designs and step-wise refinement of the sequential code fragments occurring within task bodies. Examples of suitable formal notations to be used here include Z [7], the Refinement Calculus [8] and VDM [9]. We view this distinction between the treatment of concurrent code and sequential code as an application of Dijkstra's advice to seek a clear separation of concerns at an early stage in program development.

1.1 Structure of this paper

We begin by explaining our program development method in greater detail. This leads us to consider carefully slight restrictions on the SPA which is used and to identify a subset of the target implementation language which has desirable properties for our purposes. In order that our paper should be self contained, we present an introduction to the PEPA stochastic process algebra and the Ada programming language. We go on to explain alternative possible approaches to the translation from PEPA models to Ada programs and then to give a formal definition of a translation from PEPA constructs to this subset of Ada. We first present the static analysis which is done in order to guide the translation and then present the translation itself. We then comment on our implementation of this translation. We conclude by identifying a number of extensions which can be envisaged for this work.

2 The program development method

It has long been recognised that it is desirable to integrate performance modelling into the design process for large software systems. For maximum benefit to be obtained from this, the performance aspects should be considered from the conception of the system and thus performance information naturally forms a part of a complete initial specification. The introduction of stochastic process algebras represents a significant step towards this goal by providing the means for design and performance modelling to be carried out in the same notation. However, use of these novel techniques has highlighted a difference in the level of detail required for the objectives of performance analysis and behavioural specification [10, 11]. Data can be, and often is, abstracted in performance models since the timing differences associated with different data values is generally negligible. However, in order to study behavioural correctness data values are clearly of considerable significance: although they might have a secondary status as with classical process algebras such as CCS [12]. For example, a program may have satisfactory performance characteristics but incorrect behaviour because data is corrupted because of an error in the program. Although both behavioural and performance characteristics must be verified for dependable systems, including information about the data during performance analysis can make the models unwieldy [10].

In this paper we propose a disciplined approach to establishing the performance characteristics of the proposed system before systematically introducing the additional information required for the behavioural specification. The performance characteristics identified in the initial SPA model remain to guide the extension of the basic model with the necessary data handling features such as assignment and calculation which have been previously been treated as atomic activities. This detailed sequential behaviour can be formally verified using existing formal refinement techniques. Furthermore the elaboration of the design can continue through to the complete implementation.

Although the notation used is PEPA, we focus upon a subset of the language where we only allow one active component in any synchronisation. One benefit of this restriction is that all of the recently developed SPA languages can represent this form of synchronisation. The immediate advantage of this for our purposes is that this restricted form of active/passive co-operation corresponds directly to the idea of a remote procedure call. As a further simplification here, we restrict communication between concurrently active components to be limited to a maximum of two partners in each communication. Thus the model of communication which we use is closer to that in Milner's CCS [12] than Hoare's CSP [13] but we do not adopt the device that synchronisation activity is hidden by being modified into a silent τ action [as it is in CCS, specifically to prevent multi-way synchronisation]. If PEPA model developers wish to check mechanically that their models do not rely on the use of multi-way synchronisation then they may use co-operation together with hiding where necessary to prevent multi-way synchronisation.

We do not propose that the PEPA modeller should be restricted to working always within the subset of the modelling language which we have just outlined. Instead we suggest that all of the language may be used for initial system models but that before program implementation commences the modeller should re-work the model so that it uses only binary synchronisations with one partner active. Equivalence relations on models [1] should be used to ensure that this re-working of the model does not change its behaviour.

At the heart of the program development method is a translation from the PEPA to the target language. The translation proceeds in a top-down manner by first considering top-level components, then successively translating their sub-components until the level of single activities is reached. Ultimately these single activities will be fleshed out into—perhaps long—sequences of instructions which perform calculations or read and write data. For the present all of this is omitted and instead the program contains delays which represent the intended timing behaviour of the activities within the system. Thus the result disregards the processing of data but reflects the concurrency and the behaviour of the PEPA model. The subsequent refinement of the program will make this aspect of the behaviour concrete.

2.1 Benefits of the method

The expected benefits of this program development method include the following.

- ▷ The initial investigation and subsequent analysis using PEPA has already highlighted those code sections where efficient implementation is of paramount importance while simultaneously identifying those sections where it is not. This has the benefit of guiding the application programmer to expend effort on code optimisations in the areas where the benefit will be maximised and providing less finely tuned implementations of code sections which are not time critical.
- ▷ This approach may be used to supplement one of the well-known methods for refinement of abstract specifications to efficiently executable sequential code fragments in such a way that these fragments can be embedded within a concurrent program structure.

▷ It is possible to check that changes to the concurrent structure of the program do not lead to violations of the timing requirements stipulated within the original PEPA description of the system by translating the program back to another PEPA model and performing further analysis to compare these two models.

3 Languages for modelling and programming

Stochastic process algebras are flexible and expressive modelling languages which have the virtue of being compact. They include only a small number of combinators [typically prefix, choice, hiding and composition] and some mechanism for recursive definition. Conflict between possible actions is resolved by a race condition and so, in effect, choices are weighted by probabilities which correspond to the activity rates of the associated actions.

Programming languages for concurrent and distributed systems are very different from SPA languages. Such programming languages place many restrictions on the way in which programs can be constructed and thus constrain the expressiveness of the language. In addition they are often far from compact—the Ada language is certainly an example of this—and furthermore their behaviour is not often governed entirely by probability and race conditions; non-determinacy may also play a part.

We have chosen to map PEPA models to Ada programs rather than design our own programming language, perhaps closer to the model of systems promoted by the SPA approach. Although having a purpose-built programming language would greatly simplify the program development process and consequently reduce the potential for error, it would also significantly reduce the credibility of our program development approach with regard to its potential applicability for the development of genuine control components of computer systems. One of the strongest motivations for us to choose Ada is the knowledge that it is widely used in practice as an implementation language for the kinds of systems which we are considering.

In the rest of this section we will briefly summarise PEPA and Ada and then discuss some approaches to the translation of PEPA models to Ada program skeletons.

3.1 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, τ. 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 (α, r) where α is the action type and r is the activity rate.
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

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

Figure 1 Grammar for the PEPA subset 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 [1]. 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 [1] 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 [14].

3.2 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 Approaches to translation

There appears to be a direct match between components in a PEPA model and tasks in an Ada program, and between the co-operation of components within the model and the rendezvous of tasks in the program. However, even given this correspondence it is not immediately obvious how to go about the translation. Consider the following three approaches.

Approach 1: Perhaps the most straightforward translation would involve mapping each derivative to a task in the Ada program, so that a component is represented by the collection of tasks representing its derivatives. The evolution of a PEPA component into one of its derivatives is represented by one task suspending itself and signaling another to proceed. Activities are represented as entry calls.

The fundamental problem with this approach is that the relationship between the derivatives which constitute a single sequential component is lost, and thus the Ada program does not have a structure which corresponds to the modular structure of the model. A practical problem with this approach is that it will produce a larger number of tasks than necessary and at any given time during the execution of the program many of these tasks will be lying idle, awaiting re-activation.

Approach 2: Alternatively, the derivatives within a sequential component could be mapped to mutually recursive procedures, completely analogous with the mutually recursive defining equations. These procedures would then be grouped into a single task corresponding to the model component defined by the derivatives. In this representation activities are represented by entry calls or procedure calls, and evolving to the next derivative is represented as a call to the associated procedure.

Here the causality and competitive relationships between the derivatives of a sequential component are reflected by analogous relationships between the procedures within a single task. Unfortunately such a direct translation is unsuccessful for Ada because it leads to the requirement to accept entry calls from within procedure bodies. This is not allowed in Ada—**accept** statements may only appear in the main body of the task.

Approach 3: Finally, we can again associate tasks with model components but rather than represent derivatives as active entities within the program we record the current derivative by setting a state variable to the appropriate value. This value identifies the entry calls which will currently be accepted, just as the derivative identifies the current activities in the PEPA model. Moving to the next derivative is now represented by assignment to the state variable. As above, procedures are used to represent individual activities and entry calls are used to represent shared activities.

As previously, this approach has the advantage that the behaviour corresponding to a sequential component is collected together within a single task. Thus the program reflects the structure of the model. Moreover the causality and competition relationships between derivatives are explicitly represented. The simple representation of a derivative as the value of an assignable variable is another advantage of this approach.

We should emphasise that at this stage the program has no algorithmic content, it only consists of structure and delays. In particular, the work performed during an activity, which subsequent implementation work will refine into statement blocks, is currently represented simply as a delay. The co-operation of two components to complete an activity is a rendezvous in which the passive component calls the active component. This is natural if we consider the passive component to be the one without knowledge of how the activity is to be achieved. Choice corresponds to a selection between alternatives which is captured in a variety of ways in the program depending on the initial activities enabled by the choice. Hiding can be thought of as a scoping operator and in the Ada program this corresponds to defining a task interface which limits the scope of variables.

We will present the translation more thoroughly in Section 6. In the following section we first discuss the preliminary analysis of the PEPA model which takes place prior to translation.

5 Static model analysis

The translation of a component of a model into a programming language representation depends upon an initial static investigation of the role of the component within the model. We describe this in terms of a system model, such as $(P \parallel P) \underset{L}{\bowtie} (Q \parallel Q)$, together with a collection of equations which define the behaviour of the components, in this case P and Q. The fundamental objective of the static analysis is the identification of those distinguished components which will become Ada tasks, and we explain this first.

5.1 Identification of tasks

Consider a process algebra system expression which is a composition of component identifiers such as P and Q and compound expressions such as R and S. In Figure 2 we define two functions, tasks and $\mathsf{tasktypes}$. When given a process algebra expression the tasks function will form a multiset of the identifiers of components which should be realised as tasks in the program. The symbol $\uplus$ used in the definition denotes multiset union. The $\mathsf{tasktypes}$ function selects from this the components to be formed into task types in order that multiple instances of these may be declared.

This translation has the benefit of being very direct and simple to implement. However, later we will point out that it has a slight shortcoming in that it can generate tasks in some cases where it might be thought that it should generate task types.

$$\begin{array}{llll}
\mathsf{tasks}\,(a.R) & = \emptyset & \mathsf{tasks}\,(R+S) & = \emptyset \\
\mathsf{tasks}\,(R/L) & = \mathsf{tasks}\,R & \mathsf{tasks}\,(P/L) & = \{P\} \\
\mathsf{tasks}\,(R \underset{L}{\bowtie} S) & = \mathsf{tasks}\,R \uplus \mathsf{tasks}\,S & \mathsf{tasks}\,(R \underset{L}{\bowtie} Q) & = \mathsf{tasks}\,R \uplus \{Q\} \\
\mathsf{tasks}\,(P \underset{L}{\bowtie} S) & = \{P\} \uplus \mathsf{tasks}\,S & \mathsf{tasks}\,(P \underset{L}{\bowtie} Q) & = \{P, Q\}
\end{array}$$

$$\mathsf{tasktypes}\,S = \text{identifiers with multiplicity two or more in } \mathsf{tasks}\,S$$

$$P, Q \text{ identifiers; } R, S \text{ expressions.}$$

Figure 2 Functions for identifying tasks

5.2 Calculation of derivatives

Once tasks have been identified it is necessary to consider the states which these tasks will cycle through. These correspond to the derivatives of the model component which the task represents and so the names of derivatives of components must also be recorded. This function is elementary and its definition is omitted.

5.3 Calculating alphabets

We inspect the components in the SPA model in order to determine their *alphabet* [13]. Our use of this term coincides with Hoare's definition as the set of [activity] names of events which are considered relevant for a particular component. In all of the SPA languages this alphabet does not need to be predeclared; if it is needed then it can be calculated from the model. In contrast, secure programming languages insist upon names being introduced before they are used and our preliminary inspection of our PEPA model must identify these names and categorise them in terms of their eventual use. For our purposes later we need to distinguish between the set of activities which are performed actively and the set of activities which are performed passively. The union of these two sets is the alphabet of the component. The three functions which are defined in Figure 3 will calculate these sets.

5.4 Task specifications

The work performed in the identification of tasks and the calculation of alphabets for components is used when calculating the public interface of a task, called its *specification*. The factors which determine the specification for a task are the activities in which it is an active participant, moderated by the influences of co-operation and hiding. Activities such as these will become either procedures within the task, and hence private, or entries into the task, which are public and are listed in the task interface. The influence of co-operation upon this decision is to tend to add activities to the task specification whereas the effect of hiding is to prohibit the addition of activities.

$$\begin{array}{lll@{\qquad}lll}
\mathsf{act}\,((\alpha,r).R) & = & \{\alpha\} \cup \mathsf{act}\,R & \mathsf{pass}\,((\alpha,r).R) & = & \mathsf{pass}\,R \\
\mathsf{act}\,((\alpha,\top).R) & = & \mathsf{act}\,R & \mathsf{pass}\,((\alpha,\top).R) & = & \{\alpha\} \cup \mathsf{pass}\,R \\
\mathsf{act}\,(R+Q) & = & \mathsf{act}\,R \cup \mathsf{act}\,Q & \mathsf{pass}\,(R+Q) & = & \mathsf{pass}\,R \cup \mathsf{pass}\,Q \\
\mathsf{act}\,(R/L) & = & \mathsf{act}\,R \setminus L & \mathsf{pass}\,(R/L) & = & \mathsf{pass}\,R \setminus L \\
\mathsf{act}\,(R \underset{L}{\bowtie} Q) & = & \mathsf{act}\,R \cup \mathsf{act}\,Q & \mathsf{pass}\,(R \underset{L}{\bowtie} Q) & = & \mathsf{pass}\,R \cup \mathsf{pass}\,Q
\end{array}$$

$$\mathsf{alph}\,R = \mathsf{act}\,R \cup \mathsf{pass}\,R$$

R, S expressions.

Figure 3 Functions for computing alphabets of components

The function named spec, defined in Figure 4, is given a distinguished component; a process algebra system expression in which the distinguished component occurs; and a set which is used to record the activities which are performed in co-operation with other components. From these three inputs the spec function calculates the task specification for the component. An advantage of the presentation of the definition which we use is that it avoids the definition of an auxiliary function to determine whether or not a component occurs as a subcomponent in an expression. For completeness the definition of the spec function is presented in full in Figure 4 although the reader will doubtless observe that two pairs of cases in the definition are symmetric. This arises from the inherent symmetry of the co-operation operator.

$$\begin{array}{lll}
\mathsf{spec}\,P\,(a.R)\,K & = & \emptyset \\
\mathsf{spec}\,P\,(R+S)\,K & = & \emptyset \\
\mathsf{spec}\,P\,(P/L)\,K & = & ((K \setminus L) \cap \mathsf{act}\,P) \\
\mathsf{spec}\,P\,(Q/L)\,K & = & \emptyset \\
\mathsf{spec}\,P\,(R/L)\,K & = & \mathsf{spec}\,P\,R\,(K \setminus L) \\
\mathsf{spec}\,P\,(P \underset{L}{\bowtie} S)\,K & = & (L \cap \mathsf{act}\,P) \cup (K \cap \mathsf{act}\,P) \cup \mathsf{spec}\,P\,S\,(K \cup L) \\
\mathsf{spec}\,P\,(S \underset{L}{\bowtie} P)\,K & = & (L \cap \mathsf{act}\,P) \cup (K \cap \mathsf{act}\,P) \cup \mathsf{spec}\,P\,S\,(K \cup L) \\
\mathsf{spec}\,P\,(Q \underset{L}{\bowtie} S)\,K & = & \mathsf{spec}\,P\,S\,(K \cup L) \\
\mathsf{spec}\,P\,(S \underset{L}{\bowtie} Q)\,K & = & \mathsf{spec}\,P\,S\,(K \cup L) \\
\mathsf{spec}\,P\,(R \underset{L}{\bowtie} S)\,K & = & \mathsf{spec}\,P\,R\,(K \cup L) \cup \mathsf{spec}\,P\,S\,(K \cup L)
\end{array}$$

P, Q distinct identifiers; R, S expressions; K, L sets

Figure 4 The definition of the interface for P

5.5 Task instances

In a PEPA system model, as with other process algebra languages, components are replicated simply by the use of repeated occurrences of their identifier within the system expression. In a programming language such as Ada, the programmer must explicitly declare replicated copies of a task. Each copy of the task must have an identifier and, if the task instances are being declared at the same level of scope, then all of these identifiers must be distinct. It follows that there is a small problem for the translation to generate these distinct identifiers. These can be simple variations of the model component identifier, say with an integer suffix appended, taking care to ensure that this produces a fresh identifier.

5.6 Communication between tasks

Another way in which the compact notation of process algebras abbreviates programming language constructs is found in the notation for co-operation between components. In a process algebra model the component which takes the passive role in the co-operation does not name the component which takes the active role; this is simply determined from inspection of the use of co-operation and hiding in the model. The reader should note that—even although here all co-operations are restricted to be between exactly two partners—there may be several candidates for the role of the active participant in the co-operation.

In Figure 6 we define the function which determines the communication between tasks in the system. This relies upon a subsidiary definition of connections between tasks, presented in Figure 5.

$$
\begin{array}{l}
\mathsf{connect}\,(P, Q, L) = \\
\quad \{\ (p, q, \mathit{calls}_p \cap \mathit{entries}_q \cap L), (q, p, \mathit{calls}_q \cap \mathit{entries}_p \cap L) \mid \\
\qquad (\mathit{entries}_p, p, \mathit{calls}_p) \in P, (\mathit{entries}_q, q, \mathit{calls}_q) \in Q\ \}
\end{array}
$$

P, Q sets of entry, task and call identifiers; L a co-operation set

Figure 5 Calculating connections between sets of tasks

$$
\begin{array}{rcl}
\mathsf{comms}\,\mathit{Sys}\,P & = & (\{\ (\mathsf{spec}\,P\,\mathit{Sys}\,\emptyset, P, \mathsf{pass}\,P)\ \}, \emptyset) \\
\mathsf{comms}\,\mathit{Sys}\,(S/L) & = & \text{let } \mathsf{comms}\,\mathit{Sys}\,S = (P, c) \\
 & & \text{in } (\{\ (e_p \setminus L, p, c_p \setminus L) \mid (e_p, p, c_p) \in P\ \}, c) \\
\mathsf{comms}\,\mathit{Sys}\,(R \bowtie_L S) & = & \text{let } \mathsf{comms}\,\mathit{Sys}\,R = (P, c_P) \\
 & & \text{and } \mathsf{comms}\,\mathit{Sys}\,S = (Q, c_Q) \\
 & & \text{in } (P \cup Q, c_P \cup c_Q \cup \mathsf{connect}\,(P, Q, L))
\end{array}
$$

P a task identifier; R, S, Sys expressions; L a co-operation set

Figure 6 Communication between tasks in a system

6 Translation

We define our translation from a PEPA model into the first program of a sequence of refinements towards the final system implementation. Our translation proceeds from the component definition into the iterative and conditional behaviour which is to be found inside task bodies. Our previous static model analysis has identified the tasks, task types and task variables and procedures and state variables which are to be declared in the Ada program text. We will omit the generation of these declarations here because they are straightforward and we will proceed with the rules which govern the generation of the statements which appear in the task bodies.

The symbol $\rightsquigarrow$ is used to signify the translation operation; we often write $P \rightsquigarrow \widehat{P}$, reserving the hat symbol as a decoration for terms which have been produced by translation.

6.1 Translation of task bodies

The rules which generate the bodies of the tasks in the program are given in Figure 7. The translation isolates a special case where the component has only one named derivative which, because the component is cyclic, must be itself. This corresponds to a very simple implementation which is a loop which contains the translation of the expression to which the derivative was bound. This translation involves the use of a function, ds, to determine the derivatives of the component and the use of its definition to obtain the defining expression.

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

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

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

Figure 7 Translation of task bodies

In the more general case a component will have a set of named derivatives. The names of these derivatives are used as the names of the values which the state variable can have and the translation of the loop body includes a case statement which switches to the appropriate behaviour for the appropriate derivative. Once again, the function to compute the derivative set of a component is invoked and the defining equations for the derivatives are consulted.

6.2 Translation of simple terms

The translation of simple process algebra terms is presented in Figure 8. This includes

$$\overline{P \rightsquigarrow \mathsf{state} := P;}$$

$$\frac{S \rightsquigarrow \widehat{S}}{(\alpha, r).S \rightsquigarrow \alpha(r);\ \widehat{S}} \qquad \frac{S \rightsquigarrow \widehat{S}}{(\tau, r).S \rightsquigarrow \textbf{delay}\,(1.0/r);\ \widehat{S}}$$

$$\frac{S \rightsquigarrow \widehat{S}}{(\boldsymbol{\alpha}, r).S \rightsquigarrow \begin{array}{l}\textbf{accept}\ \boldsymbol{\alpha}\ \textbf{do} \\ \quad \textbf{delay}\,(1.0/r); \\ \textbf{end}\ \boldsymbol{\alpha};\ \widehat{S}\end{array}} \qquad \frac{S \rightsquigarrow \widehat{S} \qquad \mathsf{recip}\,\boldsymbol{\alpha} = \{\ P\ \}}{(\boldsymbol{\alpha}, \top).S \rightsquigarrow P.\boldsymbol{\alpha};\ \widehat{S}}$$

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

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

Figure 8 Translation of simple terms the translation of prefixes and uses of the identifiers of other components which denote the evolution of the term in which the use occurs into the named component. In the program which is produced this change of behaviour is brought about by an assignment to the state variable within a task body. When the component which is being translated has only a single derivative then the assignment to the state variable can be elided.

The translation of prefixes varies in difficulty from being a direct representation of the process algebra prefix term to requiring a substantial amount of translation effort. Internal actions which are performed at a given rate are translated into a call to a local procedure with the rate as the actual parameter for the call. The bodies of these procedures simply contain a **delay** statement which also is the translation of a τ action, consistent with its interpretation as an activity of unknown name. The translation of an offer to take the active part in a shared activity becomes an Ada **accept** statement.

The final two rules concern the passive partner in a shared activity. These use a function named recip which identifies the recipients of an entry call of the given type. The recip function can be succinctly defined via the comms function which was presented in Figure 6. The definition of the recip function is omitted here. As before, it is useful to isolate a special case which occurs frequently. When the recipient of an entry call can be uniquely identified then the translation is to a single call and does not require the polling loop which must be used in the more general case.

6.3 Translation of choices

When a process algebra term takes the form of a choice between possible alternative next actions we approach its translation by first rearranging it into a standard form as shown below.

$$\sum_{i=1}^{k}(\boldsymbol{\alpha}_i, r_i).R_i + \sum_{i=1}^{l}(\boldsymbol{\alpha}_i, \top).S_i + \sum_{i=1}^{m}(\alpha_i, t_i).T_i$$

The sum has been commuted into three summations. The first summation includes those terms where the first activity represents active participation in a shared activity. The second summation is for passive participation in shared activities. The final summation has individual activities. In order to simplify the presentation here we will include τ actions in this third summation. To further simplify the presentation we assume that the $\boldsymbol{\alpha}_i$ are distinct. The necessity for this restriction is explained in Section 7.2.

The translation of a sum such as the one above can be broken down using two conditional **select** statements. We give the translation of individual summations separately in Figure 9 in order to suggest how a general sum is translated.

A choice between shared activities in which the term which is being translated determines the rate at which the activities are carried out will be translated with the use of the concurrency constructs of the programming language. In this case the task makes a selection between the requests for entry.

A choice between shared activities in which the term which is being translated does not determine the rate at which the activities are carried out will again be translated with the uses of the concurrency constructs of the programming language. In this case however the translation is complicated significantly because there may be a number of possible recipients for each of the possible entry calls which might take place. We represent the collection of recipients by a vector for each activity, determined once again by the recip function.

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

$$\frac{S_i \leadsto \widehat{S}_i \qquad \mathsf{recip}\,\boldsymbol{\alpha}_i = \vec{P}_i}{\sum_{i=1}^{l}(\boldsymbol{\alpha}_i, \top).S_i \;\leadsto\; \begin{array}{l}\textbf{loop}\\ \quad \textbf{select}\\ \quad\quad P_{1_1}.\boldsymbol{\alpha}_1;\ \widehat{S}_1 \textbf{ exit};\ \textbf{or } P_{1_2}.\boldsymbol{\alpha}_1;\ \widehat{S}_1 \textbf{ exit};\ \textbf{or } \cdots\\ \quad \textbf{or} \ \cdots \ \textbf{or}\\ \quad\quad P_{l_1}.\boldsymbol{\alpha}_l;\ \widehat{S}_l \textbf{ exit};\ \textbf{or } P_{l_2}.\boldsymbol{\alpha}_l;\ \widehat{S}_l \textbf{ exit};\ \textbf{or } \cdots\\ \quad \textbf{end select};\\ \textbf{end loop};\end{array}}$$

$$\frac{T_i \leadsto \widehat{T}_i}{\sum_{i=1}^{m}(\alpha_i, t_i).T_i \;\leadsto\; \begin{array}{l}\textbf{declare } \mathsf{A} : \mathsf{rate\_list}(1..m) := (t_1, \ldots, t_m);\\ \textbf{begin}\\ \quad \textbf{case } \mathsf{psrf}\ (\mathsf{A}) \textbf{ is}\\ \quad\quad [\textbf{when } i \Rightarrow \alpha_i\,(t_i);\ \widehat{T}_i]_{i=1}^{m}\\ \quad\quad \textbf{when others} \Rightarrow \textbf{null};\\ \quad \textbf{end case};\\ \textbf{end};\end{array}}$$

R_i, S_i, T_i expressions; $\widehat{R_i}, \widehat{S}_i, \widehat{T}_i$ statements

Figure 9 Translation of choices

A choice between a collection of individual activities α_i performed at rates t_i will correspond to a selection between operations based upon the internal state of the task at the time when the choice must be made. In the PEPA model we have no information about what this choice is or how it is made and so we can only design a conditional statement which will select each of the possibilities on the basis of the outcome of a weighted pseudo-random choice. We deploy a function named `psrf` in the relevant translation.

7 Problem cases for the translation

In this section we consider circumstances where our translation might seem to give rise to a program which would have behaviour which differs from the behaviour which would be seen in the PEPA model. The first problem we consider is an apparent starvation effect.

7.1 Starvation

Consider the following small example. The expected behaviour of this system is to sometimes undertake $\boldsymbol{\alpha}$ activities, and sometimes β activities. The relative probability of these alternatives can be found from the rates r and s. Our translation will produce a program in which two tasks execute in parallel as shown in Figure 10.

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

Given that the $\boldsymbol{\alpha}$ activity is always available, the β activity will never be performed since this **select** statement only chooses the **else** alternative if there are no calls waiting to enter the task. However, pathological components such as Q which perform no other action will be unlikely to form finished components of a practical, working system. Instead components such as Q would typically have other internal behaviour which is not represented in the PEPA model. In cases such as these the apparent starvation problem does not arise.

7.2 Internal choice

A special case which requires special attention arises when a component can at some point offer to perform the same activity at two different rates or with two different outcomes. An example of the former is $(\boldsymbol{\alpha}, r_1).P + (\boldsymbol{\alpha}, r_2).P$. An example of the latter is $(\boldsymbol{\alpha}, r).P_1 + (\boldsymbol{\alpha}, r).P_2$. This form of decision has previously been termed an 'internal choice' since—from the perspective of a co-operating [and hence passive] component—it seems as though a choice is being made with respect to the way in which an $\boldsymbol{\alpha}$ activity is performed. The influence of this on our translation is discussed in full in a companion paper [15].

7.3 Ambivalent activities

We wish to have activities translated into entries into tasks if they may be used in a synchronisation activity or translated into procedures within tasks if they cannot be so used. In particular we wish to avoid representing a single activity both by a task entry and by a procedure body in order to avoid the duplication of the activity name, perhaps with an attendant duplication of programming effort. Some PEPA terms would give us difficulties and thus we have designed the grammar of the PEPA used here so that a term such as the one shown below cannot occur.

$$(\boldsymbol{\alpha}, r).(((\boldsymbol{\alpha}, r).P)/\boldsymbol{\alpha})$$

In this term the first occurring $\boldsymbol{\alpha}$ is visible outside and thus is available for synchronisation whereas the second occurrence of $\boldsymbol{\alpha}$ is hidden, and thus considered to be internal to the task where this occurs. A translation of this term which represented $\boldsymbol{\alpha}$ as a task entry would perhaps have the second, unavailable $\boldsymbol{\alpha}$ protected by an always false guard. Although this translation does not replicate the second use of the activity name it is still unsuitable because it gives rise to unreachable program statements, so-called 'dead code'.

An alternative approach to treating an expression such as this would be to use the semantics of PEPA to replace the term above by the following, equivalent one.

$$(\boldsymbol{\alpha}, r).(\tau, r).P$$

There is no difficulty with translating this expression although this form of the expression has lost the [perhaps significant] information that the τ action has been caused by undertaking an $\boldsymbol{\alpha}$ activity in isolation.

7.4 System initialisation

One shortcoming of our present translation is that in certain cases it will fail to recognise the possibility of generating a task type and creating several tasks which are instance variables of this type. The circumstance which will cause this is if two instances of the same model component are initiated as different derivatives. As an example, consider the system below.

$$\begin{array}{l} P \stackrel{\mathit{def}}{=} (\alpha, r).P + (\boldsymbol{\beta}, s).P' \\ P' \stackrel{\mathit{def}}{=} (\gamma, t).P' + (\boldsymbol{\beta}, s).P \\ \cdots \\ (P \parallel P') \underset{L}{\bowtie} \cdots \end{array}$$

The derivatives P and P' will be interpreted as two tasks, not as two instances of a task type, initialised with different starting states. This could be solved by additional work to recognise that P and P' share a common derivative set and thereafter translating them as task type variables with an initialisation operation to set them to the suitable derivative. Our earlier choice of assignable variables as representations of states makes this possible.

8 Implementation

Our translator has been implemented as an extension to the PEPA Workbench [16], an analysis and solution tool for PEPA models. Both tools are implemented in Standard ML [17], a strongly-typed functional programming language which has a sophisticated module system. The translator tool accepts a PEPA model expressed in the concrete syntax of the PEPA Workbench and generates an Ada program text as output. As a convenience, the Ada program is also printed in the form of LaTeX source. This has the helpful side-effect that it will allow the model developer to detect at an early stage if they have used any of Ada's sixty-nine reserved words as an identifier in the PEPA model which was given as input to the translator. It is quite plausible to believe that a PEPA user might wish to use as an identifier the names **abort**, **access**, **delta**, **requeue**, **return**, **reverse**, **separate**, **terminate** or **use**, without knowing that all of these are reserved words in the Ada programming language, and so cannot of course be used as identifiers in an Ada program.

9 Conclusions and future work

The idea that the development of all of the software of a complete computer system will be conducted within a single programming language is becoming an increasingly out-dated one. Use of the program development method which is presented here guides the software developer into dividing a system into sub-components with a clearly defined and widely used method of communication [the remote procedure call]. Through this mechanism, we hope that we have introduced sufficient scope for flexibility that our translation method and the accompanying translator could in the future be extended to support the development of hybrid systems in which the communicating components are to be implemented in a variety of programming languages. This would be of considerable benefit because we believe that we could in principle extend our program development method to cope with the case where more than one of the components in a co-operation is playing an active role. A suitable translation for this case would probably require the use of light-weight threads. Within Ada this would make use of the task creation mechanism.

References

[1] J. Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, 1996.
[2] N. Götz, U. Herzog, and M. Rettelbach. Multiprocessor and Distributed System Design: The Integration of Functional Specification and Performance Analysis using Stochastic Process Algebras. In *Performance'93*, 1993.
[3] M. Bernardo, N. Busi, and R. Gorrieri. A distributed semantics for EMPA based on stochastic contextual nets. In Gilmore and Hillston [10], pages 492–509.
[4] J. Barnes. *Programming in Ada 95.* Addison-Wesley, 1996.
[5] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.
[6] INMOS Limited. *occam 2 reference manual.* Prentice-Hall, 1988.
[7] J.C.P. Woodcock. *Using Z: Specification, Proof and Refinement.* Prentice-Hall, 1996.

[8] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1994. Second edition.
[9] C.B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, 1990. Second edition.
[10] S. Gilmore and J. Hillston, editors. *Proceedings of the Third International Workshop on Process Algebras and Performance Modelling*. Special Issue of *The Computer Journal*, 38(7), December 1995.
[11] D.R.W. Holton. A PEPA specification of an industrial production cell. In Gilmore and Hillston [10], pages 542–551.
[12] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
[13] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
[14] 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.
[15] S. Gilmore and J. Hillston. Refining internal choice in PEPA models. In R. Pooley and J. Hillston, editors, *Proceedings of the Twelfth UK Performance Engineering Workshop*, pages 49–64. University of Edinburgh, September 1996.
[16] 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.
[17] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

```
-- Output from the PEPA-to-Ada translator [Version 0.013, 30-5-1996]
with TEXT_IO; use TEXT_IO;
with PSRF; use PSRF;
with RATES; use RATES;

procedure MAIN is

  task P is
    entry alpha;
  end P;
  task Q;

  task body P is
    procedure beta(r: in DURATION) is
    begin
      delay (1.0/r);
    end beta;
  begin
    loop
      -- accept else
      select
        accept alpha do
          delay (1.0/r);
        end alpha;
      else
        -- a procedure call
        beta(s);
      end select;
    end loop;
  end P;

  task body Q is
  begin
    loop
      P.alpha;
    end loop;
  end Q;

begin
  null;
end MAIN;
```

Figure 10 Output from the PEPA to Ada translator