Performance modelling with UML and stochastic process algebras

Catherine Canevet, Stephen Gilmore, Jane Hillston, Matthew Prowse and Perdita Stevens

Abstract: We describe a software toolset which allows UML modellers to annotate their models with performance information. An equivalent performance model is extracted from the UML, solved, and the results reflected back to the UML level. Used in this way, our toolset gives a high-level approach to software performance modelling where the benefits of the performance modelling process are achieved without significant additional notational burden.

---

[0]The authors are with the Laboratory for Foundations of Computer Science, The University of Edinburgh, Scotland, EH9 3JZ, UK.

Find this paper's LaTeX source, DVI, and PostScript versions on-line at `http://homepages.inf.ed.ac.uk/stg/UKPEW02/JOURNAL/`.

Date of this draft version: October 29, 2002

1 Introduction

The Unified Modelling Language (UML) [1] is an effective and popular notation which is used to capture high-level designs for software systems. However, one aspect of software system design which is not typically captured in a UML document is a record of the likely (or desired) rate of performance of the major activities of the system. When this information is not included in the initial UML description of a system it increases the likelihood that the performance of the software system being developed will not be considered until very late in the software development process. At this stage, errors in the design will be very costly to repair and will require significant re-engineering.

Our aims in this paper are twofold:

1. to show how UML models enhanced with performance information can be mapped onto an existing performance modelling notation, Performance Evaluation Process Algebra (PEPA) [2]; and
2. to show how the results of the performance analysis of the PEPA models which are produced by this process can be presented to the UML modeller in the terms of their model.

The work reported here forms an early part of the DEGAS project. DEGAS stands for Design Environment for Global ApplicationS, and the project's overall goal is to make sophisticated formal analysis techniques available to designers of global applications – that is, of sofware systems that run on wireless networks and may involve mobile code – in a way which is congruent with their normal way of working. Performance prediction has been identified as an important area where such formal analysis techniques might be able to make a significant contribution to the quicker design of better applications.

Designers in the DEGAS partner companies (Motorola and OMNYS) and throughout the industry have adopted UML as their main software design notation. UML is a diagrammatic notation for recording the design of systems, especially object-oriented software systems. A UML *model* is represented by a collection of diagrams describing parts of the system from different points of view; there are seven main diagram types. For example, there will typically be a *static structure diagram* (or *class diagram*) describing the classes and interfaces in the system and their static relationships (inheritance, dependency, etc.) State diagrams, a variant on Harel state charts, can be used to record the dynamic behaviour of particular classes. Interaction diagrams, such as sequence diagrams, are used to illustrate the way objects of different classes interact in a particular scenario. In this work we concentrate on state diagrams, which provide a behavioural description in automata-theoretic terms which is also familiar from process algebras and protocol specifications.

Our aim in working with UML in the performance modelling process is to introduce the benefits of performance analysis with process algebras without the complexities and conceptual challenges which are normally associated with formal description techniques such as these. To this end, we deploy the PEPA stochastic process algebra as an intermediate language in the performance analysis process. The UML modeller can compose models and solve these for performance results without needing to understand the PEPA language, its formal definition or even how their model is represented in PEPA. At the same time, we avoid requiring the designer to develop a model specifically for performance analysis; instead, we work directly with the UML model which is being developed for other purposes.

Structure of this paper: In the next section we introduce the PEPA stochastic process algebra. Section 3 discusses modelling with UML and PEPA. In Section 4 we describe how performance measures such as utilisation can be obtained directly from our results. In Section 5 we discuss the software architecture of our tool set and describe the four principal software packages which are involved. In Section 7 we present a simple example. Following this we present a larger case study in Section 8. In Section 9 we discuss related work on UML, PEPA and performance modelling. We present our conclusions in Section 10.

2 Performance Evaluation Process Algebra

PEPA extends classical process algebra with the capacity to assign rates to the activities which are described in an abstract model of a system. Taken together, the information about the rates of performance of activities and the definition of the outcome of performing an activity specify a stochastic process and thus PEPA is said to be a *stochastic process algebra*. The PEPA language has been applied as a modelling language for distributed computer and telecommunications systems such as mobile telephone systems and for components of flexible manufacturing systems such as robotic workcells.

The PEPA language provides a small set of combinators. These allow language terms to be constructed defining the behaviour of components, via the activities they undertake and the interactions between them. The syntax may be formally introduced by means of the grammar shown in Fig. 1. In the grammar S denotes a *sequential component* and P denotes a *model component* which executes in parallel. C stands for a constant which denotes either a sequential or a model component, as defined by a defining equation. C when subscripted with an S stands for constants which denote sequential components. The component combinators, together with their names and interpretations, are presented informally below: further information is in the appendix.

Prefix: The basic mechanism for describing the behaviour of

$$
\begin{array}{llr}
S & ::= & \text{(sequential components)} \\
 & (\alpha, r).S & \text{(prefix)} \\
 & \mid \; S + S & \text{(choice)} \\
 & \mid \; C_S & \text{(constant)} \\
 & & \\
P & ::= & \text{(model components)} \\
 & P \bowtie_L P & \text{(cooperation)} \\
 & \mid \; P/L & \text{(hiding)} \\
 & \mid \; C & \text{(constant)}
\end{array}
$$

Fig. 1: The syntax of PEPA

a system is to give a component a designated first action using the prefix combinator, denoted by a full stop. For example, the component $(\alpha, r).S$ carries out activity (α, r), which has action type α and an exponentially distributed duration with parameter r, and it subsequently behaves as S. Sequences of actions can be combined to build up a life cycle for a component.

Choice: The life cycle of a sequential component may be more complex than any behaviour which can be expressed using the prefix combinator alone. The choice combinator captures the possibility of competition or selection between different possible activities. The component $P+Q$ represents a system which may behave either as P or as Q. The activities of both P and Q are enabled. The first activity to complete distinguishes one of them: the other is discarded. The system will then behave as the derivative resulting from the evolution of the chosen component.

Constant: It is convenient to be able to assign names to patterns of behaviour associated with components. Constants provide a mechanism for doing this. They are components whose meaning is given by a defining equation.

Hiding: The possibility to abstract away some aspects of a component's behaviour is provided by the hiding operator, denoted by the division sign in P/L. Here, the set L of visible action types identifies those activities which are to be considered internal or private to the component. These activities are not visible to an external observer, nor are they accessible to other components for cooperation. Once an activity is hidden it only appears as the unknown type τ; the rate of the activity, however, remains unaffected.

Cooperation: Most systems are comprised of several components which interact. In PEPA direct interaction, or *cooperation*, between components is represented by the butterfly combinator. The set which is used as the subscript to the cooperation symbol determines those activities on which the *cooperands* are forced to synchronise. Thus the cooperation combinator is in fact an indexed family of combinators, one for each possible *cooperation set L*. When cooperation is not imposed, namely for action types not in L, the components proceed independently and concurrently with their enabled activities. However if a component enables an activity whose action type is in the cooperation set it will not be able to proceed with that activity until the other component also enables an activity of that type. The two components then proceed together to complete the *shared activity*. The rate of the shared activity may be altered to reflect the work carried out by both components to complete the activity.

In some cases, when an activity is known to be carried out in cooperation with another component, a component may be *passive* with respect to that activity. This means that the rate of the activity is left unspecified and is determined upon cooperation, by the rate of the activity in the other component. All passive actions must be synchronised in the final model.

If the cooperation set is empty, the two components proceed independently, with no shared activities. We use a compact notation—with the two cooperands separated by parallel lines—to represent this case.

PEPA is a high-level notation for Markov modelling because it is possible to generate directly from a PEPA model a continuous-time Markov process which faithfully encodes the behavioural (same number of states; same transitions between states) and temporal (same rates on the transitions) aspects of the PEPA model. Through the analysis and solution of this Markov process the modeller can undertake an experimental investigation of the system which the model represents.

3 Modelling with UML and PEPA

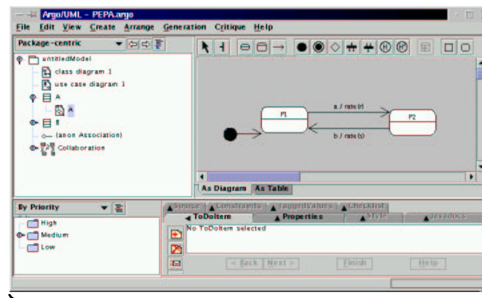
The PEPA notation is more than simply a concrete syntax for describing Markov processes. Central to the design of the language is the identification and representation of compositional structure within a model. This structure proves to be valuable both in gaining confidence that a given model correctly represents the intended system under investigation and also when seeking a solution for the corresponding Markov process.

One reason to fix on a formal notation for a task such as performance modelling is to avoid misunderstanding and misinterpretation of a model. Of course, even when a notation is carefully defined, as PEPA is, there may still be errors of misrepresentation of parts of the system within the model but all of the users of the model can at least agree on the correct interpretation of a given model through recourse to the formal

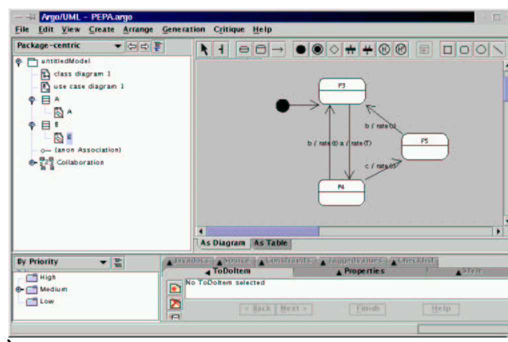
definition of the language. Another reason to fix on a formal notation for performance modelling is to be able to interface to other tools which perform other services in the manipulation and checking of models, and that is our topic here.

We bring the UML and PEPA notations together by forming a bridge between two existing applications which support these languages, the ArgoUML modelling tool [3] and the PEPA Workbench [4]. UML designs which are built using ArgoUML can be exported as XML Metadata Interchange documents (XMI) [5] as a standard part of the ArgoUML tool. The XMI format is used to represent UML models when exchanging them with other tools, as here, and facilitates the analysis and manipulation of UML models using standard XML tools [6]. We have developed an application which automatically converts the XMI documents generated by ArgoUML into the input file format of the PEPA Workbench. Fig. 2 shows screenshots of two ArgoUML designs of simple communicating state machines together with the equivalent descriptions in the PEPA stochastic process algebra. With a slight abuse of notation we show the rates of the transitions as UML actions. Thus $a/rate(r)$ is used to represent the information that the activity a is performed at rate r which is not the usual *event/actions* syntax for arc adornments in UML state diagrams.

When it is provided with an input PEPA model the PEPA Workbench explores the model to generate its full state space. This state space is used to form a CTMC representation of the system which is solved to find its steady-state probability distribution. As is usual with interleaving models of concurrent systems, the size of the state space of the system as a whole is bounded by the product of the state spaces of the individual PEPA components which are composed in parallel. Simply presenting this large probability vector back to the UML modeller as the result of the analysis would be unlikely to provide any insights into the long-run operation of the model, or hotspots or bottlenecks in the system. For this reason we look for an alternative means of communicating performance measures.



$$P_1 = (a,r).P_2; \quad P_2 = (b,s).P_1$$



$$P_3 = (a,\top).P_4;$$
$$P_4 = (b,t).P_3 + (c,r).P_5; \quad P_5 = (b,s).P_3$$

Fig. 2: Screenshots of UML designs in ArgoUML with PEPA equivalents

4 Performance measures

The most general way to describe performance measures is to build a reward structure on the model. However, associating locations in the equilibrium probability distribution with syntactic states of the model exposes details of the representation such as orderings of components in the PEPA system description. Such an approach would generally require the UML modeller to face much of the complexity of working directly with Markov Chains. Higher-level description languages for specifying performance measures exist, such as PML$_\mu$ [7] and CSL [8], but these notations would be formidable for a typical UML developer to use.

For this reason we aggregate the state space of the system over the local states of each PEPA component. This has two beneficial effects:

1. it avoids the need for any descriptions of state-space representations, whether high-level or low-level; and
2. instead of working with a large set of very small numbers the modeller works with a small set of numbers which are orders of magnitude larger.

Our approach to specifying performance measures is to define UML components which expose the configurations of interest in the model. Behaviourally, such components may be redundant, but they are necessary for expressing performance measures over the model. Typically such components will specify that they passively witness activites which have been performed and change state in order to reflect this information. By programming such components carefully it is possible that they do not increase the state space of the underlying Markov Chain but allow the modeller to observe that some sequence of activities has happened, and to learn the probability

of this. We term such components *witnesses* or *witness components*. Thus a model of a system will be comprised of components which capture the dynamics of the system and witnesses which are introduced to allow the expression of performance measures over the system. We have used this approach previously [9].

We illustrate the use of this method with a resource example and a queue example. One performance measure which is typically of interest is the calculation of the utilisation of resources in the system. To do this the modeller need only express the resource as a simple component as described below and the utilisation can be directly read from the model solution as the percentage of time that the component spends in the *Busy* state. This points to resources which are under-utilised, or over-utilised.

$$/* \text{ An idle resource can be acquired } */$$
$$Idle \stackrel{def}{=} (acquire, r_1).Busy$$

$$/* \text{ A busy resource can be released } */$$
$$Busy \stackrel{def}{=} (release, r_2).Idle$$

Taking the idea a little further, a finite-capacity M/M/n queue can be specified in PEPA with a list of component definitions ending with the following one.

$$/* \text{ no arrivals are allowed when the queue is full } */$$
$$Queue_n \stackrel{def}{=} (serve, \mu).Queue_{n-1}$$

The percentage of time that the queue will be full can be read off directly from the updated UML description. This points to the possibility of clients sending requests faster than they can be serviced by a server.

5 Software architecture

In this section we describe the architecture of our application. We have built on two existing software tools, ArgoUML and the PEPA Workbench. We have used ArgoUML with no modifications, so (up to minor XMI version differences) it could be replaced by any other XMI-capable UML tool. We modified the PEPA Workbench to make communication between the two tools easier. We begin by first describing these two tools for the benefit of readers who are not familiar with them. The architecture of the system is summarised in Fig. 3.

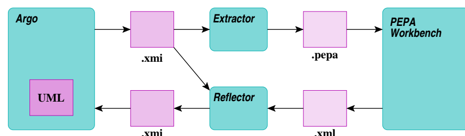


Fig. 3: Software architecture of the tool

5.1 ArgoUML

ArgoUML is a modelling tool which supports software developers who are producing software designs using UML. It provides many features which are familiar from existing CASE tools. Examples of these are editors for graphical notations such as class diagrams and state diagrams.

In addition, one of the distinctive features of ArgoUML is that it provides good support for the cognitive aspects of design, including supporting informal note-taking on “To Do” lists and other creativity aids. In all, it provides a professional and thoughtfully-engineered UML development platform.

As with most modern UML tools, ArgoUML exports UML models in the XMI document format, and loads saved models from the same XMI format. This provides the import and export formats for our other tools. The XMI document written by ArgoUML is read by our Extractor tool. The same XMI document and the results from processing by the PEPA Workbench (in XML format) are read by the Reflector tool to provide an updated input document which is loaded by ArgoUML.

5.2 The PEPA Workbench

The PEPA stochastic process algebra is supported by a range of tools including the PEPA Workbench [4], the Möbius Modeling Framework [10], the PRISM probabilistic symbolic model checker [11] and others [12]. We have used the PEPA Workbench so far in this work but the design of our companion Extractor and Reflector tools is general-purpose so that it would be possible to adapt our work to use either Möbius or PRISM instead. Both Möbius and PRISM offer capabilities which the PEPA Workbench does not. Möbius supports multi-paradigm modelling where PEPA models are combined with SANs or ball-and-bucket models as used by MARCA. PRISM provides probabilistic symbolic model checking allowing models to be checked against CSL formulae. Both of these tools could be valuable in our ongoing work but an engineering challenge would remain to allow the UML modeller to access their powerful capabilities without first needing to master their technical foundations.

The PEPA Workbench exists in two distinct versions. The first version is an experimental research tool which is coded in the functional programming language Standard ML [13]. The second is a re-implementation of this in the Java programming language. These are known as “the ML edition” and “the Java edition” respectively.

We adapted the Java edition of the PEPA Workbench to interoperate with our Extractor and Reflector tools. The Java edition provides a graphical user interface to assist the PEPA modeller in working with models, accessing tools such as the state finder tool, the simulator or the walkabout utility and choosing between a range of steady-state and transient solvers and a range of output formats. It would be impractical to use

this user interface together with the ArgoUML application so we added a command-line interface to the PEPA Workbench, allowing it to be configured with a range of command-line switches. One of these switches requests the Workbench to aggregate the performance results of the model over the local states of each PEPA component. Each such component is a simple sequential state machine which corresponds directly to a UML state diagram. Each of these has a very small state space relative to the state space of the model as a whole, making the result set much more compact. Of course this compactness is necessarily achieved at the expense of losing information about particular states of the global state space.

The PEPA Workbench processes an input PEPA model which the Extractor generates from an input UML model in XMI format. It writes its results as an XML document which is processed by the Reflector tool.

The PEPA Web page at `http://www.dcs.ed.ac.uk/pepa` is the download site for the PEPA Workbench and supporting software and papers.

5.3 The extractor and reflector tools

The extractor and reflector for PEPA also exist in two distinct versions.

The extractor and reflector were first implemented as standalone applications in the Python programming language [14]. These versions of these tools allow a UML modeller to access the PEPA tools as an alternative back end to their UML modelling tools. This provides a type of UML model analysis (computation of steady state residence probabilities) which is not currently available in the UML tools.

The extractor and reflector are also available as Java packages within the Java edition of the PEPA Workbench. This implementation allows PEPA modellers to use the UML tools as an alternative front end to the PEPA modelling tools. This provides a type of PEPA model representation (graphical descriptions of communicating state machines) which is not currently available in the PEPA tools.

Further descriptions of these extractors and reflectors now follow.

5.3.1 The Extractor as a Python module

The Extractor application has been implemented as a Python module. It reads in the XMI file generated when saving a UML model with ArgoUML (or a similar tool) and returns the corresponding input file for the PEPA Workbench.

We use the Minidom XML parser to parse the XMI file. Once we have converted the XMI file stream into a DOM object, we can then access the individual tags by name. The document is represented as a tree structure. When processing the XMI file we look for all the elements that we will need to provide in the PEPA model which we produce as output.

We have developed five different modules:

- *Extractor*, to extract information from a UML model;
- *PEPA_Extractor*, to extract information specific to PEPA;
- *StateMachine*, to extract information related to statemachines and their components;
- *Collaboration*, to extract information related to collaboration diagrams; and
- *Cooperation*, to generate the PEPA system equation (from the collaboration diagram).

An object of class *Extractor* can perform a number of fundemental operations. The two key methods are *parse*() and *get_element*().

The *PEPA_Extractor* class defined in this module is a subclass of the *Extractor* class. All methods of the *Extractor* class are inherited.

The most significant method of this class is *generate_PEPA*(). It calls three further methods (*generate_rates*(), *generate_terms*() and *generate_system_equation*()) to fill an array *PEPA_output* with the PEPA syntax corresponding to the current model.

The classes *StateMachine* and *Collaboration* call a constructor that accepts an object of class *Extractor* which is later used to retreive elements from the DOM model.

The class *StateMachine* provides methods and functions to extract information related to state machines and their components from the model.

The class *Collaboration* provides a method which is named *get_associations*. This returns an array containing pairs of associated instances.

The *Cooperation* module contains the definitions for two classes: *Node* and *Leaf*. When the PEPA Extractor is generating the system equation, it builds a tree consisting of nodes and leaves. A *Node* represents a cooperation ($\bowtie$), consisting of left and right branches, and a set of synchronisers. A *Leaf* contains a state, and the class instance from the collaboration diagram that it represents.

Now, we have all we need in order to write a PEPA Workbench input. First, we print each rate. Then, we print for each state machine a defining PEPA expression of the following form: `#source=(workload,rate).target+...` For each state, if it is a source, we find its transitions with the correct workload, rate and target.

Taking the information from a UML collaboration diagram as shown in Figure 4 we can assemble a collaboration line (e.g. `P1<a,b>P3`). We look for the initial states using the matching of context and classifier role, and then look for the activity names in the synchronisation set.

5.3.2 The Reflector as a Python module

The Reflector takes as its parameter the original XMI file and the XML file that contains all the results from the PEPA Work-

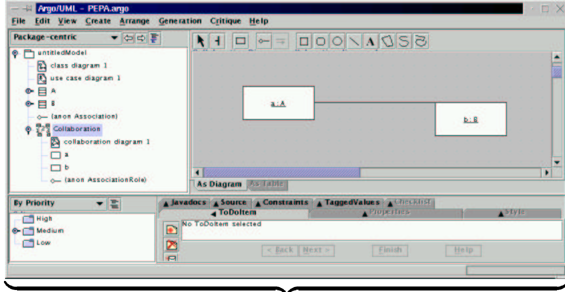


Fig. 4: UML collaboration diagram

bench. It returns a modified XMI file, which when loaded in ArgoUML will show the modified model.

We use Minidom to parse the original XMI file and also to build the modified XMI file by creating the branches and leaves of the document. The `xml.dom.minidom` module supports a very simple interface for adding new XML tags and data to an XML document.

We parse the XMI file corresponding to the UML model and the XML file corresponding to the PEPA Workbench results. When processing the PEPA Workbench results file, for each state, we look for the probability tag and identify its value. Then we modify our original XMI file where for each `statemachine`, for each `state`, we add the corresponding probability.

5.3.3 The Extractor as a Java package

There are two primary classes in the *extractor* package: *Extractor* and *PEPA_Extractor*.

The class *Extractor* performs file handling, DOM parsing, and provides convenient methods to make certain information in the DOM tree more accessible.

The *parse*() method takes an object of class *File*. If the file is a compressed archive (Zargo) file, it will locate the XMI file within the archive using the *java.util.zip* package. The XMI file is parsed using the *javax.xml.parsers.DocumentBuilder* DOM parser resulting in a *Document* object put into *dom_doc*. The file will be put into *xmi_file*. A successful parse will return `true`, otherwise `false` will be returned.

The *getElement*() method will return the *Element* from the DOM tree whose `xmi.id` attribute matches the given string value. This can be quite time consuming, and so a cache table is maintained called *dom_element_cache*.

The *getName*() static method locates the "name" child of the specified node. The value of this node is returned. Any existing performance annotations are ignored.

The *getChild*() static method will return the child of the specified node, whose tag name matches the given string value.

The *getByRef*() method uses *getChild*() to locate the named child of the specified node. The child of this node will have an attribute `xmi.idref`. Using the *getElement*() method, the node with the matching `xmi.id` attribute is returned. The *getAllByRef*() method returns all such referenced nodes.

The *getOwned*() static method uses *getChild*() to locate the "ownedElement" child of the specified node. All children of this child are returned.

The class *PEPA_Extractor* is a subclass of the class *Extractor*. It extracts from the DOM tree the information needed to generate a corresponding PEPA model. As it does so, an abstract syntax is generated which is then converted into a concrete, string representation.

There are a number of methods used to locate specific elements in the model or parts of it, in particular State Diagrams and Collaboration Diagrams. Although important, these do not affect the generation of the PEPA model. (One point worth noting is that the return type of *getAssociations*(), a two-dimensional array of *Element* objects, represents an array of pairs of associated *Elements*). The methods described below are used in the generation of the PEPA model.

The *generatePEPA*() method makes three calls to *getPEPA_definitions*(), *getPEPA_rates*() and *getPEPA_cooperation*() in that order and returns an array of strings. Although the rates appear before the definitions in the output, they are generated by *getPEPA_definitions*() and so the order in which they are called matters.

The *getPEPA_definitions*() method converts to string representation the *Definition* objects returned by *generatePEPA_definitions*() which takes a single `StateMachine` *Element* object. For each `State` *Element* in the state machine, *generatePEPA_definitions*() produces a *Definition* object, composed of objects from the *pepa.process* package. The behaviours in a definition are sorted on target state, and the list of definitions corresponding to a state machine are sorted with the initial state first.

The *getPEPA_rates*() method produces a number of rate variable assignments, initialising to a default value all rate variables encountered while producing the component behaviours.

The *getPEPA_cooperation*() method produces a single PEPA cooperation component, composed of the atomic cooperations created by *generatePEPA_cooperations*() from the associations in the Collaboration Diagram. There are two classes used to represent cooperations: *CoopNode* and *CoopLeaf*. An object of the *CoopLeaf* class has two field values determining its identity: the components initial `State` *Element* object and an *Element* representing the `ClassifierRole` (a particular instance of a class). There is a difference worth noting between the two *contains*() methods in both classes. One takes a *CoopLeaf* object as an argument,

the other an `State` *Element* object. The former will return `true` iff the *CoopNode* or *CoopLeaf* on which it is called contains an exact match, or is itself an exact match of the given instance. The latter will return `true` if there is another instance of the same class present. This makes it possible to successfully insert repeated components.

The *writePEPA*() method calls the *generatePEPA*() method, and writes the model that is returned to a `.pepa` file. The filename is determined from the input filename, simply replacing the `.xmi` or `.zargo` with `.pepa` and writing the file to the same directory.

5.3.4 The Reflector as a Java package

There is one class in this package called *Reflector*. This simple class has two static fields which store the original XMI or compressed archive (Zargo) file that was parsed using the *PEPA_Extractor*, and the XML file that holds the results generated by the PEPA Workbench.

The static method *reflect*() then performs the reflection. The two files are parsed using the *parse*() method of the *Extractor* class, the parsed *Document* objects being returned using its *getDocument*() method. The results contained within the XML file are extracted to a hashtable containing probabilities indexed by the states they represent. The XMI model is then updated with the results by annotating each state name with the probability associated to it. All state names must be unique. The PEPA Workbench will fault any model in which a component has more than one definition as being semantically ill-formed and print a diagnostic error message explaining the fault.

The modified model is then written back to file. If the original file had the suffix `.xmi`, the suffix of the reflected model will be `.reflected.xmi`. Similarly with a compressed archive file. This ensures the original model is not actually altered.

5.3.5 Using the Java packages

These two packages become *pepa.extractor* and *pepa.reflector* respectively. The two methods of invocation either instantiate the *pepa.gui.jpwb* class or the *pepa.tty.JPWBtty* class. The former is an interactive version of the Workbench, which allows the modeller to use any of the PEPA Workbench analysis methods, for example computing transient analysis measures, simulating the model or single-stepping through it with the debugger. This version is best suited for PEPA modellers who are using the UML tools as a graphical editor for PEPA. The latter performs the loading and solving at the command line, computing steady-state probabilities for the model. This version is best suited for UML modellers who are using the PEPA tools to provide an analyser for their UML model.

In both cases, before the point at which the model is loaded in the method *loadmodel*(), before the filename is processed further by the PEPA Workbench, a check is performed to determine if the input file is an XMI or a compressed archive file. If it is indeed one of these, the *PEPA_Extractor* is instantiated, the model parsed, and the PEPA model generated. The filename of this new `.pepa` file is then passed on for loading into the PEPA Workbench.

When the *PEPA_Extractor* successfully parses a model, the *File* object is passed to the *Reflector* using its static *setOriginal*() method. Similarly, when the PEPA Workbench solves a model and produces an XML results file, a *File* object is passed to the *Reflector* using its static *setResults*() method. The latter call is performed by *solve*() method of the *pepa.pepa.Peparoni* class.

Once the *Reflector* has the two files it needs to proceed with reflection, the static *reflect*() method is called, again by the *solve*() method of the *pepa.pepa.Peparoni* class. If reflection is not intended to occur, at least one of the *File* objects will remain at its default value of *null* and the method will fail safe.

6 The extractor algorithm

In this section, and in Fig. 5, 6 and 7, we describe the algorithm for extracting PEPA models from UML state diagrams and collaboration diagrams.

The algorithm for generating definitions of sequential components from state machines is relatively simple and involves traversing the transitions of the state diagrams and accumulating behaviours which are presented as choices in the definition of the component. The *generate_definitions* algorithm is presented in Fig. 5.

1: *terms* $\leftarrow$ empty list
2: **for** each statemachine *S* **do**
3:    **for** each state *s* (of *S*) **do**
4:       *beh* $\leftarrow$ empty list
5:       **for** each outgoing transition *t* (of *s*) **do**
6:          *w* $\leftarrow$ name of trigger event of *t*
7:          *r* $\leftarrow$ contents of "rate(...)" expression of *t*
8:          *target* $\leftarrow$ name of target state of *t*
9:          *beh* $\leftarrow$ *beh* + "(*w*, *r*).*target*"
10:       **end for**
11:       *n* $\leftarrow$ name of state *s*
12:       *terms* $\leftarrow$ *terms* + "# $n = beh_0$ [+ beh_1 [+ ...]]"
13:    **end for**
14: **end for**
15: **return** *terms*

Fig. 5: Algorithm *generate_definitions*

Transforming the diagrammatic representation of a sequential state machine into the textual description of a sequential

PEPA component is the most straightforward part of the extraction process. The more complex part is the extraction of the instances of these machines which are composed in parallel in the PEPA system equation. These components are instantiated by cooperation sets which specialise their behaviour and define the patterns of communication between instances.

The algorithm to generate the PEPA system equation traverses the collaboration diagram (an undirected graph) in order to generate an abstract syntax tree of the system equation. The nodes in this tree are cooperations between a left cooperand and a right cooperand over a set of activity names. Either of the left or right cooperand might be other nodes or they might be leaves containing instances of the sequential components extracted from the state diagrams.

```
 1: cooperations ← empty list
 2: for each association a (of the collaboration diagram) do
 3:     left_inst ← left element of a
 4:     right_inst ← right element of a
 5:     left_sm ← statemachine of the class of left_inst
 6:     right_sm ← statemachine of the class of right_inst
 7:     left ← Leaf(initial state of left_sm, left_inst)
 8:     right ← Leaf(initial state of right_sm, right_inst)
 9:     sync ← (events in left_sm) ∩ (events in right_sm)
10:     cooperations ← cooperations + Node(left, sync, right)
11: end for
12: root ← first element in cooperations
13: remove first element from cooperations
14: while cooperations is not empty do
15:     counter ← 0
16:     for each cooperation c (in cooperations) do
17:         root ← insert(root, c)          /* see Fig.7 for insert */
18:         if root has changed then
19:             remove c from cooperations
20:             counter ← counter + 1
21:         end if
22:     end for
23:     if counter is still 0 then
24:         break while                     /* fixed-point found */
25:     end if
26: end while
27: sys_eqn ← convert root to string
28: return sys_eqn
```

Fig. 6: Algorithm *generate_system_equation*

The algorithm *generate_system_equation*, presented in Fig. 6, traverses the collaboration diagram graph in an arbitrary order to build a list of pairwise collaborations between instances of sequential components. Synchronisation sets are inferred for these pairings. The synchronisation set which is inferred is the set of actions which are common to both the left cooperand and the right cooperand.

The list of collaboration pairs which is produced by this process is then folded to build the system equation. This is achieved by repeated inserting a new collaboration into an existing (initially empty) system equation.

```
insert(root, new, times = 0)
 1: if root is a Leaf and new is a Leaf then
 2:     return Node(root, [], new)
 3: else if root is a Leaf then
 4:     return new
 5: else if new is a Leaf then
 6:     if root.left contains a new instance then
 7:         new_left ← insert(root.left, new)
 8:         return Node(new_left, root.actions, root.right)
 9:     else if root.right contains a new instance then
10:         new_right ← insert(root.right, new)
11:         return Node(root.left, root.actions, new_right)
12:     else
13:         return root unchanged
14:     end if
15: end if
16: if root.left contains new.left and root.right contains
    new.right then
17:     new_actions ← root.actions ∪ new.actions
18:     return Node(root.left, new_actions, root.right)
19: else if root.left contains new.left then
20:     if root.right contains a new.right instance then
21:         if new.actions ⊆ root.actions then
22:             new_right ← insert(root.right, new.right)
23:             return Node(root.left, root.actions, new_right)
24:         end if
25:     end if
26:     new_left ← insert(root.left, new)
27:     return Node(new_left, root.actions, root.right)
28: else if root.right contains new.right then
29:     if root.left contains a new.left instance then
30:         if new.actions ⊆ root.actions then
31:             new_left ← insert(root.left, new.left)
32:             return Node(new_left, root.actions, root.right)
33:         end if
34:     end if
35:     new_right ← insert(root.right, new)
36:     return Node(root.left, root.actions, new_right)
37: end if
38: if times > 0 then
39:     return root unchanged
40: end if
41: swap left and right leaves of new
42: return insert(root, new, 1)
```

Fig. 7: Algorithm *insert*

The algorithm to perform this insertion is surprisingly complex (Fig. 7). Most of the complexity stems from the commonly occurring case where a model contains more than one

instance of a given sequential component. For example, a model of a system which has two servers and ten clients would have two instances of the *Server* component and ten instances of the *Client* component.

In building the system equation for such a system it is essential that the repeated copies of a component definition are not forced to synchronise on all of their common actions. Since all of their actions are common, this would force them to operate in lockstep, instead of as independent replicated instances of a component. If a modeller wished to represent two instances of a component operating in lockstep then they would represent this explicitly in the collaboration diagram with a collaboration link between the two instances.

The operation of the *insert* algorithm is complex, but the key to understanding its operation is to know that it is checking for repeated instances of components within a system equation and avoiding the accidental capture of one component's behaviour by the synchronisation sets governing the behaviour of another copy of this component.

7 A simple example

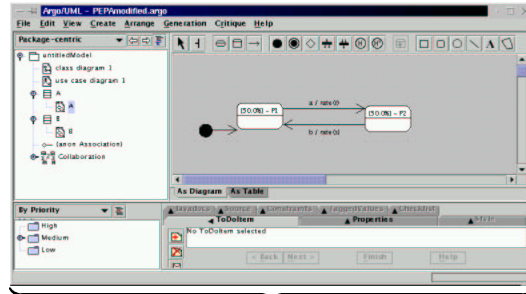
We demonstrate our method on a simple generic example model. The model is formed by a composition of a two-state component and a three-state component. To make this generic example more concrete, the two-state component might represent a client which requests a service and receives a reply and the three-state component might represent a proxy server which sometimes replies directly but at other times connects to another server before replying.

Fig. 2 shows the original UML model used for our example. When this UML model is saved in ArgoUML, an XMI file is generated. If we use the Extractor with this XMI file, we obtain the PEPA model which is shown in Fig. 8 in the concrete syntax of the PEPA Workbench. This is used as the

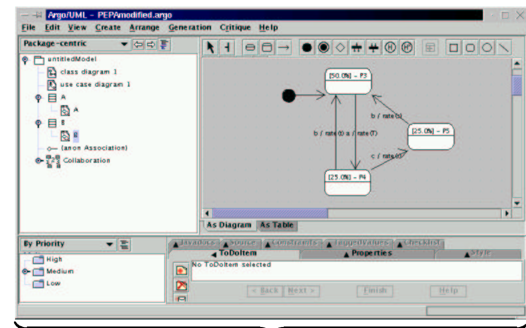
```
%r = 2.0;
%s = 2.0;
%t = 2.0;
#P4 = (c,r).P5 + (b,t).P3;
#P5 = (b,s).P3;
#P3 = (a,infty).P4;
#P1 = (a,r).P2;
#P2 = (b,s).P1;
P1 < a,b > P3
```

Fig. 8: PEPA model generated by the Extractor tool

input for the PEPA Workbench which then produces its results in an XML file. We can now use the Reflector to modify the original XMI file, and make the probability of each state appear on the UML diagrams (see Fig. 9).



P_1 : 50.0%, P_2 : 50.0%



P_3 : 50.0%, P_4 : 25.0%, P_5 : 25.0%

Fig. 9: Screenshots of ArgoUML incorporating PEPA results

8 Case study: a location tracking system

As an example here we consider the problem of modelling a system where the location of people and equipment within a building is monitored by a central tracking system. The James Clerk Maxwell Building at The University of Edinburgh is notoriously confusing to navigate and a tracking system would be helpful in finding those visitors who get lost in the maze of corridors. The system would also help secretaries find professors who may be in any number of teaching and meeting rooms or colleagues' offices and would be an invaluable aid in the hunt for the (non-networked) laptop computers which can be borrowed for the secure preparation of examination papers.

Location tracking systems such as these are implemented by the use of *active badges*, credit-card sized devices which transmit unique infra-red signals which are detected by networked sensors. Systems such as these are already in use in several European universities and in research laboratories in the USA. The battery life of such a device has typically been found to be around a year [15] so it is necessary to tune the

Fig. 10: The state diagram for a person

performance of the system by adjusting the rate at which registration is performed in order to conserve battery power while simultaneously ensuring that the system gives accurate location information.

A Markovian stochastic process algebra such as PEPA is well suited to modelling this system because exponential registration intervals are used to prevent the repeated collisions between transmitting badges which would result in lost messages [16]. This is the same use of randomness as found in the Aloha packet-switching network: without it, a collision would inevitably be followed by another collision.

To keep the example small we will consider the simple case of tracking the progress of a single person around a single floor of a building. The floor has three corridors which are numbered 14, 15 and 16, and we assume that there is only a single sensor in each corridor. The corridors are arranged in a U-shape so that it is possible to go from the 14 corridor to the 15 corridor and then to the 16 corridor (and the other way, of course) but it is not possible to go from the 14 to the 16 corridor directly.

The behaviour of a person P who is wearing an active badge can be described in terms of their movement from one corridor to a neighbouring one and the registration of their badge with the nearest sensor. The UML diagram which describes this behaviour is shown in Fig. 10.

The PEPA definitions which are extracted from this diagram by the PEPA Extractor are shown below.

$$
\begin{aligned}
P_{14} &\stackrel{def}{=} (reg_{14}, r).P_{14} + (move_{15}, m).P_{15} \\
P_{15} &\stackrel{def}{=} (move_{14}, m).P_{14} \\
&\quad + \ (reg_{15}, r).P_{15} \\
&\quad + \ (move_{16}, m).P_{16} \\
P_{16} &\stackrel{def}{=} (reg_{16}, r).P_{16} + (move_{15}, m).P_{15}
\end{aligned}
$$

Sensors accept registration information and report this back to the central database. The state diagram for the sensor in the 14 corridor is shown in Fig. 11. The state diagrams for the other sensors are similar.

The PEPA definitions extracted from the sensor diagrams are shown below.

$$
\begin{aligned}
S_{14} &\stackrel{def}{=} (reg_{14}, \top).S'_{14} \\
S'_{14} &\stackrel{def}{=} (rep_{14}, s).S_{14} \\
S_{15} &\stackrel{def}{=} (reg_{15}, \top).S'_{15}
\end{aligned}
$$

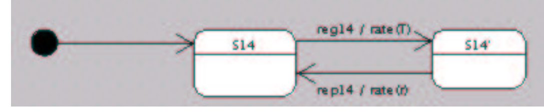


Fig. 11: The S_{14} sensor state diagram

$$
\begin{aligned}
S'_{15} &\stackrel{def}{=} (rep_{15}, s).S_{15} \\
S_{16} &\stackrel{def}{=} (reg_{16}, \top).S'_{16} \\
S'_{16} &\stackrel{def}{=} (rep_{16}, s).S_{16}
\end{aligned}
$$

For a system with only one person to be tracked the database need only store the most recently reported position. The database updates its location information as it receives reports from the sensors, changing state to store this information. The state diagram for this component is shown in Fig. 12.

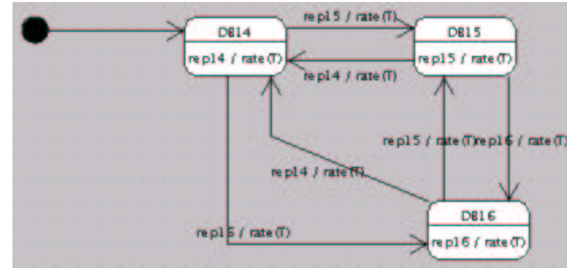


Fig. 12: The database state diagram

The PEPA definitions extracted from the database state diagram by the PEPA Extractor are shown below.

$$
\begin{aligned}
DB_{14} &\stackrel{def}{=} (rep_{14}, \top).DB_{14} + (rep_{15}, \top).DB_{15} \\
&\quad + (rep_{16}, \top).DB_{16} \\
DB_{15} &\stackrel{def}{=} (rep_{14}, \top).DB_{14} + (rep_{15}, \top).DB_{15} \\
&\quad + (rep_{16}, \top).DB_{16} \\
DB_{16} &\stackrel{def}{=} (rep_{14}, \top).DB_{14} + (rep_{15}, \top).DB_{15} \\
&\quad + (rep_{16}, \top).DB_{16}
\end{aligned}
$$

In the complete system the badge-wearer will move asynchronously but will register with the sensors. The sensors are independent but they all report back to the database. There

is no direct connection between the person and the database. All flow of information is routed through the sensors in the system. This structural information about the connectivity of the system is recorded in the collaboration diagram shown in Fig. 13.

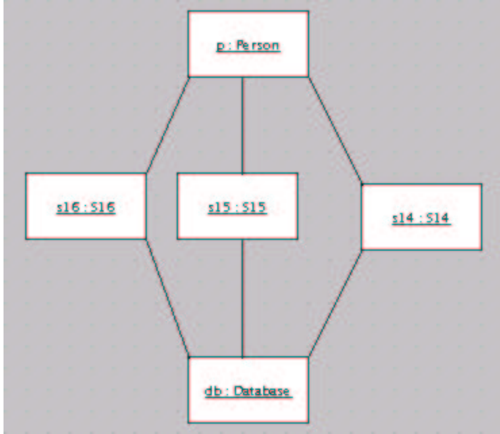


Fig. 13: The collaboration diagram for the location tracking system

We can initialise the system in any state we wish, perhaps with the badge-wearer in the 14 corridor and the database also recording this. If a PEPA modeller was composing this model directly without using our Extractor and Reflector tools then it would be likely that they might express the system equation for the model as shown below [17]:

$$P_{14} \underset{\{reg_i\}}{\bowtie} (S_{14} \parallel S_{15} \parallel S_{16}) \underset{\{rep_i\}}{\bowtie} DB_{14}$$

This expresses the information which is recorded in the collaboration diagram (for example, that there is no direct connection between the person and the database, nor are any two sensors directly linked). However, the actual system equation which is generated by our Extractor is syntactically different, but semantically identical. The system equation is obtained by a graph traversal of the collaboration diagram leading to the following expression.

$$\left((P_{14} \underset{\{reg_{14}\}}{\bowtie} S_{14}) \underset{\{reg_{16}, rep_{14}\}}{\bowtie} (DB_{16} \underset{\{rep_{16}\}}{\bowtie} S_{16}) \right) \underset{\{reg_{15}, rep_{15}\}}{\bowtie} S_{15}$$

This expression is not as tidy and compact as the one which an experienced PEPA modeller would produce but it is important to remember that the modeller using our Extractor and Reflector tools need never see this expression. The mode of operation with the PEPA Workbench is simply to load either an XMI file or a Zargo archive directly. The corresponding PEPA file is generated from this and loaded in one step without the need for further intervention. After this model has been run (to generate its state space) solving it to steady state solution will generate a modified XMI file or Zargo archive in the same directory as the original input. The PEPA Workbench is shown processing the location tracking system model in Fig. 14.

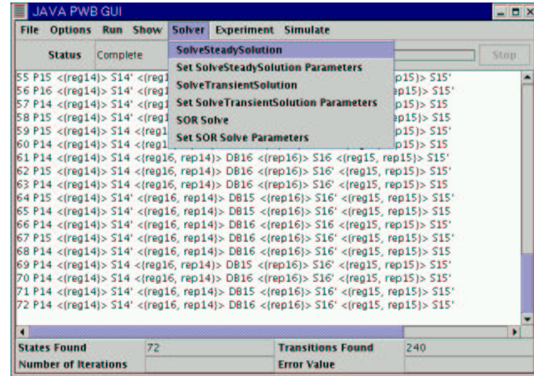


Fig. 14: The PEPA Workbench solving the location tracking system model

9 Related work

Ours is not the first work on using UML with stochastic process algebras, nor even the first on using UML with PEPA. Pooley [18] previously discussed generating PEPA models from a combination of sequence diagrams, collaboration diagrams, and a combined collaboration/state diagram. His method differs from ours in that more types of diagram are used and there seems to be no automatic procedure used to generate the PEPA model from the UML. In contrast, our method is automatic and can be re-run after changes to the UML model in order to generate and solve an equivalent updated PEPA model.

Mitton and Holton undertake an alternative mapping between PEPA and UML statecharts [20] but again their method does not appear to be automated.

In another paper Thomas, Munro, King and Pooley combine PEPA models with graphical notations for visualising derivation graphs, PEPA component interfaces and other aspects of the system under study [21]. This work provides an interesting insight into the PEPA modelling language for modellers who are not familiar with the notation. Their approach is supported by a prototype tool. Our contribution here differs in that we are using a standard and widely-understood modelling language (UML) instead of more specialised, but necessarily less well-understood bespoke graphical notations.

A work closely related to ours in spirit, if not in detail, is recent work by Petriu and Shen which maps UML models via XMI into layered queueing network (LQN) performance models [22]. This mapping is in one direction only, so that UML models are mapped into LQN models but there appears

to be no mapping of the performance results obtained from the LQN model back up to the UML level. Thus their tool appears to be similar to our Extractor tool, combined with the PEPA Workbench, but without an equivalent of our Reflector tool.

10 Conclusions

We have presented a method of deriving performance information from UML models. Our method is unusual in that it greatly reduces the amount of additional notation and concepts which need to be understood by the modeller when compared to working directly with stochastic process algebras, Petri nets, queueing networks or other traditional performance modelling formalisms. The method is supported by a tool set which comprises some existing modelling tools (ArgoUML and the PEPA Workbench) and other translators which we have written to connect them (the Extractor and the Reflector). The translators which interconnect the applications are general-purpose and can be adapted to work with other modelling tools.

We have applied our approach to a range of small examples. We plan to investigate its usefulness when applied to larger examples.

We have used a bespoke method of adding performance information to UML models here. We also plan to investigate more standard ways of representing the performance information in UML. This will probably use the Schedulability Performance and Time profile [23], as used by Petriu and Shen, depending on the availability of appropriate UML tool support.

Acknowledgements: The authors are supported by the DEGAS (Design Environments for Global ApplicationS) project IST-2001-32072 funded by the FET Proactive Initiative on Global Computing.

A Operational semantics and the underlying CTMC

Model components capture the structure of the system in terms of its *static* components. The dynamic behaviour of the system is represented by the evolution of these components, either individually or in cooperation. The form of this evolution is governed by a set of formal rules which give an operational semantics of PEPA terms. The semantic rules, in the structured operational style of Plotkin, are presented in Fig. 15 without further comment; the interested reader is referred to [2] for more details. The rules are read as follows: if the transition(s) above the inference line can be inferred, then we can infer the transition below the line.

Prefix

$$\frac{}{(\alpha, r).E \xrightarrow{(\alpha,r)} E}$$

Cooperation

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,r)} E' \underset{L}{\bowtie} F} \quad (\alpha \notin L)$$

$$\frac{F \xrightarrow{(\alpha,r)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,r)} E \underset{L}{\bowtie} F'} \quad (\alpha \notin L)$$

$$\frac{E \xrightarrow{(\alpha,r_1)} E' \quad F \xrightarrow{(\alpha,r_2)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,R)} E' \underset{L}{\bowtie} F'} \quad (\alpha \in L)$$

$$\text{where } R = \frac{r_1}{r_\alpha(E)} \frac{r_2}{r_\alpha(F)} \min(r_\alpha(E), r_\alpha(F))$$

$r_\alpha(E)$ is the apparent rate of α in E

Choice

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E + F \xrightarrow{(\alpha,r)} E'} \qquad \frac{F \xrightarrow{(\alpha,r)} F'}{E + F \xrightarrow{(\alpha,r)} F'}$$

Hiding

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E/L \xrightarrow{(\alpha,r)} E'/L} \quad (\alpha \notin L)$$

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E/L \xrightarrow{(\tau,r)} E'/L} \quad (\alpha \in L)$$

Constant

$$\frac{E \xrightarrow{(\alpha,r)} E'}{A \xrightarrow{(\alpha,r)} E'} \quad (A \stackrel{def}{=} E)$$

Fig. 15: The operational semantics of PEPA

Thus, as in classical process algebra, the semantics of each term in PEPA is given via a labelled *multi-transition* system—the multiplicities of arcs are significant. In the transition system a state corresponds to each syntactic term of the language, or *derivative*, and an arc represents the activity which causes one derivative to evolve into another. The complete set of reachable states is termed the *derivative set* of a model and these form the nodes of the *derivation graph* formed by applying the semantic rules exhaustively.

The timing aspects of components' behaviour are not represented in the states of the derivation graph, but on each arc as the parameter of the negative exponential distribution governing the duration of the corresponding activity. The interpretation is as follows: when enabled an activity $a = (\alpha, r)$ will delay for a period sampled from the negative exponential distribution with parameter r. If several activities are enabled concurrently, either in competition or independently, we assume that a *race condition* exists between them. Thus the activity whose delay before completion is the least will be the one to succeed. The evolution of the model will determine whether the other activities have been *aborted* or simply *interrupted* by the state change. In either case the memoryless property of the negative exponential distribution eliminates the need to record the previous execution time.

When two components carry out an activity in cooperation the rate of the shared activity will reflect the working capacity of the slower component. We assume that each component has a capacity for performing an activity type α, which cannot be enhanced by working in cooperation (it still must carry out its own work), unless the component is passive with respect to that activity type. For a component P and an action type α, this capacity is termed the *apparent rate* of α in P. It is the sum of the rates of the α type activities enabled in P. The apparent rate of α in a cooperation between P and Q over α will be the minimum of the apparent rate of α in P and the apparent rate of α in Q.

The derivation graph is the basis of the underlying Continuous Time Markov Chain (CTMC) which is used to derive performance measures from a PEPA model. The graph is systematically reduced to a form where it can be treated as the state transition diagram of the underlying CTMC. Each derivative is then a state in the CTMC. The *transition rate* between two derivatives P and Q in the derivation graph is the rate at which the system changes from behaving as component P to behaving as Q. It is denoted by $q(P,Q)$ and is the sum of the activity rates labelling arcs connecting node P to node Q. In order for the CTMC to be *ergodic* its derivation graph must be strongly connected. Some necessary conditions for ergodicity, at the syntactic level of a PEPA model, have been defined [2]. These syntactic conditions are imposed by the grammar introduced earlier.

References

[1] Object Management Group. Unified Modeling Language, v1.4, March 2001. OMG document number: formal/2001-09-67.

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

[3] Tigris.org project. ArgoUML: A modelling tool for design using UML. Web page and documentation at http://argouml.tigris.org/, 2002.

[4] 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.

[5] Object Management Group. OMG-XML Metadata Interchange (XMI) Specification, v1.1, November 2000. OMG document number: formal/00-11-02.

[6] P. Stevens. Small-scale XMI programming: a revolution in UML tool use? *Journal of Automated Software Engineering*, 2002. Accepted for publication.

[7] G. Clark, S. Gilmore, J. Hillston, and M. Ribaudo. Exploiting modal logic to express performance measures. In B.R. Haverkort, H.C. Bohnenkamp, and C.U. Smith, editors, *Computer Performance Evaluation: Modelling Techniques and Tools, Proceedings of the 11th International Conference*, number 1786 in LNCS, pages 211–227, Schaumburg, Illinois, USA, March 2000. Springer-Verlag.

[8] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In *Computer-Aided Verification*, volume 1102 of *LNCS*, pages 169–276. Springer-Verlag, 1996.

[9] S. Gilmore and J. Hillston. Feature interaction in PEPA. In C. Priami, editor, *Proceedings of the Sixth Annual Workshop on Process Algebra and Performance Modelling*, pages 17–26, Nice, France, September 1998. Università degli studi di Verona.

[10] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster. The Möbius modeling tool. In *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, pages 241–250, Aachen, Germany, September 2001.

[11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In Field and Harrison [24], pages 200–204.

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

[13] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[14] Python language website. http://www.python.org/, 2002.

[15] A. Harter and A. Hopper. A distributed location system for the active office. *IEEE Network Magazine*, 8(1):62–70, 1994.

[16] Y-B. Lin and P. Lin. Performance modeling of location tracking systems. *Mobile Computing and Communications Review*, 2(3):24–27, 1998.

[17] G. Clark, S. Gilmore, and J. Hillston. Specifying performance measures for PEPA. In J.-P. Katoen, editor, *Proceedings of the Fifth International AMAST Workshop on Real-Time and Probabilistic Systems*, number 1601 in LNCS, pages 211–227, Bamberg, Germany, May 1999. Springer-Verlag.

[18] R. Pooley. Using UML to derive stochastic process algebra models. In Davies and Bradley [19], pages 23–33.

[19] N. Davies and J. Bradley, editors. *Proceedings of the Fifteenth UK Performance Engineering Workshop*, Department of Computer Science, The University of Bristol, July 1999.

[20] P. Mitton and R. Holton. PEPA performability modelling using UML statecharts. In Thomas and Bradley [25], pages 19–33.

[21] N. Thomas, M. Munro, P. King, and R. Pooley. Visualisation for model comprehension. In Thomas and Bradley [25], pages 47–58.

[22] D.C. Petriu and H. Shen. Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specifications. In Field and Harrison [24], pages 159–177.

[23] B. Selic, A. Moore, M. Woodside, B. Watson, M. Bjorkander, M. Gerhardt, and D. Petriu. Response to the OMG RFP for Schedulability, Performance, and Time, revised, June 2001. OMG document number: ad/2001-06-14.

[24] A.J. Field and P.G. Harrison, editors. *Proceedings of the 12th International Conference on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation*, number 2324 in Lecture Notes in Computer Science, London, UK, April 2002. Springer-Verlag.

[25] N. Thomas and J. Bradley, editors. *Proceedings of the Sixteenth UK Performance Engineering Workshop*, Department of Computer Science, The University of Durham, July 2000.