Automatic Extraction of PEPA Performance Models from UML Activity Diagrams Annotated with the MARTE Profile

Mirco Tribastone and Stephen Gilmore
Laboratory for Foundations of Computer Science
The University of Edinburgh
{mtribast, stg}@inf.ed.ac.uk

ABSTRACT

Recent trends in software engineering lean towards model-centric development methodologies, a context in which the UML plays a crucial role. To provide modellers with quantitative insights into their artifacts, the UML benefits from a framework for software performance evaluation provided by MARTE, the UML profile for model-driven development of Real Time and Embedded Systems. MARTE offers a rich semantics which is general enough to allow different quantitative analysis techniques to act as underlying performance engines. In the present paper we explore the use of the stochastic process algebra PEPA as one such engine, providing a procedure to systematically map activity diagrams onto PEPA models. Independent activity flows are translated into sequential automata which co-ordinate at the synchronisation points expressed by fork and join nodes of the activity. The PEPA performance model is interpreted against a Markovian semantics which allows the calculation of performance indices such as throughput and utilisation. We also discuss the implementation of a new software tool powered by the popular Eclipse platform which implements the fully automatic translation from MARTE-annotated UML activity diagrams to PEPA models.

Categories and Subject Descriptors

I.6.5 [**Simulation and Modeling**]: Model Development—*Modeling methodologies*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Performance

Keywords

UML, PEPA, MARTE

WOSP'08, June 24–26, 2008, Princeton, New Jersey, USA.

1. INTRODUCTION

In the past few years research in software engineering has witnessed significant interest in the performance evaluation of software models. This has become an even more compelling and exciting issue as development practices shift towards *model-driven methodologies*. At the core of the model-driven development approach is the description of system functionality using a platform independent model and the translation of such models into concrete code by means of automated tools.

A role of premier importance in this context is played by the UML, a potent formalism which allows static, behavioural, and architectural modelling of software systems. A *view* of particular interest for the purpose of software performance engineering is the behavioural view, which describes the interactions between the components of the system. Widely-used means to represent the system dynamics are collaboration diagrams, use case diagrams, sequence diagrams, state machine diagrams, and activity diagrams. In the present paper we shall be concerned with the performance evaluation of annotated UML activity diagrams.

The UML lacks semantics and notation for the specification of time and performance-related indices of interest (see [28]). Fortunately, such a framework has been supplied by SPT, the standardised UML profile for Schedulability, Performance and Time [23] and more recently by the profile for MARTE [25]. When its specification is finalised, the MARTE profile is intended to supersede the SPT profile. Rather than provide concrete techniques and tools, these profiles both establish a framework which other parties can build upon. Their semantics are general enough to allow many different techniques and tools to be employed as performance evaluation engines, thus effectively acting as a *lingua franca* for the purposes of performance evaluation of UML models.

This paper is concerned with the use of the MARTE profile for the performance evaluation of UML activity diagrams with the PEPA stochastic process algebra [14]. As with all process algebras, PEPA describes components that may cooperate with each other to carry out some shared action. Unshared actions are executed *concurrently* by the processes in the system. In contrast to classical process algebras where time is abstracted away, PEPA has timed activities, which allow performance analysis to be carried out.

We present an algorithm which translates activity diagrams annotated with MARTE stereotypes into PEPA descriptions. At the core of this algorithm is the interpretation of independent flows in the activity diagram as concurrent processes in the underlying PEPA model. The processes synchronise at the fork and join points in the activity, and decision branches

are interpreted non-deterministically. MARTE stereotypes are applied to activity nodes to denote the amount of time taken by that unit of computation. Depending on the modelling methodology adopted and the stage at which the model is being analysed, such values may be regarded as initial guess (early in the development cycle) or measurements (from a prototype or a fully-deployed system). In accordance with the MARTE profile, performance metrics of interest to the modeller are also specified as stereotype applications to nodes of the diagram. The translation procedure is concretely applied to a running example.

Our approach is supported by a software tool implemented as a *plug-in* contribution to Rational Software Architect [17]. The tool takes a UML2 model instance compliant with the meta-model implementation of the Eclipse UML2 Project [10] and automatically extracts PEPA descriptions from activity diagrams with the MARTE annotations. Performance measures are evaluated by the PEPA Eclipse Plug-in Project [29], an Eclipse contribution for PEPA. Performance results are presented to the user as values for the output variables defined in the original model.

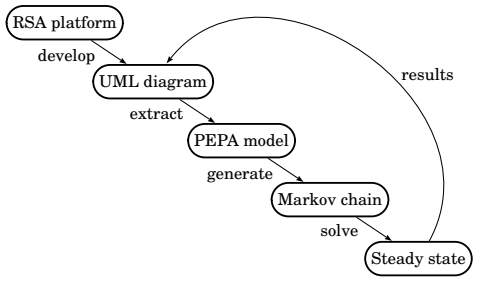

Figure 1: Workflow of the analysis process. UML activity diagrams are developed in Rational Software Architect. PEPA models are extracted from these diagrams. A CTMC is generated from the PEPA model and solved for its steady-state distribution. Performance results are calculated from this and the UML model is analysed with respect to these results.

Structure of this paper.

We discuss related work in Section 2. Section 3 presents a brief introduction to the PEPA language which we use to calculate performance results. Section 4 gives an overview of activity diagrams and the MARTE profile. This section also presents the running example used in the paper. Section 5 presents the algorithms used to translate a UML activity diagram into a PEPA model. Section 6 gives details of the software tool which implements the translation. Conclusions and further work are discussed in Section 7.

2. RELATED WORK

Performance evaluation in the software development process has been an active area of research in recent years. The survey in [2] provides an extensive overview of this field. In particular, quantitative analysis of UML models has been enjoying research interest from different model analysis methodologies. For instance, in [26] activity diagrams are translated into Layered Queueing Networks (LQN). In [3] a combination of Use Case diagrams, Activity Diagrams and Deployment Diagrams annotated with SPT are translated into Multiclass Queueing Network Models. In [8] *quality of service* agreements of service-oriented architectures are assessed through activity diagrams annotated with the UML Profile for Automated Business Processes and WSDL descriptions enhanced with SPT extensions to expose performance characteristics. Generalised Semi-Markov Processes are used by DSPNexpress [18] to carry out performance evaluation of state machine diagrams and activity diagrams. In [16] discrete-event simulation models are extracted from a combination of activity diagrams and collaboration diagrams whose performance characteristics are interactively inserted by the modeller as simulation progresses.

This is not the first work which focuses on performance evaluation of UML models using PEPA. In [7] an algorithm is discussed to transform UML state machine diagrams and collaboration diagrams into PEPA models. The implementation tool which supports such a transformation has inspired the extraction/reflection approach of our tool. Along this line of research is the contribution in [13], where a case study on mobile telephony is model-checked via an annotated UML mapped to PEPA. More closely related to the present work is [6] where UML2 activity diagrams are interpreted as PEPA nets [11], a variant of PEPA. Both control flows and object flows are considered, however the translation is tailored to a particular case study and a means to systematically map such diagrams is not provided. In addition the tool does not benefit from the framework provided by the MARTE profile, thus making it necessary to annotate the diagram according to non-standard notations.

Another work close in spirit to ours is [19], where performance evaluation of activity diagrams is carried out via translation into Generalised Stochastic Petri Nets. Although the transformation is based on the UML 1.5, the authors point out that it is closer to the UML2 token-based semantics of activity diagrams. The main difference is in the interpretation of UML activities: the authors put them into a more general framework for software performance engineering. More specifically, activities express the behaviour of the *doActivity* of the states in UML state machines. Related to this context is other research on performance modelling with use cases [21], state machines [20], and sequence diagrams [4]. In our approach, activities represent behaviour which can be analysed *per se*. However, activities are given a Markovian interpretation by means of performance annotations via the SPT profile. For this reason, we believe that our approach is complementary to that work, and helps fulfil the guiding principle of SPT-MARTE about providing a common framework for different model analysis techniques. In particular, here we exploit the large toolset that PEPA has been enjoying over the last decade, which makes Markovian steady-state analysis, passage-time analysis, model-checking, and discrete-event simulation readily available to the software performance engineering community.

Another major benefit from the use of PEPA as intermediate language is that it enables other forms of analysis which help cope with the well-known state space explosion problem of large discrete-state models. PEPA has been recently provided with a fluid-flow semantics [15] which gives rise to a system of first-order differential equations as the underlying mathematical tool for performance evaluation. Although this paper focuses on the Markovian interpretation, the translation

described here may seamlessly employ this continuous-state semantics. Even in the realm of Markovian analysis, PEPA is the possibility of performing efficient and scalable stochastic simulation, via its interpretation against a recently proposed population-based semantics [5].

3. OVERVIEW OF PEPA

PEPA is a stochastic process algebra which allows the performance evaluation of models described using the following two-level grammar:

$$\begin{aligned} S &:= (\alpha,r).S \mid S+S \mid A \\ C &:= S \mid C \underset{L}{\bowtie} C \mid C/L \end{aligned}$$

The first production defines sequential components, whereas the second allows composition of components. Below is an informal description of the operators of the grammar. For the formal definition the reader is referred to [14].

Prefix $(\alpha,r).S$ denotes an activity of type α performed at rate r by a sequential component. The sequential component is said to *enable the activity*. The rate indicates an exponential distribution with mean delay $1/r$. When the activity completes, the sequential component behaves as S.

Choice $P+Q$ indicates probabilistic choice among the activities enabled by the sequential components P and Q.

Constant $A \stackrel{def}{=} S$ is used to define cyclic behaviour. The sequential component A behaves as S.

Cooperation $P \underset{L}{\bowtie} Q$ allows composition. P and Q carry out their enabled activities concurrently if the type of the activity is not in the cooperation set L. If the activity's type is in L, they perform a *shared action* at a rate which depends on the rates of the individual components involved in the cooperation. A rate of a shared activity may be left as unspecified at a particular sequential component by using the symbol $\top$. This signifies that the shared rate is specified by other components.

Hiding C/L turns all the enabled activities of C whose type is in the action set L into *silent* activities over which cooperation is not possible. Hiding will not be used in the remainder of this paper.

PEPA descriptions are interpreted against an operational semantics which results in a labelled transition system whose states are PEPA components and transition labels are the *(type, rate)* pairs of the activities enabled by that state. A Continuous Time Markov Chain (CTMC) can be derived from the labelled transition system by associating each state of the system with a state of the Markov process. The generator matrix is extracted from the rates in the transition labels. The solution of this underlying CTMC ultimately allows for the performance evaluation of the system.

As a practical example, consider the PEPA model of an application invoking some web service modelled as two sequential components, as shown in Fig. 2. After a certain amount of thinking time (with mean duration $p_1\lambda$) the application may perform a local activity and loop back to its initial state. Alternatively (rate $p_2\lambda$), it may make a *request* to the web service (shared activity) and wait until a *respond* action can be performed. Below the PEPA description is the corresponding labelled transition system.

$$\begin{aligned} Appl &\stackrel{def}{=} (think, p_1\lambda).Appl_1 \\ &\quad + (think, p_2\lambda).Appl_2 \\ Appl_1 &\stackrel{def}{=} (local, m).Appl \\ Appl_2 &\stackrel{def}{=} (request, rq).Appl_3 \\ Appl_3 &\stackrel{def}{=} (respond, rp).Appl \\ \\ WS &\stackrel{def}{=} (request, \top).WS_1 \\ WS_1 &\stackrel{def}{=} (serve, \mu).WS_2 \\ WS_2 &\stackrel{def}{=} (respond, \top).WS \\ \\ Sys &\stackrel{def}{=} Appl \underset{\mathscr{L}}{\bowtie} WS \\ \mathscr{L} &= \{request, respond\} \end{aligned}$$

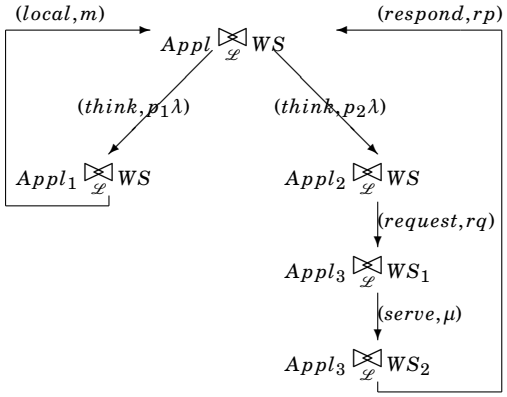

Figure 2: The PEPA model of the web service and its underlying derivation graph

4. ACTIVITY DIAGRAMS AND MARTE

In this section we describe our approach to performance evaluation via UML2 activities, developed in the context of MARTE. The specification of activities in the UML has changed significantly with the transition from UML1.5 to UML2. In this work we shall be concerned with the semantics of the latest version. Unless otherwise stated, the term UML refers to the UML2 specification in the remainder of this paper. In this section we overview the meta-model of UML activities with focus on the elements of interest for performance evaluation using PEPA. The definitive reference for UML activities is the UML2 formal specification [24].

An activity is a behavioural element of the UML which models the coordination of lower-level behaviours, both sequential and concurrent, to carry out a computational step. An activity is represented as a graph of activity nodes connected by two kinds of edges: control flows and object flows. Activity nodes may be of three types: *Action nodes*, modelling a unit of computation of the system; *Object nodes*, representing objects existing at a given point during an activity; and *Control nodes*, which are used for the coordination of flows. We are concerned with control flows here. Figure 3 shows the UML elements which will be considered for translation throughout this paper.

We use MARTE according to the *activity-based approach* discussed in [23], Sect. 7.2.1.2.[1] In particular, the *extension units* of interest here are those of the *Base Performance Com-*

[1]Although this approach is described in the SPT specification, similar arguments hold for MARTE. An easy-to-use correspondence table between MARTE and SPT is provided in Annex H of [25].

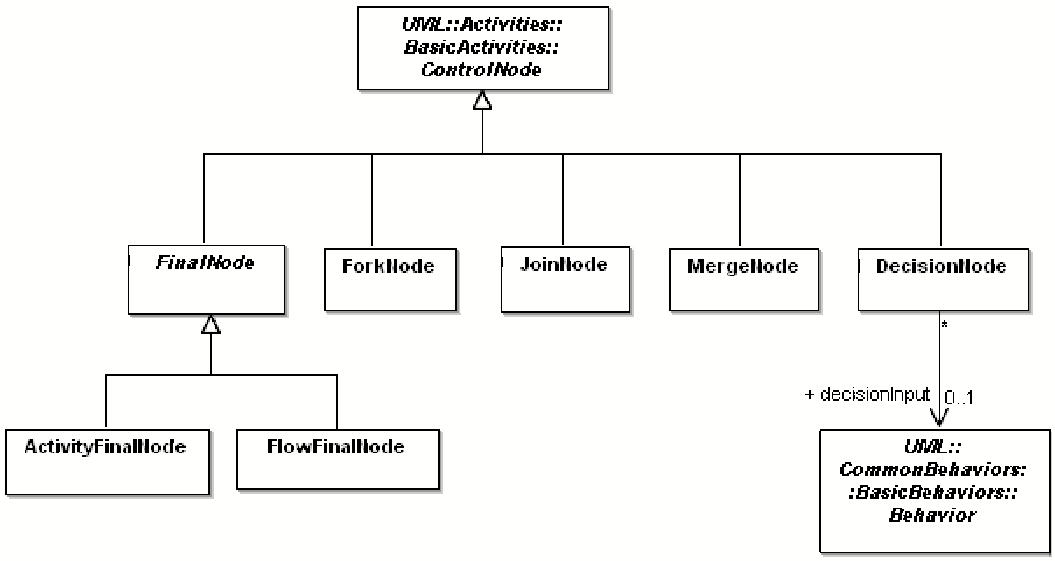

Figure 3: UML control nodes supported by the translation algorithm for PEPA.

pliance Case, except ETM (Enhanced Time Modelling): GRM (Generic Resource Modelling), NFP (Non-Functional Properties), GAM (Generic quantitative Analysis Modelling), and PAM (Performance Analysis Modelling). For the sake of completeness here we briefly summarise the relevant concepts with regards to the interpretation for PEPA.

Performance evaluation may be carried out on activities stereotyped with «GaScenario». Its *cause* property allows the extraction of workload specification, stereotyped with «GaWorkloadEvent». Closed patterns are supported, which define the workload as a population of users which interpose some thinking time between successive, cyclic executions of the activity. We support specification of synchronous activities as behaviour of action nodes to any depth of nesting, ultimately leading to single atomic steps, i.e. behaviours which are not further decomposable. This is to support iterative development methodologies whereby operations originally described as atomic are later turned into more fine-grained sub-activities of the scenario. We refer to those as *sub-scenarios* as opposed to the activity stereotyped with «GaScenario» which is called a *top-level scenario*.

The atomic units of execution are stereotyped with «PaStep». To denote the amount of time taken by a step we use its *hostDemand* attribute. Meaningful applications will typically have `hostDemand = (exp(<time>), s)` to indicate an exponentially distributed delay with mean `<time>` seconds. With similar considerations to [19] we take the *infinite resource* assumption. Every step is assumed to be executed by a dedicated process on exclusive processors—concurrency is introduced solely by the synchronisation points in the activity diagrams. Consequently, the attributes *host* and *concurRes* of a step need not be specified, nor can they be inferred from the location execution or from deployment. Swimlanes may be used, however they cannot explicitly represent the process executing the steps (via application of the «PaRunTInstance», for example).

4.1 Example

Figure 4 depicts the activity diagram which will be used throughout the remainder of this paper to illustrate a concrete application of the translation algorithm. (It is an adaption of the activity diagram shown in [27], p 97, with the *Pay* action expanded into a more fine-grained sub-scenario.) To reduce clutter the stereotype attributes of interest are shown separately in Table 1.

Table 1: Stereotype attributes for the activity in Fig. 4

Request Service	
PaStep	hostDemand=(exp(1/r_1), s)
GaWorkloadEvent	pattern=closed(population=10, extDelay=(exp(1/*think*), s))
Pay and Wait	
PaStep	hostDemand=(exp(1/r_2), s)
Take Order	
PaStep	hostDemand=(exp(1/r_3), s) throughput=out:orderTh
Fill Order	
PaStep	hostDemand=(exp(1/r_4), s)
Delivery	
PaStep	hostDemand=(exp(1/r_5), s)
Deliver Order	
PaStep	hostDemand=(exp(1/r_6), s)
Collect Order	
PaStep	hostDemand=(exp(1/r_7), s)
Select Payment Method	
PaStep	hostDemand=(exp(1/r_8), s)
Charge Credit Card	
PaStep	hostDemand=(exp(1/r_9), s) utilization=out:ccUtil
Make Bank Transfer	
PaStep	hostDemand=(exp(1/r_{10}), s)

The activity starts off with one flow requesting some service. As well as representing a unit of computation, this step is stereotyped with «GaWorkloadEvent», thus describing the external workload on the modelled system. In this case study, the workload comprises a population of 10 components with thinking time between successive requests with mean 1/*think* seconds. The completion of this action triggers the execution of two concurrent flows, concerned with payment and order

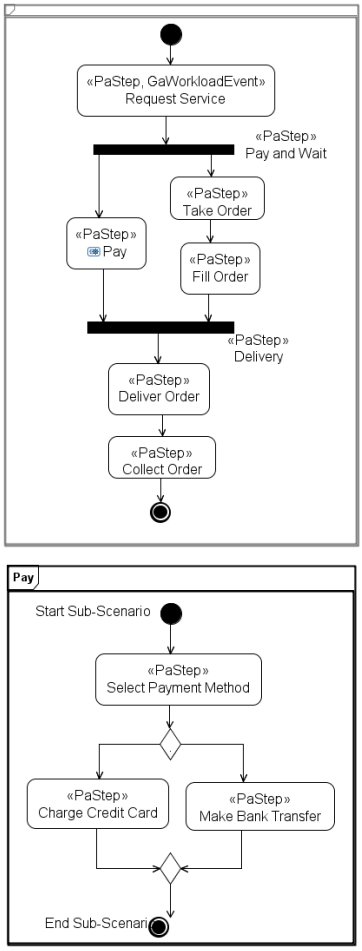

Figure 4: Running example. Top-level scenario (top diagram) and sub-scenario for the *Pay* action (lower diagram).

processing. Once they both complete, the main flow continues with delivery and collection, modelled as a sequence of steps. The *Pay* action expands into a sub-scenario where a choice is made between two available methods of payment. The sub-scenario ends when the payment is completed.

The corresponding PEPA model is shown in Fig. 5. The main concept underpinning the translation of the activity is to model each flow in the activity as a PEPA sequential component. A fork node with M outgoing edges introduces $M-1$ other flows, one being the continuation of the flow coming in. Conversely, at join nodes with N incoming edges, $N-1$ flows end, and the remaining flow continues with the behaviour described by outgoing edge. The algorithm visits the activity diagram in such a way that the flow which initiates the activity is let through the join nodes until the activity is terminated. This is accomplished via a *depth-first* visit of the activity diagram from the initial node. In the example this *main* flow performs the actions {*RequestService*, *Pay*, *DeliverOrder*, *CollectOrder*}, whereas the second flow performs {*TakeOrder*, *FillOrder*}. Notice that the main component is also assigned the execution of the *Pay* sub-scenario. In addition, the choice as to which outgoing behaviour from a fork node to assign to the main component is arbitrary.

An alternative model is possible, in which the main component performs {*RequestService*, *TakeOrder*, *FillOrder*, *DeliverOrder*, *CollectOrder*} and the other flow is in charge of the *Pay* operation. Although not identical to the former, this model yields the same performance results, because the underlying Markov chain is the same. The PEPA description of this alternative case is shown in Fig. 6.

$$
\begin{aligned}
Think &\stackrel{def}{=} (doThink, think).Start \\
Start &\stackrel{def}{=} (doRequest, \top).Think \\
RequestService &\stackrel{def}{=} (doRequest, r_1).SelectPayment \\
SelectPayment &\stackrel{def}{=} (payAndWait, r_2). \\
&\quad (doSelectPayment, r_8).PaymentType \\
PaymentType &\stackrel{def}{=} CreditCard + BankTransfer \\
CreditCard &\stackrel{def}{=} (doCharge, r_9).EndSubScenario \\
BankTransfer &\stackrel{def}{=} (doTransfer, r_{10}).EndSubScenario \\
EndSubScenario &\stackrel{def}{=} (doDelivery, r_5).DeliverOrder \\
DeliverOrder &\stackrel{def}{=} (doDeliverOrder, r_6).CollectOrder \\
CollectOrder &\stackrel{def}{=} (doCollect, r_7).Finish \\
Finish &\stackrel{def}{=} RequestService \\
TakeOrder &\stackrel{def}{=} (payAndWait, \top). \\
&\quad (doTakeOrder, r_3).FillOrder \\
FillOrder &\stackrel{def}{=} (doFillOrder, r_4).Delivery_1 \\
Delivery_1 &\stackrel{def}{=} (doDelivery, \top).TakeOrder \\
System &\stackrel{def}{=} \Big((Think[10] \underset{doRequest}{\bowtie} RequestService) \\
&\quad \underset{doDelivery, payAndWait}{\bowtie} TakeOrder \Big)
\end{aligned}
$$

Figure 5: PEPA model extracted from the activity diagram in Fig. 4

The definitions for the workload components are not shown because they are unaffected. In both cases, synchronisation points are described by the shared action types *payAndWait*, which represents the fork, and *doDelivery*, which represents the join.

The workload is represented as an array of components which is composed with the system thus generated. Each element of the array is a two-state component cycling through the actions *doThink* and *requestService*, the latter being shared with the main component. It is worthwhile noticing that the description of the workload as a composition of independent automata may reduce the size of the underlying CTMC when aggregation techniques (see [12]) are employed during the exploration of the state space.

5. ALGORITHM

This section discusses how to perform automatic mapping of activity diagrams onto PEPA performance models. The procedure takes as input an activity diagram stereotyped with «GaScenario» and consists of three main steps. In the first step, a number of preconditions are checked and the diagram may be subjected to graph transformation. This is discussed in Sect.5.1. The second step performs a depth-first visit of the diagram by starting from the initial node. This is done by means of a function called **visitNode** whose generic signature takes as input the control node to be visited as well as a label uniquely identifying the flow in which the node is executed. The function returns a (PEPA) constant, which represents the initial component that starts off the execution of the visited node. To reduce clutter, we assume that i.e., **visitNode** behaves polymorphically according to the type of the node that

$$
\begin{aligned}
RequestService &\stackrel{def}{=} (doRequest, r_1).TakeOrder \\
TakeOrder &\stackrel{def}{=} (payAndWait, \top). \\
&\quad (doTakeOrder, r_3).FillOrder \\
FillOrder &\stackrel{def}{=} (doFillOrder, r_4).Delivery \\
Delivery &\stackrel{def}{=} (doDelivery, \top).DeliverOrder \\
DeliverOrder &\stackrel{def}{=} (doDeliverOrder, r_6).CollectOrder \\
CollectOrder &\stackrel{def}{=} (doCollect, r_7).Finish \\
Finish &\stackrel{def}{=} RequestService \\
SelectPayment &\stackrel{def}{=} (payAndWait, r_2). \\
&\quad (doSelectPayment, r_8).PaymentType \\
PaymentType &\stackrel{def}{=} CreditCard + BankTransfer \\
CreditCard &\stackrel{def}{=} (doCharge, r_9).EndSubScenario \\
BankTransfer &\stackrel{def}{=} (doTransfer, r_{10}).EndSubScenario \\
EndSubScenario &\stackrel{def}{=} (doDelivery, r_5).SelectPayment \\
System &\stackrel{def}{=} \Big(\big(Think[10] \underset{doRequest}{\bowtie} RequestService \big) \\
&\qquad \underset{doDelivery, payAndWait}{\bowtie} SelectPayment \Big)
\end{aligned}
$$

Figure 6: Alternative PEPA model from the activity diagram in Fig. 4. The main flows goes along the right hand side outgoing edge of the fork node.

is passed. The pseudocode for each supported control node is presented in Sect. 5.2, accompanied by references to the running example (cfr. Fig. 4) as a concrete application. The visit of the graph provides the PEPA description of the activity. In the third step, this is composed with the description of the applied workload to obtain the performance model of the whole system (Sect. 5.3). Finally, this section concludes with a discussion on the non-functional properties that are available as output results from the model analysis (Sect. 5.4).

On a side note, it should be pointed out that we are aware of recent work on PUMA [30], an intermediate representation of the annotated design model (called Core Scenario Model) which extracts only the information relevant to performance analysis, thus easing the process of mapping to the target analysis technique. However, in the remainder of this paper we deal with direct generation from UML models. This choice is mainly due to the fact that Core Scenario Models are extracted from models annotated with SPT, not MARTE. If a revision of PUMA took account of the differences between the two profiles, the mapping to PEPA would greatly benefit from such a framework.

5.1 Preconditions and Transformations

The main precondition that has to be checked is that activity diagrams amenable to automatic translation into PEPA models must be *directed acyclic graphs*, with a single *source*. Diagrams may have more than one sink, however only one can be an activity final node. The other sinks must be flow final nodes. Future work shall be concerned with the relaxation of such assumptions.

In accordance with the UML2 formal specification, control tokens are placed at all the nodes that have no incoming edges. If there are many, the execution of the activity will have different initial flows. We also recall that, as a consequence, initial nodes are not compulsory. Thus, the assumption on the number of sources restricts activities to having one initial flow, whose execution is started by the initial node. However, as far as the translation to PEPA is concerned this is a mild limitation because such a case may encompass activities with implicit initial nodes as well as activities with more than one initial flow. This can be accomplished by subjecting the original activity to a pre-processing stage in which all the nodes with no incoming edges fork explicitly from an initial node.

Algorithm 1 Pre-processing of activity to allow multiple initial flows.

```
initialNode ⇐ ∅
floatingNodes ⇐ ∅
for Node n in Activity do
  if size(n.incomingEdges) == 0 then
    if n is InitialNodeType then
      if initialNode == ∅ then
        initialNode ⇐ n
      else
        initialNode.addOutgoingEdges(n.outgoingEdges)
        delete n
      end if
    else
      add n to floatingNodes
    end if
  end if
end for
if initialNode == ∅ then
  initialNode ⇐ new InitialNodeType
end if
if size(floatingNodes) + size(initialNode.outgoingEdges) > 1
then
  fork ⇐ new ForkNode
  fork.outgoingEdges ⇐ InitialNode.outgoingEdges
  fork.addIncomingEdge(initialNode)
  for Node n in floatingNodes do
    fork.addOutgoingEdge(n)
  end for
else
  if size(floatingNodes) == 1 then
    initialNode.addOutgoingEdges(floatingNodes)
  end if
end if
```

Algorithm 1 shows how to perform this transformation. Examples of applications of this algorithm are shown in Fig. 7. This transformation preserves the property that control tokens are available at such nodes only when the execution of the activity is started. From the point of view of performance analysis, however, the two graphs are not equal if the activity is interpreted against the original PEPA semantics which do not allow immediate actions. In this case, the execution of the nodes is delayed by the action performed at the fork node, whose duration is not instantaneous.

In the remainder of this section it is assumed that initial nodes have only one outgoing edge. In fact, in the UML

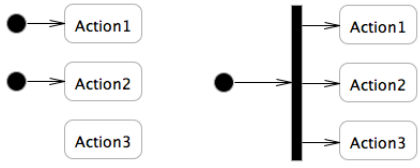

Figure 7: Graph transformation for single-source activities. Original activity (left) and transformed activity (right).

an initial node with multiple outgoing edges may be considered equivalent to a topology in which a central buffer node is inserted between the node and the edges. A flow is then selected by a non-deterministic choice. However, for the purposes of software performance evaluation, the modeller may replace non-determinism with probabilistic choice by interposing a decision node between the initial node and the edges. A procedure may be used during the pre-processing state to perform such a transformation.

Control nodes are also assumed to have only one outgoing edge. As the UML treats multiple outgoing edges as an implicit fork, this requirement does not reduce the expressiveness of the underlying performance model, though it helps reduce the clutter of the algorithm description. Again, we may think of a pre-processing stage which creates explicit fork nodes in such cases.

5.2 Node Visit Algorithms

In describing the algorithms we will make use of some utility functions. **getPEPAConstant**(Node) creates a PEPA constant for a given node in the activity graph. Here we do not want to have two different nodes with the same constant and we cannot rely on the node's name because the UML does not require them to be unique. Similarly, **getActionType** creates a unique PEPA action type for a given activity node.

We also make use of globally-visible FIFO queues for each flow identifier. Such queues record the flow's alphabets, i.e. the PEPA constants along the flow as they are found during the visit of the graph. These operations are performed by Algorithm 2. After creating the constant, it updates the system equation if the input node is the first element of the flow. Of particular interest is the first element of the queue which is the initial local state of the flow. When the flow terminates (because of a join), the first element is used to reset the local component. As far as the extraction algorithm is concerned, only the head of the queue is strictly necessary. However, the whole alphabet of the sequential component comes into play when the model's output variables are to be processed, as discussed in Sect. 5.4.

Algorithm 2 Create Constant for Node $node$, flow $flow$

Require: S current system equation
  Constant $C \Leftarrow$ **getPEPAConstant**($node$)
  enqueue C into $flow$.alphabet
  if size($flow$.alphabet) == 1 **then**
    if $S == \emptyset$ **then**
      $S \Leftarrow C$
    else
      coop $\Leftarrow$ **new Cooperation**
      coop.leftHandSide $\Leftarrow S$
      coop.rightHandSide $\Leftarrow C$
      $S \Leftarrow coop$
    end if
  end if
  return C

As we visit the graph, we keep track of the first action that is executed by the system. This may be either an atomic action executed by a node, or a fork action. This information will be used to create a synchronisation point between the closed workload and the activity model. This action type is held in the *WorkloadSyncAction* variable. As is common with graph exploration algorithms, we assume that there exists a function to mark nodes that have been already visited. This is of particular importance during the visit of join nodes (see Algorithm 6).

The function **getParent** alters the system equation of the underlying PEPA performance model. It takes a constant as input and returns the cooperation operator in the system equation which is the parent of that PEPA constant. If the prerequisite that the constant is known to appear in the system equation is satisfied, the parent exists and is unique because a PEPA model's system equation is a binary tree.[2] The tree is manipulated using its node's attributes *leftHandSide* and *rightHandSide*, holding pointers to its children. The *actionSet* attribute offers a pointer to the cooperation's action set.

The function **getRate** is called on nodes that in the PEPA interpretation represent atomic units of execution (forks, joins, and action nodes), and return the parameter of the exponentially distributed variable associated to that action. That is inferred from the *hostDemand* property of the stereotype «PaStep» which must be applied to such nodes.

Finally, as stated above activities have only one final activity node. As this will be translated to a PEPA constant, we define a function **getFinalNode** that returns the PEPA term corresponding to the final node.

5.2.1 Initial nodes

Algorithm 3 shows the pseudocode for the initial node of the activity, which simply visits its sole outgoing node. In the example, *Request Service* is visited.

Algorithm 3 Visit Initial Node $node$, flow $flow$

  return visitNode($node$.target, $flow$)

5.2.2 Action Nodes

Algorithm 4 is concerned with the transformation of an action node, i.e., an atomic unit of execution along a thread's flow, which will be mapped onto a PEPA prefix. Information on the duration of the activity is gathered from the «PaStep» stereotype which must be applied to the node. In particular, the *hostDemand* non-functional property will be processed. Because of the Markovian interpretation of the performance model, values must indicate an exponentially distributed delay. In MARTE, this can be accomplished by using VSL expressions as in Table 1.

Algorithm 4 Visit Action Node $node$, flow $flow$

  if isFirstActionVisited == **false then**
    WorkloadSyncAction $\Leftarrow doAction_i$
    isFirstActionVisited = **true**
  end if
  $C \Leftarrow$ **createConstant**($node$, $flow$)
  TargetTerm $\Leftarrow$ **visitNode**($node$.target, $flow$)
  rate $\Leftarrow$ **getRate**($node$)
  action $\Leftarrow$ **getActionType**($node$)
  add $C \stackrel{def}{=} (action, rate).TargetTerm$
  return C

For instance, *Request Service* is an action node that is processed with this algorithm. The target term in this case is the

[2] Here we are not concerned with degenerate cases where there is only one sequential component in the system.

constant corresponding to *Select Payment Method*. The component returned is *RequestService*. Notice that *Request Service* is the first action to be visited, hence the action type the system workload synchronises upon is *doRequest*. In addition, this action is the first of its flow. When the constant is created, manipulation of the PEPA system equation takes place to add this new sequential component. The same manipulation applies when *TakeOrder* is visited, because it represents the initial component of the second flow.

5.2.3 Fork Nodes

Algorithm 5 describes the mapping of fork nodes. The role of this algorithm is twofold. First, it create new flows. The outgoing edge that is first visited is assigned the same flow as the incoming edge. A new flow identifier is given to each of the remaining edges. This is how sequential components are updated in the system equation. Second, it modifies the PEPA system equation by making the outgoing flows synchronise over the *fork* action which is uniquely assigned to that node. In this framework a fork is a non-instantaneous action which must be stereotyped with «PaStep» to retrieve the duration associated with it. The resulting *multi-way* synchronisation between the outgoing flows is such that the overall rate of the shared activity is dominated by the component which is first visited. It is assigned the delay specified in the «PaStep» attributes, whereas the other components are passive.

Algorithm 5 Visit Fork Node $node$, flow $flow$

```
forkAction ⇐ getActionType(node)
if isFirstActionVisited == false then
    WorkloadSyncAction ⇐ forkAction
    isFirstActionVisited = true
end if
isFirstChild ⇐ true
for edge ∈ node.outgoingEdges do
    if isFirstChild == true then
        flow c ⇐ flow
        rate ⇐ getRate(node)
    else
        flow c ⇐ new flow
        rate ⇐ ⊤
    end if
    TargetTerm ⇐ visitNode(edge, flow)
    prefix (forkAction,rate) to TargetTerm
    if isFirstChild == true then
        isFirstChild ⇐ false
        FirstChild ⇐ TargetTerm
    else
        coop ⇐ getParent(TargetTerm)
        add {forkAction} to coop.actionSet
    end if
end for
return FirstChild
```

A concrete application of this algorithm is illustrated by the components *Pay* and *TakeOrder* in the example. Here, the call action node *Pay* is executed within the first flow. On the other hand, *TakeOrder* is the initial state of the second flow. The fork that starts off its execution is modelled as the cooperation on the shared action *payAndWait*. (In the example, the action type is named after the node's name.) Finally, notice that the component that is first visited executes at an active rate. Here, the active component is *SelectPayment*, the PEPA component mapping the first action node of the sub-scenario. On the other hand, *TakeOrder* is given a passive rate.

5.2.4 Join Nodes

Join nodes may have multiple incoming edges but must have only one outgoing edge. The algorithm for such nodes (see Algorithm 6) assigns the outgoing edge the same flow as the incoming edge that first visits the join node. All the other flows terminate their execution. The analysis of steady-state behaviour is an important preliminary to the main purpose of software performance evaluation. For the system to exhibit such behaviour, components must cycle. Hence, flow termination is interpreted as the component going back to its initial state, as recorded in the flow's alphabet data structure.

Algorithm 6 Visit Join Node $node$, flow $flow$

```
joinAction ⇐ getActionType(node)
if isFirstVisit(node) then
    TargetTerm ⇐ visitNode(node.target, flow)
    rate ⇐ getRate(node)
    JoinProcess ⇐ Join_i =def= (joinAction,rate).TargetTerm
    coop ⇐ getParent(TargetTerm)
    add {joinAction} to coop.actionSet
else
    InitialTerm ⇐ flow.alphabet.first()
    add joinAction to parent cooperation of InitialTerm
    JoinProcess ⇐ Join_{i,f} =def= (joinAction, ⊤).InitialTerm
    setFirstVisit(node)
end if
return JoinProcess
```

The treatment of join nodes is symmetrical to that of fork nodes. The join's unique action type is retrieved to add this synchronisation point to the system equation. A join is modelled as a multi-way synchronisation between all the components which model its incoming flows. A join node is visited as many times by the algorithm as the number of its flows. The first flow which visits the join is assigned an active rate, the other being passive. Similar arguments to the case of fork nodes hold for the choice of the active component.

With respect to the sample activity diagram, the behaviour of the join *Delivery* is expressed by the two components *Delivery* and $Delivery_1$. *Delivery* is a local derivative of the first flow, exposing the action *doDelivery* which is to be executed in cooperation with $Delivery_1$, a local derivative of the second flow. After the action is performed, the first component will behave as the outgoing node from the join, thus it is assigned an active rate. Instead, $Delivery_1$ is passive and loops back to its initial state.

5.2.5 Decision and Merge Nodes

Although decision and merge nodes share the same notation, decision nodes choose between flows, while merge nodes bring together multiple alternate flows. We model decision nodes as PEPA choices (see Algorithm 7). For example, the decision node in the sub-scenario is modelled with the process definition $PaymentType \stackrel{def}{=} CreditCard + BankTransfer$.

The algorithm for merge nodes simply returns the term of their target node (see Algorithm 8).

5.2.6 Call Action Nodes

Algorithm 9 is concerned with the mapping of *Call Action* nodes. Although such nodes may indicate any UML behaviour

Algorithm 7 Visit Decision Node $node$, flow $flow$

$C \Leftarrow$ **createConstant**($node$, $flow$)
$ChoiceSet \Leftarrow \emptyset$
for target $\in node$.targets **do**
  $TargetTerm \Leftarrow$ **visitNode**(target, $flow$)
  add $TargetTerm$ to $ChoiceSet$
end for
add $C \stackrel{def}{=}$ createChoice($ChoiceSet$)
return def

Algorithm 8 Visit Merge Node $node$, flow $flow$

return **visitNode**($node.target, flow$)

(such as, for instance, interactions, state machines, or use cases), here we restrict ourselves to a call of an activity, which represents a sub-scenario. This activity inherits the flow identifier of the incoming edge to the node. Another task performed by the algorithm is to link the final activity node of the nested activity with the outgoing edge of the node (otherwise the nested activity's final node would be mapped as a term going back to the initial term of the nested activity).

Algorithm 9 Visit Call Behaviour Node $node$, flow $flow$

activity $\Leftarrow node$.getNestedActivity()
$StartTerm \Leftarrow$ **visitNode**(getInitialNode(activity), $flow$)
$TargetTerm \Leftarrow$ **visitNode**($node$.target, $flow$)
$EndTerm \Leftarrow$ activity.getFinalNodeTerm()
$EndTerm$.rightHandSide $\Leftarrow$ $TargetTerm$.rightHandSide
remove $TargetTerm$
return $StartTerm$

In the example, *StartTerm* will be the first action of the sub-scenario, *SelectPayment*. *TargetTerm* will be (*doDelivery*, 0.5). *DeliverOrder*. Before the substitution of the two right hand sides, *EndTerm* is defined as $EndTerm \stackrel{def}{=} SelectPayment$.

5.2.7 Final Nodes

The UML has two kinds of control nodes that stop an activity flow: *ActivityFinalNode* and *FlowFinalNode*. When a token reaches the former, all the concurrent flows in the activity are terminated. The latter terminates the flow that arrives at it with no effect on the others. We model flow final nodes as PEPA terms that make the sequential component loop back to its initial state. The algorithm is straightforward and is shown in Algorithm 10.

Algorithm 10 Visit Flow Final Node $node$, flow $flow$

return $flow.alphabet.first()$

The visit of an activity final node is similar, however a process definition is added to the model description. If the node belongs to a sub-scenario, it serves as a the placeholder that will be manipulated during the visit of the call action node, as discussed in 5.2.6. The top-level scenario's activity final node is termed *Finish* and corresponds to the main flow's initial state.

5.3 Workload

Once the translation of the activity is completed, the system is ready to be composed with the components that model the applied workload. The procedure is shown in Algorithm 12.

Algorithm 11 Visit Activity Final Node $node$, flow $flow$

$C \Leftarrow$ **createConstant**($node$, $flow$)
$TargetTerm \Leftarrow flow$.alphabet.first()
add $C \stackrel{def}{=}$ TargetTerm
return C

Workloads are supported in the form of *closed arrival patterns*. Each individual is translated as a sequential component which cycles through the following local states: an idle state where the component spends some thinking time, and a state where the execution of the activity is triggered. Those two states correspond to the *Thinking* and *Requesting* definitions, respectively. The former performs an unshared action *delayAction*, the latter synchronises with the rest of the system over the first action that is performed by the activity. Here, the workload component is passive with respect to this cooperation.

The algorithm requires that the *cause* of «GaAnalysisContext» be a «GaWorkloadEvent» whose *pattern* property is a *ClosedPattern*. The actual parameters of the workload are extracted from the non-functional properties *population* and *extDelay*.

Algorithm 12 Composition of workload

$Thinking \stackrel{def}{=} (delayAction, extDelay).Requesting$
$Requesting \stackrel{def}{=} (WorkloadSyncAction, \top).Thinking$
newcoop $\Leftarrow$ **new Cooperation**
newcoop.leftHandSide $\Leftarrow Thinking[population]$
newcoop.rightHandSide $\Leftarrow$ systemEquation
add *WorkloadSyncAction* to newcoop.actionSet
systemEquation $\Leftarrow$ newcoop

5.4 Output variables

In MARTE, performance measures of interest to the modeller may be specified as output variables denoting non-functional properties of the system. We shall be concerned with the specification of performance indices related to the steady-state behaviour of the system. Particularly, a set of procedures of the mapping algorithm is devoted to extracting output variables for *utilisation* and *throughput*. Both indices are directly supported by MARTE via attributes of «PaStep» (as inherited attributes from «GaScenario»). As discussed in Sect. 5.2, the stereotype «PaStep» is applied to fork nodes, join nodes, action nodes, and call action nodes. In accordance with the semantics of PEPA, however, output variables to such nodes cannot be specified indiscriminately. The purpose of this paragraph is to discuss in which cases output variables are meaningful.

Action nodes may have output variables regarding utilisation as well as throughput measures. An example of throughput measure is the `out:orderTh` variable assigned to the *throughput* property of *Take Order*. Utilisation is specified as in the *utilization* property of *Charge Credit Card* through the variable `out:ccUtil`. Fork and join nodes support the specification of throughput variables only. The result indicates the rate at which the underlying shared action is performed by the system in the steady state. All the remaining nodes and edges in the diagram do not support any other specification of output variables.

5.4.1 Calculation of utilisation

Let N be the number of sequential components in the system and $M < N$ be the workload population. The generic state $s_i \in S$ of the CTMC underlying the PEPA performance model may be denoted as $(Workload_i, System_i)$, $1 \le i \le |S|$, where $Workload_i$ is an M-tuple describing the state of the workload components, and $System_i$ is the $(N-M)$-tuple of the activity state. Let the system tuple be indexed by $j : 1 \le j \le (N-M)$. We denote as $System_{i,j}$ the element at index j of the system tuple of state i. Let $\mathscr{A}_j$ be the alphabet of the flow modelled by the element j of the system tuple. We have that $\mathscr{A}_j \subseteq \bigcup_{1 \le i \le |S|} System_{i,j}$. Finally, let π be the equilibrium distribution of the CTMC. The utilisation $\mathscr{U}_{a_j}$ of a local derivative $a_j \in \mathscr{A}_j$ is calculated as follows:

$$\mathscr{U}_{a_j} = \sum_i \pi_i, 1 \le i \le |S|, System_{i,j} = a_j$$

In the model in Fig. 5, $N = 12$, $M = 10$. Suppose we wish to calculate the utilisation of the action node *FillOrder*. That is a local derivative of the second flow, whose alphabet is $\mathscr{A}_2 = \{FillOrder, TakeOrder\}$. Notice that this is only a subset of the states of the flow, as the unnamed state $(doTakeOrder, 0.33)$. *FillOrder* is not a member of the alphabet. However, according to Algorithm 4, every action node is assigned a labelled local derivative, which is added to the flow's alphabet. In this case, the node's label is identical to the local state's name and it is the value of a_j to be used in the formula for utilisation.

5.4.2 Calculation of throughput

The throughput of an action α^* can be calculated from the labelled transition system of the underlying PEPA model as follows. Let $\mathscr{B}_{s_i}$ be set of activities enabled by state s_i. The throughput $\mathscr{T}(\alpha^*)$ is:

$$\mathscr{T}(\alpha^*) = \sum_i \pi_i \cdot \sum \{r : (\alpha, r) \in \mathscr{B}_{s_i}, \alpha = \alpha^*\}$$

Each activity node whose throughput calculation is allowed is assigned an unique action type. For instance, the throughput of the node *Fill Order* is computed as $\mathscr{T}(doFillOrder)$. The throughput of the fork action *Pay and Wait* is obtained by calculating $\mathscr{T}(payAndWait)$.

6. TOOL SUPPORT

The translation algorithm has been implemented as a plug-in for Rational Software Architect (RSA) v7, a leading UML modelling tool. The plug-in is based on the implementation of the MARTE profile for RSA, available at the OMG MARTE web site [22].

Rational Software Architect is implemented on the Eclipse development environment and can be extended by special-purpose software services. Our software tool contributes an Eclipse view (*Performance Results*) to the platform, from which all the operations concerning the extraction and analysis of the PEPA performance model are accessible. The view is linked with the active editor of the Rational Software Architect workbench, and can be in one of the following states:

Inactive If the active editor is not showing an activity diagram or the activity diagram is not stereotyped with GaScenario.

Model Extracted When the underlying PEPA performance model has been generated.

Model Analysed If the analysis of the performance model has been successfully completed.

Model extraction and analysis is carried out by using services of PEPAto, the Java-based API for PEPA to manipulate the model's abstract syntax tree and run the CTMC solvers. PEPAto is a constituent plug-in of the PEPA Eclipse Plug-in Project, which is fully compatible with RSA v7.x.

When the performance model is extracted, the view is populated with the output variables which define the performance measures of interest to the modeller. The variable are presented in a tree-like fashion, according to the «ExpressionContext» in which they are defined. Performance analysis is not automatically run after model extraction, rather it is triggered by a toolbar menu item in the view. When the model is solved, the tree is updated with the values as calculated by the analyser.

In the spirit of MARTE, the tool has been designed to hide the underlying formal method and its underpinnings and present the modeller with an interface which exposes concepts only from the vocabularies of the UML and MARTE. Thus modellers using the tool do not need to be familiar with process algebras in general or with PEPA in particular. The PEPA process algebra plays the role of an intermediate language here, allowing access to the powerful solution methods made available by the PEPA tools, without having to work in the PEPA language directly.

Nevertheless, the tool also features a *preview* option for the inspection of the process calculus model which may give a more insighful perspective to the experienced theoretician. It is worth pointing out that the PEPA model can be imported into the RSA workbench and independently subjected to the range of tools provided by the PEPA Eclipse project.

The implementation of our translation of UML activity diagrams to PEPA is available in our plug-in for RSA. This can be downloaded from `http://homepages.inf.ed.ac.uk/mtribast/uml`, providing further information on the product and a link to the Eclipse update site from which the plug-in as well as its dependencies can be downloaded.

7. CONCLUSION

We presented a procedure to systematically map UML activity diagrams into PEPA models, which can be analysed through the solution of the underlying CTMC. Diagram annotations with the profile for MARTE are used to specify the timing behaviour of the actions and to denote the output variables of concern to the modeller. In this paper we showed how to extract steady-state performance indices such as throughput and utilisation. Nevertheless, PEPA enables other forms of analysis, including transient analysis, and model checking. We consign to future work extension of this approach in order to make such analysis techniques available in the context of software engineering.

Because of the Markovian interpretation of the performance model, our approach is restricted to activities with only activity final node. (As discussed above, the diagram may have other sinks, which must be flow final nodes.) Recall that, in the UML, all concurrent flows are terminated when one reaches a final node. However, the interleaving semantics for PEPA does not allow us to faithfully model such a behaviour. For instance, let us consider a system with two independent PEPA components A and B evolving through a number of local states $A_1, A_2, \ldots A_M$ and $B_1, B_2, \ldots B_N$, respectively. If we wished to

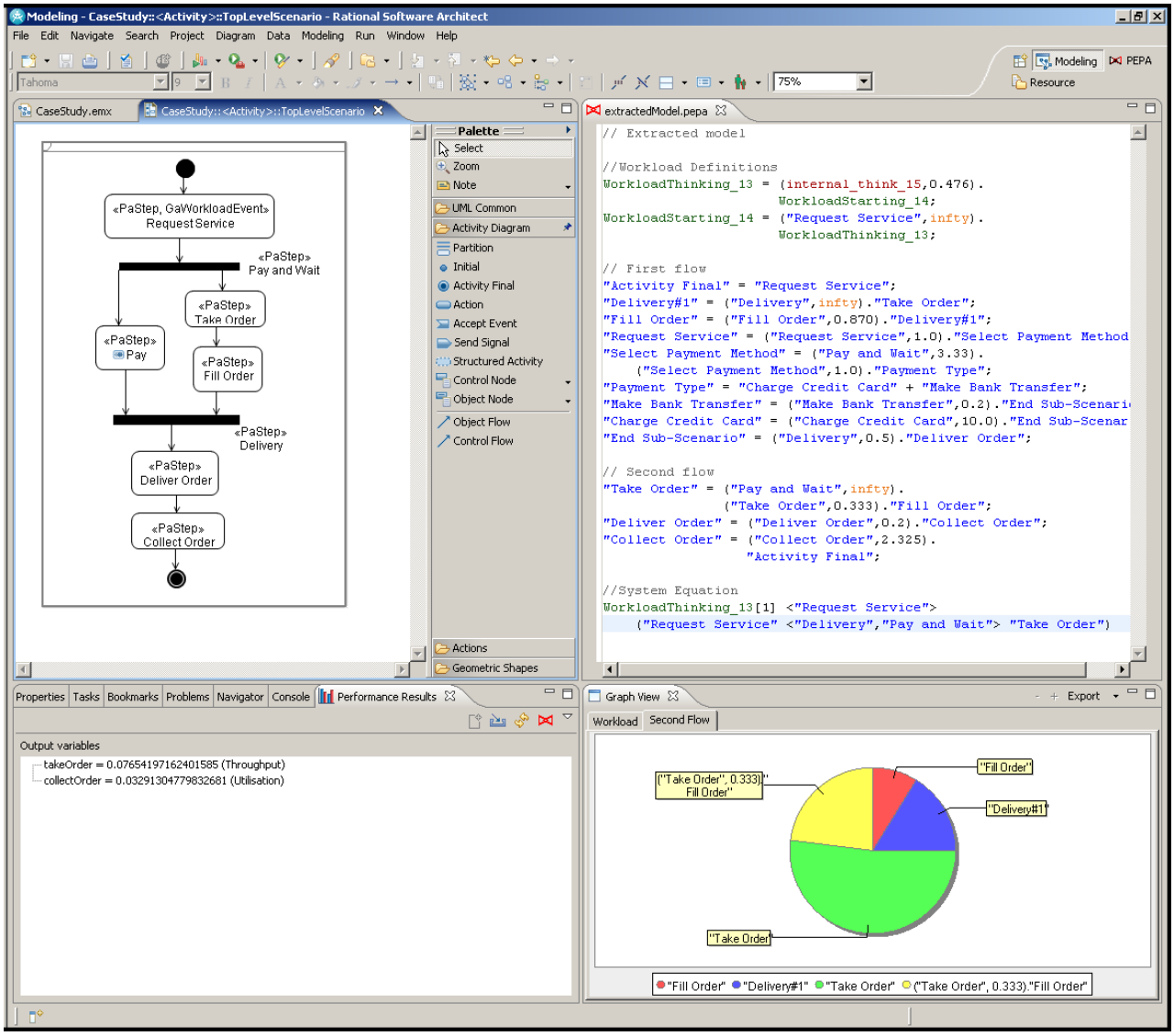

Figure 8: Rational Software Architect v 7.0 featuring the plug-in for the extraction of PEPA models from activity diagrams. On the left hand side the workbench shows the top-level scenario of the case study. Below is the *Performance Results* view from which all the extraction tasks are accessible. It currently displays the model's output variables and the latest values. On the right hand side is the description of the extracted model opened with the PEPA Eclipse Plug-in Project. The *Graph View*, one of its constituent views, displays the utilisation of the sequential component modelling the second flow of the system.

model a behaviour in which A may terminate each B, we could define a shared action *final* which can be executed by B at any state. Suppose that A_M performs such an activity. Upon the execution of the shared action, the systems goes back to its initial state. The model description is listed thus. The dots denote actions that are performed independently.

$$
\begin{aligned}
A_1 &\stackrel{def}{=} \ldots \\
A_2 &\stackrel{def}{=} \ldots \\
&\ldots \\
A_M &\stackrel{def}{=} (final, r).A_1 \\
B_i &\stackrel{def}{=} \ldots + (final, \top).B_1 \\
&\qquad 1 \le i \le N \\
\\
System &\stackrel{def}{=} A_1 \underset{final}{\bowtie} B_1
\end{aligned}
$$

Let us now consider the sub-space of the underlying CTMC where the final action is enabled, that is, $\{A_M \underset{final}{\bowtie} B_i : 1 \le i \le N\}$. In any state, *final* is not the only enabled action as B_i can perform its independent actions as well. Since PEPA is not equipped with semantics for priorities, there is a non-zero probability for that state not to perform the final action. So long as this notion of *graceful* termination of activities is taken into account, it is possible to adapt the proposed algorithms to support diagrams with multiple action final nodes. It is also worthwhile pointing out that in such an extension the modeller has control over the probability that the *final* action is performed by adjusting the rate associated to the activity.

Acknowledgments.

This work has been supported by the project SENSORIA, IST-2005-016004.

8. REFERENCES

[1] *Proceedings of the Fifth International Workshop on Software and Performance, WOSP 2005, Palma, Illes Balears, Spain, July 12-14, 2005*. ACM, 2005.

[2] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Software Eng.*, 30(5):295–310, 2004.

[3] S. Balsamo and M. Marzolla. Performance evaluation of UML software architectures with multiclass queueing network models. In *WOSP* [1], pages 37–42.

[4] S. Bernardi, S. Donatelli, and J. Merseguer. From UML Sequence Diagrams and Statecharts to analysable Petri Net models. In P. Inverardi, S. Balsamo, and B. Selic, editors, *Proceedings of the Third International Workshop on Software and Performance*, pages 35–45, Rome, Italy, July 2002. ACM.

[5] J. T. Bradley and S. T. Gilmore. Stochastic simulation methods applied to a secure electronic voting model. *Electr. Notes Theor. Comput. Sci.*, 151(3):5–25, 2006.

[6] C. Canevet, S. Gilmore, J. Hillston, L. Kloul, and P. Stevens. Analysing UML 2.0 Activity Diagrams in the Software Performance Engineering Process. In Dujmovic et al. [9], pages 74–78.

[7] C. Canevet, S. Gilmore, J. Hillston, M. Prowse, and P. Stevens. Performance Modelling with UML and Stochastic Process Algebras. *IEE Proceedings: Computers and Digital Techniques*, 150(2):107–120, Mar. 2003.

[8] A. D'Ambrogio and P. Bocciarelli. A model-driven approach to describe and predict the performance of composite services. In V. Cortellessa, S. Uchitel, and D. Yankelevich, editors, *WOSP*, pages 78–89. ACM, 2007.

[9] J. J. Dujmovic, V. A. F. Almeida, and D. Lea, editors. *Proceedings of the Fourth International Workshop on Software and Performance, WOSP 2004, Redwood Shores, California, USA, January 14-16, 2004*. ACM, 2004.

[10] Eclipse Foundation. Eclipse UML2 Project Home Page. `http://www.eclipse.org/uml2/`.

[11] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. Software Performance Modelling Using PEPA Nets. In Dujmovic et al. [9], pages 13–23.

[12] S. Gilmore, J. Hillston, and M. Ribaudo. An efficient algorithm for aggregating PEPA models. *IEEE Transactions on Software Engineering*, 27(5):449–464, May 2001.

[13] S. Gilmore and L. Kloul. A unified tool for performance modelling and predicition. In *Proceedings of the 22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP'03)*, number 2788 in LNCS, pages 179–192, Edinburgh, Scotland, Sept. 2003. Springer-Verlag.

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

[15] J. Hillston. Fluid flow approximation of PEPA models. In *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, pages 33–43, Torino, Italy, Sept. 2005. IEEE Computer Society Press.

[16] J. Hillston and Y. Wang. Performance evaluation of UML models via automatically generated simulation models. In S. A. Jarvis, editor, *Proceedings of the 19th Annual UK Performance Engineering Workshop*, pages 64–78, Warwick, UK, 2003.

[17] IBM Corporation. Rational Software Architect. `http://www-306.ibm.com/software/awdtools/architect/swarchitect/`.

[18] C. Lindemann, A. Thümmler, A. Klemm, M. Lohmann, and O. P. Waldhorst. Performance analysis of time-enhanced UML diagrams based on stochastic processes. In *Workshop on Software and Performance*, pages 25–34, 2002.

[19] J. P. López-Grao, J. Merseguer, and J. Campos. From UML activity diagrams to Stochastic Petri nets: application to software performance engineering. In Dujmovic et al. [9], pages 25–36.

[20] J. Merseguer, S. Bernardi, J. Campos, and S. Donatelli. A Compositional Semantics for UML State Machines Aimed at Performance Evaluation. In M. Silva, A. Giua, and J. Colom, editors, *Proceedings of the 6th International Workshop on Discrete Event Systems*, pages 295–302, Zaragoza, Spain, October 2002. IEEE Computer Society Press.

[21] J. Merseguer and J. Campos. Exploring Roles for the UML Diagrams in Software Performance Engineering. In *Proceedings of the 2003 International Conference on Software Engineering Research and Practice SERP03*, pages 43–47, Las Vegas, Nevada, USA, June 2003. CSREA Press.

[22] Object Management Group. OMG MARTE Tools. `http://www.omgmarte.org/Tools.htm`.

[23] Object Management Group. *UML Profile for Schedulability, Performance, and Time Specification. Version 1.1*. OMG, 2005. OMG document number formal/05-07-04.

[24] Object Management Group. *UML 2.2.1 Superstructure Specification*. OMG, 2007. OMG document number formal/05-07-04.

[25] Object Management Group. *UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE). Beta 1*. OMG, 2007. OMG document number ptc/07-08-04.

[26] D. C. Petriu and H. Shen. Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specifications. In T. Field, P. G. Harrison, J. T. Bradley, and U. Harder, editors, *Computer Performance Evaluation / TOOLS*, volume 2324 of *Lecture Notes in Computer Science*, pages 159–177. Springer, 2002.

[27] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley.

[28] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language User Guide*. Addison-Wesley.

[29] M. Tribastone. The PEPA Plug-in Project. In *Fourth International Conference on the Quantitative Evaluation of Systems*, pages 53–54, United Kingdom, September 2007. IEEE Computer Society.

[30] C. M. Woodside, D. C. Petriu, D. B. Petriu, H. Shen, T. Israr, and J. Merseguer. Performance by unified model analysis (PUMA). In *WOSP* [1], pages 1–12.