Analysing UML 2.0 activity diagrams in the software performance engineering process

C. Canevet, S. Gilmore, J. Hillston, L. Kloul[*] and P. Stevens

Laboratory for Foundations of Computer Science, The University of Edinburgh, Scotland

{ccanevet,stg,jeh,leila,perdita}@inf.ed.ac.uk

ABSTRACT

In this paper we present an original method of analysing the newly-revised UML2.0 activity diagrams. Our analysis method builds on our formal interpretation of these diagrams with respect to the UML2.0 standard. The mapping into another formalism is the first stage of a refinement process which ultimately delivers derived analytical results on the model. This process highlights latent performance problems hidden in the high-level design, allowing software developers to fix these design flaws before they are concretised in implementation code. We exercise our analysis approach on a substantial example of modelling a multi-player distributed role-playing game.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies*; D.3.2 [**Programming Languages**]: Language Classifications; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

UML2.0; PEPA; PEPA nets; performance evaluation

1. INTRODUCTION

Performance models of computer systems are used to gain insights into the behaviour of the system under expected workload on the available hardware. The state-of-the-art in the design of computer systems grows ever more sophisticated as programming languages become more complex; application programs increasingly use additional layers of middleware and infrastructure; and software developers deploy complex patterns and idioms to structure their application code. Similarly, the computing platform on which these applications execute becomes more complex. A wide range of computing devices may be deployed, from high-end servers to battery-powered handheld devices. Other layers of interpretation may also be used including virtual machines such as the JVM. Each such layer adds complexity and degrades performance.

Complex software necessitates the use of a systematic software design process in which initial high-level designs and blueprints are methodically refined towards an efficient and reliable implementation of the system. In addition to the programming language or languages which will ultimately be used to code the system, one or several modelling languages are usually deployed for design and analysis purposes. In this paper we explain how a general-purpose modelling language (the UML) can be used together with a special-purpose modelling language for performance analysis of distributed and mobile computing systems (the language of PEPA nets).

The Unified Modelling Language (UML) is an effective diagrammatic notation used to capture high-level designs of systems, especially object-oriented software systems. The UML is now considered to be the *de facto* standard for the high-level description of software systems, even in those cases where the primary interest in building these models is to undertake a performance analysis of the system under study [17].

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. Other dynamic diagrams, such as activity diagrams and sequence diagrams, show how the overall behaviour of the system is realised. As usual we expect that the UML modeller will make a number of diagrams of different kinds. Our analysis here is based on activity diagrams and complements our earlier work on mapping UML state diagrams and collaboration diagrams to PEPA [2].

The PEPA nets modelling language [4] is a high-level coloured stochastic Petri net formalism where the tokens of the net

[*]On leave from PRISM, Université de Versailles, 45, Av. des Etats-Unis 78000 Versailles, France.

are themselves programmable stochastically-timed components. The modelling language which is used for the tokens of a PEPA net is Hillston's Markovian process algebra PEPA (Performance Evaluation Process Algebra) [7]. The PEPA nets formalism is a recent research development and we are currently exploring its possibilities in tandem with developing its underlying theory [4]. To help to assess the practical usefulness of this modelling language we have undertaken a number of case studies including a peer-to-peer distributed file system [6]; the Jini discovery service and a mobile telephony scenario [4]; and the Mobile IP protocol [5].

In this paper we apply the UML and PEPA nets languages to the problem of modelling a complex distributed application, a multi-player online role-playing game. The game is one of the case studies from one of our industrial partners on the EC-funded DEGAS project (Design Environments for Global ApplicationS). The game is a characteristic "global computing" example, encompassing distribution, mobility and performance aspects. The representational challenges in modelling the game accurately include capturing location-dependent collaboration, multi-way synchronisation, and place-bounded locations holding up to a fixed number of tokens only. All of these are directly represented in the PEPA nets formalism.

Structure of this paper:
In Section 2 we summarise UML 2.0 activity diagrams. In Section 3 we provide an introduction to the PEPA nets modelling language. Section 4 explains the model refinement process at work, transforming UML 2.0 activity diagrams into PEPA nets for analysis. In Section 5 follows a high-level description of our modelling study, a role-playing game together with the PEPA net model of the game. The analysis of the system by the solution of this model is discussed in Section 6. Conclusions and remarks on future work follow.

2. UML 2.0 ACTIVITY DIAGRAMS

The Unified Modelling Language, UML [13], has been the dominant diagrammatic language for recording the design of systems, especially object-oriented software systems, since around 1997. UML 2.0 will be the next major release of the Unified Modelling Language. Its current status is that initial proposals have been accepted by the Object Management Group, the controlling standards body, and are now proceeding through the revision and voting process.

One of the major changes introduced in UML2.0 is a radical overhaul of activity diagrams. In this section we explain the main changes which have been introduced. We also describe the subset of UML2.0 activity diagrams which we address. It is necessary to choose a subset, because the whole language is extremely complex: the chapter in the draft specification [20] describing these diagrams alone runs to over 100 pages!

We begin with an informal explanation of the most basic notation, which is (in essence) common to both UML 1.x and UML 2.0. We will then go on to discuss briefly the problems with UML 1.x's view of activity diagrams, and how these have been addressed in UML2.0. Finally we discuss some new features of UML 2.0 activity diagrams.

Figure 1 shows a basic UML activity diagram. (We use

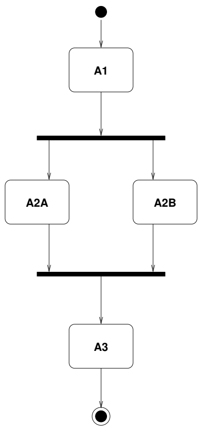


Figure 1: Basic activity diagram

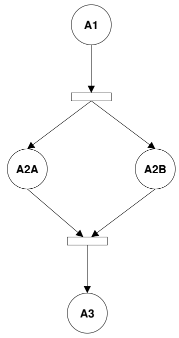


Figure 2: Basic Petri net

the shape of activity boxes used in UML2.0, but apart from this, it is equally valid in UML1.x or UML2.0, with the same meaning.) The boxes are activities. At any moment an activity is either active or inactive; several activities may be active simultaneously. The black circle at the top is a start marker; it indicates which activity is initially active. Lines are flows (transitions). Once an activity is complete, it becomes inactive, and the flow(s) from it are fired. The top thick bar is a "fork pseudostate": it indicates that the target activities are both made active simultaneously after the incoming flow is fired. The lower thick bar is a "join pseudostate" which indicates that once both of the source activities have completed, the target activity will be activated.

It is clear that this example behaviour can be modelled by a Petri net, as shown in Figure 2. Fork and join pseudostates become transitions, whilst activities become places. There is a token in a place exactly if the corresponding activity is active. The start marker in the activity diagram leading to activity $A1$ corresponds to the assumption that the initial marking of the Petri net is that there is a token on place $A1$ and no tokens elsewhere. This is a simple example of the new UML2.0 interpretation of activity diagrams. Note, however, that it does *not* correspond to our translation in this paper, because we have to take into account other features such as object flows, to be discussed.

Problems arise if we try to consider more complex examples. UML 1.x actually regards activity diagrams as special syntax for hierarchical state machines. A fork pseudostate indicates entry into a submachine consisting of several parallel regions, and a join pseudostate indicates exit from such a submachine. Therefore there are many activity diagrams which have an obvious interpretation which are not in fact legal in UML 1.x. For some further discussion of this issue see [18]. In practice, users of UML have often not obeyed the rules governing the structure of activity diagrams, and have informally used a Petri net semantics. In UML 2.0, this has been made official.

2.1 Object flows in UML 2.0

Object flows already existed in UML 1.x, but were so imprecisely defined that few practitioners made use of them. In UML 2.0 the situation has been improved. Essentially there are two kinds of flows, the normal control flows and object flows. The presence of a control token in an activity indicates merely that the activity is enabled, and flow of control tokens shows the enabling and disabling of activities. Object tokens, on the other hand, represent objects in the software system being defined. As such, they have state, behaviour and identity which may be used by the activities where they reside. The flow of object tokens shows part of the data flow of the application being designed. Control tokens flow along control flows, object tokens flow along object flows. For example, Figure 3 shows a control flow from activity reachRoom to activity fightNP, and an object flow from object NPlayer to activity fightNP. In the vocabulary of the UML 2.0 specification, an activity may require both control tokens and object tokens in order to be activated. For example, the activity fightNP cannot begin until both the activity reachRoom has been completed – so that a control token is passed on to fightNP – and the object NPlayer is available. Notice that UML2.0 tokens are not identical with basic Petri net tokens, since there is a notion that they have identity which is preserved through flows. If an activity requires two tokens to begin, it then possesses (those same) two tokens whilst it is active. As we shall see, this corresponds sensibly with the treatment of tokens in PEPA nets. We may thus view activity diagrams as a particular kind of coloured Petri net with two kinds of tokens: indistinguishable control tokens, and object tokens. UML2.0 appears to assume a sensible type discipline for object tokens, although this is not made formal. It will be an assumption of our translation that our UML activity diagrams are well typed.

We consider activity diagrams in which there is choice, looping, control and object flows, but no synchronisation, nor any use of the other features of UML2.0 activity diagrams not discussed here.

3. PEPA NETS

In this section we present a brief introduction to the PEPA nets language, emphasising the use of the language. Formal definitions and full details of the PEPA nets and PEPA languages are found in [4] and [7]. The summary which is presented here contains everything which is needed to understand the contribution which is made by the present paper.

3.1 Tokens

The tokens of a PEPA net are PEPA terms. These define the behaviour of components via the activities they undertake and the interactions between them. The component combinators, together with their names and interpretations, are presented informally below.

Prefix:

The basic mechanism for describing the behaviour of a system is to give a component a designated first action using the prefix combinator, denoted by a full stop. For example, the component $(\alpha, r).P$ carries out activity (α, r), which has action type α and an exponentially distributed duration with parameter r, and it subsequently behaves as P. The evolving terms or states of the component, e.g. P, are termed *derivatives*.

Choice:

The choice combinator captures the possibility of competition 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 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 are components whose meaning is given by a defining equation. The notation for this is $X \stackrel{def}{=} E$. The name X is in scope in the expression on the right hand side meaning that, for example, $X \stackrel{def}{=} (\alpha, r).X$ defines a component which performs α at rate r, forever.

Cooperation:

In PEPA direct interaction, or *cooperation*, between components is the basis of compositionality. The set which is used as the subscript to the cooperation symbol, the *cooperation set* L, determines those activities on which the *cooperands* are forced to synchronise. 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 (for details see [7]). We write $P \underset{L}{\bowtie} Q$ to denote cooperation between P and Q over L. We write $P \parallel Q$ as an abbreviation for $P \underset{L}{\bowtie} Q$ when L is empty.

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 (denoted $\top$) 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.

Hiding:

The component P/L behaves as P except that the activities named in the set L are hidden, preventing other components from synchronising on them.

3.2 Places

A PEPA net is made up of PEPA *contexts*, one at each place in the net. A context consists of a number of *static* components (possibly zero) and a number of *cells* (at least one). A cell is a storage location for a token, thus PEPA nets are *bounded* nets.

We use the notation $Q[\_]$ to denote a cell which could be filled by the PEPA component Q or one with the same alphabet. If Q has derivative Q' only and no other component has the same alphabet as Q then there are three possible values for such a context: $Q[\_]$, $Q[Q]$ and $Q[Q']$. $Q[\_]$ enables no transitions. $Q[Q]$ enables the same transitions as Q. $Q[Q']$ enables the same transitions as Q'.

3.3 Markings

Just as in any Petri net, the marking reflects the global state as a tuple, with one entry capturing the local state at each place of the net. In a PEPA net the local states will be the current derivative of each place, which will include information about which cells are currently occupied and the current state of any tokens present, as well as the current state of all static components in the place.

3.4 Firings and transitions

A *firing* in a PEPA net causes the transfer of one or more tokens from one place to another. The tokens which are moved are PEPA components, which causes a change in the subsequent evaluation both in the source (where existing cooperations with other components now can no longer take place) and in the target (where previously disabled cooperations are now enabled by the arrival of an incoming component which can participate in these interactions). Firings have global effect because they involve components at more than one place in the net. Each firing can only take place if, in addition to a token for each input arc of the transition, there is a vacant cell of appropriate type corresponding to each output arc of the transition. Firings are assumed to occur with an exponential firing rate.

A *transition* in a PEPA net takes place whenever a transition of a PEPA component can occur (either individually, or in cooperation with another component). Components can only cooperate if they are resident in the same place in the net. The PEPA net formalism does not allow components at different places in the net to cooperate on a shared activity. Thus transitions in a PEPA net have local effect because they involve only components at one place in the net.

Firings and transitions are distinct. We document this by representing firing activities in boldface and transitions in italic font.

4. FROM UML 2.0 ACTIVITY DIAGRAMS TO PEPA NET MODELS

In this section, we demonstrate the translation between UML 2.0 activity diagrams and PEPA net models. We consider activity diagrams in which there is choice, looping, control and object flows, but no synchronisation.

As a first step we identify the components of the PEPA net, distinguishing tokens and static components. The context object of the activity diagram is a token of the PEPA net, as is each object token involved in an object flow. The behaviour of the context object component closely reflects the structure of the activity diagram respecting sequence and choice: each activity of the diagram becomes an activity in the PEPA definition of the component. When a choice in the activity diagram is labelled by guards, the guards are elevated to the status of activities offered in competition in the PEPA token component.

The behaviour of the context object is partitioned into a number of different contexts, according to the interactions with other components which are required, i.e. according to which activities require cooperation with a object token. These joint activities must occur within a place of the PEPA net, thus we make these activities transitions, and the immediately preceding activities firings. Thus in addition to an initial place (and possibly a final place) there is a distinct place in the PEPA net for each activity which involves an object flow.

For each object token we define a PEPA token component. As well as the activity on which it cooperates with the context token, it is given activities to bring it into the place of interaction and remove it from the place of interaction. These transitions will be firings. As above, if there is a guarded choice immediately following the activity, the guards are elevated to the status of activities in the object token PEPA definition. Furthermore, a static component which arbitrates the choice is added to the definition of the appropriate place.

Since the objective of a PEPA net is to carry out performance analysis based on an underlying Continuous Time Markov Chain (CTMC), an exponential delay must be associated with each activity, whether it is a transition or a firing. We assume that these rates are added, by a performance analyst, at the time of translation.

We observe that, in general, in order to carry out performance analysis, we would aim for a PEPA net that is more abstract than the general purpose UML activity diagram which describes the activities of a system in detail. Thus at the end of the section we illustrate the process by which the translated PEPA net is refined into one more suitable for performance modelling.

First in order to demonstrate the translation, we show it on an example, which represents of fragment of the MMPORG which constitutes our case study presented in the next section. This fragment is represented as a UML 2.0 activity diagram; this is translated into a PEPA net model at the same level of abstraction. Finally we show how the PEPA net obtained relates to the more abstract PEPA net model used for performance analysis in the following section.

4.1 Example Activity Diagram

In the MMPORG (Massive Multi-Player Online Role-playing Game) there are players (users) and non-playing characters (such as monsters, witches, etc) which interact as they play the game, evolving from room to room. When players and non-players are within the same room they may meet and fight; the outcome of a fight is determined by the room. In this small example we present a simplified model of a room.

The activity diagram on Figure 3 depicts a scenario in which a player reaches a room and interacts with a non-playing character generated by the room, the result of the fight being determined by the room but reflected in the subsequent state of both the player and the non-playing character. As we will see in the next section, this is a simplification and each room offers several such possible scenarios. The player is the subject of the diagram. In UML2.0 terms this means that the class Player is the classifier context for each activity in the diagram; see [20] p255 for discussion.

If the player wins the fight, the room increases his number of points and gives him a new skill card, whereas the non-playing character passes an object to the player before being suspended by the room. If the player is defeated, the room decreases his number of points and lets the non-playing character continue its progression within the game.

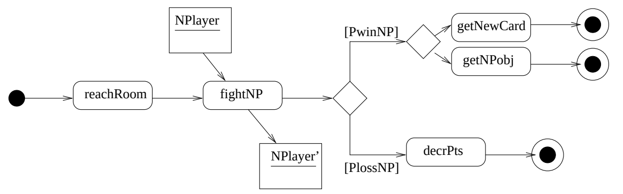


Figure 3: UML 2.0 Activity Diagram

As described earlier, the specification of object flows has been enhanced in UML2.0. It allows us to model the non-playing character generated by the room as an object NPlayer, which is used as an input to the fightNP activity. The object NPlayer' is the output of this object flow — the result of the fightNP activity on the object NPlayer. Therefore, if the player wins the fight, NPlayer will give the player an object and then be suspended by the room. In this case, NPlayer' is the suspended object. If the non-playing character wins the fight, NPlayer' is simply a non-playing character object which can be involved in other fights or moved to other rooms of the game.

4.2 The PEPA net model

Figure 4 depicts the PEPA net translation of the activity diagram shown in Figure 3. The activities of the UML diagram represent the behaviour of the player, which is represented explicitly as a token (mobile component) in the PEPA net. Each of the activities is mapped to a PEPA activity which is either a transition (local) or a firing (global). This is determined by considering the different contexts in which the player finds himself: these are *before*, *during* and *after* interaction with the non-playing character. These correspond to the places of the PEPA net: $P2$, $P3$, and $P4$. The non-playing character is represented by another token of the PEPA net and its possible contexts are represented by places $P1$ and $P3$ respectively. Since the place $P3$ is the context of interaction, based on the room, as well as cells for the player and non-player, it contains the static component *Room*.

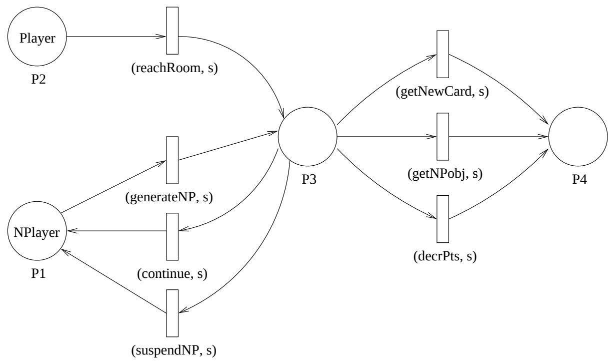


Figure 4: PEPA net model corresponding to the activity diagram in Figure 3

4.2.1 Component Player

When the player is in the room, they may be attacked by a non-playing character (*fightNP*). The result of the fight may be either a defeat of the player (*PlossNP*) or his victory (*PwinNP*). In the former case, he loses points (**decrPts**). In the latter case, the player gets cards (**getNewCard**).

$$
\begin{array}{lcl}
\mathit{Player} & \stackrel{def}{=} & (\mathbf{reachRoom}, \top).\mathit{Player}_1 \\
\mathit{Player}_1 & \stackrel{def}{=} & (\mathit{fightNP}, \alpha).\mathit{Player}_{21} \\
\mathit{Player}_{21} & \stackrel{def}{=} & (\mathit{PlossNP}, \top).\mathit{Player}_{22} \\
 & & \quad + (\mathit{PwinNP}, \top).\mathit{Player}_{23} \\
\mathit{Player}_{22} & \stackrel{def}{=} & (\mathbf{decrPts}, \beta).\mathit{Player}_1 \\
\mathit{Player}_{23} & \stackrel{def}{=} & (\mathbf{getNPobj}, \gamma_1).\mathit{Player}_1 \\
 & & \quad + (\mathbf{getNewCard}, \gamma_2).\mathit{Player}_1
\end{array}
$$

4.2.2 Component NPlayer

Once a non-playing character has been created by a room (**generateNP**), it may meet a playing character. A fight may then follow and as explained before, if the non-playing character is defeated (*PwinNP*), it has to give objects to the player. Moreover, it vanishes from the system (the room), via action type *destroyNP*. If it wins (*PlossNP*), it just continues its progression in the rooms of the current game level.

$$
\begin{array}{lcl}
\mathit{NPlayer} & \stackrel{def}{=} & (\mathbf{generateNP}, \top).\mathit{NPlayer}_1 \\
\mathit{NPlayer}_1 & \stackrel{def}{=} & (\mathit{fightNP}, \delta).\mathit{NPlayer}_2 \\
\mathit{NPlayer}_2 & \stackrel{def}{=} & (\mathit{PlossNP}, \top).\mathit{NPlayer}_3 \\
 & & \quad + (\mathit{PwinNP}, \top).\mathit{NPlayer}_4 \\
\mathit{NPlayer}_3 & \stackrel{def}{=} & (\mathbf{continue}, \sigma).\mathit{NPlayer} \\
\mathit{NPlayer}_4 & \stackrel{def}{=} & (\mathbf{suspendNP}, \top).\mathit{NPlayer}
\end{array}
$$

4.2.3 Component Room

The room makes all computations related to the fights and sends the results to the characters using action types *PlossNP* and *PwinNP*.

$$
\begin{array}{lcl}
\mathit{Room} & \stackrel{def}{=} & (\mathit{fightNP}, \top).\mathit{Room}_1 \\
\mathit{Room}_1 & \stackrel{def}{=} & (\mathit{PlossNP}, \phi_1).\mathit{Room} + (\mathit{PwinNP}, \phi_2).\mathit{Room}
\end{array}
$$

4.2.4 Markings

The places of the PEPA net are defined as follows.

$$P_1\,[\_,\ldots,\_] \stackrel{\mathit{def}}{=} \mathit{NPlayer}\,[\mathit{NPlayer}]$$

$$P_2\,[\_,\ldots,\_] \stackrel{\mathit{def}}{=} \mathit{Player}\,[\mathit{Player}]$$

$$P_3\,[\_,\ldots,\_] \stackrel{\mathit{def}}{=} (\mathit{Player}\,[\_] \underset{\mathcal{K}}{\bowtie} \mathit{NPlayer}\,[\_]\,) \underset{\mathcal{K}}{\bowtie} \mathit{Room}$$

$$P_4\,[\_,\ldots,\_] \stackrel{\mathit{def}}{=} \mathit{Player}\,[\_]$$

where $\mathcal{K} = \{\mathit{fightNP}, \mathit{PwinNP}, \mathit{PlossNP}\}$.

4.3 Level of abstraction of a PEPA net model

In a more complete model of the MMPORG the activity diagram shown in Figure 3 would be embedded within a larger diagram showing the player's progression through a number of rooms. A PEPA net model of this could look like the model shown in Figure 5. This provides a more abstract view than Figure 4.

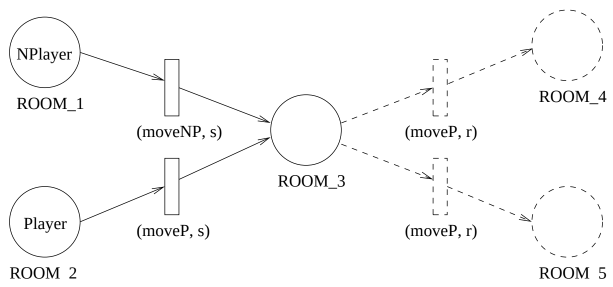


Figure 5: PEPA Net model of Figure 4 at a higher level

We can view the PEPA net's place $P1$ as contained in ROOM_1, $P2$ contained in ROOM_2, and $P3$ contained in ROOM_3 for example. The fight would occur within ROOM_3. In this representation, $P4$, where the outcomes of the fight are captured, is also subsumed within ROOM_3. The dashed transitions and places represent what could happen after the fight: the *Player* could move to other rooms.

This PEPA net model does not explicitly show the result of the fight at the net level but note that it is still implicitly defined in the *Player* and *NPlayer* definitions. The level of abstraction reflects a choice of what constitutes a separate context for the *Player* and therefore needs to represented as a distinct place at the net level. When focused on a single room the presence or absence of the *NPlayer* was considered to define a fresh context. When the game as a whole is considered the current room provides a more appropriate context, where both the more detailed contexts may be subsumed.

5. THE MMPORG

5.1 A high-level description of the game

In this section we give a more detailed account of the MMPORG and its representation as a PEPA net suitable for performance analysis. The game consists of a succession of game levels of increasing complexity. Each level is composed of a starting point and a certain number of rooms one of which is a secret room.

In the game, a player is seen as an active entity who may interact with objects, locations (rooms), playing and non-playing characters of the virtual environment. Objects such as weapons, medicine, food, or other things are one of the basic elements of the game environment that a player can collect and reuse later. The player has to explore the rooms to collect as many experience points as possible to improve character features such as strength, skill, luck, etc. Each obstacle or test successfully passed increases the number of experience points. Conversely, each failure decreases the number of points. If this number reaches zero, the player vanishes from the current room and is transferred back to the starting point of the current level. To progress to the next level, a player must find the secret room and pass the tests of this room. If they fail, they are once again transferred to the starting point of the level. The secret room can hold one player only, that is, at most one player can be inside at a time.

In other rooms a player may be in competition with one or several other players to aquire objects. The winner of a fight between two players earns experience points and takes some objects from the defeated player.

The MMPORG also features non-playing characters. Like the players, non-playing characters are active entities that may move from one room to another but they are confined to a single level and cannot access the secret room. These characters are generated by the rooms themselves.

Non-playing characters like monsters are obstacles which a player will have to pass. Fighting is a direct interaction between characters within a room. These interactions are based on a system of "cards" which can cause or neutralize some damage. The effect depends on the card features and on the features of the characters involved. Defeating a non-playing character allows the player to earn experience points, to collect objects, and to increase their current features. The player may acquire new cards and therefore increase their offensive or defensive skills.

When a player selects the next room to visit, this room clones itself and sends its image to the player. All the computations resulting from the different interactions are performed by the rooms.

5.2 The PEPA net model

Assuming that L is the number of levels in the game and N_j is the number of rooms at level j, the PEPA net model of the game consists of three types of places: $ROOM_{ji}$, $SECRET\_R_j$ and $INIT\_R_j$ where $j = 1 \ldots L$ and $i = 1 \ldots N$. Respectively, these model room i, the secret room and the starting point at level j (Figure 6). We use place OUT to stand for the environment outside the game.

Moreover we consider components *Player*, *NPlayer* and *Room* to model the behaviour of respectively the playing character, the non-playing character and the room.

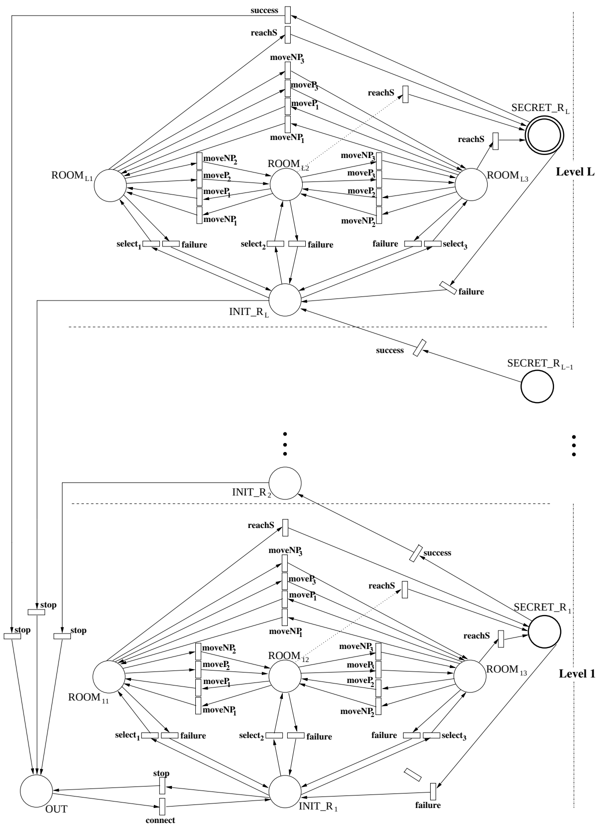


Figure 6: PEPA net model for $N_j = 3,\ j = 1 \dots L$

5.2.1 *Component Player*

Once connected (firing action **connect**), the player starts by choosing one of the rooms of the current level. This is modelled using firing action $\mathbf{select_i}$ with rate $p_i \times r_0$, i being the room number at the current level and p_i the probability to select this room number.

Once the player receives an image of the room, they may do different things: observe, walk, talk to another character (playing or non-playing). They may also try to use one of the objects they have with action type use_{obj} or to take a new one ($take_{obj}$) from the room. In this last case, the system, through the room character, may accept or refuse to let the player take the object using action type $accept_{obj}$ or $refuse_{obj}$. Here the rate of these actions is not specified by the player because the decision is made by the room.

When the player is in the room, they may be attacked by another player (*fightP*) or a non-playing character (*fightNP*). The result of the fight may be either a defeat of the player (*PlossP* or *PlossNP*) or their victory (*PwinP* or *PwinNP*). In the former case, they lose points ($less_{pts}$) and some objects ($getP_{obj}$) if the fight is against another player. If they have no more points ($zero_{pts}$), they are transferred to the starting point of the current level. This is modelled using firing action **failure**. In the latter case, the player gets objects ($getNP_{obj}$) or cards (new_{crd}) if they defeated a non-playing character.

The player may decide to move to another room i with action type $\mathbf{moveP_i}$ and probability q_i, or **reachS** if they find the secret room. The player may also decide to stop the game at any moment as long as they are in the stating point $INIT\_R_j$ of a level. This is modelled using activity $(\mathbf{stop}, s)$.

$$
\begin{array}{lcl}
Player & \stackrel{def}{=} & (\mathbf{connect}, r).Player_0 \\
Player_0 & \stackrel{def}{=} & \sum_{i=1}^{N_j} (\mathbf{select_i}, p_i \times r_0).(RImage, \top).Player_1 \\
 & + & (\mathbf{stop}, s).Player_0 \\
 & & \\
Player_1 & \stackrel{def}{=} & (observe, \alpha_1).Player_1 + (walk, \alpha_2).Player_1 \\
 & + & (talk, \alpha_3).Player_1 + (fightNP, \beta_1).Player_{21} \\
 & + & (fightP, \beta_2).Player_{31} + (test, \beta_3).Player_7 \\
 & + & (use_{obj}, \delta_1).Player_4 + (take_{obj}, \delta_2).Player_5 \\
 & + & \sum_{i=1}^{N_j - 1} (\mathbf{moveP_i}, q_i \times r_1).Player_1 \\
 & + & (\mathbf{reachS}, r_2).Player_1 \\
 & & \\
Player_{21} & \stackrel{def}{=} & (PlossNP, \top).Player_{22} + (PwinNP, \top).Player_{23} \\
Player_{22} & \stackrel{def}{=} & (less_{pts}, \gamma_1).Player_1 + (zero_{pts}, \gamma_2).Player_6 \\
Player_{23} & \stackrel{def}{=} & (getNP_{obj}, \top).Player_1 + (new_{crd}, \gamma_3).Player_1 \\
 & & \\
Player_{31} & \stackrel{def}{=} & (PlossP, \top).Player_{32} + (PwinP, \top).Player_{33} \\
Player_{32} & \stackrel{def}{=} & (less_{pts}, \gamma_1).Player_1 + (zero_{pts}, \gamma_2).Player_{34} \\
Player_{33} & \stackrel{def}{=} & (get_{pts}, \gamma_4).Player_1 + (getP_{obj}, \top).Player_1 \\
Player_{34} & \stackrel{def}{=} & (getP_{obj}, \gamma_5).Player_6 \\
 & & \\
Player_4 & \stackrel{def}{=} & (less_{pts}, \gamma_1).Player_1 + (get_{pts}, \gamma_4).Player_1 \\
 & + & (zero_{pts}, \gamma_2).Player_6 \\
Player_5 & \stackrel{def}{=} & (accept_{obj}, \top).Player_1 + (refuse_{obj}, \top).Player_1 \\
Player_6 & \stackrel{def}{=} & (\mathbf{failure}, f).Player_0 \\
Player_7 & \stackrel{def}{=} & (win, \top).Player_8 + (lose, \top).Player_6 \\
Player_8 & \stackrel{def}{=} & (get_{pts}, \gamma_4).(\mathbf{success}, c).Player_0
\end{array}
$$

5.2.2 *Component NPlayer*

Once a non-playing character has been created by a room (*generateNP*), it may walk, use its own objects and meet a playing character. A fight may then follow and as explained before, if the non-playing character is defeated (*PwinNP*), it has to give objects to the player. Moreover, it vanishes from the system (the room), via action type *destroyNP*. If it wins, it just continues its progression in the rooms of the current game level.

$$
\begin{array}{lcl}
NPlayer & \stackrel{def}{=} & (generateNP, \top).NPlayer_1 \\
NPlayer_1 & \stackrel{def}{=} & (walk, \delta_1).NPlayer_1 \\
 & + & (talk, \top).NPlayer_1 \\
 & + & (fightNP, \delta_2).NPlayer_2 \\
 & + & \sum_{i=1}^{N-1} (\mathbf{moveNP_i}, q_i \times v_1).NPlayer_1 \\
NPlayer_2 & \stackrel{def}{=} & (PlossNP, \top).NPlayer_1 \\
 & + & (PwinNP, \top).NPlayer_3 \\
NPlayer_3 & \stackrel{def}{=} & (getNP_{obj}, \delta_3).NPlayer_4 \\
 & + & (continue, \delta_4).NPlayer_4 \\
NPlayer_4 & \stackrel{def}{=} & (destroyNP, \top).NPlayer
\end{array}
$$

5.2.3 Component Room

The room creates and makes vanish the non-playing characters using respectively the activities *generateNP* and *destroyNP*. When it is chosen by a player, the room clones itself and sends an image to them (*RImage*). The room also accepts ($accept_{obj}$) or rejects ($refuse_{obj}$) any attempt by a player to take an object from the location. Moreover it makes all computations related to the fights and sends the results to the characters using action types *PlossP* or *PwinP* and also *PlossNP* and *PwinNP*.

$$\begin{array}{lcl}
Room & \stackrel{def}{=} & (generateNP, \sigma_1).Room \\
 & + & (RImage, \sigma).Room \\
 & + & (fightP, \top).Room_2 + (fightNP, \top).Room_3 \\
 & + & (take_{obj}, \top).Room_1 + (use_{obj}, \top).Room \\
Room_1 & \stackrel{def}{=} & (accept_{obj}, \rho_1).Room + (refuse_{obj}, \rho_2).Room \\
Room_2 & \stackrel{def}{=} & (PlossP, \phi_1).(PwinP, \phi_2).Room \\
Room_3 & \stackrel{def}{=} & (PlossNP, \phi_3).Room + (PwinNP, \phi_4).Room_4 \\
Room_4 & \stackrel{def}{=} & (destroyNP, \sigma_2).Room
\end{array}$$

5.2.4 Component SRoom

This component models the secret room. It is similar to the other rooms except that at most one player can be inside and non-playing characters are not allowed to get in. Once inside, the player has to pass a different test to get to the higher level.

$$\begin{array}{lcl}
SRoom & \stackrel{def}{=} & (RImage, \sigma).SRoom \\
 & + & (take_{obj}, \top).SRoom_1 + (use_{obj}, \top).SRoom \\
 & + & (test, \top).SRoom_2 \\
SRoom_1 & \stackrel{def}{=} & (accept_{obj}, \rho_1).SRoom \\
 & + & (refuse_{obj}, \rho_2).SRoom \\
SRoom_2 & \stackrel{def}{=} & (lose, \phi_3).SRoom + (win, \phi_4).SRoom
\end{array}$$

5.2.5 The Places

The places of the PEPA net are defined as follows. A typical room of the game will have storage areas for both players and non-players and will have some internal logic, encoded in the static component in the room. The following room is $ROOM_{ji}$, where $i = 1 \ldots N_j$ is the room number and $j = 1 \ldots L$ is the game level number.

$$\begin{array}{l}
ROOM_{ji} \stackrel{def}{=} \\
\quad \left(Room \underset{\mathcal{K}_1}{\bowtie} \left(Player\ [_] \underset{\mathcal{K}_2}{\bowtie} \ldots \underset{\mathcal{K}_2}{\bowtie} Player\ [_] \right) \right) \underset{\mathcal{K}_3}{\bowtie} \\
\quad\quad \left(NPlayer\ [_] \parallel \ldots \parallel NPlayer\ [_] \right)
\end{array}$$

This place uses synchronization sets $\mathcal{K}_1, \mathcal{K}_2$ and $\mathcal{K}_3$ to capture interactions with the room, between the players and with non-playing characters respectively. The synchronizing sets used in the definition above are defined as follows:

$$\begin{array}{l}
\mathcal{K}_1 = \{take_{obj}, use_{obj}, accept_{obj}, refuse_{obj}, RImage, \\
\qquad fightP, PlossP, PwinP, fightNP, PlossNP, PwinNP\} \\
\mathcal{K}_2 = \{fightP, getP_{obj}\} \\
\mathcal{K}_3 = \{generateNP, fightNP, PlossNP, PwinNP, destroyNP, \\
\qquad getNP_{obj}, talk\}
\end{array}$$

The secret room is different from the other rooms in the game in that only a single player is allowed in the secret room at a time. Non-playing characters cannot enter the secret room so no storage locations are provided for them.

$$SECRET\_R_j \stackrel{def}{=} SRoom \underset{\mathcal{K}_4}{\bowtie} Player\ [\_]$$

The synchronisation set used in this definition is simpler because it does not need to cater for non-playing characters.

$$\begin{array}{l}
\mathcal{K}_4 = \{take_{obj}, use_{obj}, accept_{obj}, refuse_{obj}, RImage, test, \\
\qquad lose, win\}
\end{array}$$

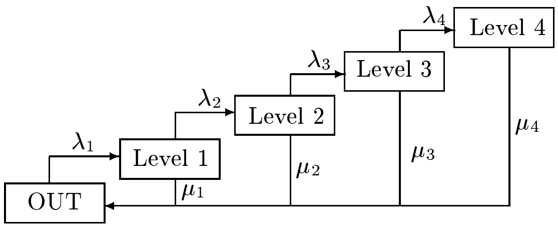
Two additional places are used to store player tokens on entry into the game ($INIT\_R_j$) and when outside the game (OUT).

$$\begin{array}{lcl}
INIT_R_j & \stackrel{def}{=} & Player\ [_] \parallel \ldots \parallel Player\ [_] \\
OUT & \stackrel{def}{=} & Player\ [Player] \parallel \ldots \parallel Player\ [Player]
\end{array}$$

6. MODEL ANALYSIS

We have analysed the MMPORG game using the PEPA Workbench for PEPA nets, the PEPA compiler and the PRISM probabilistic model checker [9]. This analysis method represents the global state space of the system as a multi-terminal binary decision diagram (MTBDD), a compact storage scheme for labelled transition systems such as those generated by interleaving-semantics concurrent modelling languages. This method of solution is feasible, but is computationally expensive since the MTBDD representation is engineered for compactness, not efficiency of access. (A traditional sparse matrix representation of the state space usually allows for more efficient numerical calculation but sometimes such a representation does not fit in the memory of the available machine.)

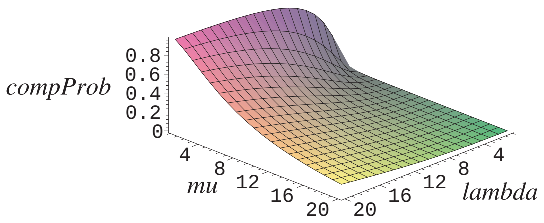
Motivated by the need to improve on this analysis approach for this example, we consider the following abstraction of our PEPA net model where each level j has one input and two output parameters. The input parameter denoted by λ_j represents the arrival rate of the players to the level. The first output parameter denoted by λ_{j+1} is nothing other than the input to the next level $j+1$. This represents the rate of successful players of level j. The second output parameter, noted μ_j, represents the rate of the players leaving the game. The diagram below depicts the first four levels of the game.



One of the stated aims of the PEPA nets modelling language is to allow analysis techniques and solution methods developed for Petri nets to be used either directly or in adaptation for PEPA nets. The structure of the present model allows us to do exactly this by deploying *flow-equivalent replacement* to bring about a dramatic reduction in the state-space of the model to be solved [8]. The method works in the following fashion for this example. The purpose of the application logic encoded in the PEPA nets model of the MMPORG is to specify in detail the necessary steps to take in order to succeed at each level of the game (and thereby progress to the next level). Once this model has been solved to find the rate at which players progress from one level to the next then the application logic has served its purpose and the sub-model can be replaced by a suitably exponentially-distributed delay, eliding all of the (now) unnecessary detail. The technique has an analogue in the programming language semantics domain of moving between "small step" (interleaving) and "big step" (evaluation) semantics of programming languages.

By using this method we are able to investigate how the probability of any of the players completing the game (*compProb*) varies as the rates of progression (λ) and rejection (μ) are varied. All of the rates here have been taken to be equal ($\lambda = \lambda_1 = \lambda_2 = \lambda_3 = \lambda_4$ and $\mu = \mu_1 = \mu_2 = \mu_3 = \mu_4$).

The graph below illustrates the expected outcome that the probability of completing the game is highest when the rate of progression from one level to the next is highest (high values of λ) and lowest when the rate at which players leave the game is highest (high values of μ), and it quantifies this information.



This technique is very well suited to this application because it allows us to evaluate one of the key performance indices of a game application: *difficulty of completion*. If it is possible to progress too quickly from commencing playing to completing the final level of the game then the application may be considered unchallenging. Conversely, if it is very arduous to make progress in the game then the developers risk losing their target audience and finding their game consigned to being suitable only for the most committed game-playing enthusiasts.

7. RELATED WORK

The field of performance analysis of UML models does not have a long history but a large number of research groups have worked energetically on the area to the point where there is now a substantial body of work on the subject. As a result, it is possible to identify some major themes of the work in this area.

One body of work in the area considers the UML to be a framework onto which it is possible to build extensions and add language features to capture properties of interest (such as mobility, reliability and performance properties). The result is typically an *extended UML* which is used as any other modelling language would be. The disadvantages of this approach include the issues that the extensions will be unfamiliar to present-day UML modellers; it is no longer possible to use UML tools (unless they are also extended to support the language extensions); and it is not usually made clear which properties of the UML *metamodel* are preserved and which are destroyed by the extension.

An alternative body of work on the performance analysis of UML models takes a different approach. The UML definition and metamodel are taken seriously; the language is used unchanged except for the addition of a *profile* (the standard UML specialisation method); and UML tools are used as before in the modelling process.

We see our work as being of the second kind and in this comparison we relate it only to work with similar themes. The work reported here builds on our own previous work in several areas. We have previously focused on extracting PEPA models from UML state diagrams and collaboration diagrams [2]. This work extends that previous work by applying an enhanced approach to different UML diagram types which support more complex and more flexible methods of documenting object interaction, movement and synchronisation. Our solution technology has also moved on, because we now map to a more expressive performance modelling language.

Work which is similar in spirit to our own approach is that of Petriu, Shen and Woodside [14, 15] where a layered queueing network model is automatically extracted from an input UML model with performance annotations in the format specified by a special-purpose UML profile [16]. We use different kinds of UML diagrams from those used by these authors (the paper [15] uses Use Case Maps). Additionally, the performance evaluation technology which we deploy (process algebras and BDD-based solution) is quite different from layered queueing networks.

Another performance engineering method which is similar to ours is that of López-Grao, Merseguer and Campos [11] where UML diagrams are mapped into GSPNs which can be solved by GreatSPN. We use UML2.0 activity diagrams which are more complex than those used by these authors and, again, a different performance evaluation technology. This work, together with the work of Bernardi, Donatelli and Merseguer [1] is notable for its careful consideration of the UML metamodel in the performance analysis process. In paper [1] the authors use sequence diagrams and statecharts as the initial UML diagram formalism.

Our work differs from the work of Lindemann *et al.* [10] in that we propose a compositional mapping from UML activity diagrams into a high-level modelling formalism whereas their approach generates the underlying stochastic process (a generalised semi-Markov process) directly from state diagrams and activity diagrams. Set against this, their work provides enhanced expressiveness in the initial time-enhanced

UML model in that they allow both exponentially distributed and deterministic delays.

8. CONCLUSIONS

In this paper we have demonstrated a mapping between the newly revised UML diagram type, UML2.0 activity diagrams, and PEPA nets, a recently defined performance modelling formalism. This mapping facilitates performance analysis at an early stage of design, using a stochastic representation consistent with the designer's intentions.

We demonstrated this via the modelling and analysis of a real-world case study, the MMPORG game provided by one of the industrial partners on our project. As we had intended, it is now possible to capture meaningful decisions about the performance of the system which is under consideration, and to methodically carry these decisions through the performance analysis process to see their effects on the system in operation. This analysis can be carried out at an early stage in the software development process. At this time it has the potential to bring significant benefit without incurring significant cost. The performance analysis of models is a hugely profitable activity. A brief period of analysis and investigation at this stage can save many hours of measurement, simulation and re-engineering effort at a later time [12].

One of the lessons which we have learned from the present work is that the encoding of a UML 2.0 activity diagram as a PEPA net is not facile and requires careful consideration. In part this is due to the inherent complexity of UML 2.0 activity diagrams which arises because they attempt to provide high-level modelling concepts such as *control flows* and *object flows* with well-specified properties. PEPA nets provide similar modelling concepts in the stochastically timed world of Markovian modelling. Our contribution here has been to show how UML 2.0 activity diagrams can be refined into models in this formalism, thereby facilitating efficient performance analysis.

9. ACKNOWLEDGMENTS

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

10. REFERENCES

[1] S. Bernardi, S. Donatelli, and J. Merseguer. From UML sequence diagrams and statecharts to analysable Petri net models. In Tucci [19], pages 35–45.

[2] 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.

[3] T. Field, P. Harrison, J. Bradley, and U. Harder, 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, Apr. 2002. Springer-Verlag.

[4] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. PEPA nets: A structured performance modelling formalism. To appear in *Performance Evaluation*, 2003.

[5] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. Performance modelling with PEPA nets and PRISM. In *Proceedings of the Second PASTA workshop*, pages 23–39, Edinburgh, Scotland, June 2003.

[6] S. Gilmore, J. Hillston, and M. Ribaudo. PEPA nets: A structured performance modelling formalism. In Field et al. [3], pages 111–130.

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

[8] H. Jungnitz and A. Desrochers. Flow equivalent nets for the performance analysis of flexible manufacturing systems. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, pages 122–127, Sacramento, CA, USA, 1991.

[9] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In J.-P. Katoen and P. Stevens, editors, *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 52–66. Springer, 2002.

[10] C. Lindemann, A. Thümmler, A. Klemm, M. Lohmann, and O. P. Waldhorst. Performance analysis of time-enhanced UML diagrams based on stochastic processes. In Tucci [19], pages 25–34.

[11] J. López-Grao, J. Merseguer, and J. Campos. From UML activity diagrams to stochastic Petri nets: Application to software performance analysis. In *Proceedings of the Seventeenth International Symposium on Computer and Information Sciences*, pages 405–409, Orlando, Florida, Oct. 2002. CRC Press.

[12] I. Mitrani. *Probabilistic modelling*. Cambridge University Press, 1998.

[13] OMG. *Unified Modeling Language Specification version 1.5*, March 2003. Available from http://www.omg.org/uml/ as formal/03-03-01.

[14] D. Petriu and H. Shen. Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specifications. In Field et al. [3], pages 159–177.

[15] D. Petriu and M. Woodside. Analysing software requirements specifications for performance. In Tucci [19], pages 1–9.

[16] 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.

[17] C. U. Smith and L. G. Williams. *Performance solutions: a practical guide to creating responsive, scalable software*. Addison-Wesley, 2002.

[18] P. Stevens. UML and concurrency. In *Proceedings of 10th International Workshop on Abstract State Machines*, number 2589 in LNCS, pages 151–165. Springer, March 2003.

[19] S. Tucci, editor. *Proceedings of the Third International Workshop on Software and Performance (WOSP 2002)*. ACM Press, Rome, Italy, July 2002.

[20] U2P. *Unified Modeling Language: Superstructure version 2.0*, April 2003. Available from http://www.omg.org/uml/ as ad/03-04-01.