# 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.

## 1. INTRODUCTION

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 [4].

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

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 [1].

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. Due to lack of space we do not describe PEPA nets in detail here but refer the reader to [2].

## 2. UML 2.0 ACTIVITY DIAGRAMS

One of the major changes introduced in UML2.0 is a radical overhaul of activity diagrams. 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. 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.

UML 2.0 also revises the concept of *object flows* in models. 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 contribute. 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 1 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 moveP 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.

# 3. 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 subcontexts, according to the interactions with other components which are required, i.e. according to which activities require cooperation with an 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 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, representing the object becoming available for interaction, and leaving the subcontext of interaction.

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 discuss 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.

## 3.1 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. If the player wins the fight, he may either obtains a new skill card or some objects belonging to the non-playing character. If the player is defeated, his number of points decreases.

The activity diagram on Figure 1 depicts a scenario in which a player moves to a room and interacts with a non-playing character. The result of the fight is 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 [5] for discussion.

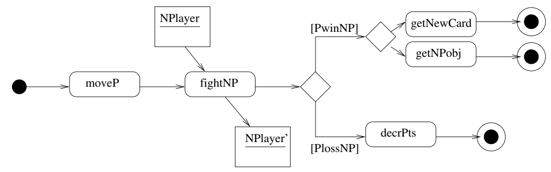



**Figure 1: UML 2.0 Activity Diagram**

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. If the player wins the fight, items belonging to NPlayer are given to the player. In this case, NPlayer' is a modified 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.

## 3.2 The PEPA net model

Figure 2 depicts the PEPA net translation of the activity diagram shown in Figure 1. 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 $P1$, $P3$ and $P5$ respectively.

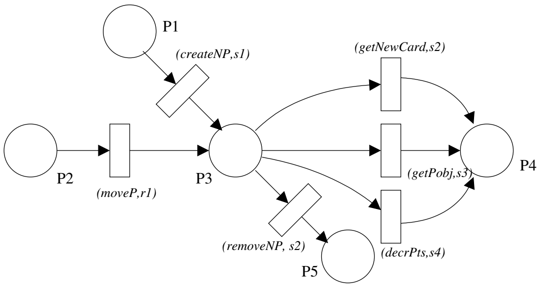



**Figure 2: PEPA net corresponding to Figure 1**

### 3.2.1 *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**).

$$\begin{array}{lcl}
\mathit{Player} & \stackrel{def}{=} & (\mathbf{moveP}, r_1).\mathit{Player}_1 \\
\mathit{Player}_1 & \stackrel{def}{=} & (\mathit{fightNP}, \alpha).\mathit{Player}_2 \\
\mathit{Player}_2 & \stackrel{def}{=} & (\mathit{PwinNP}, q_2 \times \phi_2).\mathit{Player}_3 + (\mathit{PlossNP}, q_1 \times \phi_1).\mathit{Player}_4 \\
\mathit{Player}_3 & \stackrel{def}{=} & (\mathbf{getNewCard}, s_1).\mathit{Player}_1 + (\mathbf{getNPobj}, s_2).\mathit{Player}_1 \\
\mathit{Player}_4 & \stackrel{def}{=} & (\mathbf{decrPts}, s_3).\mathit{Player}_1
\end{array}$$

### 3.2.2 *Component NPlayer*

Once a non-playing character has been created by a room, 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.

$$\begin{array}{lcl}
\mathit{NPlayer} & \stackrel{def}{=} & (\mathbf{createNP}, t_1).\mathit{NPlayer}_1 \\
\mathit{NPlayer}_1 & \stackrel{def}{=} & (\mathit{fightNP}, \delta).\mathit{NPlayer}_2 \\
\mathit{NPlayer}_2 & \stackrel{def}{=} & (\mathit{PlossNP}, \top).\mathit{NPlayer}' + (\mathit{PwinNP}, \top).\mathit{NPlayer}' \\
\mathit{NPlayer}' & \stackrel{def}{=} & (\mathbf{removeNP}, t_1).\mathit{NPlayer}
\end{array}$$

### 3.2.3 *Markings*

The places of the PEPA net are defined as follows.

$$\Big( \mathit{NPlayer}[\mathit{NPlayer}],\ \mathit{Player}[\mathit{Player}],\ \mathit{Player}[\_] \underset{\mathcal{K}}{\bowtie} \mathit{NPlayer}[\_],\ \mathit{Player}[\_],\ \mathit{NPlayer}[\_] \Big)$$

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

## 3.3 Level of abstraction of a PEPA net model

In a more complete model of the MMPORG the activity diagram shown in Figure 1 would be embedded within a larger diagram showing the player's progression through a number of rooms. When a sequence of interactions are encountered, the subcontexts representing *after* one interaction and *before* the next may be merged. A PEPA net model of this could look like the model shown in Figure 3. This provides a more abstract view than Figure 2. We make a direct correspondence between $P1$, $P2$, and $P5$, and the new

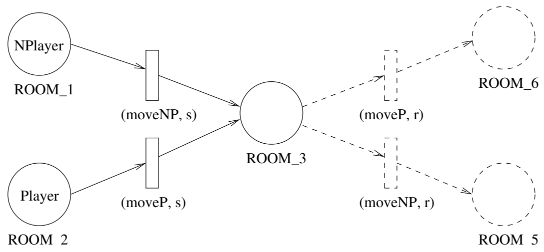



**Figure 3: PEPA Net model of Figure 2 at a higher level**

places *ROOM_1*, *ROOM_2*, and *ROOM_5* respectively. As the resultant activities of a fight (*getNewCard*, *getPobj*,*decrPts*) do not need to be firing transition activities, we use *ROOM_3* as a place where the fight occurs and its result consumed. So places $P_3$ and $P_4$ of Figure 2 become a unique place *ROOM_3*.

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* components. The level of abstraction reflects a choice of what constitutes a separate context for the *Player* and therefore needs to be 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.

# 4. THE MASSIVE MULTI-PLAYER ONLINE ROLE-PLAYING GAME

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: $\mathit{ROOM}_{ji}$, $\mathit{SECRET\_R}_j$ and $\mathit{INIT\_R}_j$ 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 4). We use place *OUT* to represent the environment outside the game. Moreover we consider components *Player*, *NPlayer* and *Room* to model the behaviour of the playing character, the non-playing character and the room respectively.

### *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 $\mathit{use}_{obj}$ or to take a new one ($\mathit{take}_{obj}$) 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 $\mathit{accept}_{obj}$ or $\mathit{refuse}_{obj}$. 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*). If defeated, they lose points ($\mathit{less}_{pts}$) and some objects ($\mathit{getP}_{obj}$) if the fight is against another player. If they have no more points ($\mathit{zero}_{pts}$), they are transferred to the starting point of the current level. This is modelled by firing

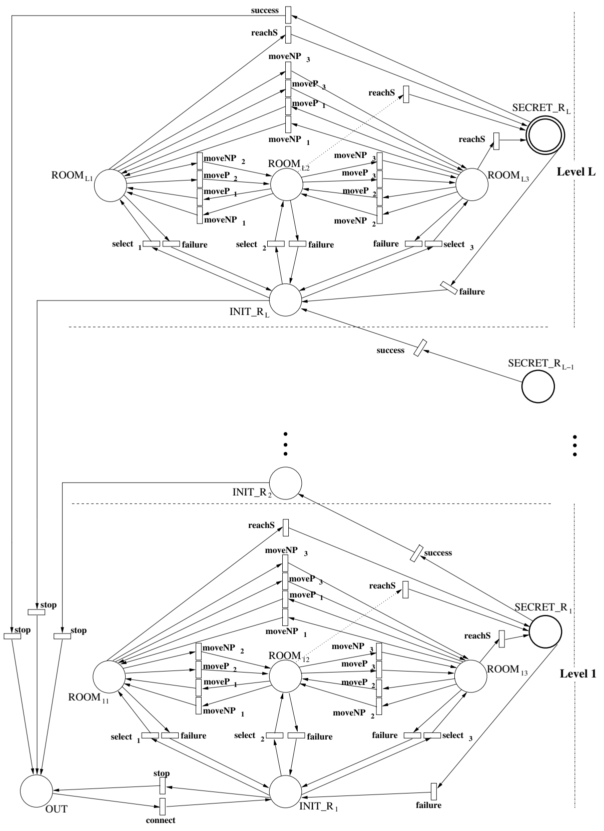



**Figure 4: PEPA net model for $N_j = 3,\ j = 1 \dots L$**

action **failure**. If victorious, 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 starting point $INIT\_R_j$ of a level. This is modelled using activity (**stop**, $s$).

$$\begin{aligned}
Player &\stackrel{def}{=} (\mathbf{connect}, r).Player_0 \\
Player_0 &\stackrel{def}{=} \textstyle\sum_{i=1}^{N_j} (\mathbf{select_i}, p_i \times r_0).(RImage, \top).Player_1 \\
&\quad + (\mathbf{stop}, s).Player_0
\end{aligned}$$

$$\begin{aligned}
Player_1 &\stackrel{def}{=} (observe, \alpha_1).Player_1 + (walk, \alpha_2).Player_1 \\
&\quad + (talk, \alpha_3).Player_1 + (fightNP, \beta_1).Player_{21} \\
&\quad + (fightP, \beta_2).Player_{31} + (test, \beta_3).Player_7 \\
&\quad + (use_{obj}, \delta_1).Player_4 + (take_{obj}, \delta_2).Player_5 \\
&\quad + \textstyle\sum_{i=1}^{N_j-1} (\mathbf{moveP}_i, q_i \times r_1).Player_1 \\
&\quad + (\mathbf{reachS}, r_2).Player_1
\end{aligned}$$

$$\begin{aligned}
Player_{21} &\stackrel{def}{=} (PlossNP, \top).Player_{22} + (PwinNP, \top).Player_{23} \\
Player_{22} &\stackrel{def}{=} (less_{pts}, \gamma_1).Player_1 + (zero_{pts}, \gamma_2).Player_6 \\
Player_{23} &\stackrel{def}{=} (getNP_{obj}, \top).Player_1 + (new_{crd}, \gamma_3).Player_1
\end{aligned}$$

$$\begin{aligned}
Player_{31} &\stackrel{def}{=} (PlossP, \top).Player_{32} + (PwinP, \top).Player_{33} \\
Player_{32} &\stackrel{def}{=} (less_{pts}, \gamma_1).(getP_{obj}, \gamma_3).Player_1 + (zero_{pts}, \gamma_2).Player_{34} \\
Player_{33} &\stackrel{def}{=} (get_{pts}, \gamma_4).(getP_{obj}, \top).Player_1 + (getP_{obj}, \top).Player_1 \\
Player_{34} &\stackrel{def}{=} (getP_{obj}, \gamma_5).Player_6
\end{aligned}$$

$$\begin{aligned}
Player_4 &\stackrel{def}{=} (less_{pts}, \gamma_1).Player_1 + (get_{pts}, \gamma_4).Player_1 \\
&\quad + (zero_{pts}, \gamma_2).Player_6 \\
Player_5 &\stackrel{def}{=} (accept_{obj}, \top).Player_1 + (refuse_{obj}, \top).Player_1 \\
Player_6 &\stackrel{def}{=} (\mathbf{failure}, f).Player_0 \\
Player_7 &\stackrel{def}{=} (win, \top).Player_8 + (lose, \top).Player_6 \\
Player_8 &\stackrel{def}{=} (get_{pts}, \gamma_4).(\mathbf{success}, c).Player_0
\end{aligned}$$

## *Component NPlayer*

Once a room has created a non-playing character (*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.

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

## *Component Room*

The room creates and destroys the non-playing characters using the activities *generateNP* and *destroyNP* respectively. 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*.

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

## *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.

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

### 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.

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

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:

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

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.

$$\mathcal{K}_4 = \{take_{obj}, use_{obj}, accept_{obj}, refuse_{obj}, RImage, test, lose, win\}$$

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

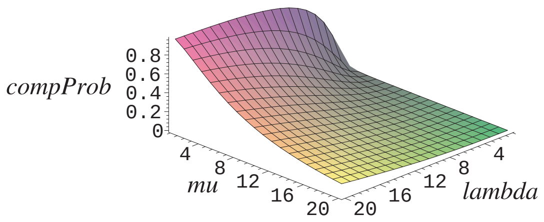
$$INIT\_R_j \stackrel{def}{=} Player[\_] \parallel \ldots \parallel Player[\_]$$
$$OUT \stackrel{def}{=} Player[Player] \parallel \ldots \parallel Player[Player]$$

# 5. MODEL ANALYSIS

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 $\lambda_j$ represents the arrival rate of the players to the level. The first output parameter denoted by $\lambda_{j+1}$ 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 $\mu_j$, represents the rate of the players leaving the game.

By using *flow-equivalent replacement* [3] we were able to use the PEPA Workbench for PEPA nets to investigate how the probability of any of the players completing the game (*compProb*) varies as the rates of progression ($\lambda$) and rejection ($\mu$) 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 $\lambda$) and lowest when the rate at which players leave the game is highest (high values of $\mu$), 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.

# 6. 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.

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.

# 7. 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.


# 8. REFERENCES


[1] 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.

[2] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. Software performance modelling using PEPA nets. In *Proc. of Int. Workshop on Software Performance Modelling (WOSP 2004)*, 2004. (this volume).

[3] 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.

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

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