From Interaction Overview Diagrams to PEPA Nets *

Leila Kloul† and Juliana Küster-Filipe‡

†PRiSM, Université de Versailles, 45, Avenue des Etats-Unis 78035 Versailles Cedex, France
email: kle@prism.uvsq.fr
‡School of Computer Science, The University of Birmingham, Birmingham B15 2TT, UK
email: J.Kuester-Filipe@cs.bham.ac.uk

1 Introduction

In this paper we show how to model mobility and performance information at the design level using UML 2.0 notation, namely interaction overview diagrams (IODs) [3]. IODs are a high-level structuring mechanism for sequence diagrams and a special kind of activity diagram (AD). Whilst ADs have been used to model mobility, they are not adequate to capture at the same time the structure of the system (locations), how objects move between locations, and how objects behave/interact within locations. By contrast, this is possible using IODs.

An IOD allows us to describe a mobile system at two levels. The higher level describes the locations of the system and how objects move between locations. The lower level describes how objects behave and interact locally and is given by the individual nodes of the IOD, namely sequence diagrams. Both levels are enriched with performance related information (i.e., activities). Furthermore, our approach of using IODs for modelling interactions and mobility enhanced with performance information offers a novel and natural UML counterpart for PEPA nets [1], a performance modelling language which consists of a restriction of Petri nets where tokens are terms in a stochastic process algebra PEPA [2]. We describe the formal translation of IODs into PEPA nets. Essentially, the structure given by the IOD corresponds to the high level net structure of the PEPA net, and the behaviour described in the IOD nodes (sequence diagrams) can be translated onto PEPA terms. Our translation allows a designer using UML 2.0 to model and analyse his/her models formally using the underlying tools available for PEPA nets.

The paper is structured as follows. In Section 2, we introduce the dynamic models in UML 2.0. We then show in Section 3 how we translate an IOD model into a PEPA net model. In Section 4, an example of the translation is provided. We finally conclude our work in Section 5 with some remarks and future work.

2 Dynamic Models in UML 2.0

Most dynamic models in UML 2.0 have been considerably revised and differ from previous versions in UML 1.x. To model interactions, UML 2.0 offers four kinds of diagrams: communication diagrams, sequence diagrams, timing diagrams and interaction overview diagrams. Here we are only interested in sequence diagrams and interaction overview diagrams.

*This work has been partially supported by the ACI Security 03-04 Sure Path
Sequence diagrams have been considerably revised and extended in UML 2.0 and are now more expressive and fundamentally better structured. Interaction overview diagrams (IODs) are new in UML 2.0 allowing a designer to describe a high-level view of the possible interactions in a system.

IODs constitute a high-level structuring mechanism that is used to compose scenarios through sequence, iteration, concurrency or choice. IODs are a special and restricted kind of activity diagrams (ADs) in UML where nodes are interactions or interaction uses, and edges indicate the flow or order in which these interactions occur [3]. Semantically, however, IODs and ADs are given different interpretations. IODs follow, similarly to sequence diagrams, a trace semantics whereas ADs in UML 2.0 are understood as Petri nets.

2.1 Sequence Diagrams

Sequence diagrams are the more commonly used diagram for capturing inter-object behaviour. In UML 2.0, a sequence diagram is enclosed in a frame and the five-sided box at the upper left-hand corner names the sequence diagram. Further, interactions can be structured using so-called interaction fragments. Each interaction fragment has at least one operator held in the five-sided box at the upper left corner of the fragment. There are several possible operators described below. Figure 1 shows an example of a sequence diagram using UML 2.0 constructs.

![Sequence Diagram Example](image)

Figure 1: A sequence diagram.

The semantics of an interaction operator is described informally in the UML 2.0 superstructure specification [3]. Below we give the meaning of some operators used in this paper:

**sd** names a sequence diagram.

**ref** references an interaction fragment which appears in a different diagram. This fragment is called an interaction use.

**seq** indicates weak sequencing of the operands in the fragment (default).

**strict** specifies that the interaction fragment represents a strict sequencing ordering between the behaviour of the operands. Messages in the fragment are fully ordered.
**alt** designates that the fragment represents a choice of behaviour. At most one of the operands will execute. The operand that executes must have a guard expression that evaluates to true at this point in the interaction. If several guards are true, one of them is selected nondeterministically for execution.

**par** designates that the fragment represents a parallel merge between the behaviours of the operands. The event occurrences of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved.

**loop** specifies an interaction fragment that shall be repeated some number of times. This may be indicated using a guard condition. The loop fragment is executed as long as the guard condition is true.

### 2.2 Interaction Overview Diagrams

An IOD is a variation of an activity diagram used to describe a high-level view of the possible interactions in a system. The notation used incorporates notation from sequence diagrams, essentially references (interaction uses) and sequence diagrams (inline interactions), with forks, joins, decision and merge nodes from ADs. However, branching and joining of branches in an IOD must be properly nested which is more restrictive than in an AD.

IODs are special kinds of ADs where the activity nodes are interactions or interaction uses and the activity edges denote control flow only. According to the UML specification [3] object flow cannot be represented in an IOD.

Object flow in an AD is shown in Figure 2. The figure shows two alternative notations to denote the flow of an object of type *Order* from one activity *FillOrder* to another activity *ShipOrder*. In the first case the object is depicted in the middle of the edge, whereas in the second case the activities have an output and input pin of type *Order*. After the activity *FillOrder* has been completed a token of type *Order* is placed in the output pin of *FillOrder*. As soon as the edge fires the token moves to the input pin of *ShipOrder*.

![Figure 2: Object flow examples.](image)

An activity can have more than one object as in/output. In this case, there are several edges between the underlying activities, one for each type of token, and the edges can be fired independently. However, whichever token reaches a target pin first will have to wait for the others before the final target activity can be initiated. Unless otherwise indicated, all pins are required as input values before an activity can be executed.

By default the number of tokens that are carried along an edge is one, but an input or output pin can collect several tokens of the same type. For instance in Figure 2, it may be the case that several orders have been filled (the activity has been executed several times) and the corresponding tokens are placed in the output pin of *FillOrder* waiting for the edge to fire and the tokens (one at a time) to move to the input pin of *ShipOrder*. It is also possible that a pin can only accept a certain number of tokens. We write `{upperBound = 50}` next to a pin to indicate that the maximum number of tokens that can be stored in that pin is 50. If the current
number of tokens collected at the pin is 50 and the pin is an input pin, then no edge leading to that pin is allowed to fire.

Further, it is possible to have multiple edges leaving an output pin as shown in Figure 3. Notice that we cannot duplicate tokens on edges (tokens can only be duplicated in forks) which means effectively that we have a case where the edges have to compete for a token. In this example, we are modelling that after a part has been made it will (nondeterministically) be either painted at Station1 or at Station2. To avoid nondeterministic choice as in this example, it is common to use mutually exclusive guards on edges. As in ADs, edges in IODs can have guards (boolean expressions) to determine whether the edge may or not enable the target node. Guards are written in square brackets on the edge.

Finally, in an AD an activity can only start if all its input pins contain tokens as required, and after execution tokens are placed in all output pins. Sometimes, however, behaviour has alternative inputs or outputs. In other words, we may want to allow the activity to execute with just a few inputs and produce only a subset of the possible outputs. To denote this we use a double box around pins as shown in Figure 4. Here, the activity node requires as input either one token of type a, or one token of type b and a token of type c. In one case it produces as output a token of type d or a token of type e. At this level we are not able to determine how input and output tokens are related. According to [4] it is also not clear what happens if both sets of inputs are available. Our interpretation here is that alternative input/output pins denote concurrent object flow and the activity node contains a parallel subactivity for each of these inputs. This will become clearer later on with an example of an IOD.

Even though, IODs only describe control flow and cannot show object flow and pins, the notion of object flow is implicitly present. A node in an IOD is a sequence diagram containing objects that can progress to a further interaction according to the edges at the IOD level. Moreover, from an IOD we can derive the expected traces of behaviour for each of the instances involved.

Take the example in Figure 5 showing an IOD with two inline interactions. The first interaction (sd 1) shows object a sending a message m1() to object b and independently object c sending a message m2() to object d. The second interaction (sd 2) shows just object a sending a message m3() to object b. At the higher level, there are two ways of understanding the edge from sd 1 to sd 2 which correspond to the two possible interpretations of sequential composition of interactions in an IOD. The first interpretation could be that interaction sd 1 has to
complete before the behaviour described in interaction \textit{sd 2} can start. This is the typical interpretation of transitions in an AD and corresponds to the notion of \textit{strong sequential composition}. However, it is not entirely justified in the case of IODs as will be made clear shortly.

A second and weaker interpretation could be that since only objects \textit{a} and \textit{b} are involved in the second interaction, these two objects can move from the first interaction to the second after completing their behaviour in the first interaction. In other words, it should be possible for \textit{a} and \textit{b} to proceed to the second interaction after \textit{a} and \textit{b} have synchronised on message \textit{m1()} independently of whether \textit{c} and \textit{d} have synchronised on message \textit{m2()} or not.

Depending on the interpretation given to sequential composition at an IOD level, flattening the IOD of Figure 5 results in the sequence diagrams shown in Figure 6 with strong sequential composition on the left and weak sequential composition on the right (equivalently, we could have omitted the \texttt{seq} operator as this is the default behaviour). For the case on the left, the operator \texttt{strict} imposes a strict sequencing between the interactions described in the operands. For the example it means that \texttt{m2()} must occur before \texttt{m3()}. For the case on the right, there is a weak sequencing of the behaviour of the operands. In this particular example, this implies that \texttt{m3()} may occur before or after \texttt{m2()} as the two pairs of objects are independent and the messages unordered.

In order to allow both interpretations, we will assume that if the first interpretation is intended we model the IOD with normal edges as in Figure 5, whilst if the second interpretation is intended, we borrow the notation of object flow and pins from activity diagrams as shown in Figure 6.
have an output pin with the name and type of the object, and an input pin with the same name and type in the following interaction. As soon as an object completes its behaviour as described in the first interaction, a token is placed in the corresponding output pin and the edge can fire provided the target pin has enough space. Whether or not the following interaction can execute depends on how many input tokens are required (recall also the case in Figure 4). In Figure 7, interaction sd 2 can only start executing once both tokens (one of type a and one of type b) are available in the respective input pins, but regardless of whether message m2 has been sent or not.

With both interpretations of sequential composition at an IOD level we obtain a powerful language to model and structure interactions. In particular, we show in the next section how IODs can be used to describe interactions for mobile applications.

2.3 Using IODs for Mobility

Consider a simple example of a mobile system consisting of two secret agents 006 and 007 that can move between locations in a system to collect and share secret information. The agent 007 is responsible for finding the information from a remote source and saving it, whilst agent 006 processes and selects that information. This example is modelled with the IOD of Figure 8. In the beginning a token 006 and 007 is placed in the respective input pins of interactions P3 and P1. The independent execution of interactions P3 and P1 each produce a token which is placed in the respective output pin. The interaction P2 has alternative inputs which essentially means that internally the behaviour of the objects 006 and 007 is independent (concurrent). Therefore, if the edge from P1 to P2 fires, then object 007 is able to start executing within P2 (interact with object Connection), and thereafter return to P1 regardless of what 006 does.

This example describes a system where secret agents move between locations and where the locations in a system are given by the names of interactions (here the inline interactions P1, P2 and P3). The example uses two different kinds of objects: static objects that remain in a given location, and dynamic objects that can move between locations in a system. Dynamic or mobile objects can be identified because they give the name and/or type to input/output pins. Here, Remote, MI5 and Connection are static objects whereas 006 and 007 are dynamic objects. To simplify the identification of static and dynamic objects in an IOD we adopt the convention that only dynamic objects are indicated in the heading of the IOD under the caption lifelines.
Figure 8: Secret agents example.

Notice that in the example of Figure 8 we omit a final state as the objects involved will carry out their interactions indefinitely. If a final state was present, it would be necessary to nest the initial fork with a join (as required for IODs according to the specification [3]). In order to avoid having to represent the initial state and fork, we introduce a tagged value \( \{ \text{initBound} = n \} \) which we write next to a pin to indicate the initial number of tokens \( n \) associated to that pin. If this tag is not given next to a pin then we are implicitly assuming \( \{ \text{initBound} = 0 \} \). Using the tag \( \text{initBound} \) simplifies our model as we do not have to indicate the initial state and fork and any required token constraints.

As an AD in UML 1.x was a special kind of state machine, it was possible to indicate events and actions on activity edges (then called transitions). A typical transition has a text label given by \( e[c]/a \) where \( e \) is the event, \( c \) is a guard condition (Boolean expression) and \( a \) is an action that will be executed if and when the transition fires. However, this is no longer available for ADs in UML2.0 as activity edges can only represent control or data flow. For modelling mobility with activity edges, it is useful to be able to indicate, if intended, the explicit action that corresponds to the movement of an object from one location to another. To avoid notational clashes we assume that we can indicate this additional action at the source pin of an activity edge. In the UML specification, a pin has a name and type (one or the other may be omitted). We assume here that a pin can have an additional action, as well as other relevant information on that action, for example the underlying rate. The following is therefore the textual label of a pin: \( \text{name:type;action/rate} \).

Actions (and corresponding rates) associated to moving between locations are not indicated in the IOD of Figure 8. Further, we did not consider rates associated to messages when capturing communication between objects. The IOD is extended with such information in Figure 9. Notice that the notation used here for message names and rates, namely \( \text{msg/rate} \), is not to be confused with the notation available in UML1.x to indicate message precedence. Whilst in UML 1.x \( a/b \) could be used as a message label to indicate that message \( b \) could only occur provided \( a \) occurred before, this notation is no longer available in UML 2.0. We use it here to denote the associated
rate of a message.

Figure 9: Extended secret agents example.

3 From IODs to PEPA nets: a formal translation

We can in general obtain a PEPA net model directly from an IOD provided the IOD has all the necessary performance related information.

An interaction overview diagram can be viewed as a PEPA net model where each place corresponds to a node of the IOD, and a firing transition between places corresponds to an edge between the respective nodes in the IOD.

<table>
<thead>
<tr>
<th>UML 2</th>
<th>PEPA nets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interaction overview diagram (IOD)</td>
<td>PEPA net</td>
</tr>
<tr>
<td>IOD node</td>
<td>Place</td>
</tr>
<tr>
<td>IOD edge</td>
<td>Firing transition</td>
</tr>
<tr>
<td>Object inside a node</td>
<td>PEPA component</td>
</tr>
<tr>
<td>Object inside a node with corresponding</td>
<td>Token</td>
</tr>
<tr>
<td>input and output pins</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Translation of UML 2 items to PEPA nets

How to obtain PEPA components  A static PEPA component is represented by an object inside an IOD node. An object is defined by its name and its type with the following syntax: \textit{name : type}. The name of an object being optional, the PEPA component name used will be the type of the object. Inside the IOD node, the behaviour of an object is described using a sequence diagram. From this diagram, we can derive the complete behaviour of the static component.

Unlike a static component, a token is a dynamic object of the IOD which may leave a node, via the output pin, and enter another node through an input pin. A type can be associated with
a pin which allows us to distinguish between the tokens types. This information can be obtained from the label on the IOD edge which links the output pin to the input pin (see Figure 10).

![Figure 10: Input and output pins](image)

**How to obtain PEPA activities** Besides the type of the pin, the label on an IOD edge at the output pin level contains an action type with its corresponding rate. As it is shown in Figure 10 the complete syntax is the following: `pin_type;action_type/rate`. This information will be translated in the PEPA model as the activity `(action_type, rate)` of a firing transition between the two places representing the nodes.

A local activity `(a, r)` to a PEPA component `C` is the translation of a message `a/r` on the sequence diagram that an object of type `C` sends to itself.

A cooperation activity between two PEPA components `C_1` and `C_2` in a place `P` is the translation of a message, in the sequence diagram of node `P`, that an object of type `C_1` sends to an object of type `C_2`. This message, which is noted `b/r_1\triangle r_2`, consists of an action type `b`, and two rates `r_1` and `r_2`. This action type will be the one on which both PEPA components `C_1` and `C_2` will cooperate with rates `r_1` and `r_2` respectively. We can distinguish between an active component and a passive one by considering which corresponding object sends the message as follows:

- In an IOD node, if an object `C_1` sends a message of the form `b/r_1` to another object `C_2`, then this message is equivalent to `b/r_1\triangle \top` and this means that, in the context of PEPA nets, `C_1` is an active component for action type `b` whereas `C_2` is a passive one.

- Similarly if an object `C_1` receives a message of the form `b/r_2` from an object `C_2`, then `C_1` should be translated as a passive component regarding action type `b`. Indeed this form of message is equivalent to `b/\top\triangle r_2`.

**How to obtain the number of cells** The number of cells in a place for a specific type of tokens can be obtained from the value assigned to parameter `upperBound` in the UML model. This parameter which specifies the maximum number of tokens that can enter an IOD node (see Figure 10) is noted as `{upperBound=value}` on the input pin of the same type as the tokens.

**Translating the UML interaction operators** In the following we show how we translate the UML interaction operators using PEPA operators.

1. **The alt operator**: in UML, the interaction operator `alt` allows us to model alternative behaviours. Figure 11 shows two cases of alternative behaviour. The first one involves only one object (of type `C`) which sends to itself either message `a_1/r_1` or message `a_2/r_2`. This behaviour can easily be translated in PEPA using the choice operator. In this case messages `a_1/r_1` and `a_2/r_2` become activities `(a_1, r_1)` and `(a_2, r_2)` respectively and both are local to component `C`. We will then have: `C \equiv (a_1, r_1).C' + (a_2, r_2).C''`. 


In the second case, two types of objects $C_1$ and $C_2$ are involved. Object $C_1$ sends to object $C_2$ either message $a_1/r_1\parallel r_2$ or message $a_2/r_1\parallel r_2$. This is also translated in PEPA using the choice operator, but in this case actions $a_1$ and $a_2$ are shared actions between components $C_1$ and $C_2$, the resulting activities being $(a_1, \min(r_1, r_2))$ and $(a_2, \min(r_1, r_2))$. In the PEPA model, we will then have: $C_1 \triangleright_{a_1,r_1} C'_1 + (a_2, r_1).C''_1$ and $C_2 \triangleright_{a_2,r_2} C'_2 + (a_2, r_2).C''_2$ with $C_1 \triangleright_{(a_1,a_2)} C_2$.

2. **The par operator**: Two components evolving independently are said to be in parallel. In UML this behaviour is represented using the interaction operator $\text{par}$. In Figure 12, two cases of parallel behaviours are considered. In the first one, two objects $C_1$ and $C_2$ evolve independently of each other. The order in which the messages are sent is not deterministic. The first message to be sent can be either $a_1/r_1$ or $a_2/r_2$. The translation to PEPA net of this case leads to component $C_1 \parallel C_2$ where $C_1 \triangleright_{(a_1,r_1)} C'_1$ and $C_2 \triangleright_{(a_2,r_2)} C'_2$. Note that in this case both activities are individual activities.

3. **The loop operator**: A loop defines a repetition of the same action several times. In sequence diagrams, that will be, sending the same message(s) several times as in Figure 13. If only one object is involved in this sending, then this can be translated in PEPA nets as: $C_1 \triangleright_{(a,r)} C_1$. In the other case (see Figure 13), we can have an object $C_1$ sending a message to another object $C_2$ and that will be translated as the cooperation between two components $C_1 \triangleright_{(a)} C_2$ such as $C_1 \triangleright_{(a,T)} C_1$ and $C_2 \triangleright_{(a,r)} C_2$. 

Figure 11: The interaction operator $\text{alt}$

![Figure 11: The interaction operator $\text{alt}$](image1)

Figure 12: The parallel behaviour

In the second case, the activities involved are shared activities between two components $C_1$ with $C_2$ for $a_1$ and $C_2$ with $C_3$ for $a_2$, that is $C_1 \triangleright_{(a_1)} C_2$ and $C_2 \triangleright_{(a_2)} C_3$. 

![Figure 12: The parallel behaviour](image2)

Figure 13: The cooperation behaviour
4 Example: the secret agents

Consider the secret agents example introduced in Section 2.3. The corresponding PEPA net model (Figure 14) can be directly obtained from the IOD provided in Figure 9. The resulting net is composed of three places $P_1$, $P_2$ and $P_3$ where two tokens $Agent_{006}$ and $Agent_{007}$ can move. The mobility of the first token is restricted to places $P_1$ and $P_2$, whereas the mobility of the second one is restricted to places $P_2$ and $P_3$.

![Figure 13: The loop operator](image1)

![Figure 14: A PEPA net example](image2)

The static components Inside each place there is a static component with which the current component token in this place must cooperate to achieve an activity. These components are given in the following in the numbering order of the places where they are.

- $Remote \overset{\text{def}}{=} (download, \top).Remote$
- $Connection \overset{\text{def}}{=} (save, \top).Connection + (select, \top).Connection$
- $MI5 \overset{\text{def}}{=} (process, \top).MI5$

The components tokens When agent $Agent_{007}$ is in $P_1$, he has to cooperate with the local static component $Remote$ to download information. Once this task is achieved, this agent has to move to $P_2$ where it has to save the data in its possession. To achieve this task, it has to cooperate with the local static component $Connection$ on action $save$. Once this is done, agent $Agent_{007}$ has to move back to place $P_1$. The behaviour of the token $Agent_{007}$ is described by the following PEPA component.

- $Agent_{007} \overset{\text{def}}{=} (download, r_1).Agent_{007} + (save, r_2).Agent_{007}'$
- $Agent_{007}' \overset{\text{def}}{=} (move, \top).Agent_{007}$

When $Agent_{006}$ is in $P_2$, it has to select the data it is interested in by cooperating with component $Remote$. Then it has to move to place $P_3$ to process these data with the help of the local static component $MI5$. The complete behaviour of the token $Agent_{006}$ is described by the
following PEPA component.

\[
\begin{align*}
\text{Agent}_006 & \overset{\text{def}}{=} (\text{select}, r_3) . \text{Agent}'_006 + (\text{process}, r_4) . \text{Agent}'_006 \\
\text{Agent}'_006 & \overset{\text{def}}{=} (\text{move}, \top) . \text{Agent}_006
\end{align*}
\]

The places The components which may be in place \( P_1 \) are \( \text{Agent}_007 \) and the static component \( \text{Remote} \) which have to cooperate on activity \( \text{download} \). Therefore, place \( P_1 \) is defined as the synchronisation of these two components on this activity.

Tokens \( \text{Agent}_007 \) and \( \text{Agent}_006 \) evolve independently in place \( P_2 \), but have to cooperate with the local static component \( \text{Connection} \). Thus place \( P_2 \) is defined as the synchronisation of this component with, on one hand, \( \text{Agent}_007 \) on activity \( \text{save} \), and on the other hand, with \( \text{Agent}_006 \) on activity \( \text{select} \).

Finally, place \( P_3 \) is defined as the synchronisation of token \( \text{Agent}_006 \) and component \( \text{MI5} \) on activity \( \text{process} \). \( \text{Agent}_006 \) and \( \text{MI5} \) are the only components possible in this place.

\[
\begin{align*}
P_1 & \overset{\text{def}}{=} \text{Remote} \\
\text{Agent}_007 & \overset{\text{def}}{=} \text{Agent}_007 \\
P_2 & \overset{\text{def}}{=} \text{Connection} \\
\text{Agent}_007 & \overset{\text{def}}{=} \text{Agent}_007[\text{Agent}_007] \\
P_3 & \overset{\text{def}}{=} \text{MI5} \\
\text{Agent}_006 & \overset{\text{def}}{=} \text{Agent}_006[\text{Agent}_006]
\end{align*}
\]

5 Conclusions

To the best of our knowledge, there is currently no UML 2.0 tool that already allows the revised activity diagrams, interaction overview diagrams or sequence diagrams. While tools may be under development for the new notation available in UML 2.0, this difficults the implementation of our approach. It is also likely that XMI has not yet been adequately extended to cope with the new notation.

We plan to extend our work to use more standard ways of representing performance information by using the notation provided in the UML profile for schedulability, performance and time, as well as relevant standard notation for mobility. Relevant in this context are existing profiles for mobility and their extensions considering performance related information. A full version of this paper will appear as a technical report at the University of Birmingham.

References


