Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 232 (2009) 145–163
www.elsevier.com/locate/entcs

Performance Analysis of a Software Retrieval Service

Leïla Kloul [1]

Laboratoire PRiSM
Université de Versailles
45, Avenue des Etats-Unis
78000 Versailles, France

Abstract

The work we present in this paper is based on an approach we have developed to model mobility and performance information at the design level. This approach consists in translating a UML2.0 model onto a process algebra, namely PEPA nets, model. Once the process algebra model generated, performance analysis of the modelled system can be carried out. In this paper, we show how to use this approach to investigate the performance of a software retrieval service.

Keywords: Distributed systems, Mobile components, Performance engineering, Stochastic process algebra, UML2.0

1 Introduction

Performance engineering designates a collection of methods and concepts to support performance-oriented system development. It is understood as an activity which aims to demonstrate whether the final version of the system being developed will meet the performance needs and the resources constraints as specified by the customer.

The methods and concepts used in performance engineering are taken from different areas among which we have performance analysis and prediction, and software engineering. Performance analysis includes all the techniques for the study of system dynamics from the, often conflicting, perspectives of timeliness of behaviour and efficient use of resources. Such a study can be carried out by direct experimentation, monitoring and measurement. However, in the domain of computer systems, it is often important that the analysis is carried out before the system is constructed or configured and therefore modelling is widely employed. However,

[1] Email: kle@prism.uvsq.fr

doi:10.1016/j.entcs.2009.02.055

many performance modelling techniques do not allow the correct representation of many modern software systems which require the distinction between several contexts of computation which may depend on physical location, operating conditions, or both.

PEPA nets [3] is a performance modelling technique which makes a clear distinction between local communications and the movement or migration of processes. Thus, in order to model mobility and performance information at the design level, we have developed an approach based on both UML2.0 notation, mainly *Interaction Overview Diagram* (IOD), and PEPA nets [5]. This approach describes a mobile system at two levels. At the high level we describe the locations of the system and how objects move between locations which is given in UML by an IOD. At the lower level we describe how objects behave and interact locally. This is given by the individual nodes of the IOD, namely sequence diagrams. Both levels are enriched with performance related information. The developed approach does, in particular, allow us to define an automatic 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) are translated onto PEPA components. Once the PEPA net model generated, performance analysis of the modelled system can then be carried out.

Such an approach allows designers using UML2.0 to model and analyse their models formally using available tools for PEPA nets. An advantage of this approach is therefore that designers in industry do not require knowledge of the underlying performance technique to be able to analyse their systems.

In this paper, we show how to investigate the performance of a software retrieval service using our approach. The retrieval service, which is based on knowledge-driven agents, allows mobile users to select, retrieve and install softwares. We show that our approach allows modelling both the service architecture and the service process itself in a natural manner.

Structure of the paper: in Section 2, after a brief overview of the UML diagrams used and the PEPA nets formalism, we present the main lines of the automatic translation of a UML model into a PEPA net one. In Section 3, we present the case study of a software retrieval service. We first describe the software retrieval service, then its corresponding UML model and the generated PEPA net model. Finally, the performance analysis of the studied service is developed. Concluding remarks and the extensions of this work are discussed in Section 4.

2 Modelling mobility using UML2.0 and PEPA nets

In [5] we have shown how UML2.0 together with PEPA nets can be used to model dynamic aspects of mobile applications. The combination of interaction overview diagrams and sequence diagrams are translated into PEPA nets models. In the following we give a brief overview of the UML diagrams used and PEPA nets formalism before giving the main lines of the automatic translation.

2.1 UML2.0 diagrams

In the following, we present the main UML diagrams we use in our approach.

2.1.1 Sequence diagrams

These diagrams are the more commonly used diagrams for capturing inter-object behaviour. In UML2.0, a sequence diagram is enclosed in a frame and the five-sided box at the upper left-hand corner names the sequence diagram (sd). Further, interactions can be structured using so-called interaction fragments. Each interaction fragment has at least one operator held at the upper left corner of the fragment. Figure 1 shows an example of a sequence diagram using UML2.0 constructs. The

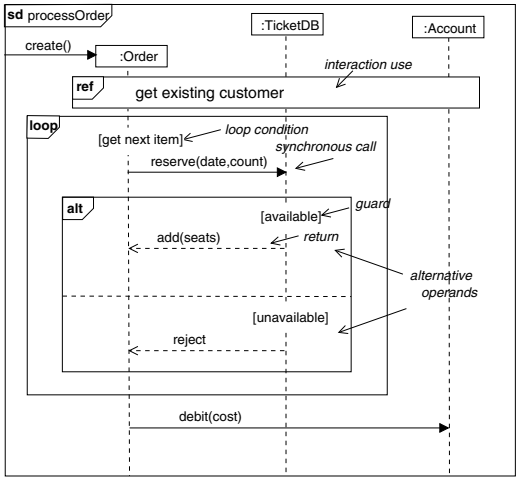

Fig. 1. A sequence diagram.

semantics of an interaction operator is described informally in the UML2.0 superstructure specification [7]. Below we give the meaning of the operators used here:
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.

2.1.2 Interaction Overview Diagrams (IODs)

IODs are a high-level structuring mechanism provided in UML2.0. An IOD is a variation of an *Activity Diagram (AD)* used to describe a high-level view of the possible interactions in a system. It consists of locations, called IOD nodes, in which the interactions between the node's objects are described using sequence

diagrams. The edges between the IOD nodes indicate the flow or order in which these interactions occur [7].

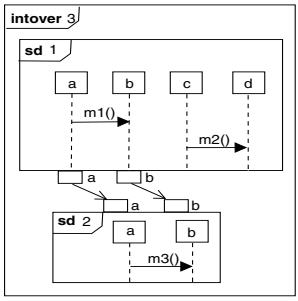

Fig. 2. Simple IODs with two inline interactions.

Even though IODs only describe control flow [7], the notion of object flow is implicitly present. The objects in the sequence diagram of a node can progress to a further inline interaction (IOD node) according to the edges at the IOD level. Moreover, from the IOD, we can derive the expected traces of behaviour for each of the instances involved.

Consider the example in Figure 2 showing an IOD with two inline interactions `sd1` and `sd2`. Since only objects `a` and `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 is possible for `a` and `b` to proceed to the second interaction after they have synchronised on message `m1()` independently of whether `c` and `d` have synchronised on message `m2()` or not.

All objects that want to progress from one interaction to another have an output pin with the name and/or 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. In Figure 2, interaction `sd2` can only start executing once one token `a` and one token `b` are available in the respective input pins, but regardless of whether message `m2` has been sent or not. 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. It is also possible that a pin can only accept a certain number of tokens. We write $\{upperBound = N\}$ next to a pin to indicate that the maximum number of tokens that can be stored in that pin is N. If the current number of tokens at the pin is N and the pin is an input pin, then no edge leading to that pin is allowed to fire.

With both interpretations of sequential composition at an IOD level we obtain a powerful language to model and structure interactions. For more details, we refer the reader to [5].

2.2 PEPA nets

PEPA nets combine the process algebra formalism PEPA [4] with coloured stochastic Petri nets. The colours used as the tokens of the net are PEPA components.

In PEPA, as in most performance modelling formalisms, we have a single modelling mechanism, *activities*, to represent changes of state within a system. In a PEPA net, there are two types of change of state. We refer to these as *firings* of the net and as *transitions* of PEPA components. Transitions of PEPA components will typically be used to model *local* changes of state as components undertake activities. Firings of the net will be used to model *global* changes of state such as context switches or a mobile software agent moving from one network host to another.

The tokens of a PEPA net are terms of the PEPA stochastic process algebra which define the behaviour of components via the activities they undertake and the interactions between them. The syntax of PEPA nets is given in Figure 3.

$$N ::= D^{+} M \qquad \text{(Net: Definitions and Marking)}$$

[0.5pt] *Identifier Definitions*

$$\begin{aligned} D &::= I \stackrel{def}{=} S && \text{(component defn)} \\ &\mid \mathbf{P}[C] \stackrel{def}{=} P[C] && \text{(place defn)} \\ &\mid \mathbf{P}[C,\ldots] \stackrel{def}{=} P[C] \underset{L}{\bowtie} P && \text{(place defn)} \end{aligned}$$

[1pt] *Marking vectors*

$$\begin{aligned} M &::= (M_{\mathbf{P}},\ldots) && \text{(marking)} \\ M_{\mathbf{P}} &::= \mathbf{P}[C,\ldots] && \text{(place marking)} \end{aligned}$$

[1pt] *Sequential components*

$$\begin{aligned} S &::= (\alpha, r).S && \text{(prefix)} \\ &\mid S + S && \text{(choice)} \\ &\mid I && \text{(identifier)} \end{aligned}$$

[1pt] *Concurrent components*

$$\begin{aligned} P &::= P \underset{L}{\bowtie} P && \text{(cooperation)} \\ &\mid P/L && \text{(hiding)} \\ &\mid P[C] && \text{(cell)} \\ &\mid I && \text{(identifier)} \end{aligned}$$

[1pt] *Cell term expressions*

$$\begin{aligned} C &::= \text{`\_'} && \text{(empty)} \\ &\mid S && \text{(full)} \end{aligned}$$

Fig. 3. The syntax of PEPA nets.

In that grammar, S denotes a *sequential component* and P denotes a *concurrent component* which executes in parallel. I stands for a constant which denotes either a sequential or a concurrent component, as bound by a definition.

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). Like a memory location in an imperative program, a cell is a storage area to be filled by a datum of a particular type. In particular in a PEPA net, a cell is a storage area dedicated to storing a PEPA component of the specified type. The components which fill cells can circulate as the tokens of the net. In contrast, the static components cannot move.

As a PEPA net differentiates between two types of change of state, so we differentiate the action types associated with each of these. The set of all firings is denoted by $\mathcal{A}_f$. The set of all transitions is denoted by $\mathcal{A}_t$.

By definition, a PEPA net $\mathcal{N}$ is a tuple $\mathcal{N} = (\mathcal{P}, \mathcal{T}, I, O, \ell, \pi, \mathcal{C}, D, M_0)$ such that

- $\mathcal{P}$ is a finite set of places;
- $\mathcal{T}$ is a finite set of net transitions;
- $I : \mathcal{T} \to \mathcal{P}$ is the input function;
- $O : \mathcal{T} \to \mathcal{P}$ is the output function;
- $\ell : \mathcal{T} \to (\mathcal{A}_f, \mathbb{R}^+ \cup \{\top\})$ is the labelling function, which assigns a PEPA activity ((type, rate) pair) to each transition. The rate determines the negative exponential distribution governing the delay associated with the transition. Note that rate $\top$ specifies that the component is passive regarding the activity. The rate will be determined by another component during the cooperation;
- $\pi : \mathcal{A}_f \to \mathbb{N}$ is the priority function which assigns priorities (represented by natural numbers) to firing action types;
- $\mathcal{C} : \mathcal{P} \to P$ is the place definition function which assigns a PEPA context, containing at least one cell, to each place;
- D is the set of token component definitions;
- M_0 is the initial marking of the net.

As explained above, each token has a type given by its definition. This type therefore determines the transitions and firings which a token can engage in; it also restricts the places in which it may be, since it may only enter a cell of the corresponding type.

PEPA net behaviour is governed by structured operational semantic rules which give rise to a labelled transition system. This gives rise to a CTMC (Continuous Time Markov Chain) which can be solved to obtain a steady state probability distribution from which performance measures can be derived.

2.3 Blending UML2.0 and PEPA nets

In [5], we have showed that a direct correspondence between the IOD nodes and the objects in the UML model, with, respectively, the places and the components in the PEPA net model can be built. And to obtain a complete model, we need to take into consideration at the UML level, not only the different types of components and the places where they may evolve, but also the notion of activity, that is an action type with its corresponding rate.

From the UML model, we can deduce the action types from the messages in the sequence diagrams, but not the rates as they are not provided. Thus, in [5] we have proposed to add the performance information, that is the *rate*, at the UML level. The syntax used for any message in a sequence diagram is: `action/rate`.

Moreover, the activity executed by a component when moving from one place to another one (labels on the firing transitions in the PEPA net) cannot be obtained from the UML model since this information is not reported on the transitions between the IOD nodes. For modelling mobility through edges in an IOD, 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. We have added this action at the source pin of an IOD edge (see Figure 4). Indeed as in the UML specification, a pin has a name and type (one or the other may be omitted), we have assumed 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: `name:type;action/rate`.

In order to avoid having to represent the initial state and fork, we have introduced a tagged value $\{initBound = n\}$ which we write next to a pin to indicate the initial number of tokens n associated with that pin. If this tag is not given then we are implicitly assuming $\{initBound = 0\}$. Using the tag $initBound$ simplifies our model as we do not have to indicate the initial state and fork and any required token constraints.

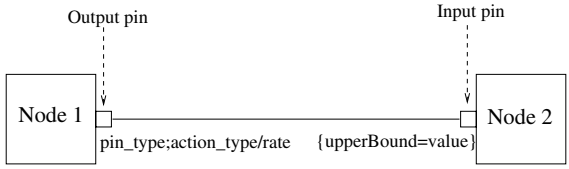

Fig. 4. Input and output pins

In order to avoid any ambiguity and obtain the exact sequence of derivatives of a PEPA component, we use a state machine diagram (SMD) for each object type. Moreover, like for the messages in the sequence diagrams, we annotate the SMD with performance information that is each transition between two states is labelled with an activity noted `action/rate`. When the rate of an activity is unspecified (noted $\top$ in PEPA net models), this rate is replaced by the symbol '$-$' like in $a/-$.

3 Case Study: a software retrieval service

The software retrieval service allows mobile users to select, retrieve and install software in an easy and efficient way. This service is based on knowledge-driven agents and has been introduced in [6]. Unlike existing solutions, such a software retrieval process, that manages semantic descriptions of available software, does not require from the users to know the location and the access method of remote software repositories [6].

The software retrieval service is situated in a concrete server called the Gateway Support Node (GSN) which consists of three places: the *Software Acquisition place*, the *Software place* and the *Broadcast place* (see Figure 5). An additional place to complete the architecture of the service is the *User place*.

Several agents take part in the software retrieval process. There are four static agents (*Knowledge Engineer*, *Broadcaster*, *Alfred*, *Software Manager*), one per place, and four mobile agents (*Integrator*, *Trader*, *Browser*, *Salesman*) who can travel from a place to another. In the following, we briefly describe the behaviour of the different agents in the places of the system.

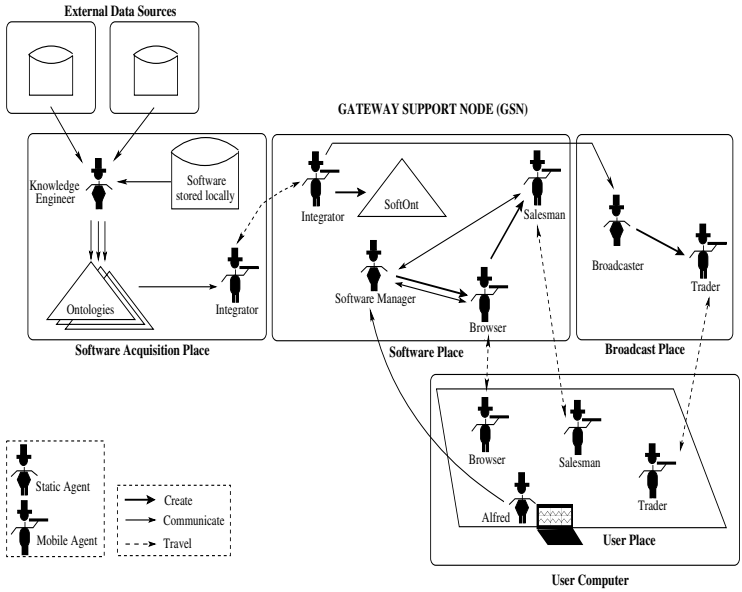

Fig. 5. Software Retrieval Service Architecture

• *The software acquisition place agents*: The *Knowledge Engineer* agent investigates the software repositories in order to obtain the semantics description (an ontology) for each repository. All the ontologies obtained by the *Knowledge Engineer* agent are carried by the *Integrator* agent to the software place to be integrated in order to build the ontology SoftOnt.

• *The broadcast place agents*: Once the *Integrator* agent performs the integration, it sends a message to the broadcast place in order to inform the *Broadcaster* agent about the new services provided by the GSN. Depending on these information, the *Broadcaster* agent can then create the *Trader* agent to carry the information to the user.

• *The user computer agents*: the user has at its disposal an efficient majordomo called *Alfred*. This static agent is in charge of storing all possible information about the user computer. Moreover, when an agent wants to show/retrieve data to/from the user, it has to do it with the help of *Alfred* who will create the appropriate user interface for each case.

Similarly, any time the user wants to perform an action or a request, it communicates with *Alfred* who will perform it by itself or by communicating with other agents. For example, once the user has expressed its need for a software and the query is built with the help of *Alfred*, this one sends a request to the software place where it will be attended by the software manager.

• *The software place agents*: the *Software Manager* agent creates and provides the *Browser* agent with the catalogue of the available software, according to the needs expressed by *Alfred* on behalf of the user. The *Browser* agent has to travel to the user place in order to refine the catalogue of software through its interactions with the user. Once the user selects a piece of software, the *Browser* agent creates the *Salesman* agent. This new agent carries the program selected by the user to its

computer, performs any electronic commerce interaction needed, and installs the program, when possible.

For more details about the software retrieval process, we refer the reader to [6]. In the following we show how to model the software retrieval service using our approach. We first present the UML model, then the generated PEPA net model.

3.1 The UML model

The software retrieval service can be modelled using an IOD. The agents involved in this service are translated into object types. Each mobile agent is modelled using a token object type, whereas each static agent is modelled as a *local* object type to a node. Moreover, the behaviour of each object in the IOD is described using a state machine diagram. All these diagrams are described and explained in the following.

3.1.1 The interaction overview diagram

The IOD describing the software retrieval service is given by Figure 6. The diagram consists of four nodes, one for each place of the retrieval service architecture. Two different kinds of objects are involved in this model: *static* objects which remain in a given node or location, and *dynamic* objects (tokens) which can move between locations of the system. Static objects are: `KEngineer`, `SoftManager`, `BroadCaster`, `Alfred`, and `User`. Dynamic or mobile objects can be identified because they give their type to input/output pins. These objects are: `Browser`, `Integrator`, `Salesman` and `Trader`.

Moreover, as in an IOD the communication between two objects may occur only when both objects are located in the same IOD node, the communications between two agents (*Alfred* and *Software Manager* in one hand, and *Integrator* and *Broadcaster* in the other hand) located in two different places of the software retriever service architecture are modelled as follows:

(i) because the communication between static agents *Alfred* and *Software Manager* may be indirect, it is modelled using a token object type `Enquirer` in the UML model. This token will travel to the IOD node modelling the software place to submit the query of the user to object type `SoftManager`.

(ii) the communication between object types modelling mobile agent *Integrator* and static agent *Broadcaster* is assumed to occur in the IOD node modelling the broadcast place where the static agent is located. Thus token type `Integrator` is assumed to travel to this IOD node in order to inform object type `Broadcaster` about the new services provided by the GSN before returning to IOD node modelling the software acquisition place. This solution takes advantage of the fact that *Integrator* is a mobile agent and the places where it travels are all located in the GSN. This avoids the introduction of an additional token type.

Figure 6 which aims to show the links between the IOD nodes represents only dynamic objects. Each output pin is labelled with the corresponding token type followed with the performance information, that is the activity (name/rate), required

when a token moves from one output pin of a node to the input pin of another one. To simplify the identification of static and dynamic objects in an IOD, only dynamic objects are indicated in the heading of the IOD under the caption **lifelines**.

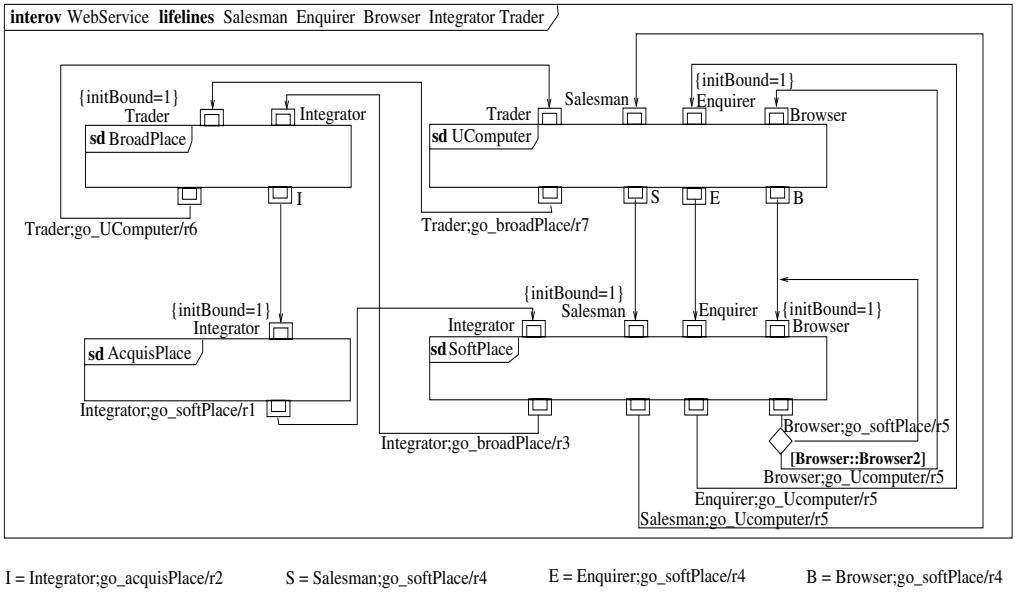

Fig. 6. Overview of the UML model

In the following, we describe the sequence diagrams of the different nodes. Note that unless specified by a UML operator, a strict sequencing ordering between the behaviours of the operands is considered.

• *Node BroadPlace*: this IOD node models the broadcast place in the architecture. Three types of objects are considered in the sequence diagram: dynamic objects `Trader` and `Integrator`, and the static object `Broadcaster`. Initially, there is one token of object `Trader` available at the corresponding input pin. However, unless a token of object `Integrator` arrives at its corresponding input pin the sequence of messages of the diagram will not occur.

Once the interaction in which token `Trader` (respectively `Integrator`) is involved is completed, it has to move to node *UComputer* (respectively *AcquisPlace*). This is specified by the performance information $go\_UComputer/r_6$ (respectively $go\_acquisPlace/r_2$) on the corresponding output pin.

• *Node AcquisPlace*: this node models the software acquisition place of the architecture. One dynamic object `Integrator` and one static object `KEngineer` are involved in the sequence diagram. Initially, there is one token of object `Integrator` available at the corresponding input pin. As `KEngineer` is a static object, no condition is required for the occurrence of message $new\_ontology/k_2$.

At the end of the interaction between the two objects, `Integrator` has to leave node *AcquisPlace* for node *SoftPlace* ($go\_softPlace/r_1$), in order to deliver the ontologies obtained by `KEngineer`.

• *Node SoftPlace*: this IOD node models the software place of the service architec-

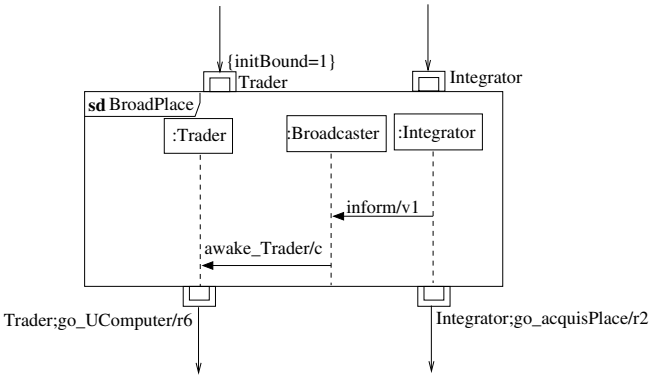

Fig. 7. IOD node modelling broadcast place

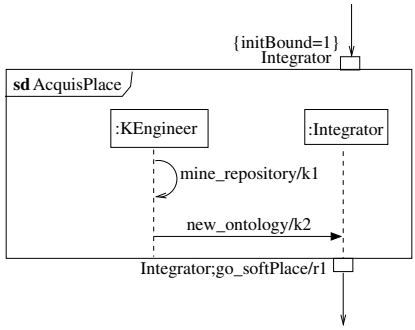

Fig. 8. IOD node modelling the software acquisition place

ture.

In the sequence diagram of this node, four types of tokens are involved: `Integrator`, `Salesman`, `Enquirer` and `Browser`. When the system starts, there is one token each for `Salesman` and `Browser`. So the messages in which objects `Integrator` and `Enquirer` are involved require the arrival of one token of each at the corresponding input pins. One additional type of objects, a static object, is involved in the sequence diagram of the node: `SoftManager`.

At the completion of the interactions in which `Browser` is involved, it has to leave the node either for node *UComputer* or to return to node *SoftPlace*. The joined node will be decided by the condition stated at the corresponding output pin, that is `[Browser::Browser2]`, `2` being the number of interactions `Browser` has already been involved in at this stage. Note that this type of numbering is used each time a condition is required.

Note that in notation $b_4; s_3$, b_4 and s_3 are the rates associated with objects `Browser` and `SoftManager` respectively.

• *Node UComputer*: this is the fourth and the last node of the IOD. It models the user's computer. Four types of token objects (`Browser`, `Enquirer`, `Salesman`, `Trader`) and two static object types (`User`, `Alfred`) are involved in the sequence diagram of this node. Among the token object types, only one token of type `Enquirer` is required initially.

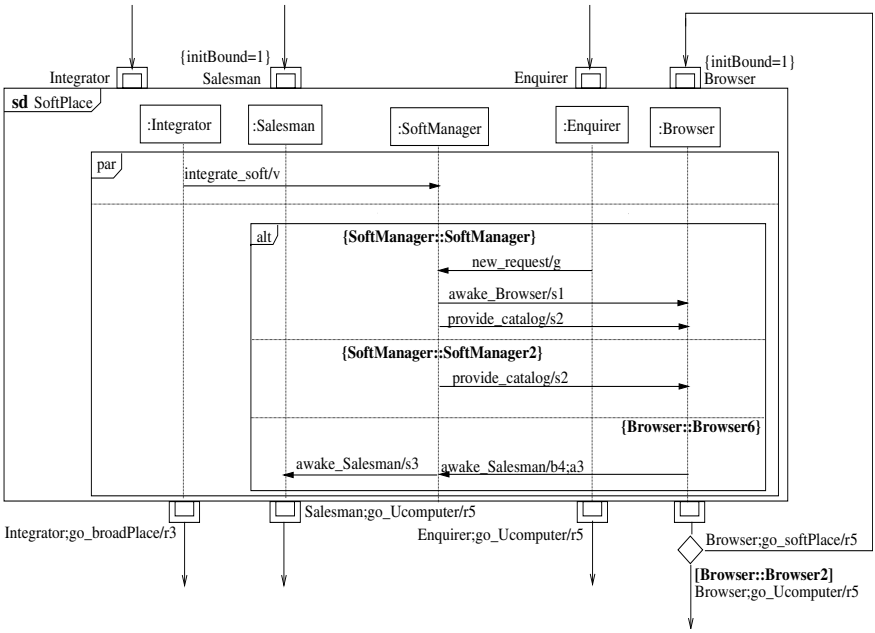

Fig. 9. IOD node modelling the software place

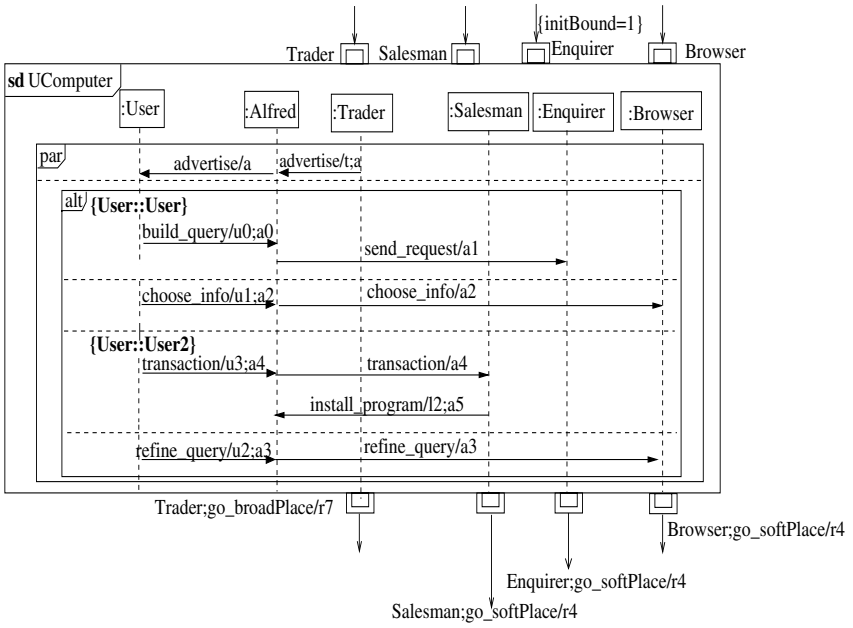

Fig. 10. IOD node modelling the User computer

In both UComputer and SoftPlace nodes, some options of the alternative behaviour modelled using the *alt* operator, require a state invariant before the execution of an interaction. For example, in *SoftPlace*, the interactions in the first option of the *alt* are possible only if the object type `SoftManager` is in its initial state, that is `[SoftManager::SoftManager]`, whereas the interaction

of the second option requires the object to be in state `SoftManager2`, thus `[SoftManager::SoftManager2]`.

Some of the interactions are labelled with the same message like those of the second option of the *alt* operator in UComputer node. These transitions are sequential in such a way that the time between them is assumed to be negligible.

3.1.2 The state machine diagrams

In the following, we present the SMD of object type `Browser`. The SMD of the other object types of the UML model are not presented as they are built in the same way.

Object type `Browser` is first woken up (awake_Browser/$\top$) by the `SoftManager` which will provide it with the catalogue (provide_catalog/$\top$). Bringing the catalogue, the `Browser` goes to *UComputer* (**go_UComputer**/r5). Once there, either the query is refined (refine_query/a3) or the software is chosen (choose_info/a2). In the first case, the `Browser` goes back to *SoftPlace* with a refined query to get a new catalogue. In the second case, it goes back to wake up `Salesman` (awake_Salesman/b4).

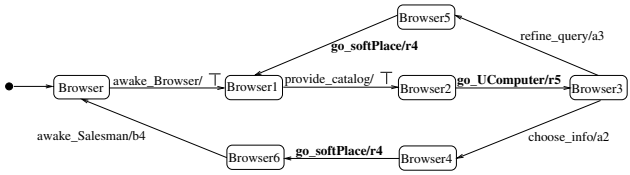

Fig. 11. The SMD modelling the browser

Using the information provided by the IOD and the different SMDs, we can translate the UML model into PEPA net using our approach. The PEPA net model obtained is described in the following section.

3.2 The PEPA net model

The result of translating the UML model into a PEPA net model is pictured in Figure 12. The PEPA net model consists of four places, each one corresponding to the IOD node of the same name. The symbols [_] in the places picture the empty cells which can receive token components. Obviously, each cell can receive only a token component of the same type. Those already filled show the initial position of the tokens in the net. This information is obtained from the value of the parameter *initBound* on the input pins of the IOD nodes. If this parameter is not specified, then the cell is initially empty.

The firing transitions between the places are labelled with activities obtained from the labels on the output pins of the IOD nodes. To avoid any ambiguity in the model, the activities appearing several times on the firings are numbered.

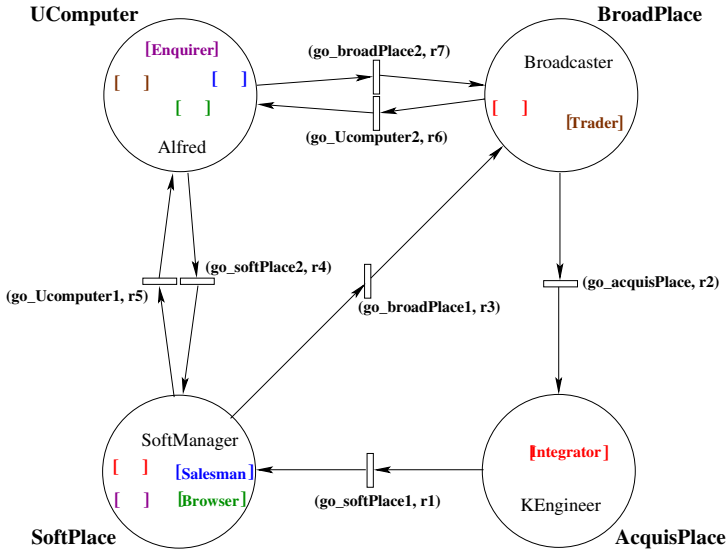

Fig. 12. The PEPA net model

3.2.1 Static components of the model

The static components model the static objects of the sequence diagrams. Thus these components are *KEngineer*, *SoftManager*, *Broadcaster*, *Alfred* and *User*. The behaviour of each of these components is derived from the sequence diagram in the IOD node where the component is located. Whereas its derivatives are obtained from the SMD of the corresponding object.

The KEngineer component The activities of the knowledge engineer modelled using component $KEngineer$ are mining repositories ($mine\_repository$) and then creating new ontologies ($new\_ontology$).

$$KEngineer \stackrel{def}{=} (mine\_repository, k_1).KEngineer_1$$
$$KEngineer_1 \stackrel{def}{=} (new\_ontology, k_2).KEngineer$$

The SoftManager component The behaviour of this component is derived from the sequence diagram in the IOD node *SoftPlace*.

$$\begin{aligned} SoftManager &\stackrel{def}{=} (new\_request, \top).SoftManager_1 \\ &\quad + (integrate\_soft, \top).SoftManager \\ SoftManager_1 &\stackrel{def}{=} (awake\_Browser, s_1).SoftManager_2 \\ &\quad + (integrate\_soft, \top).SoftManager_1 \\ SoftManager_2 &\stackrel{def}{=} (provide\_catalog, s_2).SoftManager_2 \\ &\quad + (integrate\_soft, \top).SoftManager_2 \\ &\quad + (awake\_Salesman, s_3).SoftManager \end{aligned}$$

The Broadcaster component The behaviour of this component is derived from

the sequence diagram in the IOD node *BroadPlace*.

$$Broadcaster \stackrel{def}{=} (inform, \top).Broadcaster_1$$
$$Broadcaster_1 \stackrel{def}{=} (awake\_Trader, c).Broadcaster$$

The User component The behaviour of this component is derived from the sequence diagram in the IOD node *UComputer*.

$$User \stackrel{def}{=} (build\_query, u_0).User_1 + (advertise, \top).User$$
$$User_1 \stackrel{def}{=} (choose\_info, u_1).User_2 + (advertise, \top).User_1$$
$$+ (refine\_query, u_2).User_1$$
$$User_2 \stackrel{def}{=} (transaction, u_3).User + (advertise, \top).User_2$$

The Alfred component Like for component $User$, the behaviour of component $Alfred$ is derived from the sequence diagram in the IOD node *UComputer*.

$$Alfred \stackrel{def}{=} (build\_query, a_0).Alfred_1 + (advertise, a).Alfred$$
$$Alfred_1 \stackrel{def}{=} (send\_request, a_1).Alfred_2 + (advertise, a).Alfred_1$$
$$Alfred_2 \stackrel{def}{=} (choose\_info, a_2).Alfred_3 + (refine\_query, a_3).Alfred_2$$
$$+ (advertise, a).Alfred_2$$
$$Alfred_3 \stackrel{def}{=} (transaction, a_4).Alfred_4 + (advertise, a).Alfred_3$$
$$Alfred_4 \stackrel{def}{=} (install\_program, a_5).Alfred + (advertise, a).Alfred_4$$

3.2.2 The component tokens of the model

These components correspond to the dynamic objects in the IOD. We define them as follows:

The Enquirer component Initially the corresponding object is present in IOD node *UComputer*. Its behaviour is derived starting from there.

$$Enquirer \stackrel{def}{=} (send\_request, \top).Enquirer_1$$
$$Enquirer_1 \stackrel{def}{=} (\mathbf{go\_softPlace_2}, r_4).Enquirer_2$$
$$Enquirer_2 \stackrel{def}{=} (new\_request, g).Enquirer_3$$
$$Enquirer_3 \stackrel{def}{=} (\mathbf{go\_Ucomputer_1}, r_5).Enquirer$$

The Integrator component Initially the corresponding object is present in IOD node *AcquisPlace*. Its behaviour is as follows:

$$Integrator \stackrel{def}{=} (new\_ontology, \top).Integrator_1$$

$$Integrator_1 \stackrel{def}{=} (\mathbf{go\_softPlace_1}, r_1).Integrator_2$$

$$Integrator_2 \stackrel{def}{=} (integrate\_soft, v).Integrator_3$$

$$Integrator_3 \stackrel{def}{=} (\mathbf{go\_broadPlace_1}, r_3).Integrator_4$$

$$Integrator_4 \stackrel{def}{=} (inform, v_1).Integrator_5$$

$$Integrator_5 \stackrel{def}{=} (\mathbf{go\_acquisPlace}, r_2).Integrator$$

The Trader component Initially the corresponding object is present in IOD node *BroadPlace*. Starting from there, its behaviour is as follows:

$$Trader \stackrel{def}{=} (awake\_Trader, \top).Trader_1$$

$$Trader_1 \stackrel{def}{=} (\mathbf{go\_Ucomputer_2}, r_6).Trader_2$$

$$Trader_2 \stackrel{def}{=} (advertise, t).Trader_3$$

$$Trader_3 \stackrel{def}{=} (\mathbf{go\_broadPlace_2}, r_7).Trader$$

The Browser component Initially the corresponding object is present in IOD node *SoftPlace*. Its complete behaviour is as follows:

$$Browser \stackrel{def}{=} (awake\_Browser, \top).Browser_1$$

$$Browser_1 \stackrel{def}{=} (provide\_catalog, \top).Browser_2$$

$$Browser_2 \stackrel{def}{=} (\mathbf{go\_Ucomputer_1}, r_5).Browser_3$$

$$Browser_3 \stackrel{def}{=} (choose\_info, \top).Browser_4 + (refine\_query, \top).Browser_5$$

$$Browser_4 \stackrel{def}{=} (\mathbf{go\_softPlace_2}, r_4).Browser_6$$

$$Browser_6 \stackrel{def}{=} (awake\_Salesman, b_4).Browser$$

$$Browser_5 \stackrel{def}{=} (\mathbf{go\_softPlace_2}, r_4).Browser_1$$

The Salesman component Initially the corresponding object is present in IOD node *SoftPlace*. Its complete behaviour is as follows:

$$Salesman \stackrel{def}{=} (awake\_Salesman, \top).Salesman_1$$

$$Salesman_1 \stackrel{def}{=} (\mathbf{go\_Ucomputer_1}, r_5).Salesman_2$$

$$Salesman_2 \stackrel{def}{=} (transaction, \top).Salesman_3$$

$$Salesman_3 \stackrel{def}{=} (install\_program, l_2).Salesman_4$$

$$Salesman_4 \stackrel{def}{=} (\mathbf{go\_softPlace_2}, r_4).Salesman$$

3.2.3 The places in the net

As each place in the PEPA net model corresponds to one IOD node, it is then easy to define the possible interactions in a PEPA net place. Using the information

provided by an IOD, that is the presence/absence of a token object at the initial state of the system, the interactions between the objects, and finally the contains of the messages, we can define the places of the net as follows.

$$BroadPlace[\_,\_] \stackrel{def}{=} Broadcaster \underset{K_1}{\bowtie} (Integrator[\_] \,||\, Trader[Trader])$$

$$AcquisPlace[\_] \stackrel{def}{=} KEngineer \underset{K_2}{\bowtie} Integrator[Integrator]$$

$$SoftPlace[\_,\_,\_,\_] \stackrel{def}{=} SoftManager \underset{L_1}{\bowtie} (Integrator[\_] \,||\, Enquirer[\_] \,||\, (Browser[Browser] \underset{L_2}{\bowtie} Salesman[Salesman]))$$

$$UComputer[\_,\_,\_,\_] \stackrel{def}{=} User \underset{M_1}{\bowtie} (Alfred \underset{M_2}{\bowtie} (Browser[\_] \,||\, Enquirer[Enquirer] \,||\, Salesman[\_] \,||\, Trader[\_]))$$

The sets of activities on which the components synchronise (interact) are defined as follows:

$K_1 = \{inform, awake\_Trader\}$, $K_2 = \{new\_ontology\}$,
$L_1 = \{integrate\_soft, new\_request, awake\_Browser, provide\_catalog, awake\_Salesman\}$, $L_2 = \{awake\_Salesman\}$,
$M_1 = \{build\_query, choose\_info, refine\_query, transaction, advertise\}$,
$M_2 = \{send\_request, choose\_info, refine\_query, transaction, advertise, install\_program\}$.

The PEPA net model we obtain has 4416 states, 8218 transitions and 9888 firings.

3.3 The model analysis

Recently several tools have been developed to support UML2.0 features. However the new features of UML2.0, in particular IODs, are not all supported by these tools. Moreover few tools support the XMI Import/Export functions. Our approach has been implemented using the *Enterprise Architect* tool [2] which supports both the IOD and the XMI Import/Export functions.

As one of the sensitive performance measures for the software retrieval service is the response time, we have used the HYDRA analyser [2] which uses Imperial PEPA Compiler (IPC) [1]. HYDRA allows transitive analysis by computing functions such as the cumulative passage-time density function.

In our case we compute the cumulative passage-time distribution function for completing the complete retrieval service. The parameters values we have used in our experiments are reported in Table 1. For these values, the cumulative passage-time distribution function for completing the whole process of retrieving a software, that is from the request submission to the end of the transaction, is given in Figure 13.

The distribution has been computed for different values of parameters s_1, s_2 and s_3 (see Table 2). These parameters indicate the time required by the Software Manager to respectively, create agent *Browser* (action *awake_Browser*), provide the catalogue (action *provide_catalog*) and create agent *Salesman* (awake_Salesman).

[2] http://www.sparxsystems.com

Activity	**Value** $(1/x)$	**Activity**	**Value** $(1/x)$
v, k_1	30 secs	l_2, u_2, a_3, a_5	1 mn
k_2	200 ms	u_0, a_0	1.5 mn
r_1, r_2, r_3	12.5 ms	u_1, a_2	58 secs
r_4, r_5, r_6, r_7, t	50 ms	u_3, a_4	50 secs
a_1, c, g, b_4	10 ms	v_1	20 ms

Table 1
Average activity times in the model

These parameters define the load at the software manager place, besides v, the rate at which the ontologies are integrated.

	Case 1	**Case 2**	**Case 3**	**Case 4**	**Case 5**
s_1^{-1}, s_3^{-1}	$0.01ms$	$10ms$	$25ms$	$50ms$	$0.1sec$
s_2^{-1}	$0.05ms$	$50ms$	$0.125s$	$0.25ms$	$0.5ms$

Table 2
Values of s_1, s_2 and s_3

Figure 13 shows that as these rates increase (the times decrease), the chances that the whole process of retrieving a software completes in less than $5mn$ are higher (probability closer to 1). Moreover, if we study each case individually, this figure shows that, considering the parameters values used (see Table 1), about 100% of the requests will be satisfied in less than $5mn$ (`Case 1`) whereas less than 30% of the requests will be satisfied in less than $5mn$ (`Case 5`).

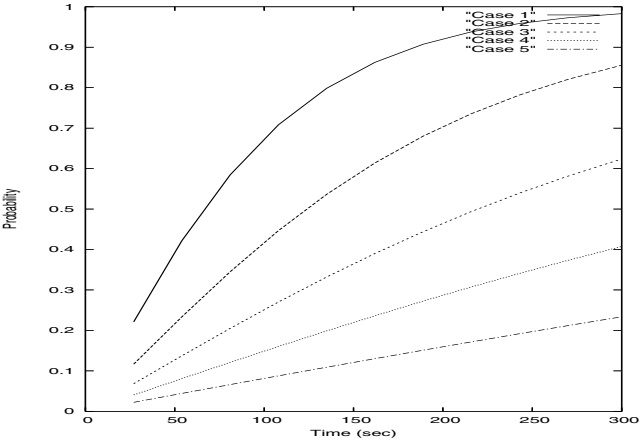

Fig. 13. Cumulative passage-time distribution function for completing a request

4 Conclusion

Using a software retrieval service application, we have showed how a UML2.0 model based on interaction overview diagrams and sequence diagrams can be translated into a PEPA net model, provided that state machine diagrams are used.

In the near future, we are interested in addressing crucial issues as design consistency and completeness. In this context, several approaches have been proposed in the literature. However, none of them is based on the IODs which are a characteristic of UML2.0. We are also interested in investigating the translation complexity.

References

[1] J. Bradley, N. Dingle, S. Gilmore, and W. Knottenbelt. Extracting passage times from PEPA models with the HYDRA tool: A case study. In *Proceedings of the Nineteenth annual UK Performance Engineering Workshop*, pages 79–90, 2003.

[2] N. Dingle. *Parallel Computation of Response Time Densities and Quantiles in Large Markov and Semi-Markov Models*. Phd. Thesis, Imperial College, London, 2004.

[3] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. PEPA nets: A structured performance modelling formalism. *Performance Evaluation, Elsevier Science*, 54:79–104, 2003.

[4] J. Hillston. *A Compositional Approach to Performance Modelling*. Phd. Thesis, University of Edinburgh, 1994.

[5] L. Kloul and J.Küster-Filipe. Modelling mobility with uml2.0 and PEPA nets. In K. Goossens and L. Petrucci, editors, *Proceedings of The Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, volume P2556, pages 153–162, Turku, 2006. IEEE Computer Society.

[6] E. Mena, A. Illarramendi, and A. Goñi. A software retrieval service based on knowledge-driven agents. In *CoopIS*, volume 1901 of *LNCS*, pages 174–185. Springer, 2000.

[7] OMG. *UML 2.0 Superstructure Specification*. OMG document ptc/05-07-04, available from www.uml.org, August 2005.