Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 49 (2007) 999–1016
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

An object-oriented approach to formally analyze the UML 2.0 activity partitions

Thouraya Bouabana-Tebibel [a,*], Mounira Belmesk [b]

[a] *Institut National d'Informatique, BP 68 M Oued-Smar, Alger, Algeria*
[b] *Edouard MonPetit College, Longueuil, Canada*

Received 3 January 2006; received in revised form 10 September 2006; accepted 23 October 2006
Available online 28 November 2006

Abstract

Nowadays, UML is the de-facto standard for object-oriented analysis and design. Unfortunately, the deficiency of its dynamic semantics limits the possibility of early specification analysis. UML 2.0 comes to precise and complete this semantics but it remains informal and still lacks tools for automatic validation. The main purpose of this study is to automate the formal validation, according a value-oriented approach, of the behavior of systems expressed in UML. The marriage of Petri nets with temporal logics seems a suitable formalism for translating and then validating UML state-based models. The contributions of the paper are threefold. We first, consider how UML 2.0 activity partitions can be transformed into Object Petri Nets to formalize the object dynamics, in an object-oriented context. Second, we develop an approach based on the object and sequence diagram information to initialize the derived Petri nets in terms of objects and events. Finally, to thoroughly verify if the UML model meets the system required properties, we suggest to use the OCL invariants exploiting their association end constructs. The verification is performed on a predicate/transition net explored by model checking. A case study is given to illustrate this methodology throughout the paper.
© 2006 Elsevier B.V. All rights reserved.

Keywords: UML 2.0; Object Petri Nets; Activity partitions; Sequence diagram; Object diagram

1. Introduction

UML activity diagrams [30] succeed as a standard in the area of organizational process modeling, in other terms the workflows, faced with alternative languages used by commercial workflow management systems for this purpose [13]. A workflow is a set of business activities that are ordered according to a set of procedural rules to deliver services [15]. Two important dimensions of workflows are the control flow dimension and the resource dimension [42]. The control flow dimension concerns the ordering of activities in time whereas the resource dimension concerns the organizational structure in terms of actors. To match with this definition, in UML 2.0 [29], the workflow modeling is expressed by activity partitions which are a process of actions performed by a group of actors to render a specific service. Nevertheless, UML suffers from ceaseless critics on the precision of its semantics when the verification of modeling correctness has become a key issue. UML 2.0 brings more precision on its semantics, but it remains informal and lacks tools for automatic analysis and validation.

On the other hand, Petri nets [23] are a mathematically precise model, supported by an abundance of tools for proving properties about them. Many works state that they constitute a natural technique for modeling workflows. Salimifard et al. [36] present an overview of existing works on modeling workflows using Petri nets. Van der Aalst shows in [42] that the application of Petri nets to workflow management is appropriate for modeling and analysis. However, their drawback, in contrast to the UML notation, is their high learning cost and their failure while modeling large-scale applications.

To fill the gaps of the UML notation and Petri nets for modeling workflows, we propose to formalize the UML

[*] Corresponding author. Tel.: +213 21 513697; fax: +213 21 516156.
E-mail addresses: t_tebibel@ini.dz (T. Bouabana-Tebibel), mounira.zoubeidi@college-em.qc.ca (M. Belmesk).

activity partitions using the Object Petri Nets, OPNs for short. Formalization of the UML state-transition diagrams has been already tackled in various works. In [5,35] the statecharts are transformed into OPNs and in [11,17] the activity diagrams are formalized with transition systems. However, through our multiple investigations, there is not yet works that are intended for the formalization of UML 2.0 workflow models via OPNs, see [14]. The OPNs are a generalization of ordinary Petri nets relying on the basic principles of the object-oriented approach- of which the modularity and encapsulation concepts- and supporting convenient definition and manipulation of object values. Lakos demonstrates in [25] that OPNs constitute a natural target formalism for the object-oriented modeling language since they provide a clean integration of the static (actors) and dynamic (actor's lifecycle and interactions) models. He applies his theory to the Rumbaugh's OMT notation [34].

Thus, the main idea from this paper is to map UML 2.0 workflow models to analyzable OPNs, in the context of a methodology which starts from the activity, object and sequence diagrams to provide a value-oriented validation of the workflows. This methodology is presented in Fig. 1, where the activity partitions are transformed into an Object Petri Net Model (OPNM) translating the dynamics and interactions of the actors. A main feature of this model is its genericity in such a way that it supports the modeling of any scenario. To deal with its analysis according to a value-oriented approach, the OPNM is first, initialized using the object and sequence diagrams. It is after that, validated using the system properties which are expressed in the Object Constraint Language OCL [31] and checked by means of a reachability analysis tool.

1.1. Initialization of the specification

Sequence diagrams constitute an attractive visual formalism, widely used to capture the system scenarios. They describe interactions by focusing on the sequencing of the exchanged messages among the objects. For formalization

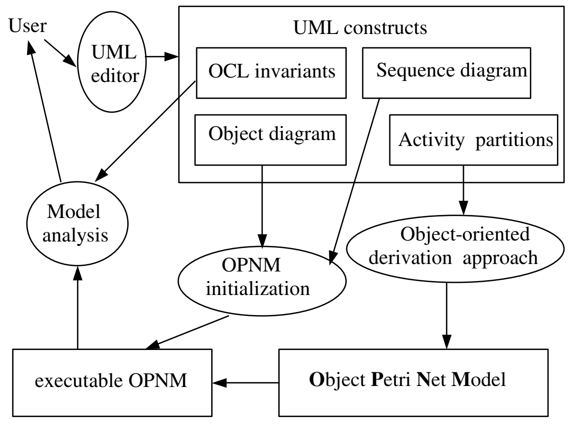

Fig. 1. Methodology of modeling and analysis.

purposes, sequence diagrams are generally combined with the statecharts in order to connect the object life cycles [8,43]. They are also transformed separately into other formalisms to validate specific scenarios [20] or composed together to describe the system's overall behaviour [41]. As far as we are concerned, we introduce three novel uses of the sequence diagrams for the purpose of validating the modeling. We call this specification *dynamic initialization*.

Contrary to the previous research works, we first, use the sequence diagram to initialize the specification. Considering the generic structure of the Petri net model (OPNM) which is derived from the activity partitions, this model supporting any form of the object interactions (scenario), we suggest to use the sequence diagram to initialize the Petri net marking with the events of the scenario that will be verified.

Second, we exploit the sequence diagram to provide an interface between the system and its environment. In spite of their suitability to describe complex behaviours such as parallelism or synchronization, one significant gap of the ordinary Petri nets is their incapacity to model open interactive systems. Many works such as [3] close this gap by proposing Open Petri nets that are ordinary Petri nets with a distinguished set of places (called open places) which are intended to represent the interface of the net towards the external world. Nevertheless, it still misses concrete propositions about the practical use of this solution when formalizing UML. Thus, we utilize the sequence diagrams to constitute the net interface towards its environment.

Finally, we utilize the creation and deletion information to instantiate or destroy the objects of the system. Another weakness of ordinary Petri nets is their failure to produce dynamically new objects. This deficiency is overcome in the literature with the super place constructs of the Object Petri nets [25], these places enabling the generation of new objects. Similarly to the open places, we propose an applicative use of the super places in the UML context using the *create* and *destroy* events.

On the other hand, object diagrams describe possible configurations. Also called instance diagrams, they show the links between the instantiated objects and their attribute values, at a given time. Data formalization are usually given by means of state-oriented languages such that Z or B [2,24,39]. We propose, as shown in Fig. 1, to specify data by means of the object diagrams. These data provide the Petri net initial marking with objects named by identities and attribute values. We call this specification *static initialization*.

1.2. Verification of the specification

The Petri nets resulting from the derivation and initialization processes, as illustrated in Fig. 1, are analyzed by means of PROD [33], a model checker tool for predicate/transition nets. Model checking is classified as the most appropriate technique for verifying operational UML models [6,19,28]. It allows a fast and simple way to check whether the property holds or not. To avoid the high learning cost of the model checker, we suggest that the UML designer specifies the system properties in the Object Constraint Language OCL which permits to formulate restrictions over UML models, in particular, invariants. The latter are afterwards, automatically translated to temporal logic properties in order to be verified by PROD during the Petri net analysis.

The invariants are specified on the class diagram which models the static structure of a system, in terms of classes and relationships between classes. A class describes a set of objects encapsulating attributes and methods. An association abstracts the links between the class instances. It has at least two ends, named association ends, each one representing a set of end objects with a size limited by a multiplicity.

However, a simple translation of OCL invariants to temporal logic properties is not sufficient for the property checking. Indeed, OCL expressions refer to classifiers (types, classes, etc.) to evaluate their attributes, association ends and methods. The attribute values and the methods are, respectively, provided by the object and sequence diagrams. As for the association ends, their values must be updated (created or modified) on the activity diagram by means of the link actions. So, in case the designer specifies OCL invariants for his models, we attract his attention on the necessity of modeling the actions treating the association ends in order his invariants could be adequately verified by the model checker. In other words, he is called on to specify by means of the link actions, the association end update that provides the dynamics of the object through the roles it plays. As far as we are concerned, we propose an approach to translate the link actions in Petri nets, in order to achieve the systematic formal verification of the OCL constraints.

1.3. Overview

In short, our most relevant contribution aims at a value-oriented validation of UML 2.0 activity partitions. For this purpose, we firstly, use the object and sequence diagrams to initialize the specification. Secondly, we resort to the OCL invariants which allow an evaluation of the system properties using UML concepts. However, the formal validation of these properties requires a specific object flow specification on the activity partitions, using the association end constructs. This approach has never been tackled by previous research works where the model initialization is generally based on anonymous objects and the object flow formalization is omitted [17] or partially tackled [11].

The remainder of the paper is organized as follows. Sections 2 and 3 define the syntax and semantics of the activity partitions and the OPNs. Section 5 presents the main constructs of the object-oriented approach that we propose. Sections 6–8 develop the transformation methodology and the used techniques to achieve a value-oriented formalization. These techniques are illustrated in a case study, presented in Section 3 and exposed throughout the paper. Section 9 deals with the validation strategy based upon the OCL invariants. In Section 10, the current trends of this research are given, discussing the novelty of the work presented herein and criticizing the limits of the works similar to ours. We conclude with some observations on the obtained results and recommendations on the future research directions.

2. Activity partition diagram syntax and semantics

2.1. Informal definition

The activity partition diagram is flowchart-like notations with constructs to express sequence, choice and parallel execution of activities. It is a specialization of the activity diagram where each actor (active object) performs its activities in a swimlane separated from neighbouring swimlanes by vertical solid lines on both sides. An activity is an amount of work represented by an action node or an activity node. An action is uninterruptible whereas an activity can be decomposed in a sequence of action nodes.

The workflow starts from one initial node (the black dot) and terminates in one final node (the bull's eye) such that every node is on a path from the initial node to the final node. Ovals and pentagons represent action nodes. These nodes are linked by directed *control* edges that express a sequence. Objects that are inputs to or outputs from an action node (exchanged objects) are represented by squares. They are connected to the action nodes using directed *object flow* edges. An edge may be internal or external. An internal edge links two nodes within the same swimlane whereas an external edge connects nodes which belong to different swimlanes. The actions are also internal or external. The internal actions specify the local treatments of an object while the external actions deal with the object interactions. Contrary to the internal actions which have only internal edges, the external actions have internal and external edges. A bar node models a synchronization. It represents a fork node (more than one outgoing edge) or a join node (more than one incoming edge). A diamond node represents an exclusive choice (more than one outgoing edge) or a merging of different choices (more than one incoming edge).

The swimlane actors interact by means of messages of *send* or *call* type. The ***send*** messages may be objects or signals. An object sending is represented by an object flow going from the sender's *SendObjectAction* to the receiver's *AcceptObjectAction*, see Fig. 2a. The stereotype «*send*» (resp. «*receipt*») is added to distinguish between an internal action and the *SendObjectAction* (resp. *AcceptObjectAction*) when a bar or diamond nodes are used.

A signal is shown with special notations. The signal sending (*SendSignalAction*) is a rectangle that has a protruding triangular point.The signal receipt (*AcceptEventAc-*

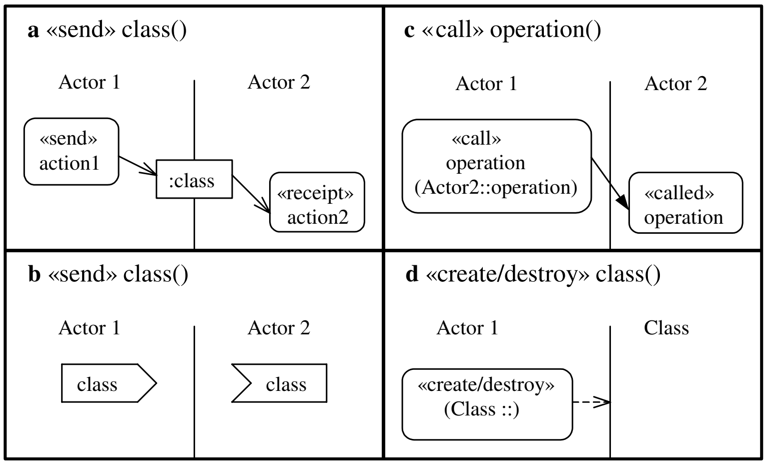

Fig. 2. Interaction actions on the activity partitions.

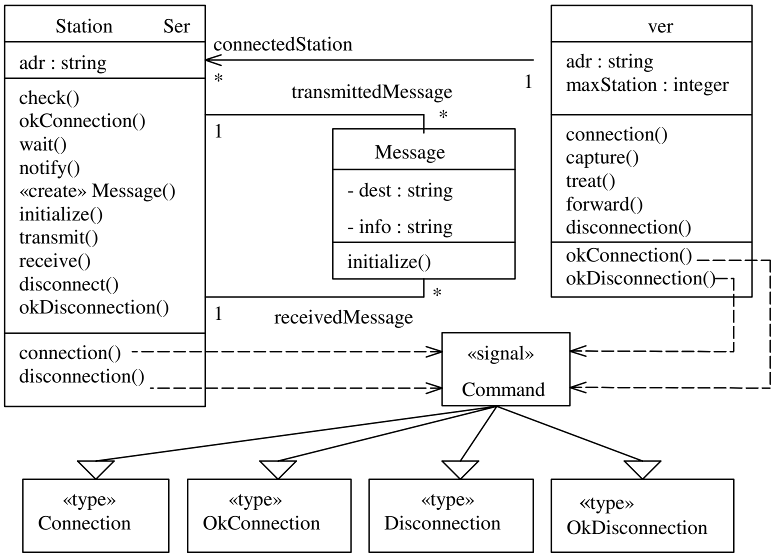

Fig. 3. Class diagram of the message server.

tion) is a rectangle with a notch in the side, see Fig. 2b. Both notations hold the name of the sent stereotype class «*signal*».

The ***call*** messages are operation calls, creation or deletion actions. The operation call is shown with an edge which leaves the caller action (*CallOperationAction*) towards the called one (*AcceptCallAction*). The *CallOperationAction* holds the name of the called operation, failing this, the operation name is mentioned after the target class name, see Fig. 2c. The stereotype «*call*» (resp. «*called*») permits to distinguish between an internal action and the *CallOperationAction* (resp. *AcceptCallAction*) when a bar or diamond nodes are used.

As for the object creation (*CreateObjectAction*) and deletion (*DestroyObjectAction*), they are shown as actions with the stereotypes «*create*» or «*destroy*», respectively, and an edge from the action towards the swimlane of the actor abstracting the object to be created or destroyed, see Fig. 2d. An example of the activity partition modeling is presented on Fig. 4.

2.2. Formal definition

More formally, the activity partitions can be defined by <*Node*, *Edge*, *Object*>, where each component designates sets of activity partition constructs and functions on these sets. We develop these components as what follows. To denote the UML constructs, we adopt the UML 2.0 symbolic notation given in [29].

Node regroups action, diamond and bar nodes as well as functions on these nodes returning constructs of node type. More formally, $Node = <S, S_{out}, S_{in}, D, B, Prec, Suc>$ where:

- $S = \{\text{InitialNode}, s_1, s_2, \ldots, s_n, \mathit{FinalNode}\}$ is a set of action nodes.

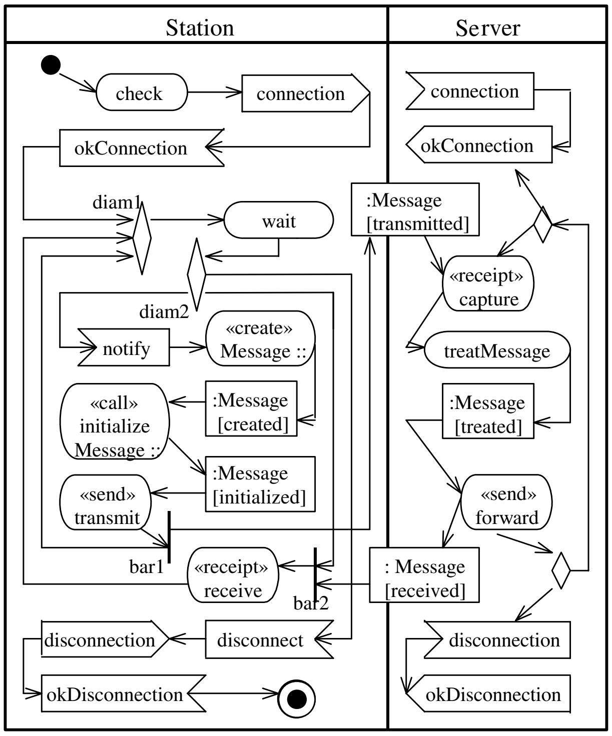

Fig. 4. Activity partition diagram of the message server.

- $S_{out} = \{SendObjectAction, SendSignalAction, CallOperationAction, CreatObjectAction, DestroyObjectAction\}$ are external output actions.
- $S_{in} = \{AcceptObjectAction, AcceptEventAction, AcceptCallAction\}$ are external input actions.
- $D = \{d_1, d_2, \ldots, d_n\}$ is a set of diamonds.
- $B = \{b_1, b_2, \ldots, b_n\}$ is a set of bars.
- *Prec* : $S \cup D \cup B \rightarrow P\ (S \cup D \cup B)$ is a precedence function such that $Prec(x_i) = \{s_1^{(i)}, s_2^{(i)}, \ldots s_k^{(i)}\}$ where $s_j^{(i)}$ is an actor's node that precedes x_i.
- *Suc*: $S \cup D \cup B \rightarrow P(S \cup D \cup B)$ is a succession function such that $Suc(x_i) = \{s_1^{(i)}, s_2^{(i)}, \ldots s_k^{(i)}\}$ where $s_j^{(i)}$ is an actor's node that follows x_i.

Edge consists of the model edges as well as functions on the action, diamond and bar nodes that return sets of their incoming edges. More formally, $Edge = <E, In, Xin>$, where:

- $E = \{e_1, e_2, \ldots, e_n\}$ is a set of edges.
- *In*: $S \cup D \cup B \rightarrow P(E)$ is an input edge function such that $In(x_i) = \{e_1^{(i)}, e_2^{(i)}, \ldots e_k^{(i)}\}$ where $e_j^{(i)}$ is an internal input edge to x_i.
- *Xin*: $S \cup D \cup B \rightarrow P(E)$ is an input edge function such that $Xin(x_i) = \{e_1^{(i)}, e_2^{(i)}, \ldots e_k^{(i)}\}$ where $e_j^{(i)}$ is an external input edge to x_i.

Object regroups sets of objects. More formally, $Object = <O, U>$ where:

- $O = \{o_1, o_2, \ldots, o_n\}$ is a set of active objects.
- $U = \{u_1, u_2, \ldots, u_n\}$ is a set of exchanged objects.

3. Case study

As defined in [1] a workflow system, in its general form, is basically a heterogeneous and distributed information system where the tasks are performed using autonomous entities. Resources, such as data, are typically required to process these tasks. We illustrate the activity partitions through the workflow of a system responding to this definition. So, the target application is a message server whose main role is to manage the communication between the connected stations. All the exchanged messages must go through this server, to be forwarded to the receivers. The corresponding class diagram is presented in Fig. 3, where the server is modeled by the *Server* class, the stations by the *Station* class and the exchanged messages by the *Message* class. The stereotype class *Command* models the used signals. It supports four static types: *Connection*, *Okconnection*, *Disconnection* and *Okdisconnection*.

After it performs a self diagnostic (check action), a station connects itself to the server. Its connection request is realized using the *connection* signal. The request message is received by the server through the *connection* action. The server confirms the station connection using the *okConnection* signal.

When connected, a station can transmit a message, receive a message or disconnect itself. After it receives a notification order (from a user for example), it creates a message, initializes it and then transmits it to the server. When receiving a client message, the server treats it and then forwards it to the receiver using the *forward* action. Its disconnection is requested by the *disconnection* signal, after reception of a disconnect order. The server confirms the station disconnection using the *okDisconnection* signal.

4. Object Petri net syntax and semantics

UML is an object-oriented notation well-suited to model complex systems due to its modularity. Petri nets offer an appropriate formalism for dynamics and concurrency, however ordinary Petri nets lack thorough modularization techniques. So, to merge UML and Petri nets, it is handier to move towards OPNs which adopt object-oriented structuring, this allowing the definition of clean interfaces for object interactions. Furthermore, OPNs can be transformed into behaviorally equivalent Colored Petri Nets [27], thus providing the basis for applying traditional analysis techniques.

OPNs have been formally [26] and informally [25] defined by Lakos. In the OPNs, classes are represented by subnets that can be instantiated as many times as needed to obtain objects. This instantiation is realized using tokens, in the form of n-tuples, as class instances. This is the most important feature of the OPNs since it is the way they support the dynamic creation of objects. In accordance with the object-oriented approach, the subnet encapsulates the class's attributes and methods (activities). The attributes are expressed as components of the tuple (token) which represents the object. As for the methods, they describe the object's life cycle (activity diagram) into a bunch of places, functions and transitions. The places are of two types: simple and super. The simple places hold tokens of simple type (such as integer, real, boolean, etc.), class type or a multiset of the above. The super place generates these tokens. It can act as a source or sink of tokens. The functions define the token consumption while the transitions modify the net's state according to the usual Petri net semantics. As for the super transition, it corresponds to a set of internal actions, determined from the abstract firing mode.

More formally, we define the Objet Petri Net by the 11-tuple $<P, T, A, C, Pre, Post, M_0>$ where:

- $P = \{p_1, p_2, \ldots, p_n\}$ is a set of places.
- $T = \{t_1, t_2, \ldots, t_n\}$ is a set of transitions.
- $A \subseteq P \times T \cup T \times P$, is a set of arcs.
- $C = \{C_1, C_2, \ldots, C_n\}$ is a set of colors defining object types where $C_i = \{<c_1, c_2, \ldots, c_k>\}$ and c_j is a variable or a constant.
- Pre: $P \times T \to P(C)$ is a precondition function for transition firing such that $Pre(p_i, t_i) = \{C_1, C_2, \ldots, C_k\}$.
- $Post$: $P \times T \to P(C)$ is a postcondition function for transition firing such that $Post(p_i, t_i) = \{C_1, C_2, \ldots, C_k\}$.
- M_o:$P \to C$ is the initial marking function, such that $M_o(p_i) = \sum_{k=1,K} C_k$.

5. Object-oriented derivation approach

5.1. Object-oriented specification

Due to UML's and OPN's suitability for the object-oriented modeling, we propose to transform each actor's swimlane activities into an object subnet called Dynamic Model or *DM* (see Fig. 5). The *DM* translates the life cycle of the objects that instantiate the swimlane's actor.

To deal with Petri net simulation, we tackle the Petri net initial marking which may be of two types: static and dynamic. Due to modularity, the static initial marking is defined for each *DM*. It provides the class instances and their attribute values. These instances are extracted from the object diagram and saved into the *Object* place in order to initialize the *DM* with tokens of *object* type (see Section 7.1). Also for modularity reasons, the dynamic initial marking concerns each *DM* separately. It provides the

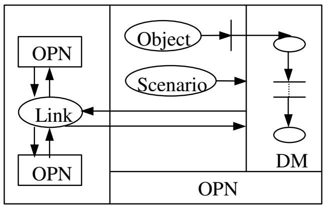

Fig. 5. OPNs interconnection architecture.

object interactions from the sequence diagram to initialize the *Scenario* place with tokens of *event* type (see Section 7.2). According to the Object Petri net approach, the *Object* and *Scenario* places are defined as super places and constitute with the *DM* an *OPN* model.

The actor interaction is abstracted in Petri nets by the *Link* place which receives the events generated by the *DMs* and then dispatches them to the target *DMs*.

5.2. Mapping of the activity partitions

Since the activity partition may contain activity nodes, its conversion to Petri nets requires that the activity nodes be flattened (just actions and arcs) [38]. When done, the flat diagram is converted into an Object Petri Net composing the *DM* of the *OPN* model. This mapping is performed conforming to the conversion rules that we define below.

The activity partition models the class objects' life cycle. An object life cycle is a sequencing of action, diamond or bar nodes with sets of incoming and outgoing edges that specify control flow or data flow.

5.2.1. Mapping of the actions

An action will not begin execution until all of its input conditions are satisfied (implicit join). The completion of the execution of an action may enable the execution of a set of successor nodes that take their inputs from the outputs of the action [29].

The sequencing of the object's actions within the swimlane, is expressed by means of the edges that connect them. Thus, even the actions with external input edge must be connected to the previous ones in the swimlane. In fact, in that case the sequencing may be given by the external edges, but regarding the modularity of the derivation approach this is not convenient any more.

An action is semantically equivalent to a Petri net transition. Each edge is an input to and output from the nodes which it connects. Only the internal input edges are translated. They are converted into arc-place-arc triplets if they come from action or join nodes. They are converted into arcs if they come from diamond nodes (see the diamond conversion).

The first action of the swimlane is the one without internal edges, excluding the loop edge or the edge coming from the initial node. In Petri nets, the first action is connected to the *Object* place, see Fig. 5.

We formalize in what follows, the conversion of an action and its internal edges (rules 1 & 2) and we illustrate this by the example of Fig. 6a, extracted from the activity partition modeling the message server system of Fig. 4. In all what follows, the symbol ∀ means *For all* and the symbol ∃ means *create*.

Rule 1: conversion of an action with internal input edges

```
— $\forall\ s_i \in S : \exists\ t_i \in T$.    # internal and external actions
– if $Prec(s_i) = InitialNode \vee In(s_i) = \phi$    # first action
  - $\exists\ Object \to t_1 \in P \times T$
```

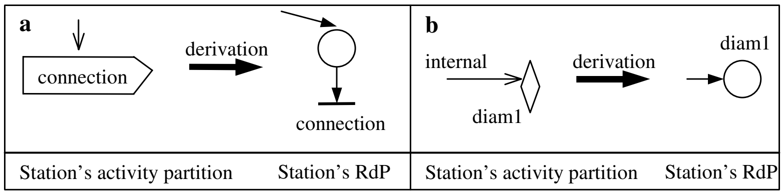

Fig. 6. (a) conversion of an action with internal input edge, (b) conversion of a diamond with internal input edge.

– $\forall\ s_p \in S \cup B$ *such that* $s_p \in Prec(s_i) \wedge s_p \neq InitialNode$: # *intermediate action preceded by an action or bar node*

- $\exists\ p_p \in P$, $\exists\ t_p \rightarrow p_p \in T \times P$, $\exists\ p_p \rightarrow t_i \in P \times T$

– $\forall\ s_p \in D$ *such that* $s_p \in Prec(s_i) \wedge s_p \neq InitialNode$: # *intermediate action preceded by a diamond*

- $\exists\ p_p \rightarrow t_i \in P \times T$

5.2.2. Mapping of the bar nodes

The activity partition supports the modeling of multiple parallel control flows. The synchronization of these flows is realized using a join node. A fork node is used to split a flow into multiple concurrent flows. To deal with the modularity concept, the join bar must belong to only one swimlane even if it synchronizes actions that belong to different actors (swimlanes). The incoming and outgoing edges from the other swimlanes are modeled as external edges.

A join/fork node (also called bar node) is semantically equivalent to an action and thus, it is converted in the same manner, see rules 1 and 2.

5.2.3. Mapping of the diamond nodes

A merge node is a control node that brings together multiple alternate flows. Contrary to an action node, it is not used to synchronize concurrent flows but to accept one among several alternate flows. A decision node is a control node that chooses between outgoing flows.

A merge/decision node (also called diamond node) is semantically equivalent to a place. Its internal input edges (merge node) are converted into arcs, when coming from a bar or action node, see Fig. 6b.

Rule 2: conversion of a diamond with internal input edges

— $\forall\ d_i \in D$: $\exists\ diam_i \in P$ (*of diamond type*)

– $\forall\ s_p \in S \cup B$ *such that* $s_p \in Prec(d_i)$: $\exists\ t_p \rightarrow diam_i \in T \times P$

6. Initialization of the Petri net marking

To simulate the generic model stemmed from the activity partition transformation, two types of arguments must be initialized, namely, the system's objects and the exchanged messages among these objects. The objects are initialized through a technique that we call static initialization while the messages have a dynamic initialization.

6.1. Static initialization

We define for our approach requirements two types of objects: active and passive. The active objects are actors characterized by control and object flows while the passive objects are just manipulated by the actors. For the static initialization, only the active objects, represented by actors on the activity partitions, are concerned.

These objects are formalized by colored tokens of the form: <*obj*, *attrib*> where *obj* designates their identity and *attrib* the set $\{attrib_1, \ldots, attrib_k\}$ of their attribute values. For the object identity, we adopt the UML notation which identifies an object by its name and its class name as follows: *object*: *class*.

The objects and their attribute values are specified on the object diagrams. They initialize Petri net marking by providing the active objects of the *DMs*. Thus, all the objects instantiated from the same class on the object diagram, are inserted in the *Object* place of the *OPN* translating the actor activities.

Fig. 7 shows an example of the object diagram of the message server application before any action (there are

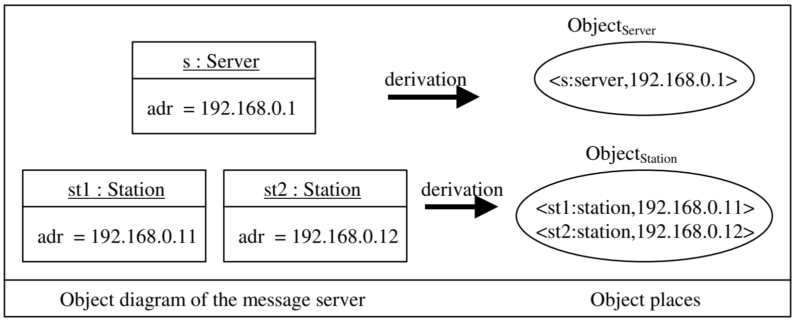

Fig. 7. Object initialization.

no links between the objects). For each station, the IP address is given.

6.2. Dynamic initialization

On the activity partitions, the exchanged messages appear in an implicit form without specification of the attribute values (see Fig. 2). To deal with a value-oriented validation, we propose to use the sequence diagrams to initialize these messages.

6.2.1. Sequence diagram syntax and semantics

The sequence diagram allows the modeling of specific scenarios. It shows exchanged messages among lifelines. The lifelines represent the participants in the interaction where each participant is identified by its name concatenated to the class name as follows: *object:class*. The «*send*» and «*call*» messages are specified with their attribute values, as follows: «*send*» *class*(*attrib*), «*call*» *operation*(*attrib*), see Fig. 8. All these messages are generated from a source object to occur on a target object. We call the generated messages *events* and the occurred ones *triggers*. This specification permits the initialization of the events that are generated on the activity partition.

The sequence diagram of Fig. 8 shows a scenario related to the server message application presented in Section 3. Two stations *St1* and *St2* request a connection from a server *S*. When done, *St1* creates, initializes and then transmits a message *M*1 which is forwarded by *S* to *St2*. After this, both stations are disconnected.

6.2.2. Dynamic initial marking design

On the activity partitions, the exchanged messages appear in an implicit form which is different from that of the sequence diagram messages. Since the latter are used to initialize (activate) the activity partition messages, they must be matched with them, in such a way that each message of the activity partition is assigned to its corresponding one on the sequence diagram. The syntax of the activity partition interactions, presented in Section 2, shows that the correspondence can be easily done. Indeed, on the activity partitions:

- the «send» action is specified using either the stereotype «*send*» and the *class* name of the sent object (see Fig. 2a) or a *pentagon* with the *class* name (see Fig. 2b). This corresponds to the used form in the sequence diagram «*send*» *class*(*attrib*),
- the «call» action is specified using the stereotype «*call*» and the *operation* name of the called operation (see Fig. 2c). This yields the used form on the sequence diagram «*call*» *operation*(*attrib*),
- the «create/destroy» action is specified using the stereotype «*create/destroy*» and the *class* name of the created/destroyed object (see Fig. 2d). This corresponds to the used form on the sequence diagram «*create/destroy*» *class*.

We formalize an interaction on the sequence diagram by the 5_tuple *(i, k, ev, srce, targ, xobj, attrib)*. The components *i* and *k* indicate the scheduling of the exchanged messages, per class and considering all the class instances. *i* gives the event number whereas *k* yields the trigger number. The component *ev* designates the exchanged message expressed without attributes as follows: «*send*» *class*( ), «*call*» *operation*( ), «*create*» *class*( ) *or* «*destroy*» *class*( ). *Srce* and *targ* are, respectively, the source object's identity and the target object's identity. The exchanged object's identity *xobj* is given only for the *send* messages. As for *attrib*, it designates the set $\{\text{attrib}_1, \ldots, \text{attrib}_k\}$ of the exchanged object's attributes if a *send* message or the operation's attributes if a *call* message.

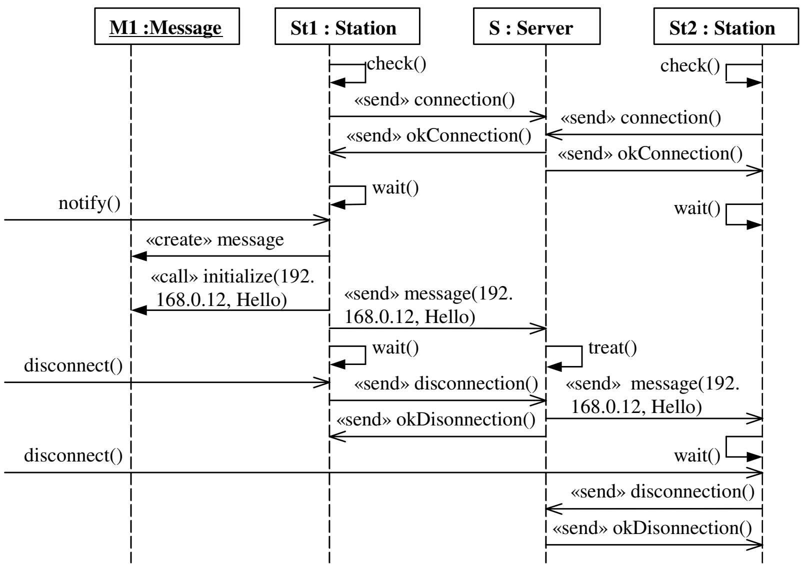

Fig. 8. A scenario from the message server application.

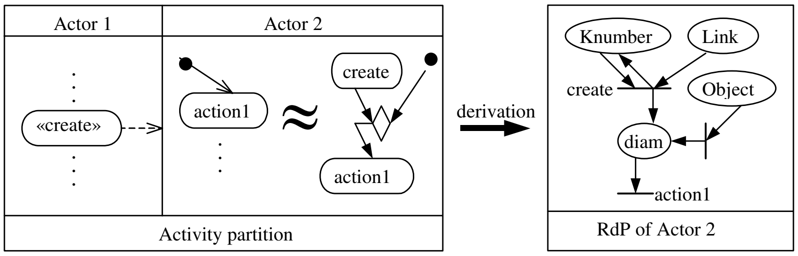

Fig. 9. Derivation of the creation trigger.

One can wonder about the necessity of numbering the events of the sequence diagram when these events can be simply matched with their corresponding actions on the activity partition. In fact, the specification of the event numbers is required when the modeling concerns numerous identified objects (instead of one anonymous object). Indeed, the class objects may evolve differently in time, throughout the dynamic model. So, if multiple objects are candidates for the same action, the latter is enabled for only the right object (whose turn arrives). Furthermore, given the implicit action sequencing on the activity partition, the more explicit event sequencing on the sequence diagram allows to check the sequencing consistency of the two diagrams.

The dynamic model *DM*, derived from an activity partition, is a generic model, specifying the object's overall behavior. Its initialization, derived from the sequence diagram, takes three different forms, depending on the message provenance and the nature of the action which it produces. The messages may be: home messages (generated by the model's objects towards other objects), border messages (generated from the system environment towards the model's objects) or creation/deletion messages (used to create or destroy objects). The latter may be home or border messages.

The home messages are converted to tokens of the form $<i^{(Sc)}, k^{(Sc)}, ev^{(Sc)}, srce^{(Sc)}, targ^{(Sc)}, xobj^{(Sc)}, attrib^{(Sc)}>$ and stored in the *Scenario* place of the *DM* corresponding to their class. Their ordered extraction from the *Scenario* place is provided by the place *Inumber* which gives the following event number. Through this initialization, the *Scenario* place activates the Petri net with the generated events.

Their source object being not represented on the model, all the border messages are directly stored in the *Link* place which is common to all classes. So, they are only numbered as triggers and are converted to tokens of the form: $<k^{(Li)}, ev^{(Li)}, srce^{(Li)}, targ^{(Li)}, xobj^{(Li)}, attrib^{(Li)}>$. Their ordered extraction from the *Link* place is provided by the place *Knumber* which gives the following event number. This initialization permits the opening of the Petri net model using the *Link* place which is defined as an *open* place.

The create/destroy messages are semantically translated to add or remove dynamically, objects from the model. They may be home or border and so assign to the *Scenario* and *Link* places the role of *super* places, as defined in the OPN concept.

An object creation must be the first action of the object's life cycle. It is realized by means of a creation trigger. So, it is semantically equivalent to an action related through a diamond, to the first action of the object's swimlane. The diamond is also related to the *InitialNode* if some objects already exist, see Fig. 9. An object creation does not specify the attribute values. Only the identity of the created object is given.

The object destruction may be implicit, if there is no loop on the object's life cycle, occurring naturally, with the end of the object's activities. It also may be explicit, occurring at any time of the object's actions. The latter is treated and transformed specifically. On the Petri nets, its arrival should be expected at all places, since the object may be at any place when the event arrives. To carry out this, for each place of the *DM*, we add a transition with two input arcs: one coming from the place in question and the other coming from the *Link* place through which the event comes, see Fig. 10. This modeling leads to an non-deterministic firing which is resolved by giving the added transitions a higher priority (using the PROD precedence concept). This assignment enables in priority the object destruction when a *destroy* event occurs.

The transformation of the sequence diagram of Fig. 6 gives an initial marking for the *Link* place, the *Scenario* place related to the Station class and the *Scenario* place for the Server class, as shown below.

$Scenario_{Station}$ = <0, 0, «send» connection( ), st1:station, s:server, connexion,>+
<1, 1, «send» connection( ), st2:station, s:server, connexion,>+
<2, 0, «create» message( ), st1:station, m1:message, ,>+

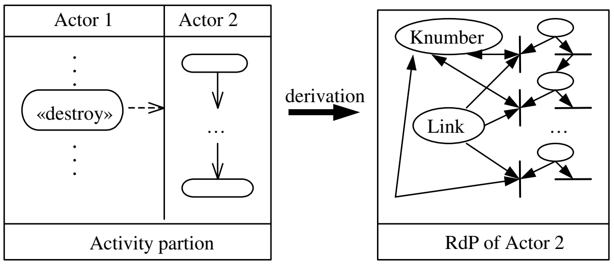

Fig. 10. Derivation of the deletion trigger.

<3, 1, «call» initialize( ), st1:station, m1:message,, 192.168.0.12, Hello>+
<4, 2, «send» message( ), st1:station, s:server, m1:message, 192.168.0.12, Hello>+
<5, 3, «send» disconnection( ), st1:station, s:server, disconnexion,>+
<6, 4, «send» disconnection( ), st2:station, s:server, disconnexion,>
$Scenario_{Server}$ = <0, 0, «send» okConnection( ), s:server, st1:station, okConnexion,>+
<1, 1, «send» okConnection( ), s:server, st2:station, okConnexion,>+
<2, 4, «send» message( ), s:server, st2:station, m1:message, 192.168.0.12, Hello>+
<3, 5, «send» okDisconnection( ), s:server, st1:station, okDisconnexion,>+
<4, 7, «send» okDisconnection( ), s:server, st2:station, okDisconnexion,>
Link = <2, «send» notify( ),, st1:station,,>+<3, «send» disconnect( ),, st1:station,,>+
<6, «send» disconnect( ),, st2:station,,>

We note that the value of *k* goes from 1 to 4 for the tokens of the Server's *Scenario* place. This can be commented as follows. For all these tokens, the field *k* designates the reception order within the *Station* class, considering the whole of its objects. These objects receive events from the server but also from the environment. So, the sequencing is established, taking into account both sources. For the tokens of the *Link* place, the identity of the source objects and sent signals are unknown.

7. Activation of the Petri net specification

The marking regarding active objects, progresses only on the dynamic model *DM*. It evolves through the *DM*'s places and transitions according to the following rule:

$$\forall p_i, p_j \in P, \forall t_i \in T, \quad Pre(p_i, t_i) = Post(p_j, t_i) = < obj, attrib >$$

The marking translating exchanged messages, takes other ways. It moves between the *DM* model and the *Scenario* and *Link* places. Since external actions are modeled in Petri nets by means of transitions, their firing requires their prior activation with the corresponding events.

7.1. Identification of the interactions on the dynamic model DM

7.1.1. Enabling the external output actions

Activation of the external output actions is performed with the *Scenario* place events. Each action activation implies a consumption of a *Scenario*'s token, controlled by means of the *Inumber* place. The token is after taht, forwarded to the *Link* place. This forwarding translates the external edge outputting the action. This mapping is formalized by rule 3 and illustrated by Fig. 11a.

Rule 3: Enabling the external output actions

— ∀ $s_i \in S_{out}$:
– ∃ *Scenario* → $t_i \in P \times T$ *such that Pre*(*Scenario*, t_i) = <$i^{(Sc)}$,$k^{(Sc)}$, $ev^{(Sc)}$, $srce^{(Sc)}$, $targ^{(Sc)}$, $xobj^{(Sc)}$, $attrib^{(Sc)}$>,
– ∃ *Inumber* → $t_i \in P \times T$ *and* t_i→*Inumber* ∈ $T \times P$ *such that Pre*(*Inumber*, t_i) = <i> *and Post*(*Inumber*, t_i) = <i+1>,
– ∃ t_i→*Link* ∈ $T \times P$ *such that Post*(*Link*, t_i) = <$k^{(Sc)}$, $ev^{(Sc)}$, $srce^{(Sc)}$, $targ^{(Sc)}$, $xobj^{(Sc)}$, $attrib^{(Sc)}$>.

7.1.2. Enabling the external input actions

The activation of an external input action is realized with events coming from the *Link* place. This implies a consumption of a *Link*'s token, iterated by means of the *Knumber* place. This activation translates external edges that may be inputs to the action. Rule 4 and Fig. 11b show the marking of this activation that concerns all external input actions excluding those outputting bar or diamond nodes whose a specific treatment is given in section d).

Rule 4: Enabling the external input actions

— ∀ $s_i \in S_{in}$:
– ∃ *Link*→$t_i \in P \times T$ *such that Pre*(*Link*, t_i) = <$k^{(Li)}$, $ev^{(Li)}$, $srce^{(Li)}$, $targ^{(Li)}$ $xobj^{(Li)}$, $attrib^{(Li)}$>,
– ∃ *Knumber*→$t_i \in P \times T$ *and* t_i→*Knumber* ∈ $T \times P$ *such that Pre*(*Knumber*, t_i) = <*k*> *and Post*(*Knumber*, t_i) = <k+1>.

7.1.3. Conversion of external edges of bar/diamond nodes

External input edges into a bar node are converted to an arc coming from the *Link* place towards the bar transition, see Fig. 8a. External input edges into a diamond node are converted to an arc-transition-arc triplet, coming from the

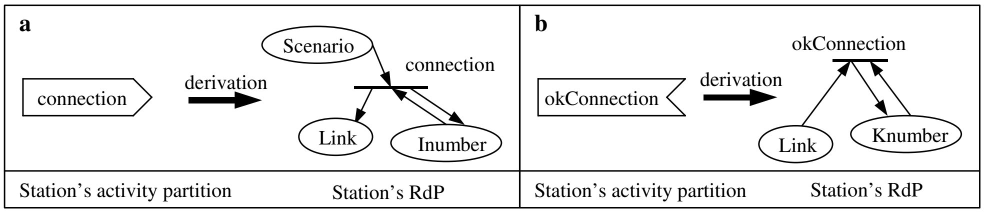

Fig. 11. (a) Activation of an external output action. (b) Activation of an external input action.

Link place towards the diamond place, see Fig. 8b. These two conversions are iterated by means of the *Knumber* place. The conversion of the bar/node external output edges is included in the activation of their external output actions (which are obligatory at the entry of the node, see Fig. 9).

7.1.4. Enabling the external input actions outputting a bar/diamond node

External actions from a bar/diamond node are obligatory external input actions. For these actions, the *Link*'s event consumption is produced by the external input edges into the bar/diamond transition (see section c). So, the event must be after that, forwarded to the external input action in order to be related to its action, see Fig. 12. This may be useful for some treatments on the object's signature, see Section 9.2. This activation is specified for an external input action at the output of a bar node by rule 5.

Rule 5: Enabling the external input actions outputting a bar node

— $\forall b_i \in B$: $\forall s_i \in Suc(b)$ such that $s_i \in S_{in}$:
$-Post(p_i, bar) = <obj, attrib>+<ev, srce^{(Li)}, targ^{(Li)}, xobj^{(Li)}, attrib^{(Li)}>$,
$-Pre(p_i, t_i) = <obj, attrib>+<ev, srce^{(Li)}, targ^{(Li)}, xobj^{(Li)}, attrib^{(Li)}>$.

7.2. Transition firing

At the transition firing, the incidence functions *Pre*() and *Post*() respectively, consume or put tokens in the corresponding places. These tokens represent objects or events. The events whose global form is $<[i^{(Sc/Li)}], k^{(Sc/Li)}, ev^{(Sc/Li)}, k^{(Sc/Li)}, srce^{(Sc/Li)}, targ^{(Sc/Li)}, xobj^{(Sc/Li)}, attrib^{(Sc/Li)}>$ are extracted from the *Scenario* or *Link* places. In the Scenario place, they are selected on the basis of two components ; first, their number $i^{(Sc)}$ whose value is given by the token $<i>$ of the *Inumber* place (whose role is to iterate the sequencing). Second, the event $ev^{(Sc)}$must be equal to the event *ev* identified on the activity partition. To ensure this, the event tokens extracted by the *Pre(Scenario, t)* function, are written in the form $<i, k^{(Sc)}, ev, srce^{(Sc)}, targ^{(Sc)}, xobj^{(Sc)}, attrib^{(Sc)}>$.

In the Link place, the tokens are selected according to the same considerations as those of the *Scenario* place: their number $k^{(Li)}$ whose value is given by the token $<k>$ of the *Knumber* place and the event $ev^{(Li)}$ which must be equal to the trigger *trg* identified on the activity partition. So, the event tokens consumed by the *Pre(Link, t)* function are of the form $<k, trg, srce^{(Li)}, targ^{(Li)}, xobj^{(Li)}, attrib^{(Li)}>$.

The objects are of the form $<obj, attrib>$. To determine which object should be extracted from the previous place, different situations may arise depending on the action type that may be: internal action, send/call action or receipt/called action.

When the action is internal, it is not constrained by external events or triggers and so all the objects can perform it. In that case, if more than one action precedes it (with an internal incoming edge), the same object must have performed all these actions (case of parallel actions) so that the transition can be fired. So, for all these actions, we set: $Pre(p_i, t) = <obj, attrib>$, $i = 1, card(Prec(s))$ and $Post(p_j, t) = <obj, attrib>$ $j = 1, card(Suc(s))$.

For the send/call actions, the extracted object is the one generating the event. It is defined by the $srce^{(Sc)}$ component of the event token extracted from the *Scenario* place, as follows: $Pre(p_i, t) = Post(p_j, t) = <scre^{(Sc)}, attrib>$.

As for the receipt/called actions, the concerned object is the one receiving the event occurrence. It is defined by the $targ^{(In)}$ component of the event token extracted from the *Link* place, as follows: $Pre(p_i, t) = Post(p_j, t) = <targ^{(Li)}, attrib>$.

8. Analysis and verification

The verification by model checking as treated in PROD, is based on the state space generation and the verification of safety and liveness properties of a system on this space. The properties may be basic, about the correctness of the model construction or specific, written by the modeler to ensure the faithfulness of the system modeling. For each of these approaches, given a property, a positive or negative reply is obtained. If the property is not satisfied, it generates a trace showing a case where it is not verified.

8.1. Basic property verification

The basic properties are verified according to two ways: the on-the-fly tester approach and the reachability graph inspection approach. The on-the-fly verification means that the property is verified during the state space generation

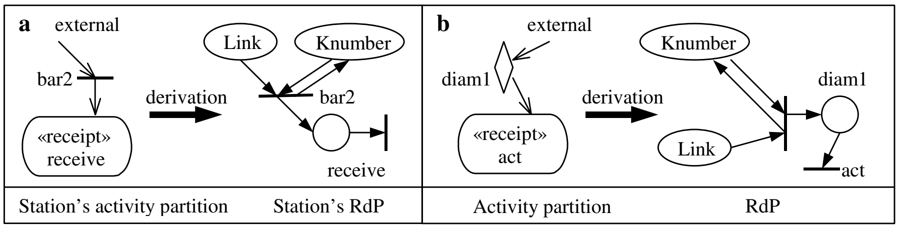

Fig. 12. (a) Conversion of an external input edge into a bar node. (b) Conversion of an external input edge into a diamond node.

which is automatically stopped as soon as the property fails. On the contrary, the traditional approach proceeds to a property verification on the reachability graph, after its generation.

The on-the-fly tester approach detects deadlock, livelock and reject states. The *deadlock* verification ensures that there exists no reachable marking where no transition is enabled. In other words, that means that there are no UML states that prevent any activity to be invoked eventually. The *livelock* detection informs about loops of actions on the graph. As for the *reject* state checking, it detects undesirable markings in critical places which is equivalent in UML, to rejecting undesirable objects at given states. These three properties are systematically verified, without intervention of the designer.

The reachability inspection approach permits the verification of some other properties such as quasi-liveness, boundedness or reinitializability. These properties are automatically verified.

The quasi-liveness property is weaker than the deadlock. It guarantees that each transition is enabled at least once, i.e., each UML activity can be invoked eventually.

The boundedness is formulated to require an upper bound to the number of objects that can be present, in a state at a given time or in an association end, according to the UML specification using the multiplicity construct. Since objects correspond to tokens, this property upper-bounds the number of tokens.

The reinitializability property checks the possibility for a system of restarting from any state, i.e., the initial marking should be reachable from any marking of the net's. This is realized by a systematic computation of the net's strongly connected components which must be equal to 1, so that the net is reinitializable.

8.2. Value-oriented property validation

For a more precise validation, the specific properties of the system can be written by the designer in Linear temporal Logic (LTL) or Computational Tree Logic (CTL) and then, verified by PROD. Since the main motivation of this work is that the UML designer may reach valid models without needs for knowledge of formal techniques, it is only reasonable that the properties are expressed by the modeler in the OCL language and afterwards, are automatically translated into LTL and CTL logics. OCL which is a part of UML for the expression of constraints over UML models, in particular invariants, is appropriate to data value handling but does not support the expression of temporal properties. For formalization purposes, it is rather, suitable for a translation to first-order predicate logic. Beckert et al. tackle this translation in [7]. So, to deal with an appropriate translation of OCL to the temporal logic which is the supported logic by the model checker tool, we first propose to extend OCL with temporal operators and then to translate it to LTL and CTL. This work is presented in [9]. Other works like those of Distefano [12] and Flake [18] have also invested this research direction.

OCL is mainly based on collection handling in order to specify object invariants. As these collections correspond to association ends, the latter must appear on Petri net specification so that the translated LTL and CTL properties (whose expression is essentially made of these constructs), can be verified. This lead us to the necessity of introducing the association end modeling onto the activity partition in order to get after transformation, the equivalent Petri net constructs. This modeling is realized by means of the link actions.

However, the usefulness of the link actions does not concern explicitly the modeling of the object life cycle. When constructing his diagrams, the designer does not necessarily think on modeling these concepts which are rather specific to the link and end object updates. For example, for connecting a station to the server, the connection request and connection confirmation actions are naturally and systematically modeled by the designer, but the addition of the connected station to the association end *connectedStation* is usually omitted from the modeling, see Figs. 3 and 13. That is why we recommend to the designer to specify the link actions on the activity partition so that the OCL invariants can be verified. But, we release him from this modeling on the sequence diagram and take in charge the treatments related to the initialization of these actions.

For this work, we are particularly interested in the create link (*CreateLink*), destroy link (*DestroyLink*) and clear association (*ClearAssociation*) actions. These three actions were defined in UML action semantics elaborated in [32] for model execution and transformation. The create link action permits to add a new end object in the association end. The destroy link action removes an end object from the association end. The clear association action destroys all links of an association in which a particular object participates. All these actions output from the activity causing the association end update which is necessarily an external action (see Fig. 13). They can not input other actions (they have not output edges) and are written in the form: *linkAction(associationEnd)*.

On Fig. 13, after it has confirmed its connection (*okconnection*), the station adds itself in the association end *connectedStation*, using the «*createLink*»(*connectedStation*). It adds a notified or received message with «*createLink*» (*transmittedMessage*) or «*createLink*» (*receivedMessage*), respectively. Finally, after reception of a disconnection confirmation, the station removes itself from the association end *connectedStation*, using «*destroyLink*»(*connectedStation*) and clears its association ends with the *Message* class, using «*clearAssociation*» (*transmittedMessage*) and «*clearAssociation*» (*receivedMessage*).

The link actions may concern an active (actor) or passive (exchanged) end object. The object-oriented approach, on which both UML and Object Petri nets rely, is based on modularity and encapsulation principles. To deal with

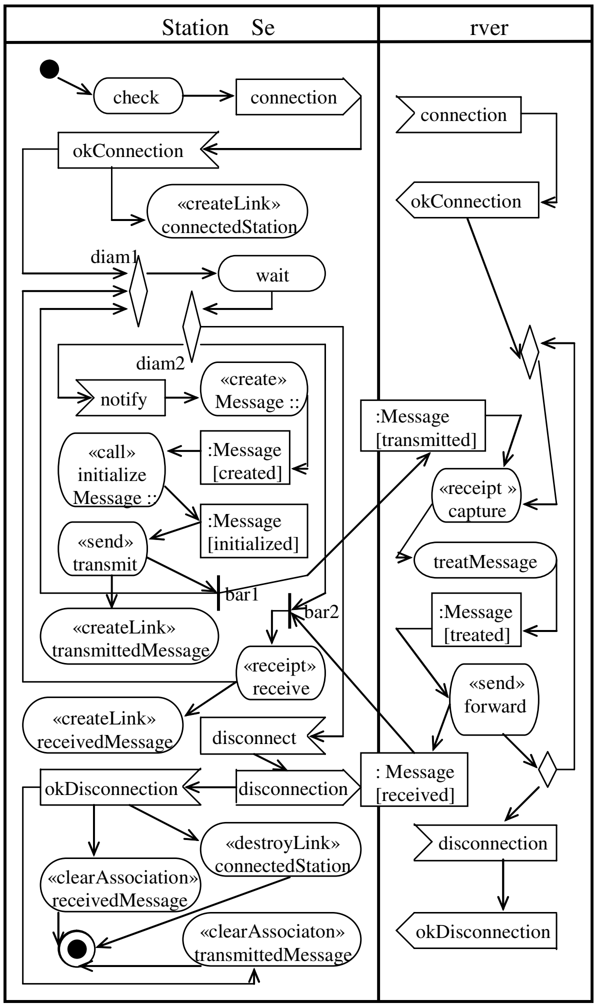

Fig. 13. Activity partitions of the message server with link actions.

modularity, a given association end should appear and be manipulated in only one swimlane. In Petri nets, the association end is translated into a place of *role* type. This place holds the name of the association end and belongs to the *DM* representing a swimlane.

Furthermore, an association end grouping active objects must be updated within the actor's swimlane, in order to comply with the **encapsulation** concept. Indeed, since the end object is saved in the *role* place with its attributes, these attributes must be accessible when updating the association end. The exchanged objects are usually manipulated by the interactive objects (actors) and are not specified by dynamic models. So, the association end representing them could be updated in the swimlane of the class that is at the opposite end. For exchanged objects, the encapsulation constraint is lifted given that the exchanged object's attributes are transmitted within the message and so accessible by the actors.

Considering the new treatments introduced on the activity partitions, we propose to complete the syntax of the latter, formalized by the tuple <*Node*, *Edge*, *Object*>, enriching the *Object* node with the association ends and link actions as follows, *Object* = <*O*, *U*, *R*, *r*, *createLink*, *destroyLink*, *clearAssociation*>, where:

- $O = \{o_1, o_2, \ldots, o_n\}$ is a set of active objects.
- $U = \{u_1, u_2, \ldots, u_n\}$ is a set of exchanged objects.
- $R = \{r_1, r_2, \ldots, r_n\}$ is a set of association ends.
- $r_i^{(k)} = \{y_1^{(i)}, y_2^{(i)}, \ldots, y_k^{(i)}\}$ is a set of objects of the association end r$_i$, $y_k^{(i)} \in O \vee y_k^{(i)} \in U$.
- *createLink*: $R \rightarrow R$ is a function that inserts an object in an association end such that *createLink* $(r_i^{(k)}) = r_i^{(k+1)} = \{y_1^{(i)}, y_2^{(i)}, \ldots, y_k^{(i)}, y_{k+1}^{(i)}\}, y_k \in O \vee y_k \in U$.
- *destroyLink*: $R \rightarrow R$ is a function that removes an object from an association end such that *destroyLink*$(r_i^{(k)}) = r_i^{(k-1)} = \{y_1^{(i)}, y_2^{(i)}, \ldots, y_{k-1}^{(i)}\}, y_k \in O \vee y_k \in U$.
- *clearAssociation* : $R \rightarrow R$ is a function that suppresses all objects of an association end r_i such that $clearAssociation(r_i^{(k)}) = r_i^{(0)} = \{\}$.

In Petri nets, the create link action is semantically equivalent to an arc connecting the transition causing the association end update to the place specifying the association end. This is formalized by rule 7 and illustrated by Fig. 14a. The destroy link action is semantically equivalent to an arc leaving the association end place towards the transition corresponding to the action provoking the update, see rule 8.

The object to be added to/removed from the association end is extracted from the components of the token (whose global form is <$[i^{(Sc/Li)}]$, $k^{(Sc/Li)}$, $ev^{(Sc/Li)}$, $srce^{(Sc/Li)}$, $targ^{(Sc/Li)}$, $xobj^{(Sc/Li)}$, $attrib^{(Sc/Li)}$>) corresponding to the message that provokes the association end update. This token is situated in the *Scenario* place if the message is generated (send/call action). It is located in the *Link* place

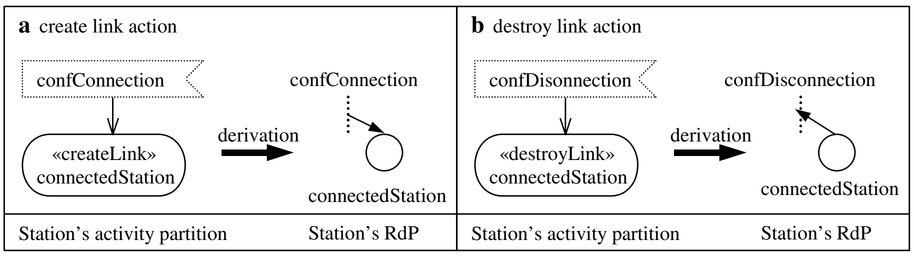

Fig. 14. Conversion of (a) create link section (b) destroy link section after an external input action.

if the message occurs (receipt/called action). The added/removed object may be the source object ($srce^{(Sc)}$) or the exchanged object ($xobj^{(Sc)}$) if the link action outputs a send/call action. It is the target object ($targ^{(Li)}$) or the exchanged object ($xobj^{(Li)}$) if the link action outputs a receipt/called action.

In Petri nets, the association end objects are colored tokens of *role* type. They are of the form <*assoc*, *obj*, *attrib*>, where *obj* is the object to be added to or removed from the association end and *assoc* is the object at the opposite end.

Rule 6: conversion of an association end *r*

— $\forall\ r_i \in R$, (rol_i is of role type).

Rule 7: conversion of a create link action

– $\forall\ createLink_i(r_i)$ *such that* $r_i \in R$: $\exists\ t_p \to rol_i \in T \times P$, *such that*:
# a link action after an external output action
– *if* $Prec(createLink) \in S_{out}$*and* $r_i^{(k)} \subset$ O: $\mathrm{Post}(\mathrm{rol}_i, t_p) = <\mathrm{targ}^{(Sc)}, \mathrm{srce}^{(Sc)}, \mathrm{attrib}>$,
– *if* $Prec(createLink) \in S_{out}$*and* $r_i^{(k)} \subset$ U: $\mathrm{Post}(\mathrm{rol}_i, t_p) = <srce^{(Sc)}, xobj^{(Sc)}, attrib^{(Sc)}>$
# a link action after an external input action
if $Prec(createLink) \in S_{in}$*and* $r_i^{(k)} \subset$ O: $\mathrm{Post}(\mathrm{rol}_i, t_p) = <\mathrm{srce}^{(Li)}, targ^{(Li)}, attrib>$
if $Prec(createLink) \in S_{in}$*and* $r_i^{(k)} \subset$ *U*: $Post(rol_i, t_p) = <targ^{(Li)}, xobj^{(Li)}, attrib^{(Li)}>$.

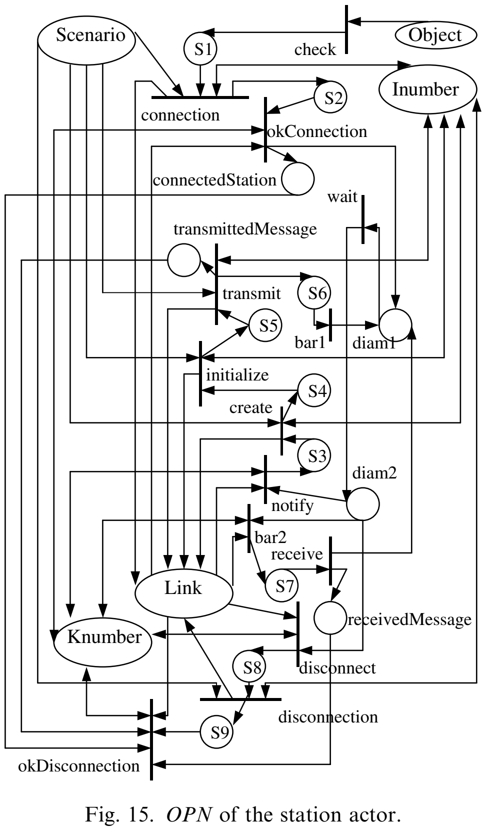

Fig. 15. *OPN* of the station actor.

An example on the translation of the create link action is presented on Fig. 14a. Two other examples are shown on the case study, see Figs. 14 and 15. The dashed symbols designate the previous constructs.

Rule 8: conversion of a destroy link action

The *DestroyLink* action is converted in a similar manner as the *CreateLink* action, except that:

- the arc incoming into the role place is replaced by an arc outgoing this place and,
- the *Post*( ) function is replaced by the *Pre*( ) function holding exactly the same tokens, see Fig. 14b.

The *ClearAssociation* is semantically equivalent to an arc from the association end place to the transition corresponding to the connected action, removing all the association end objects associated to a given object. This conversion is specified by rule 9 and illustrated by the activity partitions and *OPN* of Figs. 13 and 15.

Rule 9: conversion of a clear association action

– $\forall\ clearAssociation(r_i)$ *such that* $r_i \in R$ *and* $Prec(clearAssociation) \in S_{out}$: $\exists\ r_i \to t_p \in P \times T$, *such that*: $Pre(r_i, t_p) = <srce^{(Sc)}, xobj, attrib>$ *# link action after an external output action*
– $\forall\ clearAssociation_i(r_i)$ *such that* $r_i \in R$, $Prec(clearAssociation) \in S_{in}$: $\exists\ r_i \to t_p \in P \times T$, *such that*: $Pre(r_i, t_p) = <targ^{(Li)}, xobj, attrib>$ *# link action after an external input action*

8.3. Model verification

To test the practical implementation of our derivation approach, we built a translator whose semantic functions are drawn from the conversion rules we have set. We also developed a graphic interface for the construction of the activity partitions, class, object and sequence diagrams. These diagrams constitute the input of the translator whose outputs result into predicate/transition nets, specified in PROD syntax.

The application of the conversion rules to the station actor of the activity partitions of Fig. 13, gives the *OPN* of Fig. 15. On the latter, each transition is derived from an UML action of which it takes the name. To help the reader to understand the translation process, in Table 1, each UML node is matched with the conversion rules that are applied to it. So, to construct the *OPN*, one has only to follow the object life cycle of the station swimlane and to apply to each of its nodes the corresponding rules given by Table 1.

On Table 1, «CR» abbreviates «*CreateLink*», «DS» abbreviates “*DestroyLink*» and «CL» abbreviates «*Clear Association*».

An example of Petri net specification in PROD syntax is given in what follows, for the *check* node.

Table 1
Applied conversion rules on the station swimlane nodes

UML nodes	Rules
check	rule 1
connection	rule 1, rule 3
okConnexion	rule 1, rule 4
«CR» connectedStation	rule 6, rule 7
merge diamond - diam1-	rule 2
wait	rule 1
decision diamond -diam2-	rule 2
notify	rule 1, rule 4
create	rule 1, rule 3
initialize	rule 1, rule 3
transmit	rule 1, rule 3
fork bar - bar1 -	rule 1
«CR» transmittedMessage	rule 6, rule 7
join bar – bar2 -	rule 1
receive	rule 1, rule 5
«CR» receivedMessage	rule 6, rule 7
disconnect	rule 1, rule 4
disconection	rule 1, rule 3
okDisconnection	rule 1, rule 4
«DS» connectedStation	rule 8
«CL» transmittedMessage	rule 9
«CL» receivedMessage	rule 9

#place Object mk(<.st1_station,192_168_0_11.>+<.st2_station, 192_168_0_12.>)
#place S1
#trans check
in {Object:<.obj,attrib.>;}
out {s1:<.obj,attrib.>;}
#endtr

Note:

- In *Object* place declaration:
  - 'mk' designates the initial marking,
  - the symbol ':' of the object identity is not supported by PROD syntax. It is replaced by '_',
  - the symbol '.' of the IP address is not supported by PROD syntax. It is replaced by '_',
- For the transition *check*, the places that follow the keyword *in* correspond to the places p_i specified in the signature of the functions *Pre*(p_i, *check*). Likewise, the places that follow the keyword *on* correspond to the places specified in the signature of the functions *Post*(p_i, *check*) for the mentioned transition.

Afterwards, PROD was executed to verify the Petri net specification. The basic properties defined above, were first checked and then, some system's invariants were expressed in LTL properties and verified. Two of these properties are expressed below into a paraphrased (textual) form and then specified as OCL invariants and translated into LTL properties. To make easier the comprehension of these properties, refer to the class diagram of the server message application (Fig. 3).

Property 1
The number of connected stations is limited to maxStation.
Property 1 in OCL
context s:Server *inv*: s.connectedStation → *size* <= S.maxStation
Property 1 in PROD
For each place of the *DM** of a server s, write the property:
# verify henceforth (card(connectedStation : *field*[*0*] == s) <= (place$_{DM*server}$: *field*[*2*]))
where: - *connectedStation: field[0]* designates the first component (*assoc*) of the *connectedStation*'s tokens,
-*place*$_{DM*server}$: *field*[*2*] is the third component (*maxStation*) of the tokens of *DM** of the server,
-*DM** designates a *DM* excluding the places of role type.

Property 2
Only connected stations can transmit messages.
Property 2 expression in OCL
Context t:Station *inv*:
Server.*allinstances*() → *forAll*(s.connectedStation → *excludes*(t) *implies* t.transmittedMessage → *isEmpty*())
Property 2 expression in PROD
For each station t of a server s write the property:
#verify henceforth (connectedStation: (*field*[*0*] == s & & *field*[1] == t) == *empty*
implies (transmittedMessage: *field[*0] == t)== *empty)*
where: - *connectedStation: field[0]* designates the first component (*assoc*) of the *connectedStation*'s tokens,
-*field[1]* designates the second component (*obj*) of the *connectedStation*'s tokens,
-*transmittedMessage: field[0]* is the first component (*assoc*) of the *transmittedMessage*'s tokens.

9. Discussion and related work

9.1. On the formalization of the activity partitions

Currently, UML activity diagrams are widely used for the workflow specification [13] but in an informal way. The most recent and significant research works on the formalization of UML 1.x activity diagrams are those of Eshuis [17] and Delatour [11]. Through our multiple investigations, it seems that not more research work on the formalization of UML 2.0 activity partitions which is an extension of UML 1.x activity diagrams for workflows, has been undertaken yet. However, some works already exist, on the mapping of UML 2.0 diagrams into executable languages [37].

Delatour [11] presents a methodology for analysis and development of real-time systems. This methodology transforms into Petri nets the UML 1.x interaction and activity diagrams. The derivation process evolves interactively with the developer who introduces, when needed, more precision on his models. However, Delatour notices that his interest is limited to the activity diagrams modeling the dynamics of only one object. He does not deal with the workflows that are more complex to formalize.

Eshuis does it, but not by mapping into Petri nets. He compares the semantics of Petri nets and UML 1.x activity diagrams for workflow modeling [15]. This study allows him to observe that the main difference between them concerns the system reactivity which is a relevant characteristic of workflows. By system reactivity he means the reaction of a system faced with trigger events occurring from the environment. On the other hand, to be able to interact with its environment, a reactive system must be defined as being open. So, Eshuis states that Petri nets model the resource usage of active systems that are closed and non-reactive, whereas UML activity diagrams support the modeling of open, reactive systems. Based on this, he enriches UML activity diagrams with a formal semantics [17] for specifying workflows and develops a tool [14] to verify the specification by means of model checking. The diagram to be verified is thus translated to a transition system to be input in the model checker.

Even though Eshuis argues for the relevance of his approach by referring to the Petri nets non-reactivity, he later admits in [16] that Petri nets can be reactive only by forcing the firing of the reactive transitions. Moreover, many works [3,21] propose theoretical approaches to model workflows using Open nets.

Our approach comes in complement to Delatour's methodology extending it to the control flow of multiple interactive objects. It is also proposed against Eshuis's approach, proposing practical solutions to model open systems using Petri nets. We provide this reactivity using Open Petri nets through the object-oriented structure of the Petri net model that we propose in Fig. 5. Indeed, the super places *Scenario* and *Link* are also open places generating the environment's events which are modeled and captured from the sequence diagram.

9.2. On the use of OPNs to formalize the UML dynamic models

A few works have already tackled the use of OPNs to transform the statecharts and then to connect them using the collaboration diagrams [4,5,22,35]. This seems natural since the statechart models the object life cycle showing the trigger/sent events and the collaboration diagram connects the interactive objects by means of a static interconnection structure emphasizing the message exchange (events). This interconnection provides a generic model for the object communication supporting the whole of the scenarios. However, Baresi's approach [4,5] is restricted to some preliminary hints on ascribing formal semantics to UML statecharts through OPNs and its translation process needs to be developed.

As for Shatz et al.'s method [22,35] in contrast to us, their study is not value-oriented. They do not focus on the initialization process of the generated Petri nets. Their approach does not separate between the object dynamics which may be generic and its initialization with real data. It only allows the simulation of anonymous objects and events. We fill this gap using the *Scenario* place which defines the generated events regarding a specific scenario. These events permit to initialize those specified on the state diagram. However, Shatz et al. emphasize the development of the interface allowing the communication between the object classes whose dynamics is formalized by means of the *DMs*. This interface includes three places used to manage the message exchange. We judge this interface superfluous. We consider that the modularity is completely realized by means of the *DMs* and we choose to do without this interface, connecting the *DM*s directly through the *Link* place.

In the same area of research, we formalized in a previous work, published in [10], the UML statecharts using colored Petri nets. Colored Petri nets proved to be suitable for representing data on Petri nets while preserving a reduced size of the models. However, they fail when tackling the object-oriented modeling, in particular, encapsulation and instantiation concepts. To overcome these limits, the OPNs reveal to be more appropriate. Indeed, the statechart presents in essence, a natural modularity when modeling the class object life cycle as well as the class object interactions. This modularity is preserved by the OPN derived from a statechart. The OPN translates the class methods and encapsulates the class instances and their attribute values. As for the connection of the whole of the OPNs derived from the different statecharts, it is deduced from the collaboration diagram which represents the structure of the interactive classes. Conversely to this previous work, the activity partitions, subject of the current study, present a modularity which is less apparent, in particular about the class object interactions. This interaction does not rely on explicit exchanged messages. It is rather, expressed implicitly, by means of directed edges and then, necessitates a specific treatment on the activity partitions to identify the exchanged messages. On the other hand, as the activity partitions regroup all the interactive classes, the use of an interaction diagram for connecting the class dynamic models, becomes unnecessary.

9.3. On the initialization of large-scale systems

The partitioning of the activity diagrams organizes the workflow among the cooperative actors. It allows the modularization of the system activities per actor. For large-scale systems, this modularity simplifies and clarifies the dynamic model representation. On the other hand, the object-oriented approach that we propose, to formalize this modeling, provides an interconnection architecture of the derived Petri nets, which complies with this modularity, see Fig. 5. Indeed, the proposed derivation approach is also modular, in the sense that each swimlane activities are transformed separately, into a *DM* model which communicates with the other *DM*s through the *Link* place. Thus, the use of the activity partition diagram crowned with the object-oriented modular formalization approach that we formulate, prove to be appropriate for specifying large-scale systems by means of generic models.

Nevertheless, the initialization of these models with identified objects necessitates on the one hand, that the classes have finite domains, in other words, instances that can be represented on the object and sequence diagrams. On the other hand, a large number of class instances might explode the state space during the validation process which is performed by model checking. To overcome this problem, we propose for a future work, to start the simulation at a critical moment from the object life cycle and not obligatorily from the initial state. For this purpose, the object diagram will be used to represent the system's objects at this moment. This representation will influence the token distribution at the Petri nets initial marking.

For the present work, we only suggest to the UML designer, when performing a model validation, to proceed to a pertinent and appropriate choice of the initialization scenarios, composed from the object and sequence diagrams. This choice should allow to treat the most important and critical situations of the system. This solution remains better than the one performing the model validation with anonymous objects. It first, permits the model validation in a concrete context, although reduced, considering objects with real values. Second, it allows a more precise validation by means of the OCL invariants which require values for their validation.

9.4. On the validation of the OCL invariants

Regarding the validation strategy, no previous works tackle the integration of the association end specification within the UML dynamic models. We can explain this, arguing that the UML/OCL association is rarely used to formally validate the UML models. When done, it is limited to a subset of OCL [40] excluding the association end handling which yields the most important expressions. Generally, the formalized UML models are rather coupled with appropriate formalisms for the expression of the system properties. So, when the OCL invariants express constraints on the association ends, although correctly specified and translated to LTL or CTL properties, they could never be validated on the derived Petri nets without association end specification on the activity partitions. This specification made by means of the link actions, provides after its transformation to Petri nets, a formal basis for the validation of the translated invariants.

10. Conclusion

This paper introduces a methodology that allows the UML designer to systematically verify and validate his workflow modeling without needs for knowledge of formal checking techniques. The approach presents a technique to transform the UML activity partitions into generic RdPs which are specialized using the sequence and object diagrams. To overcome the complexity of the workflow mapping for large-scale systems, we propose to transform the activity partitions to OPNs in order to take advantages of the modularization mechanisms. Thus, through this approach, a precise dynamic semantics for the activity partition diagrams is provided preserving all UML expressiveness. This is useful, first, for allowing the formal validation of the specification, reusing existing tools and second, for reducing the level of ambiguity which the specification inherits from the UML language.

Unlike previous efforts in this area, which formalize UML dynamic models considering anonymous objects, the methodology which we propose considers identified objects. It offers to the user the opportunity of carrying out a value-oriented validation of his modeling by checking the dynamics of objects identified by identities and attribute values through specific scenarios. This involves a complementary use of the object and sequence diagrams to initialize the activity partitions – which model the object dynamics. This combination has never been tackled by previous work. It implies a specific treatment on the activity partitions to match the communication held among the activity partition actors with the messages exchanged between the objects of the sequence diagram.

The verification concerns both the correctness of the model construction and the faithfulness of the modeling. The latter is allowed thanks to the system awaited properties which are expressed by the modeler in OCL language and then translated to temporal logic properties. The formal validation of these properties required the integration of the association end specification within the activity partitions.

The interaction overview diagram is a variant of the activity diagram. It defines interactions in a way that promotes overview of the control flow [29]. In this diagram different sequences are included in an activity flow in order to show a workflow through the sequences. This appears closely related to the approach we have proposed to merge the sequence diagram with the activity partitions in order to deal with a workflow specification over concrete objects interacting using valuated messages. So, we suggest for a future work, to adapt the derivation rules we have set for the present work, to transform the overview interaction diagram.

Among the other prospects of this work, the analysis of the verification results and their feedback to the user are explored. These results must be presented to the designer in an interpreted form, where the error in modeling is simply and clearly pointed out. Since the methodology calls for UML designer to provide the input specifications, it is only reasonable for the output results to be meaningful to that user.

References

[1] N.R. Adam, V. Atluti, W. Huang, Modeling and Analysis of Workflows Using Petri Nets, Journal of Intelligent Information Systems, Kluwer Academic Publishers, Boston, 1998, pp. 131–158.

[2] N. Amálio, F. Polack, Comparison of formalization approaches of UML class constructs in Z and Object-Z, in: Proc. Int. Conf. of Z and B Users, Lecture Notes in Computer Science, vol. 2561, 2003.

[3] P. Baldan, A. Corradini, H. Ehrig, R. Heckel, Compositional modeling of reactive systems using open nets, in: Proc. CONCUR'01 Conf.Lecture Notes in Computer Science, vol. 2154, Springer, 2001, pp. 502–518.

[4] L. Baresi, Some preliminary hints on formalizing UML with Object Petri Nets, in: Proc. 6th World Conference on Integrated Design and Process Technology, Pasadena (USA), June 2002.

[5] L. Baresi, M. Pezzè, On formalizing UML with high-level Petri Nets, concurrent object-oriented programming and Petri Nets, in: Advances in Petri NetsLecture Notes in Computer Science, vol. 2001, Springer, 2001, pp. 276–304.

[6] M.E. Beato, M. Barrio-Solorzano, C.E. Cuesta, UML Automatic verification tool (TABU), in: Proc. Specification and Verification of Component-Based Systems (SAVCBS 04), California, USA, 2004.

[7] B. Beckert, U. Keller, P. Schmitt, Translating the object constraints language into first-order predicate logic, in: Proc. Verify, Workshop at Federated Logic Conferences, Copenhagen, 2002.

[8] S. Bernardi, S. Donatelli, J. Merseguer, From UML Sequence Diagrams and Statecharts to analysable Petri Net models, in: Proc. Third Int. Workshop on Software and Performance, ACM Press, Italy, 2002.

[9] T. Bouabana-Tebibel, Formal validation with OCL, in: Proc. 2006 IEEE International Conference on Systems, Man and Cybernetics, Taipei, Taiwan, October 2006.

[10] T. Bouabana-Tebibel, M. Belmesk, Formalization of UML object dynamics and behavior, in: Proc. 2004 IEEE Int. Conf. on Systems, Man and Cybernetics, The Hague, Netherlands, October 2004.

[11] J. Delatour, Contribution à la spécification des systèmes temps réel: l'approche UML/PNO, Doctorat thesis, University of Paul Sabatier, Toulouse, France, 2003.

[12] D. Distefano, J.-P. Katoen, A. Rensink, On a Temporal logic for object-based systems, in: Proc. Fourth Int. Conf. on Formal Methods for Open Object-Based Distributed System, FMOODs 00, USA, 2000.

[13] M. Dumas, A.t. Hofstede, UML activity diagrams as a workflow specification language, in: Proc. Int. Conf. on the Unified Modeling Language (UML), Toronto, Canada, 2001.

[14] R. Eshuis, R. Wieringa, Tool support for verifying UML activity diagrams, in: IEEE Transactions on Software Engineering, vol. 30, No. 7, IEEE Computer Society, 2004, pp. 437–447.

[15] R. Eshuis, R. Wieringa, Comparing Petri Net and activity diagram variants for workflow modelling—a quest for reactive Petri Nets, in: Advances in Petri Nets: Petri Net Technology for Communication Based SystemsLecture Notes in Computer Science, vol. 2472, Springer-Verlag, 2003, pp. 321–351.

[16] R. Eshuis, J. Dehnert, Reactive Petri Nets for Workflow Modeling, in: ICATPN 2003, in: W.M.P. van der Aalst, E. Best (Eds.), Lecture Notes in Computer Science, 2679, Springer-Verlag, Berlin Heidelberg, 2003, pp. 296–315.

[17] R. Eshuis, Semantics and verification of UML activity diagrams for workflow Modelling, PhD thesis, Centre for Telematics and Information Technology, University of Twente, 2002.

[18] S. Flake, UML-Based Specification of State-oriented Real-time Properties, PhD thesis, Faculty of Computer Science, Electrical Engineering and Mathematics, Paderborn University, Germany. 2003.

[19] S. Gnesi, F. Mazzanti, On the Fly Model Checking of Communicating UML State Machines, Tech. Rep. 2003-TR-63, Istituto di Scienzae Tecnologie dell'Informazione "Alessandro Faedo", Italy, 2003.

[20] D. Harel, H. Kugler, A. Pnueli, Synthesis revisited: generating statechart models from scenario-based requirements, in: Formal Methods in Software and System Modeling, Lecture Notes in Computer Science, vol. 3393, 2005.

[21] R. Heckel, Open Petri Nets as semantic model for workflow integration, in: H. Ehrig et al. (Eds.), Petri Net Technology for Communication-Based Systems, Lecture Notes in Computer Science, vol. 2472, Springer, 2003, pp. 281–294.

[22] Z. Hu, S.M. Shatz, Mapping UML diagrams to a Petri Net notation for system simulation, in: Proc. 16th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE 2004), Canada, 2004.

[23] K. Jensen, Coloured Petri Nets, vol. 1: Basic Concepts, Springer-Verlag, 1992.

[24] S.-K. Kim, D. Carrington, Formalizing The UML Class Diagram Using Object-Z, UML'99, in: The Unified Modeling Language Beyond The Standard, USA, Lecture Notes in Computer Science, vol. 1723, Springer, October 1999.

[25] C.A. Lakos, Object-Oriented Modelling with Object Petri Nets, in: G. Agha, F.D. Cindio, G. Rozenberg (Eds.), Lecture Notes in Computer Science, Springer-Verlag, 2001.

[26] C.A. Lakos, The consistent use of names and polymorphism in the definition of object Petri Nets, in: Proc. 17th Int. Conf. on the Application and Theory of Petri Nets, Osaka, Japan, Lecture Notes in Computer Science, vol. 1091, Springer-Verlag, 1996, pp. 380–399.

[27] C.A. Lakos, Object Petri Nets – Definition and Relationship to Coloured Nets, Technical Report TR94-3, Computer Science Department, University of Tasmania, 1994.

[28] I. Ober, S. Graf, I. Ober, Validating Timed UML Models by Simulation and Verification, in: Proc. Int. Workshop SVERTS: Specification and Validation of UML Models for Real Time and Embedded Systems, Fort Mason Center, San Francisco, California, USA, October 2003.

[29] Object Modeling Group, UML 2.0 Superstructure Specification, 2004.

[30] Object Modeling Group, OMG Unified Modeling Language Specification, version 1.5, 2003.

[31] OMG, Object Management Group, UML 2.0 OCL Specification, October 2003.

[32] Object Modeling Group, The UML Action Semantics, November 2001.

[33] PROD 3.4, An advanced tool for efficient reachability analysis, Laboratory for Theoretical Computer Science, Helsinki University of Technology, Espoo, Finland, 2004.

[34] J. Rumbaugh et al., Object-oriented Modelling and Design, Prentice-Hall, 1991.

[35] J.A. Saldhana, S.M. Shatz, Z. Hu, Formalization of object behavior and interactions from UML models, International Journal of Software Engineering and Knowledge Engineering (IJSEKE) vol. 11 (6) (2001) 643–673.

[36] K. Salimifard, M. Wright, Petri net-based modelling of workflow systems: an overview, European Journal of Operational Research vol. 134 (3) (2001) 218–230.

[37] T. Schattkowsky, W. Mueller, A. Rettberg, A model-based approach for executable specification on reconfigurable hardware, in: Proc. of DATE05, Munich, IEEE CS Press, Los Alamitos, USA, 2005.

[38] S. Shalaer, S.J. Mellor, Object Life Cycles – Modeling The World in States, Yourdon Press, Prentice Hall, 1992.

[39] N. Truong, J. Souquières, Validation of UML Static Diagrams Using B, in: Proc. Int. Conf. on Software Engineering Research and Practice SERP'05, Las Vegas, USA, 2005.

[40] N. Truong, J. Souquières, Validation des propriétés d'un scénario UML/OCL à partir de sa dérivation en B, in: Proc. Approches Formelles dans l'Assistance au Développement de Logiciels, France, 2004.

[41] S. Uchitel, J. Kramer, J. Magee, Synthesis of behavioral models from scenarios, IEEE Transactions on Software Engineering 29 (2) (2003) 99–115.

[42] W.M.P. Van der Aalst, The application of Petri nets to workflow management, The Journal of Circuits, Systems and Computers 8 (1) (1998) 21–66.

[43] T. Ziadi, L. Hélouët, J.-M. Jézéquel, Revisiting Statechart Synthesis with an Algebraic Approach, in: Proc. 26th Int. Conf. on Software Engineering (ICSE'04) ACM, Edimburgh, UK, 2004, pp. 242–251.