Thouraya Bouabana-Tebibel

Object Dynamics Formalization Using Object Flows within UML State Machines

UML, the de-facto standard for object-oriented modeling, currently still lacks a rigorously defined semantics for its models. This makes formal analysis and verification of model properties extremely difficult. OCL, the Object Constraint Language is part of UML for the expression of system properties. To validate formally these properties, we first present a technique for transforming a UML object life cycle model into Object Petri nets. We are especially interested in the modeling of communicating systems and for this purpose we use the state machines as models of the object behaviour. Secondly, we resort to the object and sequence diagrams which provide respectively identified objects and events for initializing the Petri nets derived from the state machines. Thirdly, validation of OCL invariants which are translated into temporal logic properties to be checked on the Petri nets derived from the UML models, requires integration of object flows within the state machines. These object flows express the dynamic creation and deletion of objects in the class association ends. Our interest in the association ends is motivated by the fact that they constitute the most important constructs of OCL expressions. A case study is provided throughout the paper to illustrate the methodology.

1 Introduction

The Unified Modeling Language UML [UML03a] is currently considered as the universal notation for object-oriented specification of complex system artefacts, in graphic and documented form. Unfortunately, it suffers from continuing criticism on the precision of its semantics at a time when the verification of the correctness of models has become a key issue. UML 2.0 [UML04] brings more precision to its semantics, but it remains informal and lacks tools for automatic analysis and validation.

On the other hand, Petri nets [Jens92] are a formal language that relies on a mathematical theory which permits abstract proof activities. They can express most of the object behaviours and consequently, seem to be a natural technique for modeling the object dynamics [BeDM02]. Their drawback, in contrast to the UML notation, is their high learning cost and their failure to model large-scale applications.

The communicating systems (Internet, satellites …) are composed of entities which continuously interact with each other or with their environment [Berr00], [HaPn85]. The dynamics of an object interacting with other objects is highlighted in UML by means of state machine diagrams. The latter model the object life cycle emphasizing the object interactions which are expressed by means of exchanged messages. Indeed, state machines describe the different states of an object where the transitions from one state to another are generally induced by event triggers. As for the static representation of the interactions among the objects of the system, it is specified using collaboration diagrams. The latter focus on the modeling of the communicating classes and the messages they exchange.

To fill gaps in the UML notation and Petri nets for modeling object interactions, we presented in [BoBe04] a technique for transforming the UML state machines into Object Petri Nets, OPNs for short. The latter are then connected using the collaboration diagram structure and information, see figure 1. The OPNs are a generalization of ordinary Petri nets relying on the basic principles of the object-oriented approach and supporting convenient definition and manipulation of object values. Lakos demonstrates in [Lako01] that OPNs constitute a natural target formalism for object-oriented modeling languages since they provide a clean integration of the static (class) and dynamic (lifecycle and interactions of a class) models. He applies his theory to the Rumbaugh's OMT notation [RBL+91].

In the present paper, we extend the work presented in [BoBe04] by developing a technique for dealing

with the verification process. It appears from our investigations that the research so far has tackled only the formalization of models with anonymous objects and there is no research yet which targets the analysis of models by considering objects identified by identities and attribute values. Thus, the main idea of this paper is to validate the UML state machines of communicating classes by proposing a methodology which on one hand, exploits the object and sequence diagrams to initialize the derived OPNs and on the other hand, uses the system properties expressed in the Object Constraint Language (OCL) [UML03b] to validate the models, see figure 1.

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. They will be used to initialize the OPNs marking with objects named by identities and attribute values. As for the sequence diagrams, they constitute an attractive visual formalism, widely used to capture system scenarios [PiJé04]. They describe interactions by focusing on the sequencing of the exchanged messages among a group of objects. They will be used to define the *OPN*'s initial marking with the events of the scenario that will be verified.

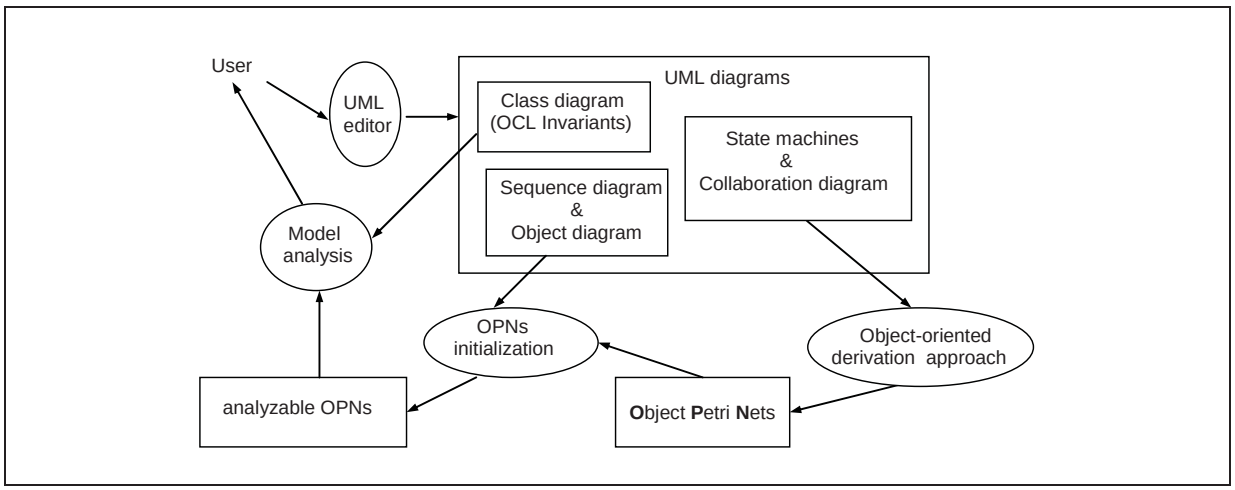
Once initialized, the OPNs are analyzed by means of PROD [PROD04], a model checker tool for predicate/transition nets. To avoid the high learning cost of the model checker, we suggest that the designer specifies the system properties in OCL which is part of UML. OCL permits the formulation of restrictions over UML models, in particular, invariants. We automate after that, the translation of these invariants to temporal logic properties so that they can be verified by PROD during the Petri net analysis.

OCL invariants are specified on class diagrams. The latter model 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 into Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) properties is not sufficient for realizing a property checking. Indeed, the association ends rank among the most important constructs of OCL expressions. Unfortunately, they do not appear on the object life cycle diagram (state machine). In order that the OCL expressions composed of association end constructs can be verified on the derived OPNs after they have been translated into temporal logic properties, the association ends should be modeled on the state machines. This modeling should show the object flow regarding the roles the object plays, in other words, the activity of creation/ deletion of objects in/from the association ends. So, to achieve a systematic formal verification of OCL invariants, we propose an approach for the integration of the association end update within the state machines. This is realized using the link actions [UML01] which are translated into Petri nets while transforming the state machine.

In short, our most relevant contribution aims at a value-oriented validation of UML state machines. For this purpose, we first use the object and sequence diagrams to initialize the specification. Then, we resort to the OCL invariants which allow an evaluation of the system properties by using UML concepts. However, the formal validation of these properties

Figure 1: Methodology of modeling and analysis



requires a specific object flow specification on the state machines by using the association end constructs. This approach has not been so far the object of research in which the model initialization is generally based on anonymous objects and the object flows are never modeled on the state machines.

The remainder of the paper starts with a brief overview on the mapping of UML state machines to Petri nets. This mapping constitutes the background of the present work. In sections 3 and 4 the proposed approach is presented and the techniques upon which it is based are developed. These techniques are illustrated throughout the paper using a case study. Some results on the model analysis are given and commented in section 5. We provide in section 6 the reasons that motivate our work and show its novelty and relevance by comparison with related works. We conclude with some observations on the obtained results and recommendations for future research directions.

2 Background

We summarize in this section the work that we presented in [BoBe04] to transform UML state machines into Object Petri nets. This work supports the approach that we develop in the present paper.

2.1 State machines

A state machine relates to the behaviour of a class describing its states and the messages it exchanges with other state machines classes. This behaviour starts from one initial state and terminates in one final state such that every intermediate state is on a path from the initial to the final state. An intermediate state models a situation during which some invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the execution of an action. This execution is modeled by using the keyword *do* which introduces an activity (called *do activity*) that is performed as long as the object is in the state.

Other actions, defined as atomic, may be specified within the state, namely those introduced by the keywords *entry* and *exit*. The entry action is performed at the entry of the state whereas the exit action is performed upon exiting the state, see figure 2.

The states are linked by means of transitions annotated with the event that triggers the transition (event trigger) and atomic actions produced by the triggered transition. An empty event indicates that the transition can fire spontaneously, while an empty action indicates that the transition produces no action.

Due to their atomicity, the entry, exit and transit actions are in fact, generated events respectively called: *entry*, *exit* or *transit* events.

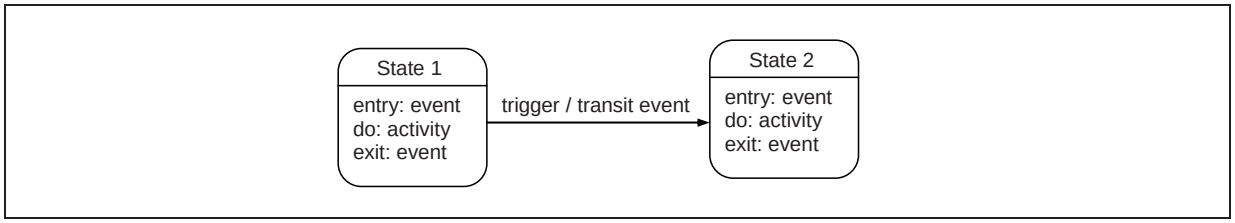
Events are of two types: *send* events and *call* events. The *call* event includes the particular events *create* and *destroy*. Events are mentioned on the state machine as follows: *«send» class(), «call» operation(), «create» class()* or *«destroy» class()*. Examples of these events are given in the case study of figure 4.

2.2 Case study

We illustrate our study and first of all, the state machine models, through an interactive application that is mainly characterized by a great number of exchanged messages between its components. This application concerns a message server whose main role is to manage the communication between the connected stations. All of the exchanged messages must go through the server, to be forwarded to the receivers. The corresponding class diagram is presented in figure 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 stereotyped class *Command* models the used signals. It supports four types: *Connection*, *Okconnection*, *Disconnection* and *Okdisconnection*.

Figure 4 presents the state machine of a station which connects itself to the server after it has performed a self diagnostic (*check* activity). Its connection request is realized by using the *«send» connection* event. The server confirms the station connection using the *«send» okconnection* event. When connected, a station can notify a message, receive a message or disconnect itself. It creates a message

Figure 2: Events and activity of the state machine



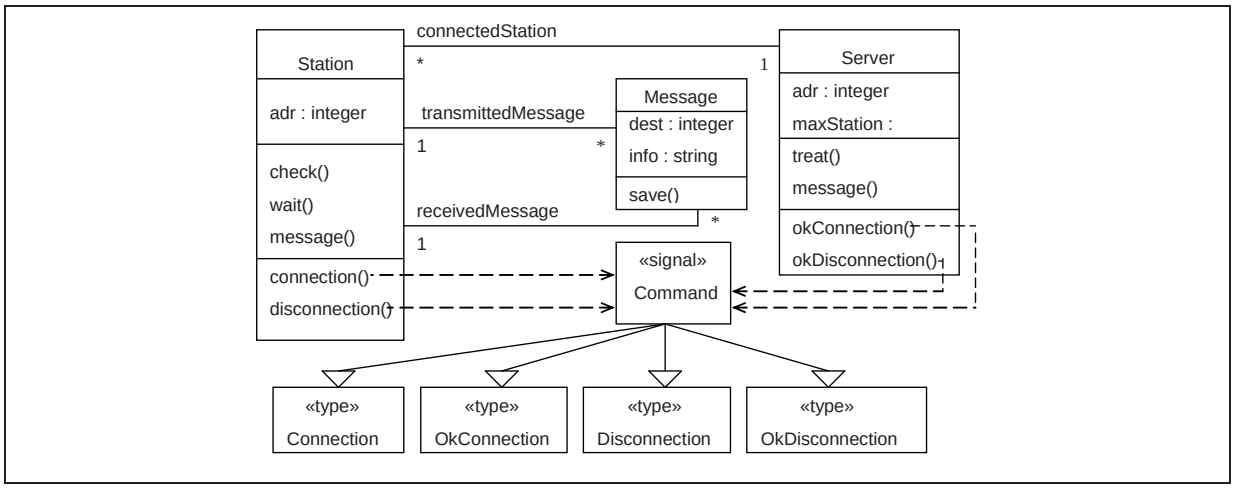


Figure 3: Class diagram of the message server

using a *«create» message* event and then notifies it by means of a *«send» message* event.

After it has received a forwarded message from the server by means of a *«send» message* trigger, the station saves it using a *«call» save* event. Its disconnection is requested by the *«send» disconnection* event and confirmed by the *«send» okdisconnection* event.

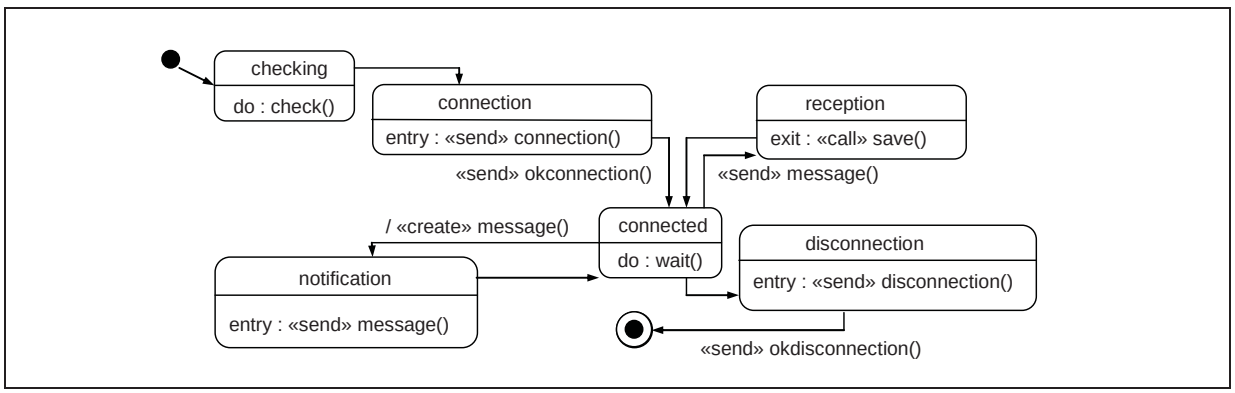
2.3 Object Petri Nets

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 appropriate to move towards OPNs which support object-oriented structuring, thus allowing the definition of clean interfaces for object interactions. Furthermore, OPNs can be transformed into behaviourally equivalent Colored Petri Nets [Lako94], thus providing the basis for applying traditional analysis techniques.

OPNs have been formally [Lako96] and informally [Lako01] 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 attributes and methods of a class. The attributes are expressed as components of the token which represents the object. As for the methods, they describe the object life cycle (state machine) in the form of a bunch of places, functions and transitions. The places are of two types: simple and super. The simple places hold tokens of simple type (integer, real, boolean, ..), 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 state according to the usual Petri net semantics. As for the super transition, it corresponds to a set of internal actions.

Figure 4: State machine of the station class



More formally, we define an OPN by the 7-tuple $<P, T, A, C, Pre, Post, M_0>$ where:

- $P = \{p\}$ is a set of places.
- $T = \{t\}$ is a set of transitions.
- $A \subseteq P \times T \cup T \times P$, is a set of arcs.
- $C = \{c\}$ is a set of colours where $c = \{<v_1, v_2, \ldots, v_k>\}$ and v_j is a variable or a constant.
- $Pre : P \times T \rightarrow \mathcal{P}(C)$ is a precondition function to the transition firing such that $Pre(p_i, t_i) = \sum c_k$.
- $Post : P \times T \rightarrow \mathcal{P}(C)$ is a postcondition function to the transition firing such that $Post(p_i, t_i) = \sum c_k$.
- $M_o : P \rightarrow C$ is the initial marking function, such that $M_o(p) = \sum c_k$.

2.4 Overview of the derivation approach

After defining in the previous sections the notation and formalism of interest, we provide in this section an overview of the derivation approach that we presented in [BoBe04]. Because of UML's and OPN's suitability for the object-oriented modeling, we proposed the transformation of each state machine modeling interactive class behaviour into an object subnet called Dynamic Model or *DM* (see figure 5). To construct the *DM*, each state $s \in S$ is converted to a place $p \in P$ and each transition $tr \in Tr$ is converted to a transition $t \in T$ with an arc at its input and an arc at its exit. As for the *do activity*, it is translated into a pair of transition-place with an arc at the input and the exit of the transition. These transformations are shown on figure 6 through the mappings M1, M2 and M3.

To deal with Petri net simulation, we address the Petri net initial marking which may be of two types: static and dynamic. The static initial marking provides the class instances and their attribute values. These instances are extracted from the object diagram to initialize the *Object* place with tokens of *object* type. The dynamic initial marking provides the exchanged messages among the interactive objects. These messages are extracted from the sequence diagram to initialize the *Scenario* place with tokens of *event* type.

The event triggers occur on the *DM* through the *Input* place. They are represented by arcs from the *Input* place to the transition on which they occur (see mapping M7 on figure 6). With the places *Object*, *Scenario* and *Input*, the *DM* constitutes an Object Petri net Model that we call *OPN*. To connect the different *OPNs*, we use the *Link* place through which all exchanged messages should pass. Thus, for each *OPN*, a directed transition from the *Link* place to the *Input* place is built. The transition firing is conditioned by the events that are inputs to the class (whose dynamics is specified by the *DM*) represented on the collaboration diagram.

As for events generated on the state machine, they are converted to an arc from the *Scenario* place to the transition to which they are related and an arc from this transition to the *Link* place (see mappings M4, M5, M6 on figure 6).

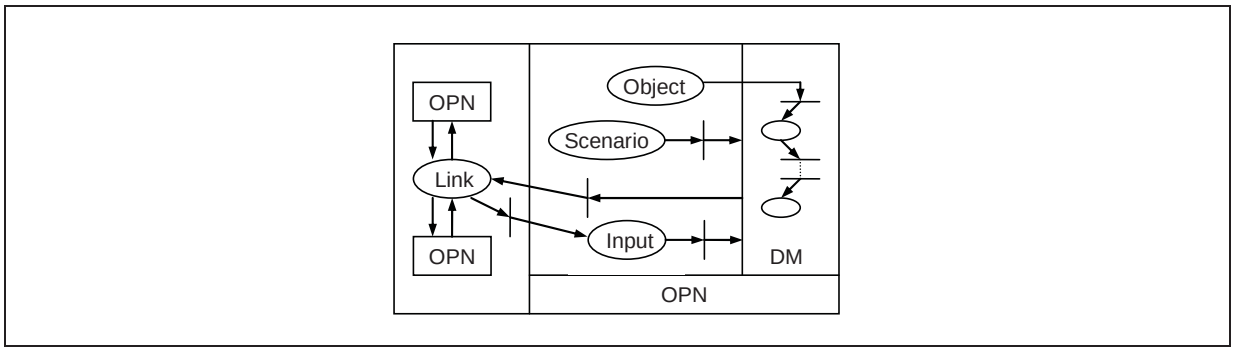
Figure 6 gives an overview of the semantics mapping of the state machine basic constructs into their counterparts in Petri nets. The dashed symbols represent associated constructs not concerned by the translation.

In figure 7, we show the *OPN* resulting from the conversion of the *station* state machine**.**

3 Initialization technique

To deal with the model simulation, starting from state machine diagrams, two types of arguments must be initialized, namely, the system objects and the exchanged messages among these objects. Thus, we proceed to two types of initialization that we call static and dynamic.

Figure 5: Petri nets interconnection architecture



3.1 Static initialization

For the requirements of our initialization approach, we distinguish between two types of objects: active and passive. Active objects interact using messages, whereas passive objects are exchanged among the active objects. For example, in the server message application, the *server* and *station* objects are active while the *message* and *command* objects are passive.

For the static initialization, only the active objects are concerned. These objects are formalized by coloured tokens of the form: <*obj, attrib*> where *obj* designates their identity and *attrib* the set {$attrib_1$, …, $attrib_k$} 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. The active objects initialize Petri net marking by providing the tokens that move on the *DM*. Thus, all objects instantiated from the same class on an object diagram are inserted in the *Object* place of the *OPN* translating the state machine of the class.

Figure 8 shows an object diagram of the message server application before any action (there are no links between the objects). For each station, the IP address is given.

3.2 Dynamic initialization

Sequence diagrams allow the modeling of specific scenarios. They show 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:

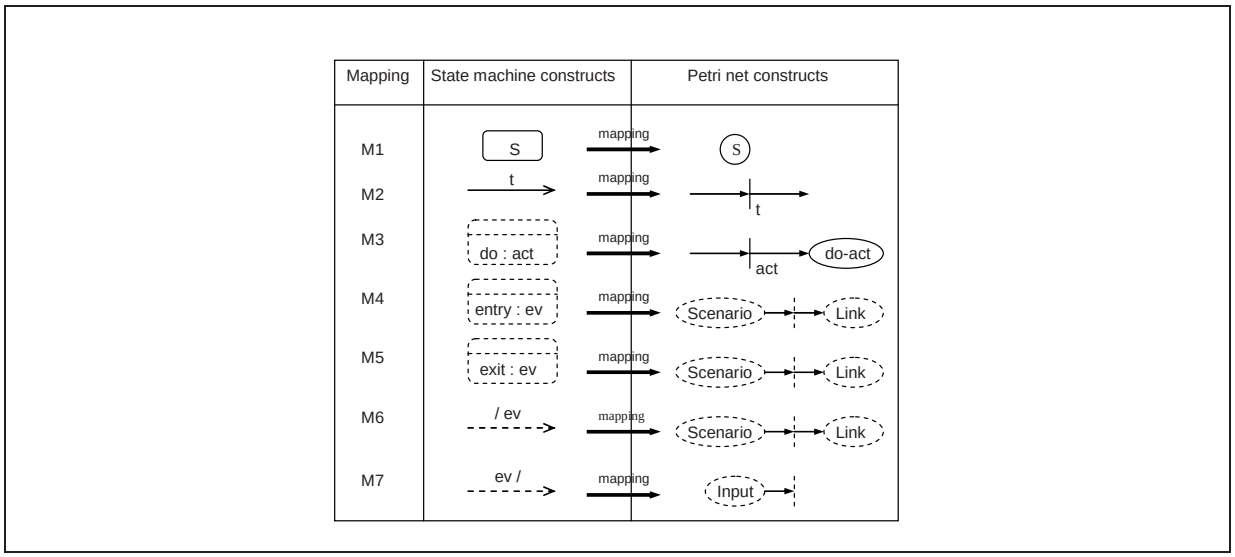
object:Class. The messages reflect events specified with their attribute values, as follows: *«send» class(attrib), «call» operation(attrib)*, *«create» class(attrib), «destroy» class(attrib),* see figure 8. This specification enables the events that are dynamically generated on the state machine.

The sequence diagram of figure 8 shows a scenario related to the server message application presented in section 2.2. Two stations *st1* and *st2* request a connection from a server *s* after performing a self diagnostic. When done, *st1* creates a message *m1*, transmits it and then, disconnects itself. *m1* is forwarded by *s* to *st2*. *st2* saves it and then, disconnects itself.

We formalize an interaction on a sequence diagram by the 5_tuple (*ev, srce, targ, xobj, attrib*). The component *ev* identifies the event *(«send» class(), «call» operation(), «create» class(), «destroy» class()*). *Srce* and *targ* are respectively the source and the target object identity. The component *xobj* gives the exchanged object identity (*object:Class*) if the sent message is an object and only the class name (*Class*) if the sent message is a signal. As for *attrib*, it designates the set {$attrib_1$, …, $attrib_k$} of the exchanged object attributes.

The dynamic model *DM*, derived from a state machine, is a generic model, specifying the object overall behaviour. Its initialization, deduced from a sequence diagram is done by means of: home messages (generated by the model objects towards other objects) or border messages (generated from the system environment towards the model objects).

Figure 6: Derivation of UML constructs to Petri nets



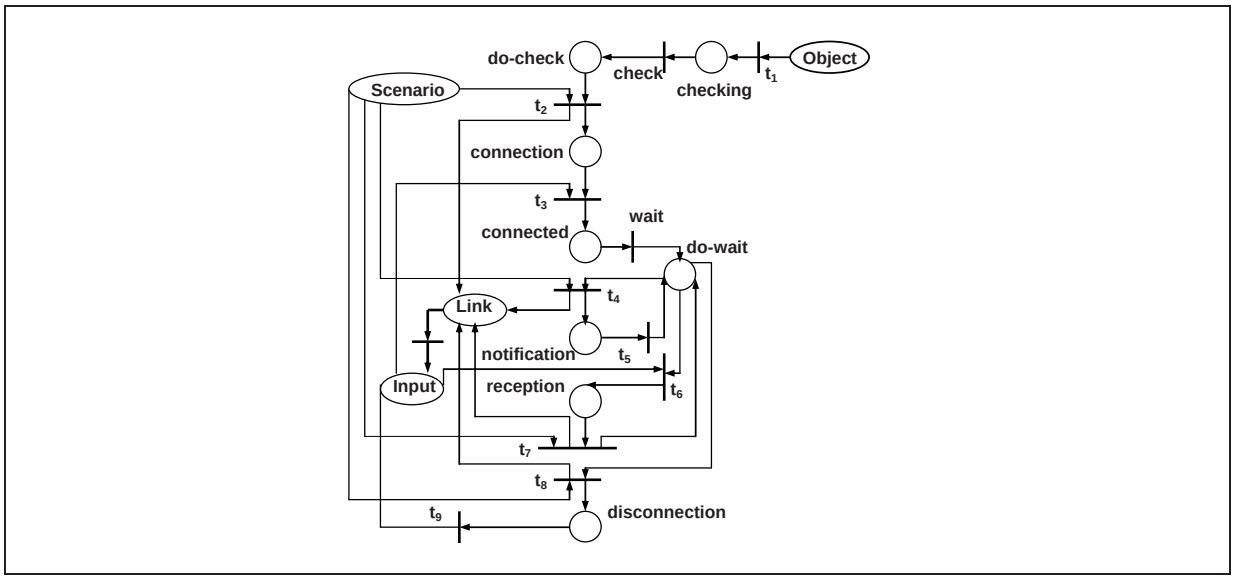


Figure 7: Petri nets derived from the state machine of the Station class

The home messages are grouped together per class, so that for each object, only the output events are retained. They are converted afterwards to tokens of the form $< ev^{(Sc)}, src e^{(Sc)}, targ^{(Sc)}, xobj^{(Sc)}, attrib^{(Sc)}>$ and stored in the *Scenario* place of the *DM* corresponding to their class. Through this initialization, the *Scenario* place enables the events generated by the *DM* and identifies them with real values, thus producing a Petri net animation.

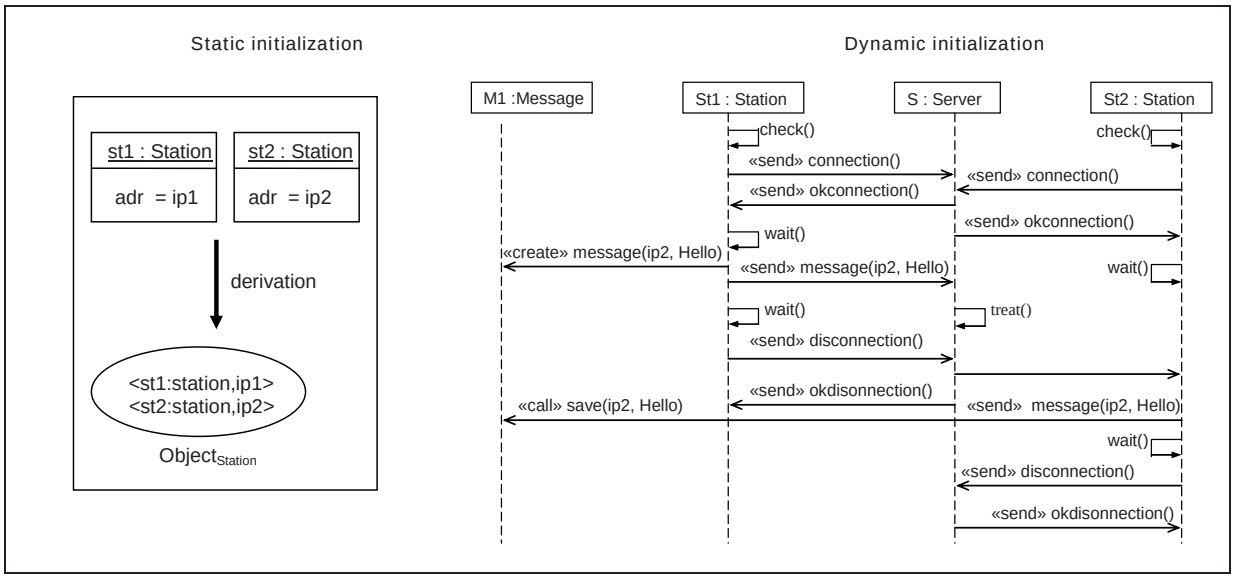
Since their source object are not represented on the model, all border messages are directly stored in the *Link* place which is common to all classes. After that, they are converted to tokens of the form: $<ev^{(Li)}, , targ^{(Li)}, xobj^{(Li)}, attrib^{(Li)}>$. This initialization permits the opening of the Petri net model by using the *Link* place which is defined as an *open* place.

The transformation of the sequence diagram of figure 8, gives the following *Scenario* place for the *OPN* of the *station* class.

4 System property validation

Verification by model checking – as treated in PROD – is based on state space generation and verification and validation of safety and liveness properties of a system on this space. The verification tackles the good construction of the model, using generic properties as deadlock, livelock, reject states, etc. As for

Figure 8: Static and dynamic initialization



```
Scenario = <«send» connection, st1:Station, s:Server, connection>
          + <«send» connection, st2:Station, s:Server, connection>
          + <«create» message, st1:Station, m1:Message, m1:Message, ip2, Hello>
          + <«send» message, st1:Station, s:Server, m1:Message, ip2, Hello>
          + <«send» disconnection, st1:Station, s:Server, disconnection>
          + <«call» save, st2:Station, m1:Message, ,>
          + <«send» disconnection, st2:Station, s:Server, disconnection>.
```

the validation, it checks whether the model is constructed in conformity with the customer initial requirements. For this purpose, specific properties of the system, written by the modeler in Linear temporal Logic (LTL) or Computational Tree Logic (CTL) are used. For both 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 counterexample.

Since the main motivation of this work is that the UML designer may reach valid models without the need for knowledge of formal techniques, it is only reasonable that the properties are expressed by the modeler in the OCL language and are automatically translated afterwards into LTL and CTL. 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 [BeKS02]. 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 [BoBe06]. Other works like those of Distefano [DiKR00] and Flake [FlMu04] have also invested this research direction.

OCL is mainly based on the use of operations on collections for specifying object invariants. Since these collections correspond to association ends, the latter must appear on the Petri net specification so that the derived LTL and CTL properties (whose expression is essentially made of these constructs) can be verified. This requires the association ends to be modeled onto the state machines in order to get, after their transformation, the equivalent Petri net constructs. This object flow 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 of 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 is never considered in the modeling, see figures 4 & 9. That is why we recommend to the designer to specify the link actions on the state machines so that the OCL invariants can be verified.

UML action semantics was defined in [UML01] for model execution and transformation. It is a practical framework for formal descriptions. For this work, we are particularly interested in the create link, destroy link and clear association actions.

The create link action permits the addition of a new end object in the association end. The destroy link removes an end object from the association end. The clear association action destroys all links of an association in which a particular object participates. These actions will be represented on the state machine as tagged values of the form *{linkAction(associationEnd)}*, following the event which provokes the association end update.

In figure 9, once the station is connected (by reception of *«send» okconnection()*) or disconnected (by reception of *«send» okdisconnection()*), it adds or removes itself from the association end *connectedStation*, using respectively, *{createLink(connectedStation)}* or *{destroyLink (connectedStation)}*. It adds a sent or received message with *{createLink(transmittedMessage)}* or *{createLink(receivedMessage)}*, respectively. Finally, after it receives a disconnection confirmation, the station clears its association ends with the *Message* class, using *{clearAssociation(transmittedMessage)} and {clearAssociation(receivedMessage)}*.

The link actions may concern an active or passive (exchanged) end object. The object-oriented approach, on which both UML and OPNs rely, is based on modularity and encapsulation principles. To deal with modularity, a given association end should appear and be manipulated in only one state machine. In Petri nets, an association end is modeled by a place of *role* type, see rule 1. This place holds the name of the association end and belongs to the *DM* translating the state machine.

The question raised is then to which state machine the association end should belong, to the state machine of the class modeling its objects or to the class at the opposite end? For example, concerning the

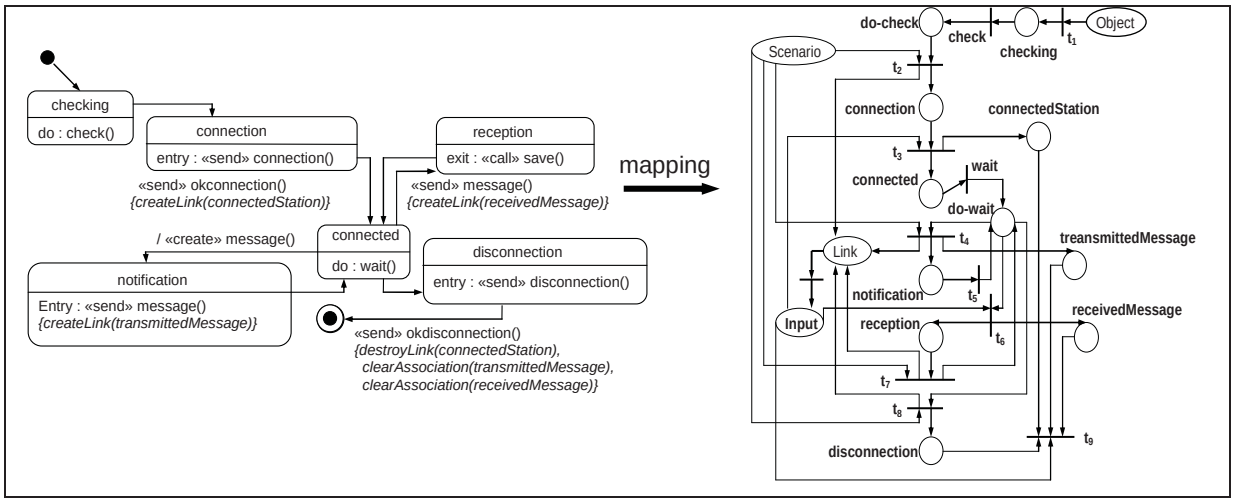


Figure 9: transformation of the state machine of the station class considering the link actions action

update of the association end connectedStation, in which class should it be carried out, in the class Station or the class Server?

An association end regrouping active objects must be updated within the state machine of the class of these objects, 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 adding the object to or removing it from the association end. They are given by the tuple <*object, attrib*> which specifies an active object. As an example, see the modeling of the association end *connectedStation* on figure 9.

As for exchanged objects, they are usually manipulated by active objects and are not specifiedby dynamic models. So, the association end representing them could be updated in the state machine of the active class that is at the opposite end. For the exchanged objects, the encapsulation constraint is lifted given that their attributes are transmitted within the message – which is of the form <*ev, srce, targ, xobj, attrib*> – and so, accessible by the active object when performing the association end update. For example, see the modeling of the association ends *transmittedMessage* and *receivedMessage* on figure 9.

In Petri nets, the create link action is semantically equivalent to an arc starting in the transition with the end update towards the place specifying the association end. This translation is formalized by rule 2. The destroy link action is semantically equivalent to an arc from the association end place to the transition corresponding to the link action, see rule 3.

The object to be added to / removed from the association end is extracted from the components of the token corresponding to the event that provokes the association end update. This token is situated in the *Scenario* place if the event is generated. It is located in the *Link* place if the event occurs, see rule 2.

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.

We propose the formalization of the mapping of the link actions by means of 4 rules. For this purpose, we define the concepts dealing with the association end update by the 7-tuple <*O, U, R, r, createLink, destroyLink, clearAssociation*> where:

- *O = {o}* is a set of active objects.
- *U = {u}* is a set of exchanged objects.
- *R = {r}* is a set of association ends.
- *r = {y}* is a set of objects of the association end *r, where* $y \in O \vee y \in U$.
- createLink : R→R is a function inserting an object into an association end such that createLink($r_i^{(k)}$) = $r_i^{(k+1)}$ = {$y_1^{(i)}$, $y_2^{(i)}$, …, $y_k^{(i)}$, $y_{k+1}^{(i)}$}, $y_k \in O \vee y_k \in U$.
- destroyLink : R→R is a function removing an object from an association end such that destroyLink($r_i^{(k)}$) = $r_i^{(k-1)}$ = {$y_1^{(i)}$, $y_2^{(i)}$, …, $y_{k-1}^{(i)}$}, $y_k \in O \vee y_k \in U$
- *clearAssociation : R→R* is a function removing all objects of an association end r_i such that *clearAssociation(r) = {}*.

Rule 1: Conversion of an association end

- For each $r \in R$, create $rol \in P$ (rol is of role type).

Rule 2: Conversion of a createLink() action

- For each *createLink(r)*, $r \in R$, following a generated event : create an arc $t \rightarrow rol \in T \times P$, such that:

  if $r \subset O$:
  Post(rol, t) = <$targ^{(Sc)}$, $srce^{(Sc)}$, attrib>

  if $r \subset U$:
  Post(rol, t) = <$srce^{(Sc)}$, $xobj^{(Sc)}$, attrib>

- For each $createLink_i(r_i^{(k)})$, $r_i \in R$, following an event trigger : create an arc $t_p \rightarrow rol_i \in T \times P$, such that :

  if $r \subset O$:
  Post(rol, t) = <$srce^{(Li)}$, $targ^{(Li)}$, attrib>

  if $r \subset U$:
  Post(rol, t) = <$targ^{(Li)}$, $xobj^{(Li)}$, attrib>

Examples of the translation of the link actions are presented on figure 9 which completes figures 4 & 7 regarding these actions.

Rule 3: Conversion of a destroyLink() action

The conversion of the *destroyLink()* action is treated in a similar manner as the *createLink()* action. The only differences between them are that:

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

The clear association action is semantically equivalent to an arc going from the association end place towards the transition related to the link action and removing all the association end objects linked to a given object. An object may only destroy itself or its associated exchanged objects. It can not destroy its associated active objects. Conversion of the clear association action is given by rule 4. An example is shown on the state machine and *OPN* of figure 9.

Rule 4: Conversion of a clearAssociation() action

- For each $clearAssociation_i(r)$ $r \in R$, after a generated event:

  create an arc rol→t ∈ P × T, such that:
  Pre(rol, t) =<$srce^{(Sc)}$, xobj, attrib>

- For each $clearAssociation_i(r)$ $r \in R$, after an event trigger:

  create an arc rol→t ∈ P × T, such that:
  Pre(rol, t) = <$targ^{(Li)}$, xobj, attrib>

5 Model Analysis

To test the practical implementation of our approach, we built a translator whose semantic functions are drawn from the conversion rules we have set in [BoBe04] and showed in section 2.4. We also developed a graphical interface for the construction of the class, state machine, collaboration, object and sequence diagrams. These diagrams constitute the input of the translator whose outputs are predicate/transition nets, specified in PROD syntax. A little part of the translated model of figure 9 is given in what follows, for the transition (*t3*) which goes from the state *connection* to the state *connected*.

```
#trans t3
in {connection:<.targ,attrib.>;
  Input:<.srce,targ,xobj,attrib1,attrib2.>}
out {connected:<.targ,attrib.>;
  connectedStation:<.srce,targ,attrib.>}
#endtr
```

where the keywords *trans, in, out* and *endtr* designate respectively the transition, its input places, its output places and its end.

PROD was executed afterwards to verify the models. The Petri net initial marking was defined by the object and sequence diagrams of figure 8.

The generic properties concerning the absence of livelock (infinite loops) and deadlock have been first checked. They were specified by means of the PROD commands:

```
#place tester lo(<.0.>)hi(<.0.>)mk(<.0.>)
  #tester tester deadlock(<.0.>).
  #place tester lo(<.0.>)
    hi(<.1.>) mk(<.0.>)
  #tester tester livelock(<.1.>)
```

This gives the following results:

```
0#statistics
Number pf nodes: 201
Number of arrows: 436
Number of terminal nodes: 1
Number of nodes that have been completely processed: 201
0#
0#goto 200
200#look
Node 200
transmittedMessage: <.st1_station,m1_message,ip2,Hello.>
receivedMessage: <.st2_station,m1_message,ip2,Hello.>
free: <.s_server,ip,2.>
```

where the nodes are the different states of the system life cycle and the arrows are the transitions between these states. Here, a node (or a state) represents a view of the system, identified by a marking and obtained after a Petri net transition firing. The last node (number 200, the first is number 0) which is given after the last transition of the system behaviour, shows the final marking of the Petri net model. Only the places with tokens are represented. The others are empty. We notice that both of the association end places *transmittedMessage* and r*eceivedMessage*, include the message m1. The association end place *connectedMessage* is empty, because the stations have disconnected themselves. As for the server, it waits for new connection requests (in the place free).

Some system invariants were afterwards expressed in LTL properties and verified. Two of these properties are expressed below in a paraphrased (textual) form and then, specified as OCL invariants and translated into LTL properties. To make easier the comprehension of the properties, refer to the class diagram of the server message application (figure 3).

Property 1

The number of connected stations is limited to maxStation.

Property 1 expression in OCL

```
context s:Server inv :
s.connectedStation→size <= s.maxStation
```

Property 1 expression in PROD

For each server s and for each place of its DM^* write the property:

```
# verify henceforth (card(connectedStation :
field[0] == s_server) <= (place_DM*Server :
field[2]))
```

where:

- field[0] designates the first component (*assoc*) of the *connectedStation* tokens,
- field[2] designates the third component ($attrib_2$ = *maxStation*) of the tokens of the server *DM**.

Property 2

Only connected stations can transmit messages.

Property 2 expression in OCL

```
Context s:Server inv:
s.connectedStation→ excludes(st1:Station)
implies st1.transmittedMessage→isEmpty()
```

Property 2 expression in PROD

```
#verify henceforth (connectedStation:
(field[0] == s_server && field[1]
==st1_station) == empty implies
(transmittedMessage: field[0] ==
st1_Station) == empty)
```

where :

- connectedStation: field[0]==s_server && field[1]==st1_station designates the 1st and 2nd components of the connectedStation tokens,
- transmittedMessage: field [0]==st1_station designates the 1st component of the *transmittedMessage* tokens.

When PROD program is executed with these three properties, it terminates without signalling errors.

6 Contributions vs. related work

6.1 On the initialization of the specification

For formalization purposes, the sequence diagrams are generally combined with the state machines in order to connect the object life cycles [BeDM02]. They are also transformed separately into other formalisms to validate specific scenarios [HaKP05] or composed together to describe the system overall behaviour [UcKM03]. As far as we are concerned, we introduce three novel uses of the sequence diagrams that we integrate within the process of model initialization.

We, first, use the sequence diagrams to initialize the specification. The generic structure of the OPNs derived from the state machines, yields a specification that supports the whole system scenarios. Based on this, we suggest the use of the sequence diagram to initialize the Petri net marking with the events of the scenario that will be verified.

Second, we exploit the sequence diagrams 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 ordinary Petri nets is their incapacity to model open interactive systems. Many works such as [BCEH01] close this gap by proposing Open Petri nets that are ordinary Petri nets with a distinguished set of places, called open places. The latter 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 new objects dynamically. This deficiency is overcome in the literature with the OPNs' super places which enable the generation of new objects [Lako01]. Similarly to the open places, we propose an applicative use of the super places in the UML context by using the events *create* and *destroy*.

On the other hand, data formalization is usually given by means of state-oriented languages such as Z [AmPo03] or B [KiCa99]. We propose to specify data by means of object diagrams. These data provides Petri net initial marking with objects named by identities and attribute values.

Hsiung et al. [HLT+04] Present a work which is close to ours. They combine the statecharts, the sequence diagrams and OCL constraints to deal with system verification. However, their interest is on timed statecharts and so in contrast to our work they use the sequence diagrams to schedule the different tasks of the system behaviour and do not deal with the objects and events valuation. As for OCL constraints, the authors emit only the idea of translating them into timed CTL logic but do not develop any strategy of translation of these constraints.

6.2 On the validation of the specification

Formalization of UML state machine semantics [PaLi99],[Trao00],[Kusk01],[EsJW02],[NiAD03], [BaPe05] and integration in the state machines of languages state-oriented or property-oriented [AtPS03] have been widely investigated. The OCL language has also been integrated within the state machines in various works, in particular, those of Flake and Mueller [FlMu04] who extend it with temporal logic to express properties over time.

However, no previous work has tackled the integration of the association end specification within UML dynamic models. We can explain this, arguing that the UML/OCL association is rarely used to formally validate UML models. When done, it is limited to OCL attribute expressions [TrSo04] or OCL pre and post-conditions [FlMu04] whereas the association ends which yield the most important expressions are never treated. Generally, the formalized UML models are rather coupled with appropriate formalisms for the expression of system properties.

Thus, when the OCL invariants express constraints on the association ends, the latter should be modeled on the state machines so that they provide once transformed to Petri nets, a formal basis for the validation of the translated invariants. Otherwise, although correctly specified and translated to LTL or CTL, the properties could never be validated on the derived Petri nets without association end specification within the state machines.

The relevance of such an approach is to exploit all OCL capabilities to formally validate the system properties. These capabilities concern particularly, the navigation and operation constructs that yield most of the OCL expressions. Its only constraint concerns the obligation for the user to specify the link actions on the state machine. However, this constraint is minimal compared to that of limiting OCL expressions or specifying using formal languages like temporal logics.

6.3 On the modeling of large-scale systems

Expression of the object behaviour and interactions using state machines connected by means of collaboration diagrams, allows the modularization of the system activities per class. For large-scale systems, this modularity simplifies and clarifies the dynamic model representation. On the other hand, the object-oriented approach that we propose for formalizing these models provides an interconnection architecture of the derived Petri nets, which complies with modularity, see figure 5. Indeed, the proposed derivation approach is also modular, in the sense that each state machine is transformed separately into a *DM* model which communicates with other *DMs* through the *Link* place. Thus, the connection of the state machines using collaboration diagrams crowned with the object-oriented modular formalization approach that we formulate, proves to be appropriate for specifying large-scale systems by means of generic models.

7 Conclusion

This paper presents an approach for validating systematically UML models without the need for the user to know formal checking techniques. The verification concerns both the correctness of the model construction and the faithfulness of the modeling. The latter is allowed because of awaited system properties which are expressed by the modeler in OCL language and then translated into LTL and CTL properties. The formal validation of these properties required the integration of an object flow specification into the object control flow model (state machine), using predefined actions (link actions) on the association ends.

Unlike previous efforts in this area, which formalize UML dynamic models considering anonymous objects, the methodology which we propose deals with a value-oriented validation. It offers to the user the opportunity of validating his models 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 object dynamics expressed by means of the state machines.

Nevertheless, the initialization of these models with identified objects needs on 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 as future work, to start the simulation at a critical moment from the object life cycle and not necessarily from the initial state. For this purpose, the object diagram will be used to represent the system objects at this moment. This representation will influence the token distribution at the Petri net 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 of object and sequence diagrams. This choice should allow the treatment of 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, allows a 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.

Another prospect of this work concerns the analysis of the validation/verification results and then, their feedback to the user. Since the methodology supposes that the designer does not master the formal specification, the analysis results rendered by the model checker PROD, will not be obviously meaningful to him. So, these results must be firstly interpreted and then presented to him in a form where the error in models is simply and clearly pointed out.

8 References

[AtPS03] Attiogbé, C.; Poizat, P.; Salaun, G.: Integration of Formal Datatypes within State Diagrams. In: Proceedings of Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science 2621 (2003), pp. 341-355.

[AmPo03] Amálio, N.; Polack, F.: Comparison of Formalization Approaches of UML Class Constructs in Z and Object-Z. In: Proceedings of International Conference of Z and B Users, Lecture Notes in Computer Science 2561 (2003).

[BCEH01] Baldan, P.; Corradini, A.; Ehrig, H.; Heckel, R.: Compositional Modeling of Reactive Systems Using Open Nets. In: Proceedings of 12th International Conference on Concurrency Theory, Lecture Notes in Computer Science 2154 (2001), pp. 502-518.

[BaPe05] Baresi, L.; Pezzè, M.: Formal Interpreters for Diagram Notations. ACM Transactions on Software Engineering and Methodology - ACM Press 14 (2005) 1, pp. 42-84.

[BeKS02] Beckert, B.; Keller, U.; Schmitt, P.: Translating the Object Constraints Language into First-order Predicate Logic. In: Proceedings of Verify, Workshop at Federated Logic Conferences, Copenhagen, 2002.

[BeDM02] Bernardi, S.; Donatelli, S.; Merseguer, J.: From UML Sequence Diagrams and Statecharts to analysable Petri Net models. In: Proceedings of third international workshop on software and performance, Italy, ACM Press, 2002, pp. 35-45.

[Berr00] Berry, G.: The Foundations of Esterel. MIT Press. 2000.

[BoBe06] Bouabana-Tebibel, T.: Formal validation with OCL. In: Proceedings of 2006 IEEE International Conference on Systems, Man & Cybernetics, Taipei, Taiwan, 2006.

[BoBe04] Bouabana-Tebibel, T.; Belmesk. M.: Formalization of UML object dynamics and behaviour. In: Proceedings of 2004 IEEE International Conference on Systems, Man & Cybernetics, Netherlands, 2004.

[DiKR00] Distefano, D.; Katoen, J-P.; Rensink, A.: On a Temporal Logic for Object-Based Systems. In: Proceedings of 4th International Conference on Formal Methods for Open Object-Based Distributed System, FMOODs 2000, USA, 2000.

[EsJW02] Eshuis, R.; Jansen, D.; Wieringa, R.: Requirements-Level Semantics and Model Checking of Object-Oriented Statecharts, Requirements Engineering 7 (2002) 4, pp. 243-263.

[FlMu04] Flake, S.; Mueller, W.: Past- and Future-Oriented Temporal Time-Bounded Properties with OCL. In: Proceedings of 2nd International Conference on Software Engineering and Formal Methods, China, © IEEE Computer Society Press, 2004, pp. 154-163

[HaKP05] Harel, D.; Kugler, H.; Pnueli, A.: Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In: Proceedings of international conference on Formal Methods in Software and System Modeling, Lecture Notes in Computer Science 3393 (2005), pp. 309-324.

[HaPn85] Harel, D.; Pnueli, A.: On the development of reactive systems. In: Logic and Models of Concurrent Systems vol. 13. NATO ASI series F, Springer Verlag, 1985.

[HLT+04] Hsiung, P-A.; Lin, S-W.; Tseng, C-H.; Lee, T-Y.; Fu, J-M.; See, W-B.: VERTAF: an Application Framework for the Design and Verification of Embedded Real-Time Software. IEEE Transactions on Software Engineering 30 (2004) 10, pp. 656-674.

[Jens92] Jensen, K.: Coloured Petri nets, Vol 1: Basic Concepts, Springer, 1992.

[KiCa99] Kim, S.-K.; Carrington, D.: Formalizing The UML Class Diagram Using Object-Z. In: Proceedings of UML'99 – The Unified Modeling Language Beyond The Standard, USA, Lecture Notes in Computer Science 1723 (1999).

[Kusk01] Kuske, S.: A formal semantics of UML state machines based on structured graph transformation. In: Proceedings of UML: The Unified Modeling Language. Modeling Languages, Concepts and Tools, Lecture Notes in Computer Science 2185 (2001), pp. 241-256.

[Lako01] Lakos, C.A.: Object-Oriented Modelling with Object Petri Nets, In: G. Agha, F.D. Cindio and G. Rozenberg eds., Lecture Notes in Computer Science, 2001.

[Lako96] Lakos, C.A.: The Consistent Use of Names and Polymorphism in the Definition of Object Petri Nets. In: Proceedings of 17th International Conference on the Application and Theory of Petri Nets, Japan, Lecture Notes in Computer Science 1091 (1996), pp. 380-399.

[Lako94] Lakos, C.A.: Object Petri Nets - Definition and Relationship to Coloured Nets, Technical Report TR94-3, Computer Science Department, University of Tasmania, 1994.

[NiAD03] Niu, J.; Atlee, J. M.; Day, N. A.: Template Semantics for Model-based Notations. IEEE Transactions on Software Engineering 29 (2003), pp. 866-882.

[PaLi99] Paltor, I.; Lilius, J.: Formalizing UML State Machines for Model Checking. UML'99 – The Unified Modeling Language. Beyond the Standard. Lecture Notes in Computer Science 1723 (1999).

[PiJé04] Pickin, S.; Jézéquel, J-M. : Using UML Sequence Diagrams as the Basis for a Formal Test Description Language. In: Proceedings of Fourth International Conference on Integrated Formal Methods, England, Lecture Notes in Computer Science 2999 (2004), pp. 481-500.

[UML04] UML 2.0 Superstructure Specification, Object Management Group, 2004.

[UML03a] Unified Modeling Language Specification, version 1.5, Object Management Group, 2003.

[UML03b] UML 2.0 OCL Specification, Object Management Group, October 2003.

[UML01] The UML Action Semantics, Object Management Group, November 2001.

[PROD04] PROD 3.4, An advanced tool for efficient reachability analysis, Laboratory for Theoretical Computer Science, Helsinki University of Technology, Finland, 2004.

[RBL+91] Rumbaugh, J; Blaha, M.R.; Lorensen, W.; Eddy, F.; Premerlani, W.: Object-oriented modelling and design, Prentice-Hall, 1991.

[Trao00] Traoré, I.: An Outline of PVS Semantics for UML Statecharts. Journal of Universal Computer Science 6 (2000), pp. 1088-1108.

[TrSo04] Truong, N.; Souquières, J.: Validation des propriétés d'un scénario UML/OCL à partir de sa dérivation en B. In: Proceedings of Approches Formelles dans l'Assitance au Développement de Logiciels. France, 2004.

[UcKM03] Uchitel, S.; Kramer, J.; Magee, J.: Synthesis of Behavioral Models from Scenarios, IEEE Transactions on Software Engineering 29 (2003) 2, pp. 99-115.

Thouraya Bouabana-Tebibel

National Institute of Computer Science
BP 68M Oued Smar
16309 Algiers
Algeria
t_tebibel@ini.dz