Integration of the Association Ends within UML State Diagrams

Thouraya Bouabana-Tebibel[1] and Mounira Belmesk[2]
[1]National Institute of Computer Science, INI BP 68 M Oued-Smar 16309, Algiers
[2]Edouard Monpetit College, longueuil, Canada

Abstract: *UML currently still lacks a rigorously defined semantics for its models, which makes formally analyzing a model and verifying its properties extremely difficult. To remedy that, we first present a technique for transforming the UML statechart diagrams into Petri nets. Then we develop an approach based on the class association ends. This approach shows how to validate the system invariants, expressed in the Object Constraint Language, on the Petri nets derived from the UML models. System property validation starts with an initialization of the model, extracted from the object and sequence diagrams. A case study is given throughout the paper to illustrate the methodology.*

Keywords: *UML, OCL, Petri nets, temporal logics, verification.*

Received March 19, 2006; accepted September 7, 2006

1. Introduction

UML suffers from ceaseless critics on the precision of its semantics when the verification of modeling correctness has become a key issue [16]. UML 2.0 brings more precision on its semantics, but it remains informal and lacks tools for automatic analysis and validation [15]. We presented in [5] a methodology to automatically transform UML modeling in Petri nets [11] which are supported by lots of tools to verifying them. In the present paper, we carry on with this work by developing a technique to deal with the verification process.

The Petri nets resulting from the derivation process are analyzed by means of PROD [19], 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 Object Constraint Language (OCL) [17], which permits to formulate restrictions over UML models, in particular invariants. The latter are afterwards automatically translated to Linear Temporal Logic (LTL) 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 LTL properties is not sufficient for the property checking. Indeed, OCL expressions refer to classifiers to evaluate their attributes and association ends. The latter can be updated (created, modified or read) on the statecharts 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 PROD. In other words, he is called on to specify the association end update that provides the dynamics of the object through the roles it plays. In addition to the OCL invariant translation to LTL properties, we propose an approach to translate the link actions in Petri nets, to achieve the systematic formal verification of the OCL constraints.

The remainder of this paper starts with a brief expose on the UML statecharts translation to Petri nets; that, 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. Section 5 presents the OCL invariant translation to LTL properties. We then, motivate our work and show its novelty and relevance versus related works. We conclude with some observations on the obtained results and recommendations on future research direction.

2. Background

We summarize in this section the work that we presented in [5] to derive UML statecharts to color Petri nets. This work supports the approach that we are developing in the present paper.

2.1. Statecharts

A statechart describes the behavior of a class in terms of states and exchanged messages with other classes' statecharts. A state is composed of two atomic actions (at its entry and its exit) and one activity. 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. Due to their atomicity, the entry exit and transit actions are in fact generated events called: *entry*, *exit* or *transit* events, respectively, as shown in Figure 1.

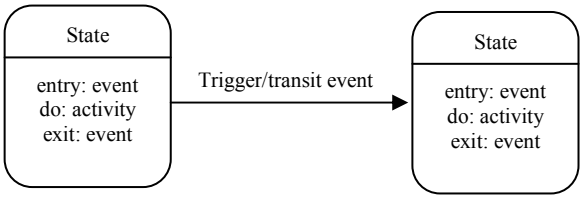

Figure 1. Statechart's events and activity.

The event is of two types: *send* event and *call* event. These events are mentioned on the statechart as follows: *«send» class () and «call» operation ()*. Examples of these events are given in the case study, as shown in Figure 3.

More formally, the statechart can be defined by the 5-tuple <*S, Act, Tr Gen, Trg*> where:

- $S = \{s_1, s_2, \ldots, s_n,\}$ is a set of states.
- $Act = \{act1, act2,\ldots, actn\}$ is a set of activities.
- $Tr = \{tr1, tr2, \ldots, trn\}$ is a set of transitions.
- $Gen = \{gen1,gen2,\ldots,genn\}$ is a set of generated events.
- $Trg = \{trg1, trg2,\ldots, trgn\}$ is a set of event triggers.

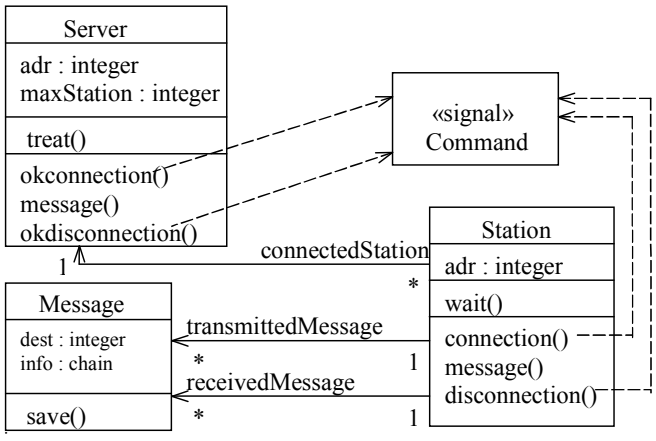

Figure 2. Class diagram of the message server application.

We illustrate our study through a message server application where the main role of the server 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 represented in Figure 2, where the server is modelled by the *Server* class and the stations by the *Station* class and the exchanged messages by the *Message* class.

Figure 3 presents the statechart of a station which can, at all times, connect itself from the server. Its connection request is realized using the «*send*» *connection* event. The server confirms the station connection using the «*send*» *okconnection* events. When connected, a station can notify a message, receive a message or disconnect itself. It notifies by means of the «*send*» *message* event. After it receives a forwarded message from the server by means of the «*send*» *message* trigger, it is saved by using the «*call*» *save* event. Its disconnection is requested by the «*send*» *disconnection* event and confirmed by the «*send*» *okdisconnection* event.

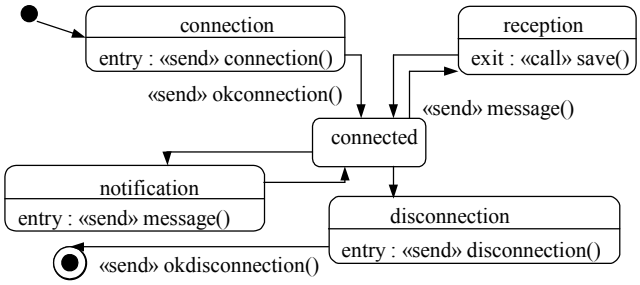

Figure 3. Statechart of the station class.

2.2. Derivation Approach

In [5], Petri nets have been selected as target formalism. We defined them by the 7-tuple <$P, T, A, C, Pre, Post, M_0$,> where:

- $P = \{p1, p2, \ldots, pn\}$ is a set of places.
- $T = \{t1, t2, \ldots, tn\}$ is a set of transitions.
- $A \subseteq P \times T \cup T \times P$, is a set of arcs.
- $C = \{C1, C2, \ldots, Cn\}$ is a set of colours.
- $Pre: P \times T \rightarrow P(C)$ is a precondition function to the transition firing such that $Pre\ (pi, ti) = \{C1, C2\ldots, Ck\}$.
- $Post: P \times T \rightarrow P(C)$ is a postcondition function to the transition firing such that $Post\ (pi, ti) = \{C1, C2\ldots, Ck\}$.
- $Mo: P \rightarrow C$ *is the initial marking function.*

The derivation process is based onto an object-oriented approach. Each statechart modelling interactive class behaviour is transformed to an object subnet called Dynamic Model *(DM)* (see Figure 4). To construct the *DM*, each state $s \in S$ is converted to a place $p \in P$ and each transition $tr \in T$r is converted to a transition $t \in T$.

To deal with Petri net simulation, we tackle 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. Associated to the *DM*, the places *Object*, *Scenario* and *Input*, constitute an Object Petri Net model that we call *OPN*. To connect the different *OPNs*, we use the *Link* place through which all the exchanged messages should pass. Thus, for each *OPN*, a directed transition from the *Link* place to the *Input* place is built. This transition is fired by the events that trigger on the statechart.

As for the generated events on the statechart, they are converted to arcs from the *Scenario* place to the transition to which they are related. Then, they are converted to arcs from this transition to the *Link* place.

Figure 5 summarizes the translation of the different constructs of the statechart into their homologues in Petri nets. The dashed symbols represent other constructs than the ones concerned by the translation. Figure 6 represents the Petri net resulting from the conversion of the statechart of the station class.

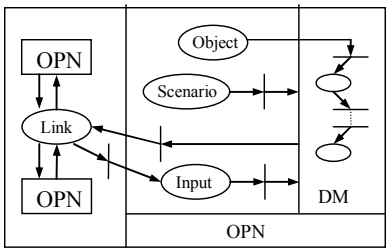

Figure 4. Petri nets interconnection architecture.

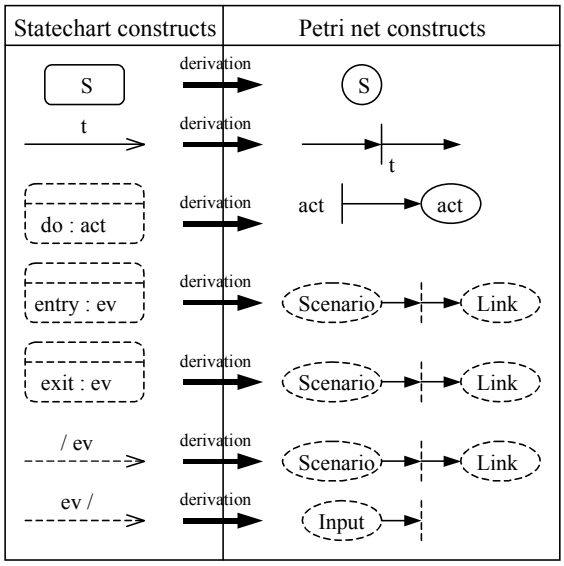

Figure 5. Petri net of the station class.

3. Initialization Technique

To deal with the model simulation, starting from the generic model which we derive from the statecharts, as shown in Figure 4, two types of arguments must be initialized, namely, the system's objects and the exchanged messages among these objects. Thus, we proceed two types of initialization that we call static and dynamic.

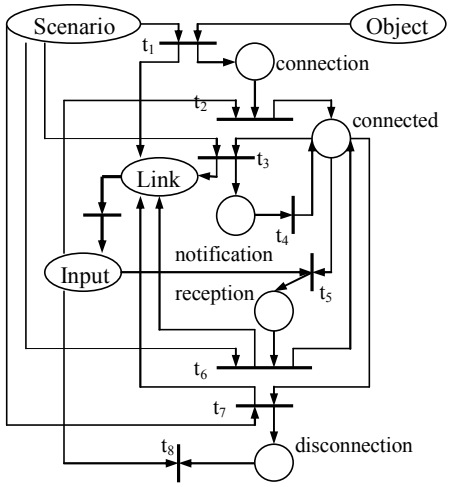

Figure 6. Petri net of the station class.

3.1. Static Initialization

We state for our approach requirements two types of objects: active and passive. The active objects interact exchanging passive objects. For example, in the server message application, the *Server* and *Station* objects are active while the *Message* object is passive.

Thus, an object is formalized by the colored token <*obj*, *attrib*> where *obj* designates its identity expressed according to the UML notation which identifies an object by its name and its class name as follows: *object:class*. As for *attrib*, it designates the set $\{attrib_1,\ldots attrib_k\}$ of the object's attribute values.

The objects and their attribute values are specified on the object diagrams. So, all the objects instantiated from the same class on the object diagram, are inserted in the *Object* place of the *OPN* translating the class's statechart. Figure 7 shows an example of the object diagram of the message server application before any action (there is no links between the objects). For each station, the IP address is given.

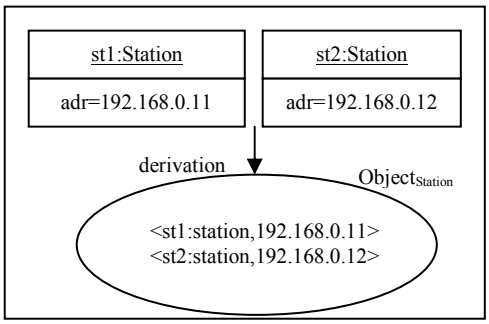

Figure 7. Object initialization.

3.2. Dynamic Initialization

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 messages reflect events specified with their attribute values, as follows: *«send» object:class(attrib)*, *«call» operation(attrib)* as shown in Figure 8. This specification permits the identification of the events that are dynamically generated on the statechart.

The sequence diagram of Figure 8 shows a scenario related to the server message application presented in section 2.1. Two stations *St1* and *St2* request a connection from a server *S*. When done, *St1* transmits a message *M1* which is forwarded by *S* to *St2*. After that, *St1* is disconnected.

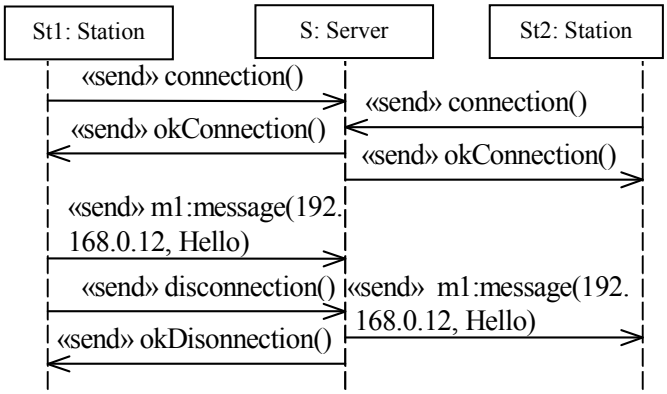

Figure 8. A scenario from the message server application.

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

The events are at the basis of three different types of initialization, depending on their provenance and the nature of the action which they produce: *home* events generated by the model's objects, *border* events generated from the system environment or *create /destroy* events used to create or destroy objects. The latter may be *home* or *border* events. The *home* events are grouped together per class, so that for each object only the output events are retained. Once converted to tokens defined by $<ev^{(Sc)}, srce^{(Sc)}, targ^{(Sc)}, xobj^{(Sc)}, attrib^{(Sc)}>$, the events are stored in the *Scenario* place of the *DM* corresponding to the class. Through this initialization, the *Scenario* place animates the Petri net with the event occurrences. Their *source* object being not represented on the model, all the *border* events are directly stored in the *Link* place which is common to all classes, as tokens of the form: $<ev^{(Li)}, srce^{(Li)}, targ^{(Li)}, xobj^{(Li)}, attrib^{(Li)}>$. This permits the opening of the Petri net model 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:

Scenario = <«*send*» connection, *st1*: station, *s*:server, connection,> + <«*send*» connection, *st2*:station, *s*:server, connection> + <«*send*» message, *st1*:station, *s*:server, *m1*:message, 92.168.0.12, Hello> + <«*send*» disconnection, *st1*:station, *s*:server, disconnection>.

4. System Property Validation

The model checking is based on the state space generation and the verification and validation of LTL properties on this space. The verification tackles the well construction of the model, using generic properties as deadlock, livelock, reject states, quasi-liveness, boundedness and reinitializability. All these properties are automatically verified by the model checker PROD. As for the validation, it checks whether the model is constructed conforming to the customer initial requirements. For this purpose, specific properties of the system, written by the modeler, are used.

Since the main motivation of this work is that the UML designer may reach a valid modeling without needs for knowledge of formal techniques, it is only reasonable that the system properties are expressed by the modeler in the OCL language and are automatically translated into LTL logic.

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 properties (whose expression is essentially made of these constructs) can be verified. This lead us to the necessity of introducing the association end modelling onto the statecharts in order to get after their transformation, the equivalent Petri net constructs. This object flow modelling is realized by means of the link actions. However, the usefulness of the link actions does not concern explicitly the modelling of the object life cycle. When constructing his diagrams, the designer does not necessarily think to model 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 modelled by the designer, but the addition of the connected station to the association end is usually omitted from the modelling, see Figures 3 and 9. That is why we recommend specifying the link actions on the statechart 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.

UML action semantics was defined in [18] for model execution and transformation. It is a practical framework for formal descriptions. For this work, we

are particularly interested in the *create* link, and *destroy* link actions. 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. These actions will be represented on the statechart as tagged values of the form *{linkAction(associationEnd)}*, following the event which provokes the association end update.

On Figure 9, after confirmation of its connection or disconnection («send» *okconnection* or «send» *okdis-connection*), the station adds or removes itself from the association end *connectedStation*, using *{createLink(connectedStation)}* or *{destroyLink (connectedStation)}*. It adds a notifed or received message with *{createLink(transmittedMessage)}* or *{createLink(receivedMessage)}*,respectively.

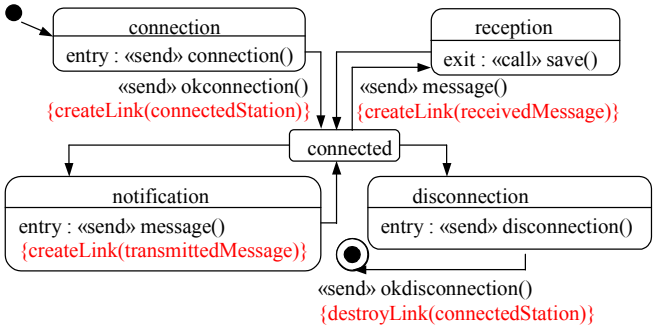

Figure 9. Statechart of the station class with link actions.

The link actions may concern an active or passive 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 *modularity*, association end should appear and be manipulated in only one statechart. In Petri nets, the association end is translated in a place of *role* type. This place holds the name of the association end and belongs to the *DM* translating the statechart.

Furthermore, an association end regrouping active objects must be updated within the statechart of these objects' class, 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 and are not specified by dynamic models. So, the association end representing them could be updated in the statechart 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 active objects.

In Petri nets, the *create* link action is semantically equivalent to an arc from the transition related to the association end update towards the place specifying the association end, adding an object within. This is formalized by Algorithm 1 in section 4.1. The *destroy* link action is semantically equivalent to an arc from the association end place to the transition corresponding to the link action, removing an object, see Algorithm 2 in section 4.2.

The object to be added-to /removed-from the association end is extracted from the components of the token (whose global form is *<ev, srce, targ, xobj, attrib>)* 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. The added /removed object may be the source object (*src*) or the exchanged object (*xobj*) if the link action follows a generated event. It is the target object (*targ*) or the exchanged object (*xobj*) if the link action follows an event trigger.

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 /removed from the association end and *assoc* is the object at the opposite end.

Considering the new treatments that we introduce on the statechart, we propose to complete its syntax as follows: *<S, Act, Tr Gen, Trg, O, U, R, r>*, where:

- $O = \{o1, o2 \ldots, on\}\ n \in \angle$, is a set of active objects.
- $U = \{u1, u2 \ldots, un\}$ is a set of exchanged objects.
- $R = \{r1, r2 \ldots, rn\}\ n \in \angle$, is a set of association ends.
- $ri = \{y1, y2, \ldots, yk\}$ is a set of objects of the association end *ri*, $yk \in O \ \vee\ yk \in U$.

4.1. Algorithm 1

A. Conversion of an association end

For each $r_i \in R$, *create* $rol_i \in P$

B. Conversion of a createLink() action

For each createLink(r_i), $r_i \in R$, *after a generated event : create an arc* $t_p \rightarrow rol_i \in T \times P$, *such that :*

- *if* $ri \subset O$: $Post(rol_i, t_p) = <targ, srce, attrib>$
- *if* $r_i \subset U$: $Post(rol_i, t_p) = <srce, xobj, attrib>$

For each createLink$_i(r_i)$, $r_i \in R$, *after an event trigger : create an arc* $t_p \rightarrow rol_i \in T \times P$, *such that :*

- *if* $ri \subset O$: $Post(rol_i, t_p) = <srce, targ, attrib>$
- *if* $r_i \subset U$: $Post(rol_i, t_p) = <targ, xobj, attrib>$

An example on the translation of a *create* link action is presented in Figure 10.

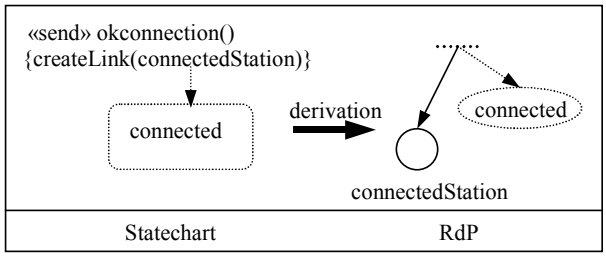

Figure 10. Conversion of a create link action after an event trigger.

4.2. Algorithm 2

The conversion of the *destroyLink()* action is treated in a similar manner as the *createLink()* action, applying the only two following changes:

- The arc incoming 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, as shown in Figure 11.

Figure 11 shows the transformed statechart of the station class considering the link actions.

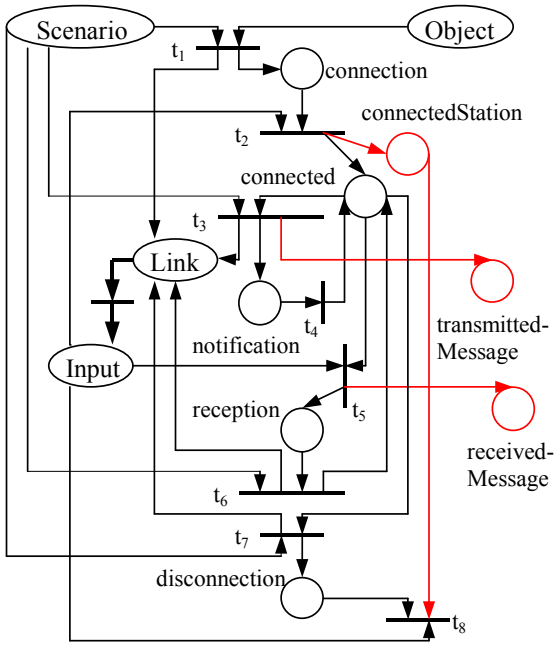

Figure 11. Petri net of the station class with link actions.

5. Mapping OCL Invariants to LTL Logic

An OCL invariant is a stereotyped constraint that must be true for all instances at any time. In general, it is given using the global expression:

Context Object: Class inv: OCL-expr

Where the *context* keyword introduces the classifier on which the expression is evaluated. The verification is performed on all instances. The keyword *inv* denotes the stereotype «invariant» which means that the constraint will be verified on all states of the system. It is followed by the OCL expression ocl-expr which specifies the condition to be verified.

PROD supports LTL logic. LTL formulas express properties of a linear system behavior on more than one state. The LTL PROD grammar that we retain to build a formula *f* is given by:

f: = prod-expr | not f | f and f | f or f | f implies f | henceforth f | eventually f

where henceforth means always and eventually i.e., exists.

For OCL invariants, the condition is entirely evaluated on each state of the system. The time precedence which involves the property evaluation on more than one state is not supported by OCL. In other words, when mapping an OCL invariant to LTL logic, the sole potential used operator is *always*.

In order to better exploit the LTL logic and permit the expression of more properties, we propose to introduce optionally in the OCL invariant, the keyword *will* which means that the condition will be verified in the future. So, the new form of the OCL invariant is:

Context Object: Class inv : ocl-expr [will ocl-expr]

We note that *T:OCL invariant→LTL property is* the translation function that transforms an OCL invariant to LTL property. It is written for each object of the context as follows:

T (Context object:class inv : ocl-expr [will ocl-expr]) = henceforth (T(ocl-expr) [eventually T(ocl-expr)]).

T (ocl-exp) gives a predicate of first-order logic independent of temporal constraints, namely *prod-expr*. We define *prod-exp* according to the following PROD grammar:

prod-expr → prod-expr op prod-expr
| marking ':' field-form
| 'card(' marking ')' | expression
field-form → field-expr | field-expr log-op field-expr
field-expr → 'field[' comp ']' rel-op 'field[' comp ']'
| 'field[' comp ']' rel-op cstvar
expression → marking | cstvar
op → rel-op | log-op | math-op
rel-op → '==' / '!=' | '<' / '>' / '<=' / '>=' / '<' | '>'
log-op → '&&' / '||'
math-op → '+' / '-' | '&' | 'or'

where *marking* is the place marking, *comp* is the component number of the tuple, and *cstvar* is a constant or a variable.

To translate the OCL expressions, we rely on the metamodel of Figure 12.

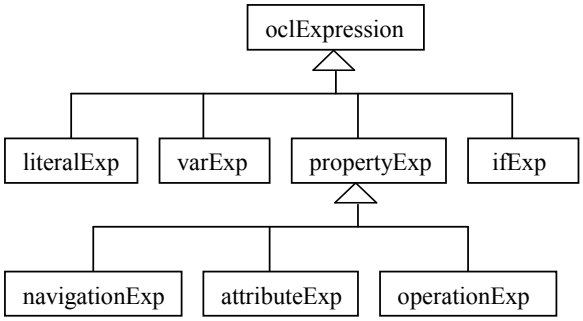

Figure 12. OCL expression metamodel.

A *literalExp* is an expression whose value is

identical to the expression symbol. This includes constants like the integer 1 or literal strings like 'this is a LiteralExp'. This expression is unchanged when translated into PROD syntax.

A *variableExp* is modeled in Petri nets using a place and its value is rendered by the number of tokens in the place such that: T (variable) = card ($place_{variable}$) – 1.

A navigationExp is a reference to an association end defined in a UML model. It is used to determine for an object, the collection of its linked objects. The object is matched with the association end using a '.' as follows: *object.associationEnd*. As seen in section 4, an association end is translated in Petri nets using a place of role type, with the name of the association end. The tokens of this place are of the form <*assoc, obj, attrib*$_1$ *..., attrib*$_n$>, where *assoc* is the object linked to the collection including the object *obj*. The expression is translated for each object of the context by:

T (object.associationEnd) = place$_{associationEnd}$*: field [0] == object, where the symbol ':' introduces a condition, and field [0] designates the first component of the tuple of the place associationEnd.*

An *attributeExp* is a reference to an attribute of a classifier defined in a UML model. It may be applied to the objects of the contextual class using the expression *object.attribute*. The translation of this expression gives for each contextual object (which may be at any place of the *DM* from which we exclude the role places and note *DM**):

T (object.attribute) = ∪ (place$_{DM*class}$*: field [0] == object: field [attributeNumber]) where attributeNumber is the attribute number in the tuple that specifies the object. We recall that the tokens of the DM** *places are of the form :< obj, attrib1,..., attrib*$_n$>.

An *operationExp* refers to two categories of operations. The first consists of the usual logical and mathematical operations applied to OCL expressions. The second concerns predefined OCL operations applicable to collections of objects. The translation of the logical and mathematical operations is given by:

T(ocl-expr log-op /math-op ocl-expr) = T(ocl-expr) T(log-op /math-op) T(ocl-expr)

The operations on collections are of the form collection→operation. Their translation is given by the Table 1. For short, we replace object.associationEnd by *col*.

An *IfExp* is of the form *if* if-ocl-expr *then* then-ocl-expr *else* else-ocl-expr. It is translated to:

T(if-ocl-expr) implies T(then-ocl-expr) or T(! if-ocl-exp) implies T(else-ocl-expr)

We propose in what follows, to express two properties extracted from the message server application. These properties are first expressed into a paraphrased (textual) form. They are after, specified as OCL invariants and finally translated into LTL properties. To make easier the comprehension of the properties refer to the class diagram of the server message application on Figure 2.

Table 1. Mapping OCL operations to temporal logic formulas.

OCL Operations	**Temporal Logic Formulas**
col.→size()	card(T(col.))
col.→isEmpty()	(T (col.)) == empty
col.→notEmpty()	(T(col.)) != empty
col.→union(col.2)	T (col.) + T (col.2)
col.→intersection (col.2)	T (col.) & T (col.2)
col.→including(object)	T (col.) + T (object)
col.→excluding(object)	T (col.) - T (object)
col.→count(object)	card(T(col.) : field[0/1]* == object)
col.→includes(object)	(T(col.) : field[0/1]* == object) != empty
col.→excludes(object)	(T(col.) : field[0/1]* == object) == empty
col.→includesAll(col.2)	T (col.) >= T (col.2)
col.→excludesAll(col.2)	(T (col.) & T (col.2)) == empty
col.→select(ocl-expr)	T (col.) : T (ocl-expr)
col.→reject(ocl-expr)	T (col.) : T (! ocl-expr)
col.→exists(ocl-expr)	(T (col.) : T (ocl-expr)) != empty
col.→one(ocl-expr)	card(T(col.) : T(ocl-expr)) == 1

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*) <= (*place$_{DM*server}$*: field [*2*]))*
where *field[0]* designates the first component (assoc) of the connectedStation's token, and *field[2]* designates the third component ($attrib_2$ = maxStation) of the tokens of DM* of the server.

Property 2
While a station *r* is connected, it receives all the messages that are transmitted from a station *t*.

Property 2 Expression in OCL
Context station *inv*: s.connectedStation→*includes(*r*) and* t.transmittedMessage→*includes (*msg*) implies will* r.receivedMessage→*includes (*msg*)*

*field[0/1] == object means that if the object comes from a collection that models class instances, it is translated in a token of the form <*obj, attrib*$_1$*, ..., attrib*$_n$*> and so, we write field[0] == object. Otherwise, the object belongs to an association end, it is modeled by a token of the form <*assoc, obj, attrib*$_1$*..., attrib*$_n$*> and so, we write field [1] == object. Likewise, T (object) = <*obj, attrib*$_1$*, ..., attrib*$_n$*> if object models a class instance. It is equal to <*assoc, obj, attrib*$_1$*, ... , attrib*$_n$*> if object models an association end.

Property 2 Expression in PROD
For each connected client *r*, each transmitter *t* and each transmitted message *msg* write the property:

*# verify henceforth ((*connectedStation *: field[0] ==* s *&& field[1] ==* r*) != empty && (*transmitted Message *: field[0]==*t *&& field[1] ==* msg*) != empty implies eventually (*receivedMessage : *field[0] ==* r *&& field[1] ==* msg*) != empty)*
where *connectedStation: field[0]==s && field[1]==*r designates the 1st and 2nd components of the *connectedStation*'s tokens, *transmittedMessage:field[0]==t && field[1]==* msg designate the 1st and 2nd components of the *transmittedMessage*'s tokens, and receivedMessage: *field[0]==r && field[1]==* msg designate the 1st and 2nd components of the *receivedMessage*'s tokens.

6. Contribution Versus Related Work

Formalization of UML statechart semantics [12, 22, 24] and integration in the statecharts of languages state-oriented [2, 14] or property-oriented [2, 21] are widely investigated in the research area. The OCL language has also been integrated within the statecharts in various works, in particular, those of Flake [8, 9] who extends it with temporal logics to express properties over time. However, through our multiple investigations, we have never encountered works that tackle the integration within the statecharts, of the object movement in the association ends. This formalization allows the use of the OCL navigation expression and OCL attribute expression to formalize and then, validate the object flows.

Translation of OCL invariant in other formalisms such as Object-z [20], B [13], first-order predicate logic [3] or object-based temporal logics [7], allows the validation of the system properties using the target formalism. Other works tackled OCL invariant extension with temporal operations [6, 8]. As far as we are concerned, we first, extend the OCL invariants with a temporal operator in order to benefit from all capabilities of the target logics. We after, automate the OCL property translation to the temporal logics expressed in PROD syntax. The relevance of such a mapping is of a practical nature. It presents the merit of providing a specific translation that takes the PROD tool's characteristics into account. This automated translation also, spares the designer the hard effort of specifying using unknown languages.

Sequence diagrams are generally combined with the statecharts in order to connect the object life cycles [4, 25]. They are also, transformed separately in other formalisms to validate specific scenarios [10] or composed together to describe the system's overall behaviour [23]. As far as we are concerned, we introduce a novel use of the sequence diagram exploiting it to animate the modeling with the events of the scenario that will be verified.

On the other hand, data formalization is usually given by means of state-oriented languages as Z or B [1, 2]. We propose to specify data using the object diagrams. This data provides the Petri net initial marking with objects named by identities and attribute values.

7. Conclusion

This paper presents an approach to systematically validate the UML modeling without 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, thanks to the system awaited properties which are expressed by the modeler in OCL language and then automatically translated into LTL properties. To efficiently deal with the property validation, we propose introducing an object flow specification into the object's control flow model (statechart), using predefined actions on the association ends.

Among the prospects of this work, the analysis of the validation /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] Amálio N. and Polack F., "Comparison of Formalization Approaches of UML Class Constructs in Z and Object-Z," *in Proceedings of the International Conference of Z and B Users, LNCS*, vol.2561, 2003.

[2] Attiogbé C., Poizat P., and Salaun G., "Integration of Formal Datatypes within State Diagrams," *in Proceedings of Fundamental Approaches to Software Engineering (FASE'2003), LNCS,* vol. 2621, 2003.

[3] Beckert B., Keller U. and Schmitt P., "Translating the Object Constraints Language into First-order Predicate Logic," *in Proceedings of Verify, Workshop at Federated Logic Conferences*, Copenhagen, 2002.

[4] Bernardi S., Donatelli S., and Merseguer J., "From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models," *in Proceedings of the Third International Workshop on Software and Performance*, Rome, Italy, *ACM Press*, pp. 35-45, 2002.

[5] Bouabana-Tebibel T. and Belmesk M., "Formalization of UML Object Dynamics and Behavior," *in Proceedings of the 2004 IEEE*

International Conference on Systems, Man & Cybernetics, Netherlands, 2004.

[6] Cengarle JM. and Knapp A., "Towards OCL/RT," *in Proceedings of Formal Methods: Getting IT Right*, *LNCS*, vol. 2391, pp. 389-408, 2002.

[7] Distefano D., Katoen J., and Rensink A., "On a Temporal Logic for Object-Based Systems," *in Proceedings of the 4th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOOD'2000)*, Stanford, USA, 2000.

[8] Flake S. and Mueller W., "Past- and Future-Oriented Temporal Time-Bounded Properties with OCL," *in Proceedings of the 2nd International Conference on Software Engineering and Formal Methods*, China, IEEE Computer Society Press, 2004,

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

[10] Harel D., Kugler H., and Pnueli A., "Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements," *in Proceedings of the Formal Methods in Software and System Modeling, LNCS*, vol. 3393, pp. 309-324, 2005.

[11] Jensen K., *Coloured Petri Nets*, Basic Concepts, Springer, vol. 1, 1992.

[12] Kuske S., "A Formal Semantics of UML State Machines Based on Structured Graph Transformation," *in Proceedings of UML: The Unified Mode-ling Language, Modeling Languages, Concepts and Tools, LNCS*, vol. 2185, pp. 241-256, 2001.

[13] Marcano R. and Lévy N., "Transformation Rules of OCL Constraints into B Formal Expressions," *in University of Versailles Saint-Quentin-en-Yvelines eds.*, May 2002.

[14] Meyer E., "Développements Formels Par Objets: Utilisation Conjointe De B Et D'uml," *PhD Thesis*, University of Nancy 2, France, 2001.

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

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

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

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

[19] PROD 3.4, *An Advanced Tool for Efficient Reachability Analysis*, Laboratory for Theoretical Computer Science, Helsinki University of Technology, Espoo, Finland, 2004.

[20] Roe D., Broda K., and Russo A., "Mapping UML Models Incorporating OCL Constraints into Object-Z," Imperial College, *Technical Report*, no. 2003/9, 2003.

[21] Royer J., "Temporal Logic Verifications for UML, the Vending Machine Example," *RSTI - L'objet*, *4th Rigorous Object-Oriented Methods Workshop*, vol. 9, no. 4, 2003.

[22] Truong N. and Souquières J., "Verification of Behavioral Elements of UML Models Using B," *in Proceedings of the 20th Annual ACM Symposium on Applied Computing*, USA, 2005.

[23] Uchitel S., Kramer J., and Magee J., "Synthesis of Behavioral Models from Scenarios," *IEEE Transactions on Software Engineering*, vol. 29, no. 2, pp. 99-115, 2003.

[24] Varro D., "A Formal Semantics of UML Statecharts by Model Transition Systems," *in Proceedings of the 1st International Conference on Graph Transformation,* Spain, 2002.

[25] Ziadi T., Hélouët L., and Jézéquel J., "Revisiting Statechart Synthesis with an Algebraic Approach," *in Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, ACM, Edimburgh, UK, pp. 242-251, 2004.

Thouraya Bouabana-Tebibel obtained the BSc degree in computer science from Houari-Boumédième Technology and Science University (USTHB), Algeria and the MSc degree in industrial engineering from Polytechnic National School, Algeria, and the PhD degree in computer science from the USTHB University. She works as an associate professor in the National Institute of Computer Science, Algeria. She is also a Cisco instructor in the network area. Her research interests include object-oriented specification in particular UML and object Petri nets, simulation, validation, and verification of interactive systems.

Mounira Belmesk obtained the BSc degree, the MSc degree and the PhD degree in computer science from Houari-Boumédième Technology and Science University, Algeria. She is now a professor at Edouard MonPetit College, Canada.