Journal of Computing and Information Technology - CIT 15, 2007, 2, 171–183
doi:10.2498/cit.1000882

Roles at the Basis of UML Validation

Thouraya Bouabana-Tebibel

National Institute of Computer Science, Algiers, Algeria

Formal validation of UML models proves to be hardly realizable, due to the imprecise semantics of UML dynamic diagrams. To remedy that, we first present a technique for transforming UML statecharts into Petri nets. We develop afterwards, an approach based on the movement of the objects throughout the roles they play. This approach allows validation of the temporal logic properties translated from the OCL invariants, on the Petri nets derived from the UML models. System property validation is realized thanks to a prior initialization of the objects and exchanged messages between the communicating objects. A case study is given to illustrate the methodology.

Keywords: UML, OCL, Petri nets, LTL, CTL, validation

1. Introduction

UML [19] suffers from continuing criticism on the precision of its semantics at the time the verification of the model correctness has become a key issue. UML 2.0 [18] brings more precision to its semantics, but it remains informal and lacks tools for automatic analysis and validation. We presented in [6] a methodology to automatically transform UML models into Petri nets [13] 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 [22], a model checker tool for predicate/transition nets. Model checking is classified as the most appropriate technique for verifying UML dynamic models [3],[11],[17]. It allows a fast and simple way to check whether the property holds or not. To avoid the high learning cost of the model checker, we suggest that the designer specifies the system properties in OCL, the Object Constraint Language [20] which is part of UML. OCL permits the formulation of restrictions over UML models, in particular, invariants. We automate the translation of these invariants to temporal logic properties so that they can be verified by PROD during the Petri net analysis.

The invariants are specified on class diagrams which 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 objects playing a given role at a given time.

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, OCL invariants refer to association ends to evaluate their expressions. 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 so that the invariants can be adequately verified by PROD. In other words, he is called on to specify the association end update using the link actions [21]. This provides the dynamics of the object throughout the roles it plays. As far as we are concerned, in addition to the OCL invariant translation into LTL and CTL 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 the paper starts with a brief overview on the mapping of UML models into Petri nets. In Sections 4 and 5 the proposed approach is presented and the techniques upon which it is based are developed. These techniques are illustrated throughout the paper, using the case study of Section 3. Section 6 presents the OCL invariant translation into LTL and CTL properties. Examples on the translation of the system properties are presented in Section 7 and some results on the model analysis are given and commented in Section 8. We provide in Section 9, the novelty and relevance of our work versus related works. We conclude with some observations on the obtained results and recommendations for future research direction.

2. Background

We summarize in this section the work that we present in [6] to transform UML statecharts into coloured Petri nets. This work supports the approach that we develop in the present paper.

2.1. Statecharts

A statechart describes the behaviour of a class in terms of states and messages it exchanges with other 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 respectively called: *entry*, *exit* or *transit* events, see Figure 1.

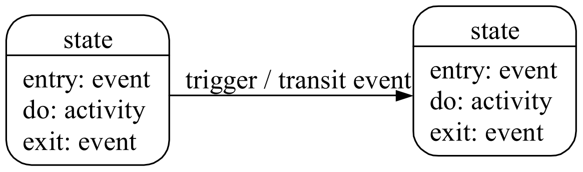


Figure 1. Statechart's events and activity.

The event is of two types: *send* event and *call* event. These events are mentioned on the statechart as follows: *"send" class(), "call" operation()*. Examples of these events are given in the case study of Figure 5.

2.2. Petri Nets

Petri nets have been presented in several works [2], [5] as a suitable formalism for translating the UML dynamic models. Both of them are classified as a state-transition system dedicated to the object life cycle modeling. Their syntax and semantics can be easily and completely matched. We used them in [6] to transform the statechart diagrams. We defined them by the 5-tuple $<P, T, A, C, M_0,>$ where:

- $P = \{p_1, p_2, \ldots, p_n\}$ is a set of places.
- $T = \{t_1, t_2, \ldots, t_n\}$ is a set of transitions.
- $A \subseteq P \times T \cup T \times P$, is a set of arcs.
- $C = \{C_1, C_2, \ldots, C_n\}$ is a set of colors where $C_i = \{\langle c_1, c_2, \ldots, c_k \rangle\}$ and c_j is a variable or a constant.
- $M_o : P \rightarrow C$ is the initial marking function, such that $M_o(p_i) \sum_{k=1}^{K} C_k$.

2.3. Derivation Approach

The derivation process is based on an object-oriented approach. Each statechart modeling an interactive class behaviour is transformed into an object subnet called Dynamic Model or *DM* (see Figure 2). To construct the *DM*, each state is converted to a place $p \in P$ and each transition is converted to a transition $t \in T$.

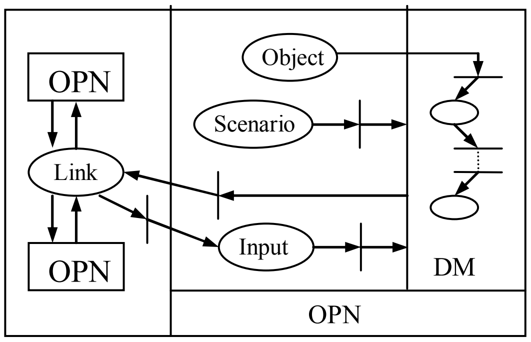


Figure 2. Petri nets interconnection architecture.

The events are modeled by tokens of *event* type. They are forwarded to the *DMs* by means of the *Input* place which constitutes an input interface of the *DM*.

Together 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 the exchanged messages should pass. Thus, for each *OPN*, a directed transition from the *Link* place to the *Input* place is built.

To deal with Petri net simulation, we address the Petri net initial marking regarding objects and exchanged messages. The marking regarding objects 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 marking in terms of messages 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.

Thus, each generated event on the statechart is converted to an arc from the *Scenario* place to the transition to which it is related and an arc from this transition to the *Link* place. As for the event trigger, it is converted to an arc from the link place to the transition on which it occurs.

Figure 3 summarizes the translation of the statechart constructs into their counterparts in Petri nets. The dashed symbols represent associated constructs not concerned by the translation.

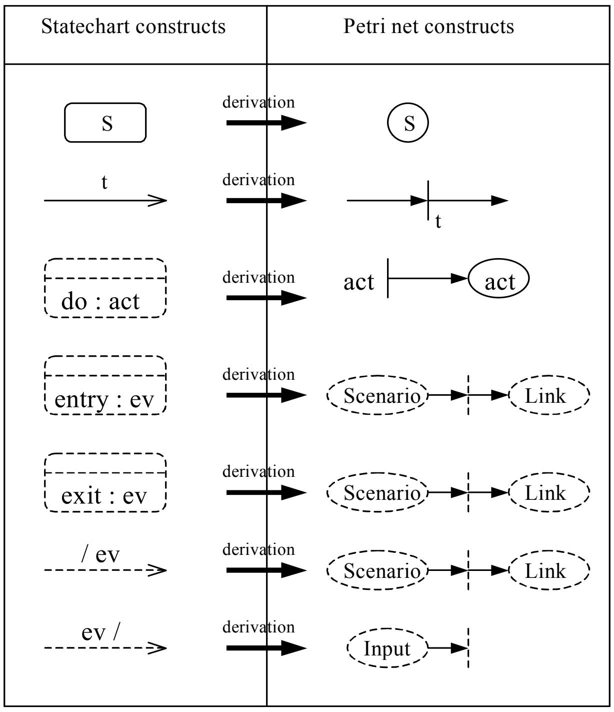


Figure 3. Mapping of UML constructs to Petri nets.

3. Case Study

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 4, where the server is modeled by the *Server* class, the stations by the *Station* class and the exchanged messages by the *Message* class.

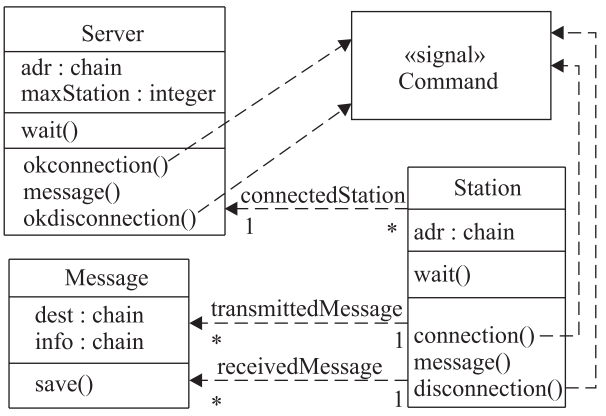


Figure 4. Class diagram of the message server.

Figure 5 presents the statechart of a station which can, at all times, connect itself to 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.

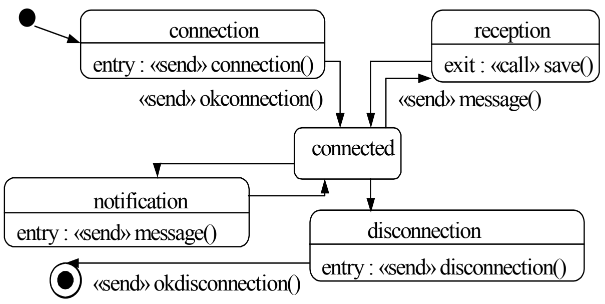


Figure 5. Statechart of the station class.

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

In Figure 6, we show the Petri net resulting from the conversion of the statechart of the station class.

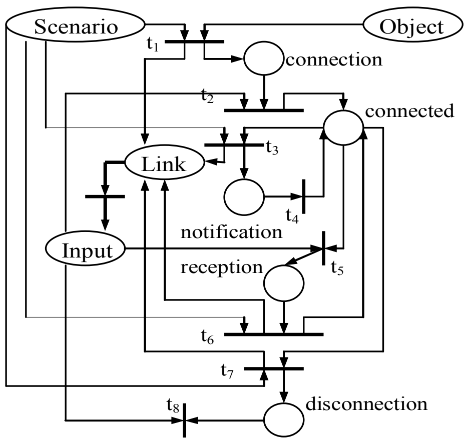


Figure 6. Petri nets of the station class.

4. Initialization Technique

To deal with the model simulation, starting from statechart diagrams, two types of arguments must be initialized, namely, the system's objects and the exchanged messages among these objects.

4.1. Object Initialization

We define 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.

The object identity is a main concern when formalizing the idea of objects. We adopt the UML notation which identifies an object by its name and its class name as follows: *object:class*. Thus, an object is formalized by the 2-tuple $(obj, attrib)$ where *obj* designates its identity and *attrib* the set $\{attrib_1, \ldots, attrib_k\}$ of its attribute values. It is modeled in Petri nets by the coloured token $<obj, attrib>$ to initialize Petri net marking in terms of objects.

The objects and their attribute values are specified on the object diagrams. These identified objects are used to initialize the *OPN* marking. This is realized by inserting all instances of the same class in the *Object* place associated to 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 are no links between the objects). For each station, the IP address is given.

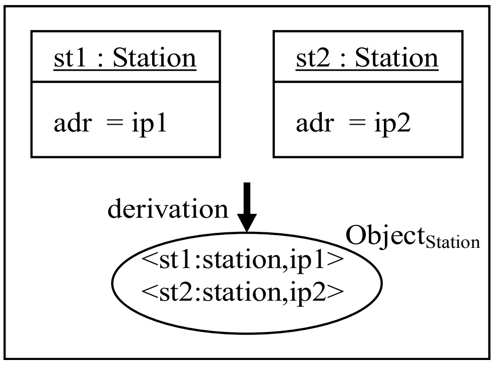


Figure 7. Object initialization.

4.2. Message 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" object:class(attrib)*, *"call" operation(attrib)*, see Figure 8. This specification permits the initialization of the events that are dynamically generated on the statechart.

The sequence diagram in Figure 8 shows a scenario related to the server message application presented in Section 3. Two stations *st1* and

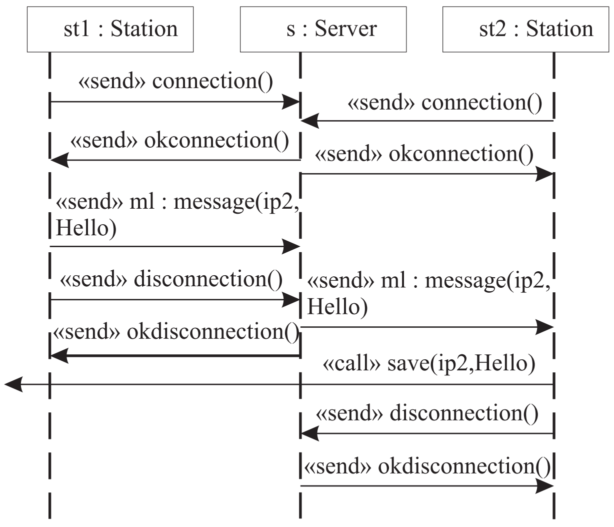


Figure 8. A scenario from the message server application.

st2 request a connection from a server *s*. When done, *st1* transmits a message *m1* and disconnects itself. *m1* is forwarded by *s* to *st2*. *st2* saves it. After this, *st2* is disconnected. We note that the arrow pointing from *st2:Station* with the message *"call" save(ip2,Hello)* goes out of the scope of the diagram towards the lifeline *m1:message* which is not represented in the diagram.

We formalize an interaction on a sequence diagram by the 5_ tuple (*ev, srce, targ, xobj/op, attrib*). The component *ev* identifies the event (*"send" class(), "call" operation()*). *Srce* and *targ* are respectively the source and the target object's identity. The component *xobj* gives the exchanged object's identity *(object:class)* if a *send* event or the called operation *op* if a *call* event. As for *attrib*, it designates the set $\{a_1, \ldots, a_k\}$ of the exchanged object attributes or the operation attributes.

The events are grouped together per class, so that for each object, only the output events are retained. They are converted afterwards to tokens defined by $<ev, srce, targ, xobj/op, attrib>$ and 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.

The transformation of the sequence diagram of Figure 8 gives the following *Scenario* place associated to the *OPN* of the *Station* class. This place contains tokens of the form $<ev, srce, targ, xobj/op, attrib>$ corresponding to the messages exchanged in the sequence diagram.

Scenario = <"send" connection(), st1:station, s:server, connection> + <"send" connection(), st2:station, s:server, connection> + <"send" message(), st1:station, s:server, m1:message, ip2, Hello> + <"send" disconnection(), st1: station, s:server, disconnection> + <"call" save(), st2:station, m1_message, save, ip2, Hello> + <"send" disconnection, st2:station, s:server, disconnection>.

5. System Property Validation

Verification by model checking as treated in PROD, is based on state space generation and verification and validation of LTL and CTL properties on this space. The verification tackles the good construction of the model, using generic properties as deadlock, livelock, reject states, quasi-liveness, boundedness or reinitializability. All these properties are automatically verified by PROD. As for 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, are used.

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 system properties are expressed by the modeler in the OCL language and are automatically translated afterwards into LTL and CTL.

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 Petri net specification so that the translated LTL and CTL properties (whose expression is essentially made of these constructs) can be verified. This requires the integration of the association ends onto the statecharts 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 usually omitted from the modelling, see Figures 5 and 9. That is why we recommend to the designer to specify the link actions on the statechart so that the OCL invariants can be verified.

UML action semantics was defined in [21] 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 the addition of 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 constraints 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(transmitted Message)*} or {*createLink(receivedMessage)*}, respectively.

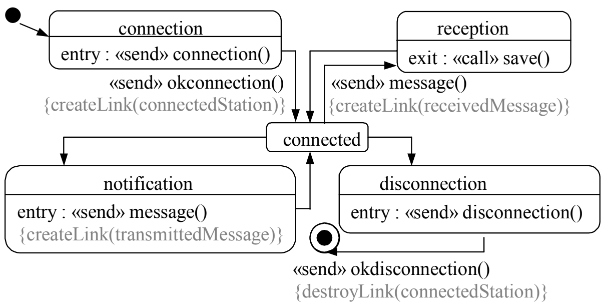

connection
entry : «send» connection()
«send» okconnection()
{createLink(connectedStation)}
reception
exit : «call» save()
«send» message()
{createLink(receivedMessage)}
connected
notification
entry : «send» message()
{createLink(transmittedMessage)}
disconnection
entry : «send» disconnection()
«send» okdisconnection()
{destroyLink(connectedStation)}

Figure 9. Statechart of the station class with link action specification.

The link actions may concern an active or passive (exchanged) end object. The object-oriented approach, on which both UML and Petri nets rely, is based on modularity and encapsulation principles. To deal with modularity, a given association end should appear and be manipulated in only one statechart. In Petri nets, the association end is modelled by 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 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. The exchanged objects are usually manipulated by the active 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.

The create link action is semantically equivalent to a Petri net arc going from the transition with the association end update towards the place specifying the association end. The destroy link action is semantically equivalent to an arc from the association end place to the transition corresponding to the link action, see Figure 10.

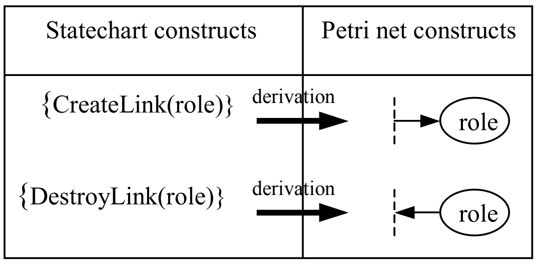


Figure 10. Translation of the link actions.

In Petri nets, the association end objects are coloured 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.

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

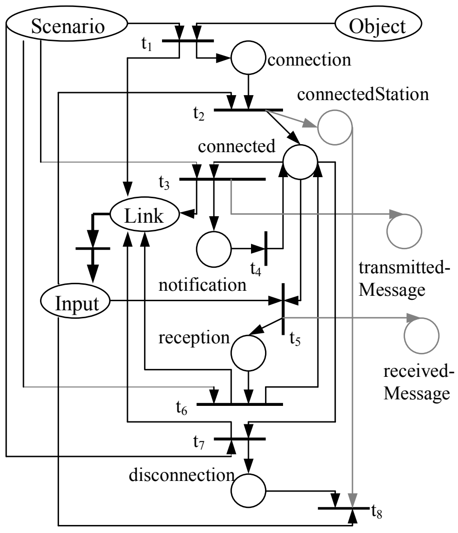


Figure 11. Petri net of the station class with link actions.

6. Mapping OCL Invariants to PROD Logics

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. A variable declaration may be used in the context. 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 both LTL and CTL logics. LTL and CTL are different regarding expressiveness: there are properties that can be specified in LTL, but not in CTL and vice versa. LTL formulas express properties of one possible system behavior. They are checked on the fly. CTL formulas express the set of all possible behaviors starting in a state. They are checked on all the state space. In LTL the future of a state in a run is inevitable, whereas in CTL a state usually has many different possible futures. Thus, generally speaking, CTL expresses possibility properties whereas LTL expresses properties that are inevitable.

For example, 'the system cannot deadlock' is a CTL property whereas 'the system will not deadlock' is an LTL property. However, many requirements can be specified both in LTL and CTL.

For OCL invariants, the condition is entirely evaluated on each state of the system. The temporal criterion which involves the property evaluation on more than one state before rendering a result is not supported. In other words, when mapping an OCL invariant to temporal logic, the only potential used operator is *always*. In order to better exploit PROD temporal logics and permit the expression of more properties, we propose to introduce optionally in OCL invariant two new operators. The first is the keyword *will* which means that the condition will be verified in the future (LTL property). The second is the keyword *can* that means that the condition will be verified in one of the possible futures (CTL property). So, the new forms of the OCL invariant are:

Context object:class *inv* : ocl-expr [*will* ocl-expr] and
Context object:class *inv* : ocl-expr [*can* ocl-expr].

In view of these new formulations, the invariants including *will* are translated to LTL properties whereas those with *can* are translated to CTL properties. If none of the keywords *will* or *can* appears on the invariant, the latter is translated both into LTL and CTL properties to be verified on the fly and on all the state space.

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 *not, and, or* and *implies* are logical operators and *henceforth* (i.e. always), *eventually* (i.e. exists) are temporal operators. We use the same grammar for the CTL PROD formulas replacing *henceforth* with *ag* and eventually with *ef*.

We note *T: OCL invariant→LTL/CTL property* the translation function that transforms an OCL invariant into a temporal logic property. The transformation yields the same results for both LTL and CTL properties, using respectively *henceforth/eventually* and *ag/ef*. It is written for each object of the context as follows:

T Context object:class *inv* : ocl-expr [*will/can* ocl-expr]) = henceforth/ag (*T*(ocl-expr) [eventually/ef T(ocl-expr)]).

T(ocl-expr) gives a predicate of first-order logic independent of temporal constraints, namely prod-expr. We define prod-expr 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. The symbol *Empty* designates an empty marking.

- comp is the component number of the tuple.

- cstvar is a constant or a variable.

To translate OCL expressions, we rely on the metamodel of Figure 12 which represents the different constructs of OCL invariants defined in [20]. This allows the covering, in a structured manner, of all these constructs.

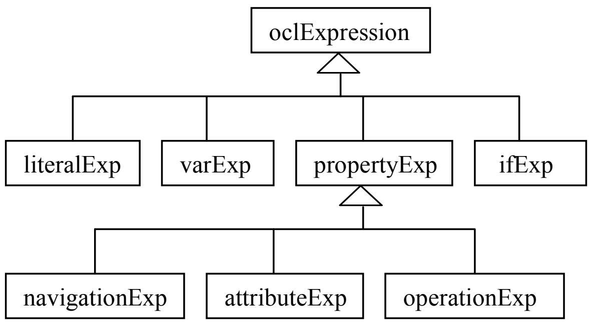


Figure 12. OCL expression metamodel.

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: $T(\text{variable}) = \text{card}(\text{place}_{variable}) - 1$

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 by using a '.' as follows: *object.associationEnd*. As seen in Section 4, an association end is translated into a place of role type, with the name of the association end. The tokens of this place are of the form $<assoc, obj, attrib_1, \ldots, attrib_n>$ where *assoc* is the object linked to the collection including the object *obj*. The expression is translated for each object of the context by:

$T(\text{object.associationEnd}) = \text{place}_{associationEnd} : \text{field}[0] == \text{object}$

where the symbol ':' introduces a condition and field[0] designates the first component of the tuple of the place associationEnd.

A collection may be also defined using the expression *class.allinstances* which gives all the instances of a class. In temporal logic this means that the condition is evaluated for each object of the class. Since these objects may be at any place of the dynamic model (from which we exclude the role places and note DM^*) during their life cycle, the condition is also verified for each place of the DM^*.

$T(\text{class.allinstances}) == \cup\ (\text{place}_{DM^*class})$

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^*):

$T(\text{object.attribute}) = \cup\ (\text{place}_{DM^*class} : \text{field}[0] == \text{object} : \text{field}[\text{attributeNumber}])$

where attributeNumber is the number of the component attribute within the tuple that specifies the object. We recall that the tokens of the DM^* places are of the form:
$<obj, attrib_1, \ldots, attrib_n>$.

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 Table 1. For short, we replace object.associationEnd and class.allinstances by *col*.

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(ocl.) : 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

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 $<obj, attrib_1, \ldots, attrib_n>$ and so, we write field[0]==object. Otherwise, the object belongs to an association end, it is modeled by a token of the form $<assoc, obj, attrib_1, \ldots, attrib_n>$ and so, we write field[1]==object.

Likewise, T(object) = $<obj, attrib_1, \ldots, attrib_n>$ if object models a class instance. It is equal to $<assoc, obj, attrib_1, \ldots, attrib_n>$ if object models an association end.

Table 1. Mapping OCL operations to temporal logic formulas.

An IfExp is of the form *if* if-ocl-expr *then* then-ocl-expr *else* else-ocl-expr. It is translated differently in LTL or CTL formulas. The LTL translation gives:

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

The CTL translation gives:

IfThen (T(if-ocl-expr), T(then-ocl-expr)) or
IfThen(T(! if-ocl-expr), T(else-ocl-expr))

7. System Property Translation

To illustrate the OCL translation into PROD logics, we present three properties covering a large spectrum of OCL expressions. These properties are first expressed into a paraphrased (textual) form, second, specified as OCL invariants and then translated into LTL properties. To make easier the comprehension of the properties, refer to the class diagram of the server message application (Figure 4).

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 place of the *DM** of the server s write the property:

# *verify henceforth*

*(card(*connectedStation : *field[0]* == s_server*)* <= *(*place$_{DM^*\ \mathrm{server}}$: *field[2]))*

where:

— *field[0]* designates the first component (*assoc*) of the *connectedStation*'s tokens,

— *field[2]* designates the third component ($attrib_2$ = *maxStation*) of the tokens of *DM** of the server.

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's tokens,

— "transmittedMessage: *field[0]* == st1_station" designates the 1st component of the transmittedMessage's tokens

Property 3

While a station st2 is connected, it receives all the messages that are transmitted from a station st1.

Property 3 expression in OCL

context station *inv* :

s.connectedStation$\rightarrow$*includes(*r) *and*

st1.transmittedMessage$\rightarrow$*includes(*msg*)* *implies will*

st2.receivedMessage$\rightarrow$*includes(*msg)

Property 3 expression in PROD

# *verify henceforth*

((connectedStation: (*field[0]* == s_server *&&*

field[1] == st2_station) != *empty) &&*

(transmittedMessage: (*field[0]* == st1_station *&&*

field[1] == m1_message) != *empty*) *implies eventually*

(receivedMessage: (*field[0]* == st2_station *&&*

field[1] == m1_message) != *empty*))

where:

— "connectedStation: *field[0]* == s_server *&& field[1]* == st2_station" designates the 1st and 2nd components of the connectedStation's tokens,

— "transmittedMessage: *field[0]* == st1_station *&& field[1]* == m1_message" designate the 1st and 2nd components of the transmittedMessage's tokens.

— "receivedMessage: *field[0]* == st2_station *&& field[1]* == m1_message" designate the 1st and 2nd components of the receivedMessage's tokens.

8. 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 [6]. We also developed a graphical interface for the construction of the statechart, class, object and sequence diagrams. These diagrams constitute the input of the translator whose outputs result into predicate/transition nets, specified in PROD syntax. A little part of the translated model is given in what follows, for the transition (t2) from the state *connection* to the state *connected*:

```
#trans t2
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.

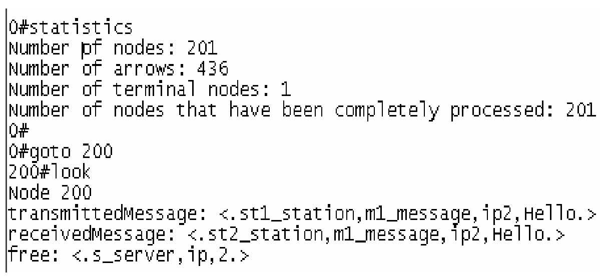
Figure 11 shows the transformed statechart of the station class considering the link actions.

PROD was executed afterwards to verify the models. The Petri net initial marking was defined by the object and sequence diagrams of Figures 7 and 8.

The generic properties concerning absence of livelock (infinite loops) and deadlock have been first checked. They were specified using 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 Petri net model. Only the places with tokens are represented. The others are empty. We notice that both the association end places *transmittedMessage* and r*eceived-Message*, 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's invariants were afterwards expressed in LTL properties and verified. Three of these properties are presented in Section 7. When executed with these properties, PROD program terminates without signalling errors.

9. Contribution vs. Related Work

Formalization of UML statechart semantics [14], [25], [28] and integration in the statecharts of languages state-oriented [1], [16] or property-oriented [1],[24] have been widely investigated. The OCL language has also been integrated within statecharts in various works, in particular, those of Flake and Mueller [9], [10] 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 statecharts. We can explain this, arguing that the UML/OCL association is rarely used to formally validate the UML models. When done, it is limited to OCL attribute expressions *AttributeExp* [26] or OCL pre and postconditions [9], [10]. Generally, the formalized UML models are rather coupled with formalisms for the expression of system properties. So, as long as navigation expressions are not used, the association end specification onto the object life cycle is not required. Otherwise, this specification provides after transformation into Petri nets, a formal basis for the validation of the translated invariants. Indeed, from Sections 5 and 6, we can easily see that the OCL navigation expressions (*navigationExp)* and OCL operation expressions (*operationExp* applied to navigation expressions) rely on the evaluation of the association end objects. So, although correctly specified and translated into LTL or CTL properties, they could never be validated on the derived Petri nets without association end specification on the statecharts. To deal with this integration, we proposed a technique based on the link actions.

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 statechart. However, this constraint is minimal compared to that of limiting OCL expressions or specifying using formal languages like temporal logics.

Translation of OCL invariants in formalisms such as Object-z [23], B [15], first-order predicate logic [4] or object-based temporal logics [8], is undertaken to precise OCL semantics. Other works tackle OCL invariant extension with temporal operations [7], [9]. As far as we are concerned, we first, extend the OCL invariants with temporal operators in order to benefit from all capabilities of the target logics. Second, we automate the OCL property translation into LTL and CTL logics expressed in PROD syntax. The relevance of such a mapping is of a practical nature. It presents the merit of providing a dedicated formal specification that is not limited to the generic constructs of temporal logic, but also takes PROD tool characteristics into account. Furthermore, the automated translation, spare the designer the hard effort of learning new formalisms as LTL and CTL.

Sequence diagrams are generally combined with the statecharts in order to connect object life cycles [5], [28]. They are also transformed separately in other formalisms to validate specific scenarios [12] or composed together to describe the system overall behaviour [27]. As far as we are concerned, we introduce a novel use of the sequence diagram exploiting it to simulate the models with the events of the scenario that will be verified.

10. 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 using the system properties which are expressed by the modeler in OCL language and then translated into LTL and CTL properties. To efficiently deal with the property validation, we propose to introduce an object flow specification into the object 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 then, their feedback to the user is explored. 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. So, the results must be presented to the designer in an interpreted form, where the error in models is simply and clearly pointed out.

Another prospect concerns large-scale systems that might explode the state space. For these systems, we propose to start the simulation at a critical moment from the object life cycle and not obligatorily from the initial state. To deal with this, the object diagram will be used to represent the system objects at this moment. This representation will have repercussions on the token distribution at the Petri nets initial marking.

References

[1] C. Attiogbé, P. Poizat, G. Salaun, Integration of Formal Datatypes within State Diagrams. *Fundamental Approaches to Software Engineering FASE'2003*, LNCS Vol. 2621, (2003), pp. 341–355.

[2] L. Baresi, M. Pezzè, On Formalizing UML with High-Level Petri Nets, *Advances in Petri Nets, LNCS Springer*, vol. 2001, (2000), pp. 276–304.

[3] M. Beato, M. Barrio-Solorzano, C. Cuesta, UML Automatic Verification Tool (TABU). *12th ACM SIGSOFT Symp. on the Foundations of Software Engineering*, 2004.

[4] B. Beckert, U. Keller, P. Schmitt, Translating the Object Constraints Language into First-order Predicate Logic. *Proc. Verify, Workshop at Federated Logic Conferences*, (2002), Copenhagen.

[5] S. Bernardi, S. Donatelli, J. Merseguer, From UML Sequence Diagrams and Statecharts to Analysable Petri Net models. *Proc. third int. workshop on software and performance*, Rome, Italy, ACM Press, (2002), pp. 35–45.

[6] T. Bouabana-Tebibel, M. Belmesk, Formalization of UML object dynamics and behavior. *Proc. 2004 IEEE Int. Conf. on Systems*, Man & Cybernetics, (2004), Netherlands.

[7] M. V. Cengarle, A. Knapp, Towards OCL/RT, In L.-H. Eriksson and P. Lindsay, eds.. *Formal Methods – Getting IT Right*, LNCS vol. 2391, Springer, (2002), pp. 389–408.

[8] D. Distefano, J.-P. Katoen, A. Rensink, On a Temporal Logic for Object-Based Systems. *Proc. 4th Int. Conf. on Formal Methods for Open Object-Based Distributed System*, FMOODs (2000), Stanford, USA.

[9] S. Flake, W. Mueller, Past-and Future-Oriented Temporal Time-Bounded Properties with OCL. *Proc. 2nd Int. Conf. on Software Engineering and Formal Methods, China*, ©IEEE Computer Society Press, (2004), pp. 154–163

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

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

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

[13] K. Jensen, Coloured Petri nets. *Vol 1: Basic Concepts*, Springer, (1992).

[14] S. Kuske, A formal semantics of UML state machines based on structured graph transformation, UML: The Unified Modeling Language. *Modeling Languages, Concepts and Tools*, LNCS, vol. 2185, (2001), pp. 241–256.

[15] R. Marcano, N. Lévy Transformation rules of OCL Constraints into B Formal Expressions. *(University of Versailles Saint-Quentin-en-Yvelines Ed.)*, (May 2002).

[16] E. Meyer, Développements formels par objets : utilisation conjointe de B et d'UML. *PhD thesis, University of Nancy 2*, France. 2001.

[17] Ober, S. Graf, I. Ober, Validating Timed UML Models by Simulation and Verification. *Proc. Int. Workshop SVERTS: Specification and Validation of UML Models for Real Time and Embedded Systems*, (2003), USA.

[18] Object Management Group, UML 2.0 Superstructure Specification. (2004).

[19] Object Management Group, OMG Unified Modeling Language Specification. *version 1.5*, (2003).

[20] Object Management Group, UML 2.0 OCL Specification. (October 2003).

[21] Object Management Group, The UML Action Semantics. (November 2001).

[22] PROD 3. 4, An advanced tool for efficient reachability analysis. *Laboratory for Theoretical Computer Science, Helsinki University of Technology*, (2004), Espoo, Finland.

[23] D. ROE, K. BRODA, A. RUSSO, Mapping UML Models Incorporating OCL Constraints into Object-Z. *Imperial College Technical Report N^{0}2003/9*, (2003).

[24] J.-C. ROYER, Temporal Logic Verifications for UML, The Vending Machine Example. *RSTI - L'objet, 4th Rigorous Object-Oriented Methods Workshop*, Vol. 9, No. 4 (2003).

[25] N. TRUONG, J. SOUQUIÈRES, Verification of behavioral elements of UML models using B. *In proc. of 20th Annual ACM Symposium on Applied Computing*, (2005), USA.

[26] N. TRUONG, J. SOUQUIÈRES, Validation des propriétés d'un scénario UML/OCL à partir de sa dérivation en B. Approches Formelles dans l'Assitance au Développement de Logiciels – AFADL'04, (2004), France.

[27] S. UCHITEL, J. KRAMER, J. MAGEE, Synthesis of Behavioral Models from Scenarios. *IEEE Transactions on Software Engineering*, Vol. 29, No. 2, (2003), pp. 99–115.

[28] T. ZIADI, L. HÉLOUËT, J.-M. JÉZÉQUEL, Revisiting Statechart Synthesis with an Algebraic Approach. *Proc. 26th Int. Conf. on Software Engineering (ICSE'04) ACM*, (2004) , pp. 242–251, Edimburgh, UK.

Received: June, 2006
Revised: January, 2007
Accepted: January, 2007

Contact address:
Thouraya Bouabana-Tebibel
National Institute of Computer Science
INI BP 68M
16309 Oued-Smar
Alger, Algeria
e-mail: t_tebibel@ini.dz

THOURAYA BOUABANA-TEBIBEL obtained the B.S degree in Computer Science from Houari-Boumédième Technology and Science University (USTHB), Algeria and the M.S degree in Industrial Engineering from Polytechnic National School (ENP) of Algeria. She is now a Ph.D. candidate in Software Engineering at the USTHB University and she works as an assistant 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.