

## Generating Maude Specifications From UML Use Case Diagrams

**Farid Mokhati**, Department of Computer Science, University of Oum-El-Bouaghi, Algeria

**Mourad Badri**, Department of Mathematics and Computer Science, University of Québec at Trois-Rivières, Canada

### Abstract

This paper presents a systematic approach supporting the translation of UML use case diagrams, describing the functional requirements of a system, into a Maude formal specification. The proposed approach also considers the static and dynamic features of object-oriented systems. The formal and object-oriented language Maude, based on rewriting logic, supports formal specification and programming of concurrent systems. The major motivations of this work are: (1) translating the functional requirements of an object-oriented system, specified using UML use case diagrams, into a Maude specification, (2) translating its static and dynamic aspects, described using UML class, communication and state-transitions diagrams respectively, into a Maude specification, and (3) integrating the formal verification of the consistency of the models, since the analysis phase. A case study is presented to illustrate our approach.

## 1 INTRODUCTION

UML (Unified Modeling Language) is a language for specifying, visualizing and constructing the artifacts of software systems [OMG05]. UML has become a standard for object-oriented modeling. Its graphical notation makes it easy to understand, in particular during the first phases of the development process. However, the fact that UML lacks formal semantics can lead to serious problems [Led01]. This weakness can lead, in particular, to inconsistencies within the developed models. Requirements analysis is a primordial step of the development process. The quality of the models produced at this phase is extremely important for the remaining phases of the development process. Their formal validation allows avoiding many problems that may affect the quality of the development as well as its cost [Dan07].

In this context, the use case diagrams and their related UML models play an important role. The use case diagram allows describing the functional requirements of an object-oriented system and represents an interesting communication tool between developers and users. However, like the other UML diagrams, it offers only a graphical

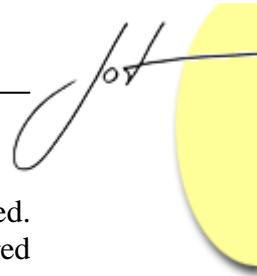
and semi-formal description of requirements. A more formal approach, in the description of the use cases, will reduce confusion and misunderstanding risks between developers and users [She03]. As mentioned in [Sno01, Bru98], an appropriate combination of object-oriented techniques and formal methods may make practically the software development more rigorous. To achieve this purpose, one of the possible approaches consists in deriving formal specifications (Maude in our case) from UML models and analyzing them formally. An object based development process begins [Jac93, Rum94, Gli00], by building the use case diagram and the class diagram of the application domain. Communication and state-transitions diagrams are also added to these models in the following steps of the analysis process to describe, respectively, the collective and individual behaviour of objects. Each use case is described by at least one communication diagram where exchanged messages between objects appear as methods in the class diagram. The behaviour of objects of the same class is described by one state-transitions diagram. The collaborative behaviour of objects, described by UML communication diagrams, shows how objects (a group of objects) interact to realize the use cases (operations of use cases). These different models allow describing, in a complementary manner, several facets of the use cases. They represent the first models of an object-oriented iterative development. It is important to ensure, at this level, their consistency.

This paper presents a systematic approach supporting the translation of UML use case diagrams into a Maude formal specification. The proposed approach also considers, the static (UML class diagram) and dynamic (UML communication and state-transitions diagrams) features of the system. The formal and object-oriented language Maude, based on rewriting logic, offers formal and sound basis for the specification and programming of concurrent systems. Furthermore, a Maude specification constitutes a coherent and executable representation of the specifications described using UML artefacts. The principal motivations of this work are: (1) translating the functional requirements of an object-oriented system, specified using UML use case diagrams, into a Maude specification, (2) translating its static and dynamic aspects, described by UML class, communication and state-transitions diagrams respectively, into a Maude specification, and (3) integrating the formal verification of the consistency of the models, since the analysis phase. A case study is presented to illustrate our approach.

The remainder of the paper is organized as follows. In section 2, we give a brief overview of related works. Section 3 briefly presents the UML diagrams we use. In section 4 we give an overview of rewriting logic and Maude. The proposed translation process is presented in section 5. Section 6 illustrates the translation process using a concrete case study. Finally, we give a conclusion and future work directions in section 7.

## 2 RELATED WORK

In [Mor99], the authors have proposed a translation process of the use case model into Object-Z. They have formalized use cases using temporal logic to define invariants in the Object-Z classes schemes. However, in their work, they only consider use case and



---

sequence diagrams. The structural aspects of the system have not been covered. Furthermore, the adopted approach does not allow the formal validation of the considered models. The developed framework also does not take into account concurrency aspects such as an object that receives a same message from several senders simultaneously.

With the objective of using jointly UML and B in a rigorous, unified and practically development process, Ledang et al. [Led01] have proposed an approach for translating the class, communication and use case diagrams into a B specification. In this approach, the UML descriptions are analyzed through the generated B specification. However, this approach does not consider explicitly the concurrent aspects of the described system. Furthermore, the B notation is not object-oriented and consequently the abstract specification generated from an object modeling may be different from which we would write directly [Tat01].

In [Fer07], the authors have proposed an approach for translating the UML 2.0 use case and sequence diagrams into a CPN formal description. The objective of this work is to develop a tool allowing software engineers to use jointly UML and CPNs. Although the CPN description is formal, it remains graphical and needs to be supported by a tool for a formal verification. The structural aspects are not covered in this approach.

We present in this paper, a generic approach supporting the generation of a Maude specification describing the functional requirements of an object-oriented system. The approach takes into account jointly the UML models mentioned in section 1, and also considers the concurrent aspects of the system. Furthermore, the use of UML and Maude is motivated by the desire to use them jointly in a practically, unified and rigorous development approach. Our choice of Maude is mainly motivated by its powerful description of object-oriented concurrent systems. It offers, indeed, a powerful formal framework for the description of intra and inter objects concurrency. Furthermore, Maude is supported by a tool, which allows validating the UML diagrams through their Maude description. This tool also incorporates a model-checker which allows analysing and verifying formally the system's properties.

## 3 UML DIAGRAMS

### Class diagram

UML class diagrams express the static structure of a system, in terms of classes and relationships between classes. Classes are essentially organized through aggregation, inheritance or association relationships [Mul00, OMG05].

### State diagram

UML state diagrams [Mul00] describe, using finite state machines, the life cycle of objects. Different types of events are defined by UML. We will focus only on the events of the "Call" type.

## Communication diagram

UML communication diagrams [Boo98, Mul00] describe how a set of objects collaborate to accomplish a specific task (for example an operation of a use case). They emphasize the dynamic interactions between those objects (message exchanges) as well as their synchronization. The concept of synchronization between messages is accomplished using the "/" symbol. A synchronization point is used to note the necessity of the completion of a particular message before the execution of another can begin, for example.

## Use case diagram

UML use case diagrams describe, in the form of action and reaction, the system's behaviour from the user's point of view. They allow defining the system's limits and the relationships between the system and the environment [Mul00]. The use case diagrams represent use cases, actors and the relationships between the use cases and the actors.

## 4 REWRITING LOGIC AND MAUDE

### Rewriting Logic

Rewriting logic, having a sound and complete semantic, was introduced by Meseguer [Mes92]. It allows describing concurrent systems [Mes03, McC03, Eke02, Cla05]. This logic unifies all the formal models that express concurrency [Mes92]. The rewriting rules are based on the general form of  $R: [t] \rightarrow [t']$  if  $C$ , which indicates that, according to rule  $R$ , term  $t$  becomes or is transformed into  $t'$  if a certain condition  $C$  is verified. This rule is of the conditional form. There also exist unconditional rules where the conditional term  $C$  is not present.

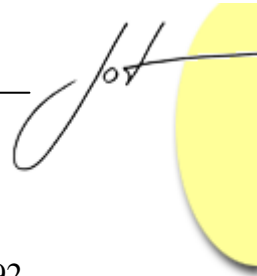
```

1. sort Configuration .
2. sort Object .
3. sort Msg .
4. subsort Object < Configuration .
5. subsort Msg < Configuration .
6. op null : -> Configuration .
7. op _ : Configuration Configuration -> Configuration [assoc
comm id : null] .

```

Figure 1. Example of a portion of a Maude program.

The example shown in Figure 1 gives the definition of three types: Configuration, Object and Msg (those two last being subtypes of Configuration). In the case where there is no floating messages or live objects, the global configuration of the system is empty. The construction of a new configuration, in terms of other configurations, is done with the operation given on line 7. This operation satisfies the structural laws of associability and commutability and possesses a neutral element called null.



---

## Maude

Maude is a specification and programming language based on rewriting logic [Mes92, Cla01, Cla05, McC03]. Three types of modules are defined in Maude. Functional modules allow defining data types and their functions. System modules allow defining the dynamic behaviour of a system. This type of module augments the functional modules by introducing rewriting rules. Finally, object-oriented modules, which can be reduced to system modules, offer a more appropriate syntax to describe the basic entities of the object paradigm. Maude environment has an incorporated model checker. However, model checking is out of the scope of this paper, but will be addressed in a future work.

## 5 TRANSLATION PROCESS

Figure 2 illustrates the translation process supported by our approach. The methodology of our approach needs, in a first time, defining the system's functional requirements using a use case diagram. For realizing use cases, we associate to each use case one or several communication diagrams, representing different possible scenarios. The messages exchanged between objects carry on methods defined in the class diagram.

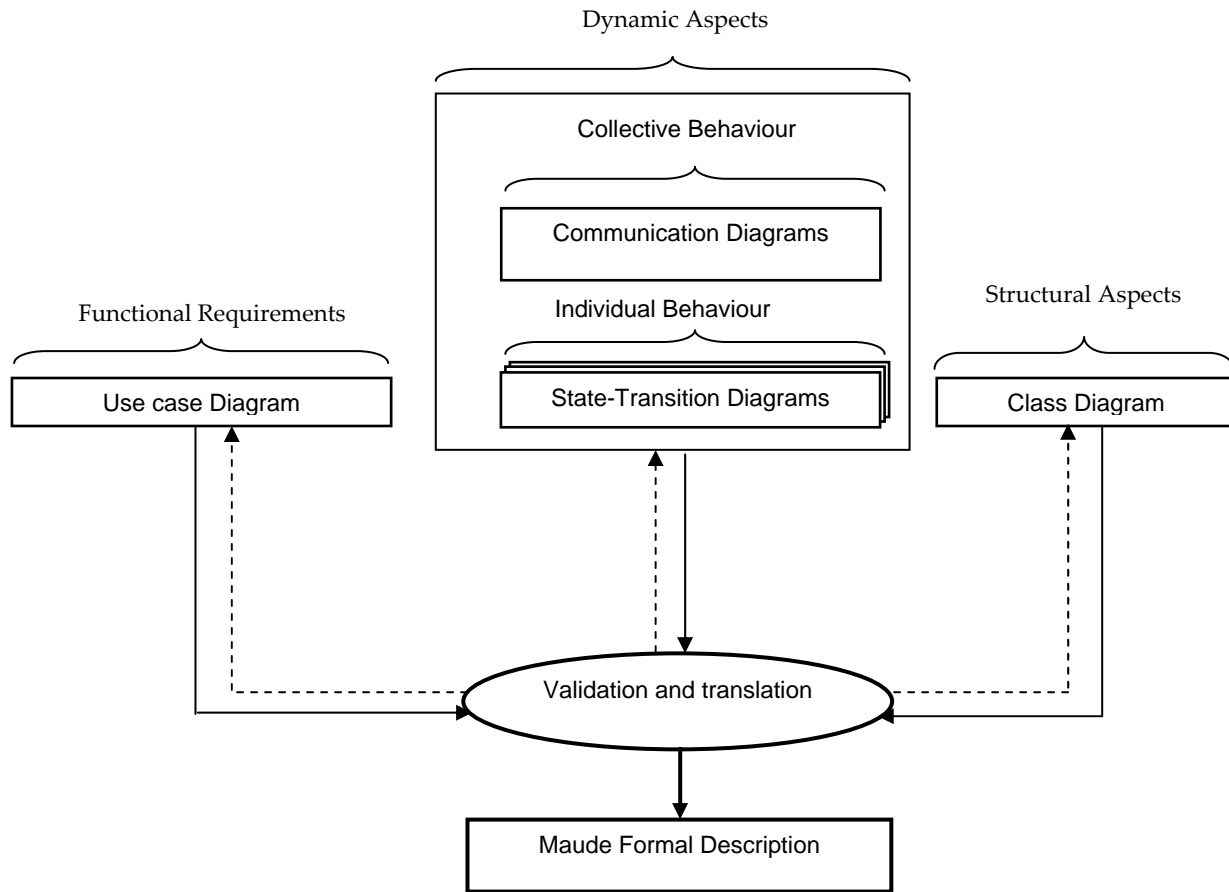


Figure 2: Methodology of the approach.

The individual behaviours of objects involved in the collaboration, to realize an operation of a use case, are described by state-transitions diagrams. The considered diagrams go through a first step of an inter-models validation in order to verify the system consistency. For example, each message sent to a destination object in the communication diagram must exist in the state diagram of the object and it is accessible. In the same way, each use case in the use case diagram must be realized by at least one communication diagram.

The proposed translation process consists of generating a Maude formal description from use case, class communication and state diagrams analysis (Figure 2). During this process, several modules are generated. Figure 3 illustrates those modules. Please note that the modules in bold are object-oriented modules, while all others are functional modules.

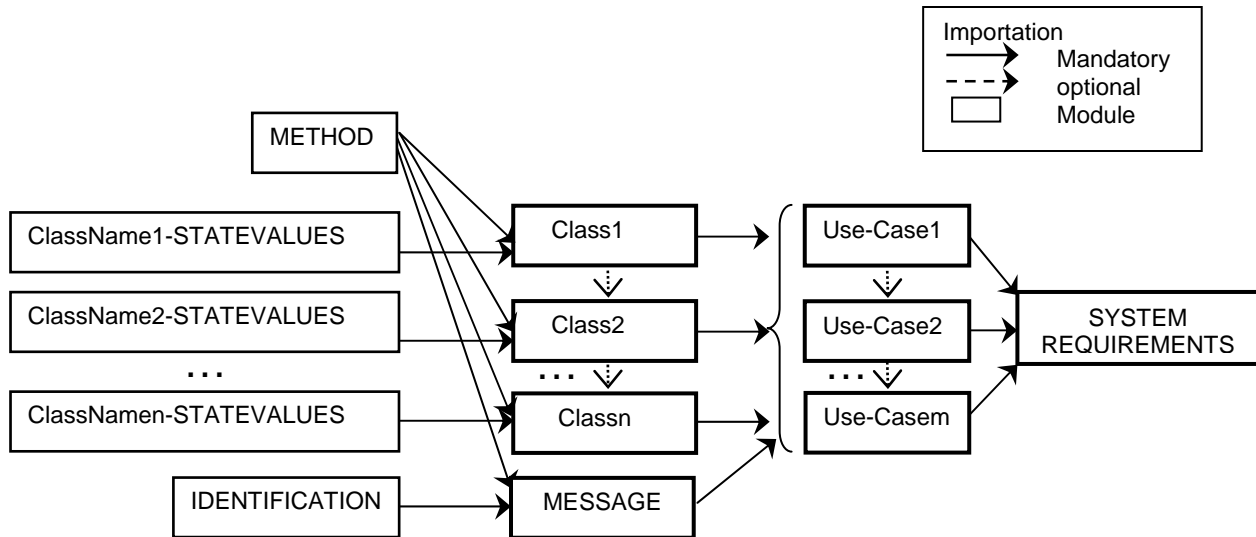
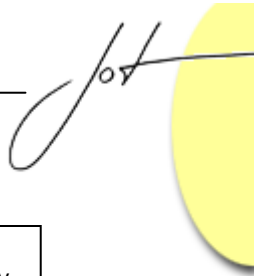


Figure 3: Generated Modules.

The functional module *METHOD* (Figure 4) contains all the types used to describe a method. Types *Parameter* and *ParameterList* are generic. They describe the type of parameters a method uses. Furthermore, *ResultType* and *Void* describe the type of the result returned by the method. *ResultType* is generic, and *Void* is a particular case of *ResultType*. The operation  $(\_,\_)$  is a constructor for the parameter list of a function.

```
fmod METHOD is
  sorts ParameterList ResultType Parameter Void .
  subsort Parmeter < ParamaeerList .
  subsort Void < ResultType .
  op EmptyParameterList : -> ParameterList .
  op _,_ : Parameter ParameterList -> ParameterList .
endfm
```

Figure 4. The functional module *METHOD*.

A functional module is associated to each state diagram for which the name is the concatenation of the class' name and the string 'STATEVALUES'. This module describes the state values a class can take according to its state diagram. The functional module *IDENTIFICATION* is generated to describe the identification mechanism of the objects of the communication diagram. For each class of the class diagram, we associate an object-oriented module bearing the same name as the class, while adopting a generic form for the classes (Figure 5).

In the case where one of such a class is in relation to other classes in the class diagram, the module associated to it must import all the other modules associated to those classes. The class is declared in a module with a state attribute called *State* and for which its type is declared in the corresponding functional module. In the case of an aggregation class, an identification list of all the aggregated classes must also be present.

```
class ClassName | State : ClassNameStateValues [,
ComponentList] .
```

Figure 5. Generic form adopted for the classes.

The methods of a class are also defined in the corresponding module using the following form (Figure 6):

```
op FunctionName : ParameterList -> ResultType .
```

Figure 6. Form adopted for the methods.

We define an object-oriented module *MESSAGE* in which are defined the forms of the messages exchanged between objects as well as the form of the synchronization message (see Figure 7).

```
mod MESSAGE is
  protecting IDENTIFICATION METHOD .
  op ComingMsg : ResultType Receiver -> Msg .
  op IsAccomplished : ResultType Receiver -> Msg .
endm
```

Figure 7. Form adopted for the messages.

Each message exchanged between two objects of the communication diagram is translated in the form of a *ComingMsg* shown in Figure 7. With this message, we specify two things. On the first hand, we identify the destination object (*Receiver*) and, on the other hand, the result type of the operation to be executed. In fact, each sending of a message in the communication diagram corresponds to a *Call Event*, launching a transition in the state diagram of the destination object.

To implement the concept of *Synchronization Point* of the messages sent within a communication diagram, a new message called *IsAccomplished* is introduced (see Figure 7). The rewriting rule that implements transition corresponding to the sending of a message on which depend other messages must generate a number of *IsAccomplished* messages equal to the number of messages to be sent. This message is also used in the case where the sending of an asynchronous message depends on the sending of another message. Furthermore, to each use case is associated an object-oriented module *Use-Casei* bearing the same name as the corresponding use case. In each module *Use-Casei* are defined the rewriting rules describing the different interaction scenarios between the objects defined in the different communication diagrams, instances of the use case. A module describing a use case can import (optional importation) another describing a use case which is linked to it. Once generated, the modules *Use-Casei* are imported in the object-oriented module *SYSTEM REQUIEREMENTS* representing the principal module (see Figure 8). This module describes, in fact, the system's dynamic behaviour from the user's point of view.





```
omod SYSTEM-REQUIERMENTS is
  protecting Use-Case1 .
  protecting Use-Case2 .
  ...
  protecting Use-Casem .
endom
```

Figure 8. The principal module *SYSTEM-REQUIERMENTS*.

## 6 CASE STUDY: THE ELEVATOR

This section illustrates the application of the proposed approach on a concrete example taken from [Mul00]. This example was simplified for the present study. It carries on the elevator working. Figure 9 shows the class diagram of that system. The functional requirements are described by the use case diagram of figure 10. The use case *TransportByElevator* is realized by the communication diagram of figure 11. It consists of the procedure done by a user at a given moment to use the elevator after it was started properly. Figure 12 shows respectively the state diagrams for classes *Door*, *SignalLight*, *Cabin*, and *Elevator*.

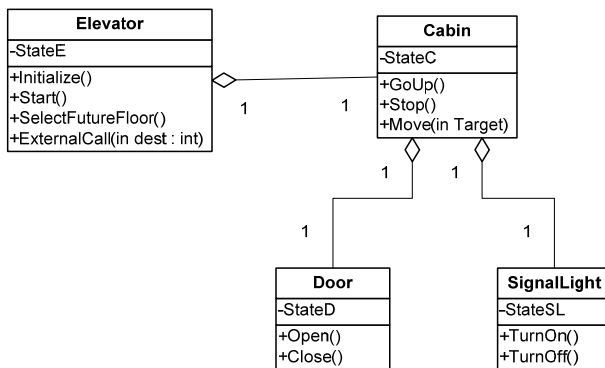


Figure 9. Class diagram.

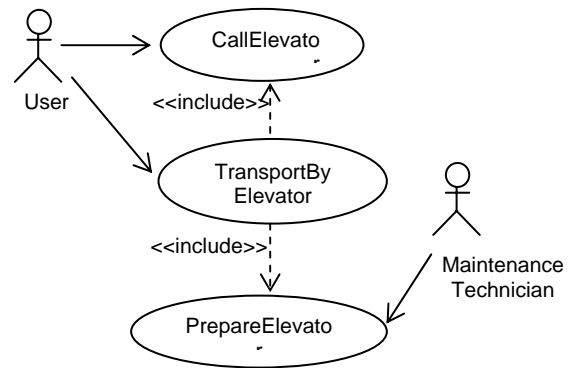


Figure 10. Use case diagram of the elevator.

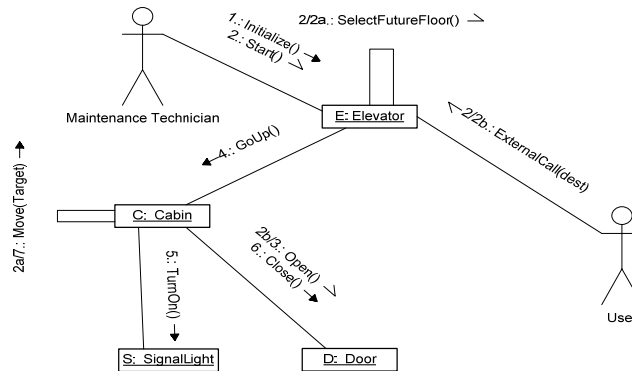


Figure 11. Communication diagram.

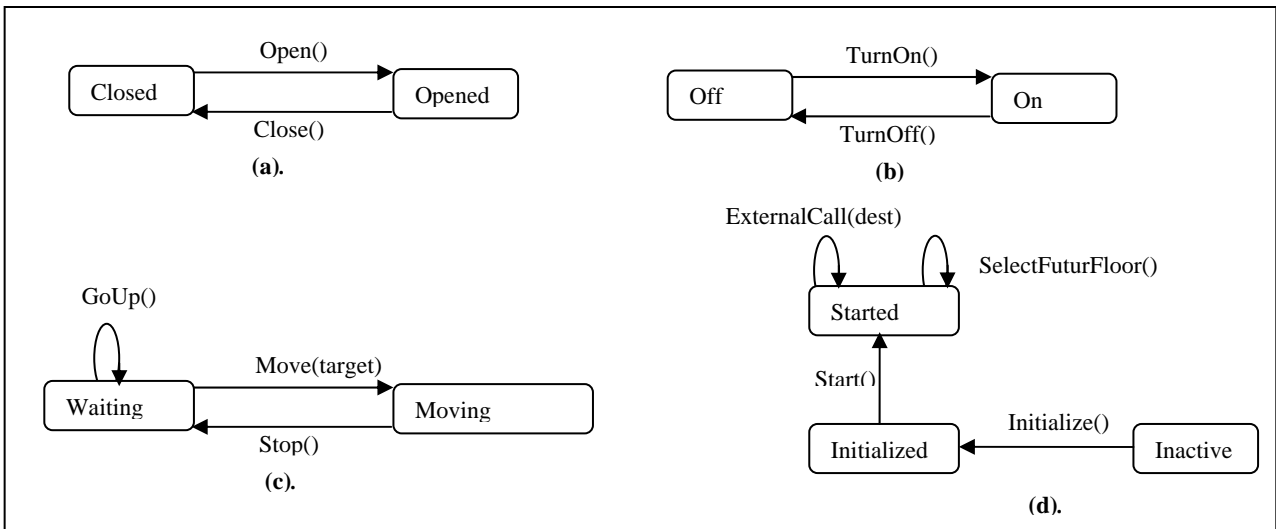


Figure 12. State diagrams for classes *Door* (a), *SignalLight* (b), *Cabin* (c) and *Elevator* (d).

### Application of the translation process

By applying the translation process, we obtain the modules described in what follows. We have four functional modules: ELEVATOR-STATEVALUES, CABIN-STATEVALUES, SIGNALLIGHT-STATEVALUES and DOOR-STATEVALUES. These modules contain respectively the states of the different classes: Elevator, Cabin, SignalLight and Door. For reasons of space limitation, only the code for one of them is given, namely ELEVATOR-STATEVALUES (see Figure 13).



```
fmod ELEVATOR-STATEVALUES is
  sort ElevatorStateValues .
  ops Inactive Initialized Started :-> ElevatorStateValues .
endfm
```

Figure 13. Module ELEVATOR-STATEVALUES.

A module *IDENTIFICATION* (see Figure 14) imports the predefined *CONFIGURATION* module. This module contains the definition of types *Eoid*, *Coid*, *Doid*, *Soid* which describe the identification mechanism of the objects *E*, *C*, *P* and *S*, instance of classes *Elevator*, *Cabin*, *Door* and *SignalLight* respectively.

```
fmod IDENTIFICATION is
  including CONFIGURATION .
  sort Eoid Coid Doid Soid Receiver .
  subsort Eoid Coid Doid Soid < Oid .
  subsort Receiver < Eoid Coid Doid Soid .
endfm
```

Figure 14. Module *IDENTIFICATION*.

```
omod CABIN is
  protecting DOOR .
  protecting SIGNALLIGHT .
  protecting CABIN-STATEVALUES .
  sort Cabin Target .
  subsort Cabin < Cid .
  subsort Target < Parameter .
  ops UP DOWN : -> Target .
  class Cabin | State : CabinStateValues, IdDoor : Oid, IdSL
    : Oid .
  ops UP DOWN : -> Target .
  op GoUp : ParameterList -> Void .
  op Stop : ParameterList -> Void .
  op Move : Target -> Void .
endom
```

Figure 15. Module *CABIN*.

We have four object-oriented modules: *ELEVATOR*, *CABIN*, *SIGNALLIGHT* and *DOOR*. In each module, a class is defined with a *State* attribute describing the current state of the object, and a list of composing objects in case of aggregate classes, as well as the different methods of the class. The code for only one of those modules is presented here, namely *CABIN* (see Figure 15).

Three object-oriented modules implementing the use cases are generated by our approach; *CALL-ELEVATOR*, *TRANSPORTBYELEVATOR* and *PREPARE-ELEVATOR*. We give in what follows the code of the module *TRANSPORTBYELEVATOR* (sww Figure 16).

```

omod TRANSPORTBYELEVATOR is
include CALL-ELEVATOR PREPARE-ELEVATOR
protecting IDENTIFICATION MESSAGE.
extending ELEVATOR CABIN DOOR SIGNALLIGHT .

*** Utility Variables *****
var E : Eoid . var C : Coid . var D : Doid . var S : SLoId .

*** Elevator's Behavior *****
rl [E1]: ComingMsg(Initialize( EmptyParameterList ), E)
  < E : Elevator | State : Inactive, IdCabin : C >
=>
  < E : Elevator | State : Initialized, IdCabin : C >
  IsAccomplished(Initialize ( EmptyParameterList ), E) .

rl [E2]: IsAccomplished(Initialize ( EmptyParameterList ), E)
  ComingMsg(Start( EmptyParameterList), E )
  < E : Elevator | State : Initialised, IdCabin : C >
=>
  < E : Elevator | State : Started, IdCabin : C >
  IsAccomplished(Start( EmptyParameterList ), E)
  IsAccomplished(Start( EmptyParameterList ), E) .

...
endom

```

Figure 16. Module TRANSPORTBYELEVATOR.

This module imports, on the first hand, the modules *CALL-ELEVATOR* and *PREPARE-ELEVATOR*, and on the other hand the modules *IDENTIFICATION* and *MESSAGE*. Furthermore, it extends modules *ELEVATOR*, *CABIN*, *DOOR* and *SIGNALLIGHT*. Figure 16 presents two rewriting rules. The rewriting rule ‘E1’ describes the reception of message *Initialize* by object *E*. After its execution, the rule generates a message *IsAccomplished* that will be used to allow asynchronous message *Start* to be sent. The execution of the second rule, namely ‘E2’, needs, aside from the *IsAccomplished* message generated by the first rule, the arrival of message *Start*. Such a rule generates two *IsAccomplished* messages, to allow two other messages, namely *SelectFutureFloor* and *ExternalCall*, to be sent (see Figure 11).

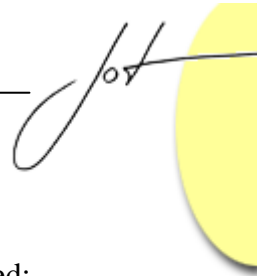
The object-oriented module *SYSTEM-REQUIEREMENTS* (Figure 17) constitutes the principal module generated by our approach. It imports modules *TRANSPORTBYELEVATOR*, *CALL-ELEVATOR* and *PREPARE-ELEVATOR*.

```

omod SYSTEM-REQUIEREMENTS is
protecting CALL-ELEVATOR .
protecting PREPARE-ELEVATOR .
protecting TRANSPORTBYELEVATOR .
endom

```

Figure 17: Module SYSTEM-REQUIEREMENTS.



---

## Validation of the generated description

To illustrate the validation of the generated description, two essential cases are presented: the case where the elevator receives an external call, after it was initialized and started by the maintenance technician, and the case where the elevator receives a message for selecting the next floor. For the first case, the initial configuration is given by figure 18:

```
< E : Elevator | StateE : Started, IdCabin : C >  
< C : Cabin | StateC : Waiting, IdDoor : D, IdSignalLight : S >  
< D : Door | StatedD : Closed > < S : SignalLight | States : Off >  
ComingMsg(ExternalCall( 4 ), E ) .
```

Figure 18. First initial configuration.

This configuration represents, an object *E*, instance of the class *Elevator* in starting state (*Started*), an object *C* of the class *CABIN* in waiting state (*Waiting*), an object *D* of the class *DOOR* in closed state (*Closed*) and an object *S* of the class *SIGNALLIGHT* in extinguished state (*Off*). It also shows the arrival of an external call accomplished by the user from the floor number 4. The unlimited rewriting of this configuration returns the result given in figure 19.

```
< E : Elevator | StateE : Started, IdCabin : C >  
< C : Cabin | StateC : Waiting, IdDoor : D, IdSignalLight : S >  
< D : Door | StateD : Closed > < S : SignalLight | StateS : On >
```

Figure 19. Result of the unlimited rewriting of the first initial configuration.

This result shows that, the *Elevator* is in its *Started* state, the *Door* is closed, the *SignalLight* is lit, but the *Cabin* is always in a waiting state because the user has not selected yet its destination.

For the second case, we extend the configuration of figure 19. We relaunch the rewriting process from the result of the rewriting of the first configuration while adding to it the arrival of a message for selecting the next floor *SelectFuturFloor* ( figure 20).

```
< E : Elevator | StateE : Started, IdCabin : C >  
< C : Cabin | StateC : Waiting, IdDoor : D, IdSignalLight : S >  
< D : Door | StatedD : Closed > < S : SignalLight | StateS : On >  
ComingMsg(SelectFuturFloor( EmptyParamatersList ), E ) .
```

Figure 20. Second initial configuration.

The unlimited rewriting of the second configuration returns the result given by the figure 21. This configuration is similar to the one of figure 19 except that the *Cabin* is in moving.

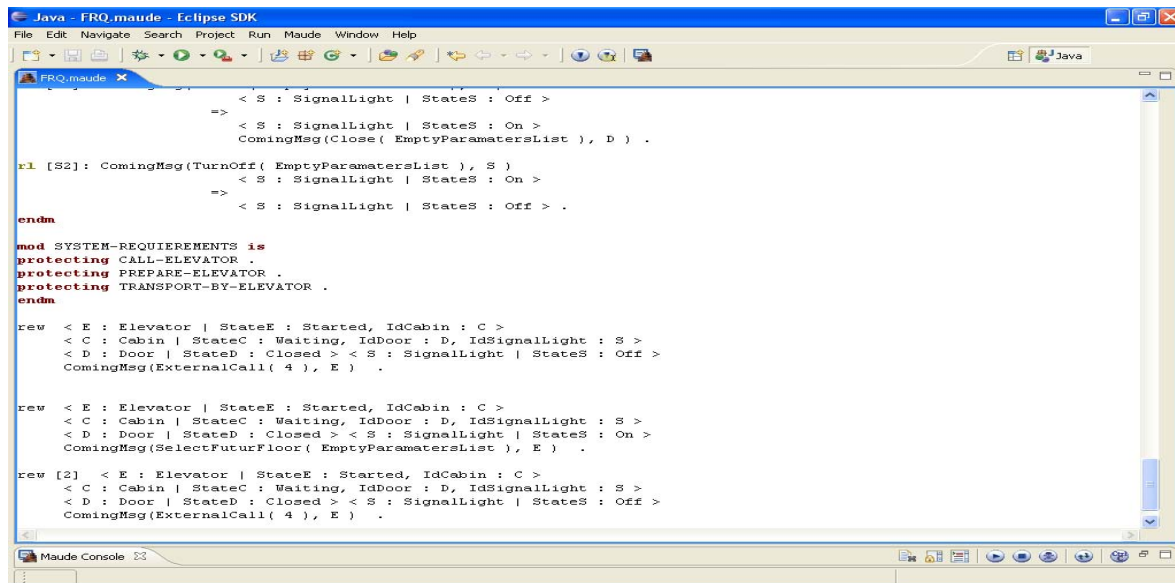
```
< E : Elevator | StateE : Started, IdCabin : C >
< C : Cabin | StateC : Moving, IdDoor : D, IdSignalLight : S >
< D : Door | StateD : Closed > < S : SignalLight | StateS : On >
```

Figure 21. Result of the unlimited rewriting of the second initial configuration.

## Implementation

Figure 22 shows part of the code developed in the Maude language, namely the principal module *SYSTEM-REQUIERMENTS* of our framework which describes the system's functional requirements illustrated by the use case diagram of figure 10. This module imports the others modules (*CALL-ELAVATOR*, *PREPAR-ELEVATOR* and *TRANSPORT-BY-ELEVATOR*) implementing the different scenarios of the different use cases of figure 10.

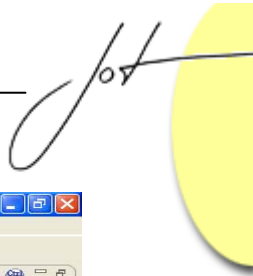
The following figure also illustrates, the launching of the unlimited rewriting process of the configurations of figures 18 and 20, and the limited rewriting (after two rewriting steps) of the first initial configuration of figure 18.



```

Java - FRQ_maude - Eclipse SDK
File Edit Navigate Search Project Run Maude Window Help
FRQ_maude
    < S : SignalLight | States : Off >
    =>
    < S : SignalLight | States : On >
    ComingMsg(Close( EmptyParametersList ), D ) .
r1 [S2]: ComingMsg(TurnOff( EmptyParametersList ), S )
    < S : SignalLight | States : On >
    =>
    < S : SignalLight | States : Off > .
endm
mod SYSTEM-REQUIERMENTS is
protecting CALL-ELEVATOR .
protecting PREPAR-ELEVATOR .
protecting TRANSPORT-BY-ELEVATOR .
endm
rev < E : Elevator | StateE : Started, IdCabin : C >
< C : Cabin | StateC : Waiting, IdDoor : D, IdSignalLight : S >
< D : Door | StateD : Closed > < S : SignalLight | States : Off >
ComingMsg(ExternalCall( 4 ), E ) .
rev < E : Elevator | StateE : Started, IdCabin : C >
< C : Cabin | StateC : Waiting, IdDoor : D, IdSignalLight : S >
< D : Door | StateD : Closed > < S : SignalLight | States : On >
ComingMsg(SelectFuturFloor( EmptyParametersList ), E ) .
rev [2] < E : Elevator | StateE : Started, IdCabin : C >
< C : Cabin | StateC : Waiting, IdDoor : D, IdSignalLight : S >
< D : Door | StateD : Closed > < S : SignalLight | States : Off >
ComingMsg(ExternalCall( 4 ), E ) .
    
```

Figure 22. Part of the developed code.



```
Java - FRQ.maude - Eclipse SDK
File Edit Navigate Search Project Run Maude Window Help
Maude Console
Ready.
Advisory: redefining module IDENTIFICATION.
Advisory: redefining module CALL-ELEVATOR.
Advisory: redefining module PREPARE-ELEVATOR.
Warning: <standard input>, line 1189 (mod TRANSPORT-BY-ELEVATOR): syntax error
Advisory: redefining module TRANSPORT-BY-ELEVATOR.
Advisory: redefining module SYSTEM-REQUIEREMENTS.
rewrite in SYSTEM-REQUIEREMENTS : (((ComingMsg(ExternalCall(4), E) < S :
  Signallight | StateS : Off >) < D : Door | Stated : Closed >) < C : Cabin |
  StateC : Waiting,IdDoor : D,IdSignallight : S >) < E : Elevator | StateE :
  Started,IdCabin : C > .
rewrites: 5 in 48608ms cpu (1ms real) (0 rewrites/second)
result Configuration: < E : Elevator | StateE : Started,IdCabin : C > < C :
  Cabin | StateC : Waiting,IdDoor : D,IdSignallight : S > < D : Door | Stated
  : Closed > < S : Signallight | StateS : On >
rewrite in SYSTEM-REQUIEREMENTS : (((ComingMsg(SelectFuturFloor(
  EmptyParametersList), E) < S : Signallight | StateS : On >) < D : Door |
  Stated : Closed >) < C : Cabin | StateC : Waiting,IdDoor : D,IdSignallight
  : S >) < E : Elevator | StateE : Started,IdCabin : C > .
rewrites: 2 in 48607ms cpu (0ms real) (0 rewrites/second)
result Configuration: < E : Elevator | StateE : Started,IdCabin : C > < C :
  Cabin | StateC : Moving,IdDoor : D,IdSignallight : S > < D : Door | Stated
  : Closed > < S : Signallight | StateS : On >
rewrite [2] in SYSTEM-REQUIEREMENTS : (((ComingMsg(ExternalCall(4), E) < S :
  Signallight | StateS : Off >) < D : Door | Stated : Closed >) < C : Cabin |
  StateC : Waiting,IdDoor : D,IdSignallight : S >) < E : Elevator | StateE :
  Started,IdCabin : C > .
rewrites: 2 in 48608ms cpu (1ms real) (0 rewrites/second)
result [Configuration]: ComingMsg(GoUp(EmptyParametersList), C) < E : Elevator
  | StateE : Started,IdCabin : C > < C : Cabin | StateC : Waiting,IdDoor : D,
  IdSignallight : S > < D : Door | Stated : Open > < S : Signallight | StateS
  : Off >
Maude>
||
```

Figure 23. Results of the rewriting of the different configurations.

The results of the rewriting of the different configurations are illustrated by figure 23. This figure shows, the results of the unlimited rewriting of the two first configurations. These results are like to those presented in figures 19 and 21. The figure shows also the result of the limited rewriting of the first configuration. This last result represents an intermediate configuration illustrating: the *elevator* in starting, the *cabin* in waiting, the *door* is open, the *signallight* is extinguished and the arrival of a message indicating that the user is ready to accede to the cabin.

## 7 CONCLUSION AND FUTURE WORK

In this paper, we proposed a generic approach that allows translating functional requirements described by UML use case diagrams into a Maude formal specification. This thematic has been addressed in several papers published in the literature. However, the structural and/or concurrenial aspects of the systems have not been covered in most of these papers.

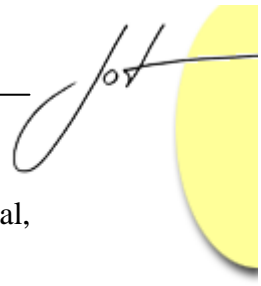
The proposed approach takes into account the system's structural and dynamic (individual and collective) features. Furthermore, concurrenial aspects have also been considered. Maude is very appropriate for describing object-oriented concurrent systems. The Maude language is supported by a tool, which allowed us to validate the generated code by simulation.

Maude offers a model checker in its environment, which uses Linear Temporal Logic (LTL) to verify properties among the developed models. As future work, we plan on extending the formal framework we developed to analyze and verify system's functional requirements through their Maude descriptions using the model checker incorporated in Maude environment. This model checker uses on-the-fly techniques to manage the state-space problem from which model checking techniques suffer. Linear Time Logic is used to define desirable or non desirable properties that are to be checked in the system under development.

## REFERENCES

- [Boo98] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modeling Language User Guide*. Addison-Wesley, Object Technology Series, 1998.
- [Bru98] J.M. Bruel. Integrating Formal and Informal Specification Techniques. Why? How? In the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques, pages 50–57, Boca Raton, Florida (USA), 1998.
- [Cla01] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. F. Quesada, *Maude: Specification and Programming in Rewriting Logic*. Theoretic Computer Science, 2001.
- [Cla05] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott, *Maude Manual (Version 2.1.1)*. April 2005.
- [Dan07] D.H. Dang. Validation of System Behavior Utilizing an Integrated Semantics of Use Case and Design Models. In Claudia Pons, editor, *Proceedings of the Doctoral Symposium at the ACM/IEEE 10th International Conference on Model-Driven Engineering Languages and Systems (MoDELS 2007)*. Nashville (TN), USA, October 1st, 2007. CEUR, Vol-262, 2007.
- [Eke02] S. Eker, J. Meseguer and A. Sridharanarayanan, *The Maude LTL Model Checker*. Elsevier Science B V, 2002.
- [Fer07] J. M. Fer et al. Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net, *SCESM*, 2007.
- [Gli00] M. Glinz. A Lightweight Approach to Consistency of Scenarios and Class Models. In *Proceedings of the Fourth International Conference on Requirements Engineering*, Illinois, June 10-23, 2000.
- [JCJO93] I. Jacobson, M. Christerson, P. Jonson, and G. Overgaard. *Le génie logiciel orienté objet*. Addison Wesley France, 1993.
- [Led01] H. Ledang and J. Souquière. Formalizing UML Behavioral Diagrams with B. Tenth OOPSLA Workshop on Behavioral Semantics: Back to Basics, Tampa Bay, Florida, USA, 2001.





- 
- [McC03] T. McCombs, Maude 2.0 Primer, Version 1.0. Internal report, SRI International, 2003.
- [Mes92] J. Meseguer, A Logical Theory of Concurrent Objects and its Realization in the Maude Language. G Agha, P Wegner and A Yonezawa, Editors, Research Directions in Object-Based Concurrency. MIT Press, 1992, pp. 314-390.
- [Mes03] J. Meseguer, Software Specification and Verification in Rewriting Logic. Computer Science Department, University of Illinois at Urbana-Champaign, 2003.
- [Mor99] A. Moreira and J. Araújo. Generating Object-Z Specifications from Use Cases. International Conference on Enterprise Information Systems (ICEIS'99). Setúbal, Portugal, 1999.
- [Mul00] P. A. Muller and N. Gaertner, Modélisation Objet avec UML. Second Edition, Paris, 2000.
- [OMG05] Object Modeling Group. Unified Modeling Language Specification, version 2.0. July 2005.
- [Rum94] J. Rumbaugh. Using use cases to capture requirements. Journal of object-oriented programming, 7(5), September 1994.
- [She03] W. Shen and S. Liu. Formalization, Testing and Execution of a Use Case Diagram. In 5th International Conference on Formal Engineering Methods (ICFEM'03), 2003.
- [Sno01] C. Snook and R. Harrison. Practitioners Views on the Use of Formal Methods: An Industrial Survey by Structured Interview. Information and Software Technology March 2001, 43:275–283, 2001.
- [Tat01] B. Tatibouet, A. Hammad, Une utilisation conjointe de B et UML sur l'étude de cas du robot typé, Conférence d'Ingénierie des Systèmes (AFIS 2001), France, Toulouse, p. 285-290, June 2001.

## About the authors



**Farid Mokhati** ([Mokhati@yahoo.fr](mailto:Mokhati@yahoo.fr)) is an assistant professor of computer science at the Department of Computer Science of the University of Oum El-Bouaghi in Algeria. He holds a Ph.D. in computer science (Distributed Artificial Intelligence) from the University of Annaba in Algeria. His main areas of interest include object and agent-oriented software engineering, and formal methods.



**Mourad Badri** ([Mourad.Badri@uqtr.ca](mailto:Mourad.Badri@uqtr.ca)) is professor of computer science at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières (Quebec, Canada). He holds a Ph.D. in computer science (software engineering) from the National Institute of Applied Sciences in Lyon (France). His main areas of interest include object and aspect-oriented software engineering, software quality assurance, and formal methods.