Verifying UML Diagrams With Model Checking: A Rewriting Logic Based Approach

Farid Mokhati,
Department of Computer Science
University of Oum-El-Bouaghi
Algeria
Mokhati@yahoo.fr

Patrice Gagnon and Mourad Badri
Department of Mathematics and Computer Science
University of Québec at Trois-Rivières
Canada
{patrice.gagnon | mourad.badri }@uqtr.ca

Abstract

We present, in this paper, a framework supporting a formal verification of UML diagrams using the Maude language. The approach considers both static and dynamic features of object-oriented systems. We focus, in particular, on UML class, state and communication diagrams. The formal and object-oriented language Maude, based on rewriting logic, supports formal specification and programming of concurrent systems, as well as model checking. The major motivations of this work are: (1) bind together the UML notation and the Maude language (2) preserve the coherence in object-oriented systems description, (3) use model checking techniques to support formally their verification process. The generated Maude specifications, from the considered UML diagrams, are validated by simulation and model checking. The approach is illustrated using a concrete case study.

1. Introduction

UML (*Unified Modeling Language*) is a language for specifying, visualizing and constructing the artifacts of software systems [18]. Nowadays, it is considered as the standard for object-oriented modeling. UML allows modeling various aspects of complex systems. However, UML models can present some ambiguities and inconsistencies as stated in many papers [31, 33, 11]. UML allows, in fact, only a semi formal specification of object-oriented systems. It suffers from a lack of formal semantics [4, 30, 33]. This weakness can lead to inconsistencies within the developed models. Using formal methods, particularly in the development of complex systems, presents notable advantages [19, 11, 33, 24, 5], like a simpler design without ambiguities, as well as a more complete documentation [5, 33].

In this paper, we present a formal framework supporting: (1) the automatic translation of UML diagrams into a formal specification based on the Maude language and (2) the verification of some LTL properties using Maude's integrated model checker. We consider both static and dynamic features of object-oriented systems. We focus, in particular, on UML class, state and communication diagrams jointly. The approach is organized in four major steps. The first step consists of describing both static and dynamic features of an object-oriented system using UML class (static structure), state (individual behavior) and communication diagrams (collective behavior in terms of dynamic interactions). The second step corresponds to an inter-diagrams validation process. Since all model elements used in our approach can be captured in Maude, we can, in the third step, generate automatically a Maude description from the considered UML diagrams. The fourth step consists of verifying some LTL properties using Maude's model checker [13, 24, 10]. We focus, in this paper, on the verification process of UML diagrams using Maude's model checking techniques. The translation process has been addressed in a previous paper [26].

The remainder of the paper is organized as follows: Section 2 gives a brief overview of related work. Section 3 gives an overview of rewriting logic and Maude. We present, in Section 4, the main phases of the translation process we propose. Section 5 illustrates the translation process using a case study. Section 6 presents how Maude's model checker can be used to verify some LTL properties on the specifications we developed. Finally, we give a conclusion and some future work directions in Section 7.

2. Related Work

Funes et al. [16] have formalized UML class diagram using the formal specification language RSL (*RAISE Specification Language*). Using the same language, Naixiao et al. [27] presented formalization for state diagrams. Furthermore, Favre [14] has proposed a translation process for class diagrams and packages in the *NEREUS* language, based on the MDA

Seventh International Conference on Quality Software (QSIC 2007)
0-7695-3035-4/07 $25.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

(*Model Driven Architecture*) methodology. The *NEREUS* specification obtained is then transformed into an object-oriented code (*Eiffel* language). Joao et al. [2] proposed a generation process to obtain *Object-Z* specifications from UML communication diagrams. In the same context, other UML diagrams have been considered [12, 20]. On the other hand, Paige and Brooke presented in [28, 29] a pragmatic approach integrating the object-oriented methodology *BON* (an alternative to UML) and the *Object-Z* language. The majority of these papers have focused on translating to a formal specification only one feature of object-oriented systems, whether static or dynamic.

In the same context, other approaches have used jointly class diagrams to describe static aspects of object-oriented systems and state diagrams to describe their dynamic aspects (individual behavior of objects). We can cite, among others, the *U2B* tool [33]. *U2B* is a script file for *Rational Rose* allowing the conversion of UML model to the *B* language. However, the collective behavior of objects is not considered. Furthermore, H. Ledang et al. have developed the *ArgoUML+B* tool [3]. Of course, those proposals have considerably forwarded the domain by integrating static and dynamic features of object-oriented systems and their translation into formal specifications. However, the dynamic features considered in those papers are only related to the individual behavior of objects. The collaborative behavior is not addressed.

Model checking issues are nowadays a very active research domain. *SPIN* is one of the most renowned model checkers available. It has been used in several works. In [32, 25], *SPIN* has been used to model check state machines and collaborations together. The authors created a tool called *HUGO* which compares the state charts descriptions defined in *PROMELA* to textual representations of collaborations. *HUGO* then uses *SPIN* to complete its verifications. In [6], the authors present a tool, called *NEPTUNE*, which contains a module, called *Checker*, supporting the verification of UML models including some properties expressed using the OCL language. Furthermore, the tool *BON-CASE* [29] contains a reasoning engine which allows the verification of different properties, which is comparable to *NEPTUNE*. In [7, 8], the authors used the *RSML* language (which is an alternative to UML to represent state charts) to formalize the *TCAS II* program (avionics anti collision software). They then use the *SMV* model checker to verify that their system accomplishes its tasks correctly. They also present a number of ways to reduce the state space explosion problem to acceptable levels.

In [15], the authors propose an approach that formalizes the static semantics of UML state charts. Furthermore, they verified the mutual orthogonality propriety between a set of source states (respectively of target states) of a complex transition. The same authors, in [1], formalized UML class diagrams using an algebraic specification theory. In those two papers, the authors used the Maude language. By using the *Object-Z* language, the authors of [17] demonstrated how an approach of meta-modeling can be extended to define consistency integrity constraints for UML state machines. However, those papers take into consideration only one model, whether static as in [1], or dynamic as in [15, 17]. Also, the authors do not take into account the concurrent aspects of OOS, and model checking is not addressed in those papers.

We present, in this paper, a more global approach allowing the generation of a Maude formal specification integrating both static and dynamic (individual and collective) features of object-oriented systems. We use UML class diagrams to represent static features of an object-oriented system, and state and communication diagrams (respectively individual and collective behavior) to represent its dynamic features. The novelty of the approach lies in the fact that it considers both the individual and collective behavior of objects through state-transition diagrams and a communication diagram. The formal and object-oriented language Maude, based on rewriting logic, supports formal specification and programming of concurrent systems [23, 9, 21, 13, 24, 22, 10]. It also offers a Model Checking environment. Aside from the semantics of concurrency (intra and inter-objects) that it offers, Maude is a multi paradigm language [10, 24]. Furthermore, the Maude language is supported by a tool, which allows validating the generated formal descriptions through simulations. Maude integrates a model checker supporting the verification of *Linear Temporal Logic* (LTL) properties [13, 24, 10]. The Maude environment is still not very used. We wished to explore its possibilities in both formal specification and model checking aspects of UML diagrams.

3. Rewriting Logic and Maude

Rewriting logic, on which Maude is based, was introduced by Meseguer [23]. It allows the description of concurrent systems [24, 9, 21, 13, 10, 22]. This type of logic unifies all formal models of concurrency [22]. The rewriting rules are of the form *R : [t] -> [t'] if C*, which indicates that, according to rule *R*, term *t* becomes *t'* if a certain condition *C* is verified. The *C* condition is also optional.

4. Translation Process

The adopted translation process consists of systematically deriving a Maude formal specification

IEEE COMPUTER SOCIETY

from an analysis of the UML class, state and communication diagrams. Fig. 1.(a) presents the steps of the translation process we presented in [26]. Because the translation process was already introduced in this previous paper, only the main elements are mentioned here. During the translation process of the considered UML diagrams, several Maude modules are generated. Fig. 1.(b) shows those modules. The modules in bold are object-oriented modules, while all others are functional modules.

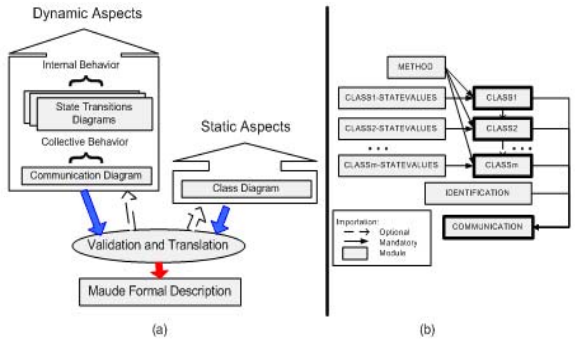


Figure 1. (a) Overview of the Translation Process and (b) Generated modules

The first step consists of defining the classes' state values in a distinct module, one for each class, a *'STATEVALUES'* module. One object module is defined for each class in which the class itself is defined according to the generic form of Fig. 2.

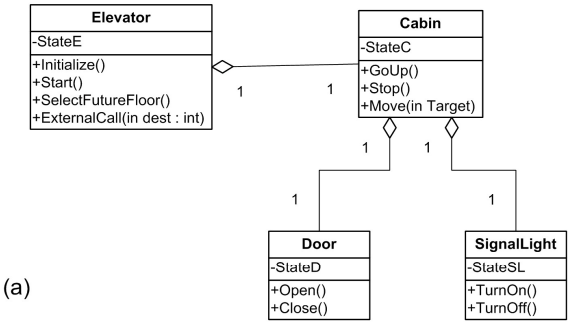
```
*** The METHOD module ***********
(fmod METHOD is
  sorts ParameterList ResultType Parameter Void .
  subsort Parameter < ParameterList .
  subsort Void < ResultType .
  op EmptyParameterList : -> ParameterList .
  op _,_ : Parameter ParameterList -> ParameterList .
endfm)
*** Generic forms **********
class ClassName | State : ClassNameStateValues [,
ComponentList] .
op FunctionName : ParameterList -> ResultType .
*** Messages and Synchronization **********
msg ComingMsg : ResultType Receiver -> Msg .
msg IsAccomplished : ResultType Receiver -> Msg .
```

Figure 2. The METHOD module and generic definitions for class, method and messages

The *State* attribute is automatically introduced to describe an object's current state. The functional module *IDENTIFICATION* is generated to describe the identification mechanism of the objects. The object-oriented module *COMMUNICATION* is the principal one generated by our approach. In this module, we extend all the other object-oriented modules by describing the behavior of the different objects implicated in the communication diagram using rewriting rules. It also takes into consideration the different state transition diagrams relevant to the different classes involved. Each message exchanged between two objects of the communication diagram is translated in the form of a *ComingMsg* shown in Fig. 2. This *ComingMsg* form is used to standardize the messages passed between objects. The *IsAccomplished* is used to model a synchronization point in a communication diagram. It is not an actual message sent between two objects.

5. Case Study: The Elevator

5.1. Presentation



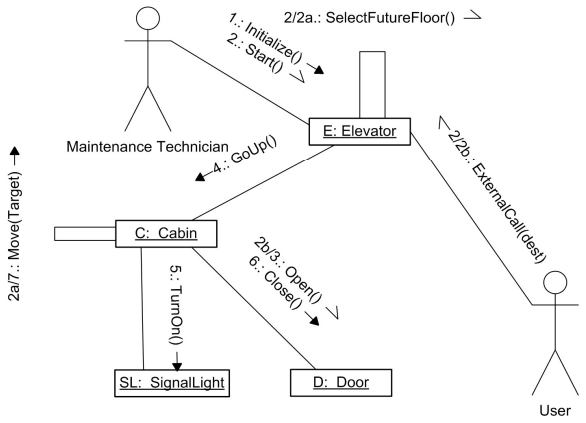


Figure 3. (a) Class diagram of the system and (b) the Communication diagram to be studied

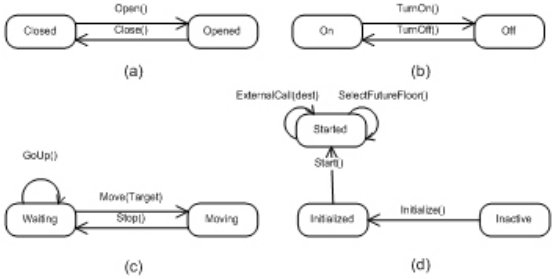


Figure 4. Transition diagrams for classes Door (a), SignalLight (b), Cabin (c) and Elevator (d)

As mentioned previously, the complete translation from the corresponding UML diagrams (Fig. 3 and 4)

0-7695-3035-4/07 $25.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

was the main purpose of a previous paper [26], therefore only the key elements are presented here. Our objective, in this paper, is to extend our work by applying a model checking technique for the verification of UML models with Maude's model checker (Section 6). Our study focuses on a system composed of four classes (Fig. 3.(a)), each of them having a corresponding state diagram (Fig. 4). The task to be studied is expressed in the communication diagram of Fig. 3.(b).

5.2. Application of the Translation Process: A Brief Review

```
*** Module CABIN-STATEVALUES **********
(fmod CABIN-STATEVALUES  is
   sort CabinStateValues .
   ops Moving Waiting :-> CabinStateValues .
endfm)
*** Module IDENTIFICATION **********
(fmod  IDENTIFICATION  is
   including CONFIGURATION .
   sort Eoid  Coid Doid Soid Receiver .
   subsort Eoid Coid Doid Soid < Oid .
   subsort Receiver < Eoid Coid Doid Soid .
endfm)
*** Module CABIN **********
(omod CABIN is
  protecting DOOR SIGNALLIGHT .
  protecting CABIN-STATEVALUES .
  sorts Cabin Target .
  subsort Cabin < Cid .
  subsort Target < Parameter .
  class Cabin | StateC : CabinState-Values,
              IdDoor : Oid, IdSL : Oid .
  ops UP DOWN : -> Target .
  op GoUp :  ParameterList -> Void .
  op Stop :  ParameterList -> Void .
  op Move : Target  -> Void .
endom)
```

Figure 5. Modules CABIN-STATEVALUES, IDENTIFICATION and CABIN

The first modules generated by the translation from UML to Maude notations are the four functional modules describing the state values of the four classes. Furthermore, a module *IDENTIFICATION* is also introduced (Fig. 5) to describe the identification mechanism of the objects involved in the communication (Fig. 3.(b)).

More specifically, it introduces several notations to better identify the objects involved in the communication diagram. These notations are object identifiers used to conform to Maude notations. In our case study, *Eoid*, *Coid*, *Doid* and *Soid* are four types of object identifiers, all sub types of *Oid*, the general object identifier of Maude. Fig. 5 introduces those modules (note that only one state values module is given, namely *CABIN-STATEVALUES*). The next modules are object-oriented and introduce the classes themselves, along with their respective methods.

An object-oriented module *COMMUNICATION* constitutes the principal module generated by our approach. In this module, rewriting rules are introduced to translate a communication diagram and all object interactions it describe while taking into consideration the different state transition diagrams for the relevant classes. It imports all the modules already defined.

```
(omod COMMUNICATION is
  protecting IDENTIFICATION ELEVATOR CABIN DOOR .
  extending SIGNALLIGHT .
  msg ComingMsg : ResultType Receiver -> Msg .
  msg IsAccomplished : ResultType Receiver -> Msg .
  var E : Eoid . var C : Coid . var D : Doid . var SL : Soid .
  **** Elevator's Behavior *****
  rl [E1]: ComingMsg(Initialize( EmptyParametersList ), E )
     < E : Elevator | StateE : Inactive, IdCabin : C >
  =>
     < E : Elevator | StateE : Initialized, IdCabin : C >
     ComingMsg(Start( EmptyParametersList ), E ) .
  rl [E2]: ComingMsg(Start( EmptyParametersList ), E )
     < E : Elevator | StateE : Initialized, IdCabin : C >
     < D : Door | StateD : Closed >
  =>
     < E : Elevator | StateE : Started, IdCabin : C >
     < D : Door | StateD : Closed >
     ComingMsg(Open( EmptyParametersList ), D )
     IsAccomplished(Start( EmptyParametersList ), E)
     IsAccomplished(Start( EmptyParametersList ), E) .
...
endom)
```

Figure 6. Part of the COMMUNICATION module

Only part of this module is shown in Fig. 6. Rewriting rules *'E1'* and *'E2'* describe how objects of the *Elevator* class behave when receiving messages *Initialize* and *Start*. The important information in this is the use of *IsAccomplished* message to ensure the synchronization in sending messages, since messages *SelectFuturFloor* and *ExternalCall* cannot be executed before *Start* is completed (see Fig. 3.(b)).

5.3. Validation of the Generated Description

Rewriting logic is very flexible when it comes to simulation of a specification [24, 21, 10]. We consider, in this section, the verification of the final behavior of the complete system by using simulations. By this process, we attempt to verify that the developed specification executes properly. Since the communication diagram of Fig. 3.(b) was translated and the developed specification is supposed to model the interaction it describes, the simulation we will attempt will verify that this exact interaction occurs. Fig. 7 shows the results returned by the Maude system following the introduction of a rewriting command. The initial configuration used is composed of the four objects in their respective initial state, as well as the sequence of messages shown in Fig. 3.(b). Note that

IEEE COMPUTER SOCIETY

the final configuration of the second rewriting is coherent for such an elevator system. It consists on the elevator being started, the cabin is moving, the door is closed and the Signal Light is on.

```
rewrite in COMMUNICATION : (((((ComingMsg(SelectFutureFloor(
    EmptyParametersList), "E") ComingMsg(ExternalCall(4), "E")) ComingMsg(
    Initialize(EmptyParametersList), "E")) < "S" : SignalLight | StateSL : Off
    >) < "D" : Door | StateD : Closed >) < "C" : Cabin | StateC : Waiting,
    IdDoor : "D",IdSL : "S" >) < "E" : Elevator | StateE : Inactive,IdCabin :
    "C" > .
rewrites: 9 in 66139ms cpu (1ms real) (0 rewrites/second)
result Configuration: < "C" : Cabin | StateC : Moving,IdDoor : "D",IdSL : "S" >
    < "D" : Door | StateD : Closed > < "E" : Elevator | StateE : Started,
    IdCabin : "C" > < "S" : SignalLight | StateSL : On >
```

Figure 7. Results of the rewritings, the last two rewritings verifying the entire system's behavior

6. Model Checking

In our opinion, the verification of the collective behavior of a system must start with the verification of the individual behavior of objects. In what follows, we adopt an incremental process in the verification of properties. Section 6.1 focuses on properties of the individual behavior of objects *E*, *C*, *D* and *S*, instances of classes *Elevator*, *Cabin*, *Door* and *SignalLight* respectively. Then, Section 6.2 introduces properties of the collective behavior (in terms of dynamic interactions) of those same objects.

6.1. Properties Relative to the Individual Behavior of Objects

We consider, in what follows, three properties (Properties 1, 2 and 3) relative to the internal behavior of objects.

– Property 1: *[] (Elevator-In-Inactive-State("E") -> (<> Elevator-In-Initialized-State("E")))*
  This property expresses that it is always true that if *Elevator* ("E") is in its *Inactive* state, it is eventually being in its *Initialized* state ("<>" is the *Eventually* temporal operator, while "->" is the Implies temporal operator and "[]" means *Always*)
– Property 2 : *[] (Elevator-In-Inactive-State("E") -> (<> Elevator-In-Started-State("E")))*
  This property expresses that it is always true that if *Elevator* ("E") is in its Inactive state, it is eventually being in its *Started* state.
– Property 3 : *[] (Cabin-In-Waiting-State("C") -> (<> Cabin-In-Moving-State("C")))*
  This property expresses the fact that it is always true that if the *Cabin* ("C") is in its *Waiting* state, it is eventually being in its *Moving* state.

6.2. Properties Relative to the Collective Behavior

In this section, we introduce four properties (Properties 4, 5, 6 and 7) concerning the collaborative behavior of objects, which was introduced in the communication diagram of Fig. 3.(b).

– Property 4 : *<> (Elevator-In-Inactive-State("E") /\ Cabin-In-Moving-State("C"))*
  This property expresses the fact that the *Elevator* and the *Cabin* are in their respective states *Inactive* and *Moving* at least once at the same time.
– Property 5 : *<> (Cabin-In-Moving-State("C") /\ Door-In-Opened-State("D"))*
  This property expresses the fact that the *Cabin* and the *Door* are respectively in their *Moving* and *Opened* states at least once at the same time.
– Property 6 : *[]~ (Cabin-In-Moving-State("C") /\ Door-In-Opened-State("D"))*
  This property expresses the fact that it is always true that the *Cabin* is not in its *Moving* while the *Door* is in its *Opened* state. This desirable property is the counterpart of Property 5.

Table 1. Global states and Corresponding Object States.

Global State	Class ELEVATOR	Class CABIN	Class DOOR	Class SL
State0	Inactive	Waiting	Closed	Off
State1	Initialized	Waiting	Closed	Off
State2	Started	Waiting	Closed	Off
State3	Started	Waiting	Opened	Off
State4	Started	Waiting	Opened	On
State5	Started	Waiting	Closed	On
State6	Started	Moving	Closed	On

– Property 7 : *[] (State0 -> <>(State1 -> <> (State2 -> <> (State3 -> <> (State4 -> <> (State5 -> <> State6))))))*
  This property expresses the fact that the system should follow a sequence of usual steps to accomplish its tasks. This property will be referred to as the *System's Invariant*, which should always be true at any given execution point in the system. Table 1 shows, for each global states (*State0* to *State6*) the corresponding states of each objects.

Fig. 8 presents part of the *COMMUNICATION-PREDICATES* module, in which we define the predicates relative to the *Elevator* system we are studying. We define, in this module, the necessary operators we used in the definition of the properties we wish to verify. We limit the shown predicates to the two, relative to class *Elevator*, as well as the first relevant to he *System's Invariant* Property.

The definition of predicates is a systematic process. One predicate is defined for each state of each class. For example, line 1 defines a predicate for state *Inactive* of class *Elevator*. The same process is

IEEE COMPUTER SOCIETY

adopted for all other classes and their respective states. Finally, line 2 shows how a global state for the system is defined. This is done in regard to Property 7, which is our *System's Invariant* property. Table 1 showed which corresponding states made a global system state. Line 2 shows how *State0* can be defined. Again, the same process is adopted for the six other global system states (according to Table 1).

```
(omod COMMUNICATION-PREDICATES is
  protecting COMMUNICATION .  including SATISFACTION .
  subsort Configuration < State .
  ***** Operators relevant to the Elevator class *****…
  ops State0 State1 State1 State2 State3 State4 State5
       State6 : -> Prop .
…
  ***** Predicates *****
  eq < E : Elevator | StateE : Inactive, IdCabin : C > conf
     |= Elevator-In-Inactive-State("E") = true . ***1
…
  eq   < E : Elevator | StateE : Inactive, IdCabin : C >
       < C : Cabin | StateC : Waiting, IdDoor : D, IdSL : S >
       < D : Door | StateD : Closed >  < S : SignalLight |
         StateSL : Off > conf |= State0  = true .    ***2
…
endom)
```

Figure 8. Part of the COMMUNICATION-PREDICATES module

6.3. Properties Verification

We used the *modelCheck* function of Maude to verify the properties we retained. Fig. 9 shows part of the *COMMUNICATION-CHECK* module. A single initial state is used to verify all the 7 properties. It consists on the four objects of the system in their respective initial states, as well as incoming messages to initialize the elevator and an external call and a selection of a future floor by a user.

```
(omod COMMUNICATION-CHECK is
…
 op initial : -> Configuration .
 eq initial = ComingMsg(Initialize( EmptyParametersList ), "E" )
   …
 < "D" : Door | StateD : Closed > < "S" : SignalLight |
 StateSL : Off > ComingMsg(ExternalCall( 4 ), "E" )
 ComingMsg(SelectFutureFloor(EmptyParametersList ), "E" ) .
endom)
```

Figure 9. Part of the COMMUNICATION-CHECK module and 2 model checking calls

Partial results are given in Fig. 10. It shows the partial result to Property 5, and the result of Properties 6. Full results and interpretation are given in what follows. Properties 1, 2 and 3 were relevant to the individual behavior of objects. Namely, Properties 1 and 2 verified that the start up process done by the maintenance technician is performed correctly. This means that, eventually, the *Elevator* will reach its *Started* state. Those two properties were verified in our case study system. Property 3 verified that a stopped *Cabin* will eventually be *Moving*. This property also evaluated to true.

```
    Cabin | StateC : Waiting,IdDoor : "D",IdSL : "S" > < "D" : Door | StateD :
    Opened > < "E" : Elevator | StateE : Started,IdCabin : "C" > < "S" :
    SignalLight | StateSL : Off >,'S1) (ComingMsg(Close(EmptyParametersList),
    "D") IsAccomplished(SelectFutureFloor(EmptyParametersList), "E") < "C" :
    Cabin | StateC : Waiting,IdDoor : "D",IdSL : "S" > < "D" : Door | StateD :
    Opened > < "E" : Elevator | StateE : Started,IdCabin : "C" > < "S" :
    SignalLight | StateSL : On >,'D2) (ComingMsg(Move(UP), "C") IsAccomplished(
    SelectFutureFloor(EmptyParametersList), "E") < "C" : Cabin | StateC :
    Waiting,IdDoor : "D",IdSL : "S" > < "D" : Door | StateD : Closed > < "E" :
    Elevator | StateE : Started,IdCabin : "C" > < "S" : SignalLight | StateSL :
    On >,'C2), (< "C" : Cabin | StateC : Moving,IdDoor : "D",IdSL : "S" > < "D"
    : Door | StateD : Closed > < "E" : Elevator | StateE : Started,IdCabin :
    "C" > < "S" : SignalLight | StateSL : On >,deadlock))
reduce in COMMUNICATION-CHECK : modelCheck(initial, []~ (Cabin-In-Moving-State(
    "C") /\ Door-In-Opened-State("D"))) .
rewrites: 33 in 7714574001ms cpu (6ms real) (0 rewrites/second)
result Bool: true
```

Figure 10. Part of the results of the model checking calls

As for Properties 4, 5, 6 and 7, they can be interpreted as follows. Property 4 attempted to verify if the system allowed for the *Cabin* to be in its *Moving* while the *Elevator* system is in its *inactive*. This property was evaluated to false, meaning the system does not allow such a situation. Property 5 attempted to verify if the system allowed to *Cabin* to be in its *Moving* while the *Door* is in its *Opened*. Again, this Property was evaluated to false and means the system does not allows such a situation. Property 6 verifies that the *Cabin* is not in its *Moving* while the *Door* is in its *Opened*. Property 6 evaluated to true, which means the important characteristic of passenger safety is respected. Finally, Property 7 insures that the system always behaves according to a specified pattern of corresponding object states. This *System Invariant* property was also evaluated to true, insuring the correct and coherent sequence of execution steps.

7. Conclusions and Future Work

The translation of UML diagrams in formal languages has been addressed in numerous papers. However, the majority of those approaches did not consider the collective behavior of objects. In this paper, we proposed a generic approach that allows translating static aspects (described by UML class diagram) and dynamic aspects (described by UML communication diagrams and state diagrams) of object-oriented systems into a Maude formal specification integrating both static and dynamic features of a system. Our approach is however limited to basic state and communication diagrams, modeling only the most common features. Moreover, Maude offers a simulation and a model checker in its engine, which uses *Linear time Temporal Logic* (LTL). We used these tools to validate our models. We defined some LTL properties and used Maude's model checker to verify them. As future directions to this work, we

IEEE COMPUTER SOCIETY

plan on extending our approach to verify UML use cases diagrams.

References

[1] Alvarez Toval, A., Fernandez Aleman, J. L.: "Improving System Reliability via Rigorous Software Modeling: The UML Case". In Proceedings of the 2001 IEEE Aerospace Conference (track 10: Software and Computing), Montana, USA IEEE Computer Society (2001).

[2] Araujo, J., Moreira, A.: "Specyfying the Behavior of UML Collaborations Using Object-Z". Departamento de Infomatica, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal (2000).

[3] ArgoUML Tigris Organisation. ArgoUML User Manual (2002).

[4] Astesiano, E.: "UML as Heterogeneous Multiview Notation. Strategie for a Formal Foundation". In Proceedings of the conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'98) -Workshop on Formalizing UML. Why ? How? , Canada (1998).

[5] Bowen, J.: Formal Specification and Documentation Using Z: A Case Study Approach. (2003).

[6] Canals et al.: "How You Could Use NEPTUNE in the Modelling Process". In Journal of Object Technology, Vol. 2 (1) (2003), pp. 69-83.

[7] Chan, W., J. Anderson, R., Beame, P., Burns, S., Modugno, F., Notkin, D., D. Reese, J.: "Model Checking Large Software Specifications". In IEEE TSE, Vol. 24 (7), July (1998), pp. 498-520.

[8] Chan, W. et al.: "Improving E□ciency of Symbolic Model Checking for State-Based System Requirements". In ISSTA 98 : Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, Michal Young, editor, Clearwater Beach (USA), March (1998), pp. 102-112.

[9] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., F. Quesada, J.: "Maude: Specification and Programming in Rewriting Logic". Theoretical Computer Science (2001).

[10] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Mesenguer, J., Talcott, C.: Maude Manual (Version 2.1.1) April (2005).

[11] del Mar Gallardo, M., Merino, P., Pimentel, E.: "Debugging UML Designs With Model Checking". In Journal of Object Technology, Vol. 1 (2) (2002), pp. 101–117.

[12] Dong, J. S.: "Formal Specification and Design Techniques". Lecture Notes (2000).

[13] Eker, S., Meseguer, J., Sridharanarayanan, A.: "The Maude LTL Model Checker". In F. Gaducci and U. Montanari, editors, Proc. of the 4th International Workshop on Rewriting Logic and its Applications (WRLA 2002).

[14] Favre, L.: "Foundations for MDA-based Forward Engineering". In Journal of Object Technology, Vol. 4 (1) (2005), pp. 129–153.

[15] Fernández Alemán, J.L., Alvarez Toval, A.: "Can Intuition Become Rigorous? Foundations for UML Model Verification Tools". International Symposium on Software Reliability Engineering, Published by IEEE Press (2000).

[16] Funes, A., George, C.: "Formal Foundations in RSL for UML Class Diagrams". Technical Report 253. UNU/IIST, May (2002).

[17] Kim, S.K., Carrington, D.: "A Formal Object-Oriented Approach to defining Consistency Constraints for UML Models". In 2004 Australian Software Engineering Conference (ASWEC'04).

[18] Object Modeling Group. Unified Modeling Language Specification, Version 2.0. July (2005).

[19] Moreira, A., Bruel, J.M., Lilius, J., Robert, B.: "Defining Precise Semantics for UML". In ECOOP'2000 Workshop Reader, Vol. 1964. Springer-Verlag, November (2000).

[20] MacColl, I., A. Carrington, D.: "Specifying Interactive Systems in Object-Z and CSP". In Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors, IFM, Springer (1999) pp. 335–352.

[21] McCombs, T.: "Maude 2.0 Primer (Version 1.0)", August (2003).

[22] Meseguer, J.: "Rewriting as a Unified Model of Concurrency". SIGPLAN OOPS Mess., Vol. 2 (2), ACM Press, New-York (NY), USA (1991), pp. 86–88.

[23] Meseguer, J.: "A Logical Theory of Concurrent Objects and Its Realization in The Maude Language". In Research Directions in Concurrent Object-Oriented Programming, MIT Press, Cambrige (MA), USA (1993) pp. 314–390.

[24] Meseguer, J.: "Software Specification and Verification in Rewriting Logic". NATO Advanced Study Institute, Marktoberdorf, Germany (2003).

[25] Merz, S.: "Model Checking : A Tutorial Overview". In MOVEP, volume 2067 of Lecture notes in Computer Science Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, Springer (2000), pp. 3-38.

[26] Mokhati, F., Badri, M., Gagnon, P.: "Translating UML Diagrams into Maude Formal Specification: A Systematic Approach". In SEKE'06: Proceedings of the 18th International Conference of Software Engineering and Knowledge Engineering, Knowledge Systems Institute, Skokie (IL), USA (2006), pp. 572-577.

[27] Naixiao, Z., Meng, S., Aichernig, B.K.: "The Formal Foundations in RSL for UML Statechart Diagrams". Technical Teport 299. UNU/IIST, July (2004).

[28] Paige, R. F., Brooke, P. J.: "Integrating BON and Object-Z". In Journal of Object Technology, Vol. 3 (3) (2004), pp. 121–141.

[29] Paige, R. F., Kaminskaya, L., Ostroff, J., Lancaric, J.: "BON-CASE: An Extensible CASE Tool for Formal Specification and Reasoning". In Journal of Object Technology, Vol. 1 (3) (2002), pp. 77–96.

[30] Reggio, G., Wieringa, R.: "Thirty One Problems in the Semantics of UML 1.3 Dynamics". In Conference on Object Oriented programming, Systems, Languages and Applications (OOPSLA'99) -Workshop "Rigorous Modeling an Analysis of the UML Challenges and Limitations" (1999).

[31] Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language User Guide. In Addition-Wesley, Object Technology Series (1998).

[32] Schäfer, T., Knapp, A., Merz, S.: "Model Checking UML State Machines and Collaborations". In Electronic Notes in Theoretical Computer Science, Vol. 55 (3) (2001).

[33] Snook, C., Butler, M.: "U2B -A Tool for Translating UML-B Models into B". In UML-B Specification for Proven Embedded Systems Design. Springer (2004).

IEEE COMPUTER SOCIETY