

Verification of UML Model Elements Using B

NINH THUAN TRUONG AND JEANINE SOUQUIÈRES

LORIA - Université Nancy

Nancy, France

E-mail: {truong; souquier}@loria.fr

We propose an approach to verify UML model elements based on the transformation of the UML meta-model into B formal specifications. The UML meta-model is described as a combination of graphical notations, natural and formal languages. The semantics of UML elements is expressed by well-formedness rules in the UML meta-model. Their correctness is ensured by the proof of the B specifications. The approach is illustrated by a simple case study: the printing system.

Keywords: B formal method, UML model and meta-model, formal verification, proof, transformation, well-formedness rules

1. INTRODUCTION

Formal methods [11] are necessary for producing good softwares. Formal specifications allow for mathematical manipulation and reasoning, and facilitate the rigorous testing procedures. They can be used for the derivation of test cases, for the validation and the verification that the specification satisfies its requirements.

The Unified Modelling Language (UML) [6] is a widely accepted modelling language that can be used to visualize, specify, construct and document the artifacts of a software system. It has been accepted as a standard object-oriented modelling language by OMG [16], and is becoming the dominant modelling language in the industry. The syntax and semantics of the notations provided in UML are defined in terms of a meta-model. In the UML meta-model [16], modelling constructs are defined using three distinct views: an abstract syntax in UML class diagrams, static semantics in OCL [22], and dynamic semantics mainly in English.

The derivation from UML specifications into the B formal method [1] is considered as an appropriate way to use jointly UML and B in practical, unified and rigorous software development. The transformation from UML diagrams to B specifications have been considered in [9, 10, 15]; these approaches provide us with a multi-view framework for the specification of a system but do not allow the complete verification of the properties of UML elements. The transformation of the UML meta-model to formal methods (Object-Z, B) has been considered by K. Soon-Kyeong and C. David in [12], and R. Laleau, F. Polack in [13]. However, these approaches only consider the specification process they do not allow to check of UML model elements with support tools. Cavarra *et al.* [7] build a framework to transform UML meta-model and UML models into ASM

(Abstract State Machines) [3] but verification is not considered. Concerning the validation of UML models and OCL constraints, M. Richter *et al.* [18] present an approach based on animation. They have developed an animator for simulating UML models and an OCL interpreter for constraint checking.

The purpose of our work is to use B support tools to check UML model elements. We transform the meta-classes of an UML specification, the objects of these classes (which are elements of UML models) and the well-formedness rules of the UML semantics proposed by OMG [16] into B abstract machines. The corresponding B specification is then proved by support tools which generate automatically proof obligations. The transformation of the Core package and the verification of class diagrams is presented in [20]. The transformation of behavioural diagrams is presented in [21]. This paper integrates these derivations into a common approach to verify UML model elements from different diagrams.

The paper is organised as follows. Section 2 provides the basic concepts of our approach. Section 3 presents a case study to illustrate our purpose. Section 4 proposes a general principle to transform the UML meta-model into B. In the three next sections, we present the transformation of the meta-model of class diagrams, collaboration diagrams and state-chart diagrams into B. We illustrate the transformation of well-formedness rules to B invariants and the proof of UML model elements using B support tools.

2. BACKGROUND

We introduce the B method and part of the UML meta-model in relation with UML models.

2.1 The B Method

B [1] is a formal software development method, originally developed by J. R. Abrial. The B notations are based on set theory, generalised substitutions and the first order logic. The B method enables an incremental development process, known as a refinement process. A system development begins by the definition of an abstract view which can be refined step by step until an implementation is reached. The refinement over models is a key feature for developing incrementally models from a textually-defined system, while preserving correctness. It implements the proof-based development paradigm [4, 19]. The method has been successfully used in the development of several complex real-life applications, like the METEOR project [17]. It is one of the few formal methods which has robust and commercially available support tools for the entire development life-cycle from specification down to code generation [5]. Specifications are composed of abstract machines which are very closed to notions well-known in programming under the name of modules, classes or abstract data types. Each abstract machine consists of a set of variables, invariant properties of those variables and operations. The state of the system, i.e. the set of variable values, is modifiable by operations which must preserve its invariant. The proof obligations to the preservation of invariants are generated automatically by support tools like AtelierB [19], B-Toolkit [4] and B4free [8], an academic version of AtelierB. The check of proof obligations with B support tools either through automatic or

interactive proofs [2], is an efficient and practical way to detect errors introduced during the specification development.

2.2 UML Meta-Model and its Relation with UML Models

The UML meta-model [16] defines the semantics for representing object models using UML. It is defined in a meta-circular manner, using a subset of UML notations and semantics to specify itself. The UML meta-model bootstraps itself in a similar way to how a compiler is used to compile itself. It is defined as one of a four-layer meta-modelling architecture: *meta-meta-model*, *meta-model*, *model* and *user objects*. Fig. 1 shows an example of the relation between the UML meta-model and the UML model of the printing system presented section 3. A model is an instance of the UML meta-model, each element of the UML model is an instance of a meta-class of the meta-model. For example, the *notifyStatus()* operation of the *Computer* class is an instance of the *Operation* meta-class of the UML meta-model.

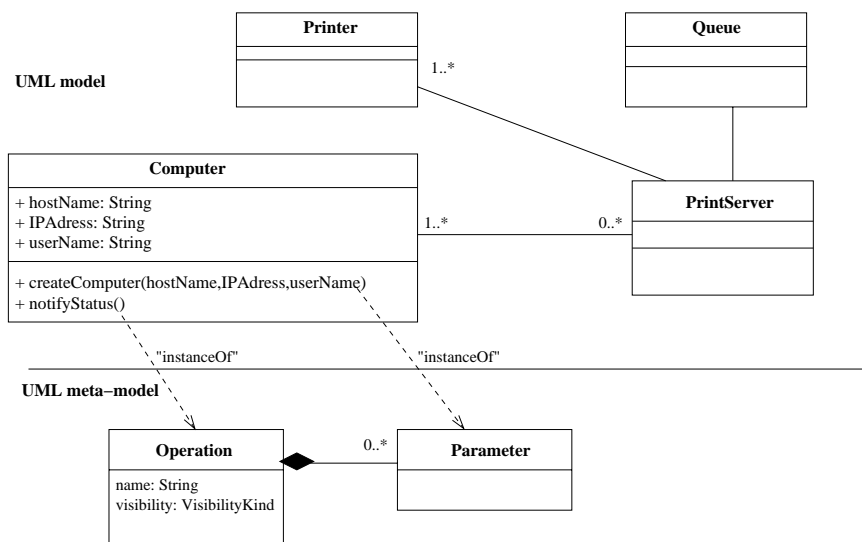


Fig. 1. Examples of relation between UML meta-model and UML model.

The meta-model is described in a semi-formal manner using three views, which are helpful to understand the UML semantics:

- Abstract Syntax. UML class diagrams are used to present the UML meta-model, its concepts (meta-classes), its relationships and its constraints.
- Well-formedness rules. A set of rules and constraints for UML model elements are defined. Rules are expressed in English prose and in the Object Constraint Language (OCL).
- Semantics. The semantics of model usage is described in English prose.

Since the meta-model layer is complex, it is decomposed into logical packages. Each package shows a strong cohesion within itself and a loose of coupling with meta-classes in other packages. The meta-model is decomposed into many packages. We focus on three packages of the meta-model: the Core package, the Collaboration package and the State Machine package which are respectively the meta-model of the class, the collaboration and the state diagrams.

3. A CASE STUDY

To illustrate our approach, we present the specification of a printing system which is in charge to print a file from a computer. This system works as follows: when a user gives a command to print a file, this command is transferred to the PrintServer. If the printer is busy, the file to print will be stored in a queue, else it will be printed. The PrintServer will notify the status of the printing process to the computer. The class diagram of this system is presented in the UML model part of Fig. 1 where we describe only the elements and properties necessary to illustrate our transformations. The collaboration diagram is described in the UML model part of Fig. 2. Each element of a collaboration diagram is an instance of a meta-class in the Collaboration package: for instance, the message 1.3 *notifyStatus* is an instance of the *Message* meta-class.

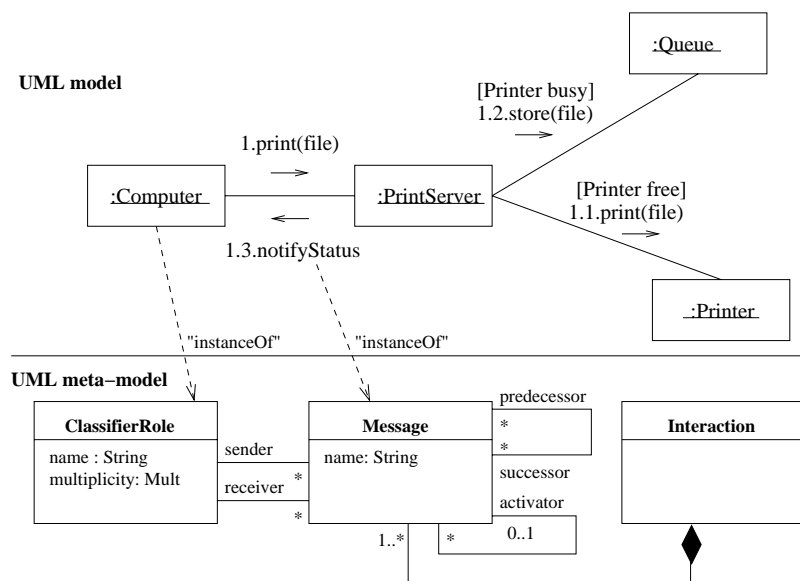


Fig. 2. Collaboration diagram of the printing system and its meta-model.

4. TRANSFORMATION OF THE UML META-MODEL INTO B

The transformation of UML diagrams into a B specification has been studied by different persons.

The transformation of an attribute of an UML class to a B variable [15] is presented as described in Fig. 3. An easy way to transform is to express attributes of classes by binary relation constructs of the set theory. However, this transformation is only applied when object identifiers will be generated in the execution of the program (in this case, if an object is created, its identifier is assigned to a random integer number because a deferred set in B is defined as a non-empty subset of integers ($OBJECTS \in P(INT)$)).

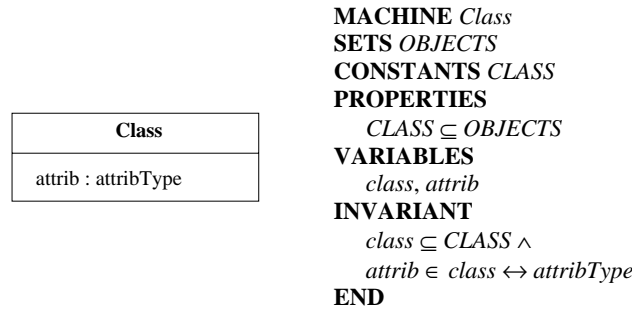


Fig. 3. Derivation of a UML class to B.

With the meta-model, it is to be noticed that the UML abstract syntax is mapped to a set of MOF packages (Meta Object Facility) called the UML Interchange Meta-model. These packages are available as an XML document which is generated from the UML Interchange Meta-model following the rules of the XML Meta-data Interchange (XMI) [16]. It is a standard for the UML models to be exchanges between tool editors of UML (Rational Rose, ArgoUML, ...) as a stream or as files. To enhance facility, we work with the XMI structure and values of attributes of XMI. That means that object identifiers of meta-classes are determined by identifiers in the XMI code generated by UML tool editors. The type of variables can be simply defined as follows:

$$attr_i \in CLASS \rightarrow TYPE(attr_i)$$

where $S \rightarrow T$ denotes the set of all partial functions from S to T , $CLASS$ is a set of object identifiers and $TYPE(attr_i)$ is the type of the attribute transformed into B. In the process of verification of the B method, support tools generate automatically proof obligations for proving predicates. These proof obligations always verify the correctness of variable values of substitution in the operations with invariants of abstract machines. When predicates in the invariant clause contains existential and/or universal quantifiers, variables of abstract machines have to contain all their potential values in order that the tool support performs the comparison and proves the predicates. With this definition of the type of variables, we introduce a new variable $attr$, typed similarly as the one of objects, in order to merge all values of variables of machine's object. The value of the variable $attr$ is a set of pairs of object identifiers mapped to the attribute values of all objects of the class:

$$attr \in CLASS \rightarrow TYPE(attr),$$

$$attr := attr_1 \cup attr_2 \cup \dots \cup attr_n,$$

$$attr = \{object_1 \mapsto value_1, object_2 \mapsto value_2, \dots, object_n \mapsto value_n\}.$$

The well-formedness rules are transformed to a B invariant. Proof obligations generated by support tools can inspect the data of all objects to verify the existential or universal quantification transformed from well-formedness rules.

The structure of the meta-model of class diagrams and behavioural diagrams is similar. Class diagrams are used to describe static properties (attributes and associations) on UML models. However, the object's attribute on the Behavioural Elements package can be valued by a set of elements, while the one of the Core package has only one value. Each attribute is transformed into a B variable the invariant of which is either a partial function or a relation.

A difference between the Behavioural Elements package and the Core package concerns well-formedness rules. In the Core package, well-formedness rules are usually simple, each rule expresses constraints for only one attribute. Well-formedness rules of the Behavioural Elements package are more complex, with many attributes which participate together into a rule. Each rule is transformed into a B invariant [20].

Definition 1 A composite class plays the role of the “whole” within a composition relationship; a composite object is an instance of a composite class.

Definition 2 A component class plays the role of the “part” within a composition relationship; a component object is an instance of a component class.

Let $attr_i$ ($i = 1 \dots m$) denoting additional variables of composite object's machine,
 $attr_{ij}$ ($i = 1 \dots m, j = 1 \dots n$), variables of machines of component objects,
 m the number of attributes of the component object and
 n the number of component objects of a composite object.

The general procedure to transform the UML meta-model of class diagrams, collaboration diagrams and state-chart diagrams into B is defined as follows:

- Each object of a meta-class (UML model element) is transformed into a B abstract machine, object attributes are transformed to variables of the abstract machine. The type of these variables is expressed in the INVARIANT clause as a partial function from the set of object identifiers to the type of the attribute:

$$attr_{ij} \in CLASS \mapsto TYPE(attr_{ij}).$$

- The value of variables will be initialized in the INITIALISATION clause with a set of the object identifier mapped to the object's attribute value:

$$attr_{ij} := \{object_j \mapsto value_{ij}\}.$$

- Machines of the composite objects contain variables which are transformed from the attributes of these objects and variables to collect values of the variables in the machines of component objects. They are typed as the one of component objects:

$$attr_i \in CLASS \rightarrow TYPE(attr_i).$$

- An extra operation is added in the OPERATIONS clause of the composite object's abstract machine to collect variables of component object's machines:

```

collectData =
PRE
   $\wedge attr_{ij} = value_{ij}$ 
THEN
   $attr_1 := attr_{11} \cup attr_{12} \cup \dots \cup attr_{1n} \parallel$ 
   $attr_2 := attr_{21} \cup attr_{22} \cup \dots \cup attr_{2n} \parallel$ 
  ...
   $attr_m := attr_{m1} \cup attr_{m2} \cup \dots \cup attr_{mn}.$ 

```

Component object's variables can be merged with additional variables of composite object's machines because they have the same type.

- The well-formedness rules of each component class in the meta-model are transformed into invariants of machines of the composite objects.

5. TRANSFORMATION AND VERIFICATION OF UML CLASS DIAGRAMS

5.1 Transformation of the UML Meta-model

We first consider the transformation of an object of the meta-class Operations of the meta-model, the *createComputer* operation, into B. An example of its XMI specification generated by UML tool editors is:

```

<UML: Operation xmi.id = "xmi.011">
  <UML: ModelElement.name> createComputer </UML: ModelElement.name>
  <UML: ModelElement.visibility xmi.value = "public"/>
  <UML: ModelElement.isSpecification xmi.value = "false"/>
  <UML: BehavioralFeature.isQuery xmi.value = "false"/>
  <UML: Operation.isRoot xmi.value = "false"/>
  <UML: Operation.isLeaf xmi.value = "false"/>
  <UML: Operation.isAbstract xmi.value = "false"/>
  <UML: Feature.owner> Here defines the parameters </UML: Feature.owner>
</UML: Operation>

```

The result of the transformation of the UML *createComputer* operation into a B abstract machine is given in Fig. 4. The specification is incomplete; this is represented by dots.

As presented in section 4, machines of composite objects contain additional variables to collect the values of variables in the machines of component objects, see Fig. 5.

```

MACHINE CreateComputer
SEES Types
VARIABLES
    createComputer_name,
    createComputer_visibility, ...
INVARIANT
    createComputer_name ∈ OPERATION → OPERATION_NAME ∧
    createComputer_visibility ∈ OPERATION → VISIBILITYKIND ∧ ...
INITIALISATION
    createComputer_name := {O11 ↦ createComputer} ||
    createComputer_visibility := {O11 ↦ Public} || ...
END

```

Fig. 4. B abstract machine for the UML *createComputer* operation.

```

MACHINE CreateComputer
...
USES CreateComputer_HostName, CreateComputer_IPAddress,
    CreateComputer_UserName
/* The structure of the abstract machine parameters is similar to the one of the CreateCom-
  puter machine in Fig. 3 */
VARIABLES
    parameter_name,
    parameter_direction, ...
INVARIANT
    parameter_name ∈ PARAMMETER → PARAMETERS_NAME ∧
    parameter_direction ∈ PARAMMETR → DIRECTIONKIND ∧ ...
INITIALISATION
    parameter_name := ∅ ||
    parameter_direction := ∅ || ...
OPERATIONS
    collectData =
        pre
            hostName_name = {P1 ↦ hostName} ∧
            hostName_direction = {P1 ↦ in} ∧
            /* from CreateComputer_HostName machine */
            ipAdress_name = {P2 ↦ ipAdress} ∧
            ipAdress_direction = {P2 ↦ in} ∧
            /* from CreateComputer_IPAddress machine */
            userName_name = {P3 ↦ userName} ∧
            userName_direction = {P3 ↦ in} ∧ ...
            /* from CreateComputer_UserName machine */
        then
            parameter_name :=
                hostName_name ∪ ipAdress_name ∪ username_name ||
            parameter_direction :=
                hostName_direction ∪ ipAdress_direction ∪ username_direction || ...
        end
END

```

Fig. 5. Additional part for the *CreateComputer* abstract machine.

Each parameter of the operation *createComputer* is transformed into a B abstract machine (CreateComputer_HostName, CreateComputer_IPAdress, CreateComputer_UserName), the structure of these machines is similar to the one of the CreateComputer machine. In the UML meta-model, the Parameter meta-class is a component of the Operations meta-class.

Based on the structure of the UML meta-model represented by the XMI structure in the left part of Fig. 6, the general structure of B abstract machines transformed from UML model elements of the printing system's class diagram is presented in the right part of Fig. 6. The machine of Model uses the machines of objects of the Association class¹ (Computer_PrintServer, PrintServer_Printer, ...) and machines of objects of the Class class (Computer, PrintServer, Queue, Printer). The machines of objects of the Association class (Computer_PrintServer) use the machines of objects of the AssociationEnd class (Computer_PrintServer_computer, Computer_PrintServer_printserver). Each machine in the system sees the *Types* machine which defines all the sets of the system (the members of these sets are extracted from the XMI specification of the meta-model of the UML class diagram):

```

CLASS = {C1, C2, C3, C4}; /* xmi.id = C1, ... */
OPERATION = {O11, O12}; /* xmi.id = O11, ... */
VISIBILITYKIND = {public, privated, protected};
DIRECTIONKIND = {in, out, inout}; ....
    
```

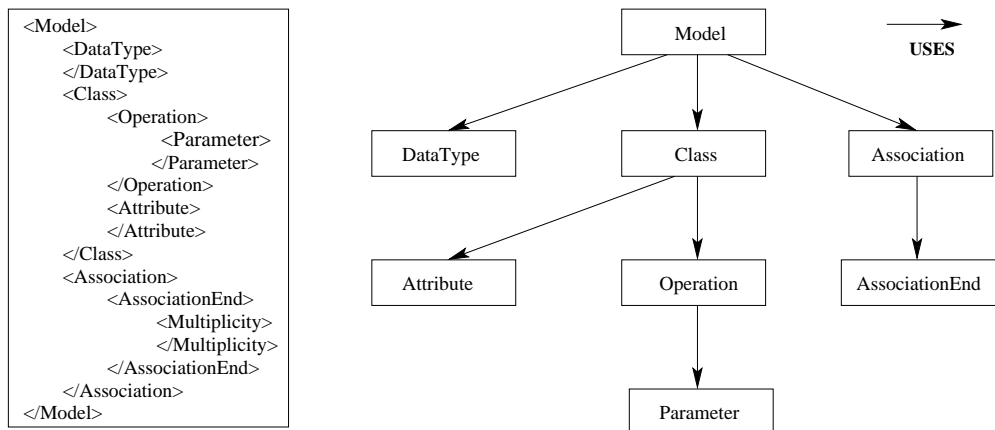


Fig. 6. General structure of the UML meta-model of class diagrams and their transformation to B.

Remarks: Abstract machines of the objects of the Multiplicity class are combined with the abstract machines of objects of AssociationEnd class to become one kind of machines: abstract machines of objects of AssociationEnd class. The attributes of objects of the Multiplicity class are transformed into variables of abstract machines of objects of the

¹ As the associations have no name, we give a name composed of the name of the two classes that are connected by the association.

AssociationEnd class. The goal of this transformation is to collect data and to work with the well-formedness rules for the verification of the Association machine.

5.2 Verification of UML Model Elements

With the proposed transformations, we can keep the structure of the machines similar to the structure of meta-classes in the UML meta-model. Well-formedness rules are transformed into invariants of abstract machines; this allows us to use the B theorem prover. Let us consider an example of well-formedness rules of the Core package:

Rule WFR1. All Parameters should have a unique name.

Self.parameter \rightarrow forAll ($p1, p2 \mid p1.name = p2.name$ implies $p1 = p2$).

This OCL predicate can be transformed to the next B invariant:

$$\begin{aligned} \forall (xx, yy). (xx \in PARAMETER_NAME \wedge \\ yy \in PARAMETER_NAME \wedge xx = yy \\ \Rightarrow parameter_name^{-1}(xx) = parameter_name^{-1}(yy)). \end{aligned}$$

This well-formedness rule of the Parameter meta-class is included in the abstract machine of the composite objects (objects of the Operation class). In the case study, it is included in the abstract machine of the *createComputer* operation presented Fig. 5, with a renaming of the attribute *name* to *parameter_name*. The *PARAMETER* and *PARAMETER_NAME* sets are defined in the *Types machine* as follows:

$$\begin{aligned} PARAMETER &= \{P1, P2, P3\}; \\ PARAMETER_NAME &= \{hostName, ipAddress, userName\}. \end{aligned}$$

The proof obligation that guarantees the preservation of the invariant for each B operation of an abstract machine is of the form $I \wedge P \Rightarrow [S]I$ where

P: is the precondition of the operation,
S: the body of the operation and
I: the invariant of the abstract machine.

The proof obligation generated by the B prover for the *collectData* operation of the *CreateComputer* abstract machine is:

$$\begin{aligned} \forall (xx, yy). (xx \in \{hostName, ipAddress, userName\} \wedge \\ yy \in \{hostName, ipAddress, userName\} \wedge \\ xx = yy \Rightarrow \\ (\{P1 \mapsto hostName, P2 \mapsto ipAddress, P3 \mapsto userName\}^{-1}(xx) = \\ (\{P1 \mapsto hostName, P2 \mapsto ipAddress, P3 \mapsto userName\}^{-1}(yy))). \end{aligned}$$

The result of this predicate is true.

6. TRANSFORMATION AND VERIFICATION OF UML COLLABORATION DIAGRAMS

The meta-model of UML collaboration diagrams, the Collaboration package, is a sub-package of the Behavioural Elements package. It specifies concepts needed to express how different elements from a structural view of a model interact with each other.

6.1 Transformation of the UML Meta-model

Following the transformation procedure presented in section 4, we transform the meta-model of UML collaboration diagrams into B.

The type of variables transformed from attributes of objects which contains a set of elements is defined as a relation. To illustrate this transformation on the printing system presented Fig. 2, we introduce B abstract machines of objects of the Message and Interaction classes. Four instances are identified for the Message meta-class: 1, 1.1, 1.2, 1.3. Based on the procedure of transformation presented in section 4, each instance of the Message class is transformed into a B abstract machine, named *Message1*, *Message11*, *Message12* and *Message13*. The *Interaction* meta-class of this case study has only one object. It is transformed into the B abstract machine presented Fig. 7. The *Interaction* abstract machine contains variables transformed from the attributes of the interaction object (prefixed with *interaction*) and variables to merge the data of the variables in the abstract machines of objects of the Message class (prefixed with *message*). This merging is realized by the operation *collectData*.

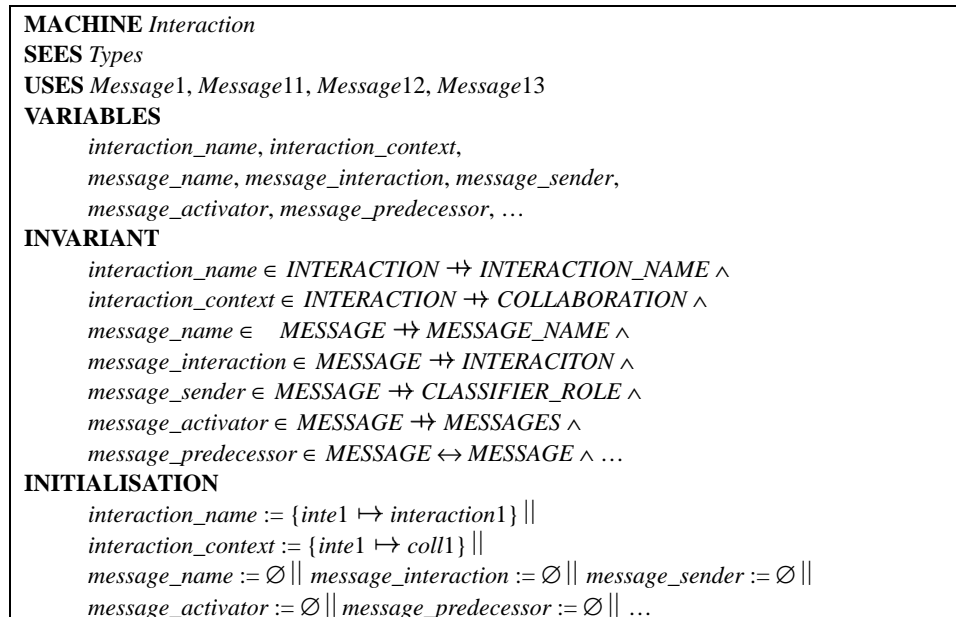


Fig. 7. Interaction B abstract machine.

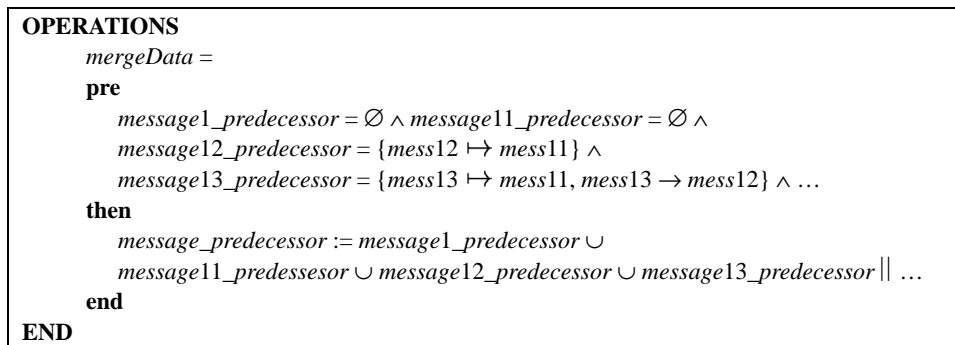


Fig. 7. (Cont'd) Interaction B abstract machine.

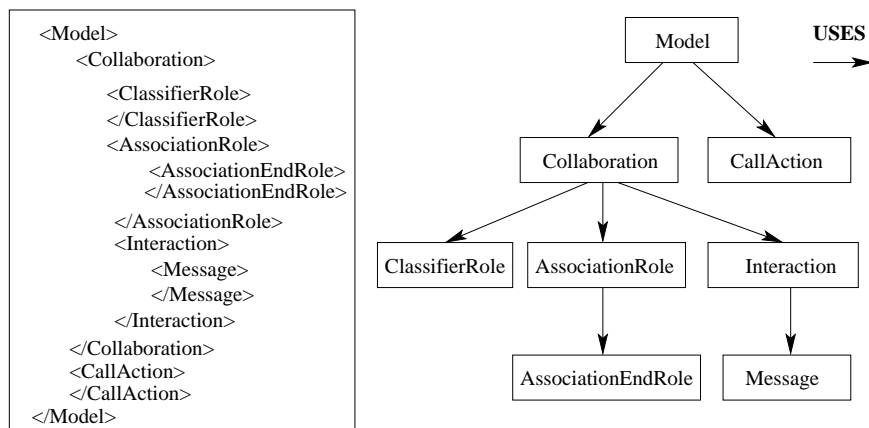


Fig. 8. General structure of the UML meta-model of collaboration diagrams and transformation into B.

The machines of component objects in the UML meta-model are used by the machine of the composite objects (in the XMI specification, component classes are expressed by siblings, composite classes are expressed by parent). The general structure of B machines transformed from the meta-model of a UML collaboration diagram is presented Fig. 8.

The left hand part of the figure gives the XMI summary description of a collaboration diagram on UML models. The right hand part is the structure of corresponding B abstract machines. The machine of object of the Model class uses (USES) the machines of objects of the Collaboration class and the CallAction class; the machines of objects of the Collaboration class uses the machines of objects of the ClassifierRole class, the AssociationRole class and the Interaction class and so on.

Each machine in the system sees the *Types machine* which defines all sets of the system. In our case study, these sets are:

$$MESSAGE = \{mess1, mess11, mess12, mess13\};$$

$MESSAGE_NAME = \{print, restore, notifyStatus\};$
 $CLASSIFIER_ROLE = \{class1, class2, class3, class4\};$
 $CLASSIFIER_ROLE_NAME = \{Computer, PrintServer, Queue, Printer\}; \dots$

6.2 Verification of UML Model Elements

Let's consider the transformation of well-formedness rules of the Messages class and the verification of UML model elements of the Collaboration package which must satisfy these rules.

Rule WFR2. The predecessors and the activator must be contained in the same Interaction.

$self.predecessor \rightarrow \text{forAll}(p \mid p.interaction = self.interaction)$
 and
 $self.activator \rightarrow \text{forAll}(a \mid a.interaction = self.interaction).$

This OCL predicate can be transformed to the next B invariant:

$$\begin{aligned} \forall pp.(pp \in MESSAGE \wedge message_predecessor[\{pp\}] \neq \emptyset & \quad WFR2a \\ \Rightarrow message_interaction[message_predecessor[\{pp\}]] = & \\ & message_interaction[\{pp\}] \wedge \\ \forall aa.(aa \in MESSAGE \wedge message_activator[\{aa\}] \neq \emptyset & \\ \Rightarrow message_interaction[message_interaction[\{aa\}]] = & \quad WFR2b \\ & message_interaction[\{aa\}]). \end{aligned}$$

Applying this rule on the case study of the printing system and taking into account the values of the variables, we have:

$MESSAGE = \{mess1, mess11, mess12, mess13\}.$

The values of $message_predecessor$, $message_activator$ and $message_interaction$ sets established by the $collectData$ operation of the $Interaction$ machine are:

$message_predecessor = \{mess12 \mapsto mess11, mess13 \mapsto mess11, mess13 \mapsto mess12\};$
 $message_activator = \{mess11 \mapsto mess1, mess12 \mapsto mess1, mess13 \mapsto mess1\};$
 $message_interaction = \{mess1 \mapsto inte1, mess11 \mapsto inte1, mess12 \mapsto inte1,$
 $mess13 \mapsto inte1\}.$

Let us examine the proof of this rule.

Proof of WRF2a: Let us examine each value of pp .

$pp = mess1, message_predecessor[\{mess1\}] = \emptyset$

$pp = mess11, message_predecessor[\{mess11\}] = \emptyset$

With these two cases, the hypothesis of $WRF2a$ is not satisfied, so $WRF2a = true$

$pp = mess12, message_predecessor[\{mess12\}] = \{mess11\}$

then $message_interaction[\{mess1\}] = \{inte1\}$
 So $message_interaction[message_predecessor[\{mess1\}]] = \{inte1\}$
 On the other hand, $message_interaction[\{mess12\}] = \{inte1\}$
 $\Rightarrow WFR2a = true$
 $pp = mess13, message_predecessor[\{mess13\}] = \{mess11, mess12\}$
 Note that: $ran(u \triangleleft r) = r[u]$ with $u \subseteq s \wedge r \in s \leftrightarrow t$ (See *The B-Book* [1], p.102)
 then $message_interaction[\{mess11, mess12\}]$
 $= ran(\{mess11, mess12\} \triangleleft message_interaction)$
 $= ran(\{mess11 \mapsto inte1, mess12 \mapsto inte1\}) = \{inte1\}$
 and $message_interaction[\{mess13\}] = \{inte1\} \Rightarrow WFR2a = true$
 We deduce that $WFR2a = true$ for each value of pp .

Proof of WFR2b:

$aa = mess1, message_activator[\{mess1\}] = \emptyset$
 $aa = mess11$ or $aa = mess12$ or $aa = mess13$
 then $message_activator[\{aa\}] = \{mess1\}$
 then $message_interaction[\{mess1\}] = \{inte1\}$
 and $message_interaction[\{aa\}] = \{inte1\} \Rightarrow WFR2b = true$
 As a consequence, we have: $WFR2 = WFR2a \wedge WFR2b = true$.

The verification of these well-formedness rules have been executed by the support tool AtelierB [19], which can both automatically and interactively demonstrate theorems.

7. TRANSFORMATION AND VERIFICATION OF UML STATE DIAGRAMS

The metamodel of state diagrams called the State Machine package, is a subpackage of the Behavioural Elements package. The State Machine package depends on concepts defined in the Common Behaviour package, enabling integration with other sub-packages in Behavioural Elements. The procedure of transformation of the State Machine package into B is similar to the one of the Collaboration package. Based on the structure of the State Machine package, the structure of B abstract machines is composed as presented in Fig. 9.

The verification of the semantics of state diagrams is similar to the one of collaboration diagrams.

8. CONCLUSION

We have presented a technique to transform the meta-model of UML class diagrams, collaboration diagrams, state diagrams and their well-formedness rules into B formal specifications. This transformation aims to verify the UML model elements which must satisfy the well-formedness rules of UML semantics.

By exploiting the advantages of formal approaches for the verification, our approach owns powerful provers like AtelierB. In addition, OCL used to specify well-

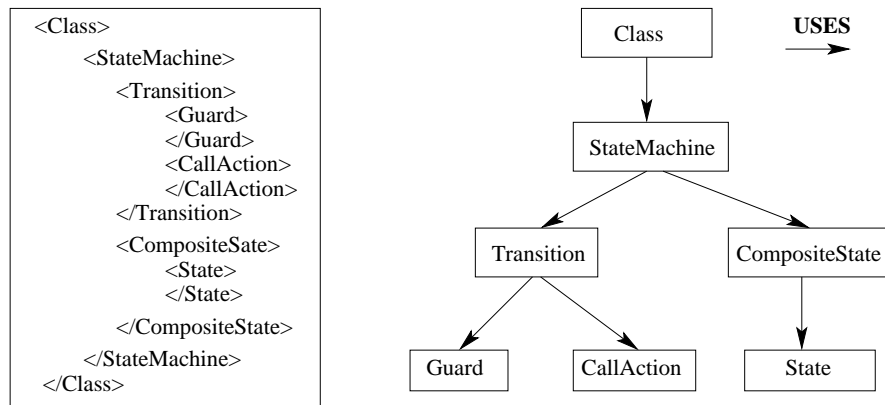


Fig. 9. General structure of the UML meta-model of state diagrams and their transformation to B.

formedness rules of UML semantics and B notations are based on the first order predicates logic so their reciprocal transformation is easy. Furthermore, B is based on the set theory, the relation between classes and their objects in UML is similar to the relation between sets and their elements. Operations on attributes of classes correspond to operations on binary relation constructs in the set theory. The proof by the B provers is automatically and easily performed.

A prototype ArgoUML+B [14] has been developed from ArgoUML², a free available platform for editing UML diagrams. This prototype automatically transforms UML diagrams (class, state, collaboration) into B. Furthermore, the internal representation of an UML model is completely generated from the specification, that means that values of objects in the UML metamodel are saved as XMI code. We continue to develop this prototype to automatically generate B abstract machines from the XMI support.

REFERENCES

1. J. R. Abrial, *The B-Book, Assigning Programs to Meanings*, Cambridge University Press, 1996.
2. J. R. Abrial and D. Cansell, "Click'n prove: interactive proofs within set theory," in *Proceedings of 16th International Conference on Theorem Proving in Higher Order Logics*, LNCS 2758, 2003, pp. 1-24.
3. Abstract State Machine, Available at <http://www.eecs.ummich.edu/gasm>.
4. B-Core (UK) Ltd., *B-Toolkit User's Manual*, Oxford (UK), 1996, Release 3.2.
5. D. Bert, S. Boulmé, M. L. Potet, A. Requet, and L. Voisin, "Adaptable translator of B specifications to embedded C programs," in *Proceedings of the International Symposium of Formal Method*, LNCS 2805, Springer-Verlag, 2003, pp. 94-113.
6. G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1998.

²<http://argouml.tigris.org>.

7. A. Cavarra, E. Riccobene, and P. Scandurra, "A framework to simulate UML models: moving from a semi-formal to a formal environment," in *Proceedings of the ACM Symposium in Applied Computing*, 2004, pp. 1519-1523.
8. Clearsy, "b4free," available at <http://www.b4free.com>, 2004.
9. P. Facon, R. Laleau, and H. P. Nguyen, "Mapping object diagram into B," in *Proceedings of Methods Integration Workshop*, 1996, available at <http://ewic.bcs.org/conferences/1996/methodsintegration/papers/paper6.htm>.
10. H. LeDang and J. Souquière, "Contributions for modelling UML state-charts in B," in *Proceedings of 3rd International Conference on Integrated Formal Methods*, LNCS 2335, Springer-Verlag, 2002, pp. 109-127.
11. M. G. Hinchey and J. P. Bowen, *Applications of Formal Methods*, Prentice Hall, 1995.
12. S. K. Kim and D. Carrington, "A formal mapping between UML models and object-Z specifications," in *Proceedings of ZB 2000 Formal Specification and Development in Z and B*, LNCS 1878, Springer-Verlag, 2000, pp. 2-21.
13. R. Laleau and F. Polack, "Metamodels for static conceptual modelling of information system," in *Workshop on Defining Precise Semantics of UML, ECOOP*, 2000, available at citeseer.ist.psu.edu/laleau00metamodels.html.
14. H. Ledang, J. Souquière, and S. Charles, "ArgoUML+B: un outil de transformation systématique de spécifications UML vers B," in *Proceedings of Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL 2003)*, 2003, pp. 35-49.
15. E. Meyer and J. Souquière, "A systematic approach to transform OMT diagrams to a B specification," in *Proceedings of the Formal Method Conference*, LNCS 1708, Springer-Verlag, 1999, pp. 875-895.
16. OMG, *Unified Modeling Language*, OMG <http://www.omg.org/docs/formal/03-03-01.pdf>, Version 1.5, 2003.
17. P. Behm, P. Benoit, and J. M. Meynadier, "METEOR: a successful application of B in a large project," in *Proceedings of Integrated Formal Methods*, LNCS 1708, Springer-Verlag, 1999, pp. 369-387.
18. M. Richters, "A precise approach to validating UML models and OCL constraints," Ph.D. thesis, Dept. of Fachbereich 3, Bremen University, 2002.
19. Steria, "Obligations de preuve: manuel de reference," *Steria – Technologies de l'information*, version 3.0, Tool is available at <http://www.atelierb.societe.com>.
20. N. T. Truong and J. Souquière, "An approach for the verification of UML models using B," in *Proceedings of 11th International Conference of Engineering of Computer Based Systems (ECBS)*, 2004, pp. 195-202.
21. N. T. Truong and J. Souquière, "Verification of behavioral elements of UML models using B," in *Proceedings of 20th Annual ACM Symposium on Applied Computing (SAC '05)*, 2005, pp. 1546-1552.
22. J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UL*, Addison-Wesley, 1999.



Ninh Thuan Truong received his B.S degree in Mathematics and Informatics from National University of Hanoi in 1999 and his M.S. degree in Computer Science from Institute Francophone for Computer Science (IFI) of Hanoi, Vietnam in 2002. He is Ph.D student of the University Nancy 2, France. His research interests include software specification and verification, formal methods and theorem proving. His thesis dissertates on the specification and verification of object-based systems using the B formal method.



Jeanine Souquière is Professor in Computer Science of the University Nancy 2, France. Her research interests include software requirements, formal and semi-formal specifications, component based approaches including verification aspects.