HAL
open science

Verification of UML Model Elements Using B

Ninh Thuan Truong, Jeanine Souquières

To cite this version:

Ninh Thuan Truong, Jeanine Souquières. Verification of UML Model Elements Using B. Journal of Information Science and Engineering, 2006, 22, pp.357-373. hal-00097566

HAL Id: hal-00097566

https://hal.science/hal-00097566

Submitted on 30 Oct 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Verification of UML model elements using B

Abstract

This paper describes the formal verification of UML model elements using B abstract machines. We study the UML metamodel of class diagrams, collaboration diagrams and state-chart diagrams as well as their well-formedness rules. Each element of UML models which is an instance of a metaclass, is transformed into a B abstract machine. The relationship between abstract machines is organised using the abstract syntax of UML class diagram of the UML metamodel. B specifications are proved by a B prover which generates automatically proof obligations, allowing UML model elements to be verified. The correctness of the UML model elements is ensured by the well-formedness rules which are transformed to B invariants. We illustrate our approach by a simple case study, the printing system.

Keywords: B method, UML models, UML metamodel, formal verification, well-formedness rules

1 Introduction

The application of formal methods [9] allows the rigorous definition and analysis of the functionality and the behaviour of a system. Starting from rigorous specifications, formal methods can be used for the derivation of test cases, or for a validation and verification technique aimed at proving that the specification satisfies the requirements.

The Unified Modelling Language (UML) [4] is a widely accepted modelling language that can be used to visualise, specify, construct and document the artifacts of a software system. It has been accepted as a standard object-oriented modelling language by OMG [14], 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 its metamodel. In the UML metamodel [14], modelling constructs are defined using three distinct views: an abstract syntax in UML class diagrams, static semantics ensuring that all UML constructs are statically well formed in OCL [20], and dynamic semantics specifying the meaning of the constructs, mainly in English.

The derivation from UML specifications into the B formal method [1] is considered as an appropriate way to jointly use UML and B in practical, unified and rigorous software development. The transformation from UML diagrams to B specification have been considered in [8, 13, 7]; these approaches provide a multi-view framework for the specification of a system but do not allow the complete verification of the properties of UML semantics. The transformation of the UML metamodel to formal methods (Object-Z, B) has been considered by K. Soon-Kyeong, C. David in [10] and

R. Laleau, F. Polack in [11]. However, these approaches only consider the specification process but do not allow the checking of UML model elements with support tools. Cavarra *et al.* [5] build a framework to transform UML metamodel and UML models into ASM (Abstract State Machines) [2] but the verification of this work is not presented. Concerning the validation of UML models and OCL constraints, M. Richter *et al.* [15] present an approach based on animation. They developed a tool called USE (UML-based Specification Environment) which is 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 meta-classes of an UML specification, objects of these classes (which are elements of UML models) and well-formedness rules of UML semantics proposed by OMG [14] 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 [18]. The transformation of behavioural diagrams is presented in [19]. This paper integrates these derivations in a common approach to verify UML model elements of 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 the transformation and verification. Section 4 presents our principal contribution; we compare the difference of well-formedness rules between packages and propose a general principle to transform the UML metamodel into B. In the three next sections, we present the transformation of the metamodel of class diagrams, collaboration diagrams and statechart 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. Section 8 ends with some concluding remarks.

2 Background

In this section, we introduce the B method, the UML metamodel and its relation with UML models.

2.1 The B method

B [1, 16] is a formal software development method, originally developed by J. R. Abrial. The B notation is based on set theory, the language of generalised substitutions and first order logic. It is one of few formal methods which has robust, commercially available support tools for the entire development life-cycle from specification through to code generation [17]. Specifications are composed of abstract machines, similar to modules or classes. Each abstract machine consists of a set of variables, invariant properties relating to those variables and operations.
Variables of B specifications are strictly typed. Type of variables is not given explicitly as in the majority of language programming but is presented in the invariants clause. The state of the system, i.e. the set of variable values, is only modifiable by operations which must preserve its invariants.
With the refinement mechanism, an abstract specification can evolve to a more concrete specification, by adding new data or operations, allowing the behaviour of the

abstract system to be 'simulated' by the refined system. At every stage of the specification, proof obligations ensure the preservation of the system invariant. These proof obligations are generated automatically by support tools like AtelierB [17], B-Tookit [3] and B4free [6]. Analysing proof obligations with B support tools is an efficient and practical way to detect errors encountered during the specification development. B provides structuring primitives like INCLUDES, IMPORT, USES and SEES allowing users to compose abstract machines. Thus, large systems can be specified in a modular way and can be reused. In this paper, we will use these advantages of the B method to specify UML model elements and to check them with UML semantics.

2.2 UML metamodel and its relation with UML models

The UML metamodel [14] defines the complete semantics for representing object models using UML. It is defined in a metacircular manner, using a subset of UML notations and semantics to specify itself. In this way the UML metamodel bootstraps itself in a similar way to how a compiler is used to compile itself. The UML metamodel is defined as one of the four-layer metamodelling architecture: *meta-metamodel*, *metamodel*, *model* and *user objects*. Figure 1 shows an example of the relation between the UML metamodel and the UML model of the printing system. A model is an instance of the UML metamodel, each element of the UML model is an instance of a metaclass of the metamodel, for example, the *notifyStatus*() operation of the *Computer* class is an instance of the *Operation* metaclass of the UML metamodel.

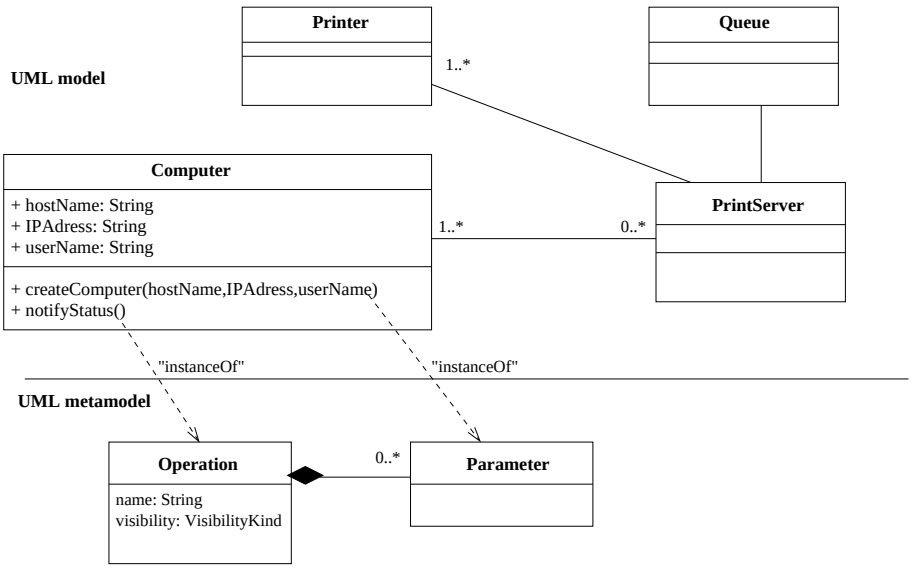


Figure 1: Example of relation between UML metamodel and UML model

The metamodel 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 metamodel, its concepts (meta-classes), relationships and constraints.

- Well-Formedness Rules. A set of rules and constraints for UML model elements are defined. Rules are expressed in an English prose and in the Object Constraint Language (OCL). OCL [20] is a specification language that uses logic for specifying invariant properties of systems.
- Semantics. The semantics of model usage is described in an English prose.

Since the metamodel layer is relatively complex, it is decomposed into logical packages. Each package show strong cohesion with each other and loose coupling with meta-classes in other packages. The metamodel is decomposed into many packages. We focus on three packages of the metamodel: the Core package, the Collaboration package and the State Machine package which are respectively the metamodel of the class diagrams, the collaboration diagrams and the statechart diagrams.

3 A case study

To illustrate our approach, we present the specification of the printing system, which is a system to print a file from a computer. This system works as follows: when a user gives a command to print a file, this command will be transfered to the PrintServer. If the printer is busy, the file to print will be stored in a queue, else it will be printed and 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 Figure 1. In this diagram, we only describe the elements and properties necessary to illustrate the transformation. The collaboration diagram is showed in the UML model part of Figure 2.

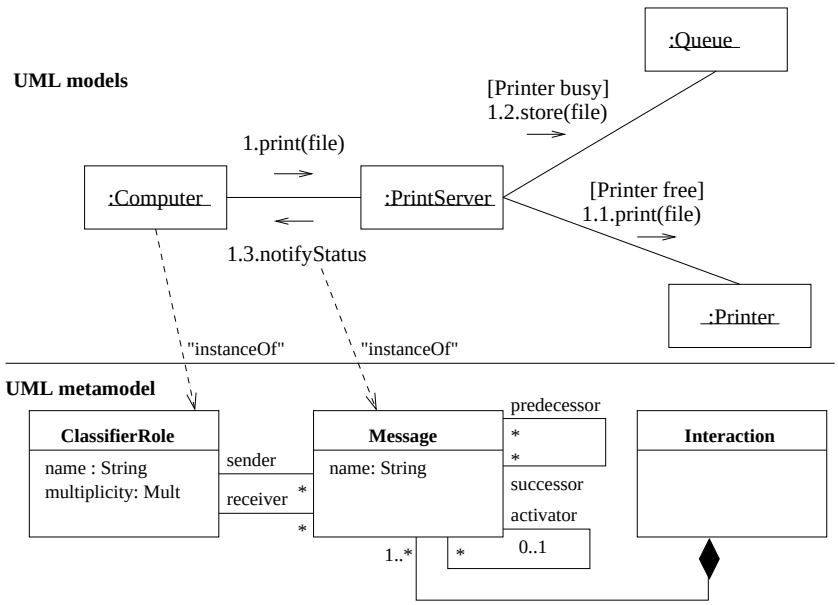


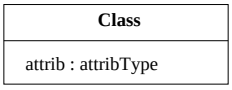
Figure 2: Collaboration diagram of the printing system and its metamodel

As for the class diagram, there is a package of the metamodel layer called the Collaboration package which is used to define collaboration diagram elements. Each element

of a collaboration diagram is an instance of a metaclass in the Collaboration package. For instance, the message 1.3 *notifyStatus* of the collaboration diagram of the printing system is an instance of the *Message* metaclass of the UML metamodel (see Figure 2).

4 Transformation of the UML metamodel into B

Many approaches of the transformation of UML diagrams into a B specification are proposed. In these approaches, the transformation of an attribute of an UML class to a variable of a B abstract machine is usually presented as follows [13]:



MACHINE *CLASS*
SETS *OBJECTS*
CONSTANTS *CLASS*
PROPERTIES
$CLASS \subseteq OBJECTS$
VARIABLES
$class,\ attrib$
INVARIANT
$class \subseteq CLASS \wedge$
$attrib \in class \leftrightarrow attribType$
....
END

With this transformation, it is easy 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 ($OBJECTS \in P(INT)$)).
With the metamodel, 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 Metamodel. These packages are available as an XML document which is generated from the UML Interchange Metamodel following the rules of the XML Metadata Interchange (XMI) [14]. It is a standard for the UML models to be exchanged 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. Hence we can simply define the type of variables as follows:

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

where $S \nrightarrow T$ 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. One usage of this expression is its application for the verification. 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 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:

$$attr \in CLASS \nrightarrow TYPE(attr)$$
$$attr := attr_1 \cup attr_2 \cup ... \cup attr_n$$

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:

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

With the *attr* set, considered as a variable, we can express the well-formedness rules as an invariant of abstract machines. Proof obligations generated by support tools can inspect the data of all objects to verify the existential and/or universal quantification transformed from well-formedness rules.

The structure of the metamodel of class diagrams and behavioural diagrams (composed of collaboration and statechart 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 (Collaboration and State Machine packages) can be valued by a set of elements, meanwhile the one of the Core package has only one value. Depending on the value of attributes, each attribute is transformed to B as a variable whose invariant is either a partial function or a relation between the type of the class in which it is introduced and the type of attributes.
Another difference between the Behavioural Elements package and the Core package is 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 in a rule. To verify the correctness of each rule (transformed as an invariant of a B abstract machine), values of all the variables which participate in invariants of B abstract machines should be determined, furthermore, all these variables must be assigned to values at the same time, the B prover thus inspects all values of variables to prove predicates. In the approach of the transformation of the Core package to B [18], we merge the data of the variables on the separate operations of the B abstract machine. This approach cannot be applied to the transformation of the Behavioural Elements package, because variables which participate in the same invariant may not be assigned to the values at the same time and hence the result of the proof can be incorrect. To solve this problem, we propose a common approach applied to the transformation of the Core package as well as the Behavioural Elements package. Before the presentation of the procedure of transformation, let's see some definitions:

Definition 1
A composite class is a class that serves as the "whole" within a composition relationship; a composite object is an instance of a composite class.

Definition 2
A component class is a class that serves as the "part" within a composition relationship; a component object is an instance of a component class.

The procedure of transformation 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 on these variables is expressed in the INVARIANT clause as a partial function from the set of object identifiers to the type of the attribute:

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

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

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

- Machines of the composite objects contain not only the variables which are transformed from the attributes of these objects, but also variables to merge values of the variables in the machines of component objects. These variables are typed identically to the one of component objects:

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

- We add an extra operation in the OPERATIONS clause of the composite object's abstract machine to merge variables of component object's machines into additional variables of composite object's machines:

  $$\begin{array}{l} mergeData = \\ PRE \\ \quad \bigwedge attr_{ij} = value_{ij} \\ THEN \\ \quad attr_1 := attr_{11} \cup attr_{12} \cup ... \cup attr_{1n} \;||\\ \quad attr_2 := attr_{21} \cup attr_{22} \cup ... \cup attr_{2n} \;||\\ \quad ... \\ \quad attr_m := attr_{m1} \cup attr_{m2} \cup ... \cup attr_{mn} \end{array}$$

  where : $attr_i$ $(i = 1..m)$: additional variables of composite object's machine,
  $attr_{ij}$ $(i = 1..m, j = 1..n)$: variables of machines of component objects,
  m is the number of attributes of the component objects,
  n is the number of component objects of a composite object.

  The substitution above allows component object's variables to be merged because the type of additional variables of composite object's machines is the same as the one of component object's machines.

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

This is the main procedure which allows us to transform the UML metamodel of class diagrams, collaboration diagrams and statechart diagrams into B.

5 Transformation of the metamodel of UML class diagrams into B

Based on the previous procedure of the transformation, we perform a transformation of the metamodel of UML class diagram into B, illustrated by the printing system presented in Figure 1 in order to verify UML model elements.

5.1 Transformation of the UML metamodel

First, we consider the transformation of an object of the metaclass Operations of the metamodel, the *createComputer* operation, into B. An example of its XMI specification generated in UML tool editors is presented as follows:

```
<UML:Operation xmi.id="xmi.O11">
  <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 to a B abstract machine is given in Figure 3.

MACHINE *CreateComputer*
SEES *Types*
VARIABLES
$createComputer_name$,
$createComputer_visibility$,
...
INVARIANT
$createComputer_name \in OPERATION \nrightarrow OPERATION_NAME \wedge$
$createComputer_visibility \in OPERATION \nrightarrow VISIBILITYKIND \wedge$
...
INITIALISATION
$createComputer_name := \{O11 \mapsto createComputer\} \parallel$
$createComputer_visibility := \{O11 \mapsto public\} \parallel$
...
END

Figure 3: B abstract machine for the UML *createComputer* operation

As presented in the procedure of transformation in the Section 4, machines of composite objects contain not only variables which are transformed from attributes of these objects, but additional variables to merge values of variables in the machines of component objects. Note that, 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 presented Figure 3. In the UML metamodel, the Parameters metaclass is a component of the Operations metaclass. The abstract machine of the operation *createComputer* have to contain additional variables to merge the values of variables in machines of its parameters. The additional part (which composed of additional variables and the *mergeData* operation) for the specification of the *CreateComputer* abstract machine is presented Figure 4.

Based on the structure of the UML metamodel represented by the XMI structure in the left part of Figure 5, the general structure of B abstract machines transformed from UML model elements of the printing system's class diagram is presented as follows (see the right part of Figure 5):

```
MACHINE CreateComputer
...
USES CreateComputer_HostName, CreateComputer_IPAdress,
        CreateComputer_UserName
   /* The structure of the abstract machine parameters is similar to
   the one of the CreateComputer machine in Figure 3 */
VARIABLES
   parameter_name,
   parameter_direction,
   ...
INVARIANT
  parameter_name ∈ PARAMETER ⇸ PARAMETERS_NAME ∧
  parameter_direction ∈ PARAMETER ⇸ DIRECTIONKIND ∧
  ...
INITIALISATION
  parameter_name := ∅ ||
  parameter_direction := ∅ ||
  ...
OPERATIONS
  mergeData =
     pre
     hostName_name = {P1 ↦ hostName} ∧
     hostName_direction = {P1 ↦ in} ∧
                    /* from CreateComputer_HostName machine */
     ipAdress_name = {P2 ↦ ipAdress} ∧
     hostName_direction = {P2 ↦ in} ∧
                    /* from CreateComputer_IPAdress machine */
     userName_name = {P3 ↦ userName} ∧
     hostName_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
```

Figure 4: Additional part for the *CreateComputer* abstract machine

The machine of Model uses the machines of objects of the Association class [1](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).

[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.

The machines of objects of Class class (Computer) use the machines of objects of the Attribute class (Computer_HostName, Computer_IPAdress, Computer_UserName) and machines of objects of the Operation class (Computer_createComputer, Computer_notifyStatus) (their names are prefixed by the name of the class). The machines of objects of the Operation class (Computer_CreateComputer) use the machines of objects of the Parameter class (CreateComputer_HostName, CreateComputer_IPAdress, CreateComputer_UserName) (their names are prefixed by the name of the operation to distinguish them with the name of Attribute's machines).
All machines in the system see the Types machine which defines all the sets of the system (members of these sets are extracted from the XMI specification of the metamodel of the UML class diagram):

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

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 machine, abstract machines of objects of AssociationEnd class. The attributes of objects of the Multiplicity class are transformed to variables of abstract machines of objects of the AssociationEnd class. The goal of this transformation is to merge data and to work with the well-formedness rules for the verification of the Association machine.

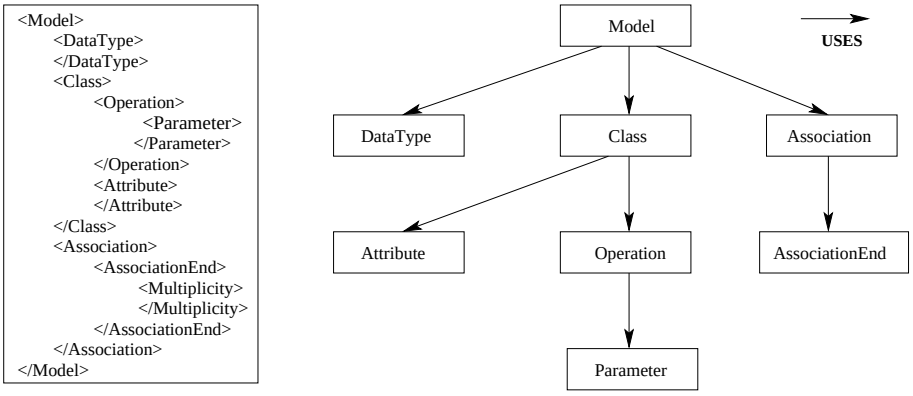


Figure 5: General structure of the UML metamodel of class diagrams and their transformation to B

With the arrangement of the above-mentioned abstract machines' structure, we can keep the structure of the machines corresponding to metaclasses in the UML metamodel. It is clear and simple, furthermore, we can also use the well-formedness rules such as invariants of abstract machines and exploit the B theorem prover to prove their own correctness.

5.2 Verification of UML model elements

Let's consider a well-formedness rule of the Core package of the UML metamodel:

```
Rule WFR1:   All Parameters should have a unique name
self.parameter -> forAll(p1,p2 | p1.name = p2.name implies p1 = p2)
```

This OCL predicate can be transformed to a B invariant as presented in Figure 6. This well-formedness rule of the Parameter meta-class is included in the abstract

INVARIANT

...

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

Figure 6: Transformation of the well-formedness rule $WFR1$ to B

machine of the composite objects (objects of the Operation class). In this case, it is presented in the abstract machine of the *createComputer* operation (Figure 4), with a renaming of the attribute *name* to *parameter_name* to have the same notation as for the variable of this abstract machine.
The *PARAMETER* and *PARAMETER_NAME* sets are defined in the Types machine as follows:

$PARAMETER = \{P1, P2, P3\}$;
$PARAMETER_NAME = \{hostName, ipAdress, userName\}$

One of the proof obligations that the B method proposes in an abstract machine is of the form $I \wedge P \Rightarrow [S]I$ where

P: precondition of the operation
S: body of the operation
I: invariant of the abstract machine

Applying this proof obligation to the invariant and the *mergeData* operation of the *CreateComputer* abstract machine, the proof obligation in the B prover will be written as follows:

$$(xx \in \{hostName,\ ipAdress,\ userName\} \wedge yy \in \{hostName,\ ipAdress,\ userName\} \wedge xx = yy \Rightarrow (\{P1 \mapsto hostName,\ P2 \mapsto ipAdress,\ P3 \mapsto userName\}^{-1}(xx) = \{P1 \mapsto hostName,\ P2 \mapsto ipAdress,\ P3 \mapsto userName\}^{-1}(yy)\ ))$$

The result of this predicate is $WFR1 = true$.

In a similar way, we transform others well-formedness rules of the Core package into B to verify the correctness of UML model elements of class diagrams.

6 Transformation of the metamodel of UML collaboration diagrams into B

The metamodel of UML collaboration diagrams called the Collaboration package is a sub-package of the Behavioural Elements package. It specifies the concepts needed to

express how different elements of a model interact with each other from a structural view. This package uses constructs defined in the Foundation package of UML as well as in the Common Behaviour package.

6.1 Transformation of the UML metamodel

Based on the procedure of transformation presented in Section 4, we transform the metamodel of UML collaboration diagrams into B to verify its elements. Attributes of each object in the Collaboration package are transformed as variables of the abstract machine with their type determined as follows:

```
MACHINE Interaction
SEES Types
USES Message1, Message11, Message12, Message13
VARIABLES
interaction_name, interaction_context,
message_name, message_interaction, message_sender,
message_activator, message_predecessor, ...
INVARIANT
interaction_name ∈ INTERACTION ⇸ INTERACTION_NAME ∧
interaction_context ∈ INTERACTION ⇸ COLLABORATION ∧

message_name ∈ MESSAGE ⇸ MESSAGE_NAME ∧
message_interaction ∈ MESSAGE ⇸ INTERACTION ∧
message_sender ∈ MESSAGE ⇸ CLASSIFIER_ROLE ∧
message_activator ∈ MESSAGE ⇸ MESSAGES ∧
message_predecessor ∈ MESSAGE ↔ MESSAGE ∧ ...
INITIALISATION
interaction_name := { inte1 ↦ interaction1 } ||
interaction_context := { inte1 ↦ coll1 } ||

message_name := ∅ || message_interaction := ∅ || message_sender := ∅ ||
message_activator := ∅ || message_predecessor := ∅...
OPERATIONS
  mergeData =
  pre
     message1_predecessor = ∅ ∧ message11_predecessor = ∅ ∧
      message12_predecessor = {mess12 ↦ mess11} ∧
     message13_predecessor = {mess13 ↦ mess11, mess13 ↦ mess12} ∧
     ...
  then
     message_predecessor := message1_predecessor ∪
     message11_predecessor ∪ message12_predecessor ∪ message13_predecessor ||
     ...
  end
END
```

Figure 7: **Interaction** B abstract machine

- The type of variables transformed from attributes of objects which contain only one value is defined as a partial function from the set of object identifiers to the type of their values: $attr_i \in CLASS \nrightarrow TYPE(attr_i)$.

- The type of variables transformed from attributes of objects which possibly contain a set of elements is defined as a relation from the set of object identifiers to the type of their values: $attr_i \in CLASS \leftrightarrow TYPE(attr_i)$.

To illustrate the transformation of the metamodel of the UML collaboration diagram of the printing system (presented Figure 2) to a B specification, 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, Message13. The Interaction meta-class of this case study have only one object. This object is transformed into a B abstract machine presented in Figure 7. The *Interaction* abstract machine does not only contain variables transformed from the attributes of the Interaction object (prefixed with *interaction*), it also contains variables to merge the data of the variables in the abstract machines of objects of the Message class (prefixed with *message*). The merging of variables is realised by the operation *mergeData* of the *Interaction* machine. The purpose of this merging, as presented above, is to verify UML model elements of the collaboration diagram.
The machines of component objects in the UML metamodel 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 metamodel of a UML collaboration diagram is presented Figure 8.

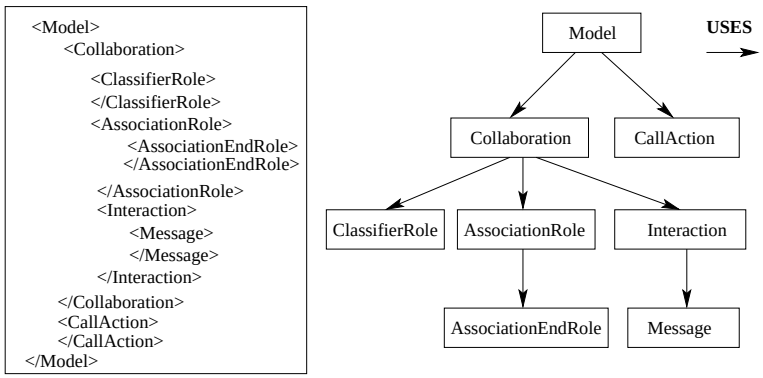


Figure 8: General structure of the UML metamodel of collaboration diagrams and their transformation to B

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.
All the machines in the system see (SEES) the Types machine which defines all sets of the system. In our case study, these sets are:

$MESSAGE = \{mess1, mess11, mess12, mess13\}$;
$MESSAGE_NAME = \{print, restore, notifyStatus\}$;
$CLASSIFIER_ROLE = \{class1, class2, class3, class4\}$;
$CLASSIFIER_ROLE_NAME = \{Computer, PrintServer, Queue, Printer\}$;
...

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 -> forAll(p | p.interaction = self.interaction)
      and
      self.activator -> forAll( a | a.interaction = self.interaction)
```

This OCL predicate can be transformed to the B invariant as presented Figure 9.

INVARIANT
...
$\forall\ pp.(pp \in MESSAGE \wedge message_predecessor[\{pp\}] \neq \varnothing$
$\Rightarrow message_interaction[message_predecessor[\{pp\}]] =$
$message_interaction[\{pp\}]\ ) \wedge$
$\forall\ aa.(aa \in MESSAGE \wedge message_activator[\{aa\}] \neq \varnothing$
$\Rightarrow message_interaction[message_interaction[\{aa\}]] =$
$message_interaction[\{aa\}]$

Figure 9: The B invariant transformed from the $WFR2$ rule

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 $mergeData$ 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's analyse the proof obligation $(I \wedge P \Rightarrow [S]I)$ of the B abstract machine for verifying the preservation of the invariant I above, where P is the precondition of the $mergeData$ operation and S is its body.

$WFR2a = \forall\, pp.(pp \in MESSAGE \wedge message_predecessor[\{pp\}] \neq \varnothing \Rightarrow$
$message_interaction[message_predecessor[\{pp\}]] = message_interaction[\{pp\}])$

Let's examine each value of pp:

if $pp = mess1 \Rightarrow message_predecessor[\{mess1\}] = \varnothing$;
if $pp = mess11 \Rightarrow message_predecessor[\{mess11\}] = \varnothing$;
In these two cases, the precondition of $WFR2a$ is not satisfied.

if $pp = mess12 \Rightarrow message_predecessor[\{mess12\}] = \{mess11\}$
$\Rightarrow message_interaction[\{mess11\}] = \{inte1\}$
So $message_interaction[message_predecessor[\{mess1\}]] = \{intel1\}$

On the other hand, $message_interaction[\{mess12\}] = \{inte1\}$
$\Rightarrow WFR2a = true$

if $pp = mess13 \Rightarrow message_predecessor[\{mess13\}] = \{mess11, mess12\}$
Note that: $ran(u \lhd r) = r[u]$ with $u \subseteq s \wedge r \in s \leftrightarrow t$ (See *The B-Book* [1], p.102)

$\Rightarrow message_interaction[\{mess11, mess12\}]$
$= ran(\{mess11, mess12\} \lhd message_interaction)$
$= ran(\{mess11 \mapsto inte1, mess12 \mapsto inte1\}) = \{inte1\}$
and $message_interaction[\{mess13\}] = \{inte1\} \quad \Rightarrow WFR2a = true$.
We deduce that $WFR2a = true$ for each value of pp.

$WFR2b = \forall\, aa.(aa \in MESSAGE \wedge message_activator[\{aa\}] \neq \varnothing$
$\Rightarrow message_interaction[message_activator[\{aa\}]] = message_interaction[\{aa\}])$

if $aa = mess1 \Rightarrow message_activator[\{mess1\}] = \varnothing$
if $aa = mess11$ or $aa = mess12$ or $aa = mess13$
$\Rightarrow message_activator[\{aa\}] = \{mess1\}$
$\Rightarrow message_interaction[\{mess1\}] = \{inte1\}$
and $message_interaction[\{aa\}] = \{inte1\} \quad \Rightarrow WFR2b = true$

As a consequence, we have: $WFR2 = WFR2a \wedge WFR2b = true$.

```
Rule WFR3.  The predecessors must have the same activator
            as the Message:
   self.allPredecessors -> forAll( p | p.activator = self.activator)
```

This OCL predicate can be transformed to the B invariant as presented Figure 10.

INVARIANT
...
$\forall\, xx.(xx \in MESSAGE \wedge message_predecessor[\{xx\}] \neq \varnothing \Rightarrow$
$message_activator[message_predecessor[\{xx\}]] = message_activator[\{xx\}])$

Figure 10: The B invariant transformed from the $WFR3$ rule

if $xx = mess1 \Rightarrow message_predecessor[\{mess1\}] = \varnothing$;
if $xx = mess11 \Rightarrow message_predecessor[\{mess11\}] = \varnothing$;
if $xx = mess12 \Rightarrow message_predecessor[\{mess12\}] = \{mess11\}$
$\Rightarrow message_activator[\{mess11\}] = \{mess1\}$

and $message_activator[\{mess12\}] = \{mess1\}$
$\Rightarrow WFR3 = true$

if $xx = mess13 \Rightarrow message_predecessor[\{mess13\}] = \{mess11, mess12\}$
$\Rightarrow message_activator[\{mess11, mess12\}]$
$= ran(\{mess11, mess12\} \lhd message_activator)$
$= ran(\{mess11 \mapsto mess1, mess12 \mapsto mess1\}) = \{mess1\}$
and $message_activator[\{mess13\}] = \{mess1\}$
$\Rightarrow WFR3 = true.$
The result of this invariant is $WFR3 = true$

```
Rule WFR4.  A Message cannot be the predecessor of itself.
            not self.allPredecessor -> includes(self)
```

This OCL predicate can be transformed to the B invariant as presented Figure 11.

INVARIANT
...
$\forall xx.(xx \in MESSAGE \Rightarrow xx \notin message_predecessor[\{xx\}])$

Figure 11: The B invariant transformed from the $WFR4$ rule

if $xx = mess1 \Rightarrow message_predecessor[\{mess1\}] = \varnothing;$
if $xx = mess11 \Rightarrow message_predecessor[\{mess11\}] = \varnothing;$
if $xx = mess12 \Rightarrow message_predecessor[\{mess12\}]$
$= ran\{mess12 \mapsto mess11\}$
$= \{mess11\}, \ (mess12 \notin \{mess11\});$
if $xx = mess13 \Rightarrow message_predecessor[\{mess13\}]$
$= ran\{mess13 \mapsto mess11, mess13 \mapsto mess12\}$
$= \{mess11, mess12\}, \ (mess13 \notin \{mess11, mess12\});$
As a result, $WFR4 = true.$

The verification of the well-formedness rules can be executed by the support tool AtelierB [17], which can both automatically and interactively demonstrate theorems. When using the AtelierB tool to prove the *Interaction* abstract machine of the printing system, 15 proof obligations are proved automatically and 3 proof obligations are proved interactively.

7 Transformation of the metamodel of UML statechart diagrams into B

The metamodel of statechart diagrams called the State Machine package, it is a subpackage of the Behavioural Elements package. It specifies a set of concepts that can be used for modelling behaviour through finite state-transition systems. It is defined as an elaboration of the Foundation 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 Figure 12.

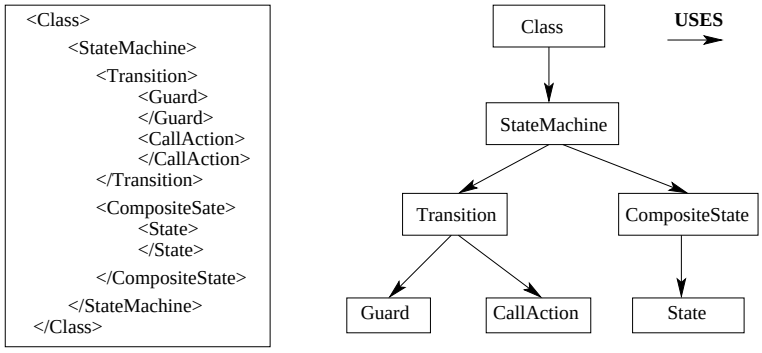


Figure 12: General structure of the UML metamodel of statechart diagrams and their transformation to B

The machines of objects of the Class class use the machines of objects of the StateMachine class; the machines of objects of the StateMachine class use machines of objects of the Transition class and machines of objects of the CompositeState class and so on.
Because of the similarity in the verification with well-formedness rules, we point out only the transformation of UML metamodel of state-chart diagrams and do not illustrate its verification in the case study.

8 Conclusion

We have presented a technique to transform the metamodel of UML class diagrams, collaboration diagrams, statechart 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-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 in the B provers is automatically and easily performed.
A prototype ArgoUML+B [12] has been developed from ArgoUML [2], a free available platform for editing UML diagrams. This prototype automatically transforms UML diagrams (class, state-chart, 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.
Inspired from this approach, we plan to build a formal approach to specify and verify object-based systems using the B method.

[2] http://argouml.tigris.org

References

[1] J. R. Abrial. *The B-Book, Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] Abstract State Machine. Available at *http://www.eecs.umich.edu/gasm.*

[3] B-Core(UK) Ltd. *B-Toolkit User's Manual.* Oxford (UK), 1996. Release 3.2.

[4] G. Booch, J. Rumbaugh, and I. Jacopson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.

[5] 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*, pages 1519–1523. ACM press, 2004.

[6] Clearsy. *B4free.* Available at *http://www.b4free.com.*

[7] P. Facon, R. Laleau, and H.P. Nguyen. Mapping Object Diagram into B. In *Methods Integration Workshop*, Leeds, March 25-26 1996.

[8] H. LeDang and J. Souquières. Contributions for Modelling UML State-Charts in B. In *Third International Conference on Integrated Formal Methods*, LNCS. Springer Verlag, May 2002.

[9] M.G. Hinchey and J. P. Bowen. *Applications of Formal Methods.* Prentice Hall, 1995.

[10] S.K Kim and D. Carrington. A Formal Mapping between UML Models and Object-Z Specifications. In *ZB 2000: Formal Specification and Development in Z and B*, volume 1878 of *LNCS*, pages 2–21. Springer Verlag, 2000.

[11] R. Laleau and F. Polack. Metamodels for Static Conceptual Modelling of Information System. In *Workshop on Defining Precise Semantics of UML*, Sophia Antipolis, France, ECOOP 2000.

[12] H. Ledang, J. Souquières, and S. Charles. ArgoUML+B: Un outil de transformation systématique de spécifications UML vers B. In *Approaches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2003)*, 2003.

[13] E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In *Proceedings of the Formal Method Conference*, number 1708 in LNCS, pages 875–895. Spring Verlag, 1999.

[14] OMG. *Unified Modeling Language.* OMG *http://www.omg.org/docs/formal/03−03−01.pdf*, Version 1.5 March 2003.

[15] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints.* PhD thesis, Bremen University, 2002.

[16] S. Schneider. *The B Method: An Introduction.* PALGRAVE, ISBN 0-333-79284-X, 2001.

[17] Steria. *Obligations de preuve: Manuel de référence.* Steria - Technologies de l'information, version 3.0. Tool is available at *http://www.atelierb.societe.com.*

[18] N.T. Truong and J. Souquières. An approach for the verification of UML models using B. In *11th International Conference of Engineering of Computer Based Systems (ECBS)*, pages 195–202. IEEE Computer Society press, 2004.

[19] N.T. Truong and J. Souquières. Verification of behavioral elements of UML models using B. In *20th Annual ACM Symposium on Applied Computing (SAC'05)*, pages 1546–1552. ACM press, 2005.

[20] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, ISBN 0-201-37940-6, 1999.