View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

Derivation Schemes from OCL Expressions to B

Hung Ledang, Jeanine Souquières

To cite this version:

Hung Ledang, Jeanine Souquières. Derivation Schemes from OCL Expressions to B. [Intern report] A02-R-042 || ledang02b, 2002, 10 p. inria-00099424

HAL Id: inria-00099424
https://hal.inria.fr/inria-00099424

Submitted on 26 Sep 2006

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.

Derivation Schemes from OCL Expressions to B

Hung LEDANG Jeanine SOUQUIÈRES

LORIA - Université Nancy 2 - UMR 7503
Campus scientifique, BP 239
54506 Vandœuvre-les-Nancy Cedex - France
E-mail: {ledang,souquier}@loria.fr

Abstract

In the continuity of our research on integration of UML and B, we address in this paper the transformation from OCL (Object Constraint Language), which is part and parcel of UML, into B. Our derivation schemes allow to automatically derive in B not only the complementary class invariants, the guard conditions in state-charts (in OCL) but also OCL specifications OCL class operations, events or use cases.

Keywords: *UML, OCL, OCL expression, B expression, B substitution.*

1 Introduction

The Unified Modelling Language (UML)[17] has become a de-facto standard notation for describing analysis and design models of object-oriented software systems. The graphical description of models is easily accessible. Developers and their customers intuitively grasp the general structure of a model and thus have a good basis for discussing system requirements and their possible implementation. However, since the UML concepts have English-based informal semantics, it is difficult even impossible to design tools for verifying or analysing formally UML specifications. This point is considered as a serious drawback of UML-based techniques.

To remedy such a drawback, one approach is to develop UML as a precise (i.e well defined) modelling language. The pUML (precise UML) group has been created to achieve this goal. However the main challenge [4] of pUML is to define a new formal notation that has been up to now an open issue. Furthermore, the support tool for such a new formalism is perhaps another challenge.

In waiting for a precise version of UML and its support tool, the necessity to analyse inconsistencies within UML specifications should be solved in a pragmatic approach (cf. [3]) : formalising UML specifications by existing formal languages and then analysing UML specifications via the derived formal specifications. In this perspective, using the B language [1] to formalise UML specifications has been considered as a promising approach [13, 10]. By formalising UML specifications in B, one can use B powerful support tools like AtelierB [18], B-Toolkit [2] to analyse and detect inconsistencies within UML specifications [9]. On the other hand, we can also use UML specifications as the starting point to develop B specifications which can be then refined automatically to an executable code [6].

Meyer and Souquières [14] and Nguyen [15], based on the previous work of Lano [7], have proposed the derivation schemes from UML structural concepts into B. Each class, attribute, association and state is modelled as a B variable. The properties of those concepts are modelled as B invariants. The inheritance relationship between classes is also modelled as B invariant between B variables for the classes in question.

In [8, 11, 12] we have proposed approaches for modelling UML behavioural concepts. Each UML behavioral concept - *use case*, *class operation*, *event* - is firstly modelled by a B abstract operation in which the expected effects of such a concept on related data is specified directly on the derived data. The B operation for use cases, class operations and events may be refined afterward.

The UML-B derivation schemes for UML structural and behavioural concepts are used in three derivation procedures based on use cases [8], events [12] and class operation [11], which allow to integrate several kind of UML diagrams into the same B specification. At this stage, only the architecture, data and the operations' signature of the derived B specification are generated automatically. For the invariant within B specification, only the part that reflects the properties of UML structural concepts expressed graphically in the UML diagrams is generated. Therefore, the specification B should be completed with invariants for supplementary class invariant, supplementary attribute properties as well as B operations' body.

As cited in the UML literature [16], OCL (Object Constraint Language) is often used to specify supplementary class invariant, supplementary attribute properties as well as pre- and post-conditions of behavioural concepts within UML specifications. In the continuity of our research on integration UML and B, we address in this paper the transformation from OCL expressions into B. This OCL-B translation is applied for generating supplementary invariant and the abstract operations' body of the derived B specification.

In Section 2 we outline what does look like the transformation from OCL expressions into B. The derivation schemes for OCL types and their operations are presented Section 3. The derivation schemes specific for postconditons are presented in Section 4. Discussions in Section 5 conclude our presentation.

2 From OCL expressions into B : an overview

2.1 The OCL language

The Object Constraint Language (OCL) is now part and parcel of the UML standard [16]. One can use OCL to write constraints that contain extra information about, or restrictions to, UML diagrams. OCL is intended to be simple to read and write. Its syntax is similar to object-oriented programming languages. Most OCL expressions can be read left-to-right where the left part usually represents - in object-oriented terminology - the receiver of a message. Frequently used language features are attribute access of objects, navigation to objects that are connected via association links, and operation calls. OCL expressions are not only used to define invariants on classes and types, they also allow specification of guard conditions in UML state-charts and pre- and postconditions on class operations, use cases or events. Figure 1 shows the OCL specification of the class operation Pump::enable_Pump according to its informal specification in [5].

CONTEXT *Pump::enable_Pump*(*pi* : *PUMPID*, *gg* : *GRADE*, *vi* : *VEH_ID*) : **void**
PRE
Pump.allInstances→*collect*(*pump_Id*)→*includes*(*pi*)
POST
let *pp* : *Set*(*Pump*) = *Pump.allInstances* → *select*(*pump_Id@pre=pi* **and** *status@pre=disabled*)
in
if *pp* → *notEmpty* **then**
pp → $forall$(*p* | *p.status* = *enabled*) **and**
pp → $forall$(*p* | *p.display.grade* = *gg*) **and**
pp → $forall$(*p* | *p.display.cost* = *costOfGrade*(*gg*)) **and**
pp → $forall$(*p* | *p.display.volume* = *0*) **and**
pp → $forall$(*p* | *p.display.veh_Id* = *vi*) **and**
pp → $forall$(*p* | *p.motor.status* = *on*) **and**
pp → $forall$(*p* | *p.clutch.status* = *freed*)
else $true$ **endif**

Figure 1. The operation enable_Pump in OCL

2.2 The B language and method

B [1] is a formal software development method that covers the software process from specifications to implementations. The B notation is based on set theory, the language of generalised substitutions and first order logic. Specifications are composed of abstract machines similar to modules or classes; they consist of variables, invariance properties relating to those variables and operations. The state of the system, i.e. the set of variable values, is only modifiable by operations. The abstract machine can be composed in various ways. Thus, large systems can be specified in a modular way, possibly reusing parts of other specifications. B refinement can be seen as an implementation technique but also as a specification technique to progressively augment a specification with more details until an implementation that can then be translated into a programming language like ADA, C or C++. At every stage of the specification, proof obligations ensure that operations preserve the system invariant. A set of proof obligations that is sufficient for correctness must be discharged when a refinement is postulated between two B components.

2.3 Principles to translate OCL expressions to B

The core of OCL is given by an expression language. OCL expressions can be used in various contexts, for example, to define constraints such as class invariants and pre- and postconditions on behavioural concepts. Our derivation schemes from OCL to B are therefore defined for concepts related to OCL expressions : (i) the OCL types and the associated operations and (ii) the postconditions on behavioural concepts.

It is natural to model an OCL type by a B type, which would be a B predefined type such as $\mathcal{Z}$, $BOOL$ etc, or a B user-defined type such as sets or relations. In addition, the formalisation in B of OCL types is guided and motivated by the wish to facilitate the formalisation in B of operations on OCL types. Intuitively, an OCL expression for class invariants, for guard conditions or for preconditions on behavioural concepts should be modelled by a B expression; meaning that every OCL operation (except oclIsNew, which is used in postconditions on behavioural concepts) should be represented by a B expression.

The derivation schemes from OCL to B for the types and the associated operations are sufficient to derive a B expression from an OCL expression of class invariants on class diagrams, guard conditions on state-charts or preconditions on behavioural concepts. To model postconditions of behavioural concepts, the use of B generalised substitutions is necessary. The OCL expressions involving values after executing the behavioural concepts are translated into B substitutions.

3 Derivation schemes for OCL types and their operations

3.1 Types OCL

The types in OCL can be classified as follows. The group of predefined basic types includes Integer, Real, Boolean and String. Enumeration types are user-defined. An object type corresponds to a classifier in an object model.

Collections of values can be described by the collection types Set(T), Sequence(T) and Bag(T). These are the classical types for bulk data, namely sets, lists and muli-sets respectively. The parameter T denotes the type of the elements. Notice that types at the meta-level such as OclExpression are not considered in the translation from OCL expressions into B.

3.2 Predefined basic types

Derivation 1 (Integer) In B there are two predefined types corresponding to the OCL type Integer : $\mathcal{Z}$ and INT. $\mathcal{Z}$ is chosen as the formalisation of Integer since $\mathcal{Z}$ is more abstract than INT. The OCL operations defined on Integer can be mapped to operations defined on $\mathcal{Z}$ as shown in Table 1, where a, b are two integers and a, b denote their B formalisation.

Operations OCL	Semantics in B
a:Integer	$a \in \mathcal{Z}$
a=b	$a = b$
a<>b	$\neg(a = b)$
a+b	$a+b$
a-b	$a-b$
-a	$-a$
a*b	$a \times b$
a div b	a/b
a mod b	$a\ mod\ b$
a<b	$a < b$
a<=b	$a \leq b$
a>b	$b < a$
a>=b	$b \leq a$
a.min(b)	$min(\{a, b\})$
a.max(b)	$max(\{a, b\})$
a.abs	$map(-a, a)$
a/b	$a \mapsto b$

Table 1. Modelling Integer OCL operations in B

Remark 1 (The operation "/")

1. In OCL, the operation a/b, where a, b are two integers, gives as result a real value. Since B does not define the data type for real values, we propose to model a/b by a pair $a \mapsto b$, where a and b denote respectively the B formalisation of a and b.

2. The fact of using a rate to express the division between two integers implies to define the formalisation in B for operations between an integer and a rate.

Derivation 2 (Boolean) The OCL type Boolean is modelled in B by its correspondence $BOOL$. The Boolean OCL operations are modelled in B by expressions on $BOOL$ as shown in Table 2, where a, b are two booleans and a, b denote their B formalisation.

Operations OCL	Semantics in B
a:Boolean	$a \in BOOL$
a=b	$a = b$
a<>b	$\neg(a = b)$
a or b	$a \vee b$
a xor b	$\neg(a = b)$
a and b	$a \wedge b$
not a	$\neg a$
a implies b	$\neg a \vee b$
if a then b else c endif	$\neg((a \wedge b) = (\neg a \wedge c))$

Table 2. Modelling Boolean OCL operations in B

Derivation 3 (String) The B predefined type $STRING$ cannot be used to model the OCL type String due to restrictions of operations on $STRING$ (only "=" and "<>" are defined for $STRING$). We propose therefore to model String by $seq(0..255)$. Hence we can use B expressions on sequences to define String OCL operations (except two operations toUpper and toLower as shown in Table 3).

Remark 2 Two operations toUpper and toLower involve a repetitive computation which is very sophisticated such that they cannot be expressed by an expression B at the level of an abstract machine.

Derivation 4 (Real) There is no B predefined type for real values, however Remark 1 suggests us a solution to approximate a real value by a rate. Hence the type Real can be modelled in B by relation $\mathcal{Z} \leftrightarrow \mathcal{Z}$. It remains to define the conversion from a real value to its corresponding rate as well as the formalisation of Real OCL operations using B expressions on $\mathcal{Z} \leftrightarrow \mathcal{Z}$, which need some further investigation and therefore is beyond the scope of the current paper.

Operations OCL	Semantics in B
a:String	$a \in seq(0..255)$
a=b	$a = b$
a<>b	$\neg(a = b)$
a.size	$size(a)$
a.concat(b)	$a \frown b$
a.subString(lower,upper)	$(a \uparrow upper) \downarrow lower$
a.toUpper	*no definition*
a.toLower	*no definition*

Table 3. Modelling String OCL operations in B

Derivation 5 (Enumeration types) Each enumeration type Enum={val1,...,valn} is modelled in B by a enumerated set serving as a user-defined type $Enum = \{val1, ..., valn\}$. Each element vali# in $Enum$ is modelled by an element $vali$ in $Enum$. The modelling in B of operations on an enumeration type is shown in Table 4, where a#, b# are two values of type Enum and a, b denote their B formalisation.

Operations OCL	Semantics in B
a#:Enum	$a \in Enum$
a#=b#	$a = b$
a#<>b#	$\neg(a = b)$

Table 4. Modelling Enumeration OCL operations in B

Derivation 6 (Object types) According to Meyer and Souquières [14], for each class class, the B constant $CLASS$ models the possible instance set and the B variable $class$ model the effective instance set of class. Therefore, the object type class is modelled in B as $CLASS$, whereas the operation class.allInstances is modelled as $class$.

Derivation 7 (Collection types) Given T an OCL type. Let's call T the B formalisation of T, the B formalisation of collection types on T is as follows :

- Set(T), which denotes all subsets of T, is modelled in B by $\mathcal{P}(T)$,
- Bag(T), which denotes all multi-sets on T, is modelled as $T \nrightarrow \mathcal{N}$. An element bag of Bag(T) is therefore modelled as $bag \in T \nrightarrow \mathcal{N}$ and for each element tt : T of bag, $bag(tt)$ denotes the occurrence number of tt in bag,
- Sequence(T) is directly modelled by $Seq(T)$.

The formalisation of OCL operations on collection types is shown in Table 5, Table 6 and Table 7, where :

- T is an OCL type on which the collection types are defined, and tt : T ; ss, ss2 : Set(T) ; bb, bb2 : Bag(T) ; se, se2 : Sequence(T) ;
- ss, $ss2$, bb, $bb2$, se, $se2$ and tt are respectively the B formalisation of ss, ss2, bb, bb2, se, se2 and tt ;
- for the operation sum, T must be of type Integer.

Remark 3 (Operations on collection types) The semantics of the operation asSequence on a set or a bag has not been defined in OCL therefore we cannot model it in B. It is the same for the operation excluding on a sequence.

Operations OCL	Semantics in B
ss:Set(T)	$ss \subseteq T$
ss=ss2	$ss = ss2$
ss<>ss2	$\neg(ss = ss2)$
ss->union(ss2)	$ss \cup ss2$
ss->union(bb)	*cf.* ss->asBag->union(bb)
ss->intersection(ss2)	$ss \cap ss2$
ss->intersection(bb)	$ss \cap dom(bb)$
ss-ss2	$ss - ss2$
ss->symmetricDifference(ss2)	$(ss \cup ss2) - (ss \cap ss2)$
ss->including(tt)	$ss \cup \{tt\}$
ss->excluding(tt)	$ss - \{tt\}$
ss->asBag	$ss \times \{1\}$
ss->asSequence	*no definition*
ss->size	$card(ss)$
ss->count(tt)	$card(ss \cap \{tt\})$
ss->includes(tt)	$tt \in ss$
ss->includesAll(ss2)	$ss2 \subseteq ss$
ss->includesAll(bb)	$dom(bb) \subseteq ss$
ss->includesAll(se)	$ran(se) \subseteq ss$
ss->excludes(tt)	$\neg(tt \in ss)$
ss->excludesAll(ss2)	$ss \cap ss2 = \phi$
ss->excludesAll(bb)	$ss \cap dom(bb) = \phi$
ss->excludesAll(se)	$ss \cap ran(se) = \phi$
ss->isEmpty	$ss = \phi$
ss->notEmpty	$\neg(ss = \phi)$
ss->sum	$\Sigma(xx).(xx \in ss \mid xx)$

Table 5. Modelling Set(T) OCL operations in B

3.3 Operations select, reject, collect, forAll, exists

Derivation 8 (select, reject, collect, forAll, exists) Given an OCL type T, let's call T the B formalisation of T. The B formalisation of OCL operations select, collect, forAll, exists on collection types on T is shown in Table 8, where :

- ss : Set(T), bb : Bag(T), se : Sequence(T), tt :T ;
- boolexprtt is a boolean expression on tt and exprtt is an expression on tt ;

Operations OCL	Semantics in B
bb:Bag(T)	$bb \in T \nrightarrow \mathcal{N}$
bb=bb2	$bb = bb2$
bb<>bb2	$\neg(bb = bb2)$
bb->union(bb2)	$\{vv, nn \mid vv \in T \wedge vv \in dom(bb) \cup dom(bb2) \wedge nn \in \mathcal{N} \wedge nn = max(bb[\{vv\}] \cup bb2[\{vv\}])\}$
bb->union(ss)	*cf.* bb->union(ss->asBag)
bb->intersection(bb2)	$\{vv, nn \mid vv \in T \wedge vv \in dom(bb) \cap dom(bb2) \wedge nn \in \mathcal{N} \wedge nn = min(bb[\{vv\}] \cup bb2[\{vv\}])\}$
bb->intersection(ss)	$dom(bb) \cap ss$
bb->including(tt)	$bb \lhd\!\!- \{tt \mapsto (\Sigma(xx).(xx \in bb[\{tt\}] \cup \{0\} \mid xx) + 1)\}$
bb->excluding(tt)	$bb \lhd\!\!- (\{tt \mapsto max(\Sigma(xx).(xx \in bb[\{tt\}] \cup \{0\} \mid xx) - 1, 0)\}) \rhd \{0\}$
bb->asSequence	*no definition*
bb->asSet	$dom(bb)$
bb->size	$\Sigma(vv).(vv \in dom(bb) \mid bb(vv))$
bb->count(tt)	$\Sigma(xx).(xx \in bb[\{tt\}] \cup \{0\} \mid xx)$
bb->includes(tt)	$tt \in dom(bb)$
bb->includesAll(bb2)	$dom(bb2) \subseteq dom(bb)$
bb->includesAll(ss)	$ss \subseteq dom(bb)$
bb->includesAll(se)	$ran(se) \subseteq dom(bb)$
bb->excludes(tt)	$\neg(tt \in dom(bb))$
bb->excludesAll(bb2)	$dom(bb) \cap dom(bb2) = \phi$
bb->excludesAll(ss)	$dom(bb) \cap ss = \phi$
bb->excludesAll(se)	$dom(bb) \cap ran(se) = \phi$
bb->isEmpty	$bb = \phi$
bb->notEmpty	$\neg(bb = \phi)$
bb->sum	$\Sigma(xx).(xx \in dom(bb) \mid xx \times bb(xx))$

Table 6. Modelling Bag(T) OCL operations in B

Operations OCL	Semantics in B
se:Sequence(T)	$se \in seq(T)$
se=se2	$se = se2$
se<>se2	$\neg(se = se2)$
se->union(se2)	$se \frown se2$
se->append(tt)	$se \leftarrow tt$
se->prepend(tt)	$tt \rightarrow se$
se->subSequence(i,j)	$(se \uparrow j) \downarrow i$
se->at(i)	$se(i)$
se->first	$first(se)$
se->last	$last(se)$
se->asSet	$ran(se)$
se->asBag	$\{vv, nn \mid vv \in ran(se) \wedge nn \in \mathcal{N} \wedge nn = card(se^{-1}[\{vv\}])\}$
se->including(tt)	$se \leftarrow tt$
se->excluding(tt)	*no definition*
se->size	$size(se)$
se->count(tt)	$card(se^{-1}[\{tt\}])$
se->includes(tt)	$tt \in ran(se)$
se->includesAll(se2)	$ran(se2) \subseteq ran(se)$
se->includesAll(ss)	$ss \subseteq ran(se)$
se->includesAll(bb)	$dom(bb) \subseteq ran(se)$
se->excludes(tt)	$not(tt \in ran(se))$
se->excludesAll(se2)	$ran(se2) \cap ran(se) = \phi$
se->excludesAll(ss)	$ss \cap ran(se) = \phi$
se->excludesAll(bb)	$dom(bb) \cap ran(se) = \phi$
se->isEmpty	$se = \phi$
se->notEmpty	$\neg(se = \phi)$
se->sum	$\Sigma(xx).(xx \in dom(se) \mid se(xx))$

Table 7. Modelling Sequence(T) OCL operations in B

- ss, bb, se, tt, $boolexprtt$ et $exprtt$ are respectively the B formalisation of ss, bb, se, tt, boolexprtt and exprtt.

3.4 Attribute and navigation operations

An attribute or navigation operation on an object might return as a single value/object, a set of values/objects, a multi-set of values/objects or a sequence of values/objects. It is also possible to apply an attribute or a navigation operation on the result of another attribute or a navigation operation. Hence the target of an attribute or navigation operation can be an object, a set of objects, a multi-set of objects or even a sequence of objects. Our derivation schemes for attribute and navigation operations are based on the derivation schemes for UML structural concepts (cf. Derivation 9).

Derivation 9 (Structural concepts (extracted from [14]))

- An attribute attr of type typeAttr in a class Class is modelled by a B variable $attr$ defined as : $attr \in class \leftrightarrow typeAttr$, where the variable $class$ models the effective instance set of Class and $typeAttr$ is defined as a B set to model typeAttr. The relation defining $attr$ might be further refined according to additional properties of attr.
- A binary association assos between two classes Class and Class2 is modelled by a B variable $assos$ defined as $assos \in class \leftrightarrow class2$. If there are eventual qualifiers q1 : Q1, ..., qn : Qn at the role end of Class, they are modelled in a similar manner to an attribute : $q1 \in class2 \leftrightarrow Q1, \ldots, q1 \in class2 \leftrightarrow Q1$. We add also a B invariant linking $assos$ and $q1$,...,qn as follows : $(assos^{-1} \otimes q1 \otimes \ldots \otimes qn)^{-1} \in class \times Q1 \times \ldots \times Qn \leftrightarrow class2$. As for attributes, the relation defining $assos$ could be further refined according to additional properties of assos.

Derivation 10 (Attribute operations) Given attr, cc, sc, bc, seqc an attribute, an object, a set of objects, a bag of objects and a sequence of objects of a class Class. Let's call $attr$, cc, sc, bc and $seqc$ their B formalisation according to Derivation 6, Derivation 7 and Derivation 9 :

1. the expression cc.attr, which denotes the value(s) of the

<table>
<tr><th>Operations OCL</th><th>Semantics in B</th></tr>
<tr><td>ss->select(tt|boolexprtt)</td><td>$\{tt|tt{\in}ss{\wedge}boolexprtt\}$</td></tr>
<tr><td>bb->select(tt|boolexprtt)</td><td>$\{tt, nn|tt{\in}dom(bb){\wedge}boolexprtt{\wedge}$ $nn{\in}\mathcal{N}{\wedge}nn = bb(tt)\}$</td></tr>
<tr><td>ss->reject(tt|boolexprtt)</td><td>$\{tt|tt{\in}ss{\wedge}\neg boolexprtt\}$</td></tr>
<tr><td>bb->reject(tt|boolexprtt)</td><td>$\{tt, nn|tt{\in}dom(bb){\wedge}\neg boolexprtt$ ${\wedge}nn{\in}\mathcal{N}{\wedge}nn = bb(tt)\}$</td></tr>
<tr><td>ss->collect(tt|boolexprtt)</td><td>$\{tt, nn|tt{\in}expr\_tt[ss]{\wedge}nn{\in}\mathcal{N}{\wedge}$ $nn = card(\{xx|xx{\in}ss{\wedge}$ $exprtt(xx) = tt\})\}$</td></tr>
<tr><td>bb->collect(tt|exprtt)</td><td>$\{tt, nn|tt{\in}exprtt[dom(bb)]{\wedge}nn{\in}\mathcal{N}$ ${\wedge}nn = \Sigma(xx).(xx{\in}dom(bb){\wedge}$ $exprtt(xx) = tt|bb(xx))\}$</td></tr>
<tr><td>se->collect(tt|exprtt)</td><td>$\lambda(ii).(ii{\in}dom(se)|exprtt(se(ii))$</td></tr>
<tr><td>ss->forAll(tt|boolexprtt)</td><td>$\forall(tt).(tt{\in}ss{\Rightarrow}boolexprtt)$</td></tr>
<tr><td>bb->forAll(tt|boolexprtt)</td><td>$\forall(tt).(tt{\in}dom(bb){\Rightarrow}boolexprtt)$</td></tr>
<tr><td>se->forAll(tt|boolexprtt)</td><td>$\forall(tt).(tt{\in}ran(se){\Rightarrow}boolexprtt)$</td></tr>
<tr><td>ss->exists(tt|boolexprtt)</td><td>$\exists(tt).(tt{\in}ss{\wedge}boolexprtt)$</td></tr>
<tr><td>bb->exists(tt|boolexprtt)</td><td>$\exists(tt).(tt{\in}dom(bb){\wedge}boolexprtt)$</td></tr>
<tr><td>se->exists(tt|boolexprtt)</td><td>$\exists(tt).(tt{\in}ran(se){\wedge}boolexprtt)$</td></tr>
</table>

Table 8. Modelling in B of operations select, reject, collect, forAll and exists

attribute attr associated to the object cc, is generally modelled in B by $attr[\{cc\}]$. If the cardinality of attr is equal to 1, cc.attr can be modelled by $attr(cc)$; otherwise and if attr is ordered, $attr[\{cc\}]$ is interpreted as a sequence ;

2. the expression sc.attr denotes a collection of values for attr associated with elements in sc. If the cardinality of attr is equal to 1 then sc.attr denotes a set and is modelled by $attr[sc]$. If the cardinality of attr is multiple but attr is not ordered then sc.attr denotes a bag and is modelled by $\{vv, nn|vv{\in}attr[sc]{\wedge}nn{\in}\mathcal{N}{\wedge}nn = card(attr^{-1}[\{vv\}]{\cap}sc)\}$. Otherwise there is no semantics for sc.attr and there is no therefore corresponding B formalisation ;

3. the expression bc.attr has no semantics if attr is multiple and ordered ; otherwise bc.attr denotes a bag of values of attr associated to the bag bc and is modelled by $\{vv, nn|vv{\in}attr[dom(bc)]{\wedge}nn{\in}\mathcal{N}{\wedge}nn = \Sigma(cc).(cc{\in}attr^{-1}[\{vv\}]{\cap}dom(bc)|bc(cc))\}$;

4. the expression seqc.attr has only semantics if the cardinality of attr is equal to 1 and in that case it denotes a sequence of values for attr and is modelled by $\lambda(ii).(ii{\in}dom(seqc)|attr(seqc(ii)))$.

Remark 4 The navigation operations without qualifiers are modelled in a similar manner to the attribute operations. For reasons of space, we omit here those derivation schemes.

Derivation 11 (Navigation operations with qualifiers) Given a binary association assos from the class Class to Class2. The association assos is qualified by attributes q1 : Q1, . . ., qn : Qn at the role end of Class. Given values and objects cc : Class, v1 : Q1, . . ., vn : Qn. Let's call roleClass2 the role end attached to Class2 in assos. The expression cc.roleClass2[v1, . . ., vn] is modelled in B by $(assos^{-1}{\otimes}q1{\otimes}\ldots{\otimes}qn)^{-1}[\{cc{\mapsto}v1{\mapsto}\ldots{\mapsto}vn\}]$, where cc, $v1$, ...,vn are the B formalisation of cc, v1,...vn and $q1$,...,qn are defined according to Derivation 9. Furthermore, if the multiplicity property of Class2 in assos is equal to 1, cc.roleClass2[v1,...,vn] can be expressed in B by $(assos^{-1}{\otimes}q1{\otimes}\ldots{\otimes}qn)^{-1}$ $(cc{\mapsto}v1{\mapsto}\ldots{\mapsto}vn)$.

Remark 5 It is always possible to define the B semantics for navigation operations with qualifiers on a set or a bag of objects. However those situations are rarely encountered and the corresponding derivation schemes are omitted in the current paper for reasons of spaces.

Derivation 12 (Navigation to association classes) Given assos a binary association class between two classes Class and Class2. Given cc, sc, bc, seqc an attribute, an object, a set of objects, a bag of objects and a sequence of objects of Class. Let's call $assos$, cc, sc, bc and $seqc$ the B formalisation of assos, cc, sc, bc and seqc according to Derivation 9 :

1. the expression cc.assos, which denotes the instance(s) of assos associated to cc, is modelled by $\{cc\}{\lhd}assos$; if the cardinality of the role end of Class2 in assos is equal to 1, cc.assos can also be modelled by $cc{\mapsto}assos(cc)$;

2. the expression sc.assos, which denotes the instances of assos associated to the elements of sc, is modelled by $sc{\lhd}assos$;

3. the expression bc.assos, which denotes a bag of instances of assos associated to the elements of bc, is modelled by $\{cc, nn|cc{\in}dom(bc){\lhd}assos{\wedge}nn$ ${\in}\mathcal{N}{\wedge}nn = bc(dom(\{cc\}))\}$;

4. the expression seqc.assos has only semantics if the cardinality of the role end of Class2 in assos is equal to 1 and in that case it denotes a sequence of instances of assos associated to the elements of seqc and is modelled by $\lambda(ii).(ii{\in}dom(seqc)|seqc(ii){\mapsto}assos(seqc(ii)))$.

Remark 6 (Let expressions) All the variables declared by let expressions should be replaced by their values before the transformation.

4 Derivation schemes specific for postconditions

This section presents the modelling of OCL expressions on postconditions of behavioural concepts in B. In the sequel, the terms postconditions refers to a class operation. However the derivation schemes can also be applied for use cases and events. As said earlier (cf. Section 2.3), the postconditions OCL of an operation oper are modelled in B by substitutions B in the body of the abstract operation B *oper* which correspond to oper. First of all are some definitions.

4.1 Definitions

Given an operation oper, the postconditions of oper can be considered as a constraint P(out1,...,outn,in1,...,inm) which links the potential “outputs” (cf. Definition 2) and potential ’“inputs” (cf. Definition 1) of oper.

Defintion 1 (Operation potential inputs) The set Input={in1,...,inm} of potential inputs of an operation oper consists of : (i) the eventual parameters stereotyped by “in” or “inout” whose value is provided upon every call to oper and (ii) the objects, the attributes and the associations available upon the operation call.

Defintion 2 (Operation potential output) The set Output={out1,...,outn} of potential outputs of an operation oper consists of : (i) the eventual return parameter, which is referenced by the name result in OCL, of oper ; (ii) the eventual parameters stereotyped by “out” or “inout” of oper ; (iii) the eventual newly created objects during the execution of oper and (iv) the eventual updated attributes and associations.

Definition 3 presents a standard style of the constraint P(out1,...,outn,in1,..,inm). In our opinion, the definition is enough generalised to be able to cover almost class operations. Our derivation schemes in the sequel are defined in reference to this definition.

Defintion 3 (Well-formed postconditions)

1. Every potential output outi is defined by an elementary constraint Pi(outi[,Input][,NewObject]), which defines outi according to elements of Input as well as the newly created objects (the elements of NewObject, which is a subset of Output) :
   (a) Pi states the creation of an object (outi) by oper ;
   (b) Pi is a comparison between the value of outi and the values of elements in Input∪NewObject. Two cases should be distinguished : (i) Pi is represented by outi=expression-CL(Input[∪NewObject]), meaning that outi is defined deterministically in terms of elements of Input[∪NewObject] ; (ii) Pi is represented by a boolean expression but not an equality on outi and eventual elements of Input[∪NewObject], meaning that outi is defined non deterministically in terms of elements of Input[∪NewObject] ;
   (c) Pi might represent the application of the operation forAll on a set of objects/values to be updated by oper.

2. The constraint P is a combination between the elementary constraints P1, ..., Pn and the operations and and if ... then ... else ... endif :
   (a) the condition part in an expression if ... then ... else ... endif refers to potential inputs ;
   (b) the elementary constraints P1, ..., Pn are linked by and in order to compose the body of expressions if ... then ... else ... endif ;
   (c) the expressions if ... then ... else ... endif can be nested ;
   (d) two body expressions in an expression if ... then ... else ... endif contain either another expression if ... then ... else ... endif or an expression and on elementary constraints ;

4.2 Modelling elementary constraints

Derivation 13 (Return parameter)

1. Given Pi in form result=expr(Input[∪NewObject]) to define deterministically the return value of oper. We add in the operation B *oper* the following substitution : $out = expr(Input[\cup NewObject])$,
   where out represents the return parameter of oper and $expr$ is the B formalisation of expr.

2. Given Pi in form expr(result,Input[∪NewObject]) to define non-deterministically the return parameter of oper. We can rewrite the constraint Pi in the following manner :
   - we introduce a temporary variable res which takes the place of result in the old Pi ;
   - we rewrite Pi in form :
     expr(res,Input[∪NewObject]) and
     result = res,

   the new form of Pi enables us to update *oper* as the following manner :

- we create a clause **any**...**where**... if it has not been created (cf. Derivation 17) ;
- we declare res, which models res :
  any..., res **where**
  $\ldots expr(res, Input[\cup NewObject])$
- we add the substitution $out := res$ in the body of **any**...**where**...

Remark 7

1. The B formalisation of expr is done using derivation schemes in Section 3. All the eventual occurrences of @pre are omitted.
2. Derivation 13 can be extended to apply for eventual parameters stereotyped by out or inout of oper.

Derivation 14 (Object creation) Given Pi a constraint specifying that an object cc of a class Class is created by oper. We create in $oper$:

- a clause **any**...**where**... if this clause has not been created (cf. Derivation 17) ;
- a temporary variable cc, which models cc :
  any ..., cc **where**
  $\ldots \wedge cc \in CLASS - class$
- a B substitution, which models the updating of effective instance set of Class by the object cc, in the body **any**...**where**...
  $\| \; class := class \cup \{cc\} \ldots$

Derivation 15 (Updating attribute value of an object) Given an attribute attr and an object cc of a class Class :

1. given Pi in form cc.attr=expr(Input[∪NewObject]) to define deterministically the new value of cc.attr in which the cardinality of attr is equal to 1. We model the constraint Pi by the substitution B
   $attr := attr \Leftarrow\!\!\!\!+ \{cc \mapsto expr\} \ldots$;
2. given Pi in form cc.attr=expr(Input[∪NewObject]) to define deterministically cc.attr in which the cardinality of attr is "*". We model Pi by the substitution :
   $attr := attr \cup \{cc\} \times expr \ldots$,

   in the two cases above, $attr$, cc and $expr$ are the B formalisation of attr, cc and expr.

Remark 8 Derivation 15 can be extended for updating associations.

Derivation 16 (The operation forAll on a set of objects) Given a set sc of objects and an attribute attr of the cardinality 1 of a class Class :

1. given Pi in form sc->forAll(p|p.attr =expr(Input[∪NewObject]) to define deterministically the value of attribute attr of all elements in sc. We model Pi by the following substitution B : $attr := attr \Leftarrow\!\!\!\!+ sc \times \{expr\} \ldots$,
2. given Pi in form sc->forAll(p|expr(p.attr,Input[∪NewObject])) to define non deterministically the value of attribute attr of all elements in sc. We model Pi in creating in $oper$:
   - a clause **any**...**where**... if this clause has not been created (cf. Derivation 17) ;
   - a temporary variable at defined as :
     any..., at **where**
     $at \in class \rightarrow\!\!\!\!\!+ typeAttr \wedge$
     $dom(at) = sc \wedge$
     $\forall (tt).(tt \in ran(at) \Rightarrow$
     $expr(at, Input[\cup NewObject])) \ldots$
   - the substitution :
     $attr := attr \Leftarrow\!\!\!\!+ at \ldots \|$
     in the body of the clause **any**...**where**...,

   in the two cases above, $attr$ and sc are the B formalisation of attr and sc; and $expr$ is the B formalisation of expr in which tt replaces p.attr.

Remark 9

1. Derivation 16 did not consider the case where several attributes of the same set of objects have changed their values. However this situation can be solved by applying Derivation 16 several times.
2. We did not consider the case where forAll is applied on a sequence or a bag of objects since those situation has no semantics in the context of postconditions.
3. Derivation 16 can be extended for the case where the body of forAll is the navigation operation.
4. Derivation 16 can also be extended for the case where the cardinality of attr is greater than 1. However this situation is rarely encountered.

4.3 Substitution unification

In the previous section, we have presented the principles for modelling an elementary constraint. In general, each elementary constraint gives rise to a B substitution. This section discusses the way to unify the derived substitutions.

Derivation 17 (Substitution unification) Given P1 and ... and Pk an expression OCL with the operation and and the elementary constraint P1, ..., Pk in the postconditions of oper. The substitutions B derived from this expression are unified by the following manner :

- all the eventual temporary variables as well as the eventual created objects are declared in the same clause **any**...**where**... ;
- all the substitutions involving the same B variable are unified ;
- the substitutions for the different B variables are placed on parallel ("||") and they constitutes the body of the clause **any**...**where**... if this clause is created.

Derivation 18 (The expression if ... then ... else ... endif) Given if cond then expr1 else expr2 endif an expression of postconditions of an operation oper. The B formalisation of this expression gives rise to a clause :

- **if** $cond$ **then** $subst(expr1)$ **else** $subst(expr2)$ **end**, if expr2 is not an expression if ... then ... else ... endif or
- **if** $cond$ **then** $subst(expr1)$
  elsif $cond2$ **then** $subst(expr21)$ ... **end**, if expr2 is also in form if cond2 then expr21 ... ;

where $cond$ is the B formalisation of cond and $subst(expr)$ denotes the substitutions derived from expr.

Remark 10 $skip$ is the substitution of true.

Derivation 19 (Operation) Given {pre,post} the preconditions and the postconditions in OCL of an operation oper, where post is defined according to Definition 3. The B operation $oper$ modelling oper is generated by the following manner :

- the signature and a part of precondition of $oper$ for typing eventual parameters are generated according to derivation schemes described in [14, 11] ;
- if pre is not true then the precondition in $oper$ is augmented by predicates derived from pre in using derivation schemes in Section 3 ;
- the substitution part of $oper$ is generated from post using derivation schemes previously presented in this section.

In the context of postconditions Remark 6 may be applied. However, there is also another alternative for let expressions as described in Derivation 20.

Derivation 20 (let ... in postconditions) The expression let v1 : type1 = val1, ..., vn : typen = valn in expr in postconditions of an operation oper gives rise to a clause $Let...$ in $oper$:

let $v1, ..., vn$ **be**
  $v1 = val1 \wedge$
  $.. \wedge$
  $vn = valn$
in
  $subst(expr)$
end

where $v1$, ..., vn, $val1$, ..., $valn$ and $subst(expr)$ denote the B formalisation of v1, ..., vn, val1, ..., valn and expr.

5 Conclusion

This paper presents a systematic way for transforming OCL expressions into B, which can be applied to generate B supplementary invariants as well as B abstract operations in B specifications, which are generated according to derivation procedures in our previous works [8, 11, 12], from the OCL specifications for the supplementary class invariants and for behavioural concepts in UML specifications.

References

[1] J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5.

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

[3] J.M. Bruel. Integrating Formal and Informal Specification Techniques. Why? How? In *the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*, pages 50–57, Boca Raton, Florida (USA), 1998. Available at http://www.univ-pau.fr/∼bruel/publications.html.

[4] J.M. Bruel, J. Lilius, A. Moreira, and R.B. France. Defining Precise Semantics for UML. In *Object-Oriented Technology*, LNCS 1964, pages 113–122, Sophia Antipolis and Cannes (F), June 12-16, 2000. ECOOP 2000 Workshop Reader.

[5] D. Coleman, P. Arnold, St. Bodoff, Ch. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development : The Fusion Method*. Prentice Hall, 1994.

[6] R. Laleau and A. Mammar. A Generic Process to Refine a B Specification into a Relational Database Implementation. In *ZB 2000: Formal Specification and Development in Z and B*, LNCS 1878, York (UK), August/September 2000. Springer-Verlag.

[7] K. Lano. *The B Language and Method : A Guide to Practical Formal Development*. FACIT. Springer-Verlag, 1996. ISBN 3-540-76033-4.

[8] H. Ledang. Des cas d'utilisation à une spécification B. In *Journées AFADL'2001 : Approches Formelles dans l'Assistance au Développement de Logiciels*, Nancy (F), 11-13 juin, 2001. http://www.loria.fr/∼ledang/publications/afadl01.ps.gz.

[9] H. Ledang and J. Souquières. Formalizing UML Behavioral Diagrams with B. In *the Tenth OOPSLA Workshop on Behavioral Semantics: Back to Basics*, Tampa Bay, Florida (USA), October 15, 2001. http://www.loria.fr/∼ledang/publications/oopsla01.ps.gz.

[10] H. Ledang and J. Souquières. Integrating UML and B Specification Techniques. In *the Informatik2001 Workshop on Integrating Diagrammatic and Formal Specification Techniques*, Vienna (Autria), September 26, 2001. http://www.loria.fr/∼ledang/publications/informatik01.ps.gz.

[11] H. Ledang and J. Souquières. Modeling Class Operations in B: Application to UML Behaviral Diagrams. In *ASE2001: the 16th IEEE International Conference on Automated Software Engineering*, pages 289–296, Loews Coronado Bay, San Diego (USA), November 26-29, 2001. IEEE Computer Society. http://www.loria.fr/∼ledang/publications/ase01.ps.gz.

[12] H. Ledang and J. Souquières. Contributions for Modelling UML State-Charts in B. In *IFM 2002: Third International Conference on Integrated Formal Methods*, LNCS 2335, pages 109–127, Turku (Fin), May 15-17, 2002. Springer-Verlag. http://www.loria.fr/∼ledang/publications/ifm2002.ps.gz.

[13] H. Ledang and J. Souquières. Integration of UML models using B notation. In *WITUML: Workshop on Integration and Transformation of UML models*, Málaga (Spain), June 11, 2002. Available at http://www.loria.fr/∼ledang/publications/wituml02.ps.gz.

[14] E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In *FM'99 : World Congress on Formal Methods in the Development of Computing Systems*, LNCS 1708, Toulouse (F), September 1999. Springer-Verlag.

[15] H.P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*. PhD thesis, Conservatoire National des Arts et Métiers - CEDRIC, Paris (F), décembre 1998.

[16] The Object Management Group (OMG). *OMG Unified Modeling Language Specification*, June 1999. Version 1.3.

[17] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998. ISBN 0-201-30998-X.

[18] STERIA - Technologies de l'Information, Aix-en-Provence (F). *Atelier B, Manuel Utilisateur*, 1998. Version 3.5.