Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B

Hung Ledang, Jeanine Souquières

Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B

Hung LEDANG Jeanine SOUQUIÈRES

LORIA - Université Nancy 2 - UMR 7503
Campus scientifique, BP 239
54506 Vandœuvre-lès-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 an integral part of UML, into B. Our derivation schemes allow to automatically derive into B not only the complementary class invariants, the guard conditions in state-charts (in OCL) but also the OCL specifications for class operations.

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

1 Introduction

The Unified Modelling Language (UML) [19] and the B language [1] are two specification techniques well recognised in software engineering for their application capability in industry. Their integration is motivated by the hope to be be able to use them jointly in a practice, unified and rigorous software development. The practicity comes from UML as the specification technique largely practised and accepted in software industry. It also comes from B as the formal technique whose industrial application are effective [9, 5]. The rigour comes from B as a formal method. The unification comes at the same time from UML and B since they are used during the whole software development from requirements expressions until the design and programming.

The transformation from UML specifications into B aims at a two-fold goal. On the one hand, one can use B powerful support tools like AtelierB [20], B-Toolkit [2] to analyse and detect inconsistencies within UML specifications (see further discussions in [12]). On the other hand, we can also use UML specifications as the starting point to develop B specifications which can then be refined automatically to an executable code [10].

Meyer and Souquières [17] and Nguyen [18], based on the previous work of Lano [11], 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 predicates between B variables for the classes in question.

In [13] we have proposed approaches for modelling UML operations (operations declared in class diagrams). Each UML operation is firstly modelled by a B abstract operation in which the expected effects of such an operation on related data is specified directly on the derived data. If a UML operation is realised by an interaction or activity diagram then the B operation corresponding is refined to give rise a B implementation operation.

The UML-B derivation schemes for UML structural concepts and for UML operations are used in the derivation procedure which allow to integrate UML class and collaboration diagrams into one 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 UML diagrams is generated. Therefore, the B specification should be completed with invariants for supplementary class invariant as well as B operations' body.

As cited in the UML literature [19], OCL (Object Constraint Language) is often used to specify supplementary class invariant as well as pre- and post-conditions of UML operations; in the continuity of our research on integration of 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.

Section 2 outlines the principles of the transformation from OCL expressions into B. The derivation schemes for OCL types and their operations are presented in Section 3. The derivation schemes specific for postconditions are presented in Section 4. A case study is presented in Section 5. Discussions in Section 6 conclude our presentation.

2 From OCL expressions to B : overview

2.1 The OCL language

The Object Constraint Language (OCL) is now an integral part of UML [19]. 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 "is-Query" 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 UML operations.

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 Zermelo-Frankel set theory and first order logic. Specifications are composed of abstract machines similar to modules or classes; they consist of variables, invariance properties related to those variables and operations. The state of the system, i.e. the set of variable values, is only modifiable by operations. The means by which B operations specifies state transitions is the *generalised substitution language* whose semantics is defined by means of predicate transformers [8] and the weakest precondition [7]. A generalised substitution is an abstract mathematical programming construct, built up from basic substitution $x := e$, corresponding to assignments to state variables, via a set of operators like No-op ($skip$), bounded choice (**choice** S_1 **or** S_2 ...), preconditioning (**pre** P **then** S **end**), unbounded non-determinism (**var** v **in** S **end**, **any** v **where** P **then** S **end**), guarding, sequential composition, multiple generalised composition and looping.

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 into 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 of UML operations. 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 of UML operations.

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 of UML operations) should be represented by a B expression.

The derivation schemes from OCL to B for types and 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 of UML operations. To model postconditions of UML operations, the use of B generalised substitutions is necessary. The OCL expressions involving values after the execution of a UML operation are translated into B substitutions.

2.4 Related work

The transformation from OCL into other formal notations have been discussed in several works [3, 4], however our choice of B as the target notation is motivated by the fact that B is a stable language with powerful support tools that have been advocated in industrial applications [9, 5].

The transformation from OCL into B has been previously discussed by Marcano and Lévy [15], in which the authors presented the derivation schemes from OCL expressions to B expressions. However there are several shortcomings in this proposal:

1. the postconditions of behavioural concepts have not been considered;

2. the fact to formalise an OCL operation by a B corresponding operation seems to be ambiguous due to the mismatch between OCL type system and B type system. For example, the OCL type String does not correspond to the B type $STRING$; B does not support the real type;

3. the fact to model OCL collection types by B sets is only appropriate if the collection type is a set; consequently the modelling in B of collection operations collect, select etc. is not appropriate.

Dealing with the three shortcoming above is essentially our contributions in this paper.

3 Modelling OCL types and their operations

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.

3.1 Predefined basic types

Derivation 1 (Integer) In B there are two predefined types $\mathcal{Z}$ and INT which correspond to the OCL type Integer. $\mathcal{Z}$ is chosen as the formalisation of Integer since $\mathcal{Z}$ is more abstract than INT. An integer value nn in OCL is modelled in B as a value nn of $\mathcal{Z}$. As shown in Table 1, all Integer OCL operations but "/" can be expressed by a B expression on $\mathcal{Z}$.

Remark 1 (Modelling the operation "/")

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

2. The fact of using a ratio to express the real division "/" between two integers implies to define the formalisation in B for operations on ratios. Operations between an integer and a ratio can be converted into operations on ratios. As an example, the operation "+" between two ratio a/b and c/d can be modelled by $a \cdot d + c \cdot b \mapsto b \cdot d$; details of a such formalisation can be found in [14].

Operations OCL	Semantics in B
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	$max(-a, a)$
a/b	$a \mapsto b$

Table 1. Operations on integers

Derivation 2 (Boolean) The OCL type Boolean is modelled in B by its correspondence $BOOL$. Table 2 shows the B formalisation of OCL Boolean operations, where a, b and c are three booleans and a, b and c are their formalisation in B. An OCL boolean variable or constant x is modelled in B by x (variable or constante). An OCL boolean expression exp is modelled in B as a predicate exp or as a boolean expression $bool(exp)$, where exp is derived from exp according to the rules in Table 2 and $bool$ is a B predefined function to convert a predicate into a boolean value.

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

Table 2. Operations on booleans

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 $STRING$). Our solution is to use a B user-defined type $seq(0..255)$ to model String. The background idea is that a string can be considered as a sequence of characters and the range $0..255$ models the set of possible characters. A string str in OCL is therefore modelled by an element str of $seq(0..255)$. The OCL operations on String (except two operations toUpper and toLower) can be expressed by B expressions on $seq(0..255)$ (cf. 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 abstract machine level.

Operations OCL	Semantics in B
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. Operations on strings

Derivation 4 (Real) There is no B predefined type for real values, a definitive solution for modelling the OCL type Real in B has not been yet achieved. In waiting for a such solution, a temporary solution, inspired from Remark 1, can be used. It is to approximate a real value by a ratio. Hence the type Real can be modelled in B by relation $\mathcal{Z} \leftrightarrow \mathcal{Z}$. The OCL operations on Real are modelled in B in a similar manner of the operations on ratios.

Derivation 5 (Enumeration types) Each enumeration type Enum={val_1,...,val_n} is modelled in B by an enumerated set serving as a user-defined type $Enum = \{val_1, ..., val_n\}$. Each element val_i# in Enum is modelled by an element val_i 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#=b#	$a = b$
a#<>b#	$\neg(a = b)$

Table 4. Operations on enumerations

Derivation 6 (Object types) According to Meyer and Souquières [17], for each class Class, the B constant $CLASS$ models the possible instance set and the B variable $class$, which is defined as a subset of $CLASS$, models 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 $class$. An object cc of Class is modelled in B by an element cc of $class$.

3.2 Collection types

Derivation 7 (Collection types) Given T an OCL type and T its B formalisation:

1. the collection type Set(T), which denotes all subsets of T, is modelled in B by $\mathcal{P}(T)$;
2. the collection type Bag(T), which denotes all multi-sets on T, is modelled in B by $T \nrightarrow \mathcal{N}$. An element bb of Bag(T) can be modelled as $bb \in T \nrightarrow \mathcal{N}$ and for each element tt of bb, $bb(tt)$ denotes the occurrence number of tt in bb;
3. the collection type Sequence(T), which denotes all lists of elements of T, is directly modelled in B by $seq(T)$.

Using Derivation 7, almost predefined operations on collection types (except asSequence on sets or bags and excluding on sequences[1]) can be expressed by a B expression. The semantics of OCL operations select, reject, collect, forAll, exists on collection types can also be interpreted by B expressions. Details of those formalisations can be found in [14], Derivation 8 shows some examples.

Derivation 8 (OCL operations on collection types) Given T an OCL type on which the collection types are defined and ss : Set(T); bb : Bag(T); se, se_2 : Sequence(T); tt :T; boolexprtt is a boolean expression on tt and exprtt is an expression on tt. Let's call T, ss, bb, se, se_2, tt, $boolexprtt$ and $exprtt$ respectively the B formalisation of T ss, bb, se, se_2, tt, boolexprtt and exprtt:

1. the OCL expression bb−>includes(tt), which checks whether tt is an element of the bag bb, is modelled in B by $tt \in dom(bb)$;
2. the OCL sequence union expression se−>union(se_2) is modelled in B by the sequence concatenation expression $se \frown ss_2$;
3. the OCL expression ss−>select(tt|boolexprtt), which extracts elements tt of ss satisfying boolexprtt, is modelled in B by $\{tt | tt \in ss \wedge boolexprtt\}$;
4. the OCL expression ss−>collect(tt|exprtt), which derives a collection (a bag) of values exprtt computed on every element tt of the set ss, is modelled in B by $\{tt, nn | tt \in exprtt[ss] \wedge nn \in \mathcal{N} \wedge nn = card(\{xx | xx \in ss \wedge exprtt(xx) = tt\})\}$.

[1] The semantics of the operation asSequence on sets or bags has not been defined in OCL therefore we cannot model it in B. It is the same for the operation excluding on a sequence.

3.3 Property access operations

OCL expressions can refer to attributes, association ends and "is-Query" operations thanks to property access operations. A property access operation on an object might return 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 a property access operation on the result of another property access operation. Hence the target of a property access operation might be an object, a set of objects, a multi-set of objects or even a sequence of objects. Our derivation schemes for property access operations are based on the derivation schemes for UML structural concepts and the derivation schemes for collection types (cf. Derivation 7). The derivation schemes for UML structural concepts are detailled in [17, 18, 16]; in Derivation 9, we recall only essential points which facilitate the presentation afterwards.

Derivation 9 (Structural concepts)

1. 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.

2. A binary association assos between two classes Class and Class_2 is modelled by a B variable $assos$ defined as $assos \in class \leftrightarrow class_2$. If there are eventual qualifiers $q_1 : Q_1$, ..., $q_n : Q_n$ at the role end of Class, they are modelled in a similar manner to an attribute: $q_1 \in class_2 \leftrightarrow Q_1, \ \ldots, \ q_n \in class_2 \leftrightarrow Q_n$. We add also a B invariant linking $assos$ and q_1,...,q_n as follows: $(assos^{-1} \otimes q_1 \otimes \ldots \otimes q_n)^{-1} \in class \times Q_1 \times \ldots \times Q_n \leftrightarrow class_2$ [2]. 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 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 3 The navigation operations without qualifiers are modelled in a similar manner to the attribute operations.

Derivation 11 (Navigation operations with qualifiers) Given a binary association assos from the class Class to Class_2. The association assos is qualified by attributes $q_1 : Q_1, \ \ldots, \ q_n : Q_n$ at the role end of Class. Given values and objects cc : Class, $v_1 : Q_1, \ \ldots, \ v_n : Q_n$. Let's call $\mathsf{roleClass}_2$ the role end of Class_2 in assos. The expression $\mathsf{cc.roleClass}_2[v_1, \ \ldots, \ v_n]$ is modelled in B by $(assos^{-1} \otimes q_1 \otimes \ldots \otimes q_n)^{-1}[\{cc \mapsto v_1 \mapsto \ldots \mapsto v_n\}]$, where cc, v_1, ...,v_n are the B formalisation of cc, v_1,...v_n and q_1,...,q_n are defined according to Derivation 9. Furthermore, if the multiplicity property of Class_2 in assos is equal to 1, $\mathsf{cc.roleClass}_2[v_1,...,v_n]$ can be expressed in B by $(assos^{-1} \otimes q_1 \otimes \ldots \otimes q_n)^{-1} \ (cc \mapsto v_1 \mapsto \ldots \mapsto v_n)$.

Remark 4 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 here.

Derivation 12 (Navigation to association classes) Given assos a binary association class between two classes Class and Class_2. Given cc, sc, bc, seqc 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 Class_2 in assos is equal to 1, cc.assos can also be modelled by $cc \mapsto assos(cc)$;

[2] '$\otimes$' is the direct product between two relations.

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 $Class_2$ 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 $\lambda(ii).(ii \in dom(seqc) | seqc(ii) \mapsto assos(seqc(ii)))$.

Derivation 13 ("is-Query" operations) The call to "is-Query" class operations can appear in OCL expressions, however, we can not call B operations in B expressions. Therefore, we propose to create for each "is-Query" operation Class::oper(p_1:P_1,...,p_n:P_n):P, which appears in OCL expressions, a B variable *oper* defined by $oper \in class \otimes P_1 \otimes \ldots \otimes P_n \nrightarrow P$. A call oper($v_1$,...,$v_n$) to an object cc of Class in OCL expressions is therefore modelled by $oper(cc \mapsto v_1 \mapsto \ldots \mapsto v_n)$ in the corresponding B expressions. Note equally that the call oper to a set, a bag or a sequence of objects of Class is rarely encountered, hence the corresponding derivation schemes are omitted here.

Remark 5 (Let expressions) All the variables declared by let expressions should be replaced by their values before the transformation.

4 Modelling postconditions

This section presents the modelling of OCL expressions on postconditions of an UML operation. As said earlier (cf. Section 2.3), the OCL postconditions of an UML operation oper are modelled in B by generalised substitutions in the body of the B abstract operation *oper* for 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(out_1...,out_n,in_1,...,in_m) which links the potential "outputs" (cf. Definition 2) and potential '"inputs" (cf. Definition 1) of oper.

Defintion 1 (Operation potential inputs) The set Input={in_1,...,in_m} of potential inputs of an operation oper consists of: (i) the possible 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={out_1,...,out_n} of potential outputs of an operation oper consists of: (i) the possible return parameter, which is referenced by the name result in OCL, of oper; (ii) the possible parameters stereotyped by "out" or "inout" of oper; (iii) the possible newly created objects during the execution of oper and (iv) the possible updated attributes and associations.

Definition 3 presents a standard style of the constraint P(out_1,...,out_n,in_1,..,in_m). 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 out_i is defined by an elementary constraint P_i(out_i[,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) P_i states the creation of an object (out_i) by oper;

   (b) P_i is a comparison between the value of out_i and the values of elements in Input∪NewObject. Two cases should be distinguished: (i) P_i is represented by out_i=expr(Input[∪NewObject]), meaning that out_i is defined deterministically in terms of elements of Input[∪NewObject]; (ii) P_i is represented by a boolean expression but not an equality on out_i and possible elements of Input[∪NewObject], meaning that out_i is defined non deterministically in terms of elements of Input[∪NewObject];

   (c) P_i 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 P_1, ..., P_n 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 P_1, ..., P_n 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 Elementary constraints

Derivation 14 (Return parameter)

1. Given P_i in the form result=expr(Input[∪NewObject]) to define deterministically the return value of oper. We add in the B operation *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 P_i in the form expr(result,Input[∪NewObject]) to define non-deterministically the return parameter of oper. We can rewrite the constraint P_i in the following manner:

   - we introduce a temporary variable res which takes the place of result in the old P_i;
   - we rewrite Pi as:
   expr(res,Input[∪NewObject]) and
   result = res,

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

   - we create a clause **any**...**where**... if it has not been created (cf. Derivation 18);
   - 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 6

1. The B formalisation of expr is done using derivation schemes in Section 3; all the possible occurrences of @pre are omitted without losing the semantics of @pre. To justify this point, let's take an exemple: in the B assignement statement $x := a$, the value of a is always refered to be the value before execution of the corresponding operation.

2. Derivation 14 can be extended to apply for possible parameters stereotyped by out or inout of oper.

Derivation 15 (Object creation) Given P_i 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 18);

- a temporary variable cc, which models cc:
any ..., cc **where**
$\ldots \wedge cc \in CLASS - class$

- a B substitution, which models the updating of a set of effective instances of Class by the object cc, in the body **any**...**where**...
$class := class \cup \{cc\}$

Derivation 16 (Updating attribute value of an object) Given an attribute attr and an object cc of a class Class:

1. given P_i in the 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 P_i by the substitution B
$attr := attr \lhd\!\!- \{cc \mapsto expr\}$;

2. given P_i in the form cc.attr=expr(Input[∪NewObject]) to define deterministically cc.attr in which the cardinality of attr is multiple. We model P_i by the substitution:
$attr := attr \cup \{cc\} \times expr$.

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

Remark 7 Derivation 16 can be extended for updating associations.

Derivation 17 (forAll operation on an object set) Given a set sc of objects and an attribute attr of the cardinality 1 of a class Class:

1. given P_i in the form sc−>forAll(p|p.attr =expr(Input[∪ NewObject]) to define deterministically the value of attribute attr of all elements in sc. We model P_i by the following substitution B: $attr := attr \lhd\!\!- sc \times \{expr\}$,

2. given P_i in the 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 P_i in creating in *oper*:

   - a clause **any**...**where**... if this clause has not been created (cf. Derivation 18);
   - a temporary variable at defined as:
   any..., at **where**
   $at \in class \nrightarrow typeAttr \wedge$
   $dom(at) = sc \wedge$
   $\forall (tt).(tt \in ran(at) \Rightarrow$
   $expr(at, Input[\cup NewObject]))$
   - the substitution:
   $attr := attr \lhd\!\!- at \ldots ||$
   in the body of the clause **any**...**where**...

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

Remark 8

1. Derivation 17 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 17 several times.

2. We did not consider the case where forAll is applied on a sequence or a bag of objects since those situations have no semantics in the context of postconditions.

3. Derivation 17 can be extended for the case where the body of forAll is the navigation operation.

4. Derivation 17 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 18 (Substitution unification) Given "P_1 and ... and P_k" an OCL and expression on the elementary constraints P_1, ..., P_k in the postconditions of oper. The B substitutions derived from this expression are unified in the following manner:

- all the possible temporary variables as well as the possible 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 constitute the body of the clause **any**...**where**... if this clause has been created.

Derivation 19 (The expression if ... then ... else ... endif) Given if cond then $expr_1$ else $expr_2$ endif an expression of postconditions of an operation oper. The B formalisation of this expression gives rise to a clause:

- **if** $cond$ **then** $subst(expr_1)$ **else** $subst(expr_2)$ **end**, if $expr_2$ is not an expression if ... then ... else ... endif or

- **if** $cond$ **then** $subst(expr_1)$
  elsif $cond_2$ **then** $subst(expr_{21})$ ... **end**, if $expr_2$ is also in the form if $cond_2$ then $expr_{21}$ ...;

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

Remark 9 $skip$ is the substitution of true.

Derivation 20 (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 in the following manner:

- the signature and a part of precondition of $oper$ for typing possible parameters are generated according to derivation schemes described in [17, 13];

- 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 5 may be applied. However, there is also another alternative for let expressions as described in Derivation 21.

Derivation 21 (let ... in postconditions) The expression let v_1 : $type_1$ = val_1, ..., v_n : $type_n$ = val_n in expr in postconditions of an operation oper gives rise to a clause Let... in $oper$:

let $v_1, ..., v_n$ **be**
$v_1 = val_1 \wedge$
.. $\wedge$
$v_n = val_n$
in
$subst(expr)$
end

where v_1, ..., v_n, val_1, ..., val_n and $subst(expr)$ denote the B formalisation of v_1, ..., v_n, val_1, ..., val_n and expr.

5 Transformation example

The class diagram in Figure 1 is extracted from the UML specification [13] for the pump component in a system controlling petrol dispensing, customer payment handling and petrol tank level monitoring as described in chapter 6 of [6].

The class diagram is composed of an aggregation of the classes Pump, Clutch, Display, Gun, Holster and Motor which model fives pumps with included components. We defined the class Delivery to model deliverance. The attributes, the operations as well as data types used by attributes and operations are presented in [6].

5.1 The operation Pump::enable_Pump in OCL

To illustrate the application of derivation schemes, let's consider the OCL specification (Figure 2) of the class operation Pump::enable_Pump.

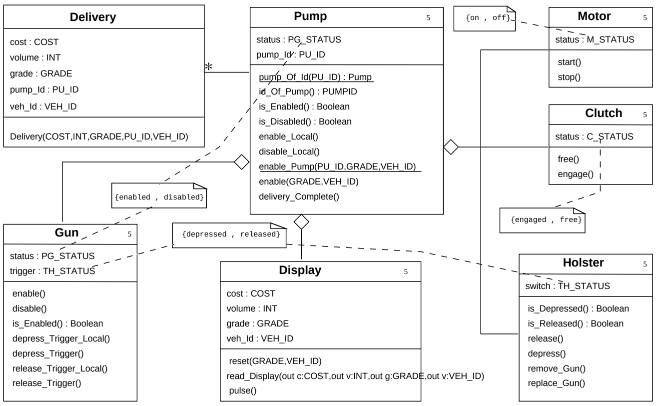

Figure 1. Pump class diagram

CONTEXT *Pump::enable_Pump*$(pi : PU\_ID, gg : GRADE,$
$vi : VEH\_ID)$: **void**
PRE
Pump.allInstances$\rightarrow$*collect*$(pump\_Id)\rightarrow$*includes*(pi)
POST
let pp : $Set(Pump)$ = *Pump.allInstances* $\rightarrow$ *select*$(tt|$
tt.pump_Id@pre=pi **and** *tt.status@pre=disabled*$)$
in
if $pp \rightarrow$ *notEmpty* **then**
$pp \rightarrow forall(p \mid p.status = enabled)$ **and**
$pp \rightarrow forall(p \mid p.display.grade = gg)$ **and**
$pp \rightarrow forall(p \mid p.display.cost = costOfGrade(gg))$ **and**
$pp \rightarrow forall(p \mid p.display.volume = 0)$ **and**
$pp \rightarrow forall(p \mid p.display.veh\_Id = vi)$ **and**
$pp \rightarrow forall(p \mid p.motor.status = on)$ **and**
$pp \rightarrow forall(p \mid p.clutch.status = free)$
else $true$ **endif**

Figure 2. Operation enable_Pump in OCL

The operation has three arguments: the identifier pi of the pump to be invoked; the category gg of the petrol to be distributed and the registration number vi of the vehicle. The precondition says that the pump with identifier pi exists. The postcondition is formatted according to Definition 3 and says that in case the pump with identifier pi is disabled, it should be enabled and its display is initialised, its motor is running and its clutch is free, otherwise nothing happens. Notice that although there is only one pump with the identifier pi but we can not extract it from the set of effective pumps (denoted by Pump.allInstances) due to restrictions of OCL collection operations. That is why the postcondition is composed of expressions on a singleton pp whose unique element is the disabled pump with identifier pi.

5.2 The operation Pump::enable_Pump in B

Figure 3 represents the B specification of the operation Pump::enable_Pump which is derived from the OCL specification in Figure 2 by applying systematically Derivation 20 and implied derivation schemes in Section 3 and Section 4.

...
OPERATIONS
pump_enable_Pump(pi, gg, vi) =
pre
$pi \in PU\_ID \wedge$
$gg \in GRADE \wedge$
$vi \in VEH\_ID \wedge$
$pi \in dom(\{tt, nn | tt \in pump\_pump\_Id[pump] \wedge nn \in \mathcal{N} \wedge$
$nn = card(\{xx | xx \in pump \wedge pump\_pump\_Id(xx) = tt\})\})$
then
let pp **be**
$pp = \{tt | tt \in pump \wedge pump\_pump\_Id(tt) = pi \wedge$
$pump\_status(tt) = pg\_status\_disabled\}$
in
if $\neg(pp = \phi)$ **then**
$pump\_status := pump\_status \lhd\!\!-$
$pp \times \{pg\_status\_enabled\} ||$
$display\_grade := display\_grade \lhd\!\!-$
$displayPump[pp] \times \{gg\} ||$
$display\_cost := display\_cost \lhd\!\!-$
$displayPump[pp] \times \{costOfGrade(gg)\} ||$
$display\_volume := display\_volume \lhd\!\!-$
$displayPump[pp] \times \{0\} ||$
$display\_veh\_Id := display\_veh\_Id \lhd\!\!-$
$displayPump[pp] \times \{vi\} ||$
$motor\_status := motor\_status \lhd\!\!-$
$motorPump[pp] \times \{m\_status\_on\} ||$
$clutch\_status := clutch\_status \lhd\!\!-$
$clutchPump[pp] \times \{c\_status\_free\}$
else skip end
end
end;
...

Figure 3. Operation enable_Pump in B

As an example, let's consider the expression Pump.allInstances−>collect(pump_Id)−>includes(pi) in the OCL precondition of Pump::enable_Pump. Analysing its sub-expressions we know that: the expression Pump.allInstances denotes the set of effective instances of the class Pump. Applying the operation collect(pump_Id) on set Pump.allInstances we obtain a bag. The operation includes(pi) will check whether the identifier pi is an element of the bag Pump.allInstances−>collect(pump_Id). Let T(expr) denotes the B formalisation of the OCL expression expr and $pump\_pump\_Id$ the B variable for the attribute pump_Id of the class Pump. The transformation process from the precondition Pump.allInstances−>collect(pump_Id)−>includes(pi) into B proceeds as follows:

T(Pump.allInstances−>collect(pump_Id)−>includes(pi))

== *(cf.* point 1 *in* Derivation 8*)*

$pi \in dom($T(Pump.allInstances−>collect(pump_Id))$)$

== *(cf.* point 4 *in* Derivation 8*)*

$pi \in dom(\{tt, nn | tt \in pump_pump_Id[$T(Pump.allInstances)$] \wedge nn \in \mathcal{N} \wedge nn = card(\{xx | xx \in$ T(Pump.allInstances) $\wedge pump\_pump\_Id(xx) = tt\})\})$

== *(cf.* Derivation 6*)*

$pi \in dom(\{tt, nn | tt \in pump\_pump\_Id[pump] \wedge nn \in \mathcal{N} \wedge nn = card(\{xx | xx \in pump \wedge pump\_pump\_Id(xx) = tt\})\})$

Remark 10 In order to avoid the eventual name conflicting in the derivation of attributes into B, we prefixe the name of attributes by the the class name. Hence the attribute pump_Id is modelled by the variable $pump\_pump\_Id$. It is similar for constants in the enumeration type; the constant enabled of the enumeration type PG_STATUS is modelled by $pg\_status\_enabled$.

Let's consider now the application of derivation schemes specific for postconditions. According to Derivation 21, we transform the let expression in the postcondition of the OCL operation Pump::enable_Pump into a clause **let**... in the B operation $pump\_enable\_pump$. The substitution body **if**... of **let**... is derived from the body expression if ... of let ... using Derivation 19. The B guard condition $\neg(pp = \phi)$ is derived from the OCL guard condition pp−>notEmpty. The body of the substitution **if**... is generated from the the expression if ... according to Derivation 18 and Derivation 17.

6 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 generated according to the derivation procedure in our previous work [13], from the OCL specifications for the supplementary class invariants and for UML operations in UML specifications.

For the further work, the prototype ArgoUML+B bas been developped from ArgoUML[3], a platform for editing UML diagrams with the code in java and freely available. We have added to ArgoUML the possibility to transform a set of class and collaboration diagrams into a B specificattion according the derivation procedure in [13]. In ArgoUML, there is a component “ocl-argo”, which is in charge to parse OCL expressions. We would like to extend this component by implementing our derivation schemes from OCL into B. So that the OCL constraints within UML class diagrams can be transformed into B.

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] Thomas Baar, Bernhard Beckert, and Peter H. Schmitt. An extension of Dynamic Logic for modelling OCL's $@pre$ operator. In *Proceedings, Fourth Andrei Ershov International Conference, Perspectives of System Informatics, Novosibirsk, Russia*, LNCS 2244, pages 47–54. Springer, 2001.

[4] Bernhard Beckert, Uwe Keller, and Peter H. Schmitt. Translating the Object Constraint Language into first-order predicate logic. In *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*, 2002. To appear.

[5] P. Behm, P. Desforges, and J.-M. Meynadier. MÉTÉOR: An Industrial Success in Formal Development, April 1998. An invited talk at the 2nd Int. B conference, LNCS 1939.

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

[7] E.W.D. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.

[8] D. Gries. *The Science of Programming*. Springer Verlag, New York (USA), 1981. 350 pages.

[9] H. Habrias, editor. *Putting Into Practice Methods and Tools for Information System Design - 1st Conference on the B Method*, Nantes (F), November 1996.

[10] 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.

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

[12] H. Ledang and J. Souquières. Formalizing UML Behavioral Diagrams with B. In *the Tenth OOPSLA Workshop on Behavioral Semantics: Back to Basics*, pages 162–171, Tampa Bay, Florida (USA), October 15, 2001. Notheastern University Press. http://www.loria.fr/∼ledang/publications/oopsla01.ps.gz.

[13] H. Ledang and J. Souquières. Modeling Class Operations in B: Application to UML Behavioural 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.

[14] H. Ledang and J. Souquières. Derivation Schemes from OCL Expressions to B. Technical Report A02-R-042, Laboratoire Lorrain de Recherche en Informatique et ses Applications, May 2002. http://www.loria.fr/∼ledang/publications/oclb.ps.gz.

[15] R. Marcano and N. Lévy. Transformation d'annotations OCL en expressions B. In *Journées AFADL'2001 : Approches Formelles dans l'Assistance au Développement de Logiciels*, Nancy (F), 11-13 juin, 2001.

[16] E. Meyer. *Développements formels par objets: utilisation conjointe de B et d'UML*. PhD thesis, LORIA - Université Nancy 2, Nancy (F), mars 2001.

[17] 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.

[18] 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.

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

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

[3] http://www.ArgoUML.org