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

Hung Ledang, Jeanine Souquières

To cite this version:

HAL Id: inria-00107556
https://inria.hal.science/inria-00107556
Submitted on 19 Oct 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.
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 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 specify 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 $S_2$ ...), pre-conditioning (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 $\mathbb{Z}$, $BOOL$ 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 shortcomings 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 multi-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 Z and INT which correspond to the OCL type Integer. Z is chosen as the formalisation of Integer since Z is more abstract than INT. An integer value nn in OCL is modelled in B as a value nn of Z. As shown in Table 1, all Integer OCL operations but “/” can be expressed by a B expression on 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→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→d+c·b→b·d; details of a such formalisation can be found in [14].

<table>
<thead>
<tr>
<th>Operations OCL</th>
<th>Semantics in B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a==b</td>
<td>a = b</td>
</tr>
<tr>
<td>a&lt;b</td>
<td>¬(a = b)</td>
</tr>
<tr>
<td>a+b</td>
<td>a + b</td>
</tr>
<tr>
<td>a-b</td>
<td>a - b</td>
</tr>
<tr>
<td>a*b</td>
<td>a × b</td>
</tr>
<tr>
<td>a div b</td>
<td>a / b</td>
</tr>
<tr>
<td>a mod b</td>
<td>a mod b</td>
</tr>
<tr>
<td>a&lt;b</td>
<td>a &lt; b</td>
</tr>
<tr>
<td>a&lt;=b</td>
<td>a ≤ b</td>
</tr>
<tr>
<td>a&gt;b</td>
<td>b &lt; a</td>
</tr>
<tr>
<td>a&gt;=b</td>
<td>b ≤ a</td>
</tr>
<tr>
<td>a&lt;=&gt;b</td>
<td>a&lt;&gt;b</td>
</tr>
<tr>
<td>a=0</td>
<td>a=0</td>
</tr>
<tr>
<td>a&lt;&gt;0</td>
<td>a&lt;&gt;0</td>
</tr>
<tr>
<td>aandr b</td>
<td>a∧b</td>
</tr>
<tr>
<td>not a</td>
<td>¬a</td>
</tr>
<tr>
<td>if a then b else c endif</td>
<td>¬(bool(a∧b) ∧ bool(¬a∧c))</td>
</tr>
</tbody>
</table>

**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 a (variable or constant). 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.

<table>
<thead>
<tr>
<th>Operations OCL</th>
<th>Semantics in B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a==b</td>
<td>bool(a) = bool(b)</td>
</tr>
<tr>
<td>a&lt;b</td>
<td>¬(bool(a) = bool(b))</td>
</tr>
<tr>
<td>a or b</td>
<td>a ∨ b</td>
</tr>
<tr>
<td>a xor b</td>
<td>¬(bool(a) = bool(b))</td>
</tr>
<tr>
<td>a and b</td>
<td>a ∧ b</td>
</tr>
<tr>
<td>not a</td>
<td>¬a</td>
</tr>
<tr>
<td>if a then b else c endif</td>
<td>¬(bool(a∧b) ∧ bool(¬a∧c))</td>
</tr>
</tbody>
</table>

**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 \( s \text{tr} \) of \( \text{seq}(0..255) \). The OCL operations on \( \text{String} \) (except two operations \( \text{toUpperCase} \) and \( \text{toLowerCase} \)) can be expressed by B expressions on \( \text{seq}(0..255) \) (cf. Table 3).

**Remark 2** Two operations \( \text{toUpperCase} \) and \( \text{toLowerCase} \) involve a repetitive computation which is very sophisticated such that they cannot be expressed by an expression B at the abstract machine level.

---

**Table 3. Operations on strings**

<table>
<thead>
<tr>
<th>Operations OCL</th>
<th>Semantics in B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a=b )</td>
<td>( a = b )</td>
</tr>
<tr>
<td>( a\leq b )</td>
<td>( \neg(a &gt; b) )</td>
</tr>
<tr>
<td>( \text{size} )</td>
<td>( \text{size}(a) )</td>
</tr>
<tr>
<td>( a\text{concat}(b) )</td>
<td>( a^b )</td>
</tr>
<tr>
<td>( a\text{substring(lower,upper)} )</td>
<td>( (a \uparrow \text{upper}) \downarrow \text{lower} )</td>
</tr>
<tr>
<td>( \text{toUpper} )</td>
<td>no definition</td>
</tr>
<tr>
<td>( \text{toLower} )</td>
<td>no definition</td>
</tr>
</tbody>
</table>

---

**Derivation 4 (Real)** There is no B predefined type for real values, a definitive solution for modelling the OCL type \( \text{Real} \) 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 \( \text{Real} \) can be modelled in B by relation \( \mathbb{Z} \leftrightarrow \mathbb{Z} \). The OCL operations on \( \text{Real} \) are modelled in B in a similar manner of the operations on ratios.

**Derivation 5 (Enumeration types)** Each enumeration type \( \text{Enum} = \{ \text{val}_1, ... , \text{val}_n \} \) is modelled in B by an enumerated set serving as a user-defined type \( \text{Enum} = \{ \text{val}_1, ..., \text{val}_n \} \). Each element \( \text{val}_i \) in \( \text{Enum} \) is modelled by an element \( \text{val}_i \) in \( \text{Enum} \). The modelling in B of operations on an enumeration type is shown in Table 4, where \( a\# \), \( b\# \) are two values of type \( \text{Enum} \) and \( a, b \) denote their B formalisation.

---

**Table 4. Operations on enumerations**

<table>
<thead>
<tr>
<th>Operations OCL</th>
<th>Semantics in B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a# = b# )</td>
<td>( a = b )</td>
</tr>
<tr>
<td>( a# &lt; b# )</td>
<td>( \neg(a &gt; b) )</td>
</tr>
</tbody>
</table>

---

**Derivation 6 (Object types)** According to Meyer and Souquières [17], for each class \( \text{Class} \), the B constant \( \text{CLASS} \) models the possible instance set and the B variable \( \text{class} \), which is defined as a subset of \( \text{CLASS} \), models the effective instance set of Class. Therefore, the object type \( \text{Class} \) is modelled in B as \( \text{CLASS} \), whereas the operation \( \text{Class.allInstances} \) is modelled as \( \text{class} \). An object \( \text{cc} \) of \( \text{Class} \) is modelled in B by an element \( \text{cc} \) of \( \text{class} \).

### 3.2 Collection types

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

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

Using Derivation 7, almost predefined operations on collection types (except \( \text{asSequence} \) on sets or bags and excluding on sequences\(^1\)) can be expressed by B expressions. The semantics of OCL operations \( \text{select} \), \( \text{reject} \), \( \text{collect} \), 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 expressions on collection types)** Given T an OCL type on which the collection types are defined and \( ss : \text{Set}(T) ; bb : \text{Bag}(T) ; se, se_2 : \text{Sequence}(T) ; tt : T ; \text{boolean} \) \( expr \) is a boolean expression on \( T \) and \( expr \) is an expression on \( tt \). Let’s call \( T, ss, bb, se, se_2, tt, \text{boolean} \) \( expr \) and \( expr \) respectively the B formalisation of \( T \) and \( T \) is modelled in B by \( \{ tt | tt e s \& \text{boolean} \} \).

1. the OCL expression \( \text{bb} \rightarrow \text{includes}(tt) \), which checks whether \( tt \) is an element of the bag \( bb \), is modelled in B by \( tt \in \text{dom}(bb) \);
2. the OCL sequence union expression \( \text{se} \rightarrow \text{union}(se_2) \) is modelled in B by the sequence concatenation expression \( se \rightarrow ss_2 \);
3. the OCL expression \( ss \rightarrow \text{select}(tt | \text{boolean} \) \( expr \)\) which extracts elements \( tt \) of \( ss \) satisfying \( \text{boolean} \) \( expr \), is modelled in B by \( \{ tt | tt e ss \& \text{boolean} \} \);
4. the OCL expression \( ss \rightarrow \text{collect}(tt | \text{boolean} \) \( expr \)\), which derives a collection (a bag) of values \( expr \) computed on every element \( tt \) of the set \( ss \), is modelled in B by \( \{ tt, mn | \{ tt e expr(tt[ss] | mn e N \& \text{expr} = \text{card}(\{ xx | xx e ss \& expr(xx) = tt \}) \} \} \).

---

\(^1\)The semantics of the operation \( \text{asSequence} \) on sets or bags has not been defined in OCL therefore we cannot model it in B. It is the same for the operation \( \text{excluding} \) 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 detailed 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∈class 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 assoc between two classes Class and Class2 is modelled by a B variable assoc defined as assoc∈class Class 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∈class Class ; q2∈class Class2 ; ... ; qn∈class Class2 . We add also a B invariant linking assoc and q1,...,qn as follows: (assoc−1⊗q1⊗...⊗qn)−1∈class Class Class2 . As for attributes, the relation defining assoc can be further refined according to additional properties of assoc.

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 attrr{cc}. If the cardinality of attr is equal to 1, cc.attr can be modelled by attrr{cc}; otherwise if attr is ordered, attrr{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 attrr{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.v∈attrr{sc}∧nn∈N∧n = card{attrr{vv}}\}. 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 modelled; otherwise bc.attr denotes a bag of values of attr associated to the bag bc and is modelled by \{vv, nn.v∈attrr{dom(bc)}∧nn∈N∧n = Σ(cc).λ(ii).ii∈dom(seqc)|attrr{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 assoc from the class Class to Class2. The association assoc 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 of Class2 in assoc. The expression cc.roleClass2[v1, ..., vn] is modelled in B by (assoc−1⊗q1⊗...⊗qn)−1(assoc→v1→...→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 assoc is equal to 1, cc.roleClass2[v1,...,vn] can be expressed in B by (assoc−1⊗q1⊗...⊗qn)−1(assoc→v1→...→vn).

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 assoc a binary association class between two classes Class and Class2. 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 assoc, cc, sc, bc and seqc the B formalisation of assoc, cc, sc, bc and seqc according to Derivation 9:

1. the expression cc.assoc, which denotes the instance(s) of assoc associated to cc, is modelled by \{cc\}assoc; if the cardinality of Class2 in assoc is equal to 1, cc.assoc can also be modelled by cc→assoc(ce).
2. the expression sc.assos, which denotes the instances of assos associated to the elements of sc, is modelled by sc<: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<:dom(bc)<:assos\&\&nn \in N\&nn = bc(dom(\{cc\}))\};

4. the expression seqc.assos has only semantics if the cardinality of Class 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(i)(i<:dom(seqc)|seqc(i)\rightarrow assos(seqc(i))).

**Definition 3 (Well-formed postconditions)**

1. Every potential output out is defined by an elementary constraint P_i(out[Input],NewObject), which defines out 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 \( P_1, \ldots, P_n \) 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 \( P_1, \ldots, P_n \) 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 $\text{result} = \text{expr}(\text{Input}[\cup \text{NewObject}])$ to define deterministically the return value of $\text{oper}$. We add in the B operation $\text{oper}$ the following substitution: $\text{out} = \text{expr}(\text{Input}[\cup \text{NewObject}])$, where $\text{out}$ represents the return parameter of $\text{oper}$ and $\text{expr}$ is the B formalisation of $\text{expr}$.

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

the new form of $P_i$ enables us to update $\text{oper}$ in the following manner:
   - we create a clause $\text{any...where}$... if it has not been created (cf. Derivation 18);
   - we declare $\text{res}$, which models $\text{res}$:
     $\text{any...res where}$
     $\text{expr}(\text{res}, \text{Input}[\cup \text{NewObject}])$
   - we add the substitution $\text{out} = \text{res}$ in the body of $\text{any...where}$...

Remark 6

1. The B formalisation of $\text{expr}$ is done using derivation schemes in Section 3; all the possible occurrences of $\text{@pre}$ are omitted without losing the semantics of $\text{@pre}$. To justify this point, let's take an example: in the B assignment statement $x := a$, the value of $a$ is always referred 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 $\text{oper}$.

Derivation 15 (Object creation) Given $P_i$ a constraint specifying that an object $\text{cc}$ of a class $\text{Class}$ is created by $\text{oper}$. We create in $\text{oper}$:
   - a clause $\text{any...where}$... if this clause has not been created (cf. Derivation 18);
   - a temporary variable $\text{cc}$, which models $\text{cc}$:
     $\text{any...cc where}$
     $\text{cc}\in \text{CLASS}\subseteq \text{class}$
   - a B substitution, which models the updating of a set of effective instances of $\text{Class}$ by the object $\text{cc}$, in the body $\text{any...where}$
     $\text{cc}\subseteq \text{class}\cup \{\text{cc}\}$

Derivation 16 (Updating attribute value of an object)

Given an attribute $\text{attr}$ and an object $\text{cc}$ of a class $\text{Class}$:

1. given $P_i$ in the form $\text{cc}.\text{attr}=\text{expr}(\text{Input}[\cup \text{NewObject}])$ to define deterministically the new value of $\text{cc}.\text{attr}$ in which the cardinality of $\text{attr}$ is equal to 1. We model the constraint $P_i$ by the substitution $B$ $\text{attr} := \text{attr}\in\{\text{cc}\}	imes\text{expr}$;

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

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

Remark 7 Derivation 16 can be extended for updating associations.

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

1. given $P_i$ in the form $\text{sc} \longrightarrow \text{forAll}(p\text{.attr}=\text{expr}(\text{Input}[\cup \text{NewObject}])$ to define deterministically the value of attribute $\text{attr}$ of all elements in $\text{sc}$. We model $P_i$ by the following substitution $B$: $\text{attr} := \text{attr}\in\text{sc}\times\text{expr}$;

2. given $P_i$ in the form $\text{sc} \longrightarrow \text{forAll}(p\text{.expr}(p\text{.attr}, \text{Input}[\cup \text{NewObject}]))$ to define non-deterministically the value of attribute $\text{attr}$ of all elements in $\text{sc}$. We model $P_i$ in creating in $\text{oper}$:
   - a clause $\text{any...where}$... if this clause has not been created (cf. Derivation 18);
   - a temporary variable $\text{at}$ defined as:
     $\text{any...at where}$
     $\text{at}\in\text{class}\rightarrow \text{type}\text{Attr}
     \text{dom}(\text{at}) = \text{sc}\wedge
     \forall (\text{tt})(\text{tt}\in\text{ran}(\text{at})\Rightarrow
     \text{expr}(\text{at}, \text{Input}[\cup \text{NewObject}])$)
   - the substitution:
     $\text{attr} := \text{attr}\in\text{eq}\text{at...}||$
     in the body of the clause $\text{any...where}$...

In the two cases above, $\text{attr}$ and $\text{sc}$ are the B formalisation of $\text{attr}$ and $\text{sc}$, $\text{expr}$ is the B formalisation of $\text{expr}$ in which $\text{tt}$ replaces $p\text{.attr}$.
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 “P1 and ... and Pn” an OCL and expression on the elementary constraints P1, ..., Pn 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 \text{cond} then \text{expr}1 else \text{expr}2 endif an expression of postconditions of an operation oper. The B formalisation of this expression gives rise to a clause:

\[
\begin{align*}
\text{if} & \quad \text{cond} \quad \text{then} \quad \text{subst}(\text{expr}1) \quad \text{else} \quad \text{subst}(\text{expr}2) \quad \text{endif,} \quad \text{if} \quad \text{expr}2 \quad \text{is not an expression if} \quad \text{then} \quad \text{... else} \quad \text{... endif or} \\
\text{if} & \quad \text{cond} \quad \text{then} \quad \text{subst}(\text{expr}1) \quad \text{elseif} \quad \text{cond2} \quad \text{then} \quad \text{subst}(\text{expr}2) \quad \text{endif,} \quad \text{if} \quad \text{expr}2 \quad \text{is also in the form} \quad \text{if} \quad \text{cond2} \quad \text{then} \quad \text{expr}21 \quad ...; \\
\end{align*}
\]

where \text{cond} is the B formalisation of \text{cond} and \text{subst}(\text{expr}) denotes the substitutions derived from \text{expr}.

Remark 9 \textit{skip} is the substitution of true.

Derivation 20 (Operation) Given \{\text{pre,post}\} the preconditions and the postconditions in OCL of an operation oper, where \text{post} is defined according to Definition 3. The B operation \text{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 \text{pre} is not true then the precondition in \text{oper} is augmented by predicates derived from \text{pre} in using derivation schemes in Section 3;
- the substitution part of \text{oper} is generated from \text{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 \text{let expressions} as described in Derivation 21.

Derivation 21 (let ... in postconditions) The expression

\[
\begin{align*}
\text{let} \quad v_1 : \text{type}_1 = \text{val}_1, \ldots, v_n : \text{type}_n = \text{val}_n & \quad \text{in} \quad \text{expr} \\
\end{align*}
\]

in postconditions of an operation oper gives rise to a clause

\[
\begin{align*}
\text{Let... in oper:} \\
\text{let} & \quad v_1, \ldots, v_n \quad \text{be} \quad v_1 = \text{val}_1 \land \ldots \land v_n = \text{val}_n \\
\text{in} & \quad \text{subst(expr)} \\
\end{align*}
\]

where \text{v}_1, ..., \text{v}_n, \text{val}_1, ..., \text{val}_n and \text{subst(expr)} denote the B formalisation of \text{v}_1, ..., \text{v}_n, \text{val}_1, ..., \text{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 \textit{Pump::enable}\textit{Pump} in OCL

To illustrate the application of derivation schemes, let’s consider the OCL specification (Figure 2) of the class operation \textit{Pump::enable}\textit{Pump}.
The operation has three arguments: the identifier \( p_{i} \) of the pump to be invoked; the category \( g_{g} \) of the petrol to be distributed and the registration number \( v_{i} \) of the vehicle. The precondition says that the pump with identifier \( p_{i} \) exists. The postcondition is formatted according to Definition 3 and says that in case the pump with identifier \( p_{i} \) 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 \( p_{i} \) but we cannot extract it from the set of effective pumps (denoted by \( \text{Pump.allInstances} \)) 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 \( p_{i} \).

5.2 The operation \text{Pump::enable} in B

Figure 3 represents the B specification of the operation \text{Pump::enable} which is derived from the OCL specification in Figure 2 by applying systematically Deriva-

As an example, let's consider the expression \text{Pump.allInstances}\rightarrow\text{collect}(\text{pumpId})\rightarrow\text{includes}(\text{pi}) in the OCL precondition of \text{Pump::enable}. Analysing its sub-expressions we know that: the expression \text{Pump.allInstances} denotes the set of effective instances of the class \text{Pump}. Applying the operation \text{collect(pumpId)} on set \text{Pump.allInstances} we obtain a bag. The operation \text{includes(pi)} will check whether the identifier \( p_{i} \) is an element of the bag \text{Pump.allInstances}\rightarrow\text{collect}(\text{pumpId}). Let \( T(\text{expr}) \) denotes the B formalisation of the OCL expression \text{expr} and \text{pump_pump_Id} the B variable for the attribute \text{pumpId} of the class \text{Pump}. The transformation process from the precondition \text{Pump.allInstances}\rightarrow\text{collect}(\text{pumpId})\rightarrow\text{includes}(\text{pi}) into B proceeds as follows:

\[
\begin{align*}
T(\text{Pump.allInstances}\rightarrow\text{collect}(\text{pumpId})\rightarrow\text{includes}(\text{pi})) &= (\text{cf. point 1 in Derivation 8}) \\
&= (\text{cf. point 4 in Derivation 8}) \\
&= (\text{cf. Derivation 6})
\end{align*}
\]
Remark 10 In order to avoid the eventual name conflicting in the derivation of attributes into B, we prefix the name of attributes by the the class name. Hence the attribute \texttt{pump\_id} is modelled by the variable \texttt{pump\_pump\_id}. It is similar for constants in the enumeration type; the constant \texttt{PG\_STATUS} is modelled by \texttt{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 \texttt{Pump::enable\_Pump} into a clause \texttt{let...} in the B operation \texttt{pump\_enable\_pump}. The substitution body if... of \texttt{let...} is derived from the body expression if ... of \texttt{let...} using Derivation 19. The B guard condition \texttt{let \( pp = \phi \) \ then \ let \( \text{notEmpty} \)} 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 specificattions for the supplementary class invariants and for UML abstract operations in UML specifications.

For the further work, the prototype ArgoUML+B has been developed 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 specification according the derivation procedure in [13]. In ArgoUML, there is a component “ocl-arg", 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


\(^3\)http://www.ArgoUML.org