Translating the Object Constraint Language into First-order Predicate Logic

Bernhard Beckert, Uwe Keller, and Peter H. Schmitt

Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
Am Fasanengarten 5, D-76128 Karlsruhe
Fax: +49 721 608 4211, Email: {beckert,keller,pschmitt}@ira.uka.de

Abstract. In this paper, we define a translation of UML class diagrams with OCL constraints into first-order predicate logic. The goal is logical reasoning about UML models, realized by an interactive theorem prover. We put an emphasis on usability of the formulas resulting from the translation, and we have developed optimisations and heuristics to enhance the efficiency of the theorem proving process.
The translation has been implemented as part of the KeY system, but our implementation can also be used stand-alone.

1 Introduction

Overview. The Unified Modeling Language (UML) [15] has been widely accepted as the standard object-oriented modelling language and is supported by a great number of CASE tools. The Object Constraint Language (OCL) is an integral part of UML, and was introduced to express subtleties and nuances of meaning that diagrams cannot convey by themselves.

There is by now a great number of papers attributing a rigorous meaning to UML class diagrams (without OCL constraints) by translating them into a language with known semantics, for example: the CASL-LTL language (an extension of CASL) [16], the Z specification language [6] and its extension Object Z [13], the logical language of PVS [14], the Mathematical System Model (MSM) [5], EER diagrams [7], the Maude language [2].

Clarification of the semantics of UML class diagrams, as provided by these papers, is a necessary prerequisite for a rigorous semantics of OCL, as e.g. developed in [8, 9, 17] and in the draft [4]. We believe that the semantics of UML class diagrams with OCL, both the issues of common consent and controversial open issues, are by now understood well enough to serve as a basis for further developments. The translation developed in this paper can be applied to OCL constraints in any UML diagram type. But since the semantical status of OCL constraints in other diagram types, such as state or sequence diagrams, is less clear, we restrict attention for the moment to OCL constraints in class diagrams.

We present in this paper a translation of UML/OCL into first-order predicate logic. Our goal is logical reasoning about UML models. The novel features of our work are that we put an emphasis on usability of the formulas resulting from the translation, and we offer alternatives for the translation of model elements. Where possible, we develop optimisations and heuristics to enhance the efficiency of the theorem proving process. For interactive theorem proving ease of use for the interacting human prover is a central factor for efficiency. Therefore readability of the translated formulas becomes a crucial issue.

The KeY Project. The work reported here is part of the KeY project (see the overview paper [1] or the web page `i12www.ira.uka.de/~key` for more information). The logical language used in this project is Dynamic Logic, a multi-modal extension of first-order predicate logic specially suited to reason about properties of programs. In this present account we restrict attention to translation into first-order logic, which is the crucial part anyhow. The extension of the translation to the OCL constructs that require Dynamic Logic as the target language, e.g. @*pre* and `result` in post-conditions and the `iterate` operation, is rather straightforward and can be found in [12]. An extensive account of how to treat the @*pre* operator in Dynamic Logic is given in [3].

Implementation. We have implemented our translation, including some optimisations and heuristics. The implementation, which is written in JAVA, is part of the KeY system. For those who wish to use the translation in a different context, we have provided a stand-alone version that reads the UML class diagram and the OCL constraints to be translated from an XML file and generates a text file containing the resulting formulas. It uses the XML dialect XMI, which is a standard for the textual representation of UML diagrams. For parsing OCL constraints, we have integrated the parser component of the OCL compiler developed by Hußmann et al. [11].

We tried to keep the implementation flexible and it should be easy to adapt to different needs arising from other application areas, such as a different syntax for the output formulas, new optimisations, and new heuristics for choosing between several possible translations. Also, adaptations to future changes in the UML/OCL standard will not require much effort.

Both the KeY system and the stand-alone version, as well as additional documentation and examples, can be downloaded from `i12www.ira.uka.de/~key`.

To the best of our knowledge, this is the first implementation of such a translation. We hope that it can serve as a means helping to promote the use and application of OCL.

Structure of this Paper. In Section 2, we briefly review the semantical pre-requisites and describe the semantical properties of our translation. The basic translation is presented in Sections 3 and 4, while Section 5 is devoted to possible optimisations that improve the readability and usability of the resulting first-order formulas. Section 6 concludes with an outlook and future extensions.

All examples presented in the following refer to the class diagram shown in Figure 1.

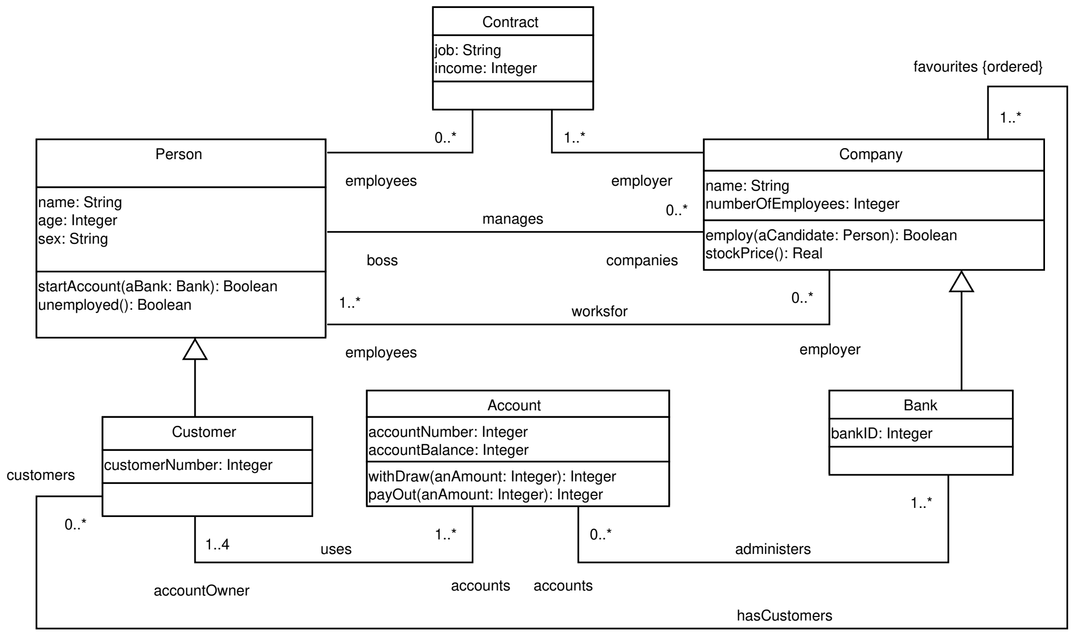

Fig. 1. Example for a UML class diagram.

2 Properties of the Translation

We start with a given UML class diagram $\mathcal{D}$ that is enriched by OCL constraints $C_1, \ldots, C_n$. Together, $\mathcal{D}$ and $C_1, \ldots, C_n$ describe the possible *states* of the system to be modelled. A system state, sometimes also called a *snapshot* in the UML framework, is a complete description of an instance of the modelled system. It details what objects exist (they are instances of the classes in $\mathcal{D}$), gives the values of attributes for the existing objects, and defines which pairs of objects (or more general, n-tuples of objects) are instances of the associations between classes in $\mathcal{D}$. We use first-order structures $\mathcal{S}$ to represent system states.

The vocabulary $\Sigma = \Sigma_{\mathcal{D}}$ of $\mathcal{S}$, i.e., the set of types, function, and relation symbols, is read off from the diagram $\mathcal{D}$. Sometimes there are choices in which symbols to include in $\Sigma_{\mathcal{D}}$: A binary association between classes A and B with multiplicity 1 at the B-end may give rise to the inclusion of a binary relation symbol in $\Sigma_{\mathcal{D}}$ or of a unary function symbol. To have a common platform for comparing these alternatives, we include (in this and similar cases) both symbols in $\Sigma_{\mathcal{D}}$. The definition of $\Sigma_{\mathcal{D}}$ follows shortly.

Of course, not all $\Sigma_{\mathcal{D}}$-structures are valid system states of $\mathcal{D}$. We will say that a structure $\mathcal{S}$ *conforms* to $\mathcal{D}$ in case that $\mathcal{S}$ satisfies the diagram $\mathcal{D}$ and its OCL constraints $C_1, \ldots, C_n$, i.e., it is a possible system state according to the UML/OCL semantics [15].

In the next two sections, we describe how to associate with a UML class diagram $\mathcal{D}$ and OCL constraints $C_1, \ldots, C_n$ formulas $Th_{\mathcal{D}}, Th_{C_1}, \ldots, Th_{C_n}$. Since new symbols are added by the translation, they are formulas over an extended signature $\Sigma^* = \Sigma_{\mathcal{D}} \cup \Sigma_{tr}$. Therefore, the correctness property of our translation reads: For every $\Sigma_{\mathcal{D}}$-structure $\mathcal{S}$,

$$\begin{array}{l} \mathcal{S} \text{ conforms to } \mathcal{D} \text{ with } C_1, \ldots, C_n \quad \text{if and only if} \\ \mathcal{S}^* \models Th_{\mathcal{D}} \wedge Th_{C_1} \wedge \ldots \wedge Th_{C_n} \text{ for every } \Sigma^*\text{-extension } \mathcal{S}^* \text{ of } \mathcal{S}. \end{array}$$

A detailed analysis of the correctness property of such a translation can be found in [12].

Our translation does not handle meta level features—with the exception of `OCLAny` and `allInstances`. This is due to that fact, that the future role of the meta level is unclear. In version 2.0 of the UML standard, now under discussion, it may undergo substantial changes or be eliminated alltogether.

3 Translating the Class Diagram

3.1 Extracting the Signature from the Class Diagram

In the following, we summarise how the first-order signature $\Sigma_{\mathcal{D}}$ is extracted from a class diagram $\mathcal{D}$. A more extensive account may be found in [18]. The set of *types* of $\Sigma_{\mathcal{D}}$ contains:

1. A type for every class in $\mathcal{D}$. Types will be denoted by the same names as the corresponding class, starting with an upper-case letter.
2. The types *Integer*, *Real*, *Boolean*, *String*.
3. If T is a type, then $Collection_T$, Set_T, Bag_T, $Sequence_T$ are types. Types of this form are called *collection types*. These collection types are only generated when T is not itself a collection type, i.e., no nesting of the collection type operators is allowed.
4. $\Sigma_{\mathcal{D}}$ will furthermore contain the type *Any*, which serves as the translation of the OCL type *OCLAny*.

The subtype relation $S_1 <_{\mathcal{D}} S_2$ is defined as in [19]. For each type T there will be an infinite supply of variables $x{:}T, y{:}T, x_i{:}T$ of type T. The set of *functions* and *relations* in $\Sigma_{\mathcal{D}}$ contains:

1. For every binary association r in $\mathcal{D}$ with association ends e_0, e_1 there are two functions in $\Sigma_{\mathcal{D}}$, which will be referred to by the role name r_i at the association end e_i $(i = 0, 1)$. If no role name is given, the name of the class attached to e_i will be used. Function names start with a lower-case letter. If e_i is attached to class S_i, then the function is of signature $r_i\colon S_{1-i} \to Set_{S_i}$. In case the multiplicity at the end e_i is 1, the signature is $r_i\colon S_{1-i} \to S_i$. If the e_i-end has the stereotype $\ll$*ordered*$\gg$, then it is $r_i\colon S_{1-i} \to Sequence_{S_i}$. For n-ary relations we proceed correspondingly.
2. For every n-ary association r in $\mathcal{D}$ there is, in addition, an n-ary *predicate* in $\Sigma_{\mathcal{D}}$.

3. For every *attribute* a of a class S in $\mathcal{D}$ there is a function in $\Sigma_{\mathcal{D}}$ that is referred to by the name of the attribute and has signature $a\colon S \to S_r$, where S_r is the value type of attribute a as specified in $\mathcal{D}$. If a is a *class* attribute (sometimes this is also called a static attribute), then a constant of type S_r is added to $\Sigma_{\mathcal{D}}$. The concrete syntax of this constant is $S.a$.
4. For every *operation* c of a class S with parameters of type $S_1, \ldots, S_k$ and result type S' there is a function $f_c : S \times S_1 \times \ldots \times S_k \to S'$ in $\Sigma_{\mathcal{D}}$. In accordance with the OCL specification [15] we require that c has no side effects, i.e., it satisfies the property `isQuery()`.
5. For every *association class* C attached to an association r, where r associates the classes S_1 and S_2, there are unary projection functions $s_1\colon C \to S_1$ and $s_2 : C \to S_2$ in $\Sigma_{\mathcal{D}}$.
6. All properties of the pre-defined OCL types, as detailed in the standard [15], are functions or relations in $\Sigma_{\mathcal{D}}$.
7. The symbol $\doteq$ will be used to denote equality. By overloading we use the same symbol for all types.

3.2 Extracting Formulas from the Class Diagram

The translation $Th_{\mathcal{D}} = (\bigwedge Ax_{ADT} \wedge \bigwedge Ax_{\mathcal{D}} \wedge \bigwedge Constr_{\mathcal{D}})$ of a class diagram $\mathcal{D}$ consists of three parts: Ax_{ADT} is actually independent of $\mathcal{D}$. It contains the axioms of the Abstract Data Types (ADTs) that are used to represent the built-in data types of OCL (*Integer*, *Boolean*, etc.), and the axioms for the ADTs Set_T, Bag_T, etc. that are used to represent the corresponding collection types of OCL.

The second part $Ax_{\mathcal{D}}$ is a set of axioms that depend on $\mathcal{D}$ but that do not express intrinsic information of $\mathcal{D}$. They deal with inter-dependencies among the function and relation symbols extracted from $\mathcal{D}$ that reflect, for example, the symmetry of associations in UML.

Example 1. Consider the association *worksfor* between the classes *Person* and *Company* (Figure 1). The signature $\Sigma_{\mathcal{D}}$ contains two function symbols and a relation symbol representing this association: *employer* with argument type *Person* and value type $Set_{Company}$, *employees* with argument type *Company* and value type Set_{Person}, and the binary relation symbol *worksfor* with first argument of type *Person* and second argument of type *Company*.

To restrict the interpretation of these symbols appropriately, the set $Ax_{\mathcal{D}}$ contains the following axioms:

$$\begin{array}{l}
\forall p{:}Person\, \forall c{:}Company\, (c \in employer(p) \leftrightarrow p \in employees(c)) \\
\forall p{:}Person\, \forall c{:}Company\, (c \in employer(p) \leftrightarrow worksfor(p, c)) \\
\forall p{:}Person\, \forall c{:}Company\, (p \in employees(c) \leftrightarrow worksfor(p, c))
\end{array}$$

The third part $Constr_{\mathcal{D}}$ of $Th_{\mathcal{D}}$ contains formulas representing the restrictions on system states expressed graphically in $\mathcal{D}$, e.g. multiplicity constraints, subtyping restrictions, and others. We require in $Constr_{\mathcal{D}}$ also that an abstract class is the union of its concrete subclasses and that enumerations are sets containing exactly their enumeration literals as elements. A detailed account of this topic is given in [12]. Instead of giving a formal definition of $Constr_{\mathcal{D}}$, we present a typical example:

Example 2. Consider the association *uses* between the classes *Customer* and *Account*. The set $Constr_{\mathcal{D}}$ contains the following formulas expressing the multiplicity constraints attached to *uses*:

$$\begin{array}{l}
\forall c{:}Customer\, (size(accounts(c)) \geq 1) \\
\forall a{:}Account\, (1 \leq size(accountOwner(a)) \wedge size(accountOwner(a)) \leq 4)
\end{array}$$

4 Translating the OCL Constraints

4.1 Overview

OCL constraints consist of an OCL expression of type *Boolean* and some declaration connecting the OCL expression to an item in the class diagram. In the case of pre- and post-conditions, the constraint is bound to an operation; invariants are bound to a class. The translation procedure for OCL constraints, therefore, cannot process OCL expressions as *isolated* entities but also has to take into account the diagram and the information it contains.

In our basic translation described below, OCL expressions in most cases are translated into a first-order *term* of the appropriate Abstract Data Type (ADT). The only exceptions are expressions of OCL type *Boolean*, which are usually transformed into first-order *formulas*. The first-order term resp. formula that is the result of translating an OCL expression *exp* is denoted by $\lceil exp \rceil$.

The translation procedure works by structural recursion on the expressions. When certain OCL features are translated (as described in the following subsections), new function or predicate symbols are introduced (they are elements of the extended signature Σ^*) as well as axioms that constrain the interpretation of the introduced symbols according to the semantics of UML/OCL.

Note, that OCL allows a modeller to use some shorthand notations. We assume that constraints have been normalised to their (longer) standard form before they are translated (in our implementation we use a normalisation provided by Hußmann et al.'s OCL compiler [11]).

The set Ax_{exp} generated during the translation of an expression *exp* includes all those axioms that are generated by the recursive translation of subexpressions of *exp*—besides the axioms that stem from the translation of the "top-level" OCL feature of *exp*.

The translation of OCL *expressions* is extended to OCL *constraints* as follows. Let I be an OCL invariant of form "`context` C `inv:` b", where C is a class in the diagram $\mathcal{D}$ and b is an OCL expression of type *Boolean*. The invariant states that, for *every* instance *self* of C existing in a system state, the property described by b holds. Accordingly, the translation Th_I of the invariant I is:

$$\bigwedge Ax_b \to \forall self{:}C \, \lceil b \rceil \ .$$

Pre- and post-conditions can be translated in a similar way; only the *@pre* operator, which may occur in post-conditions, requires a special treatment (see [3] and [12] for a detailed account).

4.2 Translating Built-in OCL Types

Translating Boolean Expressions. As said above, we usually translate OCL expressions of type *Boolean* into first-order formulas. The boolean operators `and`, `or`, `implies`, `not` are translated into the corresponding first-order operators. Equality of *Boolean* expressions is represented by the operator $\leftrightarrow$.

Translating Integer, Real, String Expressions. The OCL type *Integer* corresponds to an ADT *Integer*. As said above, the signature $\Sigma_{\mathcal{D}}$ contains a symbol for every feature[1] of *Integer*. Every feature of *Integer* is translated into the corresponding function or predicate symbol of the ADT. In the same way, the OCL types *Real* and *String* are handled with the help of ADTs *Real* and *String*.

Hußmann et al. [10] have argued convincingly that the encapsulation concepts of ADTs and UML classes are very different and that UML classes can, as a consequence, not smoothly be translated into ADTs. However, their analysis applies primarily to user defined classes and does not affect the translation of the basic OCL types just mentioned.

Example 3. Given the following OCL expression (with respect to class *Person*)

```
self.age >= 0 and self.employer->size >= 1
```

that states that person *self* is employed and has a non-negative age the translation results in the formula $age(self) \geq 0 \wedge size(employer(self)) \geq 1$.

[1] According to OCL terminology a *feature* of some OCL type T is any operation that can be applied to instances of T.

4.3 Translating the allInstances Operator

The OCL operator `allInstances` can be applied to a class (to be more precise it is applied to the object of type *OclType* that corresponds to the class in the diagram $\mathcal{D}$). It returns the set of all instances of that class in the current state. To translate this operator, we introduce a new symbol $allInstances_C$ for each class C and define $\lceil C.\texttt{allInstances}\rceil = allInstances_C$. The additional axiom $\forall o{:}C\,(o \in allInstances_C)$ is introduced to specify the meaning of the new constant.

4.4 Translating Collection Operators

Overview. OCL offers a common super-type $Collection(T)$ for the collection types $Set(T)$, $Bag(T)$, and $Sequence(T)$. Since OCL defines this super-type to be abstract, it does not occur in actual OCL constraints, but is used to define features that all collection types have in common (e.g., the `size` operator). Consequently, we provide ADTs to represent sets, bags, sequences and collections of all occurring types (e.g., Set_{Bank}, Bag_{Bank}, $Sequence_{Bank}$, $Collection_{Bank}$), where the ADTs $Collection_T$ are only relevant for the purpose of typing in pathological borderline situations and play no role for modelling with OCL in practice.

Below, we describe the translation of the features that the collection types have in common.

Translating `size`, `count`, `sum`, `includes`, `append`, *etc.* These features are translated into the functions that are their direct counterparts in the ADTs Set_T, Bag_T, and $Sequence_T$. For example, $\lceil c\texttt{->size}\rceil = size(\lceil c\rceil)$ and $\lceil s\texttt{->union(}c\texttt{)}\rceil = union(\lceil s\rceil, \lceil c\rceil)$.

Translating Equality. We translate the equality $s_1\texttt{=}s_2$ of sets s_1, s_2 of OCL type $Set(T)$ by expressing that they have the same elements: $\lceil s_1\texttt{=}s_2\rceil = \forall e{:}T\,(e \in \lceil s_1\rceil \leftrightarrow e \in \lceil s_2\rceil)$. Here and in all similar situations below, $e{:}T$ is a new variable that has not been used before.

For bags we get the formula $\lceil b_1\texttt{=}b_2\rceil = \forall e{:}T\,(count(\lceil b_1\rceil, e) \doteq count(\lceil b_2\rceil, e))$, and for sequences a similar translation is generated.

Translating `includesAll`, `excludesAll`. $E = c_1\texttt{->includesAll(}c_2\texttt{)}$ expresses that the collection c_2 is a subset of c_1. Thus, $\lceil E\rceil = \forall e{:}T\,(e \in \lceil c_2\rceil \rightarrow e \in \lceil c_1\rceil)$. `excludesAll` expresses that no element of c_2 is an element of c_1 and is treated similarly.

Translating `notEmpty`, `isEmpty`. The translation of the expression $E = c\texttt{->notEmpty}$ is the formula $\lceil E\rceil = \exists e{:}T\,(e \in \lceil c\rceil)$. `isEmpty` is treated as the negation of `notEmpty`.

Translating `forAll`, `exists`. The meaning of $E_1 = c\texttt{->forAll(e|}\ b\texttt{)}$ is that b evaluates to `true` for all possible instantiations of `e` with elements of the collection c. Thus, the translation of E_1 is $\lceil E_1\rceil = \forall e{:}T\,((e \in \lceil c\rceil) \rightarrow \lceil b\rceil)$. To translate $E_2 = c\texttt{->exists(e|}\ b\texttt{)}$ we use $\lceil E_2\rceil = \exists e{:}T\,((e \in \lceil c\rceil) \wedge \lceil b\rceil)$.

Example 4. Consider the following OCL expression, which formalises "For different objects of class *Bank*, the attribute *bankID* has different values."

```
Bank.allInstances->forAll(b1,b2 |
        not (b1 = b2)  implies  not (b1.bankID = b2.bankID))
```

Its translation is the following formula (a much shorter and optimised translation is given in Section 5.3):

<u>Translation:</u>

$$\forall b_1{:}Bank\,(b_1 \in allInstances_{Bank} \rightarrow \forall b_2{:}Bank\,(b_2 \in allInstances_{Bank} \rightarrow (\neg(b_1 \doteq b_2) \rightarrow \neg(bankID(b_1) \doteq bankID(b_2)))))$$

<u>Additional axiom:</u>

$$\forall b{:}Bank\,(b \in allInstances_{Bank})$$

Translating `isUnique`. The meaning of $E = c$`->isUnique(e|`exp`)` is that the evaluation of exp results in a different value for each instantiation of `e` with elements of c. So,

$$\begin{aligned}\lceil E \rceil \;=\; & \forall e_1{:}T\,\forall e_2{:}T\,((e_1 \in \lceil c \rceil \wedge e_2 \in \lceil c \rceil \wedge \\ & \lceil exp \rceil\{e/e_1\} \doteq \lceil exp \rceil\{e/e_2\}) \;\rightarrow\; e_1 \doteq e_2)\ .^2\end{aligned}$$

where $e_1{:}T, e_2{:}T$ are two distinct new variables.

Translating `sortedBy`. The value of $E = c$`->sortedBy(e|`exp`)` is a sequence with (a) the same elements as collection c, which are (b) ordered according to the values of the expression exp (this only makes sense if there is some order $\leq$ defined on the OCL type of exp. To translate E, we introduce a new function symbol $sortedBy_E$. Let $p_1, \ldots, p_n$ be the free variables occurring in the translations $\lceil c \rceil$ and $\lceil exp \rceil$ of the subexpressions—excluding the variable `e`. Then, the translation of E is $\lceil E \rceil = sortedBy_E(p_1, \ldots, p_n)$. To ensure that $sortedBy_E$ has the desired interpretation with properties (a) and (b), the following two axioms are added to Ax_E:

$$\forall p_1{:}T_1 \ldots \forall p_n{:}T_n \forall e'{:}T\,(count(\lceil \mathtt{c} \rceil, e') \doteq count(sortedBy_E(p_1, \ldots, p_n), e'))$$

$$\begin{aligned}&\forall p_1{:}T_1 \ldots \forall p_n{:}T_n \forall i{:}Integer, j{:}Integer\,(\\ &\quad (1 \leq i \wedge i \leq j \wedge j \leq size(sortedBy_E(p_1, \ldots, p_n))) \rightarrow \\ &\qquad \lceil \mathtt{exp} \rceil\{e/at(sortedBy_E(p_1, \ldots, p_n), i)\} \\ &\qquad \leq \lceil \mathtt{exp} \rceil\{e/at(sortedBy_E(p_1, \ldots, p_n), j)\})\end{aligned}$$

where $e'{:}T$ and $i{:}Integer, j{:}Integer$ are distinct new variables.

Translating `select`, `reject`. The expression $E = c$`->select(e|`b`)` denotes the collection consisting of those elements of c for which b evaluates to `true` when `e` is instantiated with the element. The translation is based on introducing a new function symbol $select_E$. Let $p_1, \ldots, p_n$ be the free variables occurring in the translation $\lceil b \rceil$ of the condition b excluding `e`. Then, the translation of E is $\lceil E \rceil = select_E(\lceil \mathtt{c} \rceil, p_1, \ldots, p_n)$.[3] Three axioms are added to specify the meaning of $select_E$. Their form depends on whether c is a set, a bag, or a sequence. Here, we present the axioms for sets (the axioms for the other types are similar):

$$\forall p_1{:}T_1 \ldots \forall p_n{:}T_n\, select_E(emptySet_T, p_1, \ldots, p_n) \doteq emptySet_T$$

$$\begin{aligned}&\forall p_1{:}T_1 \ldots \forall p_n{:}T_n \forall s{:}Set_T \forall e{:}T\,(\\ &\quad \lceil b \rceil \rightarrow select_E(insert(s, e), p_1, \ldots, p_n) \doteq insert(select_E(s, p_1, \ldots, p_n), e))\end{aligned}$$

$$\begin{aligned}&\forall p_1{:}T_1 \ldots \forall p_n{:}T_n \forall s{:}Set_T \forall e{:}T\,(\\ &\quad \neg\lceil b \rceil \rightarrow select_E(insert(s, e), p_1, \ldots, p_n) \doteq select_E(s, p_1, \ldots, p_n))\end{aligned}$$

Since the `reject` operator is just the opposite of `select`, we treat it by negating the filter condition b and then applying the above translation.

Example 5. The following OCL expression E formalises "There is no person who works for both company 'BankA' and company 'BankB'."

```
Person.allInstances->select(p| p.employer->exists(c1,c2 |
         c1.name = 'BankA' and c2.name = 'BankB'))->isEmpty
```

[2] The notation $t\{e/s\}$ denotes the result of syntactically replacing all occurrences of the variable e in the term t by the term s.

[3] In [9], a similar abbreviation technique is used for the translation of the `select` operator. There, however, the free variables $p_1, \ldots, p_n$ are not made arguments of the new function, which leads to incorrect results.

Its translation is the following formula (a much shorter and optimised translation is given in Section 5.3):

Translation:

$$\forall p{:}Person\,(\neg(p \in select_E(allInstances_{Person})))$$

Additional axioms:

$$\forall p{:}Person\,(p \in allInstances_{Person})$$
$$select_E(emptySet_{Person}) \doteq emptySet_{Person}$$
$$\begin{aligned}\forall s{:}Set_{Person}\,\forall p{:}Person\,(&\exists c_1{:}Company\,(c_1 \in employer(p) \wedge\\ &\exists c_2{:}Company\,(c_2 \in employer(p) \wedge\\ &name(c_1) \doteq \text{`}BankA\text{`} \wedge name(c_2) \doteq \text{`}BankB\text{`})) \rightarrow\\ &select_E(insert(s,p)) \doteq insert(select_E(s),p))\end{aligned}$$
$$\begin{aligned}\forall s{:}Set_{Person}\,\forall p{:}Person\,(&\neg(\exists c_1{:}Company\,(c_1 \in employer(p) \wedge\\ &\exists c_2{:}Company\,(c_2 \in employer(p) \wedge\\ &name(c_1) \doteq \text{`}BankA\text{`} \wedge name(c_2) \doteq \text{`}BankB\text{`}))) \rightarrow\\ &select_E(insert(s,p)) \doteq select_E(s))\end{aligned}$$

4.5 Translating Other Constructs of OCL

Translating `oclIsKindOf`, `oclIsTypeOf`. These operators allow to check which type the value of an expression *exp* has. The translation of *exp*`.oclIsKindOf(`T`)` is $\exists e{:}T\, e \doteq \lceil exp \rceil$. The operator `oclIsTypeOf` can be expressed by `oclIsKindOf` using the subtype relation extracted from the diagram $\mathcal{D}$.

Translating `oclAsType`. To translate the cast operator `oclAsType`, we introduce a new function symbol $oclAsType_{T_1,T_1} : T_1 \rightarrow T_2$ for every pair T_1, T_2 where T_2 is a subtype of T_1, and we define $\lceil o\texttt{.oclAsType(}T_2\texttt{)}\rceil = oclAsType_{T_1,T_2}(\lceil o \rceil)$ (where o is of type T_1). The additional axioms specifying these symbols are of the form $\forall x{:}T_2\,(oclAsType_{T_1,T_2}(x) \doteq x)$.

Translating Variables and Literals. The translation of an OCL variable `v`, including `self`, is a first-order variable with the same name, i.e., $\lceil \texttt{v} \rceil = v$.

OCL literals of type *Boolean*, *Integer*, *Real*, or *String* are translated into a term over the corresponding ADT. To translate literals of collection types, in case they enumerate the elements of a collection, we construct a term over the ADT Set_T (resp. Bag_T or $Sequence_T$). For example,

$$\lceil \texttt{Set\{1,2,3\}} \rceil = insert(insert(insert(emptySet_{Integer}, 1), 2), 3)\ .$$

To translate collection literals that specify a range of elements, such as $E = \texttt{Set\{}e_1\texttt{..}e_2\texttt{\}}$, we introduce a new function symbol set_E and define $\lceil E \rceil = set_E(p_1, \dots, p_n)$ (where $p_1, \dots, p_n$ are the free variables occurring in the translations of the bounds e_1 and e_2). The additional axiom specifying set_E is

$$\forall p_1{:}T_1 \dots \forall p_n{:}T_n\,\forall i{:}T\,(i \in set_E(p_1, \dots, p_n) \leftrightarrow (\lceil e_1 \rceil \leq i \wedge i \leq \lceil e_2 \rceil))\ .$$

For bags and sequences, the translation is similar. However, additional axioms are needed to express that (a) every element in the range occurs exactly once in the result and (b) for sequences, that the elements are ordered.

Example 6. The following OCL expression (used as an invariant for class *Customer*) formalises "A customer's favourite companies are ordered according to their stock price."

```
Sequence {1 .. self.favourites->size}->forAll(i,j| j >= i   implies
   self.favourites->at(i).stockPrice()  >=
   self.favourites->at(j).stockPrice())
```

Its translation is the following formula (a much shorter and optimised translation is given in Section 5.3):

Translation:

$$\forall i{:}Integer\,(i \in seq_0(self) \rightarrow \forall j{:}Integer\,(j \in seq_0(self) \rightarrow (j \geq i \rightarrow$$
$$stockPrice(at(favourites(self), i)) \geq$$
$$stockPrice(at(favourites(self), j)))))$$

Additional axioms:

$$\forall c{:}Customer\,\forall i{:}Integer\,(i \in seq_0(c) \leftrightarrow$$
$$1 \leq i \wedge i \leq size(favourites(c)))$$
$$\forall c{:}Customer\,\forall i{:}Integer, j{:}Integer\,(1 \leq i \wedge i \leq j \wedge j \leq size(seq_0(c)) \rightarrow$$
$$at(seq_0(c), i) \leq at(seq_0(c), j))$$
$$\forall c{:}Customer\,\forall i{:}Integer\,(count(seq_0(c), i) \leq 1)$$

5 Optimisations and Simplifications

5.1 Motivation

It is crucial for the usability of the formulas generated by the translation (in particular in *interactive* theorem proving)—and for the usefulness of such a translation itself—that the formulas are as easy to understand as the original OCL expressions. One way for achieving this goal is to generate a formula that is syntactically as close as possible to the translated OCL expression. For example, the names of the function symbols used in a formula should as far as possible coincide with the names of the corresponding features in OCL. We tried to satisfy this demand with our basic translation described in Section 4.

But, although the generated formulas are very similar to the original expression in their syntactic structure, they are sometimes unnecessarily complicated and hard to read. This is due to the additional axioms introduced in order to constrain the interpretation of new function symbols, which mainly represent OCL collections. Even for small OCL expression there can be a large number of constraining axioms.

A technique that aims to overcome this problem is presented in this section. The idea is to use a different representation for OCL collections. That can help to reduce the number of additional function symbols and axioms, because most of them are introduced when collection operators are translated (such as `select` and `asSequence`).

Note, that often the readability of formulas can be improved by applying rewriting rules. But there are also many cases where the effects of an unsuitable translation cannot be undone by mere simplification but where the form of the original constraint has to be known to choose a good first-order representation; such choices, consequently, have to be made at the time and as part of translation.

5.2 Representing Collections with Predicates

The translation described in Section 4 uses a functional representation of OCL collections, i.e., expressions of type *Set*, *Sequence*, or *Bag* are translated into first-order terms. An alternative is to translate such expressions predicatively, i.e., to represent them by a formula.

For *sets* a predicative translation is easy to define: A set expression s is translated into a "characteristic" formula $\phi_s(e)$ that is equivalent to $e \in \lceil s \rceil$. Using such a presentation allows us to translate most OCL set operators without the need to introduce new function symbols. For example, the expression $E = s$`->select(e|`b`)` can then be represented by $\phi_E = \phi_s(e) \wedge \lceil b \rceil$.

Unfortunately, there are also cases where a predicative representation is not useful. For example, when an expression of the form s`->size` is translated, it is better to apply a functional translation to the subexpression s. Also, for bags and sequences, a useful predicative representation is more difficult to define than for sets, and the resulting formulas are often hard to understand.

Our investigation of examples showed however, that a predicative translation of *sets* is preferable in most cases. Moreover, in many OCL constraints, expressions of type *bag* or *sequence* are actually used as sets, i.e., the additional information they contain is irrelevant. For example, when an expression $E = s$`->forAll(e|`b`)` is translated, the order of elements in s and the number of their occurrences is of no importance, and E can be translated predicatively. This basic idea gives rise to a simple but powerful heuristics to decide whether a predicative translation is preferable to a functional form. Usually such a decision has to be made globally for a whole expression, because combining the translations of subexpressions that use different representations (functional resp. predicative) is awkward and leads to formulas that are hard to read. For a detailed account of this topic please refer to [12].

5.3 Examples for Predicative Translations

In this section, we present the predicative translations of the OCL expression from the examples in Section 4. They are shorter and easier to read than the functional translations. Moreover, it is not necessary anymore to generate additional axioms since no new symbols are introduced.

Example 7. The predicative translation of the expression from Example 4 is:

$$\forall b_1{:}Bank\ \forall b_2{:}Bank\ (\neg(b_1 \doteq b_2) \to \neg(bankID(b_1) \doteq bankID(b_2)))$$

Example 8. The OCL expression from Example 5 translates to:

$$\begin{array}{l} \forall p{:}Person\,(\neg\,(\exists c_1{:}Company\,(c_1 \in employer(p) \wedge \\ \qquad\qquad \exists c_2{:}Company\,(c_2 \in employer(p) \wedge \\ \qquad\qquad name(c_1) \doteq \text{`}BankA\text{`} \wedge name(c_2) \doteq \text{`}BankB\text{`})))) \end{array}$$

Example 9. The predicative translation of the expression from Example 6 is:

$$\begin{array}{l} \forall j{:}Integer\,(1 \leq j \wedge j \leq size(favourites(self)) \to \\ \qquad \forall i{:}Integer\,(1 \leq i \wedge i \leq size(favourites(self)) \to \\ \qquad\quad (j \geq i \to \\ \qquad\quad stockPrice(at(favourites(self), i)) \geq \\ \qquad\quad stockPrice(at(favourites(self), j))))) \end{array}$$

6 Conclusions and Future Work

We have presented in this paper a translation of the logical information contained in UML class diagrams and OCL constraints into first-order predicate logic. It has been implemented as part of the KeY system. It should be easy to use it with other systems, since first-order logic by its universal nature can be readily mapped into almost all logical languages used in formal methods.

We have provided a first set of optimisations. Experimenting with case studies will give insight if and which further optimisations are necessary or desirable.

In the present account, we have deliberately excluded some items, e.g. the `iterate` operator, that are better expressed in a higher-order logic. These issues are treated in [12].

It also remains future research to compare and possibly adapt our translation to version 2.0 of UML/OCL standard once it is agreed upon.

References

1. W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I. P. de Guzman, G. Brewka, and L. M. Pereira, editors, *Proceedings, Logics in Artificial Intelligence (JELIA), Malaga, Spain*, LNCS 1919. Springer, 2000.

2. A. T. Álvarez and J. L. F. Alemán. Formally modeling UML and its evolution: A holistic approach. In S. Smith and C. Talcott, editors, *Proceedings, Formal Methods for Open Object-based Distributed Systems, Stanford, USA*, pages 183–206. Kluwer, 2000.
3. T. Baar, B. Beckert, and P. 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. Springer, 2001.
4. Boldsoft, Rational Software Co., and IONA. Response to the UML 2.0 OCL RfP. Initial submission, August 2001.
5. R. Breu, R. Gosu, F. Huber, B. Rumpe, and W. Schwerin. Towards a precise semantics for object-oriented modeling techniques. In J. Bosch and S. Mitchell, editors, *ECOOP Workshop, Jyväskylä, Finland*, LNCS 1357, pages 205–210. Springer, 1998.
6. R. France. A problem-oriented analysis of basic UML static modeling concepts. In *Proceedings, Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Denver, USA*, volume 34 (10) of *ACM SIGPLAN notices*. ACM Press, 1999.
7. M. Gogolla and M. Richters. On constraints and queries in UML. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language: Technical Aspects and Applications*, pages 109–121. Physica-Verlag, 1998.
8. A. Hamie, F. Civello, J. Howse, S. Kent, and M. Mitchell. Reflections on the Object Constraint Language. In *Post Workshop Proceedings of UML98*. Springer, 1998.
9. A. Hamie, J. Howse, and S. Kent. Interpreting the Object Constraint Language. In *Proceedings, Asia Pacific Conference in Software Engineering*. IEEE Press, July 1998.
10. H. Hußmann, M. Cerioli, G. Reggio, and F. Tort. Abstract data types and UML models. Technical Report DISI-TR-99-15, DISI – Università di Genova, Italy, 1999.
11. H. Hußmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In A. Evans, S. Kent, and B. Selic, editors, *Proceedings, International Conference on the Unified Modeling Language (UML), York, UK*, LNCS 1939, pages 278–293. Springer, 2000.
12. U. Keller. Übersetzung von OCL-Constraints in Formeln einer Dynamischen Logik für Java. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, 2002. In German.
13. S.-K. Kim and D. Carrington. Formalizing the UML class diagram using Object-Z. In R. France and B. Rumpe, editors, *Proceedings, Unified Modeling Language, Fort Collins, USA*, LNCS 1723, pages 83–98. Springer, 1999.
14. P. Krishnan. Consistency checks for UML. In *Proceedings, Asia Pacific Software Engineering Conference (APSEC)*, pages 162–169, 2000.
15. Object Management Group, Inc., Framingham/MA, USA, www.omg.org. *OMG Unified Modeling Language Specification, Version 1.3*, June 1999.
16. G. Reggio, M. Cerioli, and E. Astesiano. An algebraic semantics of UML supporting its multiview approach. In D. Heylen, A. Nijholt, and G. Scollo, editors, *Proceedings, AMiLP 2000*, number 16 in Twente Workshop on Language Technology. University of Twente, 2000.
17. M. Richters and M. Gogolla. On formalizing the UML object constraint language OCL. In *Proceedings, Conceptual Modeling*, LNCS 1507, pages 449–464. Springer, 1998.
18. P. H. Schmitt. A model theoretic semantics of OCL. In B. Beckert, R. France, R. Hähnle, and B. Jacobs, editors, *Proceedings, IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development, Siena, Italy*, pages 43–57. Technical Report DII 07/01, Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Siena, 2001.
19. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, 1999.