



Universiteit
Leiden
The Netherlands

Verifying OCL specifications of UML models : tool support and compositionality

Kyas, M.

Citation

Kyas, M. (2006, April 4). *Verifying OCL specifications of UML models : tool support and compositionality*. Lehmanns Media. Retrieved from <https://hdl.handle.net/1887/4362>

Version: Corrected Publisher's Version

License: [Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden](#)

Downloaded from: <https://hdl.handle.net/1887/4362>

Note: To cite this publication please use the final published version (if applicable).

Chapter 2

Introduction to Models of OCL and UML

In this chapter we present the main notations and concepts used in this thesis and introduce the relevant parts of the Unified Modelling Language (UML) [111, 110] and the Object Constraint Language (OCL) [113] used in this thesis. We explain the notions of class diagrams, object diagrams, object constraint language, and define a formal semantics for all these concepts, which are used in later chapters.

In this chapter we assume a non-empty and enumerable infinite set $\mathcal{N}$ of names, which is partitioned into a set $\mathcal{C}$ of class names, $\mathcal{A}$ of attribute names, $\mathcal{M}$ of operation names and $\mathcal{E}$ of association end names. All these notions will be explained in the next section.

2.1 Class Diagrams

The most well-known language of UML is the language of *class diagrams* which describes the structure of a system in terms of classifiers, their interfaces, and the relations between classifiers.

A *classifier* describes a set of instances or values that have features, for example, attributes and operations, in common. *Classes*, *interfaces*, and *data types* are classifiers. A classifier can be generalised¹ by other classifiers.

Example 2.1. The class *OclAny*, described in Section 2.3.4, is a class that generalises every other class in every model. *Integer* and *Real* are data types, where *Real* generalises *Integer*. However, *Integer* and *Real* are not classes, because you cannot create new instances of them. ♦

Specifically, the different classifiers used in this thesis, are:

Class A *class* defines structural features, for example, attributes, and behavioural features, for example, operations. Moreover, classes are required to provide *methods*, that is, implementations, for each operation they define. Classes may be instantiated.

¹Generalisation is the inverse of inheritance, see Section 2.1.1.

Interface An *interface* defines structural and behavioural features like classes, but methods must not be defined for the operations declared in an interface, but often *constraints* on operations are defined in terms of pre- and postconditions for the operations. Interfaces cannot be instantiated.

Abstract Classes An *abstract classes* is like a class, in that it specifies structural and behavioural features, and it is like an interface, in that it need not provide methods for all behavioural features it specifies. Consequently, abstract classes cannot be instantiated. The crucial difference between an interface and an abstract class is that the abstract class may provide a method, whereas an interface must not.

Data type A *data type* is a type whose instances are only identified by their value. In contrast, instances of a class are identified by their unique name. The most prominent data types in UML are the *primitive data types*, for example, *Boolean*, *Integer*, or *Set*.

By applying the stereotype «active» to a class, the class is declared *active*. Such an *active class* is a class where each instance of this class has its own thread of control. Classes, which are not declared to be active are *passive class*. Passive classes do not have their own thread of control, but their methods are executed by the thread of another active object.

Example 2.2. In the Java Programming Language, each class is passive by default. Objects whose class inherits from the class Thread, are active, after the start method has been called. See [57] for details. ♦

By applying the stereotype «signal» to a classifier, the instances of that classifier may be used as messages, which are sent asynchronously. The message itself is called a *signal*. Only active objects may receive these messages.

A classifier specifies a collection of *features*. Features are classified into *structural features*, which are attributes, and *behavioural features*, which are operations and receptions (a reception declares that an object can receive a signal of a certain type). For the sake of simplicity, we do not allow overloading of operations or generalisation on signals. Overloading of operations can be compiled away by renaming the operation names such that the name of each operation is unique.

In principle UML allows any object direct access to the data stored in an object, where this access is controlled by a *visibility* attribute of the feature. Here, we ignore this attribute. We assume later that all structural features, that is, attributes, are private to the object, that is, every object may access its own attributes only, and behavioural features are public, that is, every object may invoke operations of every other object. However, to constrain the object structure, these attributes and the associations are considered to be public in OCL and, therefore, part of the interface.

Before we formally define interfaces, we introduce the following notation: $A \rightarrow B$ designates the type of functions from A to B , whereas $A \multimap B$ designates the type of

2.1 Class Diagrams

partial functions from A to B . Observe that any function is also a partial function. We introduce a special predicate $E(f(x))$ which holds, if and only if the partial function f is defined in x . Using this predicate we define for any partial functions f its domain $\text{dom}(f) \stackrel{\text{def}}{=} \{x \mid E(f(x))\}$ and its range $\text{rng}(f) \stackrel{\text{def}}{=} \{f(x) \mid x \in \text{dom}(f)\}$. Each partial function $f : A \rightarrow B$ can be extended to a total function $f_\perp : A \rightarrow B$, by defining $f_\perp(x) = f(x)$ if $x \in \text{dom}(f)$ and $f_\perp(x) = \perp$ otherwise. In the sequel we assume that each partial function f is extended to a total function $f_\perp$ and we write simply f for $f_\perp$. We write A^B for the set of all functions of type $B \rightarrow A$.

We describe part of the interface using regular languages and identify words of the language with sequences. The symbol ϵ designates the empty sequence. A set S of letters represents the choice of letters, consequently, we write $\cup$ for the choice between two languages. Concatenation of words and languages is written by juxtaposition, or sometimes using the operator $\cdot$. The Kleene-star is written as S^* .

The interface of a class is specified using *segment descriptors*. A segment descriptor defines the interface defined by a class. The *full descriptor* defines all properties defined by a class, including the properties of all super-classes and super-interfaces (compare to Definition 2.3).

Definition 2.1 (Segment Descriptor). Let $\mathcal{N}$ be a set of names, $\mathcal{C} \subseteq \mathcal{N}$ a set of classifier names, $\mathcal{S} \subseteq \mathcal{C}$ a set of signal names, $\mathcal{A} \subseteq \mathcal{N}$ a set of names which we call attribute names, $\mathcal{M} \subseteq \mathcal{N}$ a set of names which we call operation names, such that $\mathcal{C}$, $\mathcal{A}$, and $\mathcal{M}$ are pairwise disjoint.

An *operation signature* s is an element of the language characterised by the regular language $\mathcal{C}^* \cdot (\mathcal{C} \cup \{\text{void}\})$. The last classifier in s describes the return type of the operation, where void states that the operation does not return a value.

The *segment descriptor* of a classifier c is the union of:

- A function $c_{\mathcal{A}} : \mathcal{A} \rightarrow \mathcal{C}$ which is a partial function defining the attributes defined in the class and the type of the attributes.²
- A function $c_{\mathcal{M}} : \mathcal{M} \rightarrow (\mathcal{C}^* \cdot (\mathcal{C} \cup \{\text{void}\}))$ describing the operations and the signature of the operation the class provides.
- If c is active, then the set $c_{\mathcal{S}} \subseteq \mathcal{S}$ assigns the class a set of signals which it accepts.

A segment descriptor can be interpreted as a partial function $c : \mathcal{N} \rightarrow (\mathcal{C} \cup (\mathcal{C}^* \cdot (\mathcal{C} \cup \{\text{void}\})) \cup \{\text{true}\})$ assigning attribute names to their types, operation names to their signatures, and signal names to true, which indicates that instances of the classes which are defined by this segment descriptor accept this signal. $\diamond$

Two types (or classifiers) are equal in UML, if they have the same name. This means that typing in UML is by names. Therefore, two objects are of a different type, if the

²A Function is also used as a set of pairs which describes its graph.

class they have been instantiated from have different names, even if all properties of the objects are identical. One exception to this rule is the typing of *tuples* in OCL, which are anonymous classes without operations. Two tuple-types are equal if they have the same members of the same type. This means, that typing of tuple types is by structure.

The relations between classes are *generalisation*, and *association*.

2.1.1 Generalisation

Generalisation captures the hierarchical relationship between classes. For example, “mammal” generalises “dog”, “horse”, “cat” and many others. Because a “dog” is a “mammal” it *inherits* all features of a mammal. Consequently, the *generalisation* relation is the inverse of the inheritance relation commonly defined in object-oriented programming languages. Formally, generalisation is a *partial order* $\leq$ on classes C .³ If a class $c \in C$, which we call parent, *generalises* another class c' , which we then call child, written $c' \leq c$, then all features defined by c are also features defined by c' . This means that the parent of a class has actually less features than its child. In Chapter 3 we describe how c' may override a feature of c in a type-safe way.

A class diagram describes the signature of classes in a distributed manner using *segment descriptors*, which define the attributes and operations particular to the class being described. The *full descriptor* contains the segment descriptor of the current class and, recursively, the full descriptor of all its parent classes. This way inheritance of features is realised.

The UML standard defines the following constraints for a class hierarchy.

1. For a classifier, no attribute, operation, or signal with the same signature may be declared in more than one of the segments (in other words, they may not be redefined).
2. Methods may be declared in more than one segment. Use the *youngest*. If there are two methods of the same age, the diagram is ill-formed.
3. The constraints on the full descriptor are the union on the segment current class and all of its ancestors. If any of them are inconsistent, then the model is ill-formed.

For any partial order $\leq$ define $\uparrow x \stackrel{\text{def}}{=} \{y \mid x \leq y\}$ and $\downarrow x \stackrel{\text{def}}{=} \{y \mid y \leq x\}$. Using these notations we define generalisations, taking the three properties above into account:

Definition 2.2 (Generalisation). Let $\leq$ be a partial order on classifier names C . The relation $\leq$ is called a *generalisation relation*, if:

³A *partial order* R is a relation that is *reflexive*, that is, $\forall x : xRx$, *transitive*, that is, $\forall x, y, z : xRy \wedge yRz \implies xRz$, and *anti-symmetric*, that is, $\forall x, y : xRy \wedge yRx \implies x = y$.

2.1 Class Diagrams

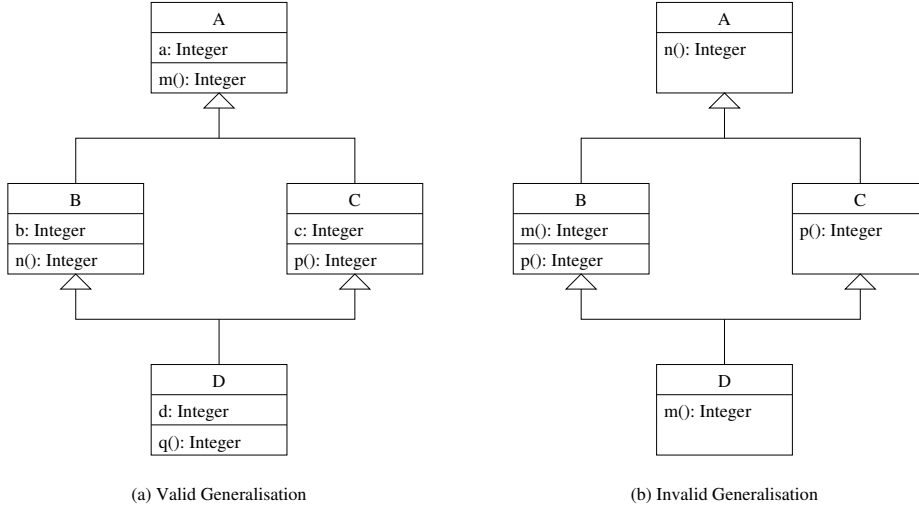


Figure 2.1: Valid and invalid generalisations

1. $\forall c, c' : c' \leq c \wedge c \neq c' \implies c \cap c' = \emptyset$, that is, c generalises c' when $c = c'$ or the segment descriptors of the class being generalised are disjoint from the segment descriptor of the generalising class.
2. $\forall c, c', c'' : c \leq c' \wedge c \leq c'' \wedge c \neq c' \wedge c \neq c'' \wedge c' \neq c'' \implies c' \cap c'' = \emptyset$, that is, if c is generalised by two different classifiers c' and c'' , then the segment descriptors of c' and c'' are disjoint. $\diamond$

Figure 2.1 displays two class diagrams with generalisations. A generalisation is displayed by an arrow between two classes, where the generalised class is at the end without any triangle, and the generalising class is at the end with the triangle. Consequently, both class diagrams claim that $B \leq A$, $C \leq A$, $D \leq B$, and $D \leq C$.

The class diagram (a) on the left hand side of Figure 2.1 displays generalisations, which all conform to Definition 2.2. All features defined in the segment descriptors are unique to their segment descriptor.

The class diagram (b) on the right hand side of Figure 2.1 displays generalisations which violate Definition 2.2. There, the class B does not generalise class D , because both define the method m in their segment descriptor. This contradicts Condition 1 of Definition 2.2. Moreover, D cannot be generalised by B and C , because both B and C define an operation p . This contradicts Condition 2 of Definition 2.2

Definition 2.3 (Full Descriptor). Let $\leq$ be a partial order on classes and c a classifier. Then the *full descriptor* of c is defined as: $\langle\langle c \rangle\rangle \stackrel{\text{def}}{=} \bigcup_{c' \in \uparrow c} c'$. $\diamond$

In the sequel we omit the parenthesis $\langle\langle c \rangle\rangle$ and simply write c for the full descriptor of a class, if this causes no ambiguity.

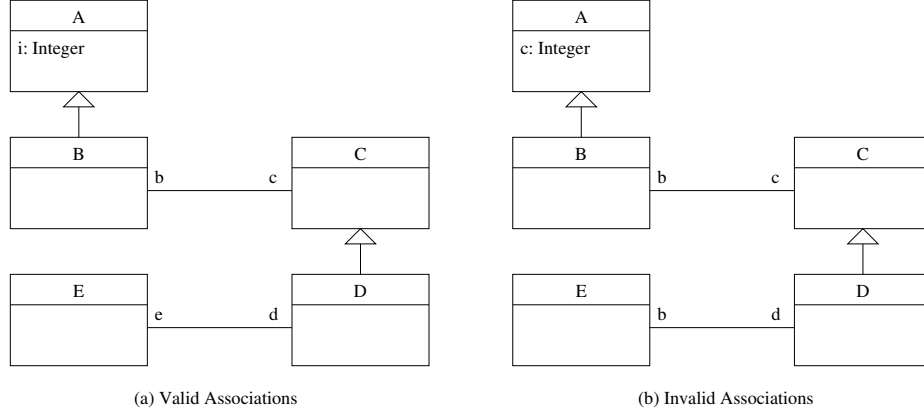


Figure 2.2: Valid and invalid associations

2.1.2 Association

Associations define relations on instances of classifiers. Associations are inspired by relations from relational databases. Associations consist of named ends referring to classifiers. An association with n ends is a n -ary relation on objects of the classifiers involved in the association. The names of association ends are treated like attributes of the objects participating in an association. Moreover, associations are *navigable*, that is, they provide names under which an object o “knows” another object o' .

Definition 2.4 (Association). Let C be a set of classifiers. An *association end* is a pair $\langle e, c \rangle$, where $e \in \mathcal{E}$ is a name and $c \in C$ is a class. An association is a set r of association ends satisfying:⁴ For any association $r = \{\langle e_1, c_1 \rangle, \langle e_2, c_2 \rangle, \dots, \langle e_n, c_n \rangle\}$ with $n \geq 2$, whose graph describes a function, we have: $\forall i : \forall c : 1 \leq i \wedge i \leq n \wedge c \in \bigcup_{1 \leq j \leq n} \downarrow c_j \implies \langle c \rangle(e_i) = \perp$. $\diamond$

By this definition the association end names occurring in an association are not allowed to be part of the full descriptor of the participating classes and their subclasses. Furthermore, in the definition of a class diagram we ensure that all associations do not introduce any conflicts.

Figure 2.2 displays two class diagrams with generalisations and associations. An association is displayed by a line between two classifiers.

The class diagram (a) on the left hand side of Figure 2.2 displays two associations, which all conform to Definition 2.4. Association $F = \{\langle b, B \rangle, \langle c, C \rangle\}$ does not conflict with any property defined in the classes A , B , C and D . It also does not conflict with the association $G = \{\langle e, E \rangle, \langle d, D \rangle\}$, because even though D “inherits” an association from C , the association end names do not conflict.

⁴Association names are typically called r from set $\mathcal{R}$ to stress that they are relations between objects and to avoid ambiguity with attributes, typically called a from set $\mathcal{A}$.

2.2 Object Diagrams

The class diagram (b) on the right hand side of Figure 2.2 displays associations which violate Definition 2.4. There, the class B inherits an attribute c from class A , but the association $F = \{\langle b, B \rangle, \langle c, C \rangle\}$ uses the name c for one of its ends, which contradicts the definition.

Finally, we define a class diagram.

Definition 2.5 (Class Diagram). A class diagram D is a triple $(C, \mathcal{R}, \leq)$, where C is a set of classes, $\mathcal{R}$ is a set of association relations over C and $\leq$ is a generalisation relation on C . Furthermore, we require the following well-formedness condition on associations in class diagrams:

For all $r, r' \in \mathcal{R}$ with $r \neq r'$ and all $c, c' \in C$ with $c' \leq c$ and $e, e' \in \mathcal{E}$ such that $\langle e, c \rangle \in r$ and $\langle e', c' \rangle \in r'$ we require that there exists not $e'' \in \mathcal{E}$ and no c'', c''' such that $\langle e'', c'' \rangle \in r$ and $\langle e'', c''' \rangle \in r'$. This means that the association end names of different associations which share related classifiers are different. $\diamond$

The class diagram (b) on the right hand side of Figure 2.2 displays associations violating Definition 2.5: associations $F = \{\langle b, B \rangle, \langle c, C \rangle\}$ and $G = \{\langle b, E \rangle, \langle d, D \rangle\}$ conflict, because both associations have an end called b and an end of type C by way of generalisation, which contradicts the definition. Note that in the latter case, it is ambiguous whether the instances associated under the name b to instances of D are instances of B or E .

2.1.3 Parameterisation

Also occurring in class-diagrams are *parameterised classes*. Parameterised classes are called *templates* in C++ [146] and *generics* in Java [57].

A parameterised class is not a classifier, but it is an *operator* mapping types and values to new types. The standard parameterised class occurring in almost all programming language is the array type. The type `ARRAY` in itself is not a type. Most of the time, a new array is declared as, for instance, `a : ARRAY 10 OF Integer`, which indicates that `ARRAY` is an operator taking two arguments: The first argument is the *size* of the array, here 10, and the second argument is the type of the elements stored in the array, here *Integer*.

2.2 Object Diagrams

As described in the previous section, a class diagram specifies the static structure of a model. Object diagrams describe a state or snap-shot of a system during runtime. An object diagram is a semantic model of a class diagram. A pair of object diagrams, which describe an action, is a semantic model for an OCL constraint.

Intuitively, an object diagram describes a state (or part of a state) of a system. It displays objects, the state of each object, and the *links* between objects.

Definition 2.6 (Object State). The *state of an object* is defined by a partial function $\omega : \mathcal{N} \rightarrow \mathbb{V}$, where $\mathcal{N}$ is the set of attribute names and $\mathbb{V}$ is a set of values. The set of all object states is called Ω . $\diamond$

Example 2.3. Let $\mathcal{N} = \{a, b, c\}$ be a set of names, and let $\mathbb{V} = \mathbb{B} \cup \mathbb{R}$ be the set of values, which consists of the disjoint union of booleans and reals. Then $\omega = \{a \mapsto \text{true}, c \mapsto 5.5\}$ is an object state, where the attribute a has the value $\omega(a) = \text{true}$ and the attribute c has the value $\omega(c) = 5.5$, whereas no value for the attribute b is defined. $\blacklozenge$

Definition 2.7 (Object Type). Let $\mathbb{O} \subseteq \mathbb{V}$ be an enumerable set of *object identities* with the special object identity $\text{null} \in \mathbb{O}$ representing the *absent object*. Let D be a class diagram. The function $\tau : \mathbb{O} \rightarrow \mathcal{C}$ assigns to object identities a class. Let $o \in \mathbb{O}$ be an object identifier. If $\tau(o)$ is defined, we say that o exists.

In the preceding definition we do not require a type for each object identity. Those object identities, which do not have a type are the identities of those objects, which have not been created yet.

2.2.1 Associations and Navigation Expressions

Next, we define the meaning of associations with the definition of the latter given by Definition 2.4. Recall that an association r maps association end names to classes.

Definition 2.8 (Association Valuation). Let r be an association. Then the semantics of r , called its *valuation*, written $\llbracket r \rrbracket$ is a multi-set of partial functions $\text{dom}(r) \rightarrow \mathbb{O} \setminus \{\text{null}\}$ such that for all $f \in \llbracket r \rrbracket$ and for all $n \in \text{dom}(r) \cap \text{dom}(f)$ we have $\tau(f(n)) \leq r(n)$. $\diamond$

We write $\wr \cdot \wr$ to construct multi-sets (or bags).

Observe that an association is not like an attribute; it is more like a table in relational databases, as developed by Codd [29].

Example 2.4. Recall the class diagram depicted in Figure 2.2 (a). Let $\mathbb{O} = \{1, 2, 3, 4, 5\}$. Define $\tau = \{1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 4 \mapsto D, 5 \mapsto E\}$. One association valuation for the association $F = \{\langle b, B \rangle, \langle c, C \rangle\}$ is given by $\llbracket F \rrbracket = \wr \{b \mapsto 2, c \mapsto 3\} \wr$, associating the object identified by 2 to the object identified by 3. Another association valuation is $\llbracket F \rrbracket = \wr \{b \mapsto 2, c \mapsto 3\}, \{b \mapsto 2, c \mapsto 4\} \wr$, which associates the object identified by 2 to the object identified by 3 and the object identified by 4. Observe, that this is a correct association state, because $D \leq C$ in the underlying class diagram. Finally, to provide a counter example, consider the interpretation $\llbracket F \rrbracket = \wr f \wr$, where $f = \{b \mapsto 1, c \mapsto 3\}$. This interpretation is not a valid association valuation, because $\tau(f(b)) = A$, $F(b) = B$, and $\neg \tau(f(b)) \leq F(b)$. $\blacklozenge$

Next, we define the interpretation of navigation expressions, for which we already use the syntax of OCL.

2.2 Object Diagrams

Definition 2.9 (Navigation). Let $\mathcal{N}$ be a set of association end names with typical element n . A navigation expression is inductively defined by the grammar $e ::= self \mid e.n$.

The meaning of a navigation expression e in a class diagram $D = \langle C, \mathcal{R}, \leq \rangle$ is defined as a function $\llbracket \cdot \rrbracket$, mapping each navigation expression to a function $\mathbb{O} \rightarrow \mathbb{N}^{\mathbb{O}}$, which maps each object identity, used to evaluate the navigation expression $self$, to a bag of objects:⁵

$$\llbracket e \rrbracket(s) \stackrel{\text{def}}{=} \begin{cases} \{s\} & , \text{ if } e = self \\ \{o \mid \exists r, f, n' : r \in \mathcal{R} \wedge n \neq n' \wedge f \in \llbracket r \rrbracket \\ \quad \wedge o = f(n) \wedge f(n') \in \llbracket e' \rrbracket(s)\} & , \text{ if } e = e'.n. \end{cases}$$

◇

Assuming that all association end names occur in the navigation expression, which type-checking can establish, as described in Chapter 3, we observe that the result of a navigation expression is always a multi-set of objects. In contrast to the usual programming languages the absent object, here denoted by *null*, does not occur in the semantic value of a navigation expression; instead, *null* is represented by the empty bag. Also, it is possible to navigate the empty bag. Navigating the empty bag results in the empty bag.

In contrast to standard OCL, our definition of the meaning of a navigation expression is flattening, that is, navigating the result of a navigation expression results in a bag of objects, and not in a bag of bags. Thereby we retain the semantics of OCL prior to version 2.0.

Example 2.5. Consider the class diagram of Figure 2.3 with classes $C = \{A, B, C, D, E\}$ and associations $X = \{\langle a, A \rangle, \langle b, B \rangle, \langle c, C \rangle\}$, $Y = \{\langle c, C \rangle, \langle e, E \rangle\}$, and $Z = \{\langle b, B \rangle, \langle d, D \rangle\}$.

Let the set of objects be $\mathbb{O} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ with $\tau = \{1 \mapsto A, 2 \mapsto A, 3 \mapsto B, 4 \mapsto C, 5 \mapsto C, 6 \mapsto C, 7 \mapsto D, 8 \mapsto D, 9 \mapsto E, 10 \mapsto E\}$. Let the valuations of X , Y , and Z be as displayed in Table 2.1. Then:

$$\llbracket self.c.e \rrbracket(1) = \{o \mid \exists y, f, n' : e \neq n' \wedge f \in \llbracket y \rrbracket \wedge o = f(e) \wedge f(n') \in \llbracket self.c \rrbracket(1)\} \quad (2.1)$$

To evaluate this, we have to evaluate:

$$\llbracket self.c \rrbracket(1) = \{o \mid \exists x, f', n'' : c \neq n'' \wedge f' \in \llbracket x \rrbracket \wedge o = f'(c) \wedge f'(n'') \in \llbracket self \rrbracket(1)\} \quad (2.2)$$

Since the only association end which is mapped to 1 in Table 2.1 is a , belonging to the association X , apparently $n'' = a$ and $x = X$. So choose $n'' = a$ and $x = X$. Returning to (2.2) we now have:

$$\llbracket self.c \rrbracket(1) = \{o \mid \exists f' : f' \in \llbracket X \rrbracket \wedge o = f'(c) \wedge f'(a) \in \llbracket self \rrbracket(1)\}$$

⁵The set of all bags can be represented by the function from its elements into the natural numbers, therefore we write $\mathbb{N}^{\mathbb{O}}$ for the bag of all objects.

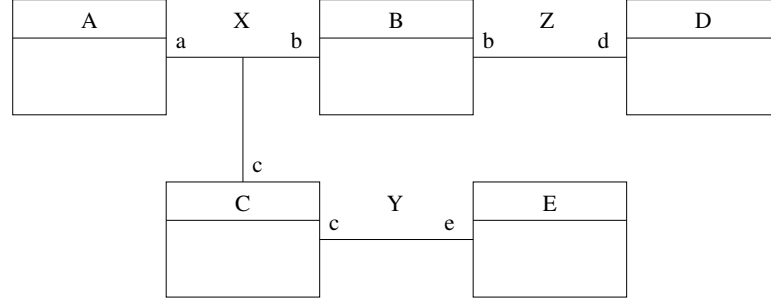


Figure 2.3: Navigation example

$\llbracket X \rrbracket$			$\llbracket Y \rrbracket$		$\llbracket Z \rrbracket$	
a	b	c	c	e	b	d
1	3	5	4	9	4	7
1		6	5	9	4	7
2	3	6	5	10		
			6	10		

Table 2.1: Example valuations of associations

In Table 2.1 we find two possibilities for f' with $f'(a) = 1$. For these $f'(c) = 5$ and $f'(c) = 6$ holds. So

$$\llbracket self.c \rrbracket(1) = \{5, 6\} \quad (2.3)$$

Substituting (2.3) in (2.1) results in:

$$\llbracket self.c.e \rrbracket(1) = \{o \mid \exists y, f, n' : e \neq n' \wedge f \in \llbracket y \rrbracket \wedge o = f(e) \wedge f(n') \in \{5, 6\}\}$$

From Table 2.1 one finds that $y = Y$ and $o = 9$ or $o = 10$ for $f(c) = 5$ and $o = 10$ for $f(c) = 6$. So, $n' = c$ and one concludes that

$$\llbracket self.c.e \rrbracket(1) = \{9, 10, 10\}$$

◆

Example 2.6. Reconsider Example 2.5. We now evaluate $\llbracket self.b.c \rrbracket(7)$.

$$\llbracket self.b.c \rrbracket(7) = \{o \mid \exists x, f, n' : c \neq n' \wedge f \in \llbracket x \rrbracket \wedge o = f(c) \wedge f(n') \in \llbracket self.b \rrbracket(7)\} \quad (2.4)$$

To evaluate this, we have to evaluate:

$$\llbracket self.b \rrbracket(7) = \{o \mid \exists z, f', n'' : b \neq n'' \wedge f' \in \llbracket z \rrbracket \wedge o = f'(b) \wedge f'(n'') \in \llbracket self \rrbracket(7)\} \quad (2.5)$$

This association end is mapped to 7 in Table 2.1 is d in association Z , therefore $n'' = d$ and $z = Z$. Returning to (2.5) we now have:

$$\llbracket self.b \rrbracket(7) = \{o \mid \exists f' : f' \in \llbracket Z \rrbracket \wedge o = f'(b) \wedge f'(d) \in \llbracket self \rrbracket(7)\} \quad (2.6)$$

2.2 Object Diagrams

We find two possibilities in the association valuation of Z with $f'(d) = 7$, which both have $f'(b) = 4$. Therefore, Equation (2.5) simplifies to:

$$\llbracket self.b \rrbracket(7) = \{4, 4\} \quad (2.7)$$

Substituting (2.7) in (2.4) results in:

$$\llbracket self.b.c \rrbracket(7) = \{o \mid \exists x, f, n' : c \neq n' \wedge f \in \llbracket x \rrbracket \wedge o = f(c) \wedge f(n') \in \{4, 4\}\} \quad (2.8)$$

Now we have to find an association where one end is 4 and the other end has the name c . Unfortunately, no association in Table 2.1 satisfies this. Note that association Y is not eligible, because 4 is in column c , but this is the association end we want to navigate. Therefore, we simplify (2.8) to

$$\llbracket self.b.c \rrbracket(7) = \{o \mid \text{false}\}$$

which is the empty bag. ♦

This last example demonstrates that any navigation expression is always defined.

We now define the state of the system, which is called *object diagram* in UML, as the state of all its objects, and an association valuation.

Definition 2.10 (Object Diagrams, Links). An *object diagram* $\sigma = \langle \tau, \xi, \llbracket \cdot \rrbracket \rangle$ is a triple that consists of a partial function $\tau : \mathbb{O} \rightarrow C$ assigning to each object identity $o \in \mathbb{O}$ its class, a partial function $\xi : \mathbb{O} \rightarrow \Omega$, where the set of all these functions is called Ξ , mapping object identities to object states, which satisfies for all $o \in \mathbb{O}$ that $\tau(o)$ is defined if and only if $\xi(o)$ is defined, and a function $\llbracket \cdot \rrbracket$ assigning to each association an association valuation. The set of Object Diagrams is denoted by Σ .

A *link* named n exists between two objects in the object diagram, written $o \xrightarrow{n} o'$, if $o' \in \llbracket self.n \rrbracket(o)$. If there exists a n such that $o \xrightarrow{n} o'$, we write $o \rightarrow o'$. ♦

Observe that $\rightarrow$ is symmetric.

Proposition 2.11. *Let $\langle \tau, \xi, \llbracket \cdot \rrbracket \rangle$ be an object diagram. Then $\rightarrow$ is symmetric.*

Proof. Let o, o' be a pair of objects such that $o \rightarrow o'$. By Definition 2.10 we then find an association r and an association end name n such that $o' \in \llbracket self.n \rrbracket(o)$. By Definition 2.9 we find a name n' and an association state f such that $f \in \llbracket r \rrbracket$ and $f(n) = o'$ and $f(n') = o$. Consequently, we have $o \in \llbracket self.n' \rrbracket(o')$, and therefore $o' \rightarrow o$. □

It is possible to encode associations as attributes of the type of the associated classifier if each object identifier, except for *null*, occurs at most once in each column. We prove this in the next lemma.

Lemma 2.12. *Let $\sigma = \langle \tau, \xi, \llbracket \cdot \rrbracket \rangle$ be an object diagram, where in any valuation of each association an object is, for each association end name, associated to at most one other object. Then the association can be represented by a set of attributes, with the respectively associated classifier as the attribute's type, named by the association ends of the object.*

Proof. Let $\sigma = \langle \tau, \xi, \llbracket \cdot \rrbracket \rangle$ be as described in the assumption and D the underlying class diagram. First, we prove that under this assumption $\rightarrow$ can be interpreted as a function with the signature $f : \mathbb{O} \rightarrow \mathcal{E} \rightarrow \mathbb{O}$, where this function results in *null* if there exists no object o' such that $o \xrightarrow{e} o'$.

Let o be an object in σ and e an association end name. Then, by assumption, there exists at most one object o' with $o \xrightarrow{e} o'$. If such an object does not exist, we assign $f(o, e) = \text{null}$. Assume that such an o' exists. If there are two distinct objects o' and o'' such that there are e and e' with $o \xrightarrow{e} o'$ and $o \xrightarrow{e'} o''$, then Definition 2.5 and the assumption imply $e \neq e'$. Therefore, $\rightarrow$ is functional.

To prove that the association can be represented by a set of attributes, with the respectively associated classifier as the attribute's type, named by the association ends of the object, let D' be the class diagram obtained from D by removing all associations but adding the association end names as attributes of the target's classifiers type to the associations' source classifiers. Observe, that by Definition 2.5 D is a well-formed class diagram. Similarly, from σ we obtain an object diagram σ' on D' , where each association end e , which now is an attribute, of o is interpreted by $\xi'(o)(e) = \iota x : o \xrightarrow{e} x$. $\square$

We use Russell's ι operator for *definite description*. The expression $\iota x : P(x)$ reads as "The uniquely determined x such that $P(x)$ holds. If no such x exists or it is not unique, the meaning of the expression $\iota x : P(x)$ is not defined. However, in the context of $\rightarrow$ we assume that if no object o' exists, such that $o \xrightarrow{e} o'$ holds, then $\iota x : o \xrightarrow{e} x$ shall represent *null*.

2.2.2 Relating Object Diagrams to Class Diagrams

In order to relate an object diagram to a class diagram we introduce two relations between class diagrams and object diagrams. The first relation formalises that an object diagram conforms to a class diagram, when the values assigned to the objects in the object diagram comply with those in the class diagram. This notion of conformance is very useful for object diagrams which display a view on the state of a system, that is, which abstract from certain details.

Definition 2.13. An object diagram $\sigma = \langle \tau, \xi, \llbracket \cdot \rrbracket \rangle$ conforms to a class diagram $D = \langle C, \mathcal{R}, \leq \rangle$, written $\sigma \triangleright D$, if and only if:

- For all objects $o \in \mathbb{O}$ we have $\tau(o) \in C$, that is, all objects in an object diagram have a type defined in the class diagram.

2.2 Object Diagrams

- For all objects $o \in \mathbb{O}$ and all attribute names $a \in \mathcal{A}$ where $\xi(o)(a)$ is not undefined it holds that $\tau(\xi(o)(a)) \leq \tau(o)(a)$, that is, whenever an object o defines a value for a name a , then the classifier of this object defines an attribute for this name with a type that conforms to the type of the value of a in o .
- For all objects $o \in \mathbb{O}$ and $o' \in \mathbb{O}$ with $o \rightarrow o'$ we find an association $r \in \mathcal{R}$ satisfying:
 - There exists $e \in \mathcal{E}$ with $\xi(o)(e) = o'$ and there exists a classifier $c' \in \mathcal{C}$ with $\tau(o') \leq c'$ such that $\langle e, c' \rangle \in r$.
 - If there exists a $e' \in \mathcal{E}$ and a classifier $c \in \mathcal{C}$ with $\tau(o) \leq c$ and $\langle e', c \rangle \in r$, then $o' \rightarrow o$. ♦

The empty object diagram, that is, $\sigma = \langle \emptyset, \emptyset, \emptyset \rangle$ conforms to all class diagrams.

Example 2.7. Consider the class diagram in Figure 2.1 (a). Assume there is only one object with identity 1 with $\tau(1) = B$. For $\xi = \{1 \mapsto \{a \mapsto 1\}\}$ (because instances of B inherit the attributes of all their super-classes) and an empty association valuation, we obtain a valid object diagram. However, for $\xi = \{1 \mapsto \{a \mapsto \text{true}\}\}$ the object diagram does not conform to the class diagram in Figure 2.1. ♦

The second relation formalises whether the object diagram displays all information defined in a class diagram. Such object diagrams completely describe a state conforming to a class diagram.

Definition 2.14. An object diagram $\sigma = \langle \tau, \xi, \llbracket \cdot \rrbracket \rangle$ is a state of a class diagram $D = \langle \mathcal{C}, \mathcal{R}, \leq \rangle$, written $\sigma \models D$, if and only if:

- $\sigma \triangleright D$.
- For all $c \in \mathcal{C}$ and $o \in \mathbb{O}$ with $\tau(o) = c$ we have for all $n \in \mathcal{N}$: $\tau(\xi(o)(n)) \leq c(n)$, that is, for every object o of class c it holds that the type of the value $\xi(o)(n)$ of its attribute n is a subtype of the type of the attribute declared in the segment descriptor of the class. ♦

Example 2.8. Consider again the class diagram in Figure 2.1 (a). Assume there is only one object with identity 1 with $\tau(1) = B$. For $\xi = \{1 \mapsto \{a \mapsto 1\}\}$ and an empty association valuation this does not define a state for the class diagram. For $\xi = \{1 \mapsto \{a \mapsto 1, b \mapsto 0\}\}$ and an empty association valuation this defines a state for the class diagram. ♦

These are all definitions needed for working with object diagrams.

2.3 Object Constraint Language

In this section we present the syntax of OCL, describe the syntactic categories, and develop the minimal syntactic categories of OCL. Well-typedness of OCL terms is described in Chapter 3.

The important notions developed in this section define when a constraint φ holds within a particular object diagram σ which conforms to its class diagram D . This is written as: $D, \sigma \models \varphi$.

Some constraints are behavioural constraints, that is, a pair of a precondition φ and a postcondition ψ . For such specifications we need to evaluate the constraints in a pair σ, σ' of object diagrams, also called an *action*. This is written as: $D, \sigma, \sigma' \models \langle \varphi, \psi \rangle$.

The question whether a constraint φ *conforms* to a class diagram D , written as $\varphi \triangleright D$, is a type-checking problem and is the topic of Chapter 3, that is, describing when φ is well-typed in D .

2.3.1 Context of Constraints

All constraints in OCL are associated with a *context*. The context of a constraint defines the entities for which the constraint has to hold. It also defines the way it is interpreted. Context declarations link constraints to class diagrams. OCL 2.0 allows classifiers, attributes, and operations as contexts for constraints. In this section we define the abstract syntax of the *context declaration*, whereas we define the abstract syntax of the constraints in the following Section 2.3.2. Syntactically, we assign a list of constraints φ to its context c using the form:

$$\textbf{context } c (s : \varphi)^* .$$

Each constraint φ is tagged with s , which stands for one of **inv**, **pre**, **post**, **init**, **deriv**, and **def**. These tags are called stereotypes in UML. The intended meaning of these stereotypes is:

- Constraints of stereotype **inv** are *invariants*.
- Constraints of stereotype **pre** are *preconditions*.
- Constraints of stereotype **post** are *postconditions*.
- Constraints of stereotype **init** are *initial value specifications*, usually attached to attributes.
- Constraints of stereotype **deriv** are *derived value specifications*, that is, they specify a rule defining how the value of an association or an object's attribute is to be computed.

2.3 Object Constraint Language

- Constraints of stereotype **def** are indeed no constraints but definitions of derived attributes and side-effect-free operations which are only used in OCL constraints. These definitions supersede the let-construct from OCL 1.4. The type of the term in a **def** context has to conform to the return type of the operation or attribute being defined.

We only consider constraints with stereotypes **inv**, **pre**, and **post**. Expressions of stereotypes **init** are used to define the initial value of an attribute, whereas expressions of stereotype **deriv** define how the value of a derived attribute or association is computed. These stereotypes do not define when a system is correct but are part of the specification of the system. Therefore, we do not consider these expressions here.

Classifier

One context of an OCL constraint is the *classifier context*. Only invariants are allowed as constraints on classifiers. A constraint in the context of a classifier c is written as

$$\textbf{context } c \textbf{ inv} : t \quad ,$$

where c is the name of the context and t is an OCL expression.

Operation

Another context of an OCL constraint is the *operation context*. In this context only pre- and postconditions are allowed, which are used to specify the behaviour of the constrained operation. A constraint in the context of an operation m defined in c is written as

$$\begin{array}{l} \textbf{context } c :: m(v_1 : T_1, \dots, v_n : T_n) : T \\ \textbf{pre} : t \\ \textbf{post} : t' \end{array} \quad ,$$

where c is the class which defines the operation m with formal parameters $v_1, \dots, v_n$ of types $T_1, \dots, T_n$, respectively, and where the operation returns a value of type T . The constraint t describes the precondition, whereas t' describes the postcondition of m . If either of them is omitted, then we define it to true.

2.3.2 Abstract Syntax

We start with a description of abstract OCL, a simple core language, into which almost all OCL expressions can be translated. The grammar of OCL is defined next. First, we define the set of all OCL types, and then we define the set of all OCL expressions.

Definition 2.15 (Grammar of OCL Types). Let $D = \langle C, \mathcal{R}, \leq \rangle$ be a class diagram.

1. Every $c \in C$ is an OCL Type.

2. Every data type defined in the OCL standard library (see Section 2.3.4), that is, *OclAny*, *OclVoid*, *OclInvalid*, *Boolean*, *Integer*, *Real* is an OCL type.
3. If τ is an OCL type, then *Collection*(τ), *Set*(τ), *Bag*(τ), *Sequence*(τ) are OCL types.
4. If $v_0, \dots, v_n$ are variable names and $\tau_0, \dots, \tau_n$ are types, then *Tuple*{ $v_0 : \tau_0, \dots, v_n : \tau_n$ } is an OCL type. $\diamond$

Definition 2.16 (Grammar of OCL). Let $\mathcal{V}$ denote an enumerable set of names we call *variables*, which contains *self* and *result*.

1. true and false are OCL expressions. null is an OCL expression.
2. $\dots, -1, 0, 1, \dots$, that is, each integer number is an OCL expression.
3. If $e_1, e_2, \dots, e_n$ is a list of OCL expressions, which may be empty (in which case $n = 0$), then *Bag*{ $e_1, e_2, \dots, e_n$ }, *Sequence*{ $e_1, e_2, \dots, e_n$ }, and *Set*{ $e_1, e_2, \dots, e_n$ } are OCL expressions. Expressions of this kind are called *collection literal expressions*.
If e, e' are expressions, then $e..e'$ is a *range expression*. Range expressions may only occur in collection literal expressions.
4. Any variable name $v \in \mathcal{V}$ is an OCL expression.
5. If t, t_1, t_2 are OCL expressions, so is **if** t **then** t_1 **else** t_2 **endif**. We call expressions of this kind *conditional expressions* (of course, t is intended to be of type *Boolean*, but in this definition syntax is separated from type-checking).
6. Let $n \in \mathbb{N}$. If t, t_i are OCL expressions, v_i variable names, and T_i are OCL types, for all $i \leq n$, then **let** $v_0 : T_0 = t_0, \dots, v_n : T_n = t_n$ **in** t is an OCL expression. We call expressions of this type *let expressions*.
7. If t is an OCL expression and v is an attribute name, then $t.v$ is an OCL expression. We call expressions of this kind *attribute call expressions*.
8. If t is an OCL expression and v is an attribute name, then $t.v@pre$ is an OCL expression. We call expressions of this kind *previous attribute call expressions*. This syntax is only allowed in postconditions and is intended to refer to previous values defined by the precondition in a pre-post specification.
9. If $t_0, t_1, \dots, t_n$ are OCL expressions and m is an operation name, then an *operation call expression* is written as $t_0.m(t_1, t_2, \dots, t_n)$.
10. If $t_0, t_1, \dots, t_n$ are OCL expressions and n is an operation name, then an *previous operation call expressions* is written $t_0.v@pre(t_1, t_2, \dots, t_n)$. This syntax is only allowed in postconditions.

2.3 Object Constraint Language

11. For OCL expressions $t_0, t_1, \dots, t_n$ and for m an operation name $t_0 \rightarrow m(t_1, t_2, \dots, t_n)$ is an OCL expression. We call expressions of this kind *collection property call expressions*.
12. An *iterate expressions* is written $t_0 \rightarrow \text{iterate}(v_0 : T_0, v_1 : T_1, \dots, v_n : T_n; a : T = t \mid t_n)$, where t, t_0, t_n are OCL expressions, $a, v_0, v_1, \dots, v_n$ are variable names, and $T, T_0, T_1, \dots, T_n$ are types.
The intention of an iterate expression is to apply the *body* t_n , in which the *accumulator* a and the *iterator variables* $v_0, v_1, \dots, v_n$ occur freely, to each element of the collection t_0 , where each iterator variable visits each member of the collection itself. The result of the iterate expression is accumulated in a , whose value is initialized with the value of t .
13. If t is an OCL expression, so is $t \rightarrow \text{flatten}()$. We call expressions of this kind *flatten expressions*. In principle, flatten expressions are a kind of collection call expressions, but we introduce them as atomic expressions, because they are treated specially in the type system. See Section 3.3.5 for details.
14. $T.\text{allInstances}()$ and $T.\text{allInstances@pre}()$ are OCL expressions for all type expressions T . These expressions are called *all instances of T* , respectively, *all previous instances of T* expressions. Because of the special semantics of this kind of expression and because it is applied to a type name, we introduce this expression here.

We call the set of expressions generated by this grammar $\mathcal{L}_{\text{OCL}}$. ◇

This inductive definition of the abstract syntax of OCL can be summarised by the following grammar given in BNF (see Appendix A for an explanation of our notation).

$$\begin{aligned}
 t ::= & \text{true} \mid \text{false} \mid \dots \mid -1 \mid 0 \mid 1 \mid \dots \mid \text{self} \mid v \mid t.a \mid t.m(t_1, \dots, t_n) \\
 & \mid t \rightarrow m(t_1, \dots, t_n) \mid t \rightarrow \text{iterate}(v_0 : \tau_0, \dots, v_n : \tau_n; a = t \mid t_0) \\
 & \mid t \rightarrow \text{flatten}() \mid \text{if } t \text{ then } t' \text{ else } t'' \text{ endif} \\
 & \mid \text{let } v_0(v_{0,0}, \dots, v_{0,m_0}) : \tau = t_0, \dots, v_n(v_{n,0}, \dots, v_{n,m_n}) : \tau = t_n \text{ in } t
 \end{aligned}$$

The preceding definition only characterises a subset of all OCL expressions. A grammar of the concrete syntax of OCL is described in Appendix A. Other expressions, for example, *forAll*, have been omitted, because they can be translated into the core language defined above. From now on we only assume well-typed expressions, where well-typedness of OCL expressions is defined in Chapter 3.

An OCL expression is called an *OCL formula* or *OCL constraint* if the type of the constraint in the current contextual class-diagram is *Boolean*. In the next section we define a formal semantics for OCL constraints based on the OCL standard [113] and the thesis of Richters [133], and discuss the problems of this semantics.

2.3.3 Semantics

We develop a semantics of OCL as specified in Section 2.3.2 by specifying an *abstract machine with oracles* that interprets OCL constraints in a class diagram and two object diagrams representing an action. This abstract machine is a formalisation of the evaluator described in the OCL 2.0 standard [113, Chapter 10]. In the standard the semantics is defined using UML diagrams and OCL constraints, whereas we define a function which interpreting constraints. A similar interpreter has been proposed by Mark Richters in [133] for OCL 1.3.

The purpose of this interpreter is to define a reference semantics of OCL which we can use to translate OCL constraints into PVS and to identify classes of constraints which cannot be represented in PVS. The translation of OCL to PVS and its soundness proof is described in Chapter 4. The second purpose is to establish a foundation on which the merits and flaws of OCL can be discussed.

Before we formally define the abstract machine, we have to point out that this machine is not computationally effective, because OCL defines a function *oclIsInvalid* which is supposed to result in true on diverging expressions and which therefore solves the halting-problem. Therefore, we first define an abstract machine for the fragment of OCL *without oclIsInvalid*. The semantics full OCL is then defined by introducing an *oracle* which decides whether *oclIsUndefined* returns true or false.

Another crucial observation is the way recursive specifications may be written. The variable *result* refers to the result of an operation call in the postcondition of the operation. Its value can be specified recursively, as described in [113, p. 31]:

The right-hand-side of this definition may refer to the operation being defined (that is, the definition may be recursive) as long as the recursion is not infinite.

One of the changes from OCL 2.0 to its preceding versions is forbidding recursive let-definitions [113, p. 104]. Instead, recursive expressions in OCL are defined by defining new side-effect-free auxiliary operations in the class diagram:

Note that let-definitions that take arguments are no longer allowed in OCL 2.0. This feature is redundant. Instead, a modeller can define an additional operation in the UML Classifier, potentially with a special stereotype to denote that this operation is only meant to be used as a helper operation in OCL expressions. The postcondition of such an additional operation can then define its result value.

The OCL 2.0 standard fails to define the meaning of such a recursive expression. One way to specify the meaning of such an expression is by constraining *result* in the operations post-condition, as described above. However, it is not defined what the meaning of such an expression is if the pre-condition does not hold.

2.3 Object Constraint Language

The second solution would be to define a method for the operation, which is recursive. The method need not be implemented in OCL. It can be specified in any other programming language, one of the behavioural notations of UML, or even using natural language. It is not our intention to formalise these different ways to specify a method body.

Furthermore, the operation, which occurs in an OCL constraint can be specified by its post-condition *and* a method. Then it is not specified whether the meaning described by the post-condition is to be used or whether the method is to be used for interpreting the constraints.

Recall that the definition provided in this section is intended to provide the basis on which we can prove the soundness of our translation and that the abstract machine defined in this section is intended as a formalisation of the interpreter defined in the OCL 2.0 standard using UML and OCL. Later, we are only concerned with the translation of OCL constraints in PVS. These translated constraints have to be type consistent in PVS, which implies that termination of recursive functions has to be proved and that each function is only applied to values where it is defined.

In order to simplify the presentation of the translation from OCL to PVS, our definitions adapt the definition of the type universe used to define the formal semantics of the logic used in the theorem prover PVS [123] to our purposes.

In the following definition of the type universe we use for each carrier D , which interprets a type of OCL, the special symbol $\mathcal{J}_D$, which denotes the undefined value for the carrier D . The value $\mathcal{J}$ is a value like any other, and subject to the same rules of reasoning. For example, given two terms t and t' , which both represent $\mathcal{J}$, we may conclude $t = t'$ is true. The values $\mathcal{J}$ represent run-time errors like division by zero, the failure of the abstract machine to proceed with its evaluation, or that the interpretation of an expression by the abstract machine may not terminate. However, OCL treats the values $\mathcal{J}$ as proper values in its interpreter (even though it does not have a corresponding literal in the concrete syntax), because, for example, one may test whether an expression evaluates to $\mathcal{J}$ using *oclIsUndefined*.⁶ Keep in mind that type consistency in PVS implies that all expressions are defined, that is, their value is not $\mathcal{J}$ (see Chapter 4, where we show that definedness of the translated constraint implies the definedness of the original constraint.).

Definition 2.17 (Type Universe). The *type universe* U_ω of OCL is inductively defined by:

- $\mathbb{B}_{\mathcal{J}} \stackrel{\text{def}}{=} \{\text{true}, \text{false}, \mathcal{J}_{\mathbb{B}}\}$ as interpretation of the boolean literals, where $\mathcal{J}_{\mathbb{B}}$ repre-

⁶We specifically use $\mathcal{J}$ instead of $\perp$ (read “bottom”) to avoid confusion with $\perp$ as it appears in domain theory. The main reason for this are that the semantics of OCL is given as an operational semantics and not as a denotational semantics. Also, in denotational semantics, $\perp$ is supposed to mean non-termination or “no value at all”, whereas $\mathcal{J}$ is a value. Also, the definition of *oclIsUndefined* in OCL defines a non-monotonic and non-computable function. Therefore, classical domain theory is not applicable for the semantics.

sents the undefined truth value.

- $\mathbb{R}_{\mathcal{J}} \stackrel{\text{def}}{=} \mathbb{R} \cup \{\mathcal{J}_{\mathbb{R}}\}$ as interpretation of numeric expression, where $\mathcal{J}_{\mathbb{R}}$ represents the undefined real number.
- $\mathbb{O}_{\mathcal{J}} \stackrel{\text{def}}{=} \mathbb{O} \cup \{\mathcal{J}_{\mathbb{O}}\}$ as interpretation of objects identities, where $\mathcal{J}_{\mathbb{O}}$ represents the undefined object. Observe that $\mathcal{J}_{\mathbb{O}}$ is distinct from $null \in \mathbb{O}$, where $null$ will be the interpretation of null.
- $U_0 = \{\mathbb{B}_{\mathcal{J}}, \mathbb{R}_{\mathcal{J}}, \mathbb{O}_{\mathcal{J}}\}$ is the type universe of the individual.
- For each $i \in \mathbb{N}$ define $U_{i+1} = U_i \cup \{(X \times Y) \cup \{\mathcal{J}_{X \times Y}\} \mid X, Y \in U_i\} \cup (\bigcup_{X \in U_i} (2^X \cup \{\mathcal{J}_{2^X}\})) \cup \{X^Y \cup \{\mathcal{J}_{X^Y}\} \mid X, Y \in U_i\}$.
- Finally define $U_{\omega} \stackrel{\text{def}}{=} \{\mathcal{J}\} \cup \bigcup_{i \in \mathbb{N}} U_i$. ◇

This means that for any level $i \in \mathbb{N}$ the set U_{i+1} contains all values defined on the level i together with the power set of all elements of this level, the set of all functions between elements of level i , and the set of all pairs of elements of level i . We assign to each type occurring in the OCL specification an element of U_{ω} .

Definition 2.18 (Type Interpretation). We define a function $\mathfrak{D}$ from OCL types to elements of U_{ω} as follows:

- $\mathfrak{D}(\text{Boolean}) \stackrel{\text{def}}{=} \mathbb{B}_{\mathcal{J}}$.
- $\mathfrak{D}(\text{Integer}) \stackrel{\text{def}}{=} \mathbb{R}_{\mathcal{J}}$, or to be precise, $\mathfrak{D}(\text{Integer}) \stackrel{\text{def}}{=} \mathbb{Z}_{\mathcal{J}}$, where $\mathbb{Z}_{\mathcal{J}} \stackrel{\text{def}}{=} \mathbb{Z} \cup \{\mathcal{J}_{\mathbb{R}}\}$. Note that $\mathbb{Z}_{\mathcal{J}} \subset \mathbb{R}_{\mathcal{J}}$ holds.
- $\mathfrak{D}(\text{Real}) \stackrel{\text{def}}{=} \mathbb{R}_{\mathcal{J}}$.
- $\mathfrak{D}(\text{Bag}(\tau)) \stackrel{\text{def}}{=} \mathbb{N}^{\mathfrak{D}(\tau)} \cup \{\mathcal{J}_{\mathbb{N}^{\mathfrak{D}(\tau)}}\}$.
- $\mathfrak{D}(\text{Set}(\tau)) \stackrel{\text{def}}{=} 2^{\mathfrak{D}(\tau)} \cup \{\mathcal{J}_{2^{\mathfrak{D}(\tau)}}\}$.
- $\mathfrak{D}(\text{Sequence}(\tau)) \stackrel{\text{def}}{=} \{\mathcal{J}_{\mathfrak{D}(\tau)^{\mathbb{N}}}\} \cup \bigcup_{n \in \mathbb{N}} (\mathfrak{D}(\tau))^{\{x \in \mathbb{N} \mid x < n\}}$, that is, $\mathfrak{D}(\text{Sequence}(\tau))$ is the set of functions from the set of the first n natural numbers into $\mathfrak{D}(\tau)$.
- Methods are treated as ordinary functions, which are curried for sake of simplicity. Given a function of $\tau \rightarrow \zeta$, define $\mathfrak{D}(\tau \rightarrow \zeta) \stackrel{\text{def}}{=} \mathfrak{D}(\zeta)^{\mathfrak{D}(\tau)} \cup \{\mathcal{J}_{\mathfrak{D}(\zeta)^{\mathfrak{D}(\tau)}}\}$. ◇

The type interpretation assigns to each type defined in OCL an element of U_{ω} in order to interpret that type. For example, we assign to the type $\text{OclAny} \rightarrow \text{Boolean}$, which represents the type of methods defined in class *OclAny* that do not accept any further arguments and return a boolean value, the set $\mathbb{B}_{\mathcal{J}}^{\mathbb{O}_{\mathcal{J}}}$.

2.3 Object Constraint Language

For any function $f : A \rightarrow B$ we define its *variant* $f\{y \mapsto z\}$, where $y \in A$ and $z \in B$, to be:

$$f\{y \mapsto z\}(x) \stackrel{\text{def}}{=} \begin{cases} f(x) & , \text{ if } x \neq y \\ z & , \text{ if } x = y \end{cases}$$

and write $f\{y_1 \mapsto z_1, y_2 \mapsto z_2\}$ for $f\{y_1 \mapsto z_1\}\{y_2 \mapsto z_2\}$, provided $y_1 \neq y_2$. The empty function, that is, the function which is not defined for any argument, is written as ϵ .

For defining the semantics of an OCL expression we further assume an interpretation function $\mathcal{I} : U_\omega \rightarrow \Sigma \rightarrow \mathcal{N} \rightarrow U_\omega \rightarrow U_\omega$ ⁷ which provides the meaning of the primitive functions defined in the standard library and the methods defined in the class diagram.⁸ The intuitive meaning of $\mathcal{I}$ is:

1. For some $o \in \mathbb{O}$ (recall that $\mathbb{O} \subseteq U_\omega$, here, we restrict to objects) the expression $\mathcal{I}(o)$ represents a function which assigns to each operation name $\mathcal{N}$ defined in the class of o a signature. In classic implementation of object-oriented programming languages this represents the so-called *virtual table*, which is used to bind operations to its method.
2. Because the state of an object is defined as part of the global state of the system, the expression $\mathcal{I}(o)(\sigma)$ provides the state in which the method is to be evaluated.
3. The expression $\mathcal{I}(o)(\sigma)(n)$ represents a function, that is, an element of U_ω , taking the arguments provided to the operation call and returning some value. These arguments do *not* include the *self* parameter, whose value is o and can be considered to be already bound because it was originally used to obtain the virtual table. Furthermore, the function is required to be type-consistent.
4. The function $\mathcal{I}(o)(\sigma)(n)$ is assumed to result $\mathcal{J}_T$, where T is the declared result type of the operation, if this function is either undefined for its argument, or if it is diverging.
5. If n is defined in OCL using the **def** stereotype, then $\mathcal{I}(o)(\sigma)(n)(v)$ is required to be the evaluation of the OCL expression associated to n , where the formal parameters are interpreted with the values of the actual arguments given to the call.

This interpretation function is not only used to encode late-binding of the self parameter. Another purpose of this function is to provide an *oracle* which evaluates all calls and functions which are not defined in OCL. Here, we do not define an operational

⁷The signature of $\mathcal{I}$ is actually $\mathcal{I} : U_\omega \rightarrow \Sigma \rightarrow \mathcal{N} \rightarrow U_\omega$. However, we want to emphasise that the result of $\mathcal{I}(o)(\sigma)(m)$ is supposed to be a method whose interpretation is $\mathcal{I}(o)(\sigma)(m) : U_k \rightarrow U_\ell$ for some $k, \ell \in \mathbb{N}$. Then there also exists an i such that $i > k$ and $i > \ell$ with $\mathcal{I}(o)(\sigma)(m) : U_i$. In order to simplify the notation, we write $U_\omega \rightarrow U_\omega$.

⁸The interpretation of the meaning of primitive operations provided in the OCL standard library is defined in Section 2.3.4.

semantics the possible languages in which methods and operations are defined or all usable behavioural notations of UML. All these notations may be used to define the meaning of operations which can be invoked from OCL expressions.⁹ Because these calls may not terminate, the interpretation function is also an oracle which can decide termination. As a consequence, the abstract machine for OCL constraints in the following definition cannot be implemented.

In the next definition, we need the following auxiliary function, which is used to interpret iterate expressions, where $X \in U_\omega$, $x \in 2^X$, $f : X \rightarrow Y \rightarrow Y$ (f is used to compute the next accumulator value from the currently chosen element and the current accumulator value) for some $Y \in U_\omega$ and $a \in Y$:

$$\text{iterate}(x, a, f) \stackrel{\text{def}}{=} \begin{cases} \mathcal{J}_{2^x} & , \text{ if } x \text{ is not finite} \\ a & , \text{ if } x \text{ is empty} \\ \text{iterate}(x \setminus \{\varepsilon y : y \in x\}, & , \text{ otherwise} \\ \quad f(\varepsilon y : y \in x, a), f) \end{cases}, \quad (2.9)$$

where the term $\varepsilon x : P(x)$ uses to Hilbert's ε -function, which selects a x satisfying $P(x)$, or is not defined if no such x exists (see, for example, Bourbaki [15]). Similar definitions can be given for multi-sets and sequences.

Now we define the abstract machine used to interpret an OCL constraint. This abstract machine is recursively defined by a function

$$\text{eval} : \mathcal{D} \times \Sigma \times \Sigma \times (\mathcal{V} \rightarrow \mathbb{V}) \rightarrow \mathcal{L}_{\text{OCL}} \rightarrow \mathbb{V}.$$

In any invocation of $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\varphi)$ the parameter D defines the class diagram over which the constraint is to be interpreted, σ and $\overleftarrow{\sigma}$ are two object diagrams over D forming an action, η is a valuation of the local variables, and φ is the constraint to be interpreted. Furthermore, we assume that all types of OCL and its operations are defined in D , and that the constraint is well-typed in D , as described in Chapter 3.

Definition 2.19 (Semantics of Expressions). Let D be a class diagram and let $\overleftarrow{\sigma} = \langle \overleftarrow{\tau}, \overleftarrow{\xi}, \llbracket \cdot \rrbracket \rangle$ and $\sigma = \langle \tau, \xi, \llbracket \cdot \rrbracket \rangle$ be two object diagrams forming an action, such that $\overleftarrow{\sigma} \models D$ and $\sigma \models D$, and $\mathcal{I} : U_\omega \rightarrow \Sigma \rightarrow \mathcal{N} \rightarrow U_\omega \rightarrow U_\omega$ be a function interpreting all methods defined in the class diagram (which contains the OCL standard library). Then the semantics of OCL expressions is defined by:

1. $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{true}) \stackrel{\text{def}}{=} \text{true}$ and $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{false}) \stackrel{\text{def}}{=} \text{false}$.
 $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{null}) \stackrel{\text{def}}{=} \text{null}$
2. For any numerical literal ℓ define $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\ell) \stackrel{\text{def}}{=} \ell$.

⁹The only constraint imposed is that evaluating the call is free from side-effects, that is, it does not change the state in which it is evaluated.

2.3 Object Constraint Language

3. For collection literal expressions define:

a) For *Bag*, define

$$\begin{aligned} \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{Bag}\{e_1, e_2, \dots, e_n\}) &\stackrel{\text{def}}{=} \\ &\wr \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e_i) \mid 1 \leq i \leq n \text{ and} \\ &\quad e_i \text{ is not a range expression} \wr \cup \\ &\bigcup \wr \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e_i) \mid 1 \leq i \leq n \text{ and} \\ &\quad e_i \text{ is a range expression} \wr \end{aligned}$$

where

$$\begin{aligned} \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e..e') &\stackrel{\text{def}}{=} \\ &\wr i \mid \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e) \leq i \leq \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e') \wr \end{aligned}$$

b) For *Set*, the definition is analogous.

c) For *Sequence*, define

$$\begin{aligned} \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{Sequence}\{\}) &\stackrel{\text{def}}{=} \langle \rangle \\ \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{Sequence}\{e_0, e_1, \dots, e_n\}) &\stackrel{\text{def}}{=} \langle \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e_0) \rangle \cdot \\ &\quad \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{Sequence}\{e_1, \dots, e_n\}) \\ \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e..e') &\stackrel{\text{def}}{=} \text{range}(\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e), \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(e')) \end{aligned}$$

where,

$$\text{range}(x, y) \stackrel{\text{def}}{=} \begin{cases} \langle \rangle & , \text{ if } x > y \\ \langle x \rangle \cdot \text{range}(x + 1, y) & , \text{ otherwise.} \end{cases}$$

4. For any $v \in \mathcal{V}$ define $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(v) \stackrel{\text{def}}{=} \eta(v)$. Observe that type checking guarantees that this value is always defined, as demonstrated in Chapter 3.

5. For if-expressions:

$$\begin{aligned} \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{if } t \text{ then } t_1 \text{ else } t_2 \text{ endif}) &\stackrel{\text{def}}{=} \\ &\begin{cases} \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_1) & , \text{ if } \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) = \text{true} \\ \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_2) & , \text{ if } \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t) = \text{false} \\ \mathcal{J} & , \text{ otherwise.} \end{cases} \end{aligned}$$

6. $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{let } v = t \text{ in } t_1) \stackrel{\text{def}}{=} \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta\{v \mapsto \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)\})(t_1)$.
7. For attribute access expressions and navigation expressions, we define:
 - a) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.n) \stackrel{\text{def}}{=} \xi(\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t))(n)$ if t evaluates to an object and n is an attribute name, otherwise:
 - b) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.n) \stackrel{\text{def}}{=} \bigcup \xi(o)(n) \mid o \in \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)$, if t evaluates to a bag of objects and n is an attribute name, otherwise:
 - c) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.n) \stackrel{\text{def}}{=} \llbracket \text{self}.n \rrbracket(\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t))$, if t evaluates to an object and n is an association end name, otherwise:
 - d) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.n) \stackrel{\text{def}}{=} \bigcup \bigcup o' \mid \exists r : \exists f : f \in \llbracket r \rrbracket \wedge \exists n' : f(n') = o \wedge f(n) = o' \mid o \in \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)$, if t evaluates to a bag of objects and n is an association end name, otherwise:
 - e) the result is $\mathcal{J}$.

For sets and sequences define the evaluation function analogously to Item 7b, respectively, Item 7d.

8. For previous attribute access expressions and previous navigation expressions, we define:
 - a) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.n@pre) \stackrel{\text{def}}{=} \overleftarrow{\xi}(\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t))(n)$ if t evaluates to an object and n is an attribute name, otherwise:
 - b) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.n@pre) \stackrel{\text{def}}{=} \bigcup \overleftarrow{\xi}(o)(n) \mid o \in \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)$, if t evaluates to a bag of objects and n is an attribute name, otherwise:
 - c) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.n@pre) \stackrel{\text{def}}{=} \overleftarrow{\llbracket \text{self}.n \rrbracket}(\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t))$, if t evaluates to an object and n is an association end name, otherwise:
 - d) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.n@pre) \stackrel{\text{def}}{=} \bigcup \bigcup o' \mid \exists r : \exists f : f \in \overleftarrow{\llbracket r \rrbracket} \wedge \exists n' : f(n') = o \wedge f(n) = o' \mid o \in \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)$, if t evaluates to a bag of objects and n is an association end name, otherwise:
 - e) the result is $\mathcal{J}$.

For sets and sequences define the evaluation function analogously to Item 8b, respectively, Item 8d.

Observe that we use $\overleftarrow{\sigma}$ instead of σ as the object diagram for obtaining the value of the attribute or navigation, which is the only difference to Item 7.

9. For method call expressions, define:
 - a) If $v = \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)$ is an object and $v_i = \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_i)$ for all $1 \leq i \leq n$, then $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.m(t_1, t_2, \dots, t_n)) \stackrel{\text{def}}{=} I(v)(\sigma)(m)(v_1, v_2, \dots, v_n)$, and otherwise:

2.3 Object Constraint Language

- b) If $v = \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)$ is a bag of objects and $v_i = \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_i)$ for all $1 \leq i \leq n$, then $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t.m(t_1, t_2, \dots, t_n)) \stackrel{\text{def}}{=} \bigcup \{x \mid o \in v \wedge x = \mathcal{I}(o)(\sigma)(m)(v_1, v_2, \dots, v_n)\}$.

For sets and sequences define the evaluation function analogously to Item 9b.

Recall, that $\mathcal{I}(o)(\sigma)(n)$ is always defined, but the value returned by this function may be undefined, which also models divergence. Also, the assumption that the expression is well-typed implies that $\mathcal{I}(o)(\sigma)(n)$ is never undefined.

10. Define $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t \rightarrow m(t_1, t_2, \dots, t_n)) \stackrel{\text{def}}{=} \mathcal{I}(v)(\sigma)(m)(v_1, v_2, \dots, v_n)$, where $v = \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)$ and $v_i = \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_i)$, for $1 \leq i \leq n$ for collection call expressions.

11. For iterate expressions, define:

- a) $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t \rightarrow \text{iterate}(v_1 : T_1; a = t_a \mid t_0)) \stackrel{\text{def}}{=} \text{iterate}(v_t, v_a, f)$, where

$$v_t = \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)$$

$$v_a = \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t_a)$$

$$f = \lambda v_x. \lambda v_a. \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(v_1 \mapsto v_x, a \mapsto v_a)(t_0) \quad ,$$

otherwise:

- b) For more than one iteration variable, define $t \rightarrow \text{iterate}(v_1 : T_1, \dots, v_n : T_n; a = t_a \mid t_0) \stackrel{\text{def}}{=} t \rightarrow \text{iterate}(v_1 : T_1, \dots, v_{n-1} : T_{n-1}; a = t_a \mid t \rightarrow \text{iterate}(v_n : T_n; a = t_a \mid t_0))$, that is, we recursively transform an iterate expression with many iterator variables into a nested iterate expression, where each iterator expression only iterates over one iterator variable.

If t evaluates to a sequence or set, then the rule under Item 11a has to be adapted, accordingly.

Recall, that iterate results in $\mathcal{J}_{\mathcal{D}(\tau)}$, where τ is the type of t , if the iteration is unbounded.

12. For flatten expressions $t \rightarrow \text{flatten}()$ let $v = \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t)$. Then:

$$\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t \rightarrow \text{flatten}()) \stackrel{\text{def}}{=} \text{flatten}(v) \quad ,$$

if v denotes a collection, that is, either a set, a bag, or a sequence. The function $\text{flatten}(v)$ is the union over all elements of a set, if v is a set, the bag-union over all elements of a bag, if v is a bag, and sequence concatenation of all members of a sequence, if v is a sequence. Otherwise, $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(t \rightarrow \text{flatten}()) \stackrel{\text{def}}{=} v$.

13. Define $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(T.\text{allInstances}()) \stackrel{\text{def}}{=} \{o \mid \tau(o) = T\}$ for all instances of T expressions and $\text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(T.\text{allInstances}@pre()) \stackrel{\text{def}}{=} \{o \mid \overleftarrow{\tau}(o) = T\}$ for all previous instances of T . $\diamond$

The semantics for dereferencing attributes or navigating objects, as described in Item 7 is not straight-forward and not very consistent. For navigation expressions this semantics is not flattening, as expressed by the third case. Accessing an attribute of a collection occurs by some kind of map function, that is, if the source (t) represents a collection, then $t.a$ represents the collection of attribute values. Only if t is an object, $t.a$ results in a value. The interpretation of such an expression is, indeed, dependent on the semantic value of the source.

The intuition of the interpretation of an iterate expression, as described in Item 11 is, that the expression t_0 , in which both a and the iteration variable v_1 may occur, is evaluated for each element of the collection described by t . The result of the interpretation of t_0 then becomes the new value of a and a next element is selected.

We observe that the abstract machine we have defined is similar to an abstract machine interpreting a typed functional programming language. This implies that the evaluation of a constraint may diverge because of recursive definitions. If the evaluation diverges for an expression, its semantic value is undefined, written $\mathcal{J}$.

The most notable difference between OCL and a functional programming language is, that in operation call expressions the operation to be called is bound late, that is, the actual function evaluated depends on the *type* of the callee. The motivation of this interpretation was, that OCL is supposed to be usable for programmers of object-oriented programming languages, and, therefore, should behave like one.

Definition 2.20 (Semantics of Context). Let D be a class diagram and $\sigma = \langle \tau, \xi, \llbracket \cdot \rrbracket \rangle \in \Sigma$ and $\overleftarrow{\sigma} = \langle \overleftarrow{\tau}, \overleftarrow{\xi}, \overleftarrow{\llbracket \cdot \rrbracket} \rangle \in \Sigma$ be two object diagrams, such that $\sigma \models D$ and $\overleftarrow{\sigma} \models D$. Then the semantics of a constraint, that is, an OCL expression of type boolean in D , in a context is defined by:

1. For any classifier c define:

$$\begin{aligned} \text{eval}(D, \epsilon, \sigma, \eta)(\text{context } c \text{ inv} : t) &\stackrel{\text{def}}{=} \\ \forall o \in \mathbb{O} : \tau(o) = c &\implies \text{eval}(D, \sigma, \epsilon, \eta\{self \mapsto o\})(t) \end{aligned}$$

2. For any operation context $c :: m(\vec{v})$ and any action $\langle \overleftarrow{\sigma}, \sigma \rangle$, which corresponds to a call of m , such that $\vec{v} \in \text{dom}(\eta)$, define:

$$\begin{aligned} \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta)(\text{context } c :: m(\vec{v}) \text{ pre} : t \text{ post} : t') &\stackrel{\text{def}}{=} \\ \forall o \in \mathbb{O} : \tau(o) = c &\implies (\text{eval}(D, \epsilon, \overleftarrow{\sigma}, \eta\{self \mapsto o\})(t) \\ \wedge E(\xi(o)(\text{result})) &\implies \text{eval}(D, \overleftarrow{\sigma}, \sigma, \eta\{self \mapsto o\})(t')). \end{aligned}$$

$\diamond$

2.3 Object Constraint Language

A class invariant of class c is true in an object diagram, if it holds for all instances of c . Note that a class invariant is not required to hold for instances of the subclasses of c . This implies that class invariants are not inherited. Also observe that while the evaluation function accepts a valuation of local variables η as a parameter, no local variable (with the exception of *self*) may occur freely in the invariant. Therefore, the result of eval is independent from the choice of η .

A constraint on an operation holds, if, for all actions $\langle \overleftarrow{\sigma}, \sigma \rangle$ it holds for all classes c which have executed that operations, which is similar to the interpretation of postconditions in VDM (see Jones [75]). The successful termination of an operation is indicated by $E(\xi(o)(result))$, where *result* is treated as a special attribute in the object diagram. It must not be an attribute for any class in the class diagram, because otherwise Definition 2.14 would be violated. Therefore, we require $\sigma \models D$, which implies that *result* is always undefined in the object diagram. On the other hand, a variable is needed in the system's state, where the result of the operation is stored, as suggested by de Roeper and Engelhardt [46, p. 314f.]. To simplify the formalisation of class diagrams, object diagrams, and the relation between them, we specifically exclude *result* from the attribute and introduce it as a *logical attribute*, which is undefined in the state the operation is invoked and carries the value of the operation's result when the operation has terminated. Also, the actual parameters supplied to an operation are treated like logical attributes for the same reason. Additionally, the parameters of an operation are *immutable* during the execution of an operation, that is, they are like logical variables in Hoare logic.

Observe that the reason for choosing only objects that exist in the post-state to check whether a pre-post specification holds, are two-fold. If an object is newly created, it does not yet exist in the state of the pre-condition. The constructor method may, however, accept parameters. If the conjunction would range over the objects which don't exist in the pre-state, then the constructor methods would not be constrained. Secondly, objects may also be destroyed. Their identities do not occur in the post-state, hence no constraint is imposed on destructor methods. Indeed, the post condition can only be true, because the object does not exist anymore, once the method terminates. Because we do not allow explicit invocation of the destructor method, it is sufficient to assume that the pre-condition is true, too.

If there are an infinite number of objects defined, then in OCL all constraints are evaluated to undefined. Because OCL only expresses safety properties, it is sufficient to consider all finite prefixes of a computation, where, if the system starts with a finite number of objects in the initial state, only a finite number of objects exist, because with every step of the system, only one action is performed, which implies that at most one object can be created with each action. Therefore, the quantification over objects of a class is always a quantification over a finite set of objects.

In UML it is a semantic variation point what the meaning of a violated constraint is. One may choose to assign an operational meaning to constraints, like in Eiffel [97]. There, the violation of a behavioural specification causes the system to raise an excep-

tion, which may be handled by the program.

Since we are interested in the *correctness* of the model, a violated constraint just means, that the system is incorrect, or “ill-defined” in UML’s terms.

2.3.4 OCL Standard Library

In the preceding section we have defined an interpreter for a core of OCL. This core of OCL is sufficient to encode many primitives needed for specifying systems. On the other hand, using this core is inconvenient. In this section we describe the OCL standard library, which defines commonly used types (or classes) and their operations. We define the signature of a subset of these operations together with their semantics.

We only describe a subset of the standard library. For a complete description of the standard library we refer to the OCL standard [113, Chapter 11].

Types and its Operation

Some expressions in OCL do not specify values but types. We usually write T in the signature instead of $x : T$ to indicate that this parameter is expected to be a type. Formally, we use the mechanisms for parametric classes, which are *not* used in the OCL specification.

Each expression evaluating to a type specifies one operation:

- *allInstances()* : *Set(T)*, where T is the type of the “callee”. This operation returns all instances of the type described by the callee which exist in the current state. The OCL standard specifies that this operation may only be used if the result is a *finite* set. *Integer.allInstances()* does not result in a finite set, but if the callee describes a class, it does.

OclInvalid, the Bottom Type

OclInvalid is a type that is a subtype of any other type. Since a type hierarchy is a partially ordered set, the types *OclInvalid* and *OclAny* (defined below) complete the subset of ground types to a complete lattice. Instances of this type represent exceptional situations like division by zero or divergence of the interpreter.

When any operation is invoked on instances of *OclInvalid* the result will be an instance of type *OclInvalid*. The only exception are *oclIsInvalid()* and *oclIsUndefined()*, which return true when invoked on instances of *OclInvalid*.

OclVoid, the Empty Type

The type *OclVoid* is a type which the standard requires to be a subtype of any other type.

2.3 Object Constraint Language

The standard introduces many complications with *OclInvalid* and its literal, *OclVoid* and its literal null, and actually requiring that null is an instance of *OclVoid*. For the purposes of this thesis we require that *OclVoid* is the uninhabited type, of which no object or value is an instance. The role of *OclVoid* coincides with the unit type found in functional programming languages [126]. Operations cannot accept arguments of type *OclVoid*, but they may return *OclVoid*, which causes the interpreter of OCL to return a “token” notifying the caller that the operation has been completed. The choice of this token is arbitrary and is not observable in OCL, except for the fact that the caller continues evaluating its operations.

As a consequence, null, which represents the empty or missing object, becomes an instance of every type. Invoking any operation on null results in a value of type *OclInvalid*, or a “dereferencing null error”. The only exception to this rule is the equality operator =, which returns true, if both operands are null, and false otherwise.

Observe that this definition of *OclVoid* and null is different from the definition in the OCL standard [113, p. 153]. Our definition is identical to the one used in virtually all (object-oriented) programming languages [56, 146, 57]. This alternative definition vastly simplifies the semantics of OCL and also makes it more usable (see Section 2.3.5).

OclAny, the Top Type

OclAny is the implicit super-type of all types occurring in the model, the UML specification, and the OCL standard library. An exception are parameterised types (compare to Section 2.1.3), which include OCL’s collection types. The type *OclAny* defines the following operations:

- $= (x : OclAny) : Boolean$ represents equality between objects. It evaluates to true if *self* is the same as *x*. A more precise characterisation of equality is given in Section 2.3.5.
- $<> (x : OclAny) : Boolean \stackrel{\text{def}}{=} not\ a = b$ represents inequality between objects.
- *oclIsInvalid()* : *Boolean* is evaluated to true whenever the callee is an instance of *OclInvalid*, and otherwise to false.
- *oclIsUndefined()* : *Boolean* evaluates to true, if the callee is undefined. In the standard, this means that the callee is either equal to null or is an instance of *OclInvalid*.
- *oclIsNew()* : *Boolean* $\stackrel{\text{def}}{=} self@pre.oclIsUndefined()$ can only be used in a post-condition and states that the object has been newly created during the execution of an operation.
- *oclAsType(T)* : *T* is a “cast” or “retyping” expression, evaluating to the value of the callee, if it is an instance of *T*, and an instance of *OclInvalid* otherwise.

- *oclIsTypeOf(T)* : *Boolean* evaluates to true if the callee is an instance of type *T*.
- *oclIsKindOf(T)* : *Boolean* evaluates to true if the callee is an instance of type *T* or one of *T*'s subtypes, that is, the callee conforms to the type *T* (see Section 3.2).
- *oclInState(S)* : *Boolean* evaluates to true if the callee is in the state *S*, where *S* specifies a state of a state machine (see Section 2.4).

OclMessage

OclMessage is a parameterised type where its type parameter *T* ranges over operation and signal specifications. Instantiating the type parameter results in a concrete type usable in specifications.

OclMessage defines the following operations:

- *hasReturned()* : *Boolean* returns true if the type parameter *T* is an operation specification and the called operation has returned a value. In all other cases, this operation returns false.
- *result()* : *S* returns the value returned by an operation, if the operation has already returned a value, and an instance of *OclInvalid* in any other case. The return type *S* refers to the return type of the operation specification *T*.
- *isSignalSent()* : *Boolean* is true if *T* is a signal specification and false otherwise.
- *isOperationCall()* : *Boolean* is true if *T* is an operation specification and false otherwise.

For a precise semantics of these constructs we refer to Section 5.3.3.

Boolean and Kleene Logic

The standard type *Boolean* represents the truth values. It defines the standard connectives as operations, which accept two instances of *Boolean*, except for *not*, which only accepts one. These operations may be written using infix notation instead of the operation call notation.

The semantics of these operations is based on the one of Kleene's logic [77]. Table 2.2 displays the truth table for interpreting these operations.

In OCL logical equivalence is written using $=$. For clarity, we write *iff* for logical equivalence in this thesis. The reasons for this are that

1. The semantics of OCL treats boolean connectives special. They are not necessarily strict in their arguments. For example, using a strict interpretation, the OCL expression *false implies a* should be $\perp$ if $a = \perp$, similarly for *a implies true*.

2.3 Object Constraint Language

a	b	$not\ b$	$a\ and\ b$	$a\ or\ b$	$a\ implies\ b$	$a\ iff\ b$	$a\ xor\ b$
true	true	false	true	true	true	true	false
true	false	true	false	true	false	false	true
true	$\perp$	$\perp$	$\perp$	true	$\perp$	$\perp$	$\perp$
false	true	false	false	true	true	false	true
false	false	true	false	false	true	true	false
false	$\perp$	$\perp$	false	$\perp$	true	$\perp$	$\perp$
$\perp$	true	false	$\perp$	true	true	$\perp$	$\perp$
$\perp$	false	true	false	$\perp$	$\perp$	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Table 2.2: Semantics of boolean connectives

2. The semantics of $=$ is not precisely defined for all types occurring in a model by the standard, but for *Boolean* we have a precise definition for $=$.
3. It removes an inconsistency, because OCL introduces *xor*, which is semantically equivalent to $<>$.

Real

The standard data type *Real* represents real-valued quantities. This class defines the usual mathematical connectives with their standard semantics. We enumerate these connectives in order to provide their signature, which is used for Chapter 3.

- $+(x : Real) : Real$ describes the sum of *self* and x .
- $-(x : Real) : Real$ describes the difference between *self* and x .
- $*(x : Real) : Real$ describes the product of *self* and x .
- $/(x : Real) : Real$ describes the quotient of *self* and x .
- $-() : Real \stackrel{\text{def}}{=} 0 - self$ describes the negation of *self*.
- $<(x : Real) : Boolean$ evaluates to true, if the value of *self* is less than the value of x . It evaluates to false, if the value of *self* is equal to or greater than the value of x . In any other case, it is undefined.
- $>(x : Real) : Boolean$ evaluates to true, if the value of *self* is greater than the value of x . It evaluates to false, if the value of *self* is less than or equal to the value of x . In any other case, it is undefined.

- $\leq (x : Real) : Boolean$ evaluates to true, if the value of *self* is less than or equal to the value of *x*. It evaluates to false, if the value of *self* is greater than the value of *x*. In any other case, it is undefined.
- $\geq (x : Real) : Boolean$ evaluates to true, if the value of *self* is equal to or greater than the value of *x*. It evaluates to false, if the value of *self* is less than the value of *x*. In any other case, it is undefined.
- $abs() : Real \stackrel{\text{def}}{=} \text{if } self < 0 \text{ then } -self \text{ else } self \text{ endif}$ describes the absolute value of *self*.
- $floor() : Integer$ describes the largest integer which is not greater than *self*.
- $round() : Integer \stackrel{\text{def}}{=} (self + 0.5).floor()$ rounds *self* to the nearest integer.
- $max(x : Real) : Real \stackrel{\text{def}}{=} \text{if } self < x \text{ then } x \text{ else } self \text{ endif}$ results in the greater value of *self* and *x*.
- $min(x : Real) : Real \stackrel{\text{def}}{=} \text{if } self > x \text{ then } x \text{ else } self \text{ endif}$ results in the smaller value of *self* and *x*.

Integer

The standard data type *Integer* represents integer-valued quantities and is a sub-class of *Real*. This class defines the usual mathematical connectives with their standard semantics. We enumerate these connectives in order to provide their signature, which is used for Chapter 3.

- $+(x : Integer) : Integer$ describes the sum of *self* and *x*.
- $-(x : Integer) : Integer$ describes the difference between *self* and *x*.
- $*(x : Integer) : Integer$ describes the product of *self* and *x*.
- $/(x : Integer) : Real$ describes the quotient of *self* and *x*.
- $-() : Integer \stackrel{\text{def}}{=} 0 - self$ describes the negation of *self*.
- $< (x : Integer) : Boolean$ evaluates to true, if the value of *self* is less than the value of *x*. It evaluates to false, if the value of *self* is equal to or greater than the value of *x*. In any other case, it is undefined.
- $> (x : Integer) : Boolean$ evaluates to true, if the value of *self* is greater than the value of *x*. It evaluates to false, if the value of *self* is less than or equal to the value of *x*. In any other case, it is undefined.

2.3 Object Constraint Language

- $\leq (x : Integer) : Boolean$ evaluates to true, if the value of *self* is less than or equal to the value of *x*. It evaluates to false, if the value of *self* is greater than the value of *x*. In any other case, it is undefined.
- $\geq (x : Integer) : Boolean$ evaluates to true, if the value of *self* is equal to or greater than the value of *x*. It evaluates to false, if the value of *self* is less than the value of *x*. In any other case, it is undefined.
- $abs() : Integer \stackrel{\text{def}}{=} \text{if } self < 0 \text{ then } -self \text{ else } self \text{ endif}$ describes the absolute value of *self*.
- $div(x : Integer) : Integer$ describes Euclidean division of *self* by *x*, that is, it results in the unique integer *y* such that there exists an $0 \leq r < x$ with $y * x + r = self$.
- $mod(x : Integer) : Integer$ describes Euclidean remainder of *self* divided by *x*, that is, it results in the unique integer $0 \leq r < x$ such that there exists an integer *y* with $y * x + r = self$.
- $max(x : Integer) : Integer \stackrel{\text{def}}{=} \text{if } self < x \text{ then } x \text{ else } self \text{ endif}$ evaluates to the greater value of *self* and *x*.
- $min(x : Integer) : Integer \stackrel{\text{def}}{=} \text{if } self > x \text{ then } x \text{ else } self \text{ endif}$ evaluates to the smaller value of *self* and *x*.

Observe that the arithmetic operators have been overloaded to assert that the result is an integer. Overloading $/$ is redundant, because the version defined in *Real* is the same operator.

String, Unlimited Integer

The string data type and the unlimited integer data type are not used in this thesis. We refer to the OCL standard [113, Section 11.5.3 and Section 11.4.5] for a definition of these data types and their semantics.

Collection

The type *Collection* (and its subtypes) is a parameterised type¹⁰, with the type parameter *T* specifying the type of the elements. A concrete type can be obtained by instantiating the template parameter. For example, *Set* is a parameterised type (which cannot be instantiated), while *Set(Integer)* is; it is the type of a set of integers.

¹⁰See also Section 2.1.3 for parameterised classes and types.

Chapter 2 Introduction to Models of OCL and UML

Most operations on collections can be expressed using iterate expressions or iterator expressions. These expressions are defined in Definition 2.19, Item 11, and in the section on *predefined iterator expressions* below (pp. 49ff.)

The following operations are defined on *Collection*:

- $size() : Integer \stackrel{\text{def}}{=} self \rightarrow iterate(e; a = 0 \mid a + 1)$
- $count(e : T) : Integer \stackrel{\text{def}}{=} self \rightarrow iterate(i; a = 0 \mid \text{if } e = i \text{ then } a + 1 \text{ else } a \text{ endif})$
counts how often e occurs in the collection.
- $includes(e : T) : Boolean \stackrel{\text{def}}{=} self \rightarrow count(e) > 0$ returns true if and only if e occurs in the collection.
- $excludes(e : T) : Boolean \stackrel{\text{def}}{=} self \rightarrow count(e) = 0$ returns true if and only if e does not occur in the collection.
- $includesAll(c : Collection(T)) : Boolean \stackrel{\text{def}}{=} c \rightarrow forAll(e \mid self \rightarrow includes(e))$.
- $excludesAll(c : Collection(T)) : Boolean \stackrel{\text{def}}{=} c \rightarrow forAll(e \mid self \rightarrow excludes(e))$.
- $isEmpty() : Boolean \stackrel{\text{def}}{=} self \rightarrow size() = 0$.
- $notEmpty() : Boolean \stackrel{\text{def}}{=} self \rightarrow size() <> 0$.
- $sum() : T \stackrel{\text{def}}{=} self \rightarrow iterate(e; a = 0 \mid a + e)$. It is not possible to formally define this operation using the means provided by OCL, a formal and precise definition is given in Section 3.3.4.

Set

Set is the type of mathematical sets.

- $= (c : Set(T)) : Boolean$ describes set-equality.
- $including(e : T) : Set(T)$ describes the set obtained from *self* by including e .
- $excluding(e : T) : Set(T)$ describes the set obtained from *self* by excluding e .
- $union(c : Set(T)) : Set(T)$ describes the union of the set *self* and the set c .
- $union(c : Bag(T)) : Bag(T)$ describes the union of the bag obtained from *self* by assuming that each element of *self* occurs exactly once and the bag c .
- $intersection(c : Set(T)) : Set(T)$ describes the intersection of the set *self* the set c .

2.3 Object Constraint Language

- $intersection(c : Bag(T)) : Set(T)$ describes the intersection of the set $self$ and the set obtained from c by including every element contained in c .
- $-(c : Set(T)) : Set(T)$ describes the difference of the set $self$ the set c .
- $flatten() : Set(T')$. If $self$ is a set of collections, then this operation returns the set-union of all its elements. Otherwise the result is the value of $self$. It is not possible to formally define this operation using the means provided by OCL; a formal and precise definition is given in Definition 2.19, Item 12, and in Section 3.3.5.
- $asSet() : Set(T) \stackrel{\text{def}}{=} self \rightarrow iterate(e; a = Set\{\} \mid a \rightarrow including(e))$ is the identity function and only defined for symmetry reasons.
- $asBag() : Bag(T) \stackrel{\text{def}}{=} self \rightarrow iterate(e; a = Bag\{\} \mid a \rightarrow including(e))$ returns a bag which includes each element of $self$ exactly once.
- $asSequence() : Sequence(T)$ returns a sequence containing all elements of $self$ exactly once. The order of the elements is arbitrary. It is equivalent to the expression

$$self \rightarrow iterate(e; a = Sequence\{\} \mid a \rightarrow append(e)) \quad .$$

Bag

Bag is the type of a bag or multi-set. Bags usually arise as the result of navigation expressions.

- $= (c : Bag(T)) : Boolean$ describes equality of multi-sets.
- $including(e : T) : Bag(T)$ describes the bag obtained from $self$ by including e .
- $excluding(e : T) : Bag(T)$ describes the bag obtained from $self$ by excluding all occurrences of e .
- $union(c : Bag(T)) : Bag(T)$ describes the union of the bag $self$ and the bag c .
- $union(c : Set(T)) : Bag(T)$ describes the union of the bag $self$ and the bag obtained from c by including each element of c exactly once.
- $intersection(c : Set(T)) : Set(T)$ describes the intersection of the bag $self$ and the set c .
- $intersection(c : Bag(T)) : Bag(T)$ describes the intersection of the bag $self$ and the bag c .

- $-(c : Set(T)) : Set(T)$ describes the bag obtained from *self* by removing all occurrences of the elements of *c*.
- $flatten() : Bag(T')$. If *self* is a bag of collections, then this operation returns the bag.union of all its elements. Otherwise the result is the value of *self*. It is not possible to formally define this operation using the means provided by OCL; a formal and precise definition is given in Definition 2.19, Item 12, and in Section 3.3.5.
- $asSet() : Set(T) \stackrel{\text{def}}{=} self \rightarrow iterate(e; a = Set\{\} \mid a \rightarrow including(e))$ returns a set which contains each element of *self*.
- $asBag() : Bag(T)$ is the identity function and only defined for symmetry reasons.
- $asSequence() : Sequence(T)$ returns a sequence containing all elements of *self* as often as they occur in the multi-set. The order of the elements is arbitrary. It is equivalent to:

$$self \rightarrow iterate(e; a = Sequence\{\} \mid a \rightarrow append(e)) \quad .$$

Sequence

Sequence is the type of mathematical sequences.

- $at(i : Integer) : T$ results in the element at the *i*th position of the sequence.
- $= (c : Sequence(T)) : Boolean \stackrel{\text{def}}{=} \mathbf{let} \ s = self \rightarrow size() \ \mathbf{in} \ s = c \rightarrow size() \ \mathbf{and} \ Sequence\{1..s\} \rightarrow \mathbf{forAll}(i : Integer \mid self \rightarrow at(i) = c \rightarrow at(i))$ describes equality of sequences.
- $union(c : Sequence(T)) : Sequence(T)$ describes the concatenation of *self* and *c*.
- $flatten() : Sequence(T')$. If *self* is a sequence of collections, then this operation returns the sequence concatenation of all its elements. Otherwise the result is the value of *self*. It is not possible to formally define this operation using the means provided by OCL; a formal and precise definition is given in Definition 2.19, Item 12, and in Section 3.3.5.
- $append(e : T) : Sequence(T) \stackrel{\text{def}}{=} self \rightarrow union(Sequence\{e\})$ results in the sequence which consists of all elements of *self* with *e* appended.
- $prepend(e : T) : Sequence(T) \stackrel{\text{def}}{=} Sequence\{e\} \rightarrow union(self)$ results in the sequence which consists of all elements of *self* with *e* prepended.
- $excluding(e : T) : Sequence(T) \stackrel{\text{def}}{=} self \rightarrow iterate(i; a = Sequence\{\} \mid \mathbf{if} \ e = i \ \mathbf{then} \ a \ \mathbf{else} \ a \rightarrow append(i) \ \mathbf{endif})$ results in the largest sub-sequence of *self*, in which *e* does not occur.

2.3 Object Constraint Language

- $subSequence(l : Integer, u : Integer) : Sequence(T)$ results in the subsequence of $self$ starting at index l and ending at index u . It is equivalent to:

$$Sequence\{1..u\} \rightarrow iterate(i; a = Sequence\{\} \mid \\ \text{if } l \leq i \text{ and } i \leq u \text{ then } a \rightarrow append(self \rightarrow at(i)))$$

- $insertAt(e : T, i : Integer) : Sequence(T) \stackrel{\text{def}}{=} (self \rightarrow subSequence(1, i - 1) \rightarrow append(e)) \rightarrow union(self \rightarrow subSequence(i, self \rightarrow size()))$ results in a sequence where e is inserted at position i .
- $first() : T \stackrel{\text{def}}{=} self \rightarrow at(1)$.
- $last() : T \stackrel{\text{def}}{=} self \rightarrow at(self \rightarrow size())$.
- $asSet() : Set(T) \stackrel{\text{def}}{=} self \rightarrow iterate(e; a = Set\{\} \mid a \rightarrow including(e))$ returns a set which contains each element of $self$.
- $asBag() : Bag(T) \stackrel{\text{def}}{=} self \rightarrow iterate(e; a = Bag\{\} \mid a \rightarrow including(e))$ returns a bag containing all elements of the sequence $self$.
- $asSequence() : Sequence(T)$ is the identity function and only defined for symmetry reasons.

Ordered Set

OrderedSet is the type of an ordered set, that is, a set with a linear order defined over its elements. Because we do not use this type in this thesis, we refer to [113, Sect. 1.7.3] for a description of the operations defined for this type and their semantics.

Predefined Iterator Expressions

Iterator expressions are not operations. However, the pre-defined iterator expressions are treated like operations of collection types. This section lists all predefined iterator types.

Quantification Quantification in OCL is expressed as an iterate expression. Indeed, the quantified expressions are defined in term of iterations:

1. $t \rightarrow forAll(v \mid t') \stackrel{\text{def}}{=} t \rightarrow iterate(v; a = true \mid a \text{ and } t')$
2. $t \rightarrow exists(v \mid t') \stackrel{\text{def}}{=} t \rightarrow iterate(v; a = false \mid a \text{ or } t')$

These iterate expressions describe the expansion of the quantifier into a conjunction or disjunction of the predicate t' for each element of the collection described by t .

Similar to the general iterate expression, a quantified expression may have more than one iterator variable. In this case the quantification is expanded analogously to Item 11b of Definition 2.19.

The expansion of quantifiers to iterators implies that if t represents an infinite set of values, then the meaning of the expression is undefined, because the evaluation of the iterate expression will not terminate. As an example and a consequence of the semantics of OCL, the expression $t \rightarrow \text{forAll}(v \mid \text{true}).\text{oclIsUndefined}()$ is testing the finiteness of t in OCL, whereas in traditional logic, where $\forall v : \text{true}$ is a tautology, and therefore this expression is always defined.

2.3.5 Critique of OCL

In the preceding section we have discussed the syntax and the semantics of OCL and its standard library, as described in the OCL standard and related publications. In this section we discuss the semantic decisions of the authors of the OCL standard and argue why it is necessary to define a semantics that is different from the one described in the standard. Such a semantics will be given in Section 4.5.3.

When OCL was designed, Warmer and Kleppe formulated the following requirements for OCL [154]:

1. OCL must be able to express extra (necessary) information on the models and other artifacts used in object-oriented development.
2. OCL must be a precise, unambiguous language that can easily be read and written by all practitioners of object technology and by their customers. This means that the language must be understood by people who are not mathematicians or computer scientists.
3. OCL must be a declarative language. Its expressions can have no side-effects; that is, the state of a system must not change because of [evaluating] an OCL expression. [...]
4. OCL must be a typed language. [...]

Expressiveness

If we suppose that all “artifacts” we are concerned with are present in the UML diagram, then OCL is indeed able to express extra information on the model, and as a consequence Requirement 1 is satisfied. Whether all extra information can be specified is not clear at all, because the meaning of “extra (necessary) information” has deliberately been unspecified.

2.3 Object Constraint Language

Indeed, the way we have formalised the standard library demonstrates that most concepts of that library can be encoded in a relatively small core. Only a few expressions have to be treated specially, mostly because of the object-oriented style.

Precise and Unambiguous

OCL fails on Requirement 2. In the following paragraphs we give a variety of reasons, of which most of them are technical. To summarize these reasons: OCL is a complex language with a complex semantics, which is only understandable by experts in the field.

Syntax It is hard to believe that OCL “can easily be read [...] by [...] customers”, that is, not by mathematicians and computer scientists. The syntax of OCL follows conventions of object-oriented programming languages, for example using a colon and an arrow for “sending a message”, and by this implying a message passing paradigm even in the specification language. Especially that quantification and iteration are written using a post-fix notation, whereas many operators of the standard data types are written in a more conventional infix notation, makes OCL very hard to read. The recent proposal by Jos Warmer of using an SQL-like syntax, which is much closer to natural languages, has, however, not yet found many supporters.

Semantics OCL also has to be understood by its users. In order to satisfy this requirement, one needs a precise and *simple* semantics which does not include surprises. But until now no consensus has been reached on how to interpret a specification in OCL among *computer scientists* and language designers, as the literature (see Section 2.5 for a selection of references) and the “OCL Issue List” [112] demonstrate.

Late Binding and Open Recursion Late binding (often called *dynamic dispatch* or *virtual binding*) and open recursion are the fundamental properties of object-oriented programming. By late binding we mean that the run-time type of the object decides how an operation call is to be interpreted. Open recursion means that a recursive call, here, the call of any operation on the object represented by *self*, is bound late.

Because OCL allows to call any side-effect free operation defined in a UML model, late binding has to be introduced into the semantics of OCL. Therefore, each OCL constraint is subject to the effect of the *fragile base class problem*. This means, that changes in a subclass can change the behaviour defined in the super-class in unexpected ways.

If late binding is used disciplined, either by avoiding *overriding*, that is, redefining the implementation of an operation, or *overloading*, that is, defining operations with the same name, but with a different signature, or by asserting that overriding and overloading coincide with behavioural subtyping, reasoning with OCL is not affected.

But in most models overriding is used in an ad-hoc fashion. The implementation of an operation is overridden with a new implementation which behaves completely different. For most modellers, adhering to Liskov’s substitution principle [94], that any instance of a sub-type can be used in place of an instance of its super-type, is less important than code-reuse.

In effect, the result of this semantics is, that in order to understand the meaning of a constraint, one has to understand the complete model and how it has been implemented. But we desire a formalism that abstracts from these details of the implementation.

Three-Valued Logic OCL has a three-valued logic. In itself a three-valued logic does not add much complexity to a specification language. The reasons for this are that each partial function f is extended in the semantics of OCL to a *strict* total function $f_{\perp} \stackrel{\text{def}}{=} \lambda x. \text{if } x \in \text{dom}(f) \text{ then } f(x) \text{ else } \perp$ and that relations and predicates are interpreted as strict functions to the three truth-values.

The undefined value also arises from the (implicit) requirement that each constraint should represent a *computable* function (see below how this requirement affects Requirement 3). But it is important to note what the additional truth value, undefined, is supposed to represent.

Usually, a formula whose meaning is undefined is considered to be nonsensical. In OCL, this is represented by calling such formulae and the models for which they have to hold, “ill-formed”. Any ill-formed model is considered to contain an error. Consequently, in most cases, a constraint that is undefined, can also be considered to be false.

In OCL, however, the undefined value is a first-class citizen. This is represented by the operation *oclIsUndefined()*, which is used to test whether a value is undefined. The interpretation $\mathcal{I}(t)(\sigma)(oclIsUndefined)$ is defined as follows:

$$\mathcal{I}(v)(\sigma)(oclIsUndefined) \stackrel{\text{def}}{=} \begin{cases} \text{true} & , \text{ if } v = \perp \\ \text{false} & , \text{ otherwise.} \end{cases}$$

A specification which evaluates to undefined, can be turned into a well-formed specification by applying the *oclIsUndefined* operation to it.

One central complication of the semantics of OCL is that the undefined value $\perp$ represents both the concept of “run-time error” and the concept of “divergence”. Both notions should *not* be concepts in the semantics of a specification language. These notions are important in the program to reason about. A specification is not meant to be executed, but to define desired properties of the program or model.

A “run-time error” means, that an OCL expression contains a function application where the argument is outside the domain of the function. Such constraints should be regarded as nonsensical, and the constraint is then considered to be false.

“Divergence” means, that a recursive OCL expression has no fixed-point or that this fixed-point is not computable using applicative order evaluation, that is, when all ar-

2.3 Object Constraint Language

guments to a function are evaluated first before the function is applied. Especially, the second case should be of no concern for the semantics of a specification language, because it is then undefined because of a particular implementation of an interpreter of the language. The meaning of quantification over infinite domains cannot be computed (or tested). However, many OCL expressions have a well-defined meaning in a declarative semantics. For Example, $T.allInstances() \rightarrow forAll(true)$ should mean the same as $\forall(x : T) : true$, which is, of course, a tautology. In OCL, the modified constraint $T.allInstances() \rightarrow forAll(true).oclIsUndefined()$ tests whether the domain of the type T is finite.

Decidability The unpleasant side-effect of *oclIsUndefined* is, that it is impossible to implement a correct, that is, standard-conforming, interpreter for OCL. The proof for this statement is defined as follows:

Theorem 2.21. *There exists no computable function f such that, for all constraints t , $f(t) = true$ if and only if $eval(D, \overleftarrow{\sigma}, \sigma, \eta)(t) = true$.*

Proof. Let t' be an OCL constraint such that $eval(D, \overleftarrow{\sigma}, \sigma, \eta)(t')$ diverges for some D , $\overleftarrow{\sigma}$, σ , and η . Choose $t \stackrel{\text{def}}{=} t'.oclIsUndefined()$. Clearly:

$$eval(D, \overleftarrow{\sigma}, \sigma, \eta)(t) = true \quad .$$

Suppose f is a computable function with $f(t'.oclIsUndefined()) = true$. Because this function is computable, it has to terminate on t' , therefore solving the Halting problem. Consequently, no such computable function exists. $\square$

A corollary of this theorem is that *eval* is not a standard-conforming implementation of an OCL interpreter. Indeed, if $eval(D, \overleftarrow{\sigma}, \sigma, \eta)(t) = \perp$ for any t where *eval* is diverging, then $eval(D, \overleftarrow{\sigma}, \sigma, \eta)(t.oclIsUndefined()) = \perp$.

Equality For three-valued logics introducing an undefined value, we have to distinguish two kinds of equality. The first one is strict in its arguments, that is, if one of the arguments is undefined, so is the equality expression. This equality is called *weak equality* and is written $\doteq$. The operator *iff* is indeed weak equality on *Boolean*.

The second one is called *strong equality*, written as $=$. Its semantics is not strict, but the undefined value is treated as a unique value by $=$. Consequently, the term $a = b$ is either true or false.

OCL defines equality $=$ as a property on the type *OclAny*, which can be overridden, that is, redefined, in subclasses of *OclAny*. This gives the flexibility to define equality by structure or equality by reference (or, God forbid, define $=$ to mean something completely different). The evaluation rules for property-call expressions, which $=$ is, have to be applied. This implies that equality is *weak*. For example, the constraint $1/x = 1/x$ is *not* necessarily true. If $x = 0$, then this constraint is undefined.

This violates the axiom of self-identity ($\vdash x = x$) used in virtually all deductive systems, which also represents one of the major surprises to users of OCL. The correct formulation of $1/x = 1/x$ in OCL is to write $x \neq 0 \text{ implies } 1/x = 1/x$, which is true for all x (see Table 2.2).

For reasoning it is preferable to use strong equality instead of weak equality, as defined in OCL. The reason for this is, that we cannot replace equals for equals anymore, which complicates reasoning a lot.

Meaning of @pre The meaning of the **@pre** modifier is confusing to users, because it does not apply to an expression as a whole, but only to the part it has been applied to. This operator allows to mix values from the pre- and the post-state of an operation call freely. This flexibility is needed, but for inexperienced users, a notation which allows to apply **@pre** to an expression as a whole is desirable. But very often, expressions are overloaded with these annotations, for example:

```
context c :: m() : Integer
post : result = (self.a@pre * self.a@pre+
                self.b@pre * self.b@pre).sqrt()
```

Characterisation OCL defines a scientifically but complex theory, which can be characterised as a first-order relational theory without equality but with late binding and fix-points. Especially the presence of late binding in a logic leads to a theory that is not yet well understood and too complex for the original purpose it was designed for. The lack of equality is surprising for many users.

Declarative

Requirement 3 states, that OCL must be a *declarative language*, which has been characterised by the absence of side-effects. OCL satisfies this requirement.

Typed Language

Whether a specification language (or constraint language) has to be typed is a matter of preference. Virtually all specification languages are typed. All of them have *manifest types*, that is, the type of the object denoted by a variable is used by the reasoning. The expression $a + b$ only has a meaning if a and b denote numerical quantities, for example, their manifest type is *Real*. Adding a set to a number is, on the other hand, considered to be nonsensical.

The more important and interesting question is, whether a specification language should be *statically typed*. This implies that all entities in a specification have a declared type and a *type checker* proves that all formulae are well-typed with respect to a type system.

OCL is a statically typed language. However, the type system used for OCL is weak, in the sense that the most precise type of each entity cannot be determined. Even the extended type system described in Chapter 3 is weak. This is a consequence of requiring a decidable type system for a statically typed language. In the case of OCL, this implies that the type system has to lose information, that is, the types computed for expressions are actually super-types of the manifest type of the expression.

OCL solves this problem by introducing re-typing operations (or casts). The use of these additional operations force the type checking procedure to assume that the manifest type of an expression is a sub-type of the type the expression is re-typed to. However, the type checker cannot prove whether the cast is justified (otherwise it would not have been necessary in the first place), so it is possible to cast an expression into a type it is not a value of. This results in an undefined value, a situation static type-checking is supposed to avoid.

On the other hand, a sufficiently strong type system, like the one used for the PVS specification language of Higher Order Logic [121, 93], is much more expressive, avoids all casts, but is not decidable. If these systems cannot determine whether a specification is well-typed, they generate proof obligations which have to be proved interactively.

2.4 State Machines

UML offers many methods for describing the behaviour of objects. One method used for specifying the behaviour of *active* objects is using UML state machines. UML state machines have evolved from Harel's state charts [60] via the object-oriented successors of state charts [61].

In this section we summarise the notation of state machines and briefly review its semantics. In this thesis we only use *flat state machines*, that is, state machines in which no state is contained in another state. Such states are also called simple.

The basic concepts of UML 2.0 state machines are states and transitions between them. Transitions are labelled with a *trigger*, which specifies a signal, which may cause the transition to be taken, followed by a *guard*, which is a boolean condition on the state of the system as defined by an object diagram. The final component of a transition label is the *action*, which will be performed if the transition *fires*, that is, is taken to obtain a new state. This label is written $t[g]/a$, where t represents the trigger, g represents the guard, and a represents the action. Components may be omitted from the label, which also leads to the omission of the decoration, for example, if the guard is omitted, we simply write t/a , if the trigger and the guard is omitted, we write $/a$, and if the transition is labelled with only the trigger we write t as a label.

The environment may send events to the state machine. These events are collected in the event pool of the state machine. A state machine may dispatch a single event from its event pool to trigger transitions.

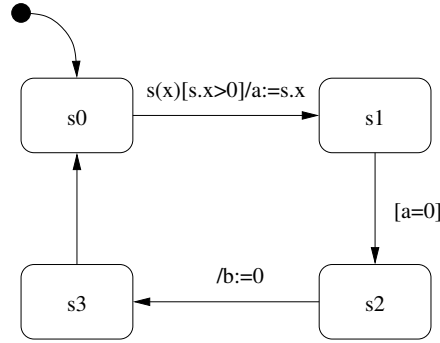


Figure 2.4: State machine

A transition is enabled if its source state is currently active, its event is dispatched from the event pool, and its guard evaluates to true. The firing of transition t leads, in this order, to:

1. the deactivation of the state that will be left by firing t ,
2. the execution of the actions of t (which we simply call the execution of t)
3. the activation of its target state.

UML state machines use a *run-to-completion* semantics, that is, “an event occurrence can only be taken from the pool and dispatched if the processing of the previous current occurrence is fully completed”

We give an intuitive explanation of the syntax and semantics using the state machine displayed in Figure 2.4. The state $s0$ is the initial state of the state machine, as indicated by the transition leading to $s0$ from the black dot. While the state $s0$ is active, the next transition can only be activated by a signal of type s , which defines at least the attribute x , indicated by the trigger $s(x)$, such that the value of $s.x$ is larger by one, which is indicated by the guard $s.x > 0$. While in state $s1$, the state machine awaits that the value of its attribute a is changed to 0. In this case, the state of the state machine is changed to $s2$, and no action is performed. Such transitions are triggered by so-called *change events* which are generated whenever a guard becomes true. Note that the change event will *not* be removed from the event pool, if the value of the condition is changed to false after the event has been generated. There is only one outgoing transition leaving $s2$, which is neither guarded by a trigger nor a guard. Therefore, it is always enabled. If neither a trigger nor a guard is defined for a transition, then it is taken, whenever a so-called completion event is generated. In our case, such events are always generated once a transition has been taken. When the transition is taken, then the attribute b will be set to 0. Finally, we have an unlabelled transition from $s3$ to $s0$, which will be taken with a completion event, and where no action will be performed. Whenever an event is chosen from the event pool, which does not trigger a transition,

it is discarded, unless it is declared *deferrable*. Deferrable events are placed back into the event pool and a new event will be chosen from the event pool.

The actions defined in this thesis are:

- Assignment to attributes, written $a := e$, where a is an attribute name of the object, whose behaviour is to be described, and e is an OCL expression.
- Object creation, written $a := \text{new } c$, where a is an association end and c is the name of a class. If the multiplicity of the association end is 0..1 or 1, the semantics is that the new object will *replace* the object associated to the creating object. Otherwise the end refers to a collection of objects and the newly created object is *added* to the collection. If adding the newly created object violates a multiplicity constraint, the system may raise a run-time error once the system becomes stable.¹¹
- Signal emission, written $a!s(\vec{e})$, where a is an association end, s is a signal name, and $\vec{e}$ is a list of expressions used to initialise the attributes of s .
- Operation calls, written $a.m(\vec{e})$, where a is an association end, m is a signal name, and $\vec{e}$ is a list of expressions providing the actual arguments to m .

The semantics of state machines used in this thesis has been provided by Hooman and van der Zwaag [149]. In their paper a formalisation of the semantics in PVS is defined, which we use in Chapter 4. Their semantics is based on the semantics described by Damm et al. in [39]. Fecher, Kvas, de Roever, and de Boer describe a compositional semantics of state machines based on the before-mentioned semantics in [52]. Because the semantics described in these papers differ considerably from the semantics described in UML 2.0 [111] state machines, we refer the reader to Schönborn [140] for a description of the semantics which is closer to the standard.

2.5 Summary

In this chapter we have introduced the essential notations of the UML 2.0 standard used in this thesis. We have defined a formal semantics in terms of an interpreter for these core concepts. For other operations of OCL the semantics is defined in terms of other OCL expressions.

The formal definition of the semantics closely follows the standards [111, 113], where we deviate from the standard if inconsistencies arise. In some cases, we simply omitted constructs which cause problems, for example, if we were not able to determine the intention or the meaning of the construct in the specification.

¹¹This semantics allows, for example, initialising a collection with multiplicity $x..*$, where $x > 1$, by successively adding objects to the collection.

The approach for interpreting UML class diagrams as algebraic signatures has been described Börger et al. [14], Reggio et al. [132], Grosse-Rhode [58], and others. We follow a very similar approach.

For example, Börger et al. [14] define the dynamic behaviour of UML state machines using abstract state machines [59].

Grosse-Rhode uses many-sorted algebras to interpret class diagrams [58]. The dynamic behaviour of such a system is modelled by *transformation systems*, which are very similar to Gurevich’s abstract state-machines [59]. Inheritance is modelled by signature-morphisms.

Formal semantics for OCL have been proposed by Richters [133] and by Baar [7]. Our definition of OCL 2.0 follows their definitions in many respects, but we also take the changes and new constructs of OCL 2.0 [113], for example, the null literal, into account. A second difference between our formalisation and their formalisation is, that we use higher-order logic as formal meta-language instead of a first-order predicate calculus and Zermelo-Fraenkel set-theory.

Brucker presents a deep embedding of OCL into Isabelle/HOL [101] in [19]. He proposes many improvements to the then current semantics. But he does not take the clarifications described in the “Amsterdam Manifesto” [34] into account, which includes many clarifications that were worked into the later versions of the OCL standard.

Finally, the interpretation of invariants and pre-/postconditions is very similar to the interpretation used by Richters and Gogolla [134] and Hennicker, Hussmann, and Bidoit [62]. However, in both references it has been left unspecified how the actual arguments and the result is represented in states.

Indeed, the semantics of OCL displays problems similar to the ones observed by MacQueen [95]: If a declarative language, in MacQueen’s paper a functional language like ML, has to be extended with a full-fledged class-based object system, then this “leads to an excessively complex type system and relatively little expressive gain, *especially if we aim to preserve that mostly functional style of programming*” (Emphasis added by us).

The semantics of navigation expressions in OCL and UML are related to relational algebras and are inspired by relational database systems, which makes it appealing to use Jackson’s Alloy Language for formalising OCL and UML [69].

A formal semantics for UML and OCL enables tool vendors to build tools which allow the exchange of models without unexpected reinterpretations by the tool, which may in practise defeat the purpose of communicating models and ideas, as well as the interchange format XMI [108]. A formal semantics also enables developers to “calculate” implementations from specifications, for example by stepwise refinement. An executable subset of UML can then be used for rapid prototyping and simulation, which helps in detecting design errors early.