5 Formal Specification

by

Andreas Roth
Peter H. Schmitt

This chapter serves as an introduction to formal specifications. In Sect. 5.1 we reconsider in greater detail, but still on a fairly general level, the basic building blocks of formal specification—pre- and postconditions, invariants, and modifies clauses—that have already been informally introduced in Sect. 1.3. The next two sections then show how these notions can be formulated in two popular specification languages, OCL and JML. A short comparison between the two languages in Section 5.4 concludes this chapter.

Methodological questions like: How should operation contracts be inherited by subclasses? At which system states are invariants required to hold? How can modular specification and verification be effected? will be postponed till Chapter 8.

5.1 General Concepts

Specifications may be used at different stages in the software development process. They may be attached to a coarse design model or to runnable code or at any stage in between. What is essentially needed for the kind of specifications we treat in this book are

1. a notion of a state, e.g., the state or snapshot of a system model or the state of computation of a JAVA program;
2. a notion of a transition from pre-state to post-state effected by an operation;
3. a language to formulate specifications. It is understood that we should be able to determine whether a statement in this language is true or false in any given state.

In the example in Sect. 1.3 user authentication was not considered. Let us address this task now. We think of a user having inserted her bankcard into an automatic teller machine (ATM). After some basic initalisation the machine performs an operation we choose to call `enterPIN`. The user is prompted to enter her pin.

Let us start thinking about what specifications we may want for `enterPIN`.

5.1.1 Operation Contracts

We decide to allow three attempts to enter the correct pin. So, it is natural to introduce an attribute `wrongPINCounter` that counts the number of unsuccessful attempts. An operation contract for `enterPIN` may then look like this:

```
precondition    card is inserted, user not yet authenticated
postcondition   if entered pin is correct
                  then the user is authenticated,
                if entered pin is incorrect
                    and wrongPINCounter < 2
                  then wrongPINCounter is increased by 1
                    and user is not authenticated,
                if entered pin is incorrect
                    and wrongPINCounter >= 2
                  then the card is confiscated
                    and user is not authenticated.
```

With this concrete example in mind we are ready for the general definition. Here and in the following we will use the word *operation* in a general sense and refer to implementations of operations as *methods.*

Definition 5.1. *A contract with precondition and postcondition for an operation op is satisfied if:*

> *When op is called in any state that satisfies the precondition then op terminates and in any terminating state of op the postcondition is true.*

This definition looks innocuous enough, but it is worth stressing the following facts.

1. No guarantees are given when the precondition is not satisfied. If for some reason the `enterPIN` method is called when no card is inserted, there is no telling what happens.
2. By default, termination is part of the contract. There are other options though and we will come back to this a bit later in this subsection.
3. The terminating state may be reached by normal or by abrupt termination, i.e., termination by *op* throwing an exception.

It is usual to allow more than one pre-/postcondition pair in a contract. The above example could be rephrased as:

precondition	card is inserted, user not yet authenticated, pin is correct
postcondition	user is authenticated,
precondition	card is inserted, user not yet authenticated, pin is incorrect and `wrongPINCounter < 2`
postcondition	`wrongPINCounter` is increased by 1, and user is not authenticated,
precondition	card is inserted, user not yet authenticated, pin is incorrect and `wrongPINCounter >= 2`
postcondition	card is confiscated and user is not authenticated.

In this example, the preconditions are mutually exclusive. This is not required in general.

In non-trivial contexts it is not easy to come up with a postcondition that precisely defines an operation. One particular difficulty is to state what items do not change. This is a notorious problem in many areas of computer science that deal with some notion of *action* and has been given a special name: the *frame problem.* The first observation towards a solution of the frame problem, at least in our context, is that we should make no attempt to enumerate the items that do not change. There are hopelessly many. Rather we should try to determine those items that at most may get changed. We thus add an explicit list of items that may be modified by a method to its specification. This is not an uncommon approach, and has consequently been pursued in the book [Morgan, 1990].

In our `enterPIN` example the modification list might look like this:

modifies	`wrongPINCounter`
	all attributes needed for user authentication
	all attributes needed for confiscating the card

Since we have at this level of the design not fixed all attributes we have to be a bit vague about the modification required for user authentication. It is easy to imagine how a card gets confiscated in the real world; it ends up in a special box of the machine to be picked up by a clerk. We will see below how this can be modelled in a specification.

The requirements given in Definition 5.1 go in program verification theory by the name of *total correctness.* When termination is not required we speak of *partial correctness.* The choice between these alternatives is realised by adding another clause `termination` to the contract. Possible values for this clause could be `required`, `not required` or `normal termination`. In the last case we do not want termination to be brought about by an exception.

For a uniform treatment we stipulate that all operations have a contract. If contract parts are not explicitly given, we assume the defaults in Table 5.1.

Table 5.1. Default contracts

Default Contracts	
precondition	*true*
postcondition	*true*
modifies	*everything*
termination	*required*

5.1.2 Invariants

Another frequently used specification method is that of an invariant, i.e., a statement that is required to be true in all system states. A possible invariant in the `enterPIN` scenario is:

invariant `wrongPINCounter` is always ≥ 0 and ≤ 2

Since in our model of the banking world the `wrongPINCounter` attribute is attached to the class `ATM`, this invariant says that in all system states, for all ATM-machines `m` that exist in this state, $0 \leq$ `m.wrongPINCounter` ≤ 2 is satisfied.

In object-oriented programming and design it is has become customary to attach invariants to classes or interfaces. We will follow this practice. Frequently invariants address only one instance of a class. This has lead to the figure of speech of an object satisfying an invariant throughout its lifetime. Though this view is helpful in many cases, it fails to cover invariants that address only static fields. Also, invariants addressing two instances of a class are not covered naturally since one of them has to be chosen as the main actor, arbitrarily. An example of the latter type of invariants is the following, attached to the class `BankCard` (see Figure 5.1 below):

invariant no two cards have the same `cardNumber`

Invariants may even address fields and objects from different classes. To give an example we expand our scenario by considering a central host to which the ATM is connected. In particular we imagine that this central host provides an attribute `validCardsCount` satisfying:

invariant `validCardsCount` equals the number of issued
bank cards that are still valid

The next step towards a thorough understanding of the concept of an invariant is the answer to the question: When should an invariant hold? The first and quick description we used at the beginning of this subsection, that an invariant should hold in all system states, is in most cases far too strong. This leads us to consider two notions of invariants:

- *strong invariants* that really hold in all system states
(these will be treated in Sections 9.2 and 9.4),

- *invariants* without further qualification
  (we will continue to consider these here).

As a first approximation we may require an invariant to be true as long as no operation is executing. The method `ATM::confiscateCard()` will probably first set `insertedCard = null` which will destroy the above invariant on `validCardsCount`. Only after `ATM::confiscateCard()` updates the field `validCardsCount` will it be true again. We record the present state of our discussion in the following definition:

Definition 5.2. *A class C satisfies an invariant Inv if,*

1. *for any operation op and any state s satisfying at least one precondition of op and Inv, the invariant Inv is also true in any terminating state.*
2. *Inv is true in the state reached after execution of any constructor.*

Notice, that nothing is said here about termination or truth of the postcondition. These issues are settled in the operations contract for *op*.

This is a first working definition that will be sufficient for the purposes of the present chapter. It does not address the special case of invariants involving only static fields and leaves open which operations should be considered. Intuitively all exported operations of the class to which the invariant is attached should suffice. This is not true in all cases (⇒ Chap. 8).

As can be seen from Definition 5.2, invariants are in principle superfluous, one could add them to pre- and postconditions of every operation contract. Obviously, this is not a very practical alternative in particular in a context where new subclasses are added to an existing program.

The reader might wonder at this point how invariants and operation contracts behave with respect to the class hierarchy. Our position on this issue is:

- an invariant of a class is inherited by all its subclasses,
- on the other hand an operation redefined in a subclass does not inherit the operation contract from the superclass.

Given the number of subtyping disciplines that have been proposed in the literature this non-committal approach for operation contracts seems to be best suited.

An example of another kind of invariant that is useful at a later stage in the software development when code is already present are loop invariants:

```
/* loop invariant Inv */
while ( guard ) {
        body
       }
```

The intention is that `Inv` is true in the state before the while loop is entered and again in the states after each execution of its body. No commitment is made on the termination of the loop.

5.2 Object Constraint Language

The Object Constraint Language (OCL) is part of the OMG standard Unified Modeling Language, UML[1]. An easy introduction is available through the book [Warmer and Kleppe, 2003]. Material on a precise semantics of OCL is contained in the volume [Clark and Warmer, 2002], in particular [Gogolla and Richters, 2002]. Another source for a formal semantics is [Brucker and Wolff, 2002]. There is also Chapter 10 of the standard describing the semantics in terms of UML plus OCL. But, not many people found this account accessible. Our text follows the draft of the OCL Version 2.0 standard as of June 6, 2005, [OCL 2.0].

OCL was introduced to express those parts of the meaning that diagrams cannot convey by themselves. It was first developed in 1995 by Jos Warmer and Steve Cook. The most extensive use of OCL so far is within the UML standard itself, where it is used in the semantics description of the UML metamodel. For an example of the use of OCL in API specification see [Larsson and Mostowski, 2004].

OCL is perceived by its creators as a *formal* language. On the other hand they put emphasis on the fact that OCL is not designed for people who have a strong mathematical background. We quote from [Warmer and Kleppe, 1999a, Preface]:

> The users of OCL are the same people as the users of UML: software developers with an interest in object technology. OCL is designed for usability, although it is underpinned by mathematical set theory and logic.

5.2.1 OCL by Example

Before we enter into a systematic treatment of OCL we start with some instructive examples. The UML standard allows one to add constraints to almost every diagram type. In this chapter we exclusively consider constraints in UML class diagrams and use the diagram in Figure 5.1 as our running example. It is in fact the UML class model for the scenario previously sketched in Sect. 5.1. It contains the three classes `ATM`, `BankCard` and `CentralHost`. Of the attributes for the `ATM` class we have already encountered `wrongPINCounter`. The attribute `insertedCard` is either `null` or points to the instance of class `BankCard` that is currently inserted in the ATM. We use the Boolean field `customerAuthenticated` to model whether the inserted card is authenticated or not. The last attribute `centralHost` points to the central host the ATM in question is attached to.

Figure 5.2 shows how the informal contract for the `enterPIN` operation given above translates into OCL.

[1] See `http://www.uml.org`

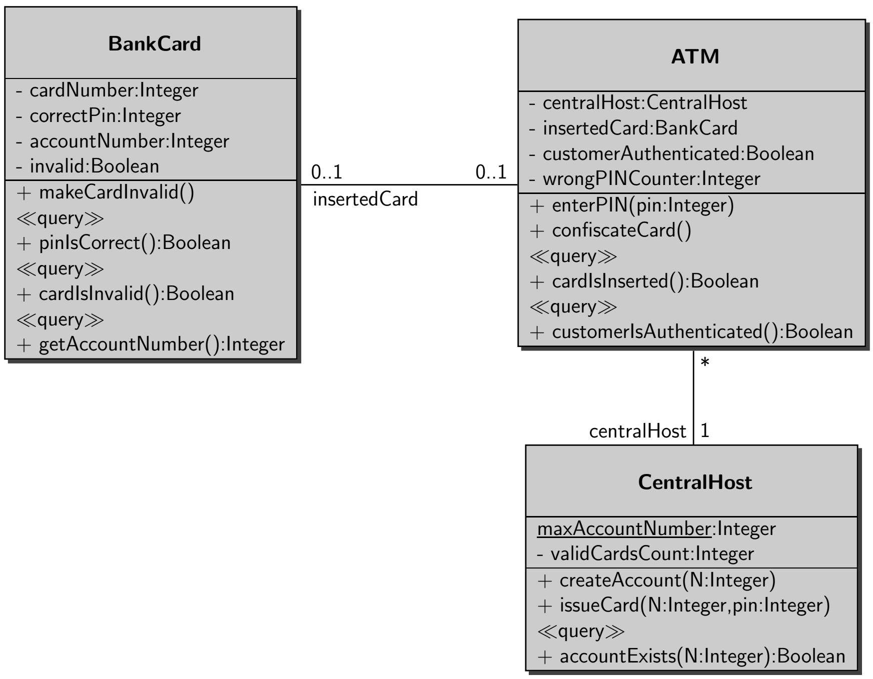

BankCard
- cardNumber:Integer
- correctPin:Integer
- accountNumber:Integer
- invalid:Boolean
+ makeCardInvalid()
≪query≫
+ pinIsCorrect():Boolean
≪query≫
+ cardIsInvalid():Boolean
≪query≫
+ getAccountNumber():Integer
0..1
insertedCard
0..1
ATM
- centralHost:CentralHost
- insertedCard:BankCard
- customerAuthenticated:Boolean
- wrongPINCounter:Integer
+ enterPIN(pin:Integer)
+ confiscateCard()
≪query≫
+ cardIsInserted():Boolean
≪query≫
+ customerIsAuthenticated():Boolean
*
centralHost
1
CentralHost
maxAccountNumber:Integer
- validCardsCount:Integer
+ createAccount(N:Integer)
+ issueCard(N:Integer,pin:Integer)
≪query≫
+ accountExists(N:Integer):Boolean

Fig. 5.1. Class diagram for the ATM scenario

This specification uses two features that are not present in the OCL standard but are implemented in the KeY system. The modifies clause we have simply added since it proved indispensable for our purposes and also since it increases compatibility with JML (⇒ Sect. 5.3). The literal `null` is the only element in the OCL type `VoidType` and serves us to denote Java's null object. We will comment on this later, Section 5.2.4. This being said let us have a closer look at this OCL operation contract.

First we observe that the syntactical entities used either come from the class diagram or are OCL built-ins. From the class diagram we are allowed to use in OCL: names of classes (does not occur in this example, but will appear shortly), attributes, association ends (in the present example this does not show since, incidentally, any association end also occurs as an attribute), and queries. Arbitrary operation names may not occur outside the context declaration since OCL is intended to be side-effect free.

Next you would expect that in an object-oriented setting an attribute like `insertedCard` in the above precondition is applied to a particular object. In fact it is, you just do not see it. By a frequently used shorthand, object references may be omitted and will be replaced by default with the reserved variable `self` which plays in OCL the same role that `this` plays in Java. Thus the precondition from Fig.5.2 reads in full:

── OCL ──
```
context  ATM::enterPIN(pin: Integer)
modifies: customerAuthenticated, wrongPINCounter,
          insertedCard, insertedCard.invalid
pre:      insertedCard <> null and not customerAuthenticated
post:     if pin = insertedCard@pre.correctPIN
             then customerAuthenticated
             else
               if wrongPINCounter@pre < 2
                  then wrongPINCounter = wrongPINCounter@pre + 1
                       and not customerAuthenticated
                  else
                     insertedCard = null
                     and insertedCard@pre.invalid
                     and not customerAuthenticated
                endif
           endif
```
── OCL ──

Fig. 5.2. An OCL contract for the `enterPIN` operation

── OCL ──
```
context ATM::enterPIN(pin: Integer)
pre:     self.insertedCard <> null and
         not self.customerAuthenticated
```
── OCL ──

OCL offers as a third possibility that you declare your own local reference, e.g., `atm1`:

── OCL ──
```
context atm1:ATM::enterPIN(pin: Integer)
pre:     atm1.insertedCard <> null and
         not atm1.customerAuthenticated
```
── OCL ──

These references `self` or `atm1` are implicitly universally quantified, i.e., the intended meaning of an operation contract is: in all system states and for all instances `atm1` of class `ATM`, if the precondition for `atm1` is satisfied, then the postcondition is satisfied for `atm1`. Implicit universal quantification also applies to any other reference occurring in the contract, like the argument `pin` in our example.

Still looking at Figure 5.2, we notice the peculiar `@pre` symbol. Only in postconditions it may be attached to attributes, associations or queries and refers to the value of the corresponding model element before execution of the operation.

Let us look at some more examples of OCL constraints. The following contract explains how we model confiscation of cards.

—— OCL ——
```
context ATM::confiscateCard()
pre:    insertedCard <> null
post:   insertedCard = null and insertedCard@pre.invalid
```
—— OCL ——

The attribute `insertedCard` is reset to `null`. This is obvious, since after confiscation there is no card inserted. In our model, however, the card in question is still an instance of class `BankCard` undistinguished from all other instances. To avoid this we introduced the attribute `invalid` of `BankCard` which is set to `true` when the card gets confiscated. Notice, we have to use `insertedCard@pre` since in the post state `insertedCard` is `null`.

So much for operation contracts. Let us now present examples of OCL invariants.The simplest invariant from Section 5.1.2 is formalised in OCL as:

—— OCL (5.1) ——
```
context ATM
inv:     0 <= wrongPINCounter && wrongPINCounter <= 2
```
—— OCL ——

In greater detail we would add explicitly the variable `self` which is thought of as universally quantified:

—— OCL ——
```
context ATM
inv:     0 <= self.wrongPINCounter &&
         self.wrongPINCounter <= 2
```
—— OCL ——

The next invariant gives us the occasion to use some of the more advanced built-in concepts.

—— OCL (5.2) ——
```
context BankCard
inv:    BankCard::allInstances() ->  forall(p1,p2|
        p1<>p2 implies p1.cardNumber<>p2.cardNumber)
```
—— OCL ——

Note, that now the context is the class `BankCard`. The intended meaning of this invariant is evident. Here `allInstances()` is a query that is available for most classes. (More precisely it is inherited from the class `OclAny` for all subclasses of `OclAny`.) In any snapshot, `A::allInstances()` evaluates to the set of all existing elements of class `A`. In the OCL standard the use of `allInstances()` is restricted to classes with finitely many elements and required to yield an undefined result when applied to a class with infinitely many elements like `String` or `Integer`. This is a viable position when you use OCL in simulation tools or for runtime checking. It is too restrictive for formal verification in general.

`BankCard::allInstances()` is our first example of an OCL expression that evaluates to a collection of objects rather than to a single object or a single value. This also accounts for the `->` symbol following `BankCard::allInstances()`. To provide for an easier reading, OCL uses `->` instead of a simple dot when applying an operation to a collection. In fact, if you change all `->` to dots in an OCL expression and then hand it to me I will be able to restore all arrows (except in one very special exceptional case). The operation that is applied to the collection `BankCard::allInstances()` in the present case is universal quantification. Unlike in other languages where you may always add quantifiers to a formula, e.g., $\forall p_1.\forall p_2.(p_1 \neq p_2 \rightarrow c(p_1) \neq c(p_2))$ in predicate logic, in OCL you first have to provide a collection that the quantifier, universal or existential, will range over. Notice also that OCL allows you to quantify two variables by one operator, just as some logic notations would allow you to write $\forall p_1, p_2.(p_1 \neq p_2 \rightarrow c(p_1) \neq c(p_2))$.

The next example introduces more operations on collections.

— OCL (5.3) —

```
context CentralHost
inv: validCardsCount =
     BankCard::allInstances() ->
               select(not invalid) -> size()
```

— OCL —

We again encounter a shorthand here. The full version could look as follows:

— OCL —

```
inv: validCardsCount =
     BankCard::allInstances() ->
               select(c | c.invalid) -> size()
```

— OCL —

Now, this might look familiar to readers with a background in basic set theory. Assume that `BankCard::allInstances()` evaluates to a set A then the whole expression evaluates to the set of those elements $c \in A$ that satisfy the condition after the | symbol.

You can think of the UML class diagram and its OCL constraints as a specification, as a blue print for a system to be built. At this level one can check consistency of the specification or try to derive other properties of the specification alone. In addition, if code has been written, you will want to prove that it satisfies the constraints. A possible implementation of the `enterPIN` method is shown in Figure 5.3. If you are interested to try out for yourself the verification of this operation contract with KeY you will find assistance in Section 10.3.

Java
```
public void enterPIN (int pin) {
  if ( !( cardIsInserted () &&
          !customerIsAuthenticated () ) ) {
     throw new RuntimeException ();
  }

  if ( insertedCard.pinIsCorrect ( pin ) ) {
    customerAuthenticated = true;
  } else {
    ++wrongPINCounter;
    if ( wrongPINCounter >= 3 )
    confiscateCard ();
  }
}
```
Java

Fig. 5.3. The `enterPIN` method

The examples we have seen so far were close to program code. The diagram in Figure 5.4, however, may occur very early in system modelling. It identifies the main classes in a role-based access scenario, `User`, `Role`, `Permission`. In addition there are associations connecting users and roles and also roles with permissions. No commitment is made at this point on how these relations will be realised in code. Notice the asterisks at each association end. They stand for multiplicities and signal that a user may have an unbounded number of roles, a role may be assigned to an unlimited number of users, a role may be granted an unlimited number of permissions and a permission may be part of an unbounded number of roles.

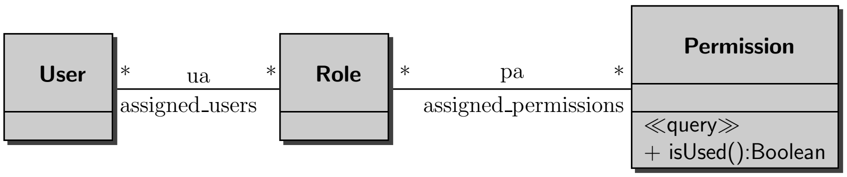


Fig. 5.4. UML class diagrams for role-based access scenario

We can think, even at this general level, of useful contracts, e.g., that every permission is used:

—— OCL ——
```
context Permission::isUsed():Boolean
post:   result = role.assigned_users -> notEmpty()
```
—— OCL ——

We already know that the postcondition is a shorthand of:

—— OCL ——
```
post:   self.result = self.role.assigned_users -> notEmpty()
```
—— OCL ——

Furthermore **result** is an OCL keyword that can only be used in postconditions of operations that return a result, exactly for specifying what this result should be. Notice, that this offers the possibility to not only give a condition that should be satisfied after execution of the operation, but to uniquely define its result.

Let us look at the expression at the right hand side of the = sign. It shows that we can string together several dot-operations. That is what in OCL jargon is called *navigation*, because it has the effect of navigating through the UML class diagram. The first leg of this navigation is towards an unnamed association end. In this case the default is to use the name of the class to which the association end is attached, spelled in lower case. This first leg yields the set $R = \{r_1, \ldots, r_k\}$ of roles that are attached to the permission represented by `self`. The whole expression `self.role.assigned_users` is a much used shorthand for:

—— OCL (5.4) ——
```
post:   self.role -> collect( r | r.assigned_users)
```
—— OCL ——

If for $r = r_i$ the OCL expression `r.assigned_users` evaluates to a set U_i of users then the whole expression evaluates to the union $U_1 \cup \ldots \cup U_k$. More precisely, this union is considered as a bag or multi-set with the consequence, that a user that is assigned to more then one role will occur more than once in the result.

5.2.2 OCL Syntax

In this subsection we try to convey a basic understanding of the syntax and semantics of OCL. The main reference for a full definition is the OMG standard draft [OCL 2.0]. This is still not the final document and does not settle all issues, but it will be more than sufficient for what we need here.

The Object Constraints Language, OCL, is a typed language; every expression has a unique type. Evaluating an expression e in a snapshot yields a value of the type of e. Figure 5.5 presents a survey of the types available in OCL. Abstract classes, i.e., classes whose instances are all instances of one of its subclasses, are written in italics. It deviates from [OCL 2.0, Chapter 8, Figure 5] in that

- it does not show `ElementType`, which is mainly used to connect to state machine diagrams, which we do not consider here,
- it does not show `InvalidType`, since we do not use it, see Section 5.2.4.
- it does not show `TypeType` since the meaning of this remains unclear,
- we chose to omit `OrderedSetType` for brevity,
- it shows `AnyType` as a subclass of `PrimitiveType` rather than as direct subclass of `Classifier`. The standard's position on this is still inconsistent and the difference does not have an impact on what we have to say here.

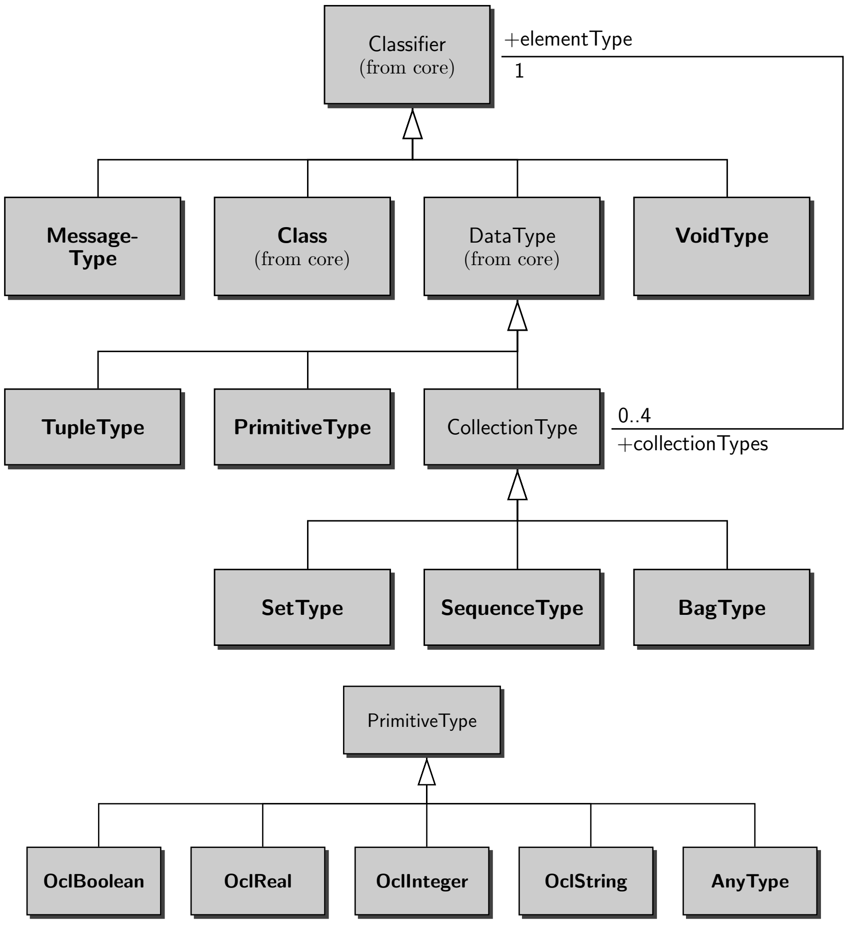


Fig. 5.5. The hierarchy of OCL types

The classes shown in Figure 5.5 are metaclasses. To illustrate what is meant by this look at `OclInteger`. This class has exactly one instance, which is named `Integer`. The instances of `Integer` in turn are the well known numbers $\ldots -1, 0, 1, \ldots$.

OCL expressions are always placed into the context of an UML class model. The classes, say $C_1, \ldots, C_k$, appearing there will be the instances of the metaclass `Class` from Figure 5.5. Evaluation of an OCL expression is always done with respect to a fixed snapshot s of the modelled system. To evaluate expressions we need to know how to evaluate types in s. This is easy for types that derive from the class diagram: the type C_i evaluates to the set of all instances of class C_i that exist in s.

`CollectionType` is an abstract class, that is to say, that any instance of it has to be an instance of one of its subclasses. For any instance C in the metaclass `Class` we have the instances `Bag(C)`, `Set(C)`, `Sequence(C)` of `BagType`, `SetType`, and `SequenceType`, respectively. The evaluation of these are, naturally, the bags, sets, sequences of elements of class `C` existing in a given snapshot s. Also `Set(Integer)`, `Set(String)` etc. occur in `SetType` and even `Set(Set(C))` and `Set(Set(Integer))` are legitimate types with the usual intended meaning.

We skip commenting on tuple types, since they are what you expect. Thus `VoidType` and `AnyType` remain. The metaclass `VoidType` has exactly one instance `OclVoid`. The only instance of `OclVoid` is the element `null`. The only instance of `AnyType` is the OCL type `OclAny`. This type is meant to be the big grab bag of almost everything. At any snapshot every instance of a model class `C`, every instance of a primitive type is also an instance of `OclAny`. The OCL standard decided that this should not apply for instances of collection or tuple types. The main reason is to steer clear of semantical problems. OCL not only uses types, but also declares a subtype relation among them, much in the same way as in the first-order logic introduced in Chapter 2. This relation is called *conformance* and is defined as follows:

Definition 5.3. *The* conforms_to *relation is the least reflexive and transitive relation on the set of all OCL types satisfying the following conditions*

1. `Integer` *conforms_to* `Real`,
2. C_1 *conforms_to* C_2 *for instances* C_i *of* `Class` *iff* C_1 *is a subtype of* C_2 *in the UML class diagram,*
3. $S(T_1)$ *conforms_to* $S(T_2)$ *for* S *one of* `Collection`, `Set`, `Bag` *or* `Sequence` *iff* T_1 *conforms_to* T_2,
4. T *conforms_to* `OclAny` *for any type* T *that is not a collection or a tuple type,*
5. `OclVoid` *conforms_to every other type,*
6. *for tuple types we have:* $Tuple(name_1{:}T_1, \ldots, name_k{:}T_k)$ *conforms_to* $Tuple(name'_1{:}S_1, \ldots, name'_k{:}S_k)$ *iff* $\{name_1, \ldots, name_k\} = \{name'_1, \ldots, name'_k\}$ *and for* $name_i = name'_j$ *we have* T_i *conforms_to* S_j.

According to item 6 above, the OCL expression

$$Tuple(first{:}Integer, second{:}Integer, node{:}C)$$

conforms to

$$Tuple(second{:}Integer, node{:}C, first{:}Real)$$

if C conforms to D.

Having completed our survey of OCL types we are now ready to explain what OCL expressions are. It has become popular to present formal languages by a metamodel. Figure 5.6 shows the top level of the metamodel for the abstract syntax of OCL expressions. We explain bit by bit how to read this model. Classes with the label "`(from core)`" are classes from the metamodel of UML. They serve to connect OCL expressions to the class diagram to which they belong. The top-level class for OCL is the abstract class *OCLExpression*. If you are more comfortable with grammar rules you could express this information equivalently by the production rule

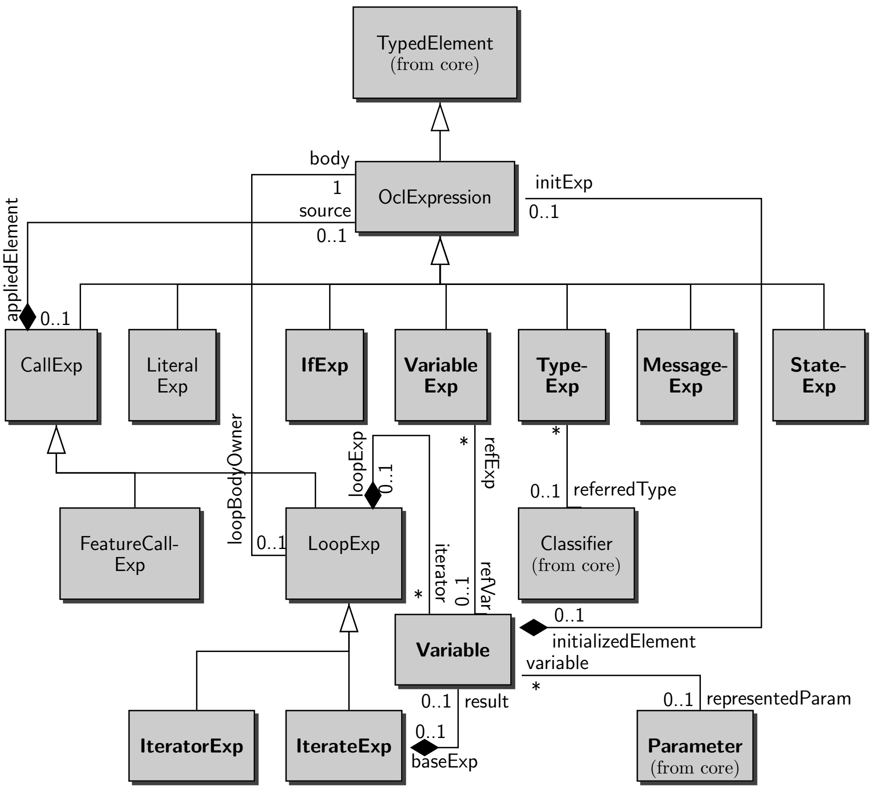


Fig. 5.6. Toplevel metaclass diagram for OCL expressions

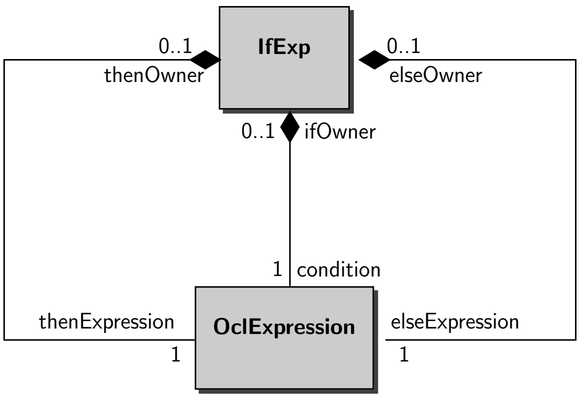


Fig. 5.7. Metamodel for conditional expressions

```
OclExpressionCS ::= CallExpCS    | VariableExpCS |
                    LiteralExpCS | LetExpCS   |
                    MessageExpCS | IfExpCS
```

We use the postfix `CS` to distinguish between the names of metaclasses and non-terminal symbols in the concrete syntax grammar. Notice that there are no non-terminals for the metaclasses `TypeExp` and `StateExp`. These will occur as parts of OCL expressions, but cannot stand alone as an OCL expression. The non-terminal `LetExpCS` has no corresponding class in Figure 5.6, because we left it out to not clutter the diagram even further.

Before we look closer into the top-level diagram, we describe the general workings of the abstract syntax model by looking at the simple metamodel for conditional expressions in Figure 5.7. The diagram shows that a conditional expression consists of OCL expressions, referred to by the association ends `condition`, `thenExpression`, and `elseExpression`. These three expressions are considered as parts of the conditional expression as signalled by the composition icons. The multiplicities at the corresponding ends, that is 1 in all cases, show that none of these may be missing. This is an elegant way to describe conditional expressions abstractly without imposing a concrete syntax.

Certainly, not every OCL expression can serve as a value for the `condition`. This restriction cannot be expressed in the metaclass diagram. Instead OCL constraints are added. For conditional expressions these are the invariants:

—— OCL ——
```
context IfExp
inv:  self.condition.type.name = 'Boolean'
inv:  self.condition.type.oclIsKindOf(PrimitiveType)
inv:  self.type = thenExp.type.commonSuperType(elseExp.type)
```
—— OCL ——

The attribute `type` is inherited from the UML metaclass `TypedElement`. The first invariant says that the type of the condition expression has to be named Boolean. Since somebody might draw a class diagram with a class named Boolean the second invariant requires that the type of the condition expression be primitive. The operation `oclIsKindOf(T)` may be found in the OCL standard library as a Boolean operation on the class `OclAny` with the explanation that `s.oclIsKindOf(T)` is true if `s` is a (not necessarily immediate) subtype of `T`. The third invariant determines the type of the *if* expression. `s.commonSuperType(t)` is a defined OCL expression that returns the least common supertype of `s` and `t` if it exists and undefined otherwise. A complete OCL definition of the `commonSyperType` operation will be given at the end of Section 5.2.2. The type of

```
if c:Boolean then t:Integer else s:Real endif
```

thus is `Real`. Finally, here is the grammar rule for the concrete syntax.

```
IfExpCS ::= 'if' OclExpCS 'then' OclExpCS
            'else' OclExpCS  'endif'
```

Let us now turn back and look at Figure 5.6 again. For abstract classes it is easy to read off the grammar rules from the metamodel:

```
CallExpCS        ::= FeatureCallExpCS | LoopExpCS
```

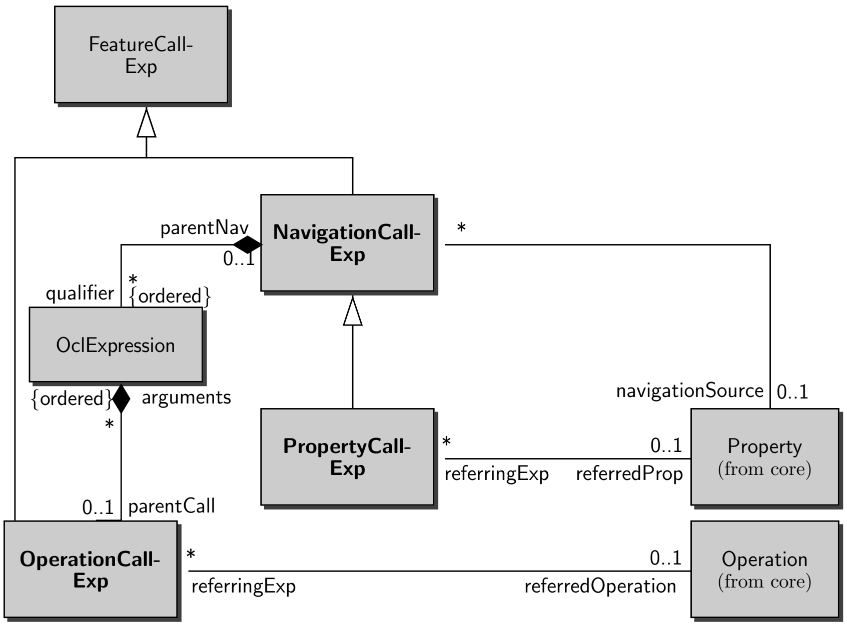


Fig. 5.8. Metamodel for OCL `featureCall` expressions

Looking at Figure 5.8 we find furthermore:

```
FeatureCallExpCS ::= NavigationCallExpCS |
                     PropertyCallCS | OperationCallExpCS
```

To gain an initial understanding, let us look at some examples taken from the constraints we had already encountered in Section 5.2.1.

The expressions `p1.cardNumber`, `p2.cardNumber` are property call expressions. So are `insertedCard@pre` and `self.insertedCard@pre`. Examples of operation call expressions are `p1.cardNumber <> p2.cardNumber`, `p1 <> p2`, `BankCard::allInstances()`. An example for the last category of feature call expressions, i.e., an example for a navigation call expression (taken from Figure 5.4 on page 255) is `self.role.assigned_users` or `role.assigned_users`.

Next we have a closer look at `OperationCallExp`. The concrete syntax grammar rule reads as:

```
OperationCallExpCS ::=
(A)  OclExpCS(1) sNameCS OclExpCS(2) |
(B)  OclExpCS '->' sNameCS'(' argumentsCS?')' |
(C)  OclExpCS'.'sNameCS ismarkedPreCS?'('argumentsCS?')' |
(D)  sNameCS ismarkedPreCS?'('argumentsCS?')' |
(G)  pathNameCS'('argumentsCS?')' |
(H)  sNameCS OclExpCS
```

As usual in grammar formalisms, a non-terminal with a question mark is optional. The rule in the standard lists additional clauses (E) and (F), which in our version are subsumed by (D) and (C). Here are typical examples for all six cases of operation call expressions:

```
(A)  wrongPINCounter + 1
     wrongPINCounter < 2
     wrongPINCounter = wrongPINCounter + 1
     insertedCard <> null
(B)  self.role.assigned_users -> notEmpty()
     s -> union(s2)
(C)  self.insertedCard.pinIsCorrect()
     self.insertedCard.pinIsCorrect@pre()
(D)  pinIsCorrect()
     pinIsCorrect()@pre
(G)  BankCard::allInstances()
(H)  -wrongPINCounter
     not cardIsInserted()
```

In the above, `sNameCS` is our shorthand notation for `simpleNameCS`. This is a string of symbols without further restrictions. There are of course additional well-formedness conditions for each of the eight production rules that make sure that the instance of `sName` is the name of an operation with the correct typing that is available in the OCL library or in the UML model the expression is attached to. First, `pathNameCS` is a non-empty sequence of simple names separated by "`::`". Applying rule (G) requires to check that `pathNameCS` ends in `className::opName()` where `className` does occur in the context diagram and `opName()` is a static operation declared in this class. `className` may optionally be prefixed by package names or might be implicit. Finally, we observe that case (D) is the same as (C) only with implicit source expression. Typically the implicit source could be the variable `self`.

Now let us look at the other two subclasses of feature call expressions. In the rules to follow we skip the rule versions for implicit source expressions.

```
(A)  PropertyCallExpCS ::= OclExpressionCS'.'
                                 sNameCS isMarkedPreCS?
(C)  PropertyCallExpCS ::= pathNameCS
```

Here `sNameCS` must match a suitable attribute name. (C) covers the case that the attribute is static.

```
(A)  NavigationCallExpCS ::= AssociationEndCallExpCS
(B)  NavigationCallExpCS ::= AssociationClassCallExpCS
(A)  AssociationEndCallExpCS ::= OclExpressionCS'.'
       sNameCS('['argumentsCS']')? isMarkedPreCS?
(A)  AssociationClassCallExpCS ::= OclExpressionCS'.'
       sNameCS('['argumentsCS']')? isMarkedPreCS?
```

Note that the rules for `AssociationEndExpCS` and `AssociationClassExpCS` are literally identical. The difference is that in the first rule `sNameCS` has to match the name of an association end and in the second rule `sNameCS` has to match the name of an association class available in the context UML model. The optional arguments within square brackets take care of qualifiers attached to association ends or classes.

This is all we want so say on feature call expressions. Next we turn to loop expressions, consult Figure 5.6. Of the two subclasses of the metaclass `LoopExp` we consider `IteratorExp` here and defer `IterateExp` to Sect. 5.2.4 on advanced topics.

```
(A)   IteratorExpCS ::=
        OclExpressionCS '->' sNameCS
        ('(VarDecl, (',' VarDecl)? '|')? OclExpressionCS')'
(B)   IteratorExpCS ::= OclExpressionCS'.'sNameCS'('argCS?')'
(C)   IteratorExpCS ::= OclExpressionCS'.'sNameCS
(D)   IteratorExpCS ::=
        OclExpressionCS'.'sNameCS ('['argumentsCS']')?
(E)   IteratorExpCS ::=
        OclExpressionCS'.'sNameCS ('['argumentsCS']')?
```

Table 5.2. Iterators from the OCL standard library

`exists`	`any`	`select`
`forAll`	`one`	`reject`
`isUnique`	`collect`	`collectNested`
`sortedBy`		

A complete listing of the built-in choices for `sNameCS` in (A) is shown in Table 5.2. New iterators may be added. The following are correct iterator expressions:

```
(A1)  source '->' 'select' '(' p '|' body ')'
(A2)  source '->' 'select' '('  body ')'
```

Assume that `source` evaluates to a collection $s = \{a_1, \dots, a_2\}$. Then the whole expression evaluates to the subset of those elements $a_i \in s$ that satisfy `body`. If the type of `source` is not a collection type it is implicitly turned into one, with the understanding that in the evaluation an object is replaced by the singleton set containing this object.

For the remaining rules (B) to (C) it is required that the source expressions be of collection type. They are all shorthand notations for a `collect` iterator. For example, an expression `source.attribute` is shorthand for

```
source -> collect(p | p.attribute) .
```

If again `source` evaluates to $s = \{a_1, \dots, a_n\}$, then the result of the whole expression is the set $\{a_i.\mathit{attribute} \mid a_i \in s\}$.

So far we have considered stand-alone OCL expressions. We now turn to the syntax OCL offers to explicitly relate constraints to UML model elements. We only discuss invariants, pre- and postconditions, and definitions, ignoring initial value, derived invariants, body, and guard expressions.

The generic form of invariants is:

— OCL —
```
context (x1,..,xk:)?classPath
inv (invName)?: expression
```
— OCL —

For operations contracts the generic form looks like this:

— OCL —
```
context (x1,..,xk:)?classPath::op(p1:T1,..,pn:Tn):T
pre     (prename1)?:  precondition1
post    (postname1)?: postcondition1
        :
pre     (prenamek)?:  preconditionk
post    (postnamek)?: postconditionk
```
— OCL —

For definitions, the generic context is shown below, where the left-hand sides are either variables or operation declarations:

—— OCL ——
```
context classPath
def:     lhs1 = ex1
         :
         lhsk = ex2
```
—— OCL ——

As an example for a definition we reproduce the definition of the least common supertype from the UML metaclass `Classifier` that had been used in the discussion of `if` expressions.

—— OCL ——
```
context Classifier
def:     commonSuperType(c:Classifier):Classifier =
         Classifier.allInstances() ->  select(cst |
            c.conformsTo(cst) and self.conformsTo(cst) and
            not  Classifier.allInstances() ->  exists(t |
             c.conformsTo(t) and self.conformsTo(t) and
             t.conformsTo(cst) and t <> s))
          ->  any()
```
—— OCL ——

For an explanation of `any()` see Figure 5.5. There is also a construct for definitions local to a single expression, e.g.:

—— OCL ——
```
context ATM::enterPIN(pin: Integer
pre:     let cardInserted = self.insertedCard <> null
         in
         cardInsterted and not self.customerAuthenticated
```
—— OCL ——

In both constructs `def` and `let` the variable or operation to be defined may also occur on the right-hand side, i.e., arbitrary, mutual recursive definitions are possible.

5.2.3 OCL Semantics

We define a precise meaning for OCL expressions indirectly by translating them to first-order logic, in some cases to dynamic logic, and then referring to the semantics explained in Chapters 2 and 3.

Signature

First we fix the signature of the target language. The types occurring in the context of the OCL expressions to be translated directly constitute a type hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ as defined in Sect. 2.1. This hierarchy includes the types $\bot$ and $\top$, even though they have no corresponding OCL type. For every type B there are the types $Set(B)$, $Sequence(B)$, etc. The functions and predicates in the signature Σ of the target language are determined as follows:

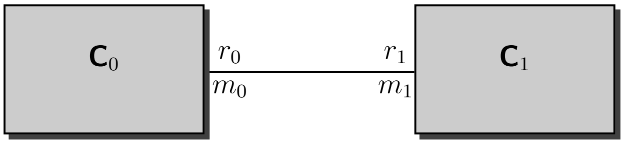


Fig. 5.9. A generic association

1. For every binary association r between classes C_0 and C_1 with role names r_0, r_1 and multiplicities m_0, m_1, see Figure 5.9, there are non-rigid function symbols in Σ with the following typing

$$\begin{array}{lll} r_1 : C_0 \to C_1 & \text{if} & m_1 = 1 \\ r_1 : C_0 \to Set(C_1) & \text{if} & m_1 \neq 1 \\ r_1 : C_0 \to Sequence(C_1) & \text{if} & m_1 \neq 1 \text{ and the association end at } C_1 \\ & & \text{is marked } \ll ordered \gg \end{array}$$

   Likewise there is a function symbol $r_0 : C_1 \to C_0$ if $m_0 = 1$ etc. In case that $m_0 = m_1 = *$, a binary predicate symbol $r : C_0 \times C_1$ is introduced in addition.
2. For n-ary associations r, an n-ary predicate symbol of the appropriate typing occurs in Σ.
3. For every attribute att in class C with result type C_r, there is a unary function $att : C \to C_r$ in Σ (if the attribute is static, the function is a constant of type C_r).
4. For every query operation op in class C with parameters of types $C_1, \dots C_n$ and result type C_r, there is an $(n+1)$-ary function symbol op in Σ with $op : C \times C_1 \times \dots \times C_n \to C_r$ (if the operation is a static the typing reduces to $op : C_1 \times \dots \times C_n \to C_r$).
5. The signature Σ contains names for all operations in the OCL standard library.
6. If C is an association class attached to an association r between classes C_1 and C_2 then function symbols $c_1 : C \to C_1$ and $c_2 : C \to C_2$, or $c_1 : C \to Collection(C_1)$ and $c_2 : C \to Collection(C_2)$ depending on the multiplicities of association r, are available in Σ.

All the functions and predicate symbols introduced in the above list are non-rigid symbols.

In this presentation we use the same names for the OCL entities and their counterparts in first-order logic with the exceptions shown in Table 5.3. Functions and predicates introduced in predicate logic as counterparts of functions in the OCL library are rigid symbols.

Table 5.3. Traditional names for Boolean and set operations

OCL	Logic	OCL	Logic
not	!	x.intersection(y)	$x \cap y$
and	&	x.union(y)	$x \cup y$
or	|	x.includes(y)	$y \in x$
implies	−>	x.excludes(y)	$y \notin x$
x.including(y)	$x \cup \{y\}$	x.includesAll(y)	$y \subseteq x$
x.excluding(y)	$x \setminus \{y\}$	x.isEmpty()	$x \doteq \emptyset$
x.excludesAll(y)	$x \cap y \doteq \emptyset$	x.notEmpty()	$x \neq \emptyset$
x.equals(y)	$x \doteq y$	x <> y	$x \neq y$

The expression `allInstances()` is a static method in the OCL standard library. It is inherited by all types `T` extending `OclAny`, which is in particular the case for all classifiers from the UML model. For any such T there is a non-rigid constant $T :: allInstances()$ of type $Set(T)$ in our signature Σ.

The translation of iterators will be deferred for the moment.

Semantics of Expressions Without Iterators

Once a type hierarchy and a signature Σ are fixed, we can form well-sorted terms (⇒ Sect. 2.3). Translating OCL expressions, for the moment without iterators, into terms of first-order typed logic amounts to nothing more than changing from one concrete syntax to another. In addition we view Boolean functions as predicates. The OCL expression

```
insertedCard <> null and not customerAuthenticated
```

from Figure 5.2 on page 252 now reads

$$insertedCard(\mathrm{self}) \neq null \;\&\; !\, customerAuthenticated(\mathrm{self}) \;\; .$$

The context information tells us, that this expression plays the role of an invariant. The variable `self` is thus implicitly understood as quantified over all existing instances of `ATM`. The translation of the invariant into the KeY input language thus is:

KeY

```
\forall ATM x;(x.<created> ->
   insertedCard(x) !=  null & customerAuthenticated(x))
```

KeY

In our logic quantification ranges over all instances of a class, also over those not yet created. So, the restriction `x.<created>` had to be added to capture the meaning of the OCL constraint correctly (⇒ Sect. 3.6.6). Since this will happen frequently in rest of this section we use $\dot{\forall} x.\phi$ and $\dot{\exists} x.\phi$ as abbreviations for $\forall x.(x.$`<created>` $\text{->}\ \phi)$ and $\exists x.(x.$`<created>` $\&\ \phi)$, respectively. The above invariant could thus be written as:

$$\dot{\forall} ATM\ x.(\mathit{insertedCard}(x)\ !\doteq \mathit{null}\ \&\ \mathit{customerAuthenticated}(x))\ .$$

In addition to what we have said so far there are also symbols `f@pre` in Σ for any $f \in \Sigma$ that is not already suffixed with `@pre`. Thus,

```
pin = insertedCard@pre.correctPIN
```

translates to

$$\mathit{pin}(\mathrm{self}) = \mathit{correctPIN}(\mathit{insertedCard@pre}(\mathrm{self}))\ .$$

In translating associations, see again Figure 5.9, one of the function symbols added to Σ already carries all the information. Nevertheless the redundancy to have one function symbol for each direction is highly desirable. But, when reasoning with terms over Σ we need axioms expressing the interrelations between them:

$$\begin{array}{ll}
\dot{\forall} C_0\ x.\dot{\forall} C_1\ y.(r_1(x) \doteq y \text{ <-> } r_0(y) \doteq x) & \text{if } m_0 = m_1 = 1 \\
\dot{\forall} C_0\ x.\dot{\forall} C_1\ y.(y \in r_1(x) \text{ <-> } r_0(y) \doteq x) & \text{if } m_0 = 1, m_1 \neq 1 \\
\dot{\forall} C_0\ x.\dot{\forall} C_1\ y.(r_1(x) \doteq y \text{ <-> } x \in r_0(y)) & \text{if } m_0 \neq 1, m_1 = 1
\end{array}$$

Similar formulas have to be added for multiplicities m different from 1 and $*$. Finally, we need axioms to reason about constants of the form `B::allInstances()`:

$$\dot{\forall} \mathit{Object}\ x.(x \in \mathrm{B}::\mathit{allInstances()} \text{ <-> } x \sqsubseteq B)\ .$$

It is important to notice that the type $Set(T)$ is treated on the same footing as any other type in our first-order logic. There is no commitment that in an interpretation I the domain $I(Set(T))$ consists of all (finite) subsets of $I(T)$. A formula like

$$\dot{\forall} T\ x.\dot{\exists} Set(T)\ u.\dot{\forall} T\ z.(z \in u \text{ <-> } \psi(x))$$

need not be universally valid for arbitrary ψ. If we want certain relationships between $I(Set(T))$ and $I(T)$ to hold, we have to add axioms to enforce it. We insist that the basic set theoretic operations are defined and have their usual

Table 5.4. First-order translations of some iterators

OCL	`e0->forAll(x | e1)`
FOL	$\dot{\forall} x.(x \in [\mathrm{e0}] \mathrel{{-}{>}} [\mathrm{e1}])$
OCL	`e0->exists(x | e1)`
FOL	$\dot{\exists} x.(x \in [\mathrm{e0}] \mathbin{\&} [\mathrm{e1}])$
OCL	`e0->select(x | e1)`
FOL	$s_{e0,e1}$ (new symbol) with definition $\dot{\forall} x.(x \in s_{e0,e1} \mathrel{{<}{-}{>}} (x \in [\mathrm{e0}] \mathbin{\&} [\mathrm{e1}]))$
OCL	`e0->collect(x | e1)`
FOL	$c_{e0,e1}$ (new symbol) with definition $\dot{\forall} z.(z \in c_{e0,e1} \mathrel{{<}{-}{>}} \dot{\exists} x.(x \in [\mathrm{e0}] \mathbin{\&} z \doteq [\mathrm{e1}]))$
OCL	`e0->isUnique(x | e1)`
FOL	$\dot{\forall} x.\dot{\forall} y.(x \in [\mathrm{e0}] \mathbin{\&} y \in [\mathrm{e0}] \mathbin{\&} [\mathrm{e1}] \doteq \{x/y\}[\mathrm{e1}] \mathrel{{-}{>}} x \doteq y)$
OCL	`e0->any(x | e1)`
FOL	$sk_{x,e0,e1}$ (new symbol) with definition $\dot{\exists} x.(x \in [\mathrm{e0}] \mathbin{\&} [\mathrm{e1}]) \mathrel{{-}{>}} sk_{x,e0,e1} \in [\mathrm{e0}] \mathbin{\&} \{x/sk_{x,e0,e1}\}[\mathrm{e1}]$

meaning. Thus we know, that for every finite subset $\{t_1, \ldots, t_n\}$ of $I(T)$ there is an $s \in Set(T)$ such that $t \in s$ is exactly true for $t = t_i$ for some $1 \leq i \leq n$. This is known as the *Henkin semantics* of higher order logic. We also insist that for every $Set(T)$ the following axiom is satisfied:

$$\dot{\forall} Object\ x.\dot{\forall} Set(T)\ u.(x \in u \mathrel{{-}{>}} x \sqsubseteq T)\ .$$

Semantics of Iterators

The OCL Standard Library is not systematic in the way it defines the meaning of its expressions. The union operation `union(s:Set(T)):Set(T)` on sets e.g., is defined via postconditions:

```
— OCL —
post: result->forAll(elem |
             self->includes(elem) or s->includes(elem))
post: self ->forAll(elem | result->includes(elem))
post: s ->forAll(elem | result->includes(elem))
— OCL —
```

It could just as well have been defined using the `iterate` construct:

```
— OCL —
self -> union(s : Set(T)) : Set(T) =
self -> iterate( x ; u:Set(T) = s | u->including(x))
— OCL —
```

On the other hand, `select` is defined in the standard as an iterator, but could just as well have benn characterised by postconditions:

```
—— OCL ——
source -> select(iterator | body)

post: result -> forAll(e | source.includes(e))
post: result -> forAll(e | body)
post: source -> forAll(e | body implies result.includes(e))
—— OCL ——
```

It is easy to see that in fact all iterators can be defined this way using only `forAll` and `exists`. to translate OCL iterators into first-order logic. See the summary in Table 5.4, where [`e`] denotes the first-order logic (FOL) translation of OCL expression `e`,[2] and $\{x/t_0\}t_1$ is the term resulting from t_1 by replacing all occurrences of variable x by the term t_0.

To illustrate how the new symbols, introduced in Table 5.4, are used, let us reconsider the expression from the invariant (5.3) on page 254:

```
validCardsCount = BankCard::allInstances() ->
                  select(not invalid ) -> size()
```

which translates to

$$\mathit{validCardsCount}(\mathrm{self}) = \mathit{size}(a)$$

plus the definition

$$\dot{\forall} x.(x \in a <-> x \in \mathit{BankCard} :: \mathit{allInstances}() \ \& \ \mathit{invalid}(x)) \ .$$

Comparing the first-order logic translation of `any(x|e)` with its definition in Figure 5.5 one might object that it does not take into account that the new constant should denote the first element of its kind. But notice that the operation `asSequence` is performed with respect to an unknown order. Choosing the first element in an arbitrary order amounts to choosing an arbitrary element.

Operation Contracts

An operation contract

```
—— OCL ——
context C::op()
pre:    pre
post:   post
—— OCL ——
```

[2] The same notation will later be used to denote translated JML expressions, but there will hardly be occasions for confusing both.

is translated into the dynamic logic formula

$$[pre] \mathrel{->} \langle \texttt{C::op()} \rangle [post] \ .$$

We notice, that this translation adopts the total correctness semantics, i.e., termination of the operation is required. We should also point out that the above translation treats the contract in isolation. The whole picture would also include invariants that can be assumed in proving the above implication (⇒ Chap. 8).

The KeY tool also offers the partial correctness semantics translation

$$[pre] \mathrel{->} [\texttt{C::op()}][post]$$

as an option. Using the modal operator $\langle\rangle$ in the total correctness semantics treats abrupt termination as non-termination. If you want a postcondition to also hold after abrupt termination the contract is translated:

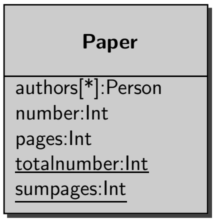
$$[pre] \mathrel{->} \langle \texttt{try\{C::op()\}catch(java.lang.Throwable exc)\{\}} \rangle [post] \ .$$

See also the definition of $\mathrm{Prg}_{op}()$ in Sect. 8.2.3. How abrupt termination is handled is explained in Sect. 3.6.7.

If the postcondition `post` contains as a subexpression `a@pre(exp)` the translation is [Baar et al., 2001]:

$$\big([pre] \ \& \ \dot{\forall} x.(a@pre(x) \doteq a(x))\big) \mathrel{->} \langle \texttt{C::op()} \rangle [post] \ .$$

Here, $a@pre$ is a new function symbol, which in particular does not occur in the body of op. On the other hand a will normally occur in the code of op. The newly added premiss $\dot{\forall} x.(a@pre(x) \doteq a(x)$ outside the scope of the modal operator allows one to conclude that the value of $a@pre(x)$ after execution of C is the value of $a(x)$ before.



```
—— OCL ——
context Paper
inv: Paper::allInstances() ->
     iterate(x:Paper;y:Integer = 0|y+x.pages)
      = Paper::totalnumber
—— OCL ——
```

Fig. 5.10. Example of a constraint with `iterate`

5.2.4 Advanced Topics

The Iterate Construct

As the first advanced construct we now consider the `iterate` construct, the second kind of loop expression, that we had skipped in Sect. 5.2.2, see Figure 5.6 for its position within the metamodel. Figure 5.10 shows an invariant of the class `Paper` which expresses that the static attribute `totalnumber` of this class equals, at all times, the sum of the `pages` attribute taken over all instances in the class. The general form of an iterate expression is given in Figure 5.11, which uses the names for association ends from the metamodel. The following restrictions apply:

1. variable y ist different from x,
2. variable y does not occur in the term t,
3. variables x and y do not occur in t_0,
4. the types of y and u coincide,
5. the type of t is a collection type $Collection(S)$ and x is of type S.

Given a model $\mathcal{M} = (\mathcal{D}, \delta, \mathcal{I})$ and an assignments β to local variables. The interpretation $\mathrm{val}_{\mathcal{M},\beta}(exp)$ for

$$exp \quad = \quad \texttt{t ->iterate(x;y = t0 | u)}$$

is obtained as follows. Let $A = \{a_1, \ldots, a_n\}$ be the evaluation $\mathrm{val}_{\mathcal{M},\beta}(t)$ of the source expression t. For the purposes of this definition, for any variable assignment γ, we use $\gamma[a, b]$ to denote

$$\gamma[a,b](z) := \begin{cases} a & \text{if } z = x \\ b & \text{if } z = y \\ \gamma(z) & \text{otherwise} \end{cases}$$

Using this notation we define

$$\begin{aligned} \beta_1 &= \beta[a_1, \mathrm{val}_{\mathcal{M},\beta}(t0)] \\ \beta_{k+1} &= \beta_k[a_{k+1}, \mathrm{val}_{\mathcal{M},\beta_k}(u)] \quad \text{for } k < n \end{aligned}$$

Then, $\mathrm{val}_{\mathcal{M},\beta}(exp) = \mathrm{val}_{\mathcal{M},\beta_n}(u)$.

This definition depends in general on the ordering of the set $\{a_1, \ldots, a_n\}$. If t is of type *Sequence* then, naturally, we use the order given by the sequence. In the other cases, it is the responsibility of the user to ensure independence from the order of evaluation.

All iterator expressions can in fact be defined in terms of an iterate expression, see Figure 5.5 below. The OCL standard is not systematic with respect to the definition of set theoretic operations. The union of two sets, e.g., is specified by an operation contract, but could just as well have been defined using `iterate`. The `select` operation on the other hand is defined via `iterate`, but could just as well have been specified by a postcondition.

We strongly recommend to avoid iterate expressions in OCL specifications, they are hard to read, they are at a low level of abstraction and they put an excessive burden on verification. If need arises, a new iterator could be defined. Its definition may use the iterate construct but in the specification only the iterator occurs.

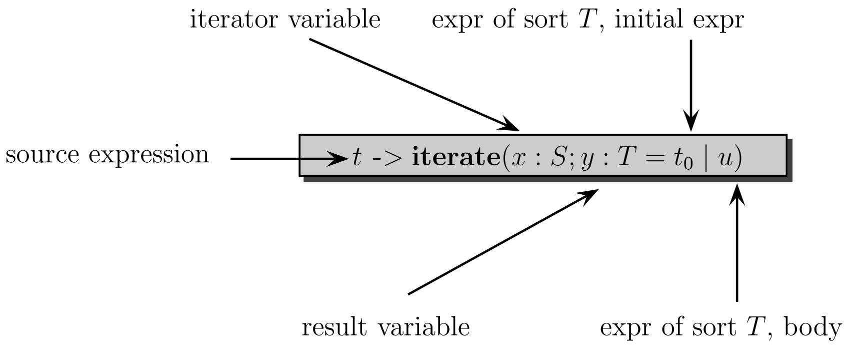

iterator variable
expr of sort T, initial expr
source expression
t -> **iterate**$(x : S; y : T = t_0 \mid u)$
result variable
expr of sort T, body

Fig. 5.11. Syntax of the `iterate` construct

Table 5.5. Definitions for some iterators

`t->forAll(x|a)`	=	`t -> iterate(x;y = `**`true`**`| y `**`and`**` a )`
`t->exists(x|a)`	=	`t -> iterate(x;y = `**`false`**`| `**`or`**` a )`

`t-> collectNested(x|u)`	=	`t -> iterate(x;y = Bag{} |`
		`y ->including(u))`

`t->collect(x|u)`	=	`t->collectNested(x|u) -> flatten()`

`t->select(x|a)`	=	`t-> iterate(x;y = Collection{} |`
		`if`` a `**`then`**` y.including(x) `**`else`**` y`

`t->any(x|e)`	=	`t->select(x|e)-> asSequence()-> first()`

`t-> flatten()` [a]	=	**`if`**
		`t.type.elementType.`
		`oclIsKindOf(CollectionType)`
		`then`
		`t -> iterate(c;acc:Bag = Bag{} |`
		`acc -> union(c->asBag))`
		`else`
		`t`
		`endif`

[a] This is the definition from the OCL standard, which only works for set nestings of level 2.

collectNested

In OCL2.0 the `collect` operation is defined via the more general operation called `collectNested`. The expression

```
self.role -> collectNested( r | r.assigned_users) ,
```

similar to the one considered in OCL example 5.4 on page 256, evaluates to $\{U_1, \ldots, U_k\}$ if for $r = r_i$ the OCL expression `r.assigned_users` evaluates to a set U_i.

Type Dependent Operations

There are three families of operators defined in `OclAny` that depend on types.

1. `oclAsType(T:TypeExp)`:`OclAny` $\rightarrow$ `T`
2. `oclIsTypeOf(T:TypeExp)`:`OclAny` $\rightarrow$ `Boolean`
3. `oclIsKindOf(T:TypeExp)`:`OclAny` $\rightarrow$ `Boolean`

Note, that the type expression is not an argument to these operations. It can be viewed as part of the operation's name. Translations to first-order logic are straight forward:

$$\begin{aligned}
[\texttt{e.oclAsType(T:TypeExp)}] &= (\mathrm{T})[\mathsf{e}] \\
[\texttt{e.oclIsTypeOf(T:TypeExp)}] &= [\mathsf{e}] \equiv \texttt{T} \\
[\texttt{e.oclIsKindOf(T:TypeExp)}] &= [\mathsf{e}] \sqsubseteq \texttt{T}
\end{aligned}$$

Exceptions in OCL

In contrast to JML (⇒ Sect. 5.3), the OCL language does not offer built-in support for talking about exceptions. This can be remedied by adding a new Boolean attribute `excThrown('T:TypeExp')` to any class. Types are attached to the `excThrown` attribute in the same way types are attached to the operations in the previous section.[3]

The use of `excThrown` only makes sense in the later stages of design when, e.g., the classes in the design model can be related to Java classes. The type `T` should then be a subtype of `Exception`. The easiest way to technically realize the use of `excThrown` would then be to automatically add the corresponding attribute `excThrown` to the Java class `Object`. Also `excThrown` can only be used in postconditions. A constraint

—— OCL ——
```
context C0::op(x1:D1,..,xn:Dn):C1
pre:    e0
post:   e1
```
—— OCL ——

with `excThrown('T1:TypeExp')`, ..., `excThrown('Tk:TypeExp')` occurring in the postcondition `e1` is translated into the dynamic logic proof obligation

—— KeY ——
```
 ==>
\forall T1 x1; .. \forall Tn xn;(
  x1.<created> & .. & xn.<created> & [e0]
  ->
```

[3] This feature is at the time of this writing only implemented in a simplified form in KeY.

```
  \< boolean thrownT1 = false;
     :
     boolean thrownTk = false;
     try {C0::self.op(x1,..,xn);}
     catch (java.lang.Throwable exc) {
      thrownT1 = exc instanceof T1;
      :
      thrownTk = exc instanceof Tk; }
  \>
[e1]*
```
KeY

Java

```
/**
 * @postconditions self.x< 0 implies
 *                          excThrown('IllegalArgumentException')
 */
public void positive()
  throws IllegalArgumentException {
   if (this.x >= 0) { do something; }
   else { throw new IllegalArgumentException(); };
}
```
Java

Fig. 5.12. Postcondition referring to exceptions

Here `thrownTi` are new local Boolean program variables, whose names are composed of the string `Thrown` appended with the string `Ti`, and `[e1]*` arises from `[e1]` by replacing all occurrences of `excThrown('Ti:TypeExp')` by `thrownTi`.

Figure 5.12 contains a concrete example of a Java program with a postcondition that refers to exceptions. For a change we have used a different way to attach an OCL condition to an operation, i.e., by placing it as a specially tagged comment directly into Java code in front of the method it refers to. Here is the translated proof obligation in dynamic logic:

KeY

```
\< boolean thrownIllegalArgumentException = false;
   try { TrivialExc()::self.positive (); }
   catch (java.lang.Throwable thrownExc) {
     thrownIllegalArgumentException=thrownExc
     instanceof java.lang.IllegalArgumentException; }
\> (self.x < 0 -> thrownIllegalArgumentException = TRUE)
```
KeY

Another example program of the same kind is shown in Figure 5.13 with the following dynamic logic proof obligation:

```
— KeY —
 ==>
\<  boolean thrownIllegalArgumentException = false;
    try { TrivialExc1()::self.positive (); }
    catch (java.lang.Throwable thrownExc) {
      thrownIllegalArgumentException=thrownExc
       instanceof java.lang.IllegalArgumentException; }
\> thrownIllegalArgumentException != TRUE
— KeY —
```

```
— Java —
/**
 * @postconditions not excThrown('IllegalArgumentException')
 */
public void positive()
  throws IllegalArgumentException {
    try{
      if (this.x >= 0) { this.b = true; }
      else { throw new IllegalArgumentException(); }
    } catch (java.lang.Throwable exc) {
      this.b = false;
    }
  }
— Java —
```

Fig. 5.13. Another postcondition referring to exceptions

Miscellaneous

In the OCL constraints throughout this chapter, we frequently made use of the constant **`null`**. We extended OCL by assuming that any UML class diagram implicitly contains a class `Null` that is a subclass of every existing class in the diagram and whose only element is **`null`**. For all attributes `attr` that `Null` inherits the value of **`null`**`.attr` is undefined. Since OCL2.0 there is now an OCL type `OclVoid` that is the only instance of the metaclass `VoidType` and in turn contains as only element the object **`null`**. So, you could identify the types `Null` and `OclVoid` if you wished. We think of our solution as a first step towards defining an OCL profile for Java specification. We stick to it for the moment till the discussion on what is in general regarded as a null object has reached a consensus.

The OCL standard uses a three-valued logic to treat undefinedness. We deviate from this. In our logic all functions are total and undefinedness is handled by underspecification as explained in the sidebar 3.3.1 on page 90. See also Sect. refsect11:partmod.

If `a` and `b` are inherited attributes in class `Null` then in all snapshots **`null`**`.a` and **`null`**`.b` are defined, but we have no information on what the values are. Thus neither **`null`**`.a = `**`null`**`.b` nor **`null`**`.a != `**`null`**`.b` are valid. For a comparison of the various logical approaches to formalise undefinedness we recommend [Hähnle, 2005]. We want to emphasise that our semantics, defined by the translation into first-order logic faithfully models OCL in that an expression is undefined according to the OCL standard if and only if it is undefined in our translation semantics. The first difference is that we do not have an equivalent of the instance `invalid` which is the only element of the OCL type `OclInvalid`. The use of `invalid` can be easily avoided by using the query `oclIsInvalid()` on the the type `OclAny`. The second difference lies in the logic employed to deal with undefined statements. OCL uses a three-valued logic while KeY uses classical two-valued logic with underspecification.

5.3 Java Modeling Language

An increasingly popular specification language for Java projects is the *Java Modeling Language*, JML. Unlike UML the language is not standardised by an organisation like the OMG, the development is more a community effort lead by Gary T. Leavens, Iowa State University.[4] The nature of such a project entails that language details change, sometimes rapidly, over time and there is no ultimate reference for JML. Fortunately, for the items that we address in this introduction, the syntax and semantics are for the greatest part already settled in [Leavens et al., 2006]. Basic design decisions and extensive examples are described in [Leavens et al., 2003].

As the major difference to UML/OCL, JML focuses solely on the phases of software development in which source code is written. Moreover the only supported programming language is Java. JML talks *directly* about Java classes represented in source files. There is no need for separate UML class diagrams. Since specifications may also serve the purpose of documenting a program, it is most natural that specifications are directly *annotated* to the entities to which they refer. So if we have a class invariant for a Java class C, the JML representation of the invariant is directly written as comment (somewhat in the style of a Javadoc comment) into the class declaration of C. If it is not desired to include specifications into source code, it is also possible to add JML specifications in extra files, which contain copies of the source file signatures.

[4] See `www.jmlspecs.org`

The close integration of JML with Java allows one to use Java expressions, for instance the side-effect free boolean expression `atm.wrongPINCounter==0`, directly in invariants and operation contracts. This possibility makes writing specifications easily accessible for developers acquainted with Java. Moreover, JML is more easily adapted to the Java specific issues, such as abrupt termination.

In this section we start, as in the previous section, with some illuminating examples from our ATM scenario, before a more thorough introduction to JML's syntax and semantics follows.

5.3.1 JML by Example

Consider now a design phase in which concrete Java code has been written for a realisation of the ATM scenario. We assume that a Java class `ATM` is part of it. Immediately preceding its method `enterPIN`, the JML representation of the operation contract described in Sect. 5.1.1 is annotated as a comment starting with the symbol `@`. It has become customary to also end a JML comment with `@` though this is not mandatory.

This can be seen in the listing in Fig. 5.14. At a first glance, we see that the JML specification from lines 9 to 40 contains three blocks, each starting with **`public`** `normal_behavior`. These blocks represent three operation contracts as introduced in Sect. 5.1.1. In JML terminology operation contracts are called *specification cases* while *contract* refers to the collection of all specification cases; we continue to stick with the term operation contract. JML annotations come together with visibility modifiers subject to the same rules as in Java. These have no bearing on the semantics, the meaning of a **`public`** contract is the same as that of a **`private`** contract. On the other hand visibility modifiers are in many cases helpful to formulate sensible contracts. JML adopts the principle that a **`public`** invariant is not allowed to talk about **`private`** fields.

The JML keyword `normal_behavior` states that the contract implicitly includes the requirement that the method *must* not throw an exception.

Let us look more closely at the third operation contract (lines 30 to 39). There are three keywords starting clauses that are terminated by a semicolon:

`requires` The condition following this keyword describes a precondition of the contract. More precisely, the conjunction of all these conditions forms the precondition of the operation contract. The expression following the first `requires` clause on line 10 resembles a Java expression, and its meaning is in fact that of a boolean Java expression. So this part of the precondition says that before calling `enterPIN` the `insertedCard` field must not be **`null`**, in order to ensure the assertions formalised in this contract. Alternatively, instead of expressing the precondition of the operation contract in separate clauses, one could have equivalently used the and-operator `&&` and written (replacing lines 31 to 34):

Java + JML

```
public class ATM {

  private /*@ spec_public @*/
          BankCard insertedCard = null;
  private /*@ spec_public @*/
          boolean  customerAuthenticated = false;


  /*@ public normal_behavior
      requires  insertedCard != null;
      requires  !customerAuthenticated;
      requires  pin == insertedCard.correctPIN;
      assignable customerAuthenticated;
      ensures   customerAuthenticated;

      also

      public normal_behavior
      requires  insertedCard != null;
      requires  !customerAuthenticated;
      requires  pin != insertedCard.correctPIN;
      requires  wrongPINCounter < 2;
      assignable wrongPINCounter;
      ensures   wrongPINCounter
                    == \old(wrongPINCounter) + 1;
      ensures   !customerAuthenticated;

      also

      public normal_behavior
      requires  insertedCard != null;
      requires  !customerAuthenticated;
      requires  pin != insertedCard.correctPIN;
      requires  wrongPINCounter >= 2;
      assignable insertedCard, wrongPINCounter,
                 insertedCard.invalid;
      ensures   insertedCard == null;
      ensures   \old(insertedCard).invalid;
      ensures   !customerAuthenticated;
    @*/
  public void enterPIN (int pin) {
    // here the implementation follows
```

Java + JML

Fig. 5.14. A JML specification for `enterPIN`

— JML (5.5) —
```
requires  insertedCard != null
          && !customerAuthenticated
          && pin != insertedCard.correctPIN
          && wrongPINCounter >= 2;
```
— JML —

`ensures` All boolean expressions of an operation contract following this keyword form (again in the sense of a conjunction) the postcondition of the contract. Our first example of a JML expression which is no Java expression turns up in line 38; here `\old` occurs. Keywords special to JML within expressions, like `\old`, start with a backslash. This one serves the same purpose as the `@pre` construct. Unlike `@pre` it refers to a whole expression. So `\old(insertedCard)` refers to the value of `insertedCard` before executing `enterPIN`. There are some subtle problems with this way of referring to pre states which we discuss later in Sect. 5.4.

`assignable` This keyword is followed by a list (items separated with a comma) of what is allowed to change during the execution of the method. JML does not allow temporary modifications of the specified location during the call deviating from the definition of modifies clauses in Sect. 3.7.4.

The JML contracts in Fig. 5.14 though marked **`public`** refer to the *private* field `insertedCard`. This is not a legal JML expression and any correct checker would reject it. To override the default we may declare a private Java field to be treated by the specification as if it were public by the annotation `/*@ spec_public @*/`. For the `insertedCard` field this was done in line 3 in Fig. 5.14. We could have omitted the keywords **`public`** `normal_behavior` because JML would assume them by default.

Clearly something is wrong if `enterPIN` is called but `insertedCard` is still equal to **`null`**. In the contracts we have seen so far the caller of the method is responsible to establish this precondition. If he does not, then no commitment is made. We could however decide otherwise and require that if the precondition is not met an exception of a type `ATMException` is thrown and no customer is authenticated. This could be specified in JML with the help of `exceptional_behavior`, a `signals_only` clause and a `signals` clause:

— Java + JML (5.6) —
```
/*@ (* the contracts as defined above *)
  @ also public exceptional_behavior
  @   requires insertedCard==null;
  @   signals_only ATMException;
  @   signals (ATMException) !customerAuthenticated;
  @*/
public void enterPIN (int pin) {
// here the implementation follows
```
— Java + JML —

The `signals_only` clause says that only exceptions of type `ATMException` must be thrown and the `signals` clause specifies that in the case of a thrown `ATMException` the `customerAuthenticated` field is set to **`false`**.

Another detail worth mentioning already here is the use of side-effect free and terminating methods in JML expressions. This is, as was the case with OCL, perfectly legal. Such methods are called *pure* in JML terminology and must be annotated with the keyword `/*@ pure @*/`. We could, e.g., add the following method, which is clearly pure, in class `ATM`:

— Java + JML —

```
  public /*@ pure @*/ boolean cardIsInserted() {
    return insertedCard!=null;
  }
```

— Java + JML —

Now `cardIsInserted()` could replace `insertedCard != null` in all the contracts above.

The next example shows how invariants are written in JML. Again we want to formalise the property that different cards have different card numbers, compare the OCL constraint (5.2) on page 253. Clearly, this requires means that go beyond Java expressions. Universal quantification, syntactically quite similar to first-order logic, is used. The range of the quantification must only include the objects which are created. This can be achieved with the help of the expression `\created(`o`)`, which says that o is a created object. Since the resulting expression does not depend on *one* particular instance of `BankCard` it is referred to as a *static invariant*. The whole annotation to the class `BankCard` now reads:

— Java + JML (5.7) —

```
public class BankCard {
  /*@ public static invariant
    @  (\forall BankCard p1, p2;
    @    \created(p1) && \created(p2);
    @    p1!=p2 ==> p1.cardNumber!=p2.cardNumber)
    @*/
    private /*@ spec_public @*/ int cardNumber;
    // rest of class follows
}
```

— Java + JML —

Opposed to static invariants are *instance invariants*. They formalise properties of a particular instance, referred to by **`this`**. The OCL invariant (5.3) from page 254 on the class `CentralHost` reads in JML as follows:

— Java + JML (5.8) —

```
public class CentralHost {
  /*@ public instance invariant this.validCardsCount
    @                 == (\num_of BankCard p; !p.invalid)
    @*/
}
```

— Java + JML —

As in Java we could have skipped **this** in **this**.validCardsCount. An instance invariant contains an implicit universal quantification in that it requires that the stated property must evaluate to true *for all* created objects of its class.

We could use this to rewrite the JML static invariant (5.7) into an equivalent instance invariant:

— Java + JML (5.9) —

```
public class BankCard {
  /*@ public instance invariant
    @   (\forall BankCard p; this != p ==>
                 this.cardNumber != p.cardNumber)
    @*/
    private /*@ spec_public @*/ int cardNumber;
    // rest of class follows
}
```

— Java + JML —

5.3.2 JML Expressions

Every Java expression according to Gosling et al. [2000] that does *not* include operators with side-effect, like `e++`, `e--`, `++e`, `--e`, non-pure method invocation expressions, and assignment operators, is a JML expression. Any such expression e has a natural representation in KeY's first-order logic, which we denote by [e]. The JML reference manual [Leavens et al., 2006] does not contain a formal semantics of JML. The paper [Jacobs and Poll, 2001] roughly sketches a semantics of JML expressions in a higher-order logic that is a common abstraction of PVS and Isabelle/HOL.

The translation to first-order logic serves us as a precise definition of the meaning of JML expression. In Table 5.6, the mapping $e \leadsto$ [e] is defined for JML expressions e_0, e_1, and e_2.

For example, the JML expression

```
insertedCard != null && !customerAuthenticated;
```

is translated as follows to first-order logic:

Table 5.6. Mapping from JML and Java expressions to FOL (selected items)

JML Expression	first-order logic formula
`!`$e0$	![e0]
$e0$ `&&` $e1$	[e0] & [e1]
$e0$ `||` $e1$	[e0] | [e1]
$e0$`?`$e1$`:`$e2$	if [e0] then [e1] else [e2]
$e0$ `!=` $e1$	!([e0] $\doteq$ [e1])
$e0$ `>=` $e1$	[e0] >= [e1]

$$!(o.\texttt{insertedCard} \doteq \texttt{null}) \;\&\; !\,o.\texttt{customerAuthenticated} \doteq \mathrm{TRUE} \quad .$$

Note that this formula contains free occurrences of a variable o of type `ATM`, which is the **`this`** type the JML expression refers to.

Moreover JML introduces operators to express implication (`==>`) and logical equivalence (`<==>`).

Finally JML extends Java by *quantified* expressions. We have already seen an example of universal quantification at work in the JML annotation (5.7). Existential quantification works analogously. Table 5.7 summarises the first-order logic translations of these expressions. Note that quantifiers bind two expressions, the range predicate and the body expression with the semantics shown in the first-order logic column. A missing range predicate is by default **`true`**. Quantifiers are meant to range over all objects including the not yet created ones. This is in accordance with our definition of quantification in Sect. 3.3. In contrast to that, JML excludes **`null`** from the range of quantification.

Table 5.7. Mapping from new JML expressions to first-order logic (selected items)

JML Expression	first-order logic formula
$e0$ `==>` $e1$	[e0] -> [e1]
$e0$ `<==>` $e1$	[e0] <-> [e1]
`(\forall` T e`;`$e0$`;`$e1$`)`	\forall T e; (([e] !$\doteq$ `null` & [e0]) -> [e1])
`(\exists` T e`;`$e0$`;`$e1$`)`	\exists T e; ([e] !$\doteq$ `null` & [e0] & [e1])

In addition to these traditional quantifiers JML offers so called generalised and numerical quantifiers. We have already seen the `\num_of` quantifier which delivers the number of values of its quantified variable for which the expression in the second argument is true. Other such quantifiers are `\sum`, `\product`, `\min`, and `\max`. Translations of these expressions have to be done similarly as for OCL (see Sect. 5.2.3).

More on the translation of JML expressions can be found in [Engel, 2005].

5.3.3 Operation Contracts in JML

We now turn our attention to operation contracts in JML. We have already encountered operation contracts starting with `normal_behavior` and `exceptional_behavior` in Figure 5.14. These are, in fact, special cases of a general contract concept starting with the keyword `behavior` which we discuss now.

An operation contract consists of a number of *clauses* each starting with one of the keywords `requires`, `assignable`, `ensures`, `diverges`, `signals`, or `signals_only`.

The boolean expressions following the `requires` clauses specify (seen as a conjunction) the preconditions of the operation contract. All other clauses must be true only under the provision that all `requires` clauses hold.

The postcondition of an operation contract is spread over the `ensures`, `signals`, and `signals_only` clauses. `ensures` describes the postcondition in the case of *normal* termination of the operation. That is, *if* the operation terminates normally then all the boolean expressions following `ensures` must hold. The `signals` clause specifies what happens if the operation terminates *abruptly*. `signals` is not directly followed by a JML expression. Instead there is first a declaration of an exception type T, and then a boolean JML expression e. If abrupt termination is caused by an exception of type T then e must be true in the post-state. Note that e does *not* specify the condition which triggers the specified expression to be thrown; such conditions can be stated in the `requires` clause of an operation contract. Finally `signals_only` lists the types of exceptions that may at most be thrown by a method. As we have done for JML expressions, we can define the meaning of a JML postcondition by translating them into the first-order fragment of Java Card DL. The postcondition of a contract

— JML —
```
ensures E;
signals (ET1) S1;
.
.
.
signals (ETn) Sn;
signals_only OT1,...,OTm;
```
— JML —

is translated into

$$
\begin{array}{ll}
 & (\ \mathtt{e} \doteq \mathtt{null} \rightarrow [\mathrm{E}]) \\
\& & (\ \mathtt{e} \sqsubseteq [\mathrm{ET}_1] \doteq \mathrm{TRUE} \rightarrow [\mathrm{S}_1]) \\
 & \ \ \ldots \\
\& & (\ \mathtt{e} \sqsubseteq [\mathrm{ET}_n] \doteq \mathrm{TRUE} \rightarrow [\mathrm{S}_n]) \\
\& & (\ \mathtt{e} \sqsubseteq [\mathrm{OE}_1] \doteq \mathrm{TRUE}\ | \\
 & \ \ \ldots \\
 & \ \ |\ \mathtt{e} \sqsubseteq [\mathrm{OE}_m] \doteq \mathrm{TRUE})
\end{array}
$$

We assume in this translation that the operation stores a thrown exception causing abrupt termination in the variable `e`. If the operation terminates normally then `e` equals `null`.

`assignable` is followed by a list of expressions which specify locations of the program. When these expressions are translated into our first-order logic, the top-level operator must be a non-rigid function symbol representing a field symbol or an array access. As special symbols we allow the JML expressions `\nothing` (which is equivalent to the empty modifies set) and `\everything` (which means that every location is allowed to be modified). The semantics of assignable clauses follows Definition 3.62. The `diverges` clause consists again of a boolean JML expression. It specifies the condition which must hold before calling the operation if the operation does *not* terminate. This sounds complicated but fortunately in practice and also as a matter of normalisation this can be reduced to two cases. As one case, we specify `diverges` **`false`**, then, in case of non-termination, **`false`** must have been satisfied before the operation call. This is never the case. Thus, `diverges` **`false`** *requires* the operation to terminate. On the other hand one could specify `diverges` **`true`**, then non-termination is always allowed. It is quite easy to figure out, that we can use appropriate `requires` clauses and these two incarnations of `diverges` to express all termination behaviour we may desire.

We can summarise the requirements imposed by an operation contract for an operation *op* as follows: When *op* is called in any state that satisfies all the `requires` clauses then:

- If *op* terminates normally then all `ensures` clauses are satisfied.
- If *op* terminates abruptly with an exception of type ET then
  - all `signals(`ET'`)` clauses for exception types ET' where ET is a subtype of ET' are satisfied and
  - there is a `signals_only(`ET''`)` clause such that ET is a subtype of ET''.
- If *op* terminates (either normally or abruptly) then at most the locations specified by `assignable` are modified compared to the pre-state.
- If *op* does not terminate, then the `diverges` condition has been true before calling *op*.

Figure 5.15 depicts the meaning of the special contracts `normal_behavior` and `exceptional_behavior` in terms of `behavior` contracts. Abbreviations, like the use of `normal_behavior` instead of a more verbose `behavior`, occur quite often in JML, and the process of resolving them is referred to as *desugaring*. Extending this scheme to specification cases with more than one occurrence of the different clauses can naturally be done.

Some JML operation contracts even have no `behavior`, `normal_behavior`, or `exceptional_behavior` header at all. Instead they start with clauses (like `requires`, `ensures`, etc.) directly. Such operation contracts are called *lightweight* in JML jargon. All others are called *heavyweight*. There is only a

```
                                 behavior
normal_behavior                   requires R;
 requires R;                      assignable A;
 assignable A;          ==>       ensures E;
 ensures E;                       diverges D;
 diverges D;                      signals (Exception) false;

                                  behavior
 exceptional_behavior              requires R;
  requires R;                      assignable A;
  assignable A;         ==>        ensures false;
  diverges D;                      diverges D;
  signals (ET) S;                  signals (E) S;
  signals_only (OT);               signals_only (OT);
```

Fig. 5.15. Desugaring of `normal_behavior` and `exceptional_behavior`

small semantical difference of lightweight specifications compared to heavyweight specifications starting with `behavior`. In lightweight specifications most missing clauses default to `\not_specified`, which leaves different JML tools different options to treat the missing items. In KeY, always the same defaults as for heavyweight specifications are used. See Table 5.8 for lightweight and heavyweight defaults. The choices correspond to Table 5.1 in Sect. 5.1.

Table 5.8. Defaults for missing JML clauses

Clause	Lightweight default	Heavyweight default
`requires`	`\not_specified`	**`true`**
`assignable`	`\not_specified`	`\everything`
`ensures`	`\not_specified`	**`true`**
`diverges`	**`false`**	**`false`**
`signals`	`\not_specified`	`(Exception)`**`true`**
`signals_only`	All exception types declared in the JAVA method declaration	

We have already seen in the introductory examples that, when describing post-states, one needs to refer to the state before the method invocation. The `ensures` and `signals` clauses describe post-states so that the JML expressions used in these clauses may include the `\old` construct.

With `/*@ pure @*/` annotations, implicit additions to all operation contracts are implied. This can again be seen as a de-sugaring. The operation contracts for an operation annotated with `/*@ pure @*/` are equivalent to adding `assignable \nothing` and `diverges false` to all operation contracts which are available for the constrained operation.

JML dictates a stricter rule of inheritance of operation contracts than required in Sect. 5.1. Every contract for a method automatically applies to overridden methods, too. Syntactically this is signified by the fact that contracts for overridden methods must start with `also`, the keyword which conjoins several contracts for an operation. The contract inheritance policy has the effect that all subtypes of a type T are behavioural subtypes (see Sect. 8.1.3) of T [Leavens and Dhara, 2000].

5.3.4 Invariants in JML

JML distinguishes two types of class invariants: instance invariants and static invariants.

An *instance invariant* is a boolean JML expression containing explicitly or implicitly the variable **`this`**. An instance invariant is satisfied in a program state if it always evaluates to true when the value of **`this`** ranges over all instances of its class. Syntactically, instance invariants are comments (as usual, starting and ending with `@`) which are explicitly marked with `instance invariant` or, if the targeted type is a class, just as `invariant`.

As illustrated in Sect. 5.3.2, we can translate boolean JML expressions into first-order logic formulae. The characteristic property of instance invariants is that there is a free variable in the resulting formulae. Consider the JML invariant (5.9) in Sect. 5.3.1. It could be represented as follows as first-order logic formula containing a program variable o of type `BankCard`:

— KeY —

```
\forall BankCard p; p.<created> = TRUE ->
     o != p -> o.cardNumber != p.cardNumber
```

— KeY —

The variable o is, according to the semantics of invariants, implicitly universally quantified over all *created* objects of the respective type. For a uniform treatment of invariants, we make this quantification explicit. We obtain closed formulae. If ϕ is the "raw" translation of a boolean JML expression in an invariant and if o is the occurring free variable of type T, then

$$\backslash\text{forall}\ T\ o;\ \ (o.\texttt{<created>} \doteq \texttt{true} \text{ -> } \phi)$$

is defined to be the translation of the JML invariant. The translation of our example yields:

— KeY —

```
\forall Bankcard o; \forall Bankcard p;
  ( o.<created>=TRUE & p.<created>=TRUE & o != p
    -> o.cardNumber != p.cardNumber )
```

— KeY —

According to Leavens et al. [2006], instance invariants defined in a class C must hold at any *visible state* for any object o of C. Visible states for an object o are the states reached when a method of o (this includes non-static methods and static methods declared in C or a super class) is invoked or finished or when a constructor of o is finished. A further visible state is when no method or constructor of o is in progress. The latter means that invariants must be established, according to JML, when in a method of o another method is called. JML thus requires invariants to hold at intermediate states of an operation. In Chapter 8 we will deviate from the visible state semantics of JML, since it is overly strong to require invariants to hold at intermediate states.

The semantics of invariants is liberalised by the possibility of JML to declare methods with `/*@ helper @*/`. It is not required that invariants hold at the entry and exit states of such helper methods.

Static invariants do not refer to a special instance of the class they are defined in. This implies that static invariants can only refer to instance fields via quantification as in the example of the static invariant in Sect. 5.3.1. We have seen there that it was in that case possible to replace it with an equivalent instance invariant (5.9). So are static invariants necessary at all? Imagine we want to express that the static integer field `maxAccountNumber` in class `CentralHost` is always greater or equal to 0. Then we want to require this condition even in states in which no object of `CentralHost` is created at all. So it is of no use to add an instance invariant

—— JAVA + JML ——

```
public class CentralHost {
 /*@ public instance invariant maxAccountNumber >= 0 @*/
//...
```

—— JAVA + JML ——

which would need to hold only after the constructor call of the first instance of this class is finished. The following *static* invariant

—— JAVA + JML ——

```
public class CentralHost {
 /*@ public static invariant maxAccountNumber >= 0 @*/
//...
```

—— JAVA + JML ——

must however hold already after the static initialisation of `CentralHost` has finished, which is the desired property.

Static invariants must be explicitly declared as `static` (as above) or they are written into an interface declaration and just start with `invariant`.

5.3.5 Model Fields and Model Methods

The operation contracts and instance invariants we have seen so far may only talk about instance (and static) fields occurring in the Java program they annotate. Since instance fields may only occur in classes and not in interfaces, how would we write operation contracts and instance invariants for *interfaces*?

In our banking scenario we could extend the simple `BankCard` class into a card which allows one to collect bonus points as well. Whenever certain transactions are done with the card, a counter `bankCardPoints` on the card is increased. We also foresee the situation that the bonus point system will be used with other cards from other vendors than our bank. It may thus be a good idea to separate the interface of accessing bonus points from the `BankCard` class. We use a Java interface `IBonusCard`, which `BankCard` implements. A Java interface is definitely the best choice since we do not want to provide implementations, as for instance in an abstract class, for the other vendors, just the mere interface:

—— Java ——

```
  public interface IBonusCard {
    public void addBonus(int newBonusPoints);
  }
```

—— Java ——

As already mentioned, we may wonder how to add a suitable specification, since there are no fields to talk about in a Java interface. Here JML model fields are the solution. We simply *assume* that a field representing bonus points was available. Let us call it `bonusPoints` of type `int`. Since it is not a true field and just for specification purposes, we add it (as usual in JML) as comment and qualified with the key word `model`. In specifications, as in the operation contract for `addBonus` this field may then be referred to:

—— Java + JML ——

```
public interface IBonusCard {
   /*@ public instance model int bonusPoints; @*/

   /*@ ensures bonusPoints == \old(bonusPoints)+newBonusPoints;
     @ assignable bonusPoints;
     @ */
   public void addBonus(int newBonusPoints);
}
```

—— Java + JML ——

The specification says that the bonus points are increased by the number given as argument in the method `addBonus`.

You may wonder how we can relate concrete implementations like that of BankCard with model fields. Let us consider the implementation of addBonus in BankCard:

— Java —
```
public class BankCard implements IBonusCard{
  /*@ public instance model int bonusPoints; @*/
  /*@ also
    @   assignable bankCardPoints;
    @*/
  public void addBonus(int newBonusPoints) {
    bankCardPoints+=newBonusPoints;
  }
}
```
— Java —

Since JML operation contracts are inherited, the contract in IBonusCard is implicitly present at this method, but it specifies the change of field bonusPoints not that of bankCardPoints as the implementation does. We thus need to specify the relation between the concrete field and the model field. In our case the relation is simple: bonusPoints *exactly* corresponds to bankCardPoints; whenever we refer to bonusPoints in a specification, we mean bankCardPoints in the implementation. This is how we denote this in JML, added directly after the header of the class declaration:

— JML (5.10) —
```
/*@ private represents bonusPoints <- bankCardPoints; @*/
```
— JML —

The expression on the right side could in fact also be a more complicated expression. If for some reason the points stored on the bank card are 100 times the points credited by the addBonus method we could write:

— JML —
```
/*@ private represents bonusPoints <- bankCardPoints * 100;
  @*/
```
— JML —

In our translation to first-order logic, we can simply replace every occurrence of the model field with the expression

The represents clauses so far are called *functional abstractions* since the relation between model field and concrete field(s) is a function. There are also *relational abstractions*

— JML —
```
/*@ represents x \such_that A(x); @*/
```
— JML —

which relate concrete fields with the model field `x`; the relation must satisfy the axiom `A(x)`. The functional abstraction (5.10) can thus also be expressed as relational abstraction:

JML
```
 /*@ private represents bonusPoints
                  \such_that bonusPoints==bankCardPoints;
   @*/
```
JML

As Breunesse and Poll [2003] point out, the translation of model fields into a logical representation is non-trivial if $A(x)$ is not a function of x or if it is not a total function. KeY roughly follows one of the solutions in that paper: All occurrences of the model field in an expression are replaced by occurrences of a reference to a pure ("model") method m with no arguments and the same result type as the type of the model method. Method m is specified with an operation contract which (a) requires in its precondition that there is an x such that $A(x)$ holds, and (b) ensures that the result r of m satisfies $A(r)$.

5.3.6 Supporting Verification with Annotations

All JML annotations considered so far are obligations for verification: We are aiming to prove that the program satisfies the given specification. There are also other kinds of annotations which can be considered more as helpers for the verification process, such as loop invariants. For the program (3.1) on page 155 a loop invariant could be specified with JML as follows.

Java + JML
```
m = a[0]; i = 1;
while (i < a.length) {
  /*@  ensures \forall integer x;0 != x && x < i ;a[x] <= m;
    @  assignable m, i;
    @ */
  if (a[i] < m) then
    m = a[i];
  i++;
}
```
Java + JML

This example also shows the use of the assignable clause for loop bodies. This is at the time of this writing not a part of the official JML syntax, but is expected to be included soon.

5.4 Comparing OCL and JML

Advantages of OCL over JML:

1. OCL lives on a higher level of abstraction. A UML diagram can be annotated with OCL constraints before code is developed. Automatic generation of constraints from patterns (described in Chapter 6) as well as editing constraints parallel to natural language phrases (as detailed in Chapter 7) would be much harder if not impossible on code level.
2. As a consequence of the previous item, OCL is not committed to a particular programming language and better suited for model driven system development.
3. OCL is an OMG standard, though one has to admit that at the time of this writing the official standard draft still contains serious inconsistencies and many unfinished items.

Advantages of JML over OCL

1. JML is closer to Java code, which encourages its use by programmers and developers. In fact, today JML specifications are much more widespread than OCL specifications.
2. JML offers a greater variety of concepts on the implementation level, like exceptional behaviour, modifies clauses, and loop invariants.

JML is not standardised and its specification document is still very incomplete.

Referring to the Pre-state

It is a detail, but nevertheless instructive to compare the differences in referring to values in pre-states in OCL and JML, i.e., to compare OCL's `@pre` construct with JML's `\old`. The former can be attached to individual symbols while the second can only be applied to whole expressions. So, `o@pre.b@pre.c@pre`, `o.b@pre.c@pre`, `o.b.c@pre`, `o@pre.b.c@pre` are all legal OCL expressions while only `\old(o.b.c)` is allowed in JML, and would correspond to the first of the OCL expressions. The JML proponents argue that the explicit scoping of the `\old` construct make it easier to read. A more substantial difference is the fact, that the `@pre` construct is hard to implement for run-time checking in full generality. A drawback of the `\old` construct comes to the surface in the following specification problem. Suppose you want to state in a postcondition to a method `m` manipulating an array `a[]` and a field `idx` that the value `a[0]` equals the old value of the array at position `idx`. Now, `\old(a[idx])` would not do, since the value of `idx` in the pre-state would be used. We resorted to

— JML —
```
(\forall int x; x==idx; \old(a[x])==a[0]);
```
— JML —

On the other hand, one has to admit that OCL does not offer a built-in construct to model Java arrays. The sequence data type does not fit since it does not take into account that Java arrays are objects and also it declares operations, e.g., union or append, that do not make sense for arrays. As an extension of OCL we introduced functions `a.get(i)` and `a.length(i)` for array object `a` and integer `i`. The above discussed expression can now easily be written as `a.get@pre(i)`.

Modifies Clauses

Our semantics of the assignable or modifies clause deviates slightly from the semantics in JML. In the JML semantics, only the locations listed in the assignable clause can be assigned to during method execution. In the KeY semantics the locations contained in the modifier terms may be assigned to, it is only important that in the end the terms have the same value as before. We found no clear statement on OCL's position on the frame problem. The unofficial position seem to be that it is assumed that locations not contained in the postcondition cannot change. In [Baar, 2006] explicit extension of OCL to deal with the frame problem are proposed.

Range of Quantification

In JML, quantification extends over all elements of a given type and not only over all created or allocated elements. Since our logic uses the same semantics the static JML invariant (5.7) translates in

$$\forall p1.\forall p2.(p1 \mathrel{!\doteq} p2 \mathrel{-\!\!>} p1.\mathit{cardNumber} \mathrel{!\doteq} p2.\mathit{cardNumber})$$

where $p1, p2$ are variables of type `BankCard`. The instance JML invariant (5.9) on the other hand translates to

$$\forall \mathit{this}.(\mathit{this}.\texttt{<created>} \mathrel{-\!\!>} \forall p.(\mathit{this} \mathrel{!\doteq} p \mathrel{-\!\!>} \\ \mathit{this}.\mathit{cardNumber} \mathrel{!\doteq} p.\mathit{cardNumber}))$$

This discrepancy is attributable to the fact that implicit quantification of the variable **`this`** is treated differently from explicitly quantified variables; they are only meant to range over existing elements.

For a not created BankCard `o`, the value of `o.cardNumber` should be undefined. The validity of the two formulae above now depends on how undefinedness is modelled. In our logic we model undefinedness by underspecification which would make both formulae invalid.

In our logic we express the intended invariant by

$$\forall p1.\forall p2.(p1.\texttt{<created>} \mathrel{\&} p2.\texttt{<created>} \mathrel{-\!\!>} (p1 \mathrel{!\doteq} p2 \mathrel{-\!\!>} \\ p1.\mathit{cardNumber} \mathrel{!\doteq} p2.\mathit{cardNumber}))$$

The JML community is at the time of this writing considering the introduction of an attribute similar to `<created>`.

The shown first-order formula is also the correct translation of the OCL constraint (5.2). The OCL method `A::allInstances()` returns the set of all existing instances of `A`.

The semantics in Appendix A of the OCL standard draft also distinguishes between existing elements and reservoir elements waiting to be created. But there seems to be no possibility to talk about these element in the language.

Integers

The following JML specification for the integer square root method can be found in [Leavens et al., 2003]

— Java + JML (5.11) —

```
/*@ requires y >= 0;
  @ ensures
  @  \result * \result <= y &&
  @ y < (abs(\result)+1) * (abs(\result)+1);
  @ */
  public static int isqrt(int y)
```

— Java + JML —

In [Chalin, 2003], the following flaw has been pointed out. For $y = 1$ and $\backslash result = 1073741821 = \frac{1}{2}(max\_int - 5)$ the above postcondition is true, though we do not want 1073741821 to be a square root of 1. The problem arises since JML uses the Java semantics of integers which yields

$$\begin{aligned} 1073741821 * 1073741821 &= -2147483639 \\ 1073741822 * 1073741822 &= 4 \end{aligned}$$

The findings in [Chalin, 2003] seem to indicate that programmers tend to have the mathematical integers in their minds and frequently make mistakes in JML specification. Chalin proposes the extension JMLa that includes a new primitive type `\bigint` of arbitrary precision integers, i.e., the mathematical integers.

The KeY system offers the option to choose between the mathematical and the Java semantics of integers (⇒ Chap. 12).

In OCL quantification over all integers is not possible. Its semantics only allows finite sets. The expression `Integer::allInstances() -> forAll(e)` is thus undefined.