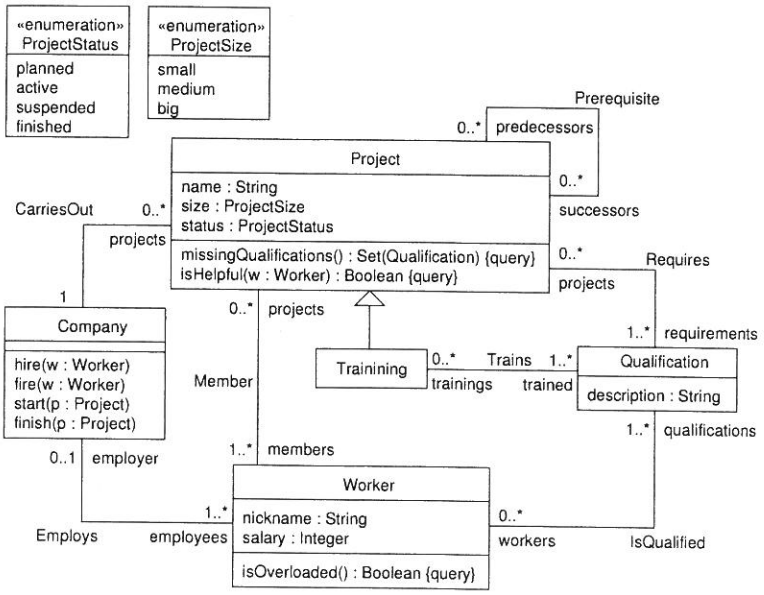

Figure 3.1: Example class diagram 'Project World'

3.2.1.2 Types

A few primitive types including numbers and strings are predefined in UML and OCL. For example, Project World uses the primitive type *String* for the attributes *name, nickname* and *description*, and the primitive type *Integer* for the attribute *salary*.

We consider types in-depth later. For now, we assume that there is a signature $\Sigma = (T, \Omega)$ where T is a set of type names, and Ω is a set of operations over types in T. The set T includes basic types such as *Integer*, *Boolean*, and *String*. All type domains include the value $\perp$ that allows to operate with undefined values. Operations in Ω include, for example, the usual arithmetic operations for integers.

3.2.1.3 Classes

"The central concept of UML for modeling entities of the problem domain is the class. A class provides a common description for a set of objects sharing the same properties.

Chapter 4

SOIL

In this chapter we introduce a simple OCL-based imperative language (SOIL) which is based on UML object models and OCL expressions as defined in [OMG06] and [Ric02] and rendered in Chap. 3. SOIL embeds OCL in a modular way as given by Def. 2.1 in Sect. 2.5.

The language is not meant to be a particular rich programming language. It is meant as a simple but still practically useful example of how to define a language that embeds OCL, providing a sound denotational semantics. Although it is a simple language, it has been successfully employed as a general purpose scripting and programming language in the UML-based Specification Environment, as described later in Sect. 5.1. The spirit of SOIL has also been successfully transferred into an industrial project in the construction of the eGovernment MDA tool XGenerator2, as described in Sect. 5.2.

This chapter is structured as follows: We first informally introduce the language SOIL in Sect. 4.1. Then, we formally define the language in Sections 4.2–4.4. In Sect. 4.5, we revisit the type-safety of the language. In Sect. 4.6, we discuss an approach to an extended handling of error conditions. Finally, in Sect. 4.7, we compare SOIL to other OCL-based imperative programming languages.

4.1 Introduction

The language SOIL provides a way to define imperative programs based on UML information structures (class and object diagrams) and OCL expressions, enabling the construction of executable UML models. The language comprises manipulation of objects and their properties, scoped variables, and flow control such as conditional execution, loops, and operation invocation. SOIL can be used to provide definitions for operations with side-effects, i. e.,

to provide method bodies for UML operations. The language is statically typed and provides implicit definition of local variables.

Unlike non-modular OCL-based languages such as ImperativeOCL that we described in Sect. 2.2, SOIL does not change the semantics of OCL itself. It is constructed as an additional layer on top of OCL.

Given the Project World class diagram from Fig. 3.1 on page 34 in the last chapter, a simple SOIL program that increases the incomes of all underpaid employees looks as follows:

```
1 minimum := 12100;
2 totalIncrease := 0;
3 for emp in Worker.allInstances->select(w| w.employer.isDefined and
      w.salary < minimum)->asSequence do
4   emp.salary := emp.salary + 200;
5   totalIncrease := totalIncrease + 200
6 end
```

Listing 4.1: A first SOIL program

On the top level, this program consists of three statements, two assignment statements (lines 1 and 2) and one iteration statement (lines 3–6). The iteration statement contains two sub-statements (lines 4 and 5). Consecutive statements have to be separated by ';'.

The first two statements assign constant values to the variables *minimum* and *totalIncrease*. It is not necessary (and not possible) to declare variables in order to assign values to them. Instead, variables declarations are implicitly inferred in SOIL. In this example, the inferred type of the variables *minimum* and *totalIncrease* is Integer. Subsequently to line 2, both variables can occur in OCL expressions as free Integer variables. The right-hand side of a variable assignment is always an OCL expression. Thus, instead of the constant values 12100 and 0, any closed OCL expressions (without free variables) could have been used in lines 1 and 2.

The third statement iterates over all Worker instances which are connected to a company and whose salary is less than the value of *minimum*. The body (lines 4 and 5) is executed for each element of this range expression. For each iteration the current element (i. e., a Worker value) is bound to the variable *emp*. This variable is assigned before each iteration of the body statement (between do and end). The range of an iteration statement is the second place in this example where OCL expressions occur. Any sequence typed expression is allowed, the range variable is bound within the body and typed with the element type of the sequence. Notice that the range expression contains a free variable *minimum* when viewed in isolation – this variable is bound by the surrounding SOIL program. The same situation follows within the iteration body, where new values are assigned to the *salary* attribute

of the current *emp*, and to the variable *totalIncrease*. The OCL expressions on both right-hand sides contain free variables, the range variable *emp* and *totalIncrease*.

The previous example used three language elements of SOIL (iteration, variable assignment and attribute assignment). It already shows how OCL is embedded into SOIL. All of the above statements contain OCL expressions, but, statements and expressions are disjoint entities. In a technical sense, the connection between statements and expressions is as follows: Statements evaluate expressions, which may contain free variables. The interpretation of these expressions depends on the values that are provided for the free variables, and, of course, the system state (i. e., the objects, links, and attribute values). Both, system state and variables can be modified by statements. Figure 4.1 illustrates this dependency in an evaluation chain.

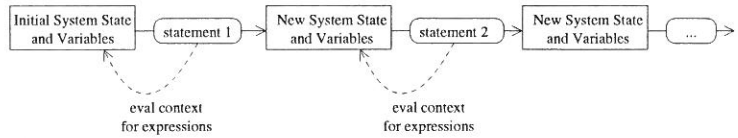

Figure 4.1: Evaluation Chain for Statements and Expressions

Apart from attribute assignments, SOIL comprises further basic statements to manipulate the system state. These are object creation, object destruction, link insertion, and link destruction. The following example illustrates these statements, as well as conditional execution.

```
1  prg := new Qualification;
2  c := new Company;
3  w1 := new Worker;
4  w2 := new Worker;
5  insert (w2,prg) into IsQualified;
6  insert (c,w1) into Employs;
7  insert (c,w2) into Employs;
8  p1 := new Project;
9  p2 := new Project;
10 insert (c,p1) into CarriesOut;
11 insert (p1, c.employees->any(e | e.qualifications->includes(prg)))
       into Member;
12 delete (c,w2) from Employs;
13 if (p2.members->isEmpty) then
14   destroy p2
15 end
```

Listing 4.2: Example for state manipulation commands in SOIL

Lines 1–4 and 8-9 contain object creation statements. These statements also assign a new (object) value to a variable, but 'new Qualification' is not an OCL expression. This form can only occur on the right-hand side of an object creation statement. Thus, `x := if happy then new Project else Undefined endif` is not a statement (neither object creation nor variable assignment), as `if happy then new Project else Undefined endif` is not an OCL expression.

Lines 5-7 and 10-12 contain link manipulation statements (insertion and deletion). For both statements, the tuple that is to be inserted into, respectively, deleted from the association extend, is given by OCL expressions. These OCL expressions must conform to the types required by the association ends. Finally, line 13–15 show conditional execution. The condition is given by a Boolean typed OCL expression. The body of this statement contains other statements. Line 14 shows the object destruction statement. The object to be destroyed is determined by an OCL expression, which has to be of an object type.

4.1.1 Execution Context

SOIL statements can be used to specify operations in UML models. They can also be executed stand-alone, e. g., to create or animate system states.

If a SOIL statement is used to specify an implementation for an operation, the formal parameters of the operation are available as if the variables were bound in the statements.

As a second example, let us now consider the (non-query) operation *fire(w : Worker)* in class Company from the 'Project World' class diagram in Fig. 3.1 on page 34. The operation has a single parameter w of type *Worker*.

We can use SOIL to define the semantics of *fire*, as shown in Listing 4.3.

```
1 def Company::fire(w : Worker)
2   delete (self, w) from Employs;
3   for p in w.projects do
4     delete (p,w) from Member
5   end
```

Listing 4.3: Operation definition for Company::fire

Within the body of *fire*, the variables *self* (the object on which *fire* is invoked) and w (the parameter) are available as bound variables. If the operation has a return type, the value of the variable *result* refers to the return value of the operation. Such an operation is illustrated in the next subsection.

4.1.2 Operation Invocations

All operations defined in a class can be applied to an object. We need to distinguish between (side-effect free) query operations and operations with side-effects.

Query operations can be applied (only) in expressions. This is already part of OCL and independent from SOIL. Given there is a bound variable IBM referring to a Company object, the statement in the following example stores the number of employees that are overloaded with work in the variable *numOverloaded*. This is achieved by using the operation isOverloaded(), which we defined by an OCL expression on page 53.

```
1 numOverloaded := IBM.employees->select(w|w.isOverloaded())->size
```

Non-query operations are invoked by one of two explicit statements, one for operations without result values, and one for operations with result values. We can use *fire()* to illustrate the first one.

```
1 IBM := new Company;
2 Bob := new Worker;
3 insert (IBM, Bob) into Employs;
4 IBM.fire(Bob)
```

The statement in line 4 is the operation invocation. After executing the last statements, the objects IBM and Bob will not longer be connected by the Employs association.

To illustrate the second kind of invocation of operations, those with result values, we assume a class *Calculator* with a single *Integer* typed attribute *myState* and an operation *fac(n : Integer)* → *Integer*, defined as follows:

```
1 def Calculator::fac(n : Integer) : Integer
2   self.myState := 1;
3   if (n >= 1) then
4     self.myState := self.fac(n - 1);
5     self.myState := n * self.myState
6   end;
7   result := self.myState
```

We can invoke this operation as follows:

```
1 c := new Calculator;
2 n := c.fac(4)
```

After executing the above statements, the variable n will contain the value 24. In the recursive evaluation the *myState* attribute of the Calculator object referenced by c will bet set to $1, 2, 6$, and finally 24. Notice that line 2 is an explicit invocation statement for operations with result value, and not

a variable assignment statement. Thus, analogously to the object creation command, `n := c.fac(3) * c.fac(3)` is not a statement, because `c.fac(3) * c.fac(3)` is not an OCL expression. The operation *fac()* is not a query operation. We defined it, by purpose, as an operation with side-effects (although it can be expressed in a functional manner very easily). The last example also shows that operation invocations can be nested. Of course, this introduces potentially non-terminating statements. However, for practical reasons, recursion is allowed in SOIL. We will discuss recursion in more depth later.

4.1.3 Type-Safety and Variable Scopes

A well-typed statement will never produce type errors at evaluation time (i. e., at runtime). Therefore, a statement such as the compound statement `x := 1; y := '2'; z := x + y` is not valid in SOIL, assuming there is no operation $+ : \text{Integer} \times \text{String} \rightarrow t$ defined for any type t. The meaning of this statement would be undefined. Therefore, it is rejected statically by the type-checker.

We also exclude statements that *may* have an undefined meaning, depending on actual variable values. Consider the following compound statement which is not a valid in SOIL:

```
1 x := 1;
2 b := <any boolean expression>
3 if b then
4   x := '1'
5 end;
6 x := x + 1
```

The above statement would have a well-defined interpretation if the Boolean expression evaluates to false. However, if the expression evaluates to true, the meaning of the statement would be undefined, for the same reason as described above (we assume there is no way to add strings and numbers). We formally discuss type-soundness for SOIL later in Sect. 4.5.

We already mentioned that variables are implicitly declared (we say 'bound') in SOIL. There are no explicit variable declarations. In general, a variable can be regarded as declared, as soon as a value is assigned to it. The language automatically infers the type for us. There are, however, restrictions w. r. t. iteration and conditional execution. As variables can be assigned several times, the declaration of a variable does not necessarily remain constant for all subsequent statements. Fig. 4.2 on the facing page illustrates the binding by examining a sequence of statements (i. e., a compound statement).

In statement (1), an *Integer* typed value is assigned to x. Thus, x is implicitly declared as type *Integer* until there is a further (non-*Integer*) assignment to x

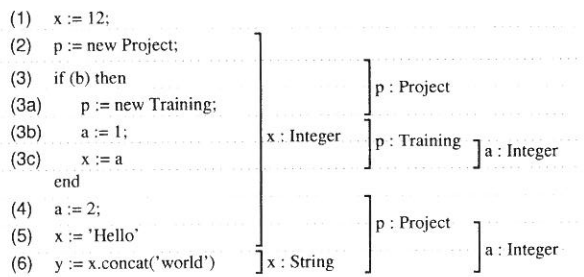

Figure 4.2: Implicit Variable Declarations in SOIL

(until after statement (5)). Thus, within statements (2) – (5), an *Integer* value can be assumed for x in OCL expressions.

Similarly, after statement (2), the variable p is implicitly declared as type *Project*. In statement (3a), within the conditional execution (3), a new value of type *Training* is assigned to p (we assume b to be a Boolean-typed expression). Thus, from now on, p can be assumed to be of type *Training*, which is a subtype of *Project* (c. f. Fig. 3.1 on page 34). For example, the term p.trained may occur in expressions after the assignment in (3a). However, variables bound within conditional executions and iteration statements are not propagated by these statements. Thus, after statement (3), we have to assume p to be of type *Project*, again. The reason for this typing scheme is, that the sub-statements within these compound statements are not guaranteed to be executed. For the same reasons, only type-conforming assignments are allowed within conditional executions and iterations: We could not say `p := 42` as a replacement for statement (3a). The syntax of SOIL does not allow such an assignment.

In addition to the restrictions on variable assignments that are imposed by the type system, an additional restriction regards the iterator variable of a *for* loop. Within the body statement of a loop, the iterator variable must not be changed (no matter which type). Therefore, the following statement is not valid:

```
1 for x in Sequence{1,2,3} do x := 1 end
```

On Explicit and Implicit Variable Declaration

As we explained in the previous section, no explicit variable declarations are required in SOIL. Types for variables are automatically inferred. There are, however, certain situations where an upfront variable *initialization* is required. These initializations are ordinary assignments. They are needed,

whenever a variable would have been assigned otherwise within a conditional execution or an iteration for the first time. As these constructs do not propagate their bound variables, we need to put an explicit initialization in front of them. Thus, we cannot write

```
1 for i in Sequence{1,2,3} do
2   x := i;
3 end;
4 y := x;
```

The type-checker of SOIL rejects the previous statement as, in general, the sequence expression which determines the range of a *for* loop might evaluate to the empty sequence. We have to add an initialization for x to the beginning of this example.

```
1 x := 0;
2 for i in Sequence{1,2,3} do
3   x := i;
4 end;
5 y := x;
```

We find a similar pattern for variables being initialized (assigned for the first time) within blocks in several scripting languages with implicit variable declaration, for example in Ruby [FM08].

4.1.4 Error Handling

For the sake of simplicity, there is no sophisticated exception or error handling mechanism in SOIL. This does, however, not mean that SOIL programs may just abort in an undefined way. In fact, there is no statement in SOIL that could lead to program abortion. Therefore, the following situations are treated specifically:

Inserting tuples into an association extent more than once: If a tuple is already present in an association extent, a second insert operation has no further effect, as in the following example.

```
1 x := new C;
2 y := new D;
3 insert (x,y) into A;
4 insert (x,y) into A
```

Deleting non-existing tuples from an association extent: If a tuple is not present in an association extent, a delete operation has no effect.


```
1 x := new C;
2 y := new D;
3 delete (x,y) from A
```

Inserting tuples with undefined components: If a tuple contains an undefined value, the insert operation has no effect.

```
1 x := new C;
2 insert (x,Undefined) into A
```

Assigning attribute values for an undefined object: The assignment of an attribute value to Undefined has no effect.

```
1 x := new C;
2 destroy x;
3 x.a := 42
```

Operation invocation on Undefined: An operation invocation on the undefined value has no effect. In the following example, line 3 does nothing.

```
1 x := new C;
2 x := Undefined;
3 x.f()
```

Destroying an undefined object: The destruction of Undefined has no effect.

```
1 destroy Undefined
```

Remark. Although a different treatment for multiple inserts of the same tuple and for the removal of non-existing links seems reasonable, the above treatment for the first two items is in accordance with the common understanding of association extents as relations.

However, the latter items clearly lead to unexpected behavior. An error could be expected. Nevertheless, for reasons of simplicity, we have decided not to include a more sophisticated error handling mechanism in this version of SOIL. After our presentation of the formal semantics for SOIL, we sketch how more sophisticated error handling could be added to SOIL in Sect. 4.6.

4.1.5 Garbage Collection

There is no implicit garbage collection for objects. Objects are destroyed explicitly only by the *destroy* command. Therefore, objects can exist without any reference from variables, links or attribute values to them.

Example. This following program creates an instance of class C, storing the instance reference in the variable x, and then assigns the undefined value to x in the next step.

```
1  x := new C;
2  x := Undefined
```

After executing these two statements, no reference to the new instance of C exists. However, the object itself still exists in the system state. It can be referenced by the *allInstances* property of class C (assuming there is no other instance of C):

```
1  x := C.allInstances->any(true)
```

□

However, objects generally need to be distinguishable by some proposition φ about their properties (i.e., their attribute values and the links they participate in) in order to be uniquely identified by C.allInstances $\rightarrow$ any(φ). This is not possible in general, as there is no way to guarantee the identity uniqueness of an object by an OCL expression.

While there is no garbage collection for objects, there is implicit garbage collection for links. If an object is destroyed, all links, in which this object is participating in, are deleted as well.

Example. The following program creates a company and a project, establishes a link between them, and then destroys one of the objects.

```
1  c : = new Company;
2  p : = new Project;
3  insert (c,p) into CarriesOut;
4  destroy c
```

After executing these statements, the tuple (c, p) is not longer present in the extent of the CarriesOut association, and the expressions `c` and `p.company` will evaluate to Undefined. □

Notice that all variables that reference an object are set to Undefined when the object is destroyed. Thus, after

```
1  x := new Project;
2  y := x;
3  destroy y
```

Both variables x and y contain the undefined value.

4.1.6 Modularity of SOIL and OCL

We already described that only query operations can be used within OCL expressions, even if these expressions occur embedded in statements. For the same reason we did not allow, for example, *new Qualification* to occur in expressions. We motivated this restriction in Sect. 2.2.

Consequently, SOIL does not support a *compute* expression, as ImperativeOCL does. We shortly recapitulate *compute*. The following 'expression' is valid in ImperativeOCL (but not in OCL).

```
1  y := 1 + compute(x)(<statements calculating x>) + 2
```

Since SOIL embeds OCL unchanged, such constructions are not possible in SOIL and have to be rewritten using pure OCL.

```
1  <statements calculating x>;
2  y := 1 + x + 2
```

Of course, more complex, nested compute expressions require more than two statements. The following imperative expression

```
1  s : = Set{1,2,3}
2  z := 1 + s.collect(y|compute(x)(<calculate x based on y>)->sum + 2)
```

could be rewritten as

```
1  s : = Set{1,2,3};
2  sx := oclEmpty(Bag(Integer));
3  for y in s do
4    <calculate x based on y>;
5    sx := sx->including(x)
6  end;
7  z := 1 + sx->sum + 2
```

Remark: We use the predefined constructor oclEmpty(Bag(Integer)) in the above example to get an empty Bag of type *Bag(Integer)*.

4.1.7 Summary

Table 4.1 on the following page summarizes the informal introduction of SOIL.

Operation invocation

- Query operations can be only used in OCL expressions.
- Operations with side-effects be only invoked by a special kind of invocation statements. They cannot be used as part of expressions such as $z := f(x) + g(y)$ or $z := f(g(x), h(y))$ (given f, g, h being operations with side-effects). Such nested constructions have to be split up into several statements using intermediate variables.

Variable declaration and type-safety

- There is no explicit variable declaration. The general rule is: If a statement assigns a value to a variable, this variable is allowed to occur as a free variable in subsequent statements (assuming consistent types). The type of the variable is inferred.
- A variable may be assigned more than once in a compound statement. This overrides previously inferred variable types.
- Special rules hold for *if* and *for* statements:
  - If a variable is assigned within the body of such a statement, the inferred type is not propagated.
  - In the body, only type-conforming assignments must be made to variables which are bound by a previous statement.
  - No assignments are allowed to an iterator variable.

Garbage collection

- Objects are destroyed explicitly. An object's lifetime is not related to variables referencing it.
- If an object is destroyed, all references to it (variables, attributes, links) are removed implicitly.

Error handling

- There is no sophisticated error handling (this is future work, as explained later in Sect. 4.6).
- The occurrence of the undefined value where it is not expected results in a statement that has no effect.

Table 4.1: Summary of important aspects of SOIL