Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

Sergey Brazhnik, Alexandre Zamulin

AN IMPERATIVE EXTENSION OF THE OBJECT CONSTRAINT LANGUAGE OCL

Preprint
123

Novosibirsk 2005

An extension of the object constraint language OCL by a mechanism permitting specification of methods updating the object state is described and investigated in the paper. An original version of transition rules of Abstract State Machines is proposed as such a mechanism. The new language facilities are investigated by specification of the class diagram of a representative example.

Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

С. А. Бражник, А. В. Замулин

ИМПЕРАТИВНОЕ РАСШИРЕНИЕ ЯЗЫКА СПЕЦИФИКАЦИИ ОБЪЕКТОВ OCL

Препринт
123

Новосибирск 2005

В данной работе описывается и исследуется расширение языка спецификации объектов OCL средствами спецификации методов, изменяющих состояние объекта. В качестве таких средств предложена собственная версия правил перехода машин абстрактных состояний. Новые языковые средства исследованы путем спецификации диаграммы классов репрезентативного примера.

1. INTRODUCTION*

In this paper we describe in short the IOCL, an imperative extension of the Object Constraint Language (OCL) standard 1.5 [1], and investigate its applicability for specification of state updating methods (full description of IOCL can be found in [2]). OCL is a formal language used to express constraints. These typically specify invariant conditions that must hold for the system being modeled. In addition to the standard OCL facilities, IOCL permits specification of state updating methods (called *mutators* in the sequel) by means of the so called *transition rules,* which resemble statements of an imperative programming language. The idea is to provide an executable specification of mutators which can be compiled into executable code. The set of transition rules is based on the set of rules of Abstract State Machines [3] modified to fit our purposes.

No standard OCL facility is changed in IOCL.

2. STATE DIAGRAM OVERVIEW

In this section, using the class diagram of Fig. 1 (the diagram will be also used in the subsequent examples), we remind the main notions of UML class diagrams.

Class (`Bank`, `Person`, …) – a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics.

Classifier (**`type`**, **`class`**, …) – a mechanism that describes behavioral and structural features of an entity.

Attribute (`birthdate`, `age`, …) – a feature within a classifier that describes a range of values that instances of the classifier may hold.

Operation (`income`, `stockPrice`, …) – a service that can be requested from an object to effect behavior. An operation has a signature, which may restrict the actual parameters that are possible.

Method – the implementation of an operation. It specifies the algorithm or procedure associated with an operation.

Relationship (`customer`, `husband`, …) – a semantic connection among model elements. Examples of relationships include associations and generalizations.

Association (`customer`, `husband`, …) – the semantic relationship between two or more classifiers that specifies connections among their instances.

---

* This research is supported in part by Russian Foundation for Basic Research under Grant 04-01-00272.

Association end – the endpoint of an association, which connects the association to a classifier (i.e., the point where the line representing an association touches the box representing a classifier).

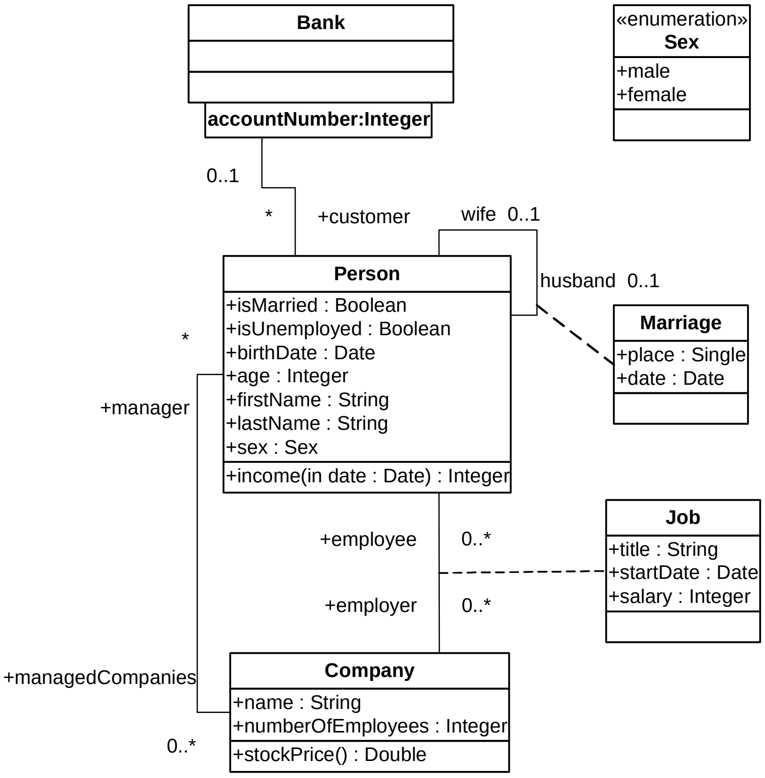


Fig. 1. Class Diagram Example

3. OCL OVERVIEW

3.1. Purpose of OCL

A UML diagram, such as a class diagram, is typically not refined enough to provide all relevant aspects of a specification. There is, among other things, a need for describing additional constraints concerning the objects in the model. Such constraints are often described in natural language. Practice has shown that this always results in ambiguities. In order to write unambiguous constraints, the so-called formal languages have been developed. OCL is a formal specification language that is yet easy to read and write. It is a *pure expression language*; therefore, an OCL expression is guaranteed to be side-effect free. This means that the state of the system will never change because of the evaluation of an OCL expression, even though an OCL expression can be used to *specify* a state change (for example, in a postcondition). OCL is a typed language, so each OCL expression has a type.

3.2. Where to Use OCL

OCL can be used for a number of different purposes:

- to specify invariants on classes and types in the class model,
- to describe pre- and postconditions on Operations and Methods,
- to describe Guards,
- as a navigation language,
- to specify constraints on Operations.

3.3. Main components of OCL

3.3.1. Expressions

Each expression in OCL is constructed in a certain context denoted by keyword `context`. The subsequent keyword `inv`, `pre,` or `post` indicates whether the constraint concerns the «invariant», «precondition», or «postcondition». The actual OCL expression comes after the colon.

Each OCL expression is written in the context of an instance of a specific type. The reserved word `self` is used to refer to the contextual instance. For example, if the context is `Company`, then `self` refers to an instance of `Company`.

3.3.2. Invariants

The OCL expression can be part of an *invariant* of a type, and in this case it must be true for all instances of the type at any time (all OCL expressions that express invariants are of the type Boolean). For example, for the `Company` type, the following expression would specify an invariant that the number of employees must always exceed 50:

```
        context Company inv: self.numberOfEmployees > 50
```

where `self` is an instance of type `Company` (one can view `self` as the object "owning" the expression). This invariant holds for every instance of the `Company` type.

In most cases, the keyword `self` can be dropped because the context is clear, (as `this` in C++ or Java). As an alternative for `self`, a different name can be defined playing the part of self:

```
        context c : Company inv: c.numberOfEmployees > 50
```

This invariant is equivalent to the previous one.

3.3.3. Pre- and Postconditions

The OCL expression can be part of a precondition or postcondition associated with an operation or method. The general form:

```
     context TypeName::OpName(par1: Type1, ... ): ReturnType
          pre : par1 > ...
          post: result = ...
```

The name `self` can be used in the expression referring to the object on which the operation was called. The precondition (like the postcondition) can also include operation parameters (*par1, …*), thus putting extra requirements on them.

. In the example diagram, we can write:

```
        context Person::income(d : Date) : Integer
          pre : d >= currentDate
          post: result = 5000
```

The reserved word `result` in a postcondition denotes the result of the operation, if there is one. It is obligatory in the specification of an operation for which the predicate *isQuery* produces `True`, and icannot be used for an operation for which the predicate *isQuery* produces `False`.

The predicate *isQuery* specifies whether the execution of the operation leaves the state of the system unchanged. `True` indicates that the state is unchanged; `False` indicates that a side-effect may occur.

3.4. Basic Values and Types

3.4.1. Atomic types

In OCL, a number of basic types are predefined and available to the modeler at all times. The most basic value in OCL is a value of one of the basic types. Typical basic types used in the examples in this document are `Boolean`, `Integer`, `Real`, and `String`.

Enumerations are data types in UML and have a name, just like any other Classifier. An enumeration defines a number of enumeration literals, which are the possible values of the enumeration. When we have a data type named `Sex` with values `female` or `male`, they can be used as follows:

```
        context Person inv: sex = Sex::male
```

3.4.2. Collections

The type `Collection` is predefined in OCL. Consistent with the definition of OCL as an expression language, collection operations never change collections. They may result in a collection, but rather than changing the original collection they project the result into a new one.

`Collection` is an abstract type, with concrete collection types as its subtypes. OCL distinguishes three different collection types: `Set`, `Sequence`, and `Bag`. A set is the mathematical set. A bag is like a set, which may contain duplicates; that is, the same element may be in a bag twice or more. A sequence is like a bag in which the elements are ordered. Both bags and sets have no order defined on them.

Sets, sequences, and bags can be specified by a literal in OCL. Curly brackets embrace the elements of the collection, elements in the collection are written within, separated by commas. The type of the collection is written before the curly brackets:

```
        Set { 1 , 2 , 5 , 88 }
        Set { 'apple' , 'orange', 'strawberry' }
        Sequence { 1, 3, 45, 2, 3 }
        Sequence { 'ape', 'nut' }
        Bag {1 , 3 , 4, 3, 5 }
```

Because of the usefulness of a sequence of consecutive integers, there is a separate literal to create them. The elements inside the curly brackets can be replaced by an interval specification, which consists of two expressions of type Integer separated by '..'. This denotes all the integers between the values of the expressions (including their values as well):

```
Sequence{ 1..(6 + 4) }
Sequence{ 1..10 }
-- are both identical to
Sequence{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
```

Within OCL, all collections of collections are flattened automatically; therefore, the following two expressions have the same value:

```
Set{ Set{1, 2}, Set{3, 4}, Set{5, 6} }
Set{ 1, 2, 3, 4, 5, 6 }
```

3.4.3. Collection Operations

OCL defines many operations on the collection types. A collection operation is accessed by using an arrow '->' followed by the name of the operation and arguments, if any. Examples:

```
context Person inv: self.employer->size() < 3
```

This applies the `size` property on the set `self.employer`, which results in the number of employers of the person `self`.

```
context Person inv: self.employer->isEmpty()
```

This applies the `isEmpty` property on the set `self.employer`. This also evaluates to `true` if the set of employers is empty and `false` otherwise.

Several collection operations are specifically meant to enable a flexible and powerful way of projecting new collections from existing ones.

The `select` operation specifies a subset of a collection. Its parameter has a special syntax that enables one to specify which elements of the collection we want to select. There are three different forms, of which the simplest one is:

```
collection->select(boolean-expression)
```

This results in a collection that contains all the elements from *`collection`* for which the *`boolean-expression`* evaluates to `true`. As an example, the following OCL expression specifies that the collection of all the employees older than 50 years is not empty:

```
context Company inv:
        self.employee->select(age > 50)->notEmpty()
```

In the above example, it is impossible to refer explicitly to the persons themselves; one can only refer to properties of them. To enable to refer to the persons themselves, there is a more general syntax for the select expression:

```
collection->select( v | boolean-expression-with-v )
```

The variable *`v`* is called the *iterator*. When the `select` is evaluated, *`v`* iterates over *`collection`* and *`boolean-expression-with-v`* is evaluated for each *`v`*. The following example is identical to the previous one:

```
context Company inv:
   self.employee->select(p | p.age > 50)->notEmpty()
```

As a final extension to the select syntax, the expected type of the variable *v* can be given. The select now is written as:

```
collection->select(v : Type | boolean-expression-with-v)
```

The meaning of this is that the objects in `collection` must be of type `Type`. The next example is identical to the previous ones:

```
context Company inv:
  self.employee.select(p : Person | p.age > 50)->notEmpty()
```

The `reject` operation is identical to the `select` operation, but with `reject` we get the subset of all the elements of the collection for which the expression evaluates to False. The `reject` syntax is identical to the `select` syntax:

```
collection->reject(v : Type | boolean-expression-with-v)
collection->reject(v | boolean-expression-with-v)
collection->reject(boolean-expression)
```

As an example, we specify that the collection of all the employees who are not married is empty:

```
context Company inv:
        self.employee->reject(isMarried)->isEmpty()
```

The `collect` operation serves for specifying a collection that is derived from some other collection. The operation uses the same syntax as `select` and `reject`:

```
collection->collect( v : Type | expression-with-v )
collection->collect( v | expression-with-v )
collection->collect( expression )
```

The value of the collect operation is the collection of the results of all the evaluations of `expression-with-v`. An example: specify the collection of `birthDates` for all employees in the context of a company. In the most general form, this can be written as follows:

```
self.employee->collect(person : Person | person.birthDate)
```

An important issue here is that the resulting collection is not a set, but a bag (more than one employee may have the same value for `birthDate`). It is possible to make a set from the bag, by using the `asSet`() operation. Example:

```
self.employee->collect(birthDate)->asSet()
```

Because navigation through many objects is very common, there is a shorthand notation for the `collect` that makes the OCL expressions more readable. Instead of

```
self.employee->collect(birthdate)
```

we can also write:

```
self.employee.birthdate
```

In general, when we apply a property to a collection of Objects, then it will automatically be interpreted as a `collect` over the members of the collection with the specified property.

Many times a constraint is needed on all elements of a collection. The `forAll` operation in OCL allows specifying a Boolean expression, which must hold for all objects in a collection. It has the same syntax as the previous operations. For example, in the context of a company:

```
context Company
        inv: self.employee->forAll(forename = 'Jack')
```

This invariant evaluates to true if the forename feature of each employee is equal to `'Jack'`.

When one needs to know whether there is at least one element in a collection for which a constraint holds, the `exists` operation is used. It allows one to specify a Boolean expression that must hold for at least one object in a collection. For example, in the context of a company:

```
context Company inv:
        self.employee->exists(forename = 'Jack')
```

the invariant holds if the forename feature of at least one employee is equal to `'Jack'`.

The `any` operation returns any element in the `collection` for which `expr` evaluates to true. If there is more than one element for which `expr` is true, one of them is returned. The precondition states that there must be at least one element fulfilling `expr`; otherwise, the result of this operation is Undefined.

```
collection->any(expr : OclExpression) : T
```

3.4.4. Type Conformance

OCL is a typed language and the basic value types are organized in a type hierarchy. This hierarchy determines conformance of different types to each other. A type *`type1`* conforms to a type *`type2`* when an instance of *`type1`* can be substituted at each place where an instance of *`type2`* is expected. Type conformance rules for types in the class diagrams are simple:

- each type conforms to itself and each of its supertypes;
- type conformance is transitive: if *`type1`* conforms to *`type2`*, and *`type2`* conforms to *`type3`*, then *`type1`* conforms to *`type3`*.

The effect of this is that a type conforms to its supertype, and all the supertypes above.

The type conformance rules for the value types are as follows:

`Set(T)`	conforms to (is a subtype of)	`Collection(T)`
`Sequence(T)`	conforms to (is a subtype of)	`Collection(T)`
`Bag(T)`	conforms to (is a subtype of)	`Collection(T)`
`Integer`	conforms to (is a subtype of)	`Real`

In addition to the type conformance rules indicated above, the following rule holds for all collection types:

`Collection(`*`type1`*`)` conforms to `Collection(`*`type2`*`)`, when *`type1`* conforms to *`type1`*.

For example, if `Bicycle` and `Car` are two separate subtypes of `Transport`:

`Set(Bicycle)` conforms to `Set(Transport)`
`Set(Bicycle)` conforms to `Collection(Bicycle)`
`Set(Bicycle)` conforms to `Collection(Transport)`

An OCL expression which satisfies type conformance is a *valid* expression.

3.4.5. Re-typing or Casting

In some circumstances, it is desirable to use a property of an object that is defined on a subtype of the current known type of the object. Because the property is not defined on the current known type, this results in a type conformance error.

When it is certain that the actual type of the object is the subtype (the effect of the operation is undefined, otherwise), the object can be retyped using the operation `oclAsType(`*`OclType`*`)`. This operation results in the same object, but the known type is the argument type. When there is an object *`object`* of type *`type1`* and *`type2`* is its subtype, it is allowed to write:

`object``.oclAsType(`*`type2`*`)` — *evaluates to an object of type* *`type2`*

The operation `oclAsType` can also be used to regard a subtype object as a supertype object.

In both cases, if *`type1`* and *`type2`* are not in the supertype-subtype relationship, the expression is undefined.

3.4.6. Undefined Values

Whenever an OCL expression is being evaluated, there is a possibility that one or more of its subexpressions are undefined. If this is the case, then the complete expression will be undefined. There is an exception for the expression of type `Boolean`: it is always defined, i.e., it always produces `True` or `False`, independent of definedness of its subexpressions In particular:

- `True OR`-ed with anything is `True`
- `False AND`-ed with anything is `False`
- `e1 = e2` is `True` iff both `e1` and `e2` are defined and equal.

The above three rules are valid irrespective of the order of the arguments and whether or not the value of the other sub-expression is known.

In IOCL, there is a special predicate `defined` used to check whether an expression is defined in the current state. Thus, if *`t`* is an expression of any type, then `defined` *`t`* produces `True` if *`t`* is defined and `False` in the opposite case.

3.5. Let Expressions

Sometimes a sub-expression is used more than once in a constraint. The `let` expression allows one to define an attribute or operation that can be used in the constraint.

```
context Person inv:
  let income : Integer = self.job.salary->sum()
  let hasTitle(t : String) : Boolean =
      self.job->exists(title = t) in
  if isUnemployed then
      self.income < 100
  else
      self.income >= 100 and self.hasTitle('manager')
  endif
```

A `let` expression may be included in an invariant or pre- or postcondition. It is then only known within this specific constraint.

3.6. Properties

All attributes, association-ends, methods, and operations without side-effects that are defined in data types can be used in expressions. In a class model, an operation or method is defined to be side-effect-free if the `isQuery` attribute of the operation is `true`. Attributes, association-ends, and side-effect-free methods and operations are called *properties*.

3.6.1. Attributes

The value of a property on an object that is defined in a class diagram is specified by a dot followed by the name of the property.

```
context AType inv: self.property
```

If `self` is a reference to an object, then `self.`*property* is the value of the *`property`* property on `self`. For example, the age of a `Person` is written as `self.age`:

```
context Person inv: self.age > 0
```

The value of the subexpression `self.age` is the value of the *age* attribute on the particular instance of `Person` identified by `self`.

3.6.2. Operations

Operations may have parameters. For example, as shown earlier, a `Person` object has an income expressed as a function of the date. This operation would be accessed as follows, for a person *`aPerson`* and a date *`aDate`*:

```
aPerson.income(aDate)
```

The operation itself could be defined by a postcondition constraint. The object that is returned by the operation can be referred to by `result`. It takes the following form:

```
context Person::income (d: Date) : Integer
        post: result = age * 1000
```

The right-hand-side of this definition may refer to the operation being defined; that is, the definition may be recursive as long as the recursion is not infinite. To refer to an operation or a method that doesn't take a parameter, parentheses with an empty argument list are mandatory:

```
context Company inv: self.stockPrice() > 0
```

3.6.3. Association Ends and Navigation

Starting from a specific object, one can navigate an association on the class diagram to refer to other objects and their properties. To do so, one navigates the association by using the opposite association-end:

```
object.rolename
```

The value of this expression is the set of objects on the other side of the *`rolename`* association. If the multiplicity of the association-end has a maximum of one ("0..1" or "1"), then the value of this expression is an object. In the example class diagram, if one starts in the context of a `Company`, that is, `self` is an instance of `Company`, one can write:

```
context Company
        inv: self.manager.isUnemployed = false
        inv: self.employee->notEmpty()
```

In the first invariant, `self.manager` is a `Person`, because the multiplicity of the association is one. In the second invariant, `self.employee` will evaluate in a `Set(Person)`. When the association on the class diagram is adorned with `{ordered}`, the navigation results in a `Sequence`.

When a rolename is missing at one of the ends of an association, the name of the type at the association end, starting with a lowercase character, is used as the rolename. If this results in an ambiguity, the rolename is mandatory. This is the case with unnamed rolenames in reflexive associations. If the rolename is ambiguous, then it cannot be used in OCL.

A single object can be also used as a singleton set. The usage as a set is done through the arrow followed by a property of `Set`. This is shown in the following example:

```
context Company inv: self.manager->size() = 1
```

The sub-expression `self.manager` is used as a `Set`, because the arrow is used to access the `size` property on `Set`.

In the case of an optional (0..1 multiplicity) association, it is especially useful to check whether there is an object or not when navigating the association. In the example we can write:

```
context Person inv: self.wife->notEmpty() implies
    self.wife.sex = Sex::female
```

4. IMPERATIVE EXTENSION OF OCL

In addition to the purposes of OCL, IOCL also serves to describe methods updating the state. Being an imperative extension of OCL, IOCL is designed to fill the gap between OCL, which is an unexecutable specification language, and a programming language. Thus, an IOCL specification is absolutely formal just like an OCL specification, and is executable like a program. One can consider IOCL as an abstract programming language designed to support UML class diagrams. An IOCL compiler converts such a diagram into executable code.

Imperative extensions of OCL deal with *updateable properties*, which are attributes and association ends. An update of a property results in an update of the state. Operations and methods updating the state (i.e., producing a side effect) are called *mutators*. Thus, a mutator is one of:

- an Operation with `isQuery` being false,
- a Method with `isQuery` being false.

A mutator sharing its name with the name of its class is called a *constructor*.

4.1. Transition rules

A *transition rule* is used for the specification of a mutator. The rule is indicated in a specification by label **`rule`** before the actual rule:

```
context Typename::OpName(param1: Type1, ... ): ReturnType
        pre: param1 > ...
        rule: transtionRule
```

The name `self` can be used in the rule referring to the object calling the operation being specified. Using the dot notation, it is followed by the name of the property being updated.

There are typed and untyped transition rules. If the interpretation of a typed transition rule should produce a result of some type `T`, the transition rule is called a *transition term of type* `T`.

4.2. Basic transition rules

Basic transition rules are *mutator call, object creation instruction, update instruction, collection update,* and *skip instruction*.

Mutator call. If `p(T1, ..., Tn)` is a mutator declaration in a class `T`, `obj` an expression of type `T`, and `t1, ..., tn` expressions of respective types `T1, ..., Tn`, then `obj.p(t1, ..., tn)` is a *transition rule* called a *mutator call*.

If `p(T1 , ..., Tn):T'` is a mutator declaration in a class `T`, `obj` an expression of type `T`, and `t1, ..., tn` expressions of respective types `T1, ..., Tn`, then `obj.p(t1, ..., tn)` is a *transition term of type* `T'` called a *mutator call*.

Both kinds of rule are interpreted by the execution of the corresponding method with `obj` passed to it as an extra argument.

Object creation instruction. If `C` is a class name, then `oclNew C` is a transition term of type `C` whose interpretation creates a new object with undefined attributes and produces a reference to it. If `C(T1, ..., Tn)` is a constructor declaration in class `C` and `t1, ..., tn` expressions of respective types `T1, ..., Tn`, then `oclNew C(t1, ..., tn)` is a transition term of type `C` whose interpretation creates a new object initialized by the constructor.

Update instructions. Let `at` and `at1` be updateable properties of type `T` and `t` either an expression of type `T` or a transition term of type `T`. Then

`at := t` and `at1 := undef`

are transition rules called *update instructions*.

Interpretation 1. If `A` is a state and `t` an expression of type `T`, then

`at = t` in the new state if `t` is defined in `A`,
`at` is undefined if `t` in not defined in `A`,
`at1` is undefined in the new state.

Examples. In the context of `Person`, the execution of the transition rule

```
age := age + 1
```

will transform a state `A` into a state `B` so that the value of age will be increased by one. A transition rule

```
age := undef
```

will make `age` undefined in the new state.

The above examples are respectively equivalent to the following ones:

```
self.age := self.age + 1
self.age := undef
```

Interpretation 2.

If `A` is a state and *`t`* a transition term of type *`T`*, then

1) *`t`* is first evaluated in state `A`, producing a state `B`, and then this state is updated by setting *`at`* to `b` where `b` is the result produced by the evaluation of *`t`*.

2) *`at`* is undefined if *`t`* is not defined in `A`.

Collection update. If *c* is an updateable collection and *e* an expression, then

```
c -> include(e),
c -> delete(e),
c -> replace(e)
```

are transition rules called *collection updates*.

Interpretation. The rules are respectively equivalent to the following update instructions:

```
c := c -> including(e),
c := c -> reject(e),
c := c -> collect(e)
```

with the exception that *c* is evaluated once. See examples on pp. 24–26.

Skip instruction. The `skip` instruction does not change the state update.

4.3. Transition rule constructors

Complex transition rules are constructed recursively from basic transition rules by means of several rule constructors, e.g., *sequence constructor, set constructor, condition constructor, guarded update, loop constructor* and *block.*

Sequence constructor. If *`R1, R2, ..., Rn`* are transition rules and *`t`* an expression of type *`T`*, then

```
seq R1, R2, ..., Rn end and
seq R1, R2, ..., Rn result = t end
```

are, respectively, a transition rule and a transition term of type *T* called a *sequence of transition rules*.

Interpretation. The component rules are executed sequentially so that the rule *`Ri+1`* is executed in the state produced by the rule *`Ri`*. If there is a resulting expression *`t`*, it is evaluated in the final state. See examples on pp. 25–27.

The set constructor. If *`R1,..., Rn`* are transition rules and *`t`* an expression of type *`T`*, then

```
set R1, ..., Rn end and
set R1, ..., Rn result = t end
```

are, respectively, a transition rule and a transition term of type *T* called a *set of transition rules*.

Interpretation. The component rules are executed in parallel in the same state and the results are united if they are consistent (i.e., if there is no update of the same property by different values). If there is a resulting expression `t`, it is evaluated in the initial state.

Example. The mutator `hireEmployee` in the class `Company` can be specified as follows:

```
context Company::hireEmployee(p: Person)
rule: set employee - >include(p),
          stockprice := stockprice + 10
      end
```

The guarded update. The guarded update is a transition rule of the form **`if`** *`g`* **`then`** *`R`*, where *`g`* is a Boolean expression and *`R`* is a transition rule.

Interpretation. Rule *`R`* is executed if *`g`* is evaluated to `true` and nothing is done in the opposite case.

The conditional constructor. If *`k`* is a natural number, *`g0, ..., gk`* Boolean expressions, and *`R0, ..., Rk`* transition rules, then the following expression is a transition rule called a *conditional transition rule*:

```
if g0 then R0
elseif g1 then R1
...
elseif gk then Rk
endif
```

If *`gk`* evaluates to `true`, then the last `elseif` clause can be replaced with **`else`** *`Rk`*.

Interpretation. The expressions *`g0, …, gk`* are evaluated one after another and *`Ri`* for the first *`gi`* evaluated to `true` is executed. If all *`g0, …, gk`* are evaluated to `false`, no effect is produced.

The loop constructors. The guarded update together with the sequence constructor enables one to define some loop constructors. If *`R`* is a transition rule, *`g`* a Boolean expression, *`i`* an identifier, *`e1`* and *`e2`* expressions of type `Integer` not containing *`i`*, and *`C`* a collection, then

```
while g do R,
repeat R until g and
for i in C do R
```

are transition rules called *loops*.

Interpretation. In the first rule, *`R`* is executed repeatedly while *`g`* is `true`; if it is `false` at the beginning, *`R`* is not executed at all. In the second rule, *`R`* is executed repeatedly until *`g`* is `true`; if it is `false` at the beginning, *`R`* is executed once. In the third case, the loop parameter *`i`* runs over all elements of the collection *`C`* and the rule *`R`* is executed for each element assigned to *`i`*.

The block. If *x* is an identifier, *t* either an expression or a transition term of type *T*, and *R* a transition rule or transition term of type *T1*, then

```
        let x = t in R
```

is, depending on *R*, either a transition rule or transition term of type *T1*, called a *block*.

Interpretation. Evaluate *t*, then extend the state with a variable *x* of type *T* initialized with the value of *t*, execute *R* and delete *x* from the state. See examples on pp. 24–25.

4.4. Error handling

The invariant, precondition, or postcondition in an operation specification can have the optional clause `else` *`throwExpression`* where *`throwExpression`* is `throw` *`expression`*. In this case, when the operation is called within a transition rule and one of these fails, then *`throwExpression`* is executed, which transfers control to the catch clause of the closest enclosing *try-block* whose parameter type matches the type of the *`throwExpression`*. If the else clause is omitted or there is no matching catch clause, the execution of the operation is stopped in a predetermined way.

The try-block is defined as follows: if *`R`* is a rule, *`x1, ..., xn`* identifiers, *`T1, ..., Tn`* type names, and *`R1, ..., Rn`* rules, then

```
        try R  catch parameterDeclaration1 R1; ...
               catch parameterDeclarationN Rn
        end,
```

where *`parameterDeclarationI`*, *`I = 1, ..., n`*, is one of *`xi`* : `Ti` or `Ti` or "`...`", is a rule called *try-block*.

Interpretation. If during the execution of *`R`* the *`throwExpression`* of type *`Tj`* is executed, the control passes to the first `catch` clause with the parameter type *`Tj`* or to the `catch` clause with the parameter declaration "`...`" if any, and the corresponding *`Rj`* is executed. If the parameter declaration has the form *`x`*`:` *`T`*, then the value of the *`throwExpression`* is assigned to *`x`*. If no matching `catch` clause is found, the control passes to the enclosing try-block. If no matching `catch` clause is found, the execution of the block is also stopped in some predetermined way. See an example on p. 27.

5. EXPERIMENT

5.1. Class diagram

The class diagram used in the experiment is presented in Fig. 2. Because of the layout limitations, only class names are indicated for each class in the diagram. The declarations of class components are presented in the subsequent figures.

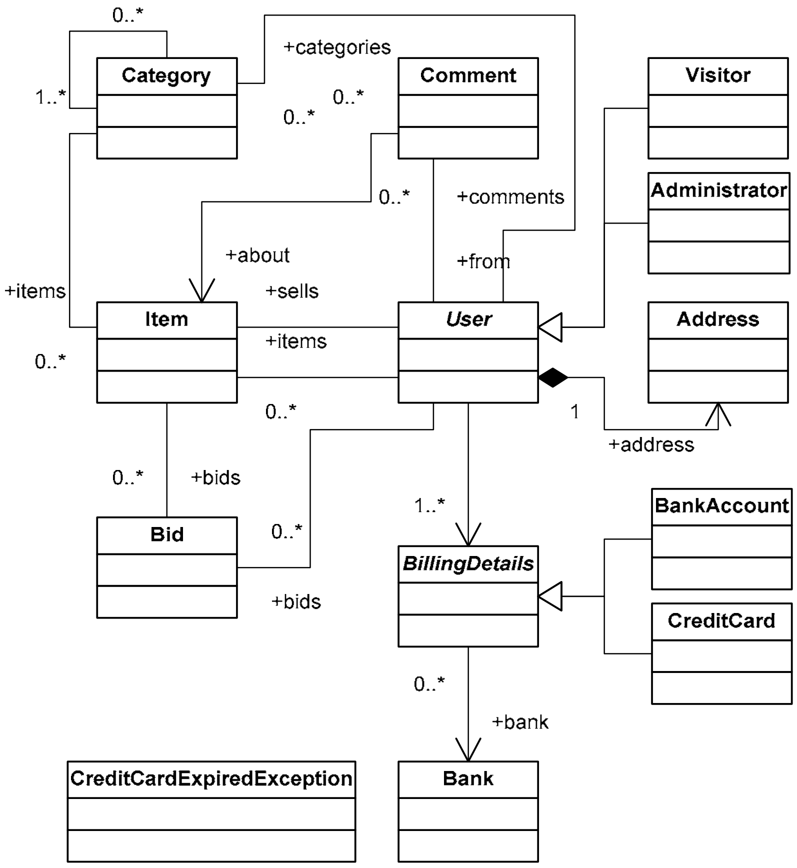


Fig. 2. The class diagram of the experiment

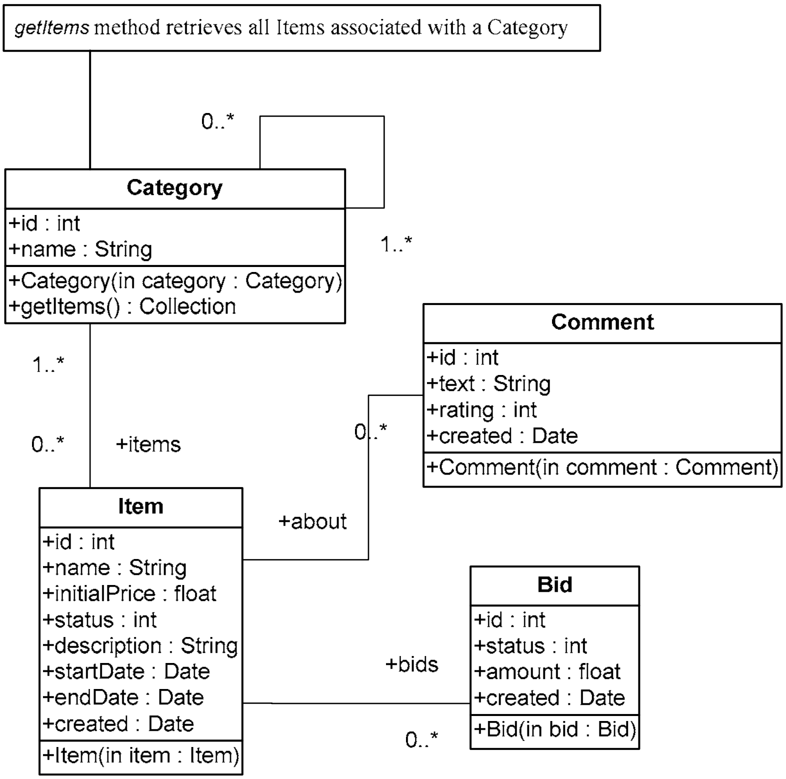
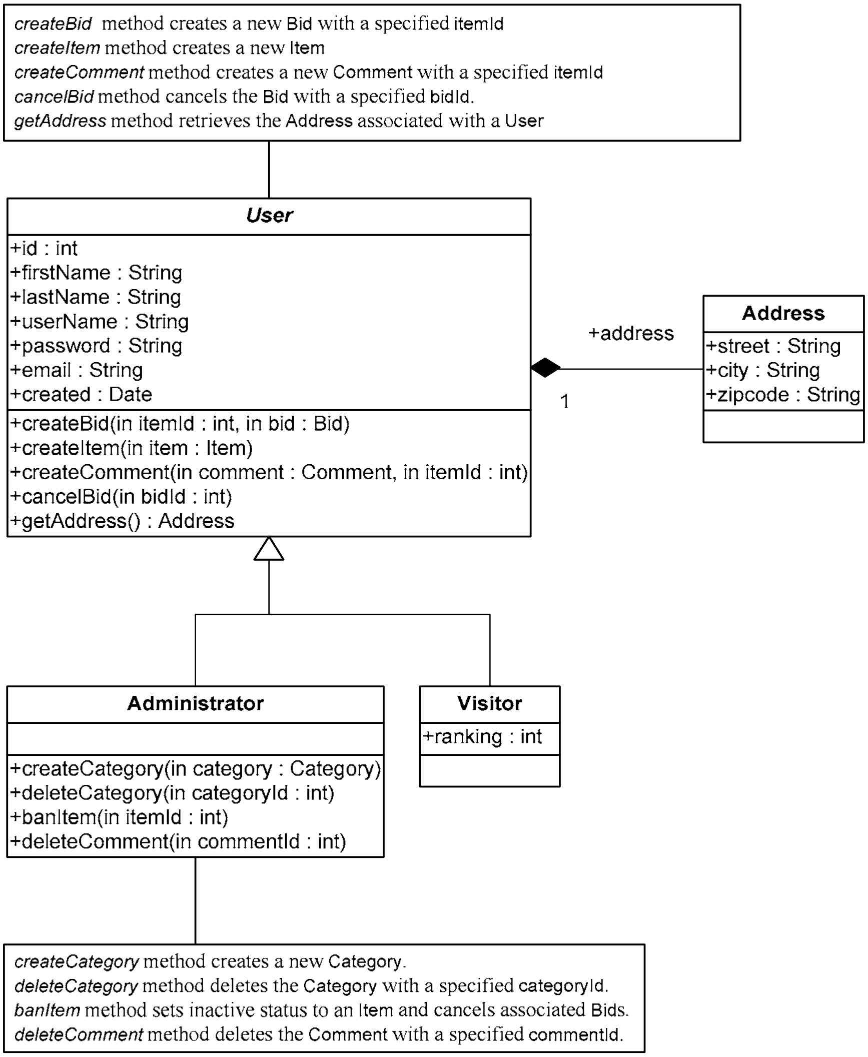


Fig. 3. Classes `Category`, `Item`, `Comment`, and `Bid`

This simple model contains entities that are likely to be found in any typical auction system: `Category`, `Item`, `Bid`, `Comment`, and `User`. In any store, similar goods are categorized by type and grouped into sections and onto shelves. Clearly, our auction catalog requires some kind of hierarchy of item categories. An auction consists of a sequence of bids. One particular bid is a winning bid.

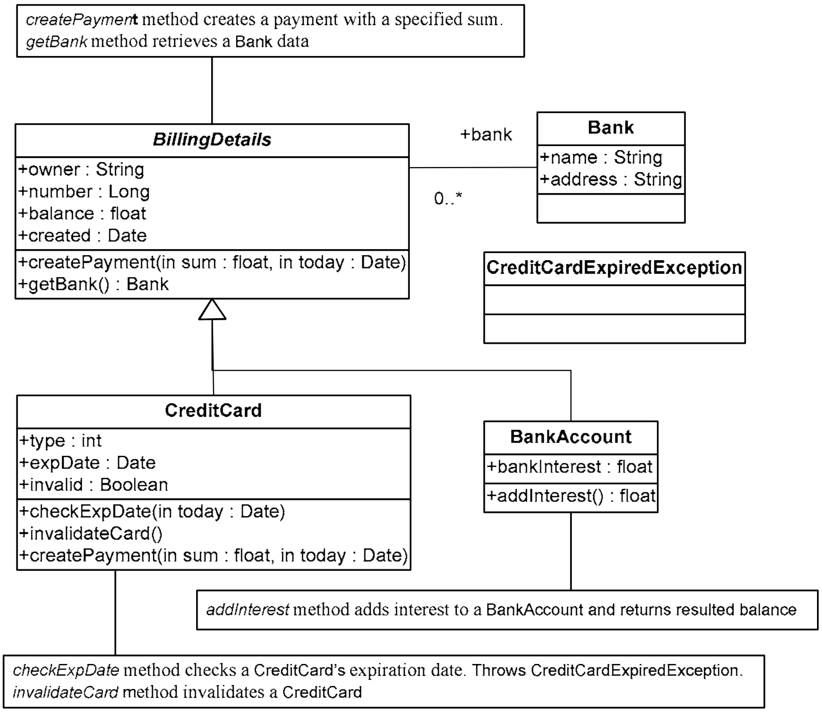
A web of trust is an essential feature of an online auction site. The web of trustallows users serves for building a reputation for trustworthiness (or untrustworthiness). Buyers can create comments about sellers (and vice versa), and the comments are visible to all other users.

The status of a new Item or Bid object is set to 0, which indicates the active status. During the program execution, we can ban an `Item` instance (status field is set to 1) and cancel a `Bid` instance (status field is set to 1). User details include `name`, `login`, `address`, `email address`, and `billing information`.



createBid method creates a new Bid with a specified itemId
createItem method creates a new Item
createComment method creates a new Comment with a specified itemId
cancelBid method cancels the Bid with a specified bidId.
getAddress method retrieves the Address associated with a User
User
+id : int
+firstName : String
+lastName : String
+userName : String
+password : String
+email : String
+created : Date
+createBid(in itemId : int, in bid : Bid)
+createItem(in item : Item)
+createComment(in comment : Comment, in itemId : int)
+cancelBid(in bidId : int)
+getAddress() : Address
+address
1
Address
+street : String
+city : String
+zipcode : String
Administrator
+createCategory(in category : Category)
+deleteCategory(in categoryId : int)
+banItem(in itemId : int)
+deleteComment(in commentId : int)
Visitor
+ranking : int
createCategory method creates a new Category.
deleteCategory method deletes the Category with a specified categoryId.
banItem method sets inactive status to an Item and cancels associated Bids.
deleteComment method deletes the Comment with a specified commentId.

Fig. 4. Class `User` and associated classes



createPayment method creates a payment with a specified sum.
getBank method retrieves a Bank data
BillingDetails
+owner : String
+number : Long
+balance : float
+created : Date
+createPayment(in sum : float, in today : Date)
+getBank() : Bank
+bank
0..*
Bank
+name : String
+address : String
CreditCardExpiredException
CreditCard
+type : int
+expDate : Date
+invalid : Boolean
+checkExpDate(in today : Date)
+invalidateCard()
+createPayment(in sum : float, in today : Date)
BankAccount
+bankInterest : float
+addInterest() : float
addInterest method adds interest to a BankAccount and returns resulted balance
checkExpDate method checks a CreditCard's expiration date. Throws CreditCardExpiredException.
invalidateCard method invalidates a CreditCard

Fig. 5. Class `BillingDetails` and associated classes

Each item may be auctioned only once, so we don't need to make `Item` distinct from the auction entities. Instead, we have a single auction item entity named `Item`. Thus, `Bid` is associated directly with `Item`. Users can write comments about other users only in the context of an auction; hence the association between `Item` and `Comment`. The `Address` information of a `User` is modeled as a separate class, even though the `User` may have only one `Address`. `User` contains two subclasses `Visitor` and `Administrator`.

We do allow the user to have multiple `BillingDetails`. The various billing strategies are represented as subclasses of an abstract class.

A `Category` might be nested inside another `Category`. This is expressed by a recursive association, from the `Category` entity to itself. Note that a single

`Category` may have multiple child categories but at most one parent category.

Each `Item` belongs to at least one `Category`. The entities in a domain model should encapsulate state and behavior.

5.2. Specification of the methods

```
context User::createBid(itemId: int, bid : Bid)
pre itemIdOk:  itemId > 0
pre itemExist: self.items->select(v:Item | id = itemId)->
                                                   notEmpty()
pre bidOk:     def bid
pre bidAmountOk:   bid.amount > 0
pre statusActive:  bid.status = 0
rule : let newBid = oclNew Bid(bid) in
       set self.items->any(v: Item | v.id = itemId).bids->
                                              include(newBid),
           self.bids->include(newBid)
       end
```

The `createBid` method creates a new Bid with a specified `itemId`. The method preconditions check that the `itemId` and `bid.amount` are positive, an Item with the specified `itemId` exists, `bid` is defined and `bid.status` is zero (i.e., it is active). The **`rule`** clause of the `createBid` method indicates the creation of a new Bid instance and simultaneous insertion of the created `newBid` object into collections `self.bids` and `self.items.bids`.

```
context User::createItem(item: Item)
pre itemOk:           def item
pre itemPriceOk: item.initialPrice > 0
pre itemNameOk:  item.name <> null
rule : let newItem = oclNew Item(item) in
       set self.items->include(newItem),
           self.sells->include(newItem)
       end
```

The `createItem` method creates a new Item. The method preconditions check that `item` is defined, `item.name` is not null and `item.initialPrice` is positive. The **`rule`** clause of the `createItem` method indicates the creation of a new Item instance and simultaneous insertion of the created `newItem` object into collections `self.items and self.sells`.

```
context User::createComment(comment: Comment, itemId : int)
pre itemIdOk:     itemId > 0
pre commentOk:     def comment
pre commentTextOk:   comment.text <> null
pre commentRatingOk: comment.rating > 0
```

```
rule : let newComment = oclNew Comment(comment) in
   set newComment.about := self.items->any(v: Item | v.id =
itemId),
       self.comments->include(newComment)
   end
```
The `createComment` method creates a new Comment with a specified `itemId`. The method preconditions check that `comment` is defined, `comment.text` is not null and `itemId` and `comment.rating` are positive. The **`rule`** clause of the method indicates the creation of a new Comment instance, association of created Comment instance with specified by `itemId` Item object and insertion of the created `newComment` object into collection `self.comments`.

```
context User::cancelBid(bidId : int)
pre bidOk: bidId > 0
rule : self.bids->any(v: Bid | v.id = bidId).status := 1
```
The `cancelBid` method cancels the Bid with the specified `bidId`. The method precondition checks that `bidId` is positive. The **`rule`** clause of the method indicates the setting of the cancel status (value 1).

```
context User::getAddress(): Address
post : result = self.address
```
`The getAddress` method retrieves the address associated with a user.

```
context Administrator::createCategory(category: Category)
pre categoryOk: def category
pre categoryNameOk: category.name <> null
rule : let newCategory = oclNew Category(category) in
        self.categories->include(newCategory)
```
The `createCategory` method creates a new Category. The method precondition checks that `category` is defined and `category.name` is not null. The **`rule`** clause of the method indicates the creation of a new Category instance and insertion of the created `newCategory` object into the collection `self.categories`.

```
context Administrator::deleteCategory(categoryId: int)
pre categoryIdOk: categoryId > 0
rule : self.categories->delete(self.categories->
        select(v: Category | v.id = categoryId))
```
The `deleteCategory` method deletes a Category with a specified `categoryId`. The method precondition checks that `categoryId` is positive. The **`rule`** clause of the method contains deletion of the reference of the object specified by `categoryId` from the collection of Category instances `self.categories`.

```
context Administrator::banItem(itemId: int)
pre itemIdOk: itemId > 0
rule : set
        self.items->any(v:Item | v.id = itemId).status := 1,
        for iBid : Bid in
            self.items->any(v: Item | v.id = itemId).bids
        do iBid.status : = 1
       end
```

The `banItem` method sets the inactive status for an Item and the cancel status for the associated Bid objects. The method precondition checks that `itemId` is positive. The **`rule`** clause of the method indicates the setting of the inactive status (value 1) and the cancel status for the associated Bid objects (value 1).

```
context Administrator::deleteComment(commentId: int)
pre commentIdOk: commentId > 0
rule : self.comments->delete(self.comments->
                      select(v: Comment | v.id = commentId))
```

The `deleteComment` method deletes a Comment with a specified `commentId`. The method precondition checks that `commentId` is positive. The **`rule`** clause of the method contains deletion of the reference of the object specified by `commentId` from the collection of Comment instances `self.comments`.

```
context BillingDetails inv: self.balance > -1
```

This invariant checks that the `balance` field of the `BillingDetails` instance is always non-negative.

```
context BillingDetails::createPayment(sum:float,today:Date)
pre sumOk: sum > 0
pre fundsOk: self.balance > sum
rule : self.balance := self.balance - sum
```

The `createPayment` method creates a Payment with a specified `sum`. The method preconditions check that `sum` and `self.balance` are positive.

```
context BillingDetails::getBank():Bank
post : result = self.bank
```

The `getBank` method retrieves a Bank data.

```
context BankAccount::addInterest():float
pre balanceOk: self.balance > 0
rule : seq self.balance := self.balance + self.balance *
self.bankInterest
       result = self.balance
   end
```

The `addInterest` method adds interest to a BankAccount and returns resulted balance. Method preconditions checks that `self.balance` is positive.

```
context CreditCard::checkExpDate(today: Date)
pre :   self.expDate > today
else throw oclNew CreditCardExpiredException()
```

The `checkExpDate` method checks a CreditCard expiration date and throws the exception `CreditCardExpiredException` if it is expired.

```
context CreditCard::createPayment(sum: float, today: Date)
pre sumOk:   sum > 0
pre fundsOk: self.balance > sum

rule : try seq
                self.checkExpDate(today),
                self.balance := self.balance - sum
           end
     catch CreditCardExpiredException  self.invalidateCard()
     end
```

The `createPayment` method is an overridden method of the BillingDetails class. The method preconditions check that `sum` and `self.balance` are positive. The **`rule`** clause indicates the method checks the CreditCard expiration date and either creates a payment or invalidate the card if the method call `self.checkExpDate(today)` throws the exception `CreditCardExpired-Exception`.

```
context CreditCard::invalidateCard()
rule : self.invalid := true
```

The `invalidateCard` method sets the value of the field `self.invalid` to true.

```
context Category::getItems():Collection
post : result = self.items
```

The `getItems` method retrieves all items associated with a Category.

6. CONCLUSION

An extension of the object constraint language OCL by imperative facilities is proposed in the paper. This extension permits an adequate specification of state-updating methods of a class. The facilities are tested by an experimental specification of a representative example class diagram including 13 classes. The experiment has covered all the proposed facilities and shown that they provide a simple and powerful means for method specification. The next step of this research project is to construct a compiler translating a specification into an executable code with the aim of prototyping a specified program system.

REFERENCES

[1] *OMG UML 1.5 – Final Adopted Specification*. OMG, March 2003. — Available electronically from `http://www.omg.org/uml`.

[2] Brazhnik S., Zamulin A.. *IOCL — an Imperative OCL extension*. Working draft, available electronically from http://`www.iis.nsk.su/persons/zamulin/iocl_specification.doc`

[3] Gurevich Y.. *May 1997 Draft of the ASM Guide*. — Available electronically from `http://www.eecs.umich.edu/gasm/`.

С. А. Бражник, А. В. Замулин

ИМПЕРАТИВНОЕ РАСШИРЕНИЕ ЯЗЫКА СПЕЦИФИКАЦИИ ОБЪЕКТОВ OCL

Препринт
123

Рукопись поступила в редакцию 27.04.05
Редактор Т.М. Бульонкова

Подписано в печать 04.07.05
Формат бумаги 60×84 1/16 Объем 1.7 уч.-изд.л., 1.9 п.л.
Тираж 60 экз.

ЗАО РИЦ «Прайс-курьер»
630090, г. Новосибирск, пр. Акад. Лаврентьева, 6, тел. (383) 330-72-02