Chapter 3

The Epsilon Object Language (EOL)

The primary aim of EOL is to provide a reusable set of common model management facilities, atop which task-specific languages can be implemented. However, EOL can also be used as a general-purpose standalone model management language for automating tasks that do not fall into the patterns targeted by task-specific languages. This section presents the syntax and semantics of the language using a combination of abstract syntax diagrams, concrete syntax examples and informal discussion.

3.1 Module Organization

In this section the syntax of EOL is presented in a top-down manner. As displayed in Figure 3.1, EOL programs are organized in *modules*. Each module defines a *body* and a number of *operations*. The body is a block of statements that are evaluated when the module is executed. Each operation defines the kind of objects on which it is applicable (*context*), a *name*, a set of *parameters* and optionally a *return type*. Modules can also import other modules using *import* statements, and access their operations.

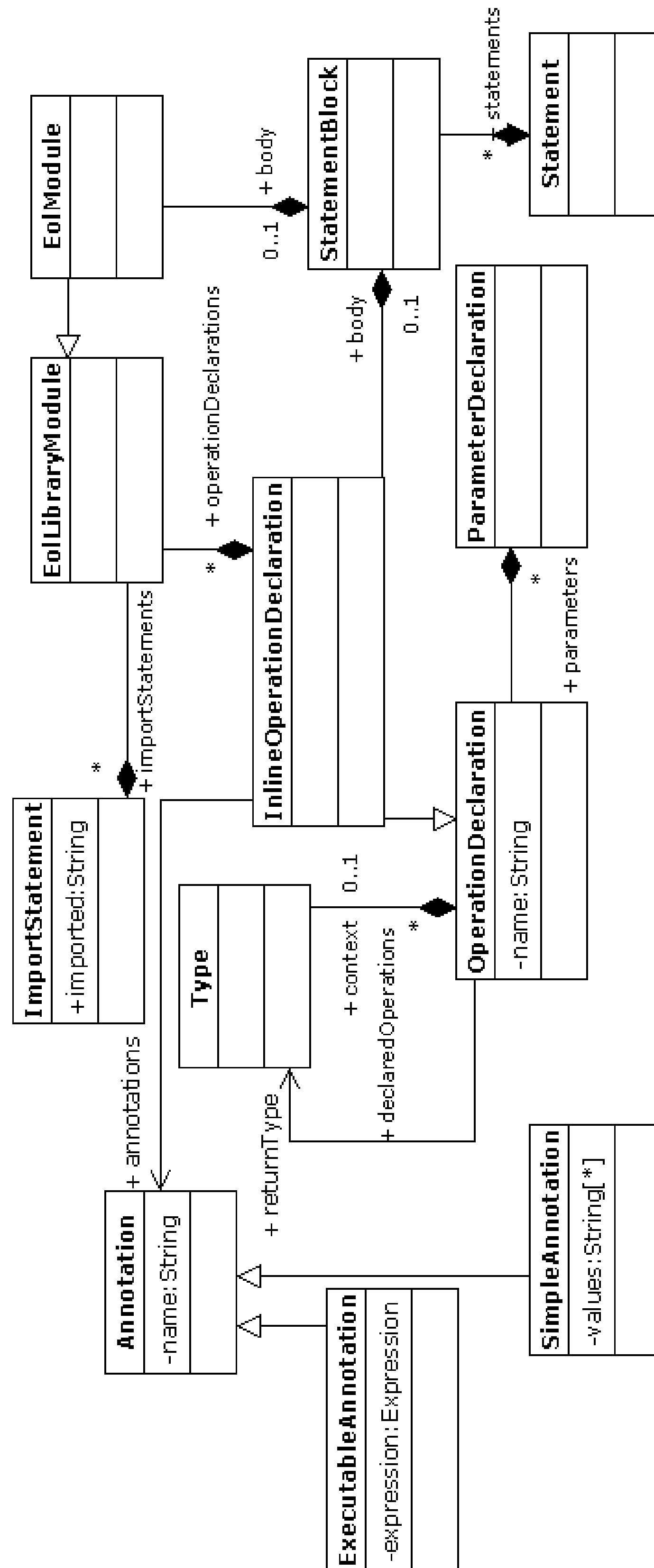

Figure 3.1: EOL Module Structure

```
1.add1().add2().println();

operation Integer add1() : Integer {
 return self + 1;
}

operation Integer add2() : Integer {
 return self + 2;
}
```

Listing 3.1: Exemplar context-defining EOL operations

3.2 User-Defined Operations

In typical object oriented languages such as Java and C++, operations are defined inside classes and can be invoked on instances of those classes. EOL on the other hand is not object-oriented in the sense that it does not define classes itself, but nevertheless needs to manage objects of types defined externally to it (e.g. in metamodels). By defining the context-type of an operation explicitly, the operation can be called on instances of the type as if it was natively defined by the type. Alternatively, context-less operations could be defined; however the adopted technique significantly improves readability of the concrete syntax.

For example, consider the code excerpts displayed in Listings 3.1 and 3.2. In Listing 3.1, the operations *add1* and *add2* are defined in the context of the built-in *Integer* type, which is specified before their names. Therefore, in line 1 they can be invoked using the *1.add1().add2()* expression: the context (the integer *1*) will be assigned to the special variable *self*. On the other hand, in Listing 3.2 where no context is defined, they have to be invoked in a nested manner which follows an in-to-out direction instead of the left to right direction used by the former excerpt. As complex model queries often involve invoking multiple properties and operations, this technique is particularly beneficial to the overall readability of the code.

```
add2(add1(1)).println();

operation add1(base : Integer) : Integer {
  return base + 1;
}

operation add2(base : Integer) : Integer {
  return base + 2;
}
```

Listing 3.2: Exemplar EOL context-less EOL operations

```
"1".test();
1.test();

operation String test() {
  (self + " is a string").println();
}

operation Integer test() {
  (self + "is an integer").println();
}
```

Listing 3.3: Demonstration of polymorphism in EOL

EOL supports polymorphic operations using a runtime dispatch mechanism. Multiple operations with the same name and parameters can be defined, each defining a distinct context type. For example, in Listing 3.3, the statement in line 1 invokes the test operation defined in line 4, while the statement in line 2 invokes the test operation defined in line 8.

3.2.1 Annotations

EOL supports two types of annotations: simple and executable. A simple annotation specifies a name and a set of String values while an executable annotation specifies a name and an expression. The concrete syntaxes of simple and executable annotations are displayed in Listings 3.4 and 3.5 respectively. Several examples for simple annotations are shown in List-

```
1 @name value(,value)*
```

Listing 3.4: Concrete syntax of simple annotations

```
1 $name expression
```

Listing 3.5: Concrete syntax of executable annotations

```
1 @colors red
2 @colors red, blue
3 @colors red, blue, green
```

Listing 3.6: Examples of simple annotations

ing 3.6. Examples for executable annotations will be given in the following sections.

In stand-alone EOL, annotations are supported only in the context of operations, however as discussed in the sequel, task-specific languages also make use of annotations in their constructs, each with task-specific semantics. EOL operations support three particular annotations: the *pre* and *post* executable annotations for specifying pre and post-conditions, and the *cached* simple annotation, which are discussed below.

3.2.2 Pre/post conditions in user-defined operations

A number of *pre* and *post* executable annotations can be attached to EOL operations to specify the pre- and post-conditions of the operation. When an operation is invoked, before its body is evaluated, the expressions of the *pre* annotations are evaluated. If all of them return *true*, the body of the operation is processed, otherwise, an error is raised. Similarly, once the body of the operation has been executed, the expressions of the *post* annotations of the operation are executed to ensure that the operation has had the desired effects. *Pre* and *post* annotations can access all the variables in the parent scope, as well as the parameters of the operation

```
1.add(2);
1.add(-1);

$pre i > 0
$post _result > self
operation Integer add(i : Integer) : Integer {
  return self + i;
}
```

Listing 3.7: Example of pre- and post-conditions in an EOL operation

and the object on which the operation is invoked (through the *self* variable). Moreover, in *post* annotations, the returned value of the operation is accessible through the built-in *_result* variable. An example of using pre and post conditions in EOL appears in Listing 3.7.

In line 4 the *add* operation defines a pre-condition stating that the parameter *i* must be a positive number. In line 5, the operation defines that result of the operation (*_result*) must be greater than the number on which it was invoked (*self*). Thus, when executed in the context of the statement in line 1 the operation succeeds, while when executed in the context of the statement in line 2, the pre-condition is not satisfied and an error is raised.

3.2.3 Operation Result Caching

EOL supports caching the results of parameter-less operations using the *@cached* simple annotation. In the following example, the Fibonacci number of a given Integer is calculated using the *fibonacci* recursive operation displayed in Listing 3.8. Since the *fibonacci* operation is declared as *cached*, it is only executed once for each distinct Integer and subsequent calls on the same target return the cached result. Therefore, when invoked in line 1, the body of the operation is called 16 times. By contrast, if no *@cached* annotation was specified, the body of the operation would be called recursively 1973 times. This feature is particularly useful

```
15.fibonacci().println();

@cached
operation Integer fibonacci() : Integer {
 if (self = 1 or self = 0) {
  return 1;
 }
 else {
  return (self-1).fibonacci() + (self-2).fibonacci();
 }
}
```

Listing 3.8: Calculating the Fibonacci number using a cached operation

for performing queries on large models and caching their results without needing to introduce explicit variables that store the cached results.

3.3 Types

As is the case for most programming languages, EOL defines a built-in system of types, illustrated in Figure 3.2. The *Any* type, inspired by the *OclAny* type of OCL, is the basis of all types in EOL including Collection types. The operations supported by instances of the Any type are outlined in Table 3.1[1].

[1]Parameters within square braces [ ] are optional

Table 3.1: Operations of type Any

Signature	**Description**
isDefined() : Boolean	Returns true if the object is defined and false otherwise
isUndefined() : Boolean	Returns true if the object is undefined and false otherwise
ifUndefined(alt : Any) : Any	If the object is undefined, it returns alt else it returns the object
isTypeOf(type : Type) : Boolean	Returns true if the object is of the given type and false otherwise
isKindOf(type : Type) : Boolean	Returns true if the object is of the given type or one of its subtypes and false otherwise
type() : Type	Returns the type of the object. The EOL type system is illustrated in Figure 3.2
asString() : String	Returns a string representation of the object
asInteger() : Integer	Returns an Integer based on the string representation of the object. If the string representation is not of an acceptable format, an error is raised
asReal() : Real	Returns a Real based on the string representation of the object. If the string representation is not of an acceptable format, an error is raised

asBoolean() : Boolean	Returns a Boolean based on the string representation of the object. If the string representation is not of an acceptable format, an error is raised
asBag() : Bag	Returns a new Bag containing the object
asSequence() : Sequence	Returns a new Sequence containing the object
asSet() : Set	Returns a new Set containing the object
asOrderedSet() : OrderedSet	Returns a new OrderedSet containing the object
print([prefix : String]) : Any	Prints a string representation of the object on which it is invoked prefixed with the optional *prefix* string and returns the object on which it was invoked. In this way, the *print* operation can be used for debugging purposes in a non-invasive manner
println([prefix : String]) : Any	Has the same effects with the *print* operation but also produces a new line in the output stream.

format([pattern : String]) : String	Uses the provided pattern to form a String representation of the object on which the method is invoked. The pattern argument must conform to the format string syntax defined by Java[2].

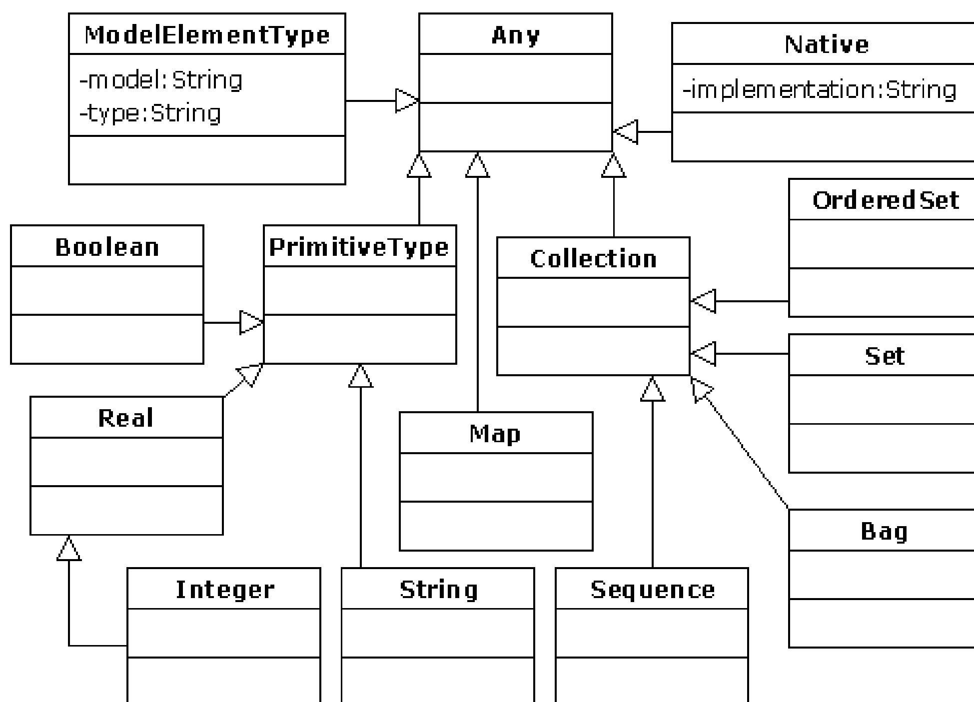

Figure 3.2: Overview of the type system of EOL

[2] http://download.oracle.com/javase/6/docs/api/java/util/Formatter.html#syntax

3.3.1 Primitive Types

EOL provides four primitive types: String, Integer, Real and Boolean. The String type represents finite sequences of characters and supports the following operations which can be invoked on its instances.

Table 3.2: Operations of type String

Signature	**Description**
charAt(index : Integer) : String	Returns the character in the specified index
concat(str : String) : String	Returns a concatenated form of the string with the *str* parameter
length() : Integer	Returns the number of characters in the string
toLowerCase() : String	Returns a new string where all the characters have been converted to lower case
firstToLowerCase() : String	Returns a new string the first character of which has been converted to lower case
toUpperCase() : String	Returns a new string where all the characters have been converted to upper case
firstToUpperCase() : String	Returns a new string, the first character of which has been converted to upper case
isSubstringOf(str : String) : Boolean	Returns true iff the string the operation is invoked on is a substring of *str*

matches(reg : String) : Boolean	Returns true if there are occurrences of the regular expression *reg* in the string
replace(source : String, target : String) : String	Returns a new string in which all instances of *source* have been replaced with instances of *target*
split(reg : String) : Sequence(String)	Splits the string using as a delimiter the provided regular expression, *reg*, and returns a sequence containing the parts
startsWith(str : String) : Boolean	Returns true iff the string starts with *str*
endsWith(str : String) : Boolean	Returns true iff the string ends with *str*
isInteger() : Boolean	Returns true iff the string is an integer
isReal() : Boolean	Returns true iff the string is a real number
toCharSequence() : Sequence(String)	Returns a sequence containing all the characters of the string
substring(index : Integer) : String	Returns a sub-string of the string starting from the specified *index* and extending to the end of the original string
substring(startIndex : Integer, endIndex : Integer) : String	Returns a sub-string of the string starting from the specified *startIndex* and ending at *endIndex*

pad(length : Integer, padding : String, right : Boolean) : String	Pads the string up to the specified length with specified padding (e.g. "foo".pad(5, "*", true) returns "foo**")
trim() : String	Returns a trimmed copy of the string

The Real type represents real numbers and provides the following operations.

Table 3.3: Operations of type Real

Signature	**Description**
ceiling() : Integer	Returns the nearest Integer that is larger than the real
floor() : Integer	Returns the nearest Integer that is greater than the real
round() : Integer	Rounds the real to the nearest Integer
pow(exponent : Real) : Real	Returns the real to the power of exponent
log() : Real	Returns the natural logarithm of the real
log10() : Real	Returns the 10-based logarithm of the real
abs() : Real	Returns the absolute value of the real
max(other : Real) : Real	Returns the maximum of the two reals

min(other : Real) : Real	Returns the minimum of the two reals

The Integer type represents natural numbers and negatives and extends the Real primitive type. It also defines the following operations:

Table 3.4: Operations of type Integer

Signature	Description
to(other : Integer) : Sequence(Integer)	Returns a sequence of integers (e.g. 1.to(5) returns Sequence{1,2,3,4,5})
iota(end : Integer, step : Integer) : Sequence(Integer)	Returns a sequence of integers up to *end* using the specified step (e.g. 1.iota(10,2) returns Sequence{1,3,5,7,9})

Finally, the Boolean type represents true/false states and provides no additional operations to those provided by the base Any type.

3.3.2 Collections and Maps

EOL provides four types of collections and a Map type. The Bag type represents non-unique, unordered collections, the Sequence type represents non-unique, ordered collections, the Set type represents unique and unordered collections and the OrderedSet represents unique and ordered collections.

All collection types inherit from the abstract Collection type. Apart from simple operations, EOL also supports first-order logic operations on collections. The following operations apply to all types of collections:

Table 3.5: Operations of type Collection

Signature	**Description**
add(item : Any)	Adds an item to the collection. If the collection is a set, addition of duplicate items has no effect
addAll(col : Collection)	Adds all the items of the *col* argument to the collection. If the collection is a set, it only adds items that do not already exist in the collection
remove(item : Any)	Removes an *item* from the collection
removeAll(col : Collection)	Removes all the items of *col* from the collection
clear()	Empties the collection
includes(item : Any) : Boolean	Returns true if the collection includes the *item*
excludes(item : Any) : Boolean	Returns true if the collection excludes the *item*
includesAll(col : Collection) : Boolean	Returns true if the collection includes all the items of collection *col*
excludesAll(col : Collection) : Boolean	Returns true if the collection excludes all the items of collection col
including(item : Any) : Collection	Returns a new collection that also contains the *item* – unlike the add() operation that adds the *item* to the collection itself

excluding(item : Any) : Collection	Returns a new collection that excludes the item – unlike the remove() operation that removes the *item* from the collection itself
includingAll(col : Collection) : Collection	Returns a new collection that is a union of the two collections. The type of the returned collection (i.e. Bag, Sequence, Set, OrderedSet) is same as the type of the collection on which the operation is invoked
excludingAll(col : Collection) : Collection	Returns a new collection that excludes all the elements of the col collection
flatten() : Collection	Recursively flattens all items that are of collection type and returns a new collection where no item is a collection itself
count(item : Any) : Integer	Returns the number of times the item exists in the collection
size() : Integer	Returns the number of items the collection contains
isEmpty() : Boolean	Returns true if the collection does not contain any elements and false otherwise
random() : Any	Returns a random item from the collection
clone() : Collection	Returns a new collection of the same type containing the same items with the original collection

concat() : String	Returns the string created by converting each element of the collection to a string
concat(separator : String) : String	Returns the string created by converting each element of the collection to a string, using the given argument as a separator

The following operations apply to ordered collection types (i.e. Sequence and OrderedSet):

Table 3.6: Operations of types Sequence and OrderedSet

Signature	**Description**
first() : Any	Returns the first item of the collection
last() : Any	Returns the last item of the collection
at(index : Integer) : Any	Returns the item of the collection at the specified index
removeAt(index : Integer) : Any	Removes and returns the item at the specified index.
indexOf(item : Any) : Integer	Returns the index of the item in the collection or -1 if it does not exist
invert() : Collection	Returns an inverted copy of the collection

Also, EOL collections support the following first-order operations:

Table 3.7: First-order logic operations on Collections

Signature	**Description**
select(iterator : Type | condition) : Collection	Returns a sub-collection containing only items of the specified type that satisfy the condition
selectOne(iterator : Type | condition) : Any	Returns the first element that satisfies the condition
reject(iterator : Type | condition) : Collection	Returns a sub-collection containing only items of the specified type that do not satisfy the condition
collect(iterator : Type | expression) : Collection	Returns a collection containing the results of evaluating the expression on each item of the collection that is of the specified type
closure(iterator : Type | expression) : Collection	Returns a collection containing the results of evaluating the transitive closure of the results produced by the expression on each item of the collection that is of the specified type
aggregate(iterator : Type | keyExpression, valueExpression) : Map	Returns a map containing key-value pairs produced by evaluating the key and value expressions on each item of the collection that is of the specified type
one(iterator : Type | condition) : Boolean	Returns true if there exists exactly one item in the collection that satisfies the condition

exists(iterator : Type | condition) : Boolean	Returns true if there exists at least one item in the collection that satisfies the condition
forAll(iterator : Type | condition) : Boolean	Returns true if all items in the collection satisfy the condition
sortBy(iterator: Type | expression) : Collection	Returns a copy of the collection sorted by the results of evaluating the expression on each item of the collection that conforms to the iterator type

The Map type represents an array of key-value pairs in which the keys are unique. The type provides the following operations.

Table 3.8: Operations of type Map

Signature	**Description**
put(key : Any, value : Any)	Adds the key-value pair to the map. If the map already contains the same key, the value is overwritten
get(key : Any) : Any	Returns the value for the specified keys
containsKey(key : Any) : Boolean	Returns true if the map contains the specified key
keySet() : Set	Returns the keys of the map
values() : Bag	Returns the values of the map
clear()	Clears the map

```
1 var frame = new Native("javax.swing.JFrame");
2 frame.title = "Opened with EOL";
3 frame.setBounds(100,100,300,200);
4 frame.visible = true;
```

Listing 3.9: Demonstration of NativeType in EOL

```
1 var file = new Native("java.io.File")("myfile.txt");
2 file.absolutePath.println();
```

Listing 3.10: Demonstration of NativeType in EOL

3.3.3 Native Types

As discussed earlier, while the purpose of EOL is to provide significant expressive power to enable users to manage models at a high level of abstraction, it is not intended to be a general-purpose programming language. Therefore, there may be cases where users need to implement some functionality that is either not efficiently supported by the EOL runtime (e.g. complex mathematical computations) or that EOL does not support at all (e.g. developing user interfaces, accessing databases). To overcome this problem, EOL enables users to create objects of the underlying programming environment by using *native* types. A native type specifies an $implementation$ property that indicates the unique identifier for an underlying platform type. For instance, in a Java implementation of EOL the user can instantiate and use a Java class via its class identifier. Thus, in Listing 3.9 the EOL excerpt creates a Java window (Swing JFrame) and uses its methods to change its title and dimensions and make it visible.

To pass arguments to the constructor of a native type, a parameter list must be added, such as that in Listing 3.10.

3.3.4 Model Element Types

A model element type represents a meta-level classifier. As discussed in Section 2, Epsilon intentionally refrains from defining more details about the meaning of a model element type to be able to support diverse modelling technologies where a type has different semantics. For instance a MOF class, an XSD complex type and a Java class can all be regarded as model element types according to the implementation of the underlying modelling framework.

In case of multiple models, as well as the name of the type, the name of the model is also required to resolve a particular type since different models may contain elements of homonymous but different model element types. In case a model defines more than one type with the same name (e.g. in different packages), a fully qualified type name must be provided.

In terms of concrete syntax, inspired by ATL, the ! character is used to separate the name of the type from the name of the model it is defined in. For instance $Ma!A$ represents the type A of model Ma. Also, to support modelling technologies that provide hierarchical grouping of types (e.g. using packages) the :: notation is used to separate between packages and classes. A model element type supports the following operations:

Table 3.9: Operations of Model Element Types

Signature	**Description**
allOfType() : Set	Returns all the elements in the model that are instances of the type
allOfKind() : Set	Returns all the elements in the model that are instances either of the type itself or of one of its subtypes

```
1 UML14!Core::Foundation::Class.allInstances();
```

Listing 3.11: Demonstration of the concrete syntax for accessing model element types

allInstances() : Set	Alias for allOfKind() (for compatibility with OCL)
all() : Set	Alias for allOfKind() (for syntax-compactness purposes)
isInstantiable() : Boolean	Returns true if the type is instantiable (i.e. non-abstract)
createInstance() : Any	Creates an instance of the type in the model

As an example of the concrete sytnax, Listing 3.11 retrieves all the instances of the Class type (including instances of its subtypes) defined in the Core package of the UML 1.4 metamodel that are contained in the model named UML14.

3.4 Expressions

3.4.1 Literal Values

EOL provides special syntax constructs to create instances of each of the built-in types:

Integer literals are defined by using one or more decimal digits (such as *42* or *999*). Optionally, long integers (with the same precision as a Java *Long*) can be produced by adding a “l” suffix, such as *42l*.

Real literals are defined by:

- Adding a decimal separator and non-empty fractional part to the integer part, such as *42.0* or *3.14*. Please note that *.2* and *2.* are *not* valid literals.
- Adding a floating point suffix: "f" and "F" denote single precision, and "d" and "D" denote double precision. For example, *2f* or *3D*.
- Adding an exponent, such as *2e+1* (equal to *2e1*) or *2e-1*.
- Using any combination of the above options.

String literals are sequences of characters delimited by single (`'hi'`) or double (`"hi"`) quotes. Quotes inside the string can be escaped by using a backslash, such as in `'A\'s'` or `"A\"s"`. Literal backslashes need to be escaped as well, such as in `'A\\B'`. Special escape sequences are also provided: `\n` for a newline, `\t` for a horizontal tab and `\r` for a carriage return, among others.

Boolean literals use the *true* reserved keyword for the true Boolean value, and *false* reserved keyword for the false Boolean value.

Sequence and most other collections (except *Maps*) also have literals. Their format is *T* {*e*}, where *T* is the name of the type and *e* are zero or more elements, separated by commas. For instance, *Sequence* {} is the empty sequence, and *Set* {*1, 2, 3*} is the set of numbers between 1 and 3.

Map literals are similar to the sequential collection literals, but their elements are of the form *key = value*. For instance, *Map* {*'a' = 1, 'b' = 2*} is a map which has two keys, "a" and "b", which map to the integer values 1 and 2, respectively.

Please note that, when defining an element such as *1 = 2 = 3*, the key would be *1* and the value would be the result of evaluating *2 =*

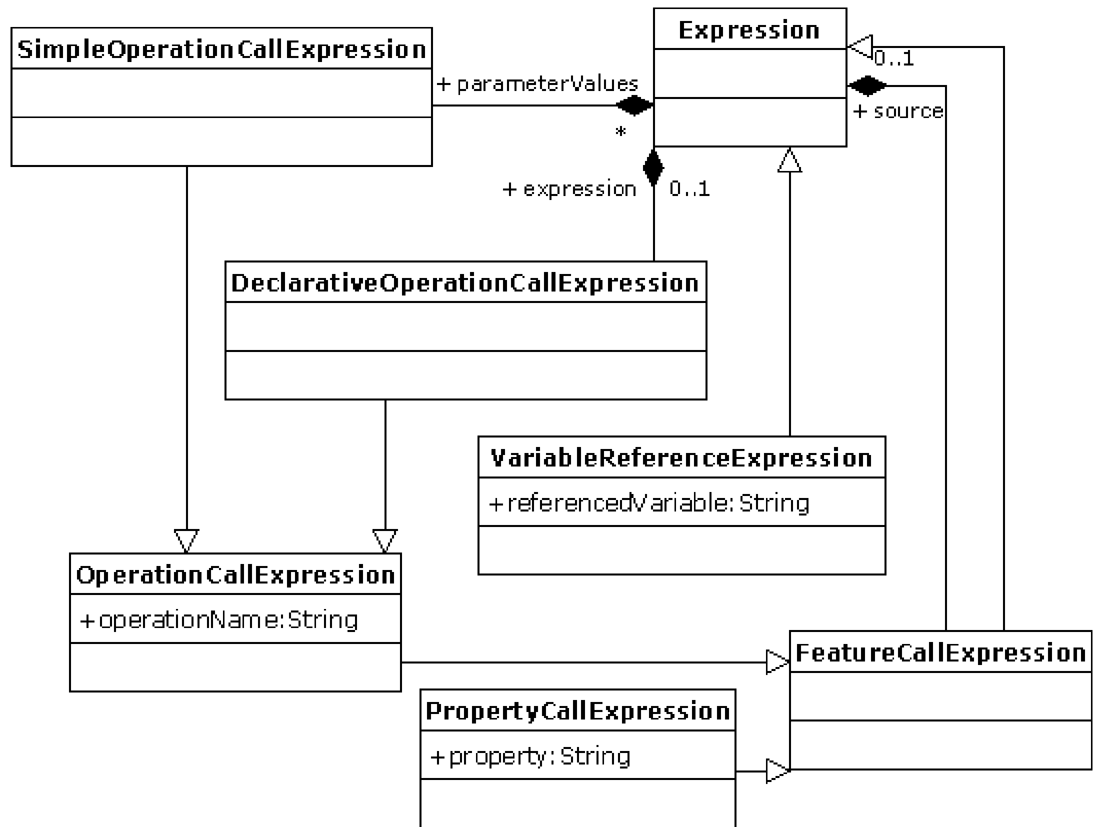

Figure 3.3: Overview of the feature navigation EOL expressions

3 (false). If you would like to use the result of the expression *1 = 2* as key, you will need to enclose it in parenthesis, such as in *(1 = 2) = 3*.

3.4.2 Feature Navigation

Since EOL needs to manage models defined using object oriented modelling technologies, it provides expressions to navigate properties and invoke simple and declarative operations on objects (as presented in Figure 3.3).

```
"Something".println();

operation Any println() : Any {
  ("Printing : " + self)->println();
}
```

Listing 3.12: Invoking operations using EOL

In terms of concrete syntax, '.' is used as a uniform operator to access a property of an object and to invoke an operation on it. The '→' operator, which is used in OCL to invoke first-order logic operations on sets, has been also preserved for syntax compatibility reasons. In EOL, every operation can be invoked both using the '.' or the '→' operators, with a slightly different semantics to enable overriding the built-in operations. If the '.' operator is used, precedence is given to the user-defined operations, otherwise precedence is given to the built-in operations. For instance, the Any type defines a println() method that prints the string representation of an object to the standard output stream. In Listing 3.12, the user has defined another parameterless println() operation in the context of Any. Therefore the call to println() in Line 1 will be dispatched to the user-defined println() operation defined in line 3. In its body the operation uses the '→' operator to invoke the built-in println() operation (line 4).

3.4.3 Arithmetical and Comparison Operators

EOL provides common operators for performing arithmetical computations and comparisons illustrated in Tables 3.10 and 3.11 respectively.

Table 3.10: Arithmetical operators

Operator	Description
+	Adds reals/integers and concatenates strings

−	Subtracts reals/integers
− (unary)	Returns the negative of a real/integer
*	Multiplies reals/integers
/	Divides reals/integers

Table 3.11: Comparison operators

Operator	**Description**
=	Returns true if the left hand side equals the right hand side. In the case of primitive types (String, Boolean, Integer, Real) the operator compares the values; in the case of objects it returns true if the two expressions evaluate to the same object
<>	Is the logical negation of the (=) operator
>	For reals/integers returns true if the left hand side is greater than the right hand side number
<	For reals/integers returns true if the left hand side is less than then right hand side number
>=	For reals/integers returns true if the left hand side is greater or equal to the right hand side number

<=	For reals/integers returns true if the left hand side is less or equal to then right hand side number

3.4.4 Logical Operators

EOL provides common operators for performing logical computations illustrated in Table 3.12. Logical operations apply only to instances of the Boolean primitive type.

Table 3.12: Logical Operators

Operator	**Description**
and	Returns the logical conjunction of the two expressions
or	Returns the logical disjunction of the two expressions
not	Returns the logical negation of the expression
implies	Returns the logical implication of the two expressions. Implication is calculated according to the truth table 3.13
xor	returns true if only one of the involved expressions evaluates to true and false otherwise

Table 3.13: Implies Truth Table

Left	**Right**	**Result**

true	true	true
true	false	false
false	true	true
false	false	true

3.4.5 Enumerations

EOL provides the # operator for accessing enumeration literals. For example, the VisibilityEnum#vk_public expression returns the value of the literal *vk_public* of the *VisibilityEnum* enumeration. For EMF metamodels, *VisibilityEnum#vk_public.instance* can also be used.

3.5 Statements

3.5.1 Variable Declaration Statement

A variable declaration statement declares the name and (optionally) the type and initial value of a variable in an EOL program. If no type is explicitly declared, the variable is assumed to be of type Any. For variables of primitive type, declaration automatically creates an instance of the type with the default values presented in Table 3.14. For non-primitive types the user has to explicitly assign the value of the variable either by using the *new* keyword or by providing an initial value expression. If neither is done the value of the variable is undefined. Variables in EOL are strongly-typed. Therefore a variable can only be assigned values that conform to its type (or a sub-type of it).

Scope The scope of variables in EOL is generally limited to the block of statements where they are defined, including any nested blocks. Nevertheless, as discussed in the sequel, there are cases in task-specific lan-

Table 3.14: Default values of primitive types

Type	**Default value**
Integer	0
Boolean	false
String	""
Real	0.0

guages that build atop EOL where the scope of variables is expanded to other non-nested blocks as well. EOL also allows variable shadowing; that is to define a variable with the same name in a nested block that overrides a variable defined in an outer block.

In Listing 3.13, an example of declaring and using variables is provided. Line 1 defines a variable named *i* of type *Integer* and assigns it an initial value of *5*. Line 2 defines a variable named *c* of type *Class* (from model Uml) and creates a new instance of the type in the model (by using the *new* keyword). The commented out assignment statement of line 3 would raise a runtime error since it would attempt to assign a *String* value to an *Integer* variable. The condition of line 4 returns true since the *c* variable has been initialized before. Line 5 defines a new variable also named *i* that is of type *String* and which overrides the *Integer* variable declared in line 1. Therefore the assignment statement of line 6 is legitimate as it assigns a string value to a variable of type String. Finally, as the program has exited the scope of the *if* statement, the assignment statement of line 7 is also legitimate as it refers to the *i* variable defined in line 1.

3.5.2 Assignment Statement

The assignment statement is used to update the values of variables and properties of native objects and model elements.

```
var i : Integer = 5;
var c : new Uml!Class;
//i = "somevalue";
if (c.isDefined()) {
  var i : String;
  i = "somevalue";
}
i = 3;
```

Listing 3.13: Example illustrating declaration and use of variables

```
var a : new Uml!Class;
var b = a;
a.name = "Customer";
b.name.println();
```

Listing 3.14: Assigning the value of a variable by reference

Variable Assignment When the left hand side of an assignment statement is a variable, the value of the variable is updated to the object to which the right hand side evaluates to. If the type of the right hand side is not compatible (kind-of relationship) with the type of the variable, the assignment is illegal and a runtime error is raised. Assignment to objects of primitive types is performed by value while assignment to instances of non-primitive values is performed by reference. For example, in Listing 3.14, in line 1 the value of the a variable is set to a new Class in the Uml model. In line 2, a new untyped variable b is declared and its value is assigned to a. In line 3 the name of the class is updated to Customer and thus, line 4 prints Customer to the standard output stream. On the other hand, in Listing 3.15, in line 1 the a String variable is declared. In line 2 an untyped variable b is declared. In line 3, the value of a is changed to Customer (which is an instance of the primitive *String* type). This has no effect on b and thus line 4 prints an empty string to the standard output stream.

```
var a : String;
var b = a;
a = "Customer";
b.println();
```

Listing 3.15: Assigning the value of a variable by value

```
EStructuralFeature feature = x.eClass().getEStructuralFeature("y");
x.eSet(feature, a);
```

Listing 3.16: Java code that assigns the value of a property of a model element that belongs to an EMF-based model

Native Object Property Assignment When the left hand side of the assignment is a property of a native object, deciding on the legality and providing the semantics of the assignment is delegated to the execution engine. For example, in a Java-based execution engine, given that x is a native object, the statement $x.y = a$ may be interpreted as *x.setY(a)* or if x is an instance of a map *x.put("x",a)*. By contrast, in a C# implementation, it can be interpreted as $x.y = a$ since the language natively supports properties in classes.

Model Element Property Assignment When the left hand side of the assignment is a property of a model element, the model that owns the particular model element (accessible using the *ModelRepository.getOwningModel()* operation) is responsible for implementing the semantics of the assignment using its associated *propertyGetter* as discussed in Section 2.5. For example, if x is a model element, the statement $x.y = a$ may be interpreted using the Java code of Listing 3.16 if x belongs to an EMF-based model or using the Java code of Listing 3.17 if it belongs to an MDR-based model.

```
1  StructuralFeature feature = findStructuralFeature(x.refClass(), "y");
2  x.refSetValue(feature, a);
```

Listing 3.17: Java code that assigns the value of a property of a model element that belongs to an MDR-based model

3.5.3 Special Assignment Statement

In task-specific languages, an assignment operator with task-specific semantics is often required. Therefore, EOL provides an additional assignment operator. In standalone EOL, the operator has the same semantics with the primary assignment operator discussed above, however task-specific languages can redefine its semantics to implement custom assignment behaviour. For example, consider the simple model-to-model transformation of Listing 3.18 where a simple object oriented model is transformed to a simple database model using an ETL (see Section 5) transformation. The Class2Table rule transforms a Class of the OO model into a Table in the DB model and sets the name of the table to be the same as the name of the class. Rule Atribute2Column transforms an Attribute from the OO model into a column in the DB model. Except for setting its name (line 12), it also needs to define that the column belongs to the table which corresponds to the class that defines the source attribute. The commented-out assignment statement of line 13 cannot be used for this purpose since it would illegaly attempt to assign the owningTable feature of the column to a model element of an inappropriate type (OO!Class). However, the special assignment operator in the task-specific language implements the semantics discussed in Section 5.5.4, and thus in line 14 it assigns to the owningTable feature not the class that owns the attribute but its corresponding table (calculated using the Class2Table rule) in the DB model.

```
rule Class2Table
 transform c : OO!Class
 to t : DB!Table {

 t.name = c.name;
}

rule Attribute2Column
 transform a : OO!Attribute
 to c : DB!Column {

 c.name = a.name;
 --c.owningTable = c.owningClass;
 c.owningTable ::= c.owningClass;
}
```

Listing 3.18: A simple model-to-model transformation demonstrating the special assignment statement

```
if (a > 0) {
 "A is greater than 0".println();
}
else { "A is less equal than 0".println(); }
```

Listing 3.19: Example illustrating an if statement

3.5.4 If Statement

As in most programming languages, an if statement consists of a condition, a block of statements that is executed if the condition is satisfied and (optionally) a block of statements that is executed otherwise. As an example, in Listing 3.19, if variable a holds a value that is greater than 0 the statement of line 3 is executed, otherwise the statement of line 5 is executed.

3.5.5 Switch Statement

A switch statement consists of an expression and a set of cases, and can be used to implement multi-brancing. Unlike Java/C, switch in EOL doesn't

```
1  var i = "2";
2
3  switch (i) {
4   case "1" : "1".println();
5   case "2" : "2".println();
6   case "3" : "3".println();
7   case default : "default".println();
8  }
```

Listing 3.20: Example illustrating a switch statement

```
1  var i = "2";
2
3  switch (i) {
4   case "1" : "1".println();
5   case "2" : "2".println(); continue;
6   case "3" : "3".println();
7   case default : "default".println();
8  }
```

Listing 3.21: Example illustrating falling through cases in a switch statement

by default fall through to the next case after a successful one. Therefore, it is not necessary to add a *break* statement after each case. To enable falling through to the next case you can use the *continue* statement. Also, unlike Java/C, the switch expression can return anything (not only integers). As an example, when executed, the code in Listing 3.20 prints 2 while the code in Listing 3.21 prints 2,3,default.

3.5.6 While Statement

A while statement consists of a condition and a block of statements which are executed as long as the condition is satisfied. For example, in Listing 3.22 the body of the while statement is executed 5 times printing the numbers 0 to 4 to the output console. Inside the body of a *while* statement, the built-in read-only *loopCount* integer variable holds the number

```
var i : Integer = 0;
while (i < 5) {
  -- both lines print the same thing
  i.println();
  (loopCount - 1).println();
  -- increment the counter
  i = i+1;
}
```

Listing 3.22: Example of a while statement

of times the innermost loop has been executed so far (including the current iteration). Right after entering the loop for the first time and before running the first statement in its body, *loopCount* is set to 1, and it is incremented after each following iteration.

3.5.7 For Statement

In EOL, for statements are used to iterate the contents of collections. A for statement defines a typed iterator and an iterated collection as well as a block of statements that is executed for every item in the collection that has a kind-of relationship with the type defined by the iterator. As with the majority of programming languages, modifying a collection while iterating it raises a runtime error. To avoid this situation, users can use the clone() built-in operation of the Collection type discussed in 3.3.2.

Inside the body of a *for* statement two built-in read-only variables are visible: the *loopCount* integer variable (explained in Section 3.5.6) and the *hasMore* boolean variable. *hasMore* is used to determine if there are more items if the collection for which the loop will be executed. For example, in Listing 3.23 the col heterogeneous Sequence is defined that contains two strings (a and b), two integers (1,2) and one real (2.5). The for loop of line 2 only iterates through the items of the collection that are of kind Real and therefore prints 1,2,2.5 to the standard output stream.

```
var col : Sequence = Sequence{"a", 1, 2, 2.5, "b"};
for (r : Real in col) {
 r.print();
 if (hasMore){",".print();}
}
```

Listing 3.23: Example of a for statement

```
for (i in Sequence{1..3}) {
 if (i = 1) {continue;}
 for (j in Sequence{1..4}) {
  if (j = 2) {break;}
  if (j = 3) {breakAll;}
  (i + "," + j).println();
 }
}
```

Listing 3.24: Example of the break breakAll and continue statements

3.5.8 Break, BreakAll and Continue Statements

To exit from for and while loops on demand, EOL provides the break and breakAll statements. The break statement exits the innermost loop while the breakAll statement exits all outer loops as well. On the other hand, to skip a particular loop and proceed with the next one, EOL provides the continue statement. For example, the excerpt of Listing 3.24, prints *2,1 3,1* to the standard output stream.

3.5.9 Throw Statement

EOL provides the throw statement for throwing a value as an EOLUSEREXCEPTION Java exception. This is especially useful when invoking EOL scripts from Java code: by catching and processing the exception, the Java code may be able to automatically handle the problem without requiring user input. Any value can be thrown, as shown in Listing 3.25, where we throw a number and a string.

```
1 throw 42;
2 throw "Error!";
```

Listing 3.25: Example of the throw statement

3.5.10 Transaction Statement

As discussed in Section 2.6, the underlying EMC layer provides support for transactions in models. To utilize this feature EOL provides the transaction statement. A transaction statement (optionally) defines the models that participate in the transaction. If no models are defined, it is assumed that all the models that are accessible from the enclosing program participate. When the statement is executed, a transaction is started on each participating model. If no errors are raised during the execution of the contained statements, any changes made to model elements are committed. On the other hand, if an error is raised the transaction is rolled back and any changes made to the models in the context of the transaction are undone. The user can also use the abort statement to explicitly exit a transaction and roll-back any changes done in its context. In Listing 3.26, an example of using this feature in a simulation problem is illustrated.

In this problem, a system consists of a number of processors. A processor manages some tasks and can fail at any time. The EOL program in Listing 3.26 performs 100 simulation steps, in every one of which 10 random processors from the model (lines 7-11) are marked as failed by setting their *failed* property to true (line 14). Then, the tasks that the failed processors manage are moved to other processors (line 15). Finally the availability of the system in this state is evaluated.

After a simulation step, the state of the model has been drastically changed since processors have failed and tasks have been relocated. To be able to restore the model to its original state after every simulation step, each step is executed in the context of a transaction which is explicitly aborted (line 20) after evaluating the availability of the system. Therefore

```
var system : System.allInstances.first();

for (i in Sequence {1..100}) {

  transaction {

    var failedProcessors : Set;

    while (failedProcessors.size() < 10) {
      failedProcessors.add(system.processors.random());
    }

    for (processor in failedProcessors) {
      processor.failed = true;
      processor.moveTasksElsewhere();
    }

    system.evaluateAvailability();

    abort;
  }

}
```

Listing 3.26: Example of a for statement

after each simulation step the model is restored to its original state for the next step to be executed.

3.6 Extended Properties

Quite often, during a model management operation it is necessary to associate model elements with information that is not supported by the metamodel they conform to. For instance, the EOL program in listing 3.27 calculates the depth of each Tree element in a model that conforms to the Tree metamodel displayed in Figure 3.4.

As the Tree metamodel doesn't support a *depth* property in the Tree metaclass, each Tree has to be associated with its calculated depth (line

```
var depths = new Map;

for (n in Tree.allInstances.select(t|not t.parent.isDefined())) {
 n.setDepth(0);
}

for (n in Tree.allInstances) {
 (n.name + " " + depths.get(n)).println();
}

operation Tree setDepth(depth : Integer) {
 depths.put(self,depth);
 for (c in self.children) {
  c.setDepth(depth + 1);
 }
}
```

Listing 3.27: Calculating and printing the depth of each Tree

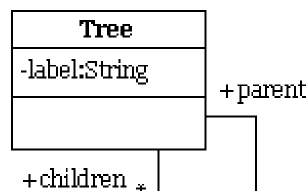

Figure 3.4: The Tree Metamodel

12) using the *depths* map defined in line 1. Another approach would be to extend the Tree metamodel to support the desired *depth* property; however, applying this technique every time an additional property is needed for some model management operation would quickly pollute the metamodel with properties of secondary importance.

To simplify the code required in such cases, EOL provides the concept of *extended properties*. In terms of concrete syntax, an extended property is a normal property, the name of which starts with the tilde character (~). With regards to its execution semantics, the first time the value of an extended property of an object is assigned, the property is created and

```
for (n in Tree.allInstances.select(t|not t.parent.isDefined())) {
  n.setDepth(0);
}

for (n in Tree.allInstances) {
  (n.name + " " + n.~depth).println();
}

operation Tree setDepth(depth : Integer) {
  self.~depth = depth;
  for (c in self.children) {
    c.setDepth(depth + 1);
  }
}
```

Listing 3.28: A simplified version of Listing 3.27 using extended properties

associated with the object. Then, the property can be accessed as a normal property. Listing 3.28 demonstrates using a *depth* extended property to eliminate the need for using the *depths* map in Listing 3.27.

3.7 Context-Independent User Input

A common assumption in model management languages is that model management tasks are only executed in a batch-manner without human intervention. However, as demonstrated in the sequel, it is often useful for the user to provide feedback that can precisely drive the execution of a model management operation.

Model management operations can be executed in a number of runtime environments in each of which a different user-input method is more appropriate. For instance when executed in the context of an IDE (such as Eclipse) visual dialogs are preferable, while when executed in the context of a server or from within an ANT workflow, a command-line user input interface is deemed more suitable. To abstract away from the different runtime environments and enable the user to specify user interaction

statements uniformly and regardless of the runtime context, EOL provides the *IUserInput* interface that can be realized in different ways according to the execution environment and attached to the runtime context via the *IEolContext.setUserInput(IUserInput userInput)* method. The *IUserInput* specifies the methods presented in Table 3.15.

Table 3.15: Operations of IUserInput

Signature	**Description**
inform(message : String)	Displays the specified message to the user
confirm(message : String, [default : Boolean]) : Boolean	Prompts the user to confirm if the condition described by the message holds
prompt(message : String, [default : String]) : String	Prompts the user for a string in response to the message
promptInteger(message : String, [default : Integer]) : Integer	Prompts the user for an Integer
promptReal(message : String, [default : Real]) : Real	Prompts the user for a Real
choose(message : String, options : Sequence, [default : Any]) : Any	Prompts the user to select one of the options
chooseMany(message : String options : Sequence, [default : Sequence]) : Sequence	Prompts the user to select one or more of the options

As displayed above, all the methods of the *IUserInput* interface accept a *default* parameter. The purpose of this parameter is dual. First, it enables the designer of the model management program to prompt the user with

the most likely value as a default choice and secondly it enables a concrete implementation of the interface (*UnattendedExecutionUserInput*) which returns the default values without prompting the user at all and thus, can be used for unattended execution of interactive Epsilon programs. Figures 3.5 and 3.6 demontrate the interfaces through which input is required by the user when the exemplar *System.user.promptInteger('Please enter a number', 1);* statement is executed using an Eclipse-based and a command-line-based *IUserInput* implementation respectively.

User-input facilities have been found to be particularly useful in all model management tasks. Such facilities are essential for performing operations on live models such as model validation and model refactoring

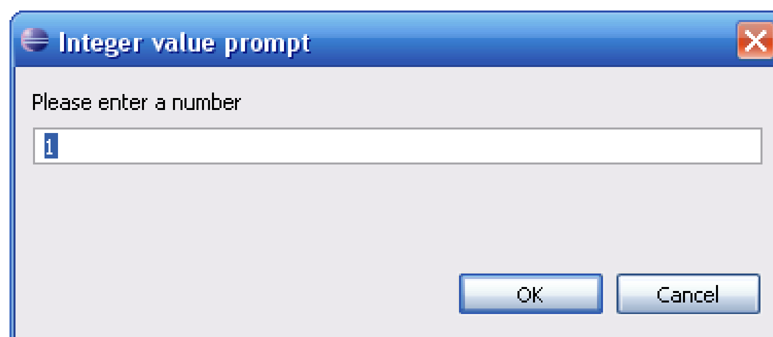

Figure 3.5: Example of an Eclipse-based IUserInput implementation

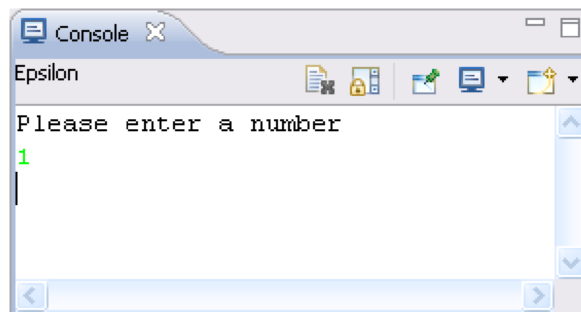

Figure 3.6: Example of a command-line-based IUserInput implementation

but can also be useful in model comparison where marginal matching decisions can be delegated to the user and model transformation where the user can interactively specify the elements that will be transformed into corresponding elements in the target model. Examples of interactive model management operations that make use of the input facilities provided by EOL are demonstrated in Sections 5.6 and 8.5

3.8 Task-Specific Languages

Having discussed EOL in detail, in the following chapters, the following task-specific languages built atop EOL are presented:

- Epsilon Validation Language (EVL)
- Epsilon Transformation Language (ETL)
- Epsilon Generation Language (EGL)
- Epsilon Wizard Language (EWL)
- Epsilon Comparison Language (ECL)
- Epsilon Merging Language (EML)

For each language, the abstract and concrete syntax are presented. To enhance readability, the concrete syntax of each language is presented in an abstract, pseudo-grammar form. Also provided is an informal but detailed discussion, accompanied by concise examples for each feature of interest, of its execution semantics and the runtime structures that are essential to implement those semantics.

Descriptions of the abstract and concrete syntaxes of the task-specific languages are particularly brief since they inherit most of their syntax and features from EOL. As discussed earlier, this contributes to establishing a

platform of uniform languages where each provides a number of unique task-specific constructs but does not otherwise deviate from each other.

To reduce unnecessary repetition, the following sections do not repeat all the features inherited from EOL. However, the reader should bear in mind that by being supersets of EOL, all task-specific languages can exploit the features it provides. For example, by reusing EOL's user-input facilities (discussed in 3.7), it is feasible to specify interactive model to model transformations in ETL. As well, *Native* types can be used to access or update information stored in an external system/tool (e.g. in a database or a remote server) during model validation with EVL or model comparison with ECL.

Following the presentation, in Chapters 4 – 9, of the task-specific languages implemented in Epsilon, Chapter 11 provides a brief overview of the process needed to construct a new language that addresses a task that is not supported by one of the existing languages.