THE epsilon BOOK

Dimitrios Kolovos - Louis Rose - Richard Paige

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The purpose of this book is to provide a complete reference of the languages provided by the Epsilon project (`http://www.eclipse.org/gmt/epsilon`).

1.1 What is Epsilon?

Epsilon, standing for Extensible Platform of Integrated Languages for mOdel maNagement, is - as it's extended name hints - a platform for building consistent and interoperable task-specific languages for model management tasks such as model transformation, code generation, model comparison, merging, refactoring and validation.

Epsilon currently provides the following languages:

- Epsilon Object Language (EOL)
- Epsilon Validation Language (EVL)
- Epsilon Transformation Language (ETL)
- Epsilon Comparison Language (ECL)

- Epsilon Merging Language (EML)
- Epsilon Wizard Language (EWL)
- Epsilon Generation Language (EGL)

For each language Epsilon provides Eclipse-based development tools and an interpreter[1] that can execute programs written in this language. Epsilon also provides a set of ANT tasks for creating workflows of different tasks (e.g. a validation followed by a transformation followed by code generation). The following chapters present the syntax of each language and a few usage examples.

1.2 How To Read This Book

If you are reading this book, there are good chances that you are already interested in using a particular task-specific language provided by Epsilon (e.g. EVL for model validation or EWL for refactoring). In this case, you don't have to need to read about all the languages: you first need to install Epsilon using the instructions provided in `http://www.eclipse.org/gmt/epsilon/download`, and then spend some time reading Chapter 3 that presents the core Epsilon Object Language (EOL), as all languages of the platform extend EOL both syntactically and semantically. Then you can proceed to the chapter that discusses the particular language you are interested in (e.g. Chapter 4 for EVL).

1.3 Questions and Feedback

Our intention is to keep this book a live project that will evolve in parallel with the evolution of Epsilon. Therefore, your feedback on any omis-

[1] The interpreters are not bound in any way with Eclipse and can also be used in standalone Java applications.

sions, errors or outdated content is critical and much appreciated (and also you will win a place for your name in the Acknowledgements section of the book). Please send your feedback to the Epsilon newsgroup (see `http://www.eclipse.org/gmt/epsilon/newsgroup/` for detailed instructions).

1.4 Additional Resources

As mentioned above, information about Epsilon and examples are available in many different places. If you can't find what you are looking for in this book there are a few other places where you may try.

1.4.1 Epsilon Eclipse GMT

Epsilon is a component of the Eclipse Modelling GMT project and it is hosted in `http://www.eclipse.org/gmt/epsilon`. In the documentation section `http://www.eclipse.org/gmt/epsilon/doc` there is documentation about several features of Epsilon, and the Cinema `http://www.eclipse.org/gmt/epsilon/cinema` contains a number of Flash screencasts that demonstrate different languages and tools of Epsilon in action.

1.4.2 EpsilonLabs

EpsilonLabs is a satellite project of Epsilon that hosts experimental applications/extensions of Epsilon or other content that cannot be shared under Eclipse.org due to licensing issues (e.g. incompatibility with EPL). EpsilonLabs is located in `http://epsilonlabs.sf.net`

1.4.3 Epsilon Weblog

In November 2007 we started a blog where we've been reporting new applications and extensions of Epsilon. The blog provides the latest information on the project and is located in `http://epsilonblog.wordpress.com`

1.4.4 Twitter

To keep in touch with the latest news on Epsilon, please follow @epsilonews on Twitter.

Chapter 2

The Epsilon Model Connectivity Layer (EMC)

In this section the design of the Epsilon Model Connectivity layer. A graphical overview of the design is displayed in Figure 2.1.

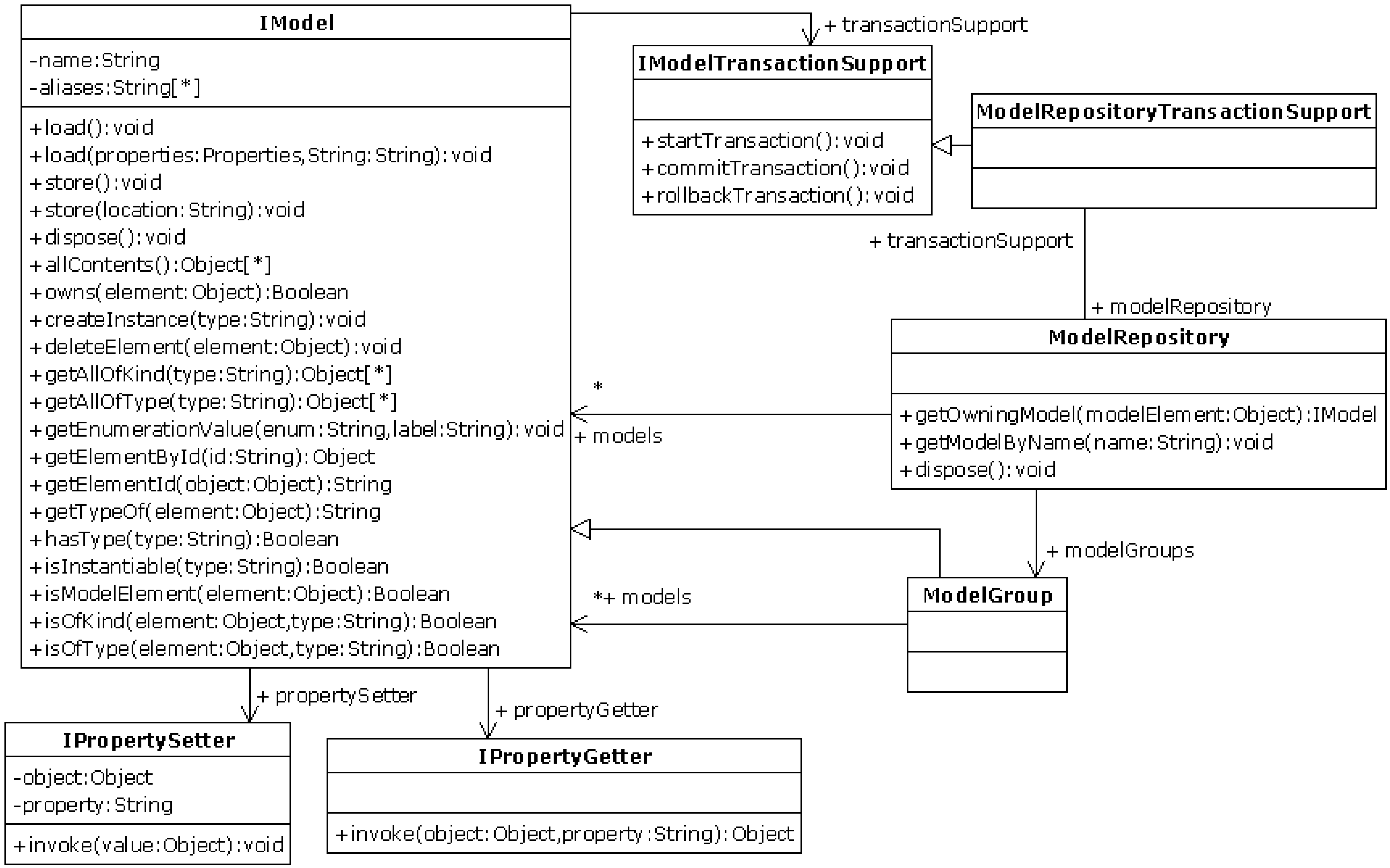

Figure 2.1: Overview of the Epsilon Model Connectivity layer

To abstract away from diverse model representations and APIs provided by different modelling technologies, EMC defines the *IModel* interface. *IModel* provides a number of methods that enable querying and modifying the model elements it contains at a higher level of abstraction. To enable languages and tools that build atop EMC to manage multiple models simultaneously, the *ModelRepository* class acts as a container that offers façade services. The following sections discuss these two core concepts in detail.

2.1 The IModel interface

Each model specifies a name which must be unique in the context of the model repository in which it is contained. Also, it defines a number of aliases; that is non-unique alternate names; via which it can be accessed. The interface also defines the following services.

2.2 Loading and Persistence

The *load()* and *load(properties : Properties)* methods enable extenders to specify in a uniform way how a model is loaded into memory from the physical location in which it resides. Similarly, the *store()* and *store(location : String)* methods are used to define how the model can be persisted from memory to a permanent storage location.

2.3 Type-related Services

The majority of metamodelling architectures support inheritance between meta-classes and therefore two types of type-conformance relationships generally appear between model elements and types. The *type-of* relationship appears when a model element is an instance of the type and the *kind-of* relationhip appears when the model element is an instance of the type or any of its sub-types. Under this definition, the *getAllOfType(type : String)* and the *getAllOfKind(type : String)* operations return all the elements in the model that have a type-of and a kind-of relationship with the type in question respectively.

Similarly, the *isTypeOf(element : Object, type : String)* and *isKindOf(element : Object, type : String)* return whether the element in question has a type-of or a kind-of relationship with the type respectively. The *getTypeOf(element : Object)* method returns the fully-qualified name of the type an element conforms to.

The *hasType(type : String)* method returns true if the model supports a type with the specified name. To support technologies that enable users to define abstract (non-instantiable) types, the *isInstantiable(type : String)* method returns if instances of the type can be created.

2.4 Ownership

The *allContents()* method returns all the elements that the model contains and the *owns(element : Object)* method returns true if the element under question belongs to the model.

2.5 Creation, Deletion and Modifications

Model elements are created and deleted using the *createInstance(type : String)* and *deleteElement(element : Object)* methods respectively.

To retrieve and set the values of properties of its model elements, *IModel* uses its associated *propertyGetter* (*IPropertyGetter*) and *propertySetter* (*IPropertySetter*) respectively. Technology-specific implementations of those two interfaces are responsible for accessing and modifying the value of a property of a model element through their *invoke(element:Object,property:String)* and *invoke(value:Object)* respectively.

2.6 The IModelTransactionSupport interface

In its *transactionSupport* property, a model can optionally (if the target modelling technology supports transactions) specify an instance of an implementation of the *IModelTransactionSupport* interface. The interface provides transaction-related services for the specific modelling technology. The interface provides the *startTransaction()*, *commitTransaction()* and *rollbackTransaction()* methods that start a new transaction, commit and roll back the current transaction respectively.

2.7 The ModelRepository class

A model repository acts as a container for a set of models that need to be managed in the context of a task or a set of tasks. Apart from a reference to the models it contains, *ModelRepository* also provides the following façade functionality.

The *getOwningModel(element:Object)* method returns the model that owns a particular element. The *transactionSupport* property specifies an instance of the *ModelRepositoryTransactionSupport* class which is responsible for aggregate management of transactions by delegating calls to its *startTransaction()*, *commitTransaction()* and *abortTransaction()* methods, to the respective methods of instances of *IModelTransactionSupport* associated with models contained in the repository.

2.8 The ModelGroup class

A *ModelGroup* is a group of models that have a common alias. *ModelGroups* are calculated dynamically by the model repository based on common model aliases. That is, if two or more models share a common alias, the repository forms a new model group. Since *ModelGroup* implements the *IModel* interface, clients can use all the methods of *IModel* to perform aggregate operations on multiple models, such as collecting the contents of more than one models. An exception to that is the *createInstance(type : String)* method which cannot be defined for a group of models as it cannot be determined in which model of the group the newly created element should belong.

2.9 Assumptions about the underlying modelling technologies

The discussion provided above has demonstrated that EMC makes only minimal assumptions about the structure and the organization of the underlying modelling technologies. Thus, it intentionally refrains from defining classes for concepts such as *model element*, *type* and *metamodel*. By contrast, it employs a lightweight approach that uses primitive strings for type names and objects of the target implementation platforms as model elements. There are two reasons for this decision.

The primary reason is that by minimizing the assumptions about the underlying technologies EMC becomes more resistant to future changes of the implementations of the current technologies and can also embrace new technologies without changes.

Another reason is that if a heavy-weight approach was used, extending the platform with support for a new modelling technology would involve providing wrapping objects for the native objects which represent model elements and types in the specific modelling technology. Experiments in the early phases of the design of EMC demonstrated that such a heavy-weight approach significantly increases the amount of memory required to represent the models in memory, degrades performance and provides little benefits in reward[1].

[1]Recent developments in the context of the ATL transformation language have also demonstrated significant performance gains delivered by using native model element representations. Relevant benchmarks can be found `http://wiki.eclipse.org/ATL_VM_Testing`

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*. Each module can also import other modules, using *import* statements, and access the operations they define.

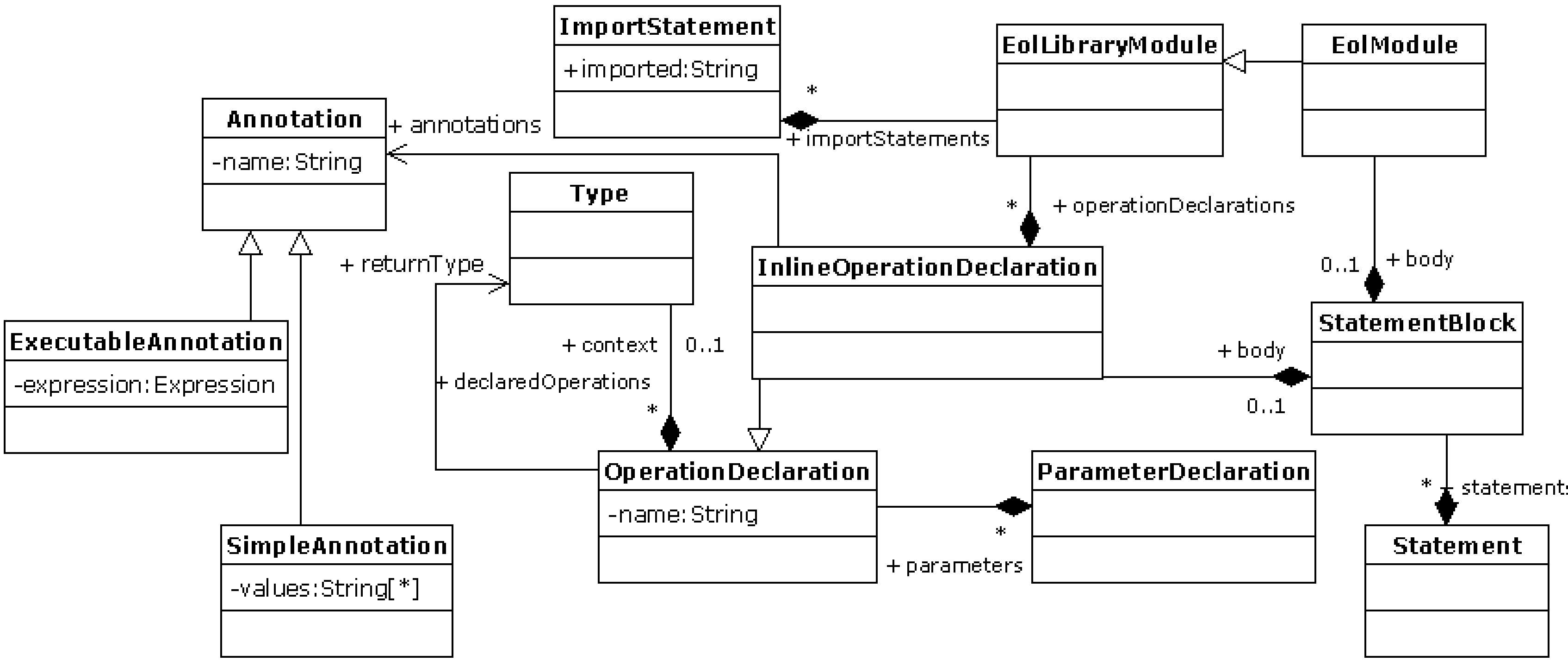

Figure 3.1: EOL Module Structure

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. Therefore, in line 3 they can be invoked using the *1.add1().add2()* expression. 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.

Listing 3.1: Exemplar context-defining EOL operations

```
1 1.add1().add2().println();
2
3 operation Integer add1() : Integer {
4   return self + 1;
5 }
6
7 operation Integer add2() : Integer {
8   return self + 2;
9 }
```

Listing 3.2: Exemplar EOL context-less EOL operations

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

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

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

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.

Listing 3.3: Demonstration of polymorphism in EOL

```
1 "1".test();
2 1.test();
3
4 operation String test() {
5  (self + " is a string").println();
6 }
7
8 operation Integer test() {
9  (self + "is an integer").println();
10 }
```

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.

Listing 3.4: Concrete syntax of simple annotations

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

Listing 3.5: Concrete syntax of executable annotations

```
$name expression
```

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 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 3.6.

Listing 3.6: Example of pre- and post-conditions in an EOL operation

```
1 1.add(2);
2 1.add(-1);
3
4 $pre i > 0
5 $post _result > self
6 operation Integer add(i : Integer) : Integer {
7   return self + i;
8 }
```

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

Listing 3.7: Calculating the Fibonacci number using a cached operation

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

```

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

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.

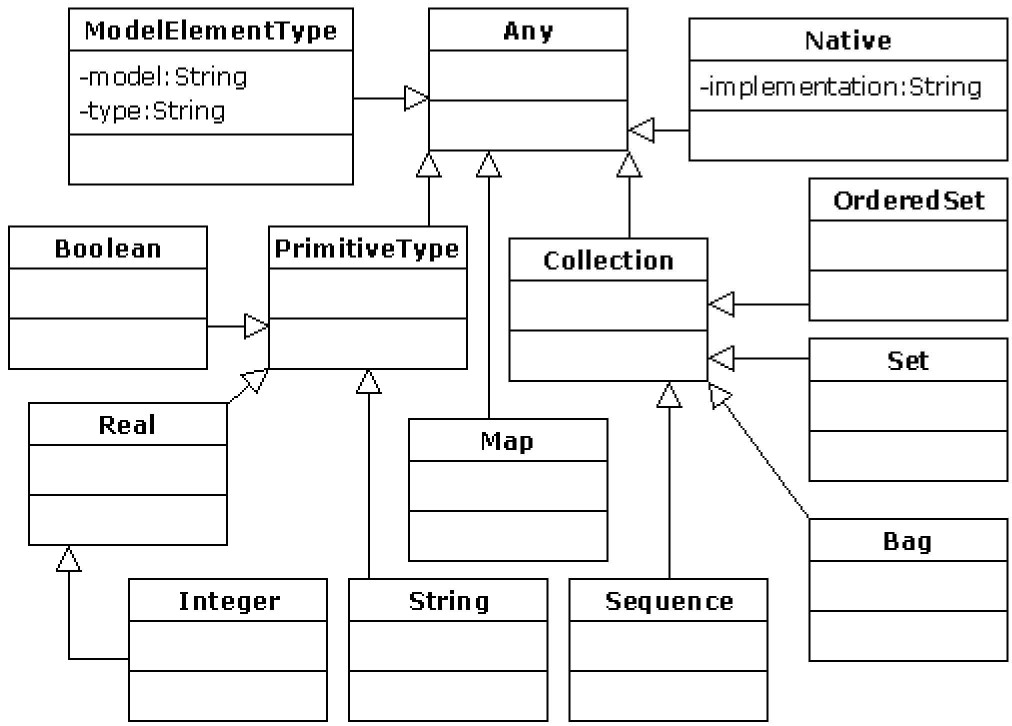

Figure 3.2: Overview of the type system of EOL

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*
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
hline 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

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
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 collection 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
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
exists(iterator : Type | condition) : Boolean	Returns true if there exists 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

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 3.8 the EOL excerpt creates a Java window (Swing JFrame) and uses its methods to change its title and dimensions and make it visible.

Listing 3.8: Demonstration of NativeType in EOL

```
var frame = new Native("javax.swing.JFrame");
```

```
frame.title = "Opened with EOL";
frame.setBounds(100,100,300,200);
frame.visible = true;
```

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 sub-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 3.9 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.

Listing 3.9: Demonstration of the concrete syntax for accessing model element types

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

3.4 Expressions

3.4.1 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).

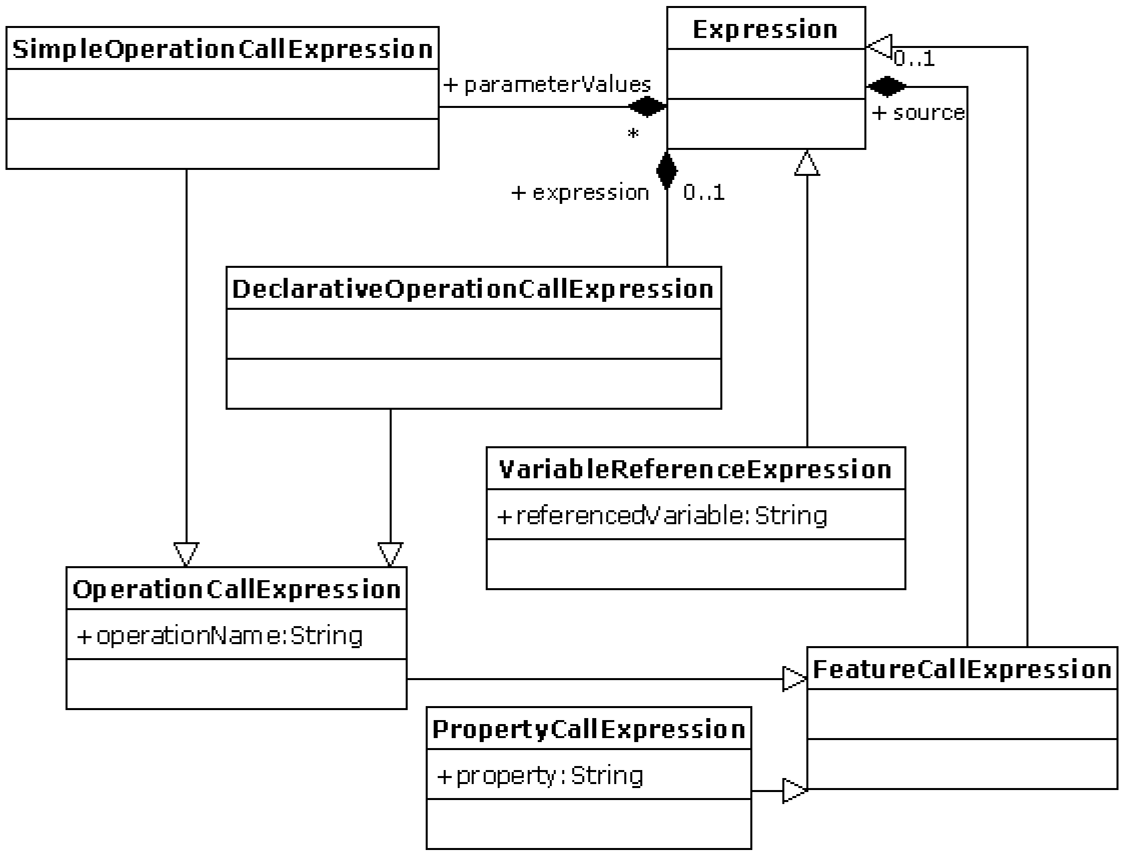

Figure 3.3: Overview of the feature navigation EOL expressions

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 3.10, 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).

Listing 3.10: Invoking operations using EOL

```
1 "Something".println();
2 
3 operation Any println() : Any {
4   ("Printing : " + self)->println();
5 }
```

3.4.2 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

3.4.3 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.

3.4.4 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

3.5 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).

Table 3.14: Default values of primitive types

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

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 languages 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 3.11, 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.

Listing 3.11: Example illustrating declaration and use of variables

```
1 var i : Integer = 5;
2 var c : new Uml!Class;
3 //i = "somevalue";
4 if (c.isDefined()) {
5   var i : String;
6   i = "somevalue";
7 }
8 i = 3;
```

3.5.2 Assignment Statement

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

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 3.12, 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 3.13, 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.

Listing 3.12: Assigning the value of a variable by reference

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

Listing 3.13: Assigning the value of a variable by value

```
1 var a : String;
2 var b = a;
3 a = "Customer";
4 b.println();
```

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 3.14 if x belongs to an EMF-based model or using the Java code of Listing 3.15 if it belongs to an MDR-based model.

Listing 3.14: Java code that assigns the value of a property of a model element that belongs to an EMF-based model

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

Listing 3.15: Java code that assigns the value of a property of a model element that belongs to an MDR-based model

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

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 3.16 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.

Listing 3.16: A simple model-to-model transformation demonstrating the special assignment statement

```
1 rule Class2Table
2  transform c : OO!Class
3  to t : DB!Table {
4
5  t.name = c.name;
6 }
7
8 rule Attribute2Column
9  transform a : OO!Attribute
10  to c : DB!Column {
11
12  c.name = a.name;
13  --c.owningTable = c.owningClass;
14  c.owningTable ::= c.owningClass;
15 }
```

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 3.17, 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.

Listing 3.17: Example illustrating an if statement

```
1 if (a > 0) {
2  "A is greater than 0".println();
3 }
4 else { "A is less equal than 0".println(); }
```

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 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 3.18 prints 2 while the code in Listing 3.19 prints 2,3,default.

Listing 3.18: Example illustrating a switch statement

```
var i = "2";

switch (i) {
 case "1" : "1".println();
 case "2" : "2".println();
 case "3" : "3".println();
 case default : "default".println();
}
```

Listing 3.19: Example illustrating falling through cases in a switch statement

```
var i = "2";

```

```
switch (i) {
  case "1" : "1".println();
  case "2" : "2".println(); continue;
  case "3" : "3".println();
  case default : "default".println();
}
```

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 3.20 the body of the while statement is executed 5 times printing the numbers 0 to 4 to the output console.

Listing 3.20: Example of a while statement

```
var i : Integer = 0;
while (i < 5) {
  i.println();
  i = i+1;
}
```

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 statements two built-in read-only variables are visible: the hasMore boolean variable is used to determine if there are more items if the collection for which the loop will be executed and the loopCount integer variable holds the number of times the innermost loop has been executed so far (including the current iteration). For example, in Listing 3.21 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.

Listing 3.21: Example of a for statement

```
1 var col : Sequence = Sequence{"a", 1, 2, 2.5, "b"};
2 for (r : Real in col) {
3  r.print();
4  if (hasMore){",".print();}
5 }
```

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 3.22, prints *2,1 3,1* to the standard output stream.

Listing 3.22: Example of the break breakAll and continue statements

```
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();
 }
}
```

3.5.9 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 3.23, 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 3.23 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 after each simulation step the model is restored to its original state for the next step to be executed.

Listing 3.23: Example of a for statement

```
1 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;
 }

}
```

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 3.24 calculates the depth of each Tree element in a model that conforms to the Tree metamodel displayed in Figure 3.4.

Listing 3.24: Calculating and printing the depth of each Tree

```
var depths = new Map;

```

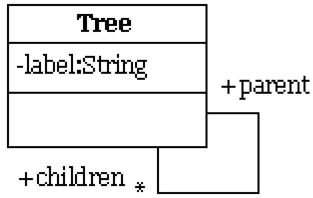

Figure 3.4: The Tree Metamodel

```
3  for (n in Tree.allInstances.select(t|not t.parent.isDefined())) {
4    n.setDepth(0);
5  }
6
7  for (n in Tree.allInstances) {
8    (n.name + " " + depths.get(n)).println();
9  }
10
11 operation Tree setDepth(depth : Integer) {
12   depths.put(self,depth);
13   for (c in self.children) {
14     c.setDepth(depth + 1);
15   }
16 }
```

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 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 associated with the object. Then, the property can be accessed as a normal property. Listing 3.25 demonstrates using a *depth* extended property to eliminate the need for using the *depths* map in Listing 3.24.

Listing 3.25: A simplified version of Listing 3.24 using extended properties

```
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);
  }
}
```

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 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 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

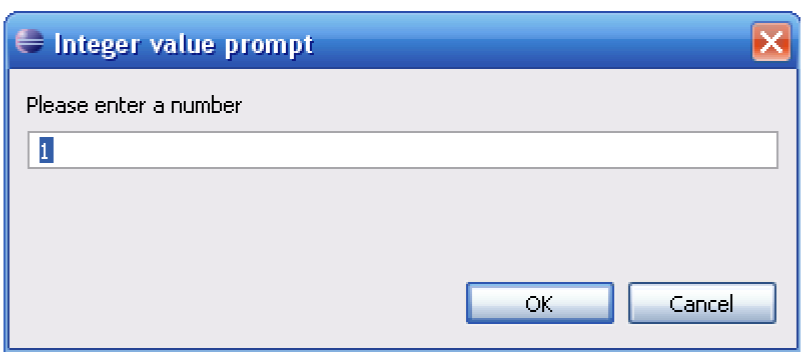

Figure 3.5: Example of an Eclipse-based IUserInput implementation

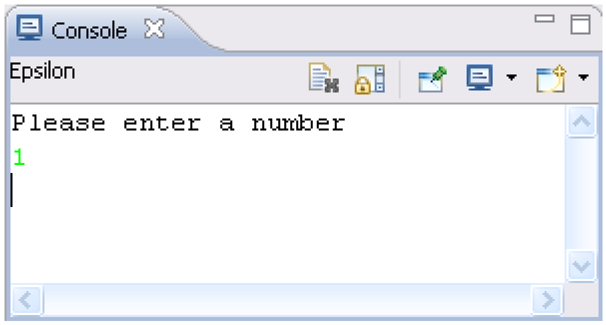

Figure 3.6: Example of a command-line-based IUserInput implementation

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 10 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.

Chapter 4

The Epsilon Validation Language (EVL)

The aim of EVL is to contribute model validation capabilities to Epsilon. More specifically, EVL can be used to specify and evaluate constraints on models of arbitrary metamodels and modelling technologies. This section provides a discussion on the motivation for implementing EVL, its abstract and concrete syntax as well as its execution semantics. It also provides two examples using the language to verify inter-model and intra-model consistency.

4.1 Motivation

Although many approaches have been proposed to enable automated model validation, the Object Constraint Language (OCL) [1] is the de facto standard for capturing constraints in modelling languages specified using object-oriented metamodelling technologies. While its powerful syntax enables users to specify meaningful and concise constraints, its purely declarative and side-effect free nature introduces a number of limitations in the context of a contemporary model management environment. In this section, the shortcomings of OCL that have motivated the design of EVL are discussed in detail.

In OCL, structural constraints are captured in the form of *invariants*. Each invariant is defined in the context of a meta-class of the metamodel and specifies a name and a body. The body is an OCL expression that must evaluate to a *Boolean* result, indicating whether an instance of the meta-class satisfies the invariant or not. Execution-wise, the body of each invariant is evaluated for each instance of the meta-class and the results are stored in a set of <Element, Invariant, Boolean> triplets. Each triplet captures the *Boolean* result of the evaluation of an *Invariant* on a qualified *Element*. An exemplar OCL invariant for UML 1.4, requiring that abstract operations only belong to abstract classes, is shown in Listing 4.1.

Listing 4.1: OCL constraint on UML operations

```
context Operation
 inv AbstractOperationInAbstractClassOnly :
  self.isAbstract implies self.owner.isAbstract
```

While in its current version OCL enables users to capture particularly complex invariants, it also demonstrates a number of shortcomings, as follows.

4.1.1 Limited user feedback

OCL does not support specifying meaningful messages that can be reported to the user in case an invariant is not satisfied for certain elements. Therefore, feedback to the user is limited to the name of the invariant and the instance(s) for which it failed. Weak support for proper feedback messages implies that the end users must be familiar with OCL so that they can comprehend the meaning of the failed invariant and locate the exact reason for the failure. This is a significant shortcoming as in practice only a very small number of end users are familiar with OCL.

4.1.2 No support for warnings/critiques

Contemporary software development environments typically produce two types of feedback when checking artefacts for consistency and correctness: errors and warnings. Errors indicate critical deficiencies that contradict basic principles and invalidate the developed artefacts. By contrast, warnings (or critiques) indicate non-critical issues that should nevertheless be addressed by the user. To enable users to address warnings in a priority-based manner, they are typically categorized into three levels of importance: High, Medium and Low (although other classifications are also possible).

Nevertheless, in OCL there is no such distinction between errors and warnings and consequently all reported issues are considered to be errors. This adds an additional burden to identifying and prioritizing issues of major importance, particularly within an extensive set of unsatisfied invariants in complex models.

4.1.3 No support for dependent constraints

Each OCL invariant is a self-contained unit that does not depend on other invariants. There are cases where this design decision is particularly restrictive. For instance consider the invariants *I1* and *I2* displayed in Listing 4.2. Both I1 and I2 are applicable on UML classes with *I1* requiring that: *the name of a class must not be empty* and *I2* requiring that: *the name of a class must start with a capital letter*. In the case of those two invariants, if *I1* is not satisfied for a particular UML class, evaluating *I2* on that class would be meaningless. In fact it would be worse than meaningless since it would consume time to evaluate and would also produce an extraneous error message to the user. In practice, to avoid the extraneous message, *I2* needs to replicate the body of *I1* using an *if* expression (lines 2 and 5).

Listing 4.2: Conceptually related OCL constraints

```
context Class
 inv I1 : self.name.size() > 0

 inv I2 :
  if self.name.size > 0 then
   self.name.substring(0,1) =
   self.name.substring(0,1).toUpper()
  else
   true
  endif
```

4.1.4 Limited flexibility in context definition

As already discussed, in OCL invariants are defined in the context of meta-classes. While this achieves a reasonable partitioning of the model element space, there are cases where more fine-grained partitioning is required. For instance, consider the following scenario. Let $IA_{1..N}$, $IB_{1..M}$ be invariants applying to classes that are stereotyped as `<<A>>` and `<<B>>` respectively. Since OCL only supports partitioning the model element space using meta-classes, all $IA_{1..N}$, $IB_{1..M}$ must appear under the same context (i.e. *Class*). Moreover, each invariant must explicitly define that it addresses the one or the other conceptual sub-partition. Therefore, each of $IA_{1..N}$ must limit its scope initially (using the $self.isA$ expression) and then express the real body. In our example the simplest way to achieve this would be by combining a scope-limiting expression with the real invariant body using the *implies* clause as demonstrated in Listing 4.3.

Listing 4.3: Demonstration of OCL constraints with duplication

```
context Class
 inv I1 : self.isA implies <real-invariant-body>
 inv I2 : self.isA implies <real-invariant-body>
 ...
```

```
inv IN : self.isA implies <real-invariant-body>

def isA :
  let isA : Boolean =
  self.stereotype->exists(s|s.name = 'A')
```

Furthermore, if the *real* body of the invariant needs to assume that self is stereotyped with `<<A>>`, this technique is not applicable because OCL does not support lazy evaluation of Boolean clauses [1] and therefore although the first part of the expression (`self.isA`) may fail for some instances, the second part will still be evaluated thus producing runtime errors. In this case, an *if* expression must be used, further complicating the specified invariants.

4.1.5 No support for repairing inconsistencies

While OCL can be used for detecting inconsistencies, it provides no means for repairing them. The reason is that OCL has been designed as a side-effect free language and therefore lacks constructs for modifying models. Nevertheless, there are many cases where inconsistencies are trivial to resolve and users can benefit from semi-automatic repairing facilities.

This need has been long recognized in the related field of code development tools (e.g. Eclipse, Microsoft Visual Studio, NetBeans). In such tools, errors are not only identified but also context-aware actions are proposed to the user for automatically repairing them. This feature significantly increases the usability of such tools and consequently enhances users' productivity.

4.1.6 No support for inter-model constraints

OCL expressions (and therefore OCL constraints) can only be evaluated in the context of a single model at a time. Consequently, OCL cannot be used to express constraints that span across different models. In the context of a large-scale model driven engineering process that involves many different models (that potentially conform to different modelling languages) this limitation is particularly severe.

Following this discussion on the shortcomings of OCL for capturing structural constraints in modelling languages, the following sections present the abstract and concrete syntax of EVL as well as their execution semantics, and explain how they address the aforementioned limitations.

4.2 Abstract Syntax

In EVL, validation specifications are organized in modules (*EvlModule*). As illustrated in Figure 4.1, *EvlModule* extends *EolLibraryModule* which means that it can contain user-defined operations and import other EOL library modules and EVL modules. Apart from operations, an EVL module also contains a set of invariants grouped by the context they apply to, and a number of *pre* and *post* blocks.

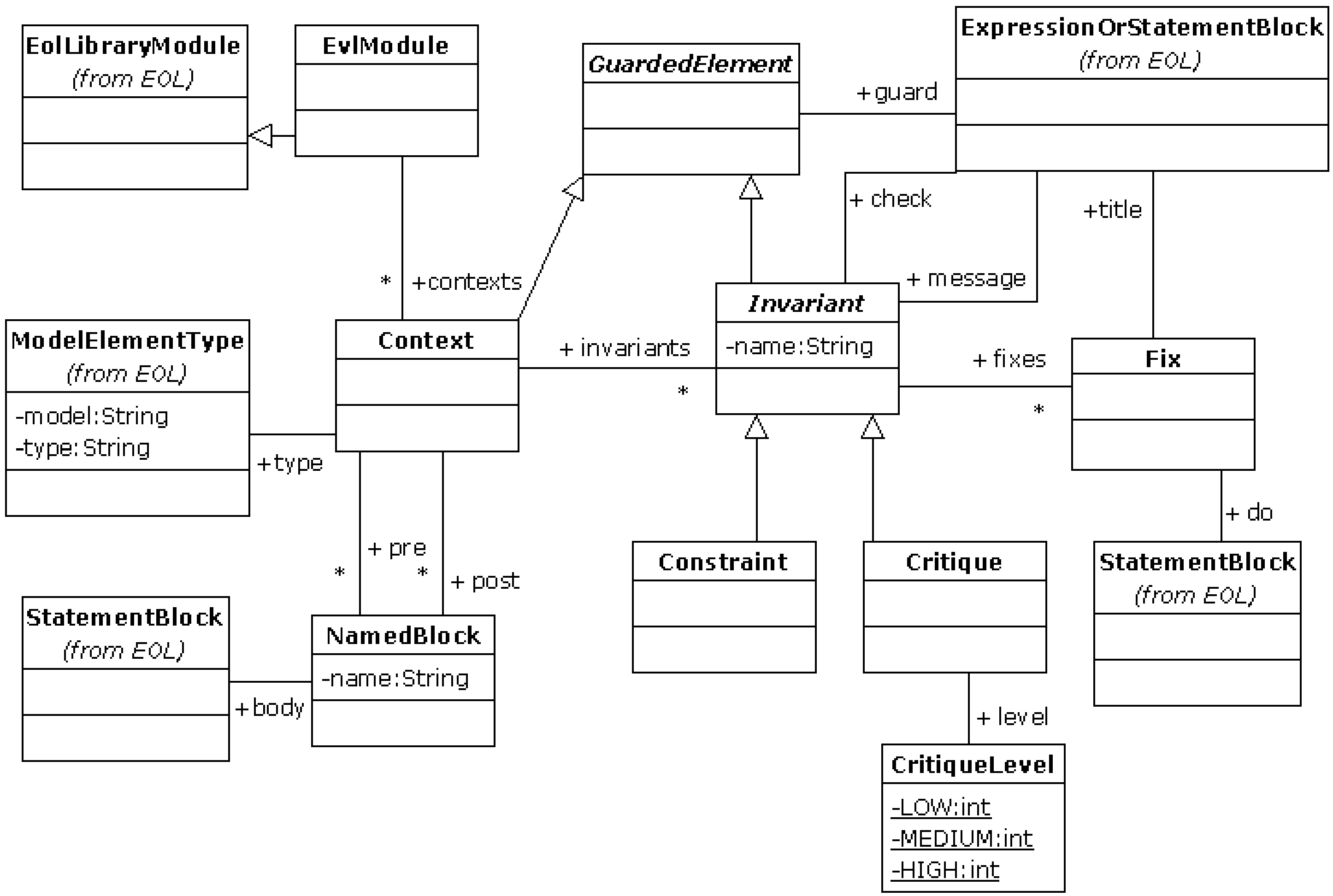

Figure 4.1: Abstract Syntax of EVL

Context A context specifies the kind of instances on which the contained invariants will be evaluated. Each context can optionally define a guard which limits its applicability to a narrower subset of instances of its specified type. Thus, if the guard fails for a specific instance of the type, none of its contained invariants are evaluated.

Invariant As with OCL, each EVL invariant defines a *name* and a body (*check*). However, it can optionally also define a *guard* (defined in its abstract *GuardedElement* supertype) which further limits its applicability to a subset of the instances of the type defined by the embracing *context*. To achieve the requirement for detailed user feedback (Section 4.1.1), each invariant can optionally define a *message* as an *ExpressionOrStatementBlock* that should return a String providing a description of the reason(s) for which the constraint has failed on a particular element. To support semi-automatically fixing of elements on which invariants have failed (Section 4.1.5), an invariant can optionally define a number of *fixes*. Finally, as displayed in Figure 4.1, *Invariant* is an abstract class that is used as a superclass for the specific types *Constraint* and *Critique*. This is to address the issue of separation of errors and warnings/critiques (Section 4.1.2).

Guard Guards are used to limit the applicability of invariants (Section 4.1.4). This can be achieved at two levels. At the *Context* level it limits the applicability of all invariants of the context and at the *Invariant* level it limits the applicability of a specific invariant.

Fix A fix defines a title using an *ExpressionOrStatementBlock* instead of a static String to allow users to specify context-aware titles (e.g. *Rename class customer to Customer* instead of a generic *Convert first letter to upper-case*). Moreover, the *do* part is a statement block where the fixing functionality can be defined using EOL. The developer is responsible for ensuring that the actions contained in the *fix* actually repair the identified inconsistency.

Constraint *Constraints* in EVL are used to capture critical errors that invalidate the model. As discussed above, *Constraint* is a sub-class of *Invariant* and therefore inherits all its features.

Critique Unlike *Constraints*, *Critiques* are used to capture non-critical situations that do not invalidate the model, but should nevertheless be addressed by the user to enhance the quality of the model. This separation addresses the issue raised in Section 4.1.2. Moreover, to enable users to define different levels of importance in critiques, the *CritiqueLevel* enumeration supports a 3-level classification. Fixed-level classification has been preferred in EVL over infinite level classification (e.g. using Integer levels) since it is more common in development tools and easier to visualize.

Pre and Post An EVL module can define a number of named *pre* and a *post* blocks that contain EOL statements which are executed before and after evaluating the invariants respectively.

4.3 Concrete Syntax

Listings 4.4, 4.5 and 4.6 demonstrate the concrete sytnax of the *context*, *invariant* and *fix* abstract syntax constructs discussed above.

Listing 4.4: Concrete Syntax of an EVL context

```

(context) <name> {

 (invariant)*

}
```

Listing 4.5: Concrete Syntax of an EVL invariant

```

(@lazy)?
(constraint|critique) <name> {

  (guard (:expression)|({statementBlock}))?

  (check (:expression)|({statementBlock}))?

  (message (:expression)|({statementBlock}))?

  (fix)*

}
```

Listing 4.6: Concrete Syntax of an EVL fix

```
fix {

  (guard (:expression)|({statementBlock}))?

  (title (:expression)|({statementBlock}))?

 do {
   statementBlock
  }

}
```

4.4 Execution Semantics

Having discussed the abstract and concrete syntaxes of EVL, this section provides an informal discussion of the execution semantics of the language. The execution of an EVL module is separated into four phases:

Phase 1 Before any invariant is evaluated, the *pre* sections of the module are executed in the order in which they have been specified.

Phase 2 For each *context*, the instances of the meta-class it defines are collected. For each instance, the *guard* of the *context* is evaluated. If the *guard* is satisfied, then for each non-lazy invariant contained in the context the invariant's *guard* is also evaluated. If the *guard* of the invariant is satisfied, the *body* of the invariant is evaluated. In case the *body* evaluates to *false*, the *message* part of the rule is evaluated and the produced message is added along with the instance, the invariant and the available *fixes* to the *ValidationTrace*.

The execution order of an EVL module follows a top-down depth-first scheme that respects the order in which the *contexts* and *ivnariants* appear in the module. However, the execution order can change in case one of the *satisfies*, *satisfiesOne*, *satisfiesAll* built-in operations, discussed in detail in the sequel, are called.

Phase 3 In this phase, the validation trace is examined for unsatisfied constraints and the user is presented with the message each one has produced. The user can then select one or more of the available *fixes* to be executed. Execution of *fixes* is performed in a transactional manner using the respective facilities provided by the model connectivity framework, as discussed in Section 2.6. This is to prevent runtime errors raised during the execution of a *fix* from compromising the validated model by leaving it in an inconsistent state.

Phase 4 When the user has performed all the necessary *fixes* or chooses to end Phase 3 explicitly, the *post* section of the module is executed. There, the user can perform tasks such as serializing the validation trace or producing a summary of the validation process results.

4.4.1 Capturing Dependencies Between Invariants

As discussed in Section 4.1.3, it is often the case that invariants conceptually depend on each other. To allow users capture such dependencies, EVL provides the *satisfies(invariant : String) : Boolean*, *satisfiesAll(invariants : Sequence(String)) : Boolean* and *satisfiesOne(invariants : Sequence(String)) : Boolean* built-in operations. Using these operations, an invariant can specify in its *guard* other invariants which need to be satisfied for it to be meaningful to evaluate.

When one of these operations is invoked, if the required *invariants* (either lazy or non-lazy) have been evaluated for the instances on which the operation is invoked, the engine will return their cached results; otherwise it will evaluate them and return their results.

4.5 Intra-Model Consistency Checking Example

This section presents a case study comparing EVL and OCL in the context of a common scenario. The purpose of the case study is to present readers with the concrete syntax of the language and demonstrate the benefits delivered by the additional constructs it facilitates.

4.5.1 Scenario: The Singleton Pattern

The *singleton* pattern is a widely used object oriented pattern. A *singleton* is a class for which *exactly one instance is allowed* [2]. In UML, a singleton is typically represented as a class which is stereotyped with a `<<singleton>>` stereotype and which also defines a static operation named *getInstance()* that returns the unique instance.

To ensure that all singletons have been modelled correctly in a UML model one needs to evaluate the following invariants on all classes that are stereotyped with the `<<singleton>>` stereotype:

- DefinesGetInstance : Each stereotyped class must define a getInstance() method
- GetInstanceIsStatic : The getInstance() method must be static
- GetInstanceReturnsSame : The return type of the getInstance() method must be the class itself

Obviously, invariants *GetInstanceIsStatic* and *GetInstanceReturnsSame* depend on *DefinesGetInstance* because if the singleton does not define a *getInstance()* operation, checking for the operation's scope and return type is meaningless. Moreover, in case an invariant fails, there are corrective actions (fixes) that users may want to perform semi-automatically: e.g. for *DefinesGetInstance*, such an action would be to add the missing *getInstance()* operation, for *GetInstanceIsStatic* to change it to static and for *GetInstanceRetunrsSame* to set the return type to the class itself. In the following sections OCL and EVL are used to express the three constraints and then the two solutions are compared.

4.5.2 Using OCL to Express the Invariants

Listing 4.7 shows the aforementioned invariants implemented in OCL.

Listing 4.7: OCL Module for Validating Singletons

```
package Foundation::Core

  context Class

  def isSingleton :
    let isSingleton : Boolean =
    self.stereotype->exists(s|s.name = 'singleton')

  def getInstanceOperation :
    let getInstanceOperation : Operation =
```

```
    self.feature->select(f|f.oclIsTypeOf(Operation)
    and f.name = 'getInstance')->first().oclAsType(Operation)

  inv DefinesGetInstanceOperation :
    if isSingleton
      then getInstanceOperation.isDefined
      else true
    endif

  inv GetInstanceOperationIsStatic :
    if isSingleton then
      if getInstanceOperation.isDefined
        then getInstanceOperation.ownerScope = #classifier
        else false
      endif
    else
      true
    endif

  inv GetOperationReturnsSame :
    if isSingleton then
      if getInstanceOperation.isDefined then
        if getInstanceOperation.returnParameter.isDefined
          then getInstanceOperation.returnParameter.type = self
          else false
        endif
      else
        false
      endif
    else
      true
    endif

  context Operation

  def returnParameter :
    let returnParameter : Parameter =
```

```
48     self.parameter->select(p|p.kind = #return)->first()
49
50 endpackage
```

By examining the OCL solution it can be observed that all invariants first check that the class is a singleton (lines 15, 21 and 31) by using the *isSingleton* derived property defined in line 5. If the isSingleton returns *false*, the invariants return *true* since returning false would cause them to fail for all non-singleton classes. This reveals an additional shortcoming of OCL: if a constraint returns *true* it may mean two different things: either that the instance satisfies the constraint or that the constraint is not applicable to the instance at all. In our view, this overloading reduces understandability.

By further studying the solution of Listing 4.7 it can be noticed that dependency between constraints is captured artificially using nested *if* expressions. For instance, both *GetInstanceIsStatic* and *GetInstanceRetunrsSame* contain an *if* expression in lines 22 and 32 respectively, requiring that they recalculate the value of the *getInstanceOperation* defined in line 9, where they actually recalculate the result of the *DefinesGetInstanceOperation* invariant. As discussed in Section 4.1.3, this happens because OCL lacks constructs for capturing dependencies in a structured manner.

4.5.3 Using EVL to Express the Invariants

Listing 4.8 provides a solution for this problem expressed in EVL.

Listing 4.8: EVL Module for Validating Singletons

```
context Singleton typeOf Class {

 guard : self.stereotype->exists(s|s.name = "singleton")

 constraint DefinesGetInstance {
  check : self.getGetInstanceOperation().isDefined()
  message : "Singleton " + self.name +
   " must define a getInstance() operation"
```

```
  fix {
    title : "Add a getInstance() operation to " + self.name
    do {
      // Create the getInstance operation
      var op : new Operation;
      op.name = "getInstance";
      op.owner = self;
      op.ownerScope = ScopeKind#sk_classifier;

      // Create the return parameter
      var returnParameter : new Parameter;
      returnParameter.type = self;
      op.parameter = Sequence{returnParameter};
      returnParameter.kind = ParameterDirectionKind#pdk_return;
    }
  }
}

constraint GetInstanceIsStatic {
  guard : self.satisfies("DefinesGetInstance")
  check : self.getGetInstanceOperation().ownerScope =
        ScopeKind#sk_classifier
  message : " The getInstance() operation of singleton "
          + self.name + " must be static"

  fix {
    title : "Change to static"
    do {
      self.getGetInstanceOperation.ownerScope
        = ScopeKind#sk_classifier;
    }
  }
}

constraint GetInstanceReturnsSame {

  guard : self.satisfies("DefinesGetInstance")
```

```
  check {
   var returnParameter : Parameter;
   returnParameter = self.getReturnParameter();
   return (returnParameter->isDefined()
       and returnParameter.type = self);
  }
  message : " The getInstance() operation of singleton "
       + self.name + " must return " + self.name

  fix {
   title : "Change return type to " + self.name
   do {
    var returnParameter : Parameter;
    returnParameter = self.getReturnParameter();

    // If the operation does not have a return parameter
    // create one
    if (not returnParameter.isDefined()){
     returnParameter = Parameter.newInstance();
     returnParameter.kind = ParameterDirectionKind#pdk_return;
     returnParameter.behavioralFeature =
      self.getInstanceOperation();
    }
    // Set the correct return type
    returnParameter.type = self;
   }
  }
 }
}

operation Class getGetInstanceOperation() : Operation {
 return self.feature.
  select(o:Operation|o.name = "getInstance").first();
}

operation Operation getReturnParameter() : Parameter {
 return self.parameter.
```

```
  select(p:Parameter|p.kind =
   ParameterDirectionKind#pdk_return).first();
}
```

The *Singleton* context defines that the invariants it contains will be evaluated on instances of the UML *Class* type. Moreover, its guard defines that they will be evaluated only on classes that are stereotyped with the *singleton* stereotype. Therefore, unlike the OCL solution of Listing 4.7, invariants contained in this context do not need to check individually that the instances on which they are evaluated are singletons.

Constraint *DefinesGetInstance* defines no guard which means that it will be evaluated for all the instances of the context. In its *check* part, the constraint examines if the class defines an operation named *getInstance()* by invoking the *getGetInstanceOperation()* operation. If this fails, it proposes a fix that adds the missing operation to the class.

Constraint *GetInstanceIsStatic* defines a guard which states that for the constraint to be evaluated on an instance, the instance must first satisfy the *DefinesGetInstance* constraint. If it doesn't, it is not evaluated at all. In its *check* part it examines that the *getInstance()* operation is static. Note that here the constraint needs not check that the *getInstance()* operation is defined again since this is assumed by the *DefinesGetInstance* constraint on which it depends. If the constraint fails for an instance, the fix part can be invoked to change the scope of the *getInstance()* operation to static.

Constraint *GetInstanceReturnsSame* checks that the return type of the *getInstance()* operation is the singleton itself. Similarly to the *GetInstanceIsStatic* constraint, it defines that to be evaluated the *DefinesGetInstance* constraint must be satisfied. If it fails for a particular instance, the fix part can be invoked. In the fix part, if the operation defines a return parameter of incorrect type, its type is changed and if it does not define a return parameter at all, the parameter is created and added to the parameters of the operation.

By observing the two solutions the OCL solution resembles the concept of defensive programming, where conditions are embedded in supplier code, while the EVL one is closer to the design by contract [3] approach where conditions are explicitly checked in guards.

This case study has demonstrated that the additional constructs provided by EVL can reduce repetition significantly and thus enable specification of more concise constraints. Moreover, in case a constraint is not satisfied for a particular instance, the user is provided with a meaningful context-aware message and with automated facilities (fixes) for repairing the inconsistency.

4.6 Inter-Model Consistency Checking Example

In the previous example, EVL was used to check the internal consistency of a single UML model. By contrast, this example demonstrates using EVL to detect and repair occurrences of incompleteness and contradiction between two different models. In this example the simplified *ProcessLang* metamodel, which captures information about hierarchical processes, is used. To add performance information in a separate aspect *ProcessPerformanceLang* metamodel is also defined. The metamodels are displayed in Figures 4.2 and 4.3 respectively.

There are two constraints that need to be defined and evaluated in this example: that each *Process* in a process model (*PM*) has a corresponding *ProcessPerformance* in the process performance model (*PPM*), and that the *maxAcceptableTime* of a process does not exceed the sum of the *maxAcceptableTimes* of its children. This is achieves with the *PerformanceIsDefined* and the *PerformanceIsValid* EVL constraints displayed in Listing 4.9.

Listing 4.9: Exemplar EVL module containing a cross-model constraint

```
context PM!Process {

```

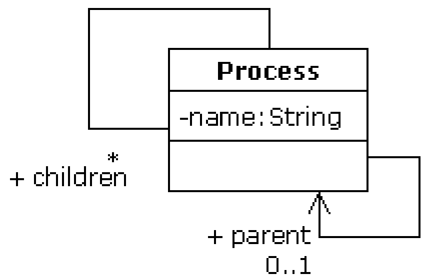

Figure 4.2: The ProcessLang Metamodel

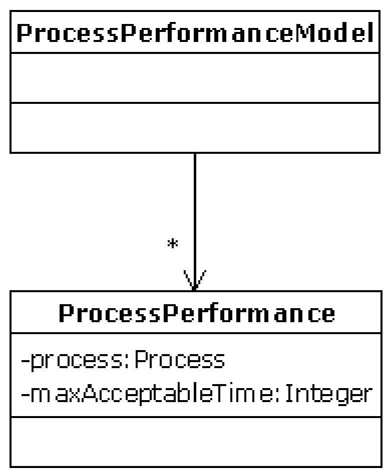

Figure 4.3: The ProcessPerformanceLang Metamodel

```
constraint PerformanceIsDefined {

  check {
    var processPerformances =
      PPM!ProcessPerformance.
      allInstances.select(pt|pt.process = self);

    return processPerformances.size() = 1;
  }

  message {
```

```
    var prefix : String;
    if (processPerformances.size() = 1) {
      prefix = "More than one performance info";
    }
    else {
      prefix = "No performance info";
    }
    return prefix + " found for process "
      + self.name;
  }

  fix {
    title : "Set the performance of " + self.name

    do {
      for (p in processPerformances.clone()) {
        delete p;
      }
      var maxAcceptableTime : Integer;
      maxAcceptableTime = UserInput.
        promptInteger("maxAcceptableTime", 0);
      var p :
        new PPM!ProcessPerformance;
      p.maxAcceptableTime = maxAcceptableTime;
      p.process = self;
    }
  }
}

constraint PerformanceIsValid {

  guard : self.satisfies("PerformanceIsDefined")
    and self.children.forAll
      (c|c.satisfies("PerformanceIsDefined"))

  check {
    var sum : Integer;
```

```
51       sum = self.children.
52         collect(c|c.getMaxAcceptableTime())
53         .sum().asInteger();
54       return self.getMaxAcceptableTime() >= sum;
55     }
56 
57     message : "Process " + self.name +
58       " has a smaller maxAcceptableTime "
59       + "than the sum of its children"
60 
61     fix {
62       title : "Increase maxAcceptableTime to " + sum
63       do {
64         self.setMaxAcceptableTime(sum);
65       }
66     }
67 
68   }
69 
70 }
71 
72 operation PM!Process getMaxAcceptableTime()
73   : Integer {
74   return PPM!ProcessPerformance.
75     allInstances.selectOne(pt|pt.process=self)
76       .maxAcceptableTime;
77 }
78 
79 operation PM!Process setMaxAcceptableTime
80   (time : Integer) {
81   PPM!ProcessPerformance.allInstances.
82     selectOne(pt|pt.process=self).maxAcceptableTime =
83     time;
84 }
```

In line 5, the check part of the *PerformanceIsDefined* constraint calculates the instances of *ProcessPerformance* in the *ProcessPerformanceModel* that have their *process* reference set to the currently examined *Process* (accessible via the *self* built-in variable) and stores it in the *processPerformances* variable. If exactly one *ProcessPerformance* is defined for the *Process*, the constraint is satisfied. Otherwise, the *message* part of the constraint, in line 13, is evaluated and an appropriate error message is displayed to the user.

Note that the *processPerformances* variable defined in the *check* part is also used from within the *message* part of the constraint. As discussed in [4], EVL provides this feature to reduce the need for duplicate calculations as our experience has shown that the message for a failed constraint often needs to utilize side-information collected in the *check* part.

To repair the inconsistency, the user can invoke the *fix* defined in line 25 that will delete any existing *ProcessPerformance* instances and create a new one with a user-defined *maxAcceptableTime* obtained using the *UserInput.promptInteger()* statement of line 33.

Unlike the *PerformanceIsDefined* constraint, the *PerformanceIsValid* constraint, line 43, defines a *guard* part (line 45). As discussed in [4], the guard part of a constraint is used to further limit the applicability of the constraint beyond the simple type check performed in the containing *context*. In this rule, the validity of the *maxAcceptableTime* of a *Process* needs to be checked only if one has been defined in the *ProcessPerformanceModel*. Therefore, the guard part of the constraint specifies that this constraint is only applicable to *Processes* where, both they and they children, satisfy the *PerformanceIsDefined* constraint.

The check part of the constraint retrieves the *maxAcceptableTime* of the process and that of its children and compares them. As the *Process* itself does not define performance information, retrieval of the value of the *maxAcceptableTime* of the respective *ProcessPerformance* object is implemented using the user-defined *getMaxAcceptableTime()* operation that is defined in line 72. In case the constraint is not satisfied, the user can invoke the *fix* defined in line 61 to repair the inconsistency by setting the *maxAcceptable-*

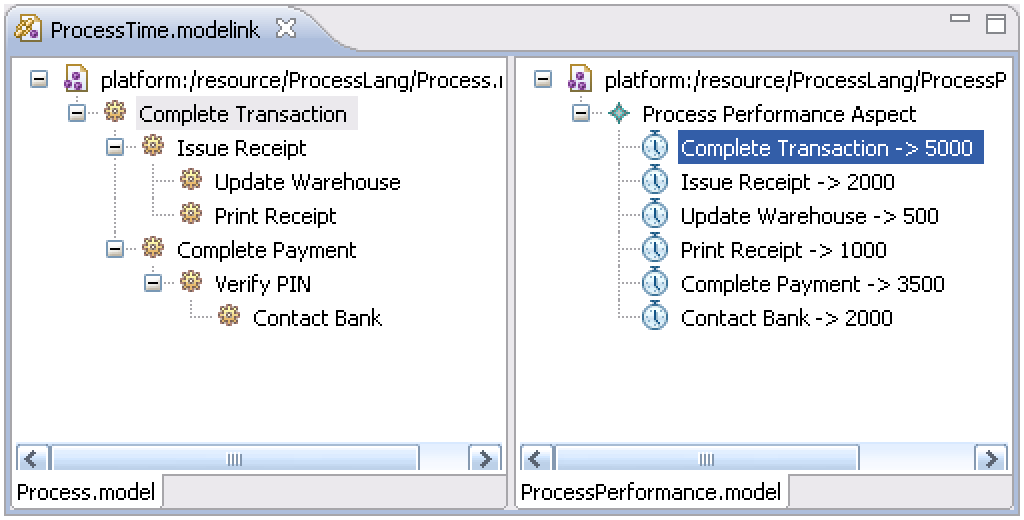

Figure 4.4: Exemplar Process and ProcessPerformance models

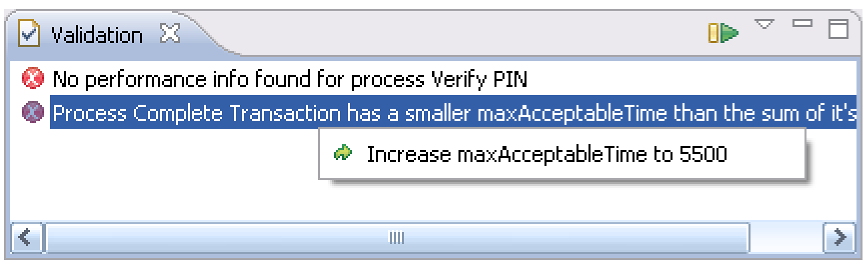

Figure 4.5: Screenshot of the validation view reporting the identified inconsistencies

Time of the process to the *sum* calculated in line 51. As discussed earlier, the fix parts of EVL invariants do not in any way guarantee that they do fix the problem they target or that in their effort to fix one problem they do not create another problem; this is left to the user. For instance, in this particular example, changing the *maxAcceptableTime* of a process through a *fix* block may render its parent process invalid.

To demonstrate the evaluation of these constraints two exemplar models that conform to the *ProcessLang* and *ProcessPerformanceLang* metamodels are used. A visual representation of the models is displayed in Figure

4.4.

Evaluating the constraints in the context of those two models reveals two problems which are reported to the user via the view displayed in Figure 4.5. Indeed by examining the two models of Figure 4.4, it becomes apparent that there is no *ProcessPerformance* linked to the *Verify PIN* process and also that the *maxAcceptableTime* of *Complete Transaction* (5000) is less than the sum of the *maxAcceptableTimes* of its children (2000 + 3500).

4.7 Summary

This section has provided a detailed discussion on the EVL model-validation language which conceptually (as opposed to technically) extends OCL. EVL provides a number of features such as support for detailed user feedback, constraint dependency management, semi-automatic transactional inconsistency resolution and (as it is based on EOL) access to multiple models of diverse metamodels and technologies.

Chapter 5

The Epsilon Transformation Language (ETL)

The aim of ETL [5] is to contribute model-to-model transformation capabilities to Epsilon. More specifically, ETL can be used to transform an arbitrary number of input models into an arbitrary number of output models of different modelling languages and technologies at a high level of abstraction.

5.1 Style

Three styles are generally recognized in model transformation languages: declarative, imperative and hybrid, each one demonstrating particular advantages and shortcomings. Declarative transformation languages are generally limited to scenarios where the source and target metamodels are similar to each other in terms of structure and thus, the transformation is a matter of a simple mapping. However they fail to address cases where significant processing and complex mappings are involved. On the other hand, purely imperative transformation languages are capable of addressing a wider range of transformation scenarios. Nevertheless, they operate at a low level of abstraction which means that users need to manually address issues such as tracing and resolving target elements from their source counterparts and orchestrating the transformation execution. To address those shortcomings, hybrid languages (such as ATL [6] and QVT [7]) provide both a declarative rule-based execution scheme as well as imperative features for handling complex transformation scenarios.

Under this rationale, ETL has been designed as a hybrid language that implements a task-specific rule definition and execution scheme but also inherits the imperative features of EOL to handle complex transformations where this is deemed necessary.

5.2 Source and Target Models

The majority of model-to-model transformation languages assume that only two models participate in each transformation: the source model and the target model. Nevertheless, it is often essential to be able to access/update additional models during a transformation (such as trace or configuration models). Building on the facilities provided by EMC and EOL, ETL enables specification of transformations that can transform an arbitrary number of source models into an arbitrary number of target models.

Another common assumption is that the contents of the target models are insignificant and thus a transformation can safely overwrite its contents. As discussed in the sequel, ETL - like all Epsilon languages - enables the user to specify, for each involved model, whether its contents need to be preserved or not.

5.3 Abstract Syntax

As illustrated in Figure 5.1, ETL transformations are organized in modules (*EtlModule*). A module can contain a number of transformation rules (*TransformationRule*). Each rule has a unique name (in the context of the module) and also specifies one *source* and many *target* parameters. A transformation rule can also *extend* a number of other transformation rules and be declared as *abstract*, *primary* and/or *lazy*[1]. To limit its applicability to a subset of elements that conform to the type of the *source* parameter, a rule can optionally define a guard which is either an EOL expression or a block of EOL statements. Finally, each rule defines a block of EOL statements (*body*) where the logic for populating the property values of the target model elements is specified.

Besides transformation rules, an ETL module can also optionally contain a number of *pre* and *post* named blocks of EOL statements which, as discussed later, are executed before and after the transformation rules respectively.

[1]The concept of lazy rules was first introduced in ATL

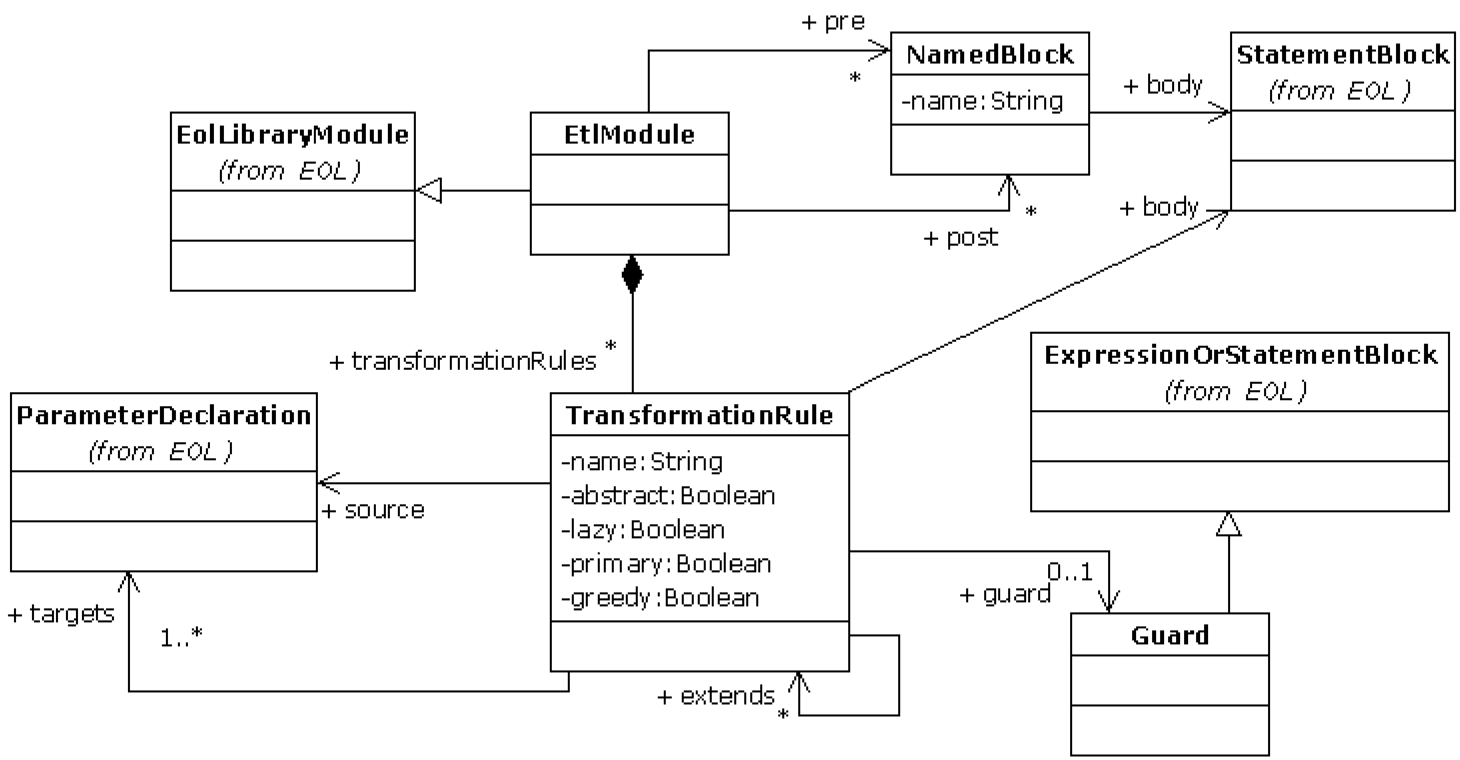

Figure 5.1: ETL Abstract Syntax

5.4 Concrete Syntax

The concrete syntax of a transformation rule is displayed in Listing 5.1. The optional *abstract*, *lazy* and *primary* attributes of the rule are specified using respective annotations. The name of the rule follows the *rule* keyword and the *source* and *target* parameters are defined after the *transform* and *to* keywords. Also, the rule can define an optional comma-separated list of rules it extends after the *extends* keyword. Inside the curly braces ({}), the rule can optionally specify its *guard* either as an EOL expression following a colon (:) (for simple guards) or as a block of statements in curly braces (for more complex guards). Finally, the *body* of the rule is specified as a sequence of EOL statements.

Listing 5.1: Concrete Syntax of a TransformationRule

```
(@abstract)?
(@lazy)?
(@primary)?
rule <name>
 transform <sourceParameterName>:<sourceParameterType>
 to (<rightParameterName>:<rightParameterType>
 (, <rightParameterName>:<rightParameterType>)*
 (extends (<ruleName>,)*<ruleName>)? {

 (guard (:expression)|({statement+}))?

 statement+
}
```

Pre and *post* blocks have a simple syntax that, as presented in Listing 5.2, consists of the identifier (*pre* or *post*), an optional name and the set of statements to be executed enclosed in curly braces.

Listing 5.2: Concrete Syntax of Pre and Post blocks

```
(pre|post) <name> {
 statement+
```

```
}
```

5.5 Execution Semantics

5.5.1 Rule and Block Overriding

Similarly to ECL, an ETL module can import a number of other ETL modules. In this case, the importing ETL module inherits all the rules and pre/post blocks specified in the modules it imports (recursively). If the module specifies a rule or a pre/post block with the same name, the local rule/block overrides the imported one respectively.

5.5.2 Rule Execution Scheduling

When an ETL module is executed, the *pre* blocks of the module are executed first in the order in which they have been specified.

Following that, each *non-abstract* and *non-lazy* rule is executed for all the elements on which it is applicable. To be applicable on a particular element, the element must have a kind-of relationship with the type defined in the rule's *sourceParameter* and must also satisfy the *guard* of the rule (and all the rules it extends). When a rule is executed on an applicable element, the target elements are initially created by instantiating the *targetParameters* of the rules, and then their contents are populated using the EOL statements of the *body* of the rule.

Finally, when all rules have been executed, the *post* blocks of the module are executed in the order in which they have been declared.

5.5.3 Source Elements Resolution

Resolving target elements that have (or can be) transformed from source elements by other rules is a frequent task in the body of a transformation rule. To automate this task and reduce coupling between rules, ETL contributes the *equivalents()* and *equivalent()* built-in operations that automatically resolve source elements to their transformed counterparts in the target models.

When the *equivalents()* operation is applied on a single source element (as opposed to a collection of them), it inspects the established transformation trace (displayed in Figure 5.3) and invokes the applicable rules (if necessary) to calculate the counterparts of the element in the target model. When applied to a collection it returns a *Bag* containing *Bags* that in turn contain the counterparts of the source elements contained in the collection. The *equivalents()* operation can be also invoked with an arbitrary number of rule names as parameters to invoke and return only the equivalents created by specific rules. Unlike the main execution scheduling scheme discussed above, the *equivalents()* operation invokes both *lazy* and *non-lazy* rules.

With regard to the ordering of the results of the *equivalents()* operations, the returned elements appear in the respective order of the rules that have created them. An exception to this occurs when one of the rules is declared as *primary*, in which case its results precede the results of all other rules.

ETL also provides the convenience *equivalent()* operation which, when applied to a single element, returns only the first element of the respective result that would have been returned by the *equivalents()* operation discussed above. Also, when applied to a collection the *equivalent()* operation returns a flattened collection (as opposed to the result of *equivalents()* which is a *Bag* of *Bags* in this case). As with the *equivalents()* operation, the *equivalent()* operation can also be invoked with or without parameters.

The semantics of the *equivalent()* operation are further illustrated through a simple example. In this example, we need to transform a model that conforms to the Tree metamodel displayed in Figure 8.3 into a model that conforms to the Graph metamodel of Figure 5.2. More specifically, we need

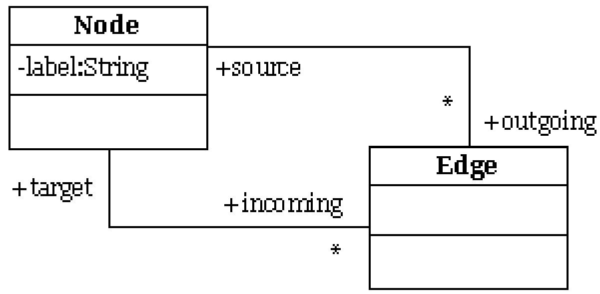

Figure 5.2: A Simple Graph Metamodel

to transform each *Tree* element to a *Node*, and an *Edge* that connects it with the *Node* that is equivalent to the tree's *parent*. This is achieved using the rule of Listing 5.3.

In lines 1-3, the *Tree2Node* rule specifies that it can transform elements of the *Tree* type in the *Tree* model into elements of the *Node* type in the *Graph* model. In line 4 it specifies that the name of the created Node should be the same as the name of the source Tree. If the parent of the source *Tree* is defined (line 7), the rule creates a new *Edge* (line 8) and sets its *source* property to the created *Node* (line 9) and its *target* property to the *equivalent Node* of the source *Tree*'s *parent* (line 10).

Listing 5.3: Exemplar ETL Rule demonstrating the *equivalent()* operation

```
1  rule Tree2Node
2   transform t : Tree!Tree
3   to n : Graph!Node {
4
5   n.label = t.label;
6
7   if (t.parent.isDefined()) {
8     var edge = new Graph!Edge;
9     edge.source = n;
10    edge.target = t.parent.equivalent();
11  }
12 }
```

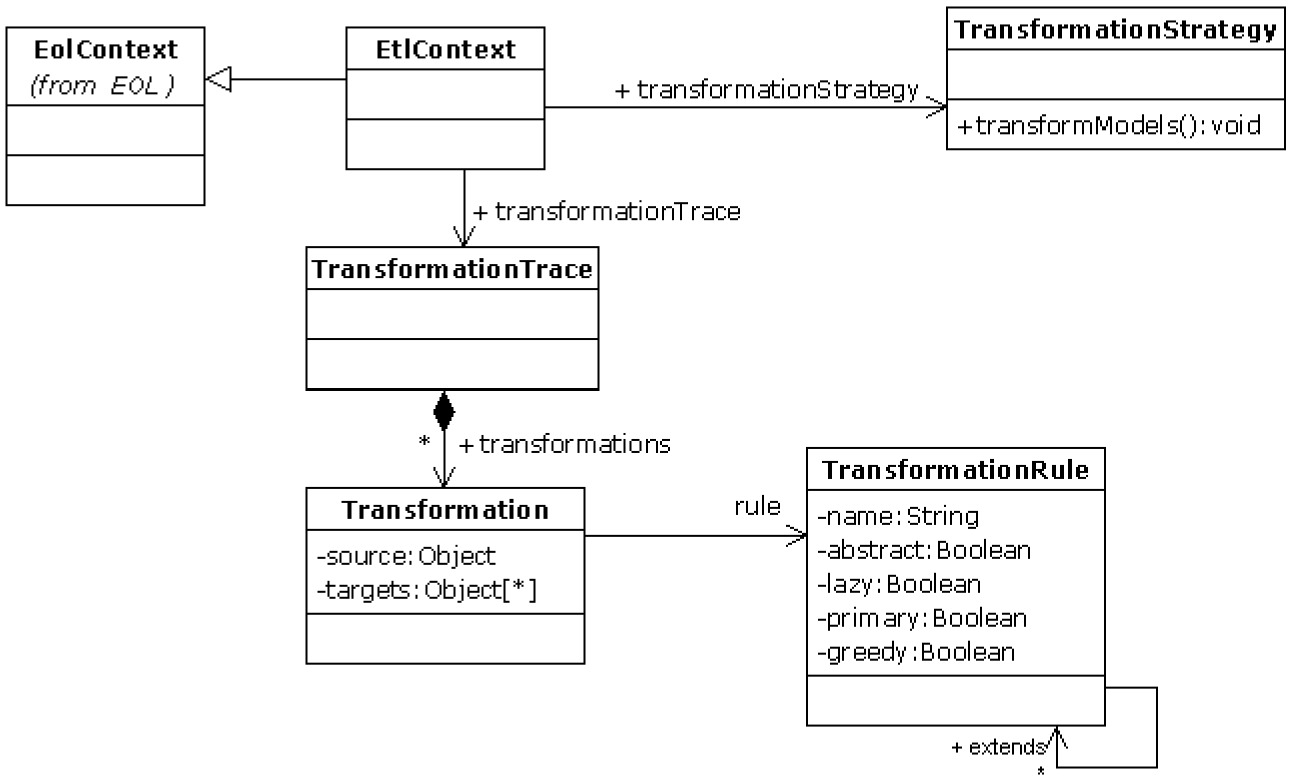

Figure 5.3: ETL Runtime

5.5.4 Overriding the semantics of the EOL SpecialAssignmentOperator

As discussed above, resolving the equivalent(s) or source model elements in the target model is a recurring task in model transformation. Furthermore, in most cases resolving the equivalent of a model element is immediately followed by assigning/adding the obtained target model elements to the value(s) of a property of another target model element. For example, in line 10 of Listing 5.3 the *equivalent* obtained is immediately assigned to the *target* property of the generated *Edge*. To make transformation specifications more readable, ETL overrides the semantics of the *SpecialAssignmentStatement* (*::=* in terms of concrete syntax), described in Section 3.5.3 to set its left-hand side, not to the element its right-hand side evaluates to, but to its *equivalent* as calculated using the *equivalent()* operation discussed above. Using this feature, line 10 of the *Tree2Node* rule can be rewritten as shown in Listing 5.4

Listing 5.4: Rewritten Line 10 of the *Tree2Node* Rule Demonstrated in Listing 5.3

```
edge.target ::= t.parent;
```

5.6 Interactive Transformations

Using the user interaction facilities of EOL discussed in Section 3.7, an ETL transformation can become interactive by prompting the user for input during its execution. For example in Listing 5.5, we modify the *Tree2Node* rule originally presented in Listing 5.3 by adding a *guard* part that uses the user-input facilities of EOL (more specifically the *UserInput.confirm(String,Boolean)* operation) to enable the user select manually at runtime which of the Tree elements need to be transformed to respective Node elements in the target model and which not.

Listing 5.5: Exemplar Interactive ETL Transformation

```
rule Tree2Node
 transform t : Tree!Tree
 to n : Graph!Node {

 guard : UserInput.confirm
   ("Transform tree " + t.label + "?", true)

 n.label = t.label;
 var target : Graph!Node ::= t.parent;
 if (target.isDefined()) {
   var edge = new Graph!Edge;
   edge.source = n;
   edge.target = target;
 }
}
```

5.7 Summary

This section has provided a detailed discussion on the Epsilon Transformation Language (ETL). ETL is capable of transforming an arbitrary number of source models into an arbitrary number of target models. ETL adopts a hybrid style and features declarative rule specification using advanced concepts such as *guards*, *abstract*, *lazy* and *primary* rules, and automatic resolution of target elements from their source counterparts. Also, as ETL is based on EOL reuses its imperative features to enable users to specify particularly complex, and even interactive, transformations.

Chapter 6

The Epsilon Wizard Language (EWL)

There are two types of model-to-model transformations: mapping and update transformations [8]. Mapping transformations typically transform a source model into a set of target models expressed in (potentially) different modelling languages by creating zero or more model elements in the target models for each model element of the source model. By contrast, update transformations perform in-place modifications in the source model itself. They can be further classified into two subcategories: transformations in the small and in the large. Update transformations in the large apply to sets of model elements calculated using well-defined rules in a batch manner. An example of this category of transformations is a transformation that automatically adds accessor and mutator operations for all attributes in a UML model. On the other hand, update transformations in the small are applied in a user-driven manner on model elements that have been explicitly selected by the user. An example of this kind of transformations is a transformation that renames a *user-specified* UML class and all its incoming associations consistently.

In Epsilon, mapping transformations can be specified using ETL as discussed in Section 5, and update transformations in the large can be implemented either using the model modification features of EOL or using an ETL transformation in which the source and target models are the same model. By contrast, update transformations in the small cannot be effectively addressed by any of the languages presented so far.

The following section discusses the importance of update transformations in the small and motivates the definition of a task-specific language (Epsilon Wizard Language (EWL)) that provides tailored and effective support for defining and executing update transformations on models of diverse metamodels.

6.1 Motivation

Constructing and refactoring models is undoubtedly a mentally intensive process. However, during modelling, recurring patterns of model update activities typically appear. As an example, when renaming a class in a UML class diagram, the user also needs to manually update the names of association ends that link to the renamed class. Thus, when renaming a class from *Chapter* to *Section*, all associations ends that point to the class and are named *chapter* or *chapters* should be also renamed to *section* and *sections* respectively. As another example, when a modeller needs to refactor a UML class into a singleton [2], they need to go through a number of well-defined, but trivial, steps such as attaching a stereotype ($<< singleton >>$), defining a static *instance* attribute and adding a static *getInstance()* method that returns the unique instance of the singleton.

It is generally accepted that performing repetitive tasks manually is both counter-productive and error-prone [9]. On the other hand, failing to complete such tasks correctly and precisely compromises the consistency, and thus the quality, of the models. In Model Driven Engineering, this is particularly important since models are increasingly used to automatically produce (parts of) working systems.

6.1.1 Automating the Construction and Refactoring Process

Contemporary modelling tools provide built-in transformations (*wizards*) for automating common repetitive tasks. However, according to the architecture of the designed system and the specific problem domain, additional repetitive tasks typically appear, which cannot be addressed by the preconceived built-in wizards of a modelling tool. To address the automation problem in its general case, users must be able to easily define update transformations (wizards) that are tailored to their specific needs.

To an extent, this can be achieved via the extensible architecture that state-of-the-art modelling tools often provide and which enables users to add functionality to the tool via scripts or application code using the implementation language of the tool. Nevertheless, as discussed in [10], the majority of modelling tools provide an API through which they expose an edited model, which requires significant effort to learn and use. Also, since each API is proprietary, such scripts and extensions are not portable to other tools. Finally, API scripting languages and third-generation languages such as Java and C++ are not particularly suitable for model navigation and modification [10].

Furthermore, existing languages for mapping transformations, such as QVT, ATL and ETL, cannot be used as-is for this purpose, because these languages have been designed to operate in a batch manner without human involvement in the process. By contrast, as discussed above, the task of constructing and refactoring models is inherently user-driven.

6.2 Update Transformations in the Small

Update transformations are actions that automatically create, update or delete model elements based on a selection of existing elements in the model and information obtained otherwise (e.g. through user input), in a user-driven fashion. In this section such actions are referred to as *wizards* instead of *rules* to reduce confusion between them and rules of mapping transformation languages. In the following sections the desirable characteristics of wizards are elaborated informally.

6.2.1 Structure of Wizards

In its simplest form, a wizard only needs to define the actions it will perform when it is applied to a selection of model elements. The structure of such a wizard that transforms a UML class into a *singleton* is shown using pseudo-code in Listing 6.1.

Listing 6.1: The simplest form of a wizard for refactoring a class into a singleton

```
do :
  attach the singleton stereotype
  create the instance attribute
  create the getInstance method
```

Since not all wizards apply to all types of elements in the model, each wizard needs to specify the types of elements to which it applies. For example, the wizard of Listing 6.1, which automatically transforms a class into a singleton, applies only when the selected model element is a class. The simplest approach to ensuring that the wizard will only be applied on classes is to enclose its body in an *if* condition as shown in Listing 6.2.

Listing 6.2: The wizard of Listing 6.1 enhanced with an if condition

```
do :
```

```
if (selected element is a class) {
  attach the singleton stereotype
  create the instance attribute
  create the getInstance method
}
```

A more modular approach is to separate this condition from the body of the wizard. This is shown in Listing 6.3 where the condition of the wizard is specified as a separate *guard* stating that the wizard applies only to elements of type Class. The latter is preferable since it enables filtering out wizards that are not applicable to the current selection of elements by evaluating only their *guard* parts and rejecting those that return *false*. Thus, at any time, the user can be provided with only the wizards that are applicable to the current selection of elements. Filtering out irrelevant wizards reduces confusion and enhances usability, particularly as the list of specified wizards grows.

Listing 6.3: The wizard of Listing 6.2 with an explicit $guard$ instead of the if condition

```
guard : selected element is a class
do :
  attach the singleton stereotype
  create the instance attribute
  create the getInstance method
```

To enhance usability, a wizard also needs to define a short human-readable description of its functionality. To achieve this, another field named *title* has been added. There are two options for defining the title of a wizard: the first is to use a static string and the second to use a dynamic expression. The latter is preferable since it enables definition of context-aware titles.

Listing 6.4: The wizard of Listing 6.3 enhanced with a $title$ part

```
guard : selected element is a class
title : Convert class <class-name> into a singleton
```

```
do :
 attach the singleton stereotype
 create the instance attribute
 create the getInstance method
```

6.2.2 Capabilities of Wizards

The *guard* and *title* parts of a wizard need to be expressed using a language that provides model querying and navigation facilities. Moreover, the *do* part also requires model modification capabilities to implement the transformation. To achieve complex transformations, it is essential that the user can provide additional information. For instance, to implement a wizard that addresses the class renaming scenario discussed in Section 6.1, the information provided by the selected class does not suffice; the user must also provide the new name of the class. Therefore, EWL must also provide mechanisms for capturing user input.

6.3 Abstract Syntax

Since EWL is built atop Epsilon, its abstract and concrete syntax need only to define the concepts that are relevant to the task it addresses; they can reuse lower-level constructs from EOL. A graphical overview of the abstract syntax of the language is provided in Figure 6.1.

The basic concept of the EWL abstract syntax is a *Wizard*. A wizard defines a *name*, a *guard* part, a *title* part and a *do* part. Wizards are organized in *Modules*. The *name* of a wizard acts as an identifier and must be unique in the context of a module. The *guard* and *title* parts of a wizard are of type *ExpressionOrStatementBlock*, inherited from EOL. An *ExpressionOrStatementBlock* is either a single EOL expression or a block of EOL statements that include one or more *return* statements. This construct allows users to express simple declarative calculations as single expressions

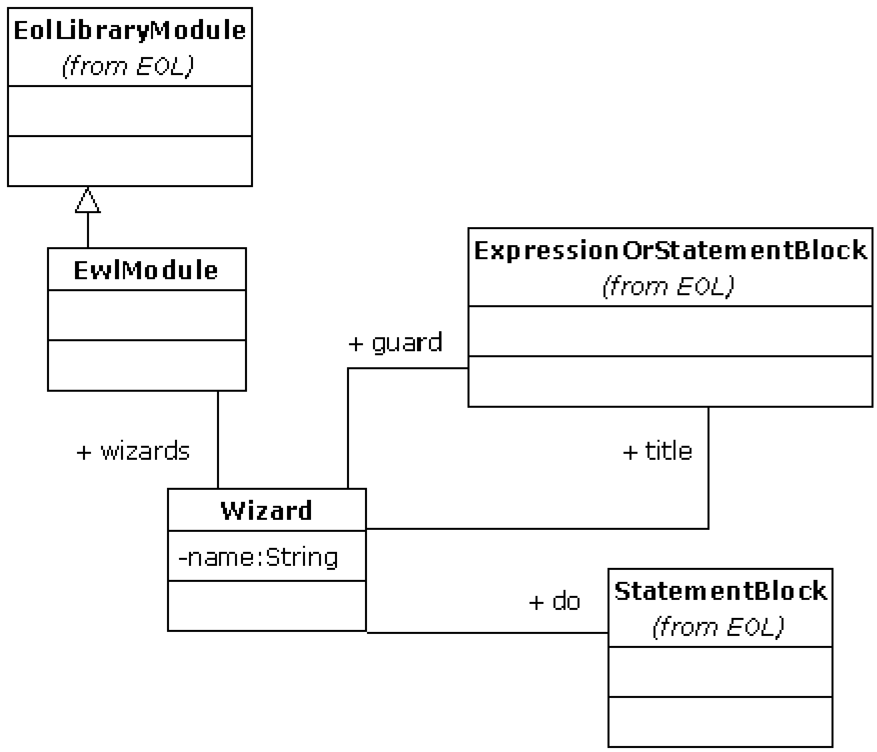

Figure 6.1: EWL Abstract Syntax

and complex calculations as blocks of imperative statements. The usefulness of this construct is further discussed in the examples presented in Section 6.6. Finally, the *do* part of the wizard is a block of EOL statements that specify the effects of the wizard when applied to a compatible selection of model elements.

6.4 Concrete Syntax

Listing 6.5 presents the concrete syntax of EWL wizards.

Listing 6.5: Concrete syntax of EWL wizards

```
wizard <name> {
  (guard (:expression)|({statementBlock}))?
  (title (:expression)|({statementBlock}))?
  do {
    statementBlock
  }
}
```

6.5 Execution Semantics

The process of executing EWL wizards is inherently user-driven and as such it depends on the environment in which they are used. In general, each time the selection of model elements changes (i.e. the user selects or deselects a model element in the modelling tool), the guards of all wizards are evaluated. If the guard of a wizard is satisfied, the *title* part is also evaluated and the wizard is added to a list of *applicable* wizards. Then, the user can select a wizard and execute its *do* part to perform the intended transformation.

In EWL, variables defined and initialized in the *guard* part of the wizard can be accessed both by the *title* and the *do* parts. In this way, results of calculations performed in the *guard* part can be re-used, instead of recalculated in the subsequent parts. The practicality of this approach is discussed in more detail in the examples that follow. Also, the execution of the *do* part of each wizard is performed in a transactional mode by exploiting the transaction capabilities of the underlying model connectivity framework, so that possible logical errors in the *do* part of a wizard do not leave the edited model in an inconsistent state.

6.6 Examples

This section presents three concrete examples of EWL wizards for refactoring UML 1.4 models. The aim of this section is not to provide complete implementations that address all the sub-cases of each scenario but to provide enhanced understanding of the concrete syntax, the features and the capabilities of EWL to the reader. Moreover, it should be stressed again that although the examples in this section are based on UML models, by building on Epsilon, EWL can be used to capture wizards for diverse modelling languages and technologies.

Converting a Class into a Singleton

The singleton pattern [2] is applied when there is a class for which only one instance can exist at a time. In terms of UML, a singleton is a class stereotyped with the $<< singleton >>$ stereotype, and it defines a static attribute named *instance* which holds the value of the unique instance. It also defines a static *getInstance()* operation that returns that unique instance. Wizard *ClassToSingleton*, presented in Listing 6.6, simplifies the process of converting a class into a singleton by adding the proper stereotype, attribute and operation to it automatically.

Listing 6.6: Implementation of the ClassToSingleton Wizard

```
wizard ClassToSingleton {

 // The wizard applies when a class is selected
 guard : self.isTypeOf(Class)

 title : "Convert " + self.name + " to a singleton"

 do {
  // Create the getInstance() operation
  var gi : new Operation;
```

```
    gi.owner = self;
    gi.name = "getInstance";
    gi.visibility = VisibilityKind#vk_public;
    gi.ownerScope = ScopeKind#sk_classifier;

    // Create the return parameter of the operation
    var ret : new Parameter;
    ret.type = self;
    ret.kind = ParameterDirectionKind#pdk_return;
    gi.parameter = Sequence{ret};

    // Create the instance field
    var ins : new Attribute;
    ins.name = "instance";
    ins.type = self;
    ins.visibility = VisibilityKind#vk_private;
    ins.ownerScope = ScopeKind#sk_classifier;
    ins.owner = self;

    // Attach the <<singleton>> stereotype
    self.attachStereotype("singleton");
  }
}

// Attaches a stereotype with the specified name
// to the Model Element on which it is invoked
operation ModelElement attachStereotype(name : String) {
    var stereotype : Stereotype;

    // Try to find an existing stereotype with this name
    stereotype = Stereotype.allInstances.selectOne(s|s.name = name);

    // If there is no existing stereotype
    // with that name, create one
    if (not stereotype.isDefined()){
      stereotype = Stereotype.createInstance();
      stereotype.name = name;
```

```
48        stereotype.namespace = self.namespace;
49      }
50
51      // Attach the stereotype to the model element
52      self.stereotype.add(stereotype);
53  }
```

The *guard* part of the wizard specifies that it is only applicable when the selection is a single UML class. The *title* part specifies a context-aware title that informs the user of the functionality of the wizard and the *do* part implements the functionality by adding the *getInstance* operation (lines 10-14), the *instance* attribute (lines 23-28) and the $<< singleton >>$ stereotype (line 31).

The stereotype is added via a call to the *attachStereotype()* operation. Attaching a stereotype is a very common action when refactoring UML models, particularly where UML profiles are involved, and therefore to avoid duplication, this reusable operation that checks for an existing stereotype, creates it if it does not already exists, and attaches it to the model element on which it is invoked has been specified.

An extended version of this wizard could also check for existing association ends that link to the class and for which the upper-bound of their multiplicity is greater than one and either disallow the wizard from executing on such classes (in the $guard$ part) or update the upper-bound of their multiplicities to one (in the do part). However, the aim of this section is not to implement complete wizards that address all sub-cases but to provide a better understanding of the concrete syntax and the features of EWL. This principle also applies to the examples presented in the sequel.

Renaming a Class

The most widely used convention for naming attributes and association ends of a given class is to use a lower-case version of the name of the class as the name of the attribute or the association end. For instance, the two ends of a one-to-many association that links classes `Book` and `Chapter` are most likely to be named `book` and `chapters` respectively. When renaming a class (e.g. from `Chapter` to `Section`) the user must then manually traverse the model to find all attributes and association ends of this type and update their names (i.e. from `chapter` or `bookChapter` to `section` and `bookSection` respectively). This can be a daunting process especially in the context of large models. Wizard `RenameClass` presented in Listing 6.7 automates this process.

Listing 6.7: Implementation of the RenameClass Wizard

```
wizard RenameClass {

 // The wizard applies when a Class is selected
 guard : self.isKindOf(Class)

 title : "Rename class " + self.name

 do {
  var newName : String;

  // Prompt the user for the new name of the class
  newName = UserInput.prompt("New name for class " + self.name);
  if (newName.isDefined()) {
   var affectedElements : Sequence;

   // Collect the AssociationEnds and Attributes
   // that are affected by the rename
   affectedElements.addAll(
    AssociationEnd.allInstances.select(ae|ae.participant=self));
   affectedElements.addAll(
    Attribute.allInstances.select(a|a.type = self));

   var oldNameToLower : String;
   oldNameToLower = self.name.firstToLowerCase();
```

```
    var newNameToLower : String;
    newNameToLower = newName.firstToLowerCase();

    // Update the names of the affected AssociationEnds
    // and Attributes
    for (ae in affectedElements) {
      ae.replaceInName(oldNameToLower, newNameToLower);
      ae.replaceInName(self.name, newName);
    }
    self.name = newName;
   }
  }

}

// Renames the ModelElement on which it is invoked
operation ModelElement replaceInName
  (oldString : String, newString : String) {

  if (oldString.isSubstringOf(self.name)) {
    // Calculate the new name
    var newName : String;
    newName = self.name.replace(oldString, newString);

    // Prompt the user for confirmation of the rename
    if (UserInput.confirm
      ("Rename " + self.name + " to " + newName + "?")) {
      // Perform the rename
      self.name = newName;
    }
  }
}
```

As with the `ClassToSingleton` wizard, the `guard` part of `RenameClass` specifies that the wizard is applicable only when the selection is a simple class and the *title* provides a context-aware description of the functionality of the wizard.

As discussed in Section 6.2, the information provided by the selected class itself does not suffice in the case of renaming since the new name of the class is not specified anywhere in the existing model. In EWL, and in all languages that build on EOL, user input can be obtained using the built-in `UserInput` facility. Thus, in line 12 the user is prompted for the new name of the class using the `UserInput.prompt()` operation. Then, all the association ends and attributes that refer to the class are collected in the `affectedElements` sequence (lines 14-21). Using the `replaceInName` operation (lines 31 and 32), the name of each one is examined for a substring of the upper-case or the lower-case version of the old name of the class. In case the check returns true, the user is prompted to confirm (line 48) that the feature needs to be renamed. This further highlights the importance of user input for implementing update transformations with fine-grained user control.

Moving Model Elements into a Different Package

A common refactoring when modelling in UML is to move model elements, particularly Classes, between different packages. When moving a pair of classes from one package to another, the associations that connect them must also be moved in the target package. To automate this process, Listing 6.8 presents the `MoveToPackage` wizard.

Listing 6.8: Implementation of the MoveToPackage Wizard

```
wizard MoveToPackage {

 // The wizard applies when a Collection of
 // elements, including at least one Package
 // is selected
 guard {
  var moveTo : Package;
  if (self.isKindOf(Collection)) {
```

```
9         moveTo = self.select(e|e.isKindOf(Package)).last();
10      }
11      return moveTo.isDefined();
12    }
13
14    title : "Move " + (self.size()-1) + " elements to " + moveTo.name
15
16    do {
17      // Move the selected Model Elements to the
18      // target package
19      for (me in self.excluding(moveTo)) {
20        me.namespace = moveTo;
21      }
22
23      // Move the Associations connecting any
24      // selected Classes to the target package
25      for (a in Association.allInstances) {
26        if (a.connection.forAll(c|self.includes(c.participant))){
27          a.namespace = moveTo;
28        }
29      }
30    }
31
32  }
```

The wizard applies when more than one element is selected and at least one of the elements is a *Package*. If more than one package is selected, the last one is considered as the target package to which the rest of the selected elements will be moved. This is specified in the *guard* part of the wizard.

To reduce user confusion in identifying the package to which the elements will be moved, the name of the target package appears in the title of the wizard. This example shows the importance of the decision to express the title as a dynamically calculated expression (as opposed to a static string). It is worth noting that in the *title* part of the wizard (line 14), the *moveTo* variable declared in the *guard* (line 7) is referenced. Through experimenting with a number of wizards, it has been noticed that in complex wizards repeated calculations need to be performed in the *guard*, *title* and *do* parts of the wizard. To eliminate this duplication, the scope of variables defined in the *guard* part has been extended so that they are also accessible from the *title* and *do* part of the wizard.

6.7 Summary

This section has presented the Epsilon Wizard Language (EWL), a language for specifying and executing update transformations in the small on models of diverse metamodels. EWL provides a textual concrete syntax tailored to the task and features such as dynamically calculated wizard titles, transactional execution of the *do* parts of wizards and user interaction.

Chapter 7

The Epsilon Generation Language (EGL)

EGL provides a language for M2T in the large. EGL is a model-driven template-based code generator, built atop Epsilon, and re-using all of EOL. In this section, we discuss the design of EGL and its construction from existing Epsilon tools.

7.1 Abstract Syntax

Figure 7.1 depicts the abstract syntax of EGL's core functionality.

In common with other template-based code generators, EGL defines *sections*, from which templates may be constructed. Static sections delimit sections whose contents appear verbatim in the generated text. Dynamic sections contain executable code that can be used to control the generated text.

In its dynamic sections, EGL re-uses EOL's mechanisms for structuring program control flow, performing model inspection and navigation, and defining custom operations. EGL provides an EOL object, `out`, for use within dynamic sections. This can be used to perform operations on the

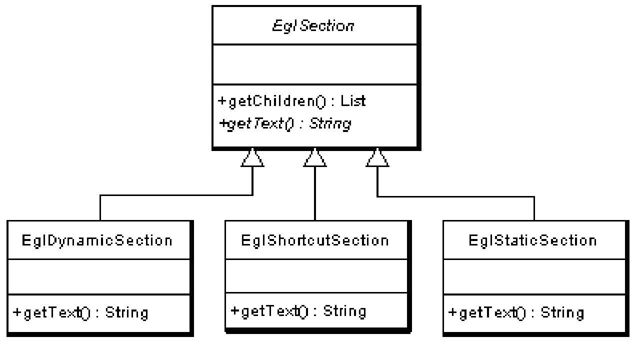

Figure 7.1: The abstract syntax of EGL's core.

generated text, such as appending and removing strings and specifying the type of text to be generated.

EGL also provides syntax for defining *dynamic output* sections, which provide a convenient shorthand for outputting text from within dynamic sections. Similar syntax is often provided by template-based code generators.

7.2 Concrete Syntax

The concrete syntax of EGL mirrors the style of other template-based code generation languages. The tag pair `[% %]` is used to delimit a dynamic section. Any text not enclosed in such a tag pair is contained in a static section. Listing 7.1 illustrates the use of dynamic and static sections to form a basic EGL template.

Listing 7.1: A basic EGL template.

```
[% for (i in Sequence{1..5}) { %]
i is [%=i%]
[% } %]
```

The *[%=expr%]* construct is shorthand for *[% out.print(expr); %]*, which appends *expr* to the output generated by the transformation. Note that the `out` keyword also provides *println(Object)* and *chop(Integer)* methods, which can be used to construct text with linefeeds, and to remove the specified number of characters from the end of the generated text.

EGL exploits EOL's model querying capabilities to output text from models specified as input to transformations. For example, the EGL template depicted in Listing 7.2 may be used to generate text from a model that conforms to a metamodel that describes an object-oriented system.

Listing 7.2: Generating the name of each Class contained in an input model.

```
[% for (class in Class.allInstances) { %]
[%=class.name%]
[% } %]
```

7.3 Parsing and Preprocessing

EGL provides a parser which generates an abstract syntax tree comprising static, dynamic and dynamic output nodes for a given template. A preprocessor then translates each section into corresponding EOL: static and dynamic output sections generate `out.print()` statements. Dynamic sections are already specified in EOL, and require no translation.

Consider the EGL depicted in Listing 7.1. The preprocessor produces the EOL shown in Listing 7.3 – the `[% %]` and `[%=  %]` tag pairs have been removed, and the text to be output is translated into `out.print()` statements.

Listing 7.3: Resulting EOL generated by the preprocessor.

```
for (i in Sequence{1..5}) {
  out.print("i is ");
  out.print(i);
  out.print("\r\n");
```

```
}
```

When comparing Listings 7.1 and 7.3, it can be seen that the template-based syntax is more concise, while the preprocessed syntax is arguably more readable. For templates where there is more dynamic than static text, such as the one depicted in Listing 7.1, a template-based syntax is often less readable. However, this loss of readability is somewhat mitigated by EGL's developer tools, which are discussed in Section 7.7.1. By contrast, for templates that exhibit more static than dynamic text, a template-based syntax is often more readable than its preprocessed equivalent.

7.4 Deriving EGL from EOL

In designing functionality specific to M2T transformation, one option was to enrich the existing EOL syntax with keywords such as *print*, *contentType* and *merge*. However, EOL underpins all Epsilon languages, and the additional keywords were needed only for M2T. Furthermore, the refactorings needed to support the new keywords affect many components – the lexer, parser, execution context and execution engine – complicating maintenance and use by other developers. Instead, we define a minimal syntax for EGL, allowing easy implementation of an EGL execution engine as a simple preprocessor for EOL.

The EGL execution engine augments the default context used by EOL during execution with two read-only, global variables: *out* (Section 7.2) and *TemplateFactory* (Section 7.5). The *out* object defines methods for performing operations specific to M2T translation, and the *TemplateFactory* object provides methods for loading other templates. The implementation for the latter was extended, late in the EGL development, to provide support for accessing templates from a file-system – a trivial extension that caused no migration problems for existing EGL templates, due to the way in which EGL extends EOL.

7.5 Co-ordination

In the large, M2T transformations need to be able to not only generate text, but also files, which are then used downstream as development artefacts. An M2T tool must provide the language constructs for producing files and manipulating the local file system. Often, this requires that the destination, as well as the contents, be dynamically defined at a transformation's execution time.

The EGL co-ordination engine supplies mechanisms for generating text directly to files. The design encourages decoupling of generated text from output destinations. The *Template* data-type is provided to allow nested execution of M2T transformations, and operations on instances of this data-type facilitate the generation of text directly to file. A factory object, *TemplateFactory*, is provided to simplify the creation of *Template* objects. In Listing 7.4, these objects are used in an EGL template that loads the the EGL template in Listing 7.2 from the file, ClassNames.egl, and writes out to disk the text generated by executing ClassNames.egl.

Listing 7.4: Storing the name of each Class to disk.

```
[%
 var t : Template = TemplateFactory.load("ClassNames.egl");
 t.process();
 t.generate("Output.txt");
%]
```

This approach to co-ordination allows EGL to be used to generate one or more files from a single input model. Moreover, EGL's co-ordination engine facilitates the specification of platform-specific details (the destination of any files being generated) separately from the platform-independent details (the contents of any files being generated).

7.6 Merge Engine

EGL provides language constructs that allow M2T transformations to designate regions of generated text as *protected*. The contents of protected regions are preserved every time a M2T transformation generates text to the same destination.

Protected regions are specified by the *preserve(String, String, String, Boolean, String)* method on the `out` keyword. The first two parameters define the comment delimiters of the target language. The other parameters provide the name, enable-state and content of the protected region, as illustrated in Listing 7.5.

Listing 7.5: Protected region declaration using the preserve method.

```
[%=out.preserve("/*", "*/", "anId", true,
            "System.out.println(foo);")
%]
```

A protected region declaration may have many lines, and use many EGL variables in the contents definition. To enhance readability, EGL provides two additional methods on the `out` keyword: *startPreserve(String, String, String, Boolean)* and `stopPreserve`. Listing 7.6 uses these to generate a protected region equivalent to that in Listing 7.5.

Listing 7.6: Protected region declaration.

```
[%=out.startPreserve("/*", "*/", "anId", true)%]
System.out.println(foo);
[%=out.stopPreserve()%]
```

Because an EGL template may contain many protected regions, EGL also provides a separate method to set the target language generated by the current template, *setContentType(String)*. By default, EGL recognises Java, HTML, Visual Basic, Perl and EGL as valid content types. An alternative configuration file can be used to specify further content types. Following a call to `setContentType`, the first two arguments to the `preserve` and `startPreserve` methods can be omitted, as shown in Listing 7.7.

Listing 7.7: Setting the content type.

```
[% out.setContentType("Java"); %]
[%=out.preserve("anId", true, "System.out.println(foo);")%]
```

Because some languages define more than one style of comment delimiter, EGL allows mixed use of the styles for `preserve` and `startPreserve` methods.

Once a content type has been specified, a protected region may be declared entirely from a static section, using the syntax in Listing 7.8.

Listing 7.8: Declaring a protected region from within a static section.

```
[% out.setContentType("Java"); %]
// protected region anId [on|off] begin
System.out.println(foo);
// protected region anId end
```

When a template that defines one or more protected regions is processed by the EGL execution engine, the target output destinations are interrogated and existing contents of any protected regions are preserved. If either the output generated by from the template or the existing contents of the target output destination contains protected regions, a merging process is invoked. Table 7.1 shows the default behaviour of EGL's merge engine.

7.7 Readability and traceability

Conscientious developers apply various *conventions* to produce readable code. EGL encourages template developers to prioritise the readability of templates over the text that they generate. EGL provides a number of text post-processors – or *beautifiers* – that can be executed on output of transformations to improve readability. Currently, beautifiers are invoked via Epsilon's extensions to Apache Ant, an XML-based build tool for Java.

Protected Region Status		Contents taken from
Generated	Existing	
On	On	Existing
On	Off	Generated
On	Absent	Generated
Off	On	Existing
Off	Off	Generated
Off	Absent	Generated
Absent	On	Neither (causes a warning)
Absent	Off	Neither (causes a warning)

Table 7.1: EGL's default merging behaviour.

EGL also provides a traceability API, as a debugging aid, and to support auditing of the M2T transformation process. This API facilitates exploration of the templates executed, files affected and protected regions processed during a transformation. Figure 7.2 shows sample output from the traceability API after execution of an EGL M2T transformation to generate Java code from an instance of an OO metamodel.

The beautification interface is minimal, in order to allow re-use of existing code formatting algorithms. Consequently, there is presently no traceability support for beautified text. However, due to the coarse-grained approach employed by EGL's traceability API, this has little impact: Clicking on a beautified protected region in the traceability view might not highlight the correct line in the editor.

7.7.1 Tool Support

The Epsilon platform provides development tools for the Eclipse development environment. Re-use of Eclipse APIs allows Epsilon's development tooling to incorporate a large number of features with minimal effort. Furthermore, the flexibility of the plug-in architecture of Eclipse enhances

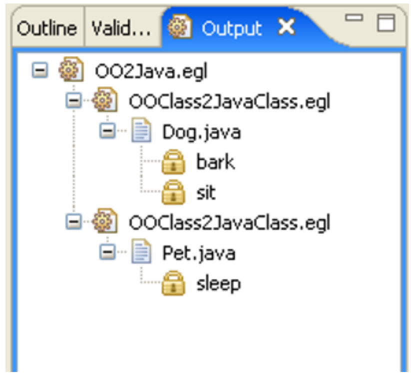

Figure 7.2: Sample output from the traceability API.

modular authoring of development tools for Epsilon.

In addition to the traceability view shown in Figure 7.2, EGL includes an Eclipse editor and an outline view. In order to aid template readability, these tools provide syntax highlighting and a structural overview for EGL templates, respectively. Through its integration in the Epsilon perspective, EGL provides an Eclipse workbench configuration that is tailored for use with Epsilon's development tools.

EGL, like other Epsilon languages, provides an Apache ANT task definition, to facilitate invocation of model-management activies from within a build script.

Chapter 8

The Epsilon Comparison Language (ECL)

Model comparison is the task of identifying *matching* elements between models. In general, *matching* elements are elements that are involved in a relationship of interest. For example, before merging homogeneous models, it is essential to identify overlapping (common) elements so that they do not appear in duplicate in the merged model. Similarly, in heterogeneous model merging, it is a prerequisite to identify the elements on which the two models will be merged. Finally, in transformation testing, matching elements are pairs consisting of elements in the input model and their generated counterparts in the output model.

The aim of the Epsilon Comparison Language (ECL) is to enable users to specify comparison algorithms in a rule-based manner to identify pairs of matching elements between two models of potentially different metamodels and modelling technologies. In this section, the abstract and concrete syntax, as well as the execution semantics of the language, are discussed in detail.

8.1 Abstract Syntax

In ECL, comparison specifications are organized in modules (*EcLModule*). As illustrated in Figure 8.1, EclModule extends EOLLibraryModule which means that it can contain user-defined operations and import other library modules and ECL modules. Apart from operations, an ECL module contains a set of match-rules (*MatchRule*) and a set of *pre* and *post* blocks.

MatchRules enable users to perform comparison of model elements at a high level of abstraction. Each match-rule declares a name, and two parameters (*leftParameter* and *rightParameter*) that specify the types of elements it can compare. It also optionally defines a number of rules it inherits (*extends*) and if it is *abstract*, *lazy* and/or *greedy*. The semantics of the latter are discussed shortly.

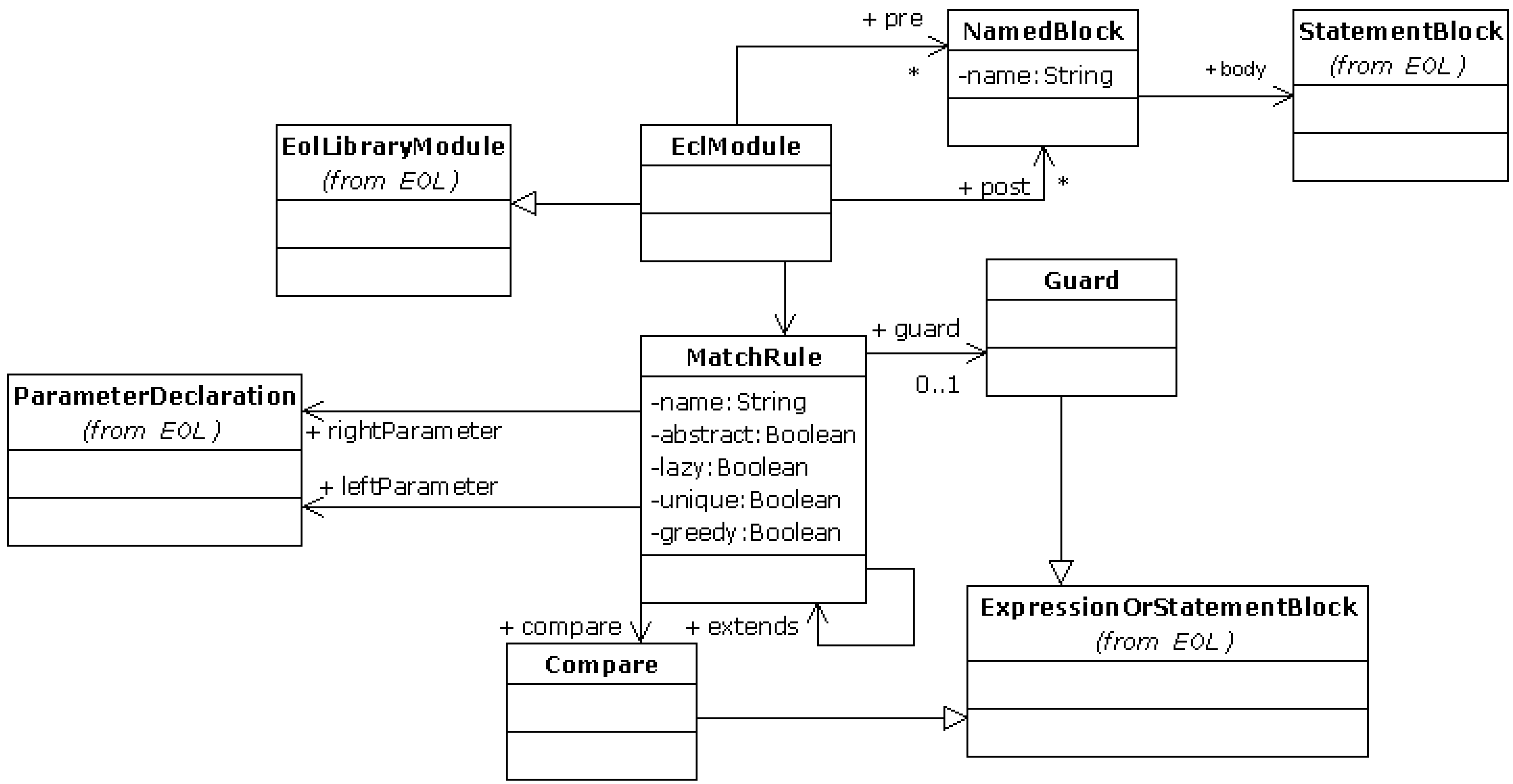

Figure 8.1: ECL Abstract Syntax

A match rule has three parts. The *guard* part is an EOL expression or statement block that further limits the applicability of the rule to an even narrower range of elements than that specified by the *left* and *right* parameters. The *compare* part is an EOL expression or statement block that is responsible for comparing a pair of elements and deciding if they match or not. Finally, the *do* part is an EOL expression or block that is executed if the *compare* part returns true to perform any additional actions required.

Pre and *Post* blocks are named blocks of EOL statements which as discussed in the sequel are executed before and after the match-rules have been executed respectively.

8.2 Concrete Syntax

The concrete syntax of a match-rule is displayed in Listing 8.1.

Listing 8.1: Concrete Syntax of a MatchRule

```

(@lazy)?
(@greedy)?
(@abstract)?
rule <name>
 match <leftParameterName>:<leftParameterType>
 with <rightParameterName>:<rightParameterType>
 (extends (<ruleName>,)*<ruleName>)? {

 (guard (:expression)|({statementBlock}))?

 compare (:expression)|({statementBlock})

 (do {statementBlock})?

}
```

Pre and *post* blocks have a simple syntax that, as presented in Listing 8.2, consists of the identifier (*pre* or *post*), an optional name and the set of statements to be executed enclosed in curly braces.

Listing 8.2: Concrete Syntax of Pre and Post blocks

```
(pre|post) <name> {
 statement+
}
```

8.3 Execution Semantics

8.3.1 Rule and Block Overriding

An ECL module can import a number of other ECL modules. In such a case, the importing ECL module inherits all the rules and pre/post blocks specified in the modules it imports (recursively). If the module specifies a rule or a pre/post block with the same name, the local rule/block overrides the imported one respectively.

8.3.2 Comparison Outcome

As illustrated in Figure 8.2, the result of comparing two models with ECL is a trace (*MatchTrace*) that consists of a number of matches (*Match*). Each match holds a reference to the objects from the two models that have been compared (*left* and *right*), a boolean value that indicates if they have been found to be *matching* or not, a reference to the *rule* that has made the decision, and a Map (*info*) that is used to hold any additional information required by the user (accessible at runtime through the *matchInfo* implicit variable). During the matching process, a second, temporary, match trace is also used to detect and resolve cyclic invocation of match-rules as discussed in the sequel.

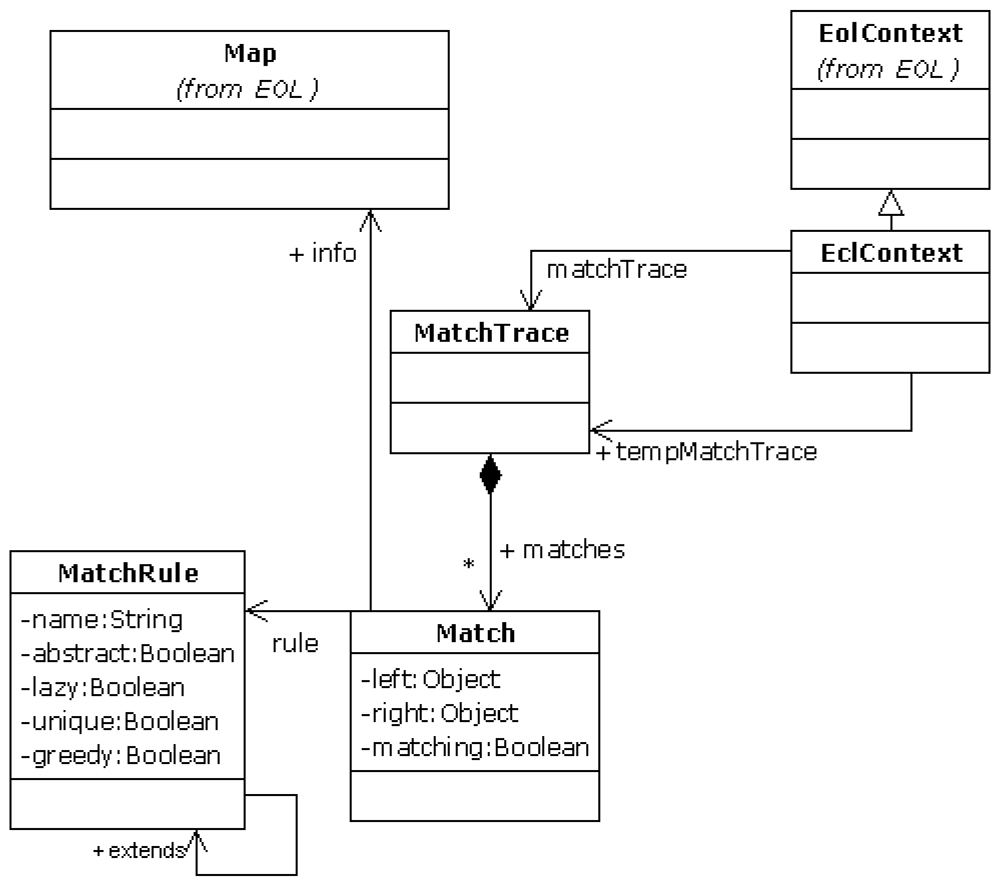

Figure 8.2: ECL Match Trace

8.3.3 Rule Execution Scheduling

Non-abstract, non-lazy match-rules are evaluated automatically by the execution engine in a top-down fashion - with respect to their order of appearance - in two passes. In the first pass, each rule is evaluated for all the pairs of instances in the two models that have a type-of relationship with the types specified by the *leftParameter* and *rightParameter* of the rule. In the second pass, each rule that is marked as *greedy* is executed for all pairs that have not been compared in the first pass, and which have a kind-of relationship with the types specified by the rule. In both passes, to evaluate the compare part of the rule, the guard must be satisfied.

Before the compare part of a rule is executed, the compare parts of all of the rules it extends (super-rules) must be executed (recursively). Before executing the compare part of a super-rule, the engine verifies that the super-rule is actually applicable to the elements under comparison by checking for type conformance and evaluating the guard part of the super-rule.

If the compare part of a rule evaluates to true, the optional do part is executed. In the do part the user can specify any actions that need to be performed for the identified matching elements, such as to populate the *info* map of the established *match* with additional information. Finally, a new match is added to the match trace that has its *matching* property set to the logical conjunction of the results of the evaluation of the compare parts of the rule and its super-rules.

8.3.4 The *matches()* built-in operation

To refrain from performing duplicate comparisons and to de-couple match-rules from each other, ECL provides the built-in *matches(opposite : Any)* operation for model elements and collections. When the *matches()* operation is invoked on a pair of objects, it queries the main and temporary match-traces to discover if the two elements have already been matched and if so it returns the cached result of the comparison. Otherwise, it attempts to find an appropriate match rule to compare the two elements and if such a rule is found, it returns the result of the comparison, otherwise it returns false. Unlike the top-level execution scheme, the *matches()* operation invokes both *lazy* and *non-lazy* rules.

In addition to objects, the *matches* operations can also be invoked to match pairs of collections of the same type (e.g. a Sequence against a Sequence). When invoked on ordered collections (i.e. *Sequence* and *Or-*

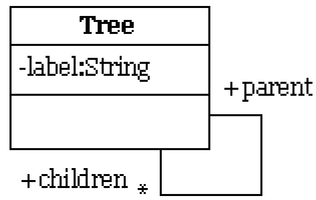

Figure 8.3: The Tree Metamodel

deredSet), it examines if the collections have the same size and each item of the source collection matches with the item of the same index in the target collection. Finally, when invoked on unordered collections (i.e. *Bag* and *Set*), it examines if for each item in the source collection, there is a matching item in the target collection irrespective of its index. Users can also override the built-in *matches* operation using user-defined operations with the same name, as discussed in Section 3.4.1, that loosen or strengthen the built-in semantics.

8.3.5 Cyclic invocation of *matches()*

Providing the built-in *matches* operation significantly simplifies comparison specifications. It also enhances decoupling between match-rules from each other as when a rule needs to compare two elements that are outside its scope, it does not need to know/specify which other rule can compare those elements explicitly.

On the other hand, it is possible - and quite common indeed - for two rules to implicitly invoke each other. For example consider the match rule of Listing 8.3 that attempts to match nodes of the simple Tree metamodel displayed in Figure 8.3.

Listing 8.3: The Tree2Tree rule

```
rule Tree2Tree
 match l : T1!Tree
```

```
 with r : T2!Tree {

 compare : l.label = r.label and
   l.parent.matches(r.parent) and
   l.children.matches(r.children)
}
```

The rule specifies that for two Tree nodes (*l* and *r*) to match, they should have the same label, belong to matching parents and have matching children. In the absence of a dedicated mechanism for cycle detection and resolution, the rule would end up in an infinite loop. To address this problem, ECL provides a temporary match-trace which is used to detect and resolve cyclic invocations of the *match()* built-in operation.

As discussed above, a match is added to the primary match-trace as soon as the compare part of the rule has been executed to completion. By contrast, a temporary match (with its *matching* property set to *true*) is added to the temporary trace before the compare part is executed. In this way, any subsequent attempts to match the two elements from invoked rules will not re-invoke the rule. Finally, when a top-level rule returns, the temporary match trace is reset.

8.4 Fuzzy and Dictionary-based String Matching

In the example of Listing 8.3, the rule specifies that to match, two trees must - among other criteria - have the same label. However, there are cases when a less-strict approach to matching string properties of model elements is desired. For instance, when comparing two UML models originating from different organizations, it is common to encounter ontologically equivalent classes which however have different names (e.g. Client and Customer). In this case, to achieve a more sound matching, the use of a dictionary or a lexical database (e.g. WordNet [11]) is necessary. Alternatively, fuzzy string matching algorithms such as those presented in [12] can be used.

As several such tools and algorithms have been implemented in various programming languages, it is a sensible approach to reuse them instead of re-implementing them. For example, in Listing 8.4 a wrapper for the Simmetrics [13] fuzzy string comparison tool is used to compare the labels of the trees using the Levenshtein [14] algorithm. To achieve this, line 11 invokes the *fuzzyMatch()* operation defined in lines 16-18 which uses the simmterics native tool (instantiated in lines 2-4) to match the two labels using their Levenshtein distance with a threshold of 0.5.

Listing 8.4: The FuzzyTree2Tree rule

```
1  pre {
2   var simmetrics =
3    new Native("org.epsilon.ecl.tools.
4     textcomparison.simmetrics.SimMetricsTool");
5  }
6
7  rule FuzzyTree2Tree
8   match l : T1!Tree
9   with r : T2!Tree {
10
11  compare : l.label.fuzzyMatch(r.label) and
12   l.parent.matches(r.parent) and
13   l.children.matches(r.children)
14 }
15
16 operation String fuzzyMatch(other : String) : Boolean {
17  return simmetrics.similarity(self,other,"Levenshtein") > 0.5;
18 }
```

8.5 Interactive Matching

Using the user interaction features discussed in Section 3.7 the comparison can become interactive by replacing the *fuzzyMatch* operation of listing 8.4 with the one specified in Listing 8.5. The fuzzyMatch operation of Listing 8.5, performs the fuzzy string comparison and – as the previous version – if the result is greater than 0.5 it returns true. However, in this updated version if the result is lower than 0.5 but greater than 0.3, it prompts the user to confirm if the two strings match, and if it is lower than 0.3 it returns false.

Listing 8.5: An interactive version of the fuzzyMatch operation of Listing 8.4

```
operation String fuzzyMatch(other : String) : Boolean {
  var similarity : Real;
  similarity = simmetrics.similarity(self,other,"Levenshtein");
  if (similarity > 0.5) {
    return true;
  }
  else if (similarity > 0.3) {
    return UserInput.confirm(self + " matches " + other + "?");
  }
  else {
    return false;
  }
}
```

8.6 Exploiting the Comparison Outcome

Users can query and modify the match trace calculated during the comparison process in the post sections of the module or export it into another application or Epsilon program. For example, in a post section, the trace can be printed to the default output stream or serialized into a model of an arbitrary metamodel. In another use case, the trace may be exported to be used in the context of a validation module that will use the identified matches to evaluate inter-model constraints, or in a merging module that will use the matches to identify the elements on which the two models will be merged. The topic of interoperability - that includes importing and exporting objects - between modules expressed in different Epsilon languages is discussed in Chapter 11.

Chapter 9

The Epsilon Merging Language (EML)

The aim of EML is to contribute model merging capabilities to Epsilon. More specifically, EML can be used to merge an arbitrary number of input models of potentially diverse metamodels and modelling technologies. This section provides a discussion on the motivation for implementing EML, its abstract and concrete syntax, as well as its execution semantics. It also provides two examples of merging homogeneous and heterogeneous models.

9.1 Motivation

A mechanism that enables automatically merging models on a set of established correspondences has a number of applications in a model driven engineering process. For instance, it can be used to unify two complementary, but potentially overlapping, models that describe different views of the same system. In another scenario, it can be used to merge a core model with an aspect model (potentially conforming to different metamodels), as discussed in [15] where a core *Platform Independent Model (PIM)* is merged with a *Platform Definition Model (PDM)*, that contributes platform-specific aspects, into a *Platform Specific Model (PSM)*.

9.1.1 Phases of Model Merging

Existing research [16, 17] has demonstrated that model merging can be decomposed into four distinct phases: comparison, conformance checking, merging and reconciliation (or restructuring).

Comparison Phase In the comparison phase, correspondences between equivalent elements of the source models are identified, so that such elements are not propagated in duplicate in the merged model.

Conformance Checking Phase In this phase, elements that have been identified as matching in the previous phase are examined for conformance with each other. The purpose of this phase is to identify potential conflicts that would render merging infeasible. The majority of proposed approaches, such as [18], address conformance checking of models complying with the same metamodel.

Merging Phase Several approaches have been proposed for the merging phase. In [16, 19], graph-based algorithms for merging models of the same metamodel are proposed. In [18], an interactive process for merging of UML 2.0 models is presented. There are at least two weaknesses in the methods proposed so far. First, they only address the issue of merging models of the same metamodel, and some of them address a specific metamodel indeed. Second, they use an inflexible merging algorithm and do not provide means for extending or customizing its logic.

Reconciliation and Restructuring Phase After the merging phase, the target model may contain inconsistencies that need fixing. In the final step of the process, such inconsistencies are removed and the model is *polished* to acquire its final form. Although the need for a reconciliation phase is discussed in [17, 19], in the related literature the subject is not explicitly targeted.

9.1.2 Relationship between Model Merging and Model Transformation

A merging operation is a transformation in a general sense, since it transforms some input (source models) into some output (target models). However, as discussed throughout this section, a model merging facility has special requirements (support for comparison, conformance checking and merging pairs of input elements) that are not required for typical *one-to-one* or *one-to-many* transformations [8] and are therefore not supported by contemporary model transformation languages.

9.2 Realizing a Model Merging Process with Epsilon

The first two steps of the process described above can be realized with existing languages provided by Epsilon. As discussed in Section 8, the comparison step can be realized with the Epsilon Comparison Language (ECL). Following that, the Epsilon Validation Language (EVL) can be used to validate the identified correspondences using the match trace calculated by ECL. The Epsilon Merging Language (EML) presented below provides support for the last two steps of the process (merging and reconciliation/restructuring).

9.3 Abstract Syntax

In EML, merging specifications are organized in modules (*EmlModule*). As displayed in Figure 9.1, *EmlModule* inherits from *EtlModule*.

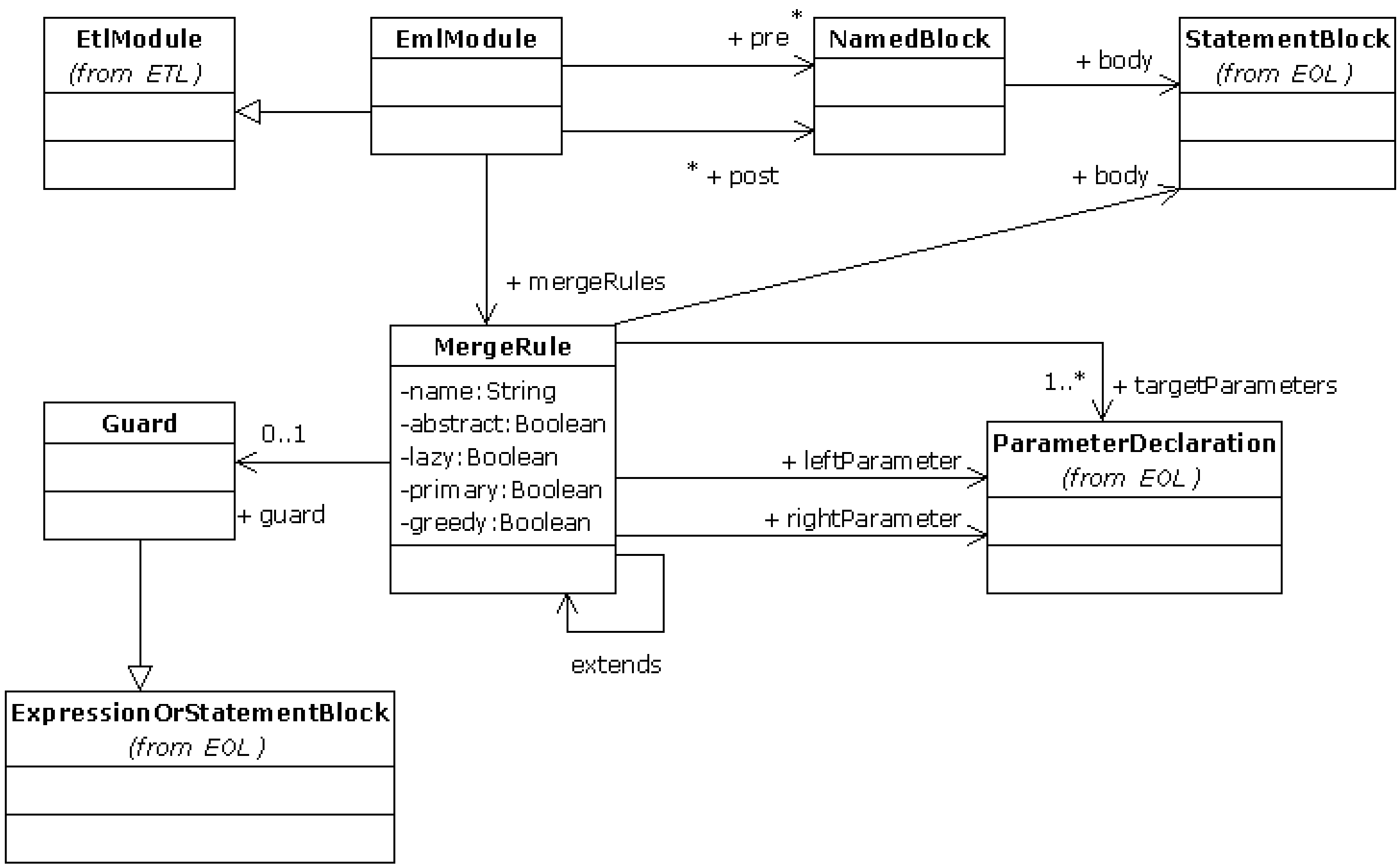

Figure 9.1: The Abstract Syntax of EML

By extending *EtlModule*, an EML module can contain a number of transformation rules and user-defined operations. An EML module can also contain one or more merge rules as well as a set of pre and post named statement blocks.

Each merge rule defines a name, a left, a right, and one or more target parameters. It can also extend one or more other merge rules and be defined as having one or more of the following properties: abstract, greedy, lazy and primary.

9.4 Concrete Syntax

Listing 9.1 demonstrates the concrete syntax of EML merge-rules.

Listing 9.1: Concrete syntax of an EML merge-rule

```
(@abstract)?
(@lazy)?
(@primary)?
(@greedy)?
rule <name>
 merge <leftParameter>
 with <rightParameter>
 into (<targetParameter>(, <targetParameter>)*)?
 (extends <ruleName>(, <ruleName>)*)? {

 statementBlock

}
```

9.5 Execution Semantics

9.5.1 Rule and Block Overriding

An EML module can import a number of other EML and ETL modules. In this case, the importing EML module inherits all the rules and pre/post blocks specified in the modules it imports (recursively). If the module specifies a rule or a pre/post block with the same name, the local rule/block overrides the imported one respectively.

9.5.2 Rule Scheduling

When an EML module is executed, the *pre* blocks are executed in the order in which they have been defined.

Following that, for each *match* of the established *matchTrace* the applicable non-abstract, non-lazy merge rules are executed. When all *matches* have been merged, the transformation rules of the module are executed on all applicable elements - that have not been merged - in the models.

Finally, after all rules have been applied, the *post* blocks of the module are executed.

9.5.3 Rule Applicability

By default, for a merge-rule to apply to a *match*, the *left* and *right* elements of the match must have a *type-of* relationship with the *leftParameter* and *rightParameter* of the rule respectively. This can be relaxed to a *kind-of* relationship by specifying that the merge rule is *greedy* (using the *@greedy* annotation in terms of concrete syntax).

9.5.4 Source Elements Resolution

As with model transformation, in model merging it is often required to resolve the counterparts of an element of a source model into the target models. In EML, this is achieved by overloading the semantics of the *equivalents()* and *equivalent()* operations defined by ETL. In EML, in addition to inspecting the transformation trace and invoking any applicable transformation rules, the *equivalents()* operation also examines the *mergeTrace* (displayed in Figurer 9.2) that stores the results of the application of merge-rules and invokes any applicable (both lazy and non-lazy) rules.

Similarly to ETL, the order of the results of the *equivalents()* operation respects the order of the (merge or transform) rules that have produced them. An exception to that occurs if one of the rules has been declared as primary, in which case its results are prepended to the list of elements returned by equivalent.

9.6 Homogeneous Model Merging Example

In this scenario, two models conforming to the Graph metamodel need to be merged. The first step is to compare the two graphs using the ECL module of Listing 9.2.

Listing 9.2: ECL module for comparing two instances of the Graph metamodel

```
rule MatchNodes
 match l : Left!Node
 with r : Right!Node {

 compare : l.label = r.label
}

rule MatchEdges
 match l : Left!Edge
```

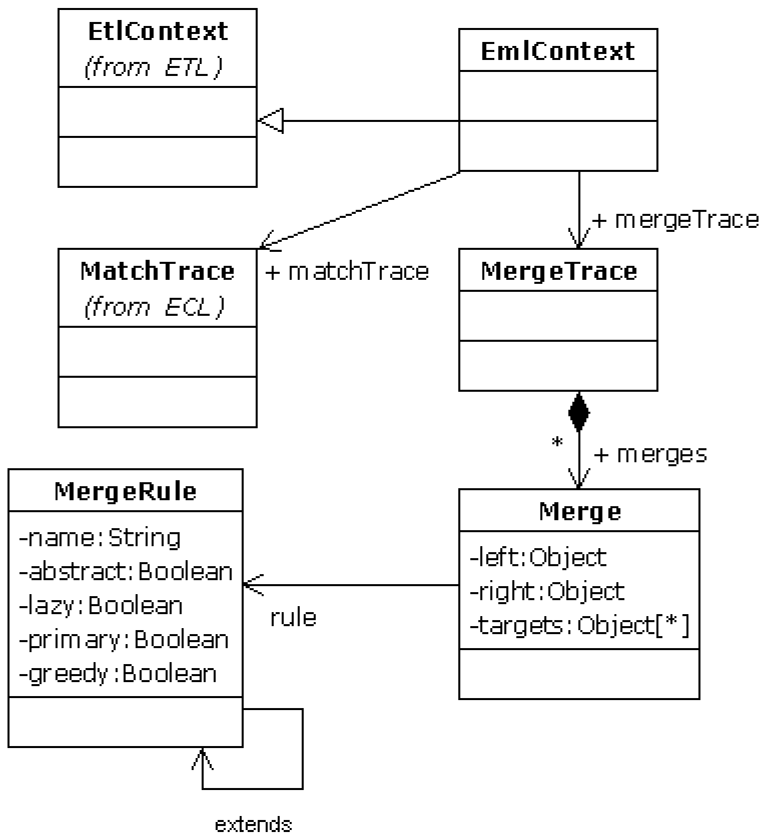

Figure 9.2: The EML runtime

```
  with r : Right!Edge {

  compare : l.source.matches(r.source)
    and l.target.matches(r.target)
}

rule MatchGraphs
  match l : Left!Graph
  with r : Right!Graph {

  compare : true
}
```

The *MatchNodes* rule in line 1 defines that two nodes match if they have the same label. The *MatchEdges* rule in line 8 specifies that two edges match if both their source and target nodes match (regardless of whether the labels of the edges match or not as it is assumed that there can not be two distinct edges between the same nodes). Finally, since only one instance of Graph is expected to be in each model, the *MatchGraphs* rule in line 16 returns *true* for any pair of Graphs[1].

Having established the necessary correspondences between matching elements of the two models, the EML specification of listing 9.3.

Listing 9.3: EML module for merging two instances of the Graph metamodel on the correspondences identified in Listing 9.2

```
import "Graphs.etl";

rule MergeGraphs
 merge l : Left!Graph
 with r : Right!Graph
 into t : Target!Graph {

 t.label = l.label + " and " + r.label;

}

@abstract
rule MergeGraphElements
 merge l : Left!GraphElement
 with r : Right!GraphElement
 into t : Target!GraphElement {

 t.graph ::= l.graph;

}

```

[1]Both assumptions can be checked using EVL before matching/merging takes place but this is out of the scope of this example

```
22 rule MergeNodes
23  merge l : Left!Node
24  with r : Right!Node
25  into t : Target!Node
26  extends GraphElements {
27
28  t.label = "c_" + l.label;
29
30 }
31 rule MergeEdges
32  merge l : Left!Edge
33  with r : Right!Edge
34  into t : Target!Edge
35  extends GraphElements {
36
37  t.source ::= l.source;
38  t.target ::= l.target;
39
40 }
```

In line 3, the *MergeGraphs* merge rule specifies that two matching Graphs (*l* and *r*) are to be merged into one Graph *t* in the target model that has as a label, the concatenation of the labels of the two input graphs separated using 'and'. The Nodes merge rule In line 22 specifies that two matching Nodes are merged into a single Node in the target model. The label of the merged node is derived by concatenating the c (for common) static string with the label of the source Node from the left model. Similarly, the *MergeEdges* rule specifies that two matching Edges are merged into a single Edge in the target model. The source and target nodes of the merged Edge are set to the equivalents (::=) of the source and target nodes of the edge from the left model.

To reduce duplication, the *MergeNodes* and *MergeEdges* rules extend the abstract *MergeGraphElements* rule specified in line 13 which assigns the *graph* property of the graph element to the equivalent of the left graph.

The rules displayed in Listing 9.3 address only the matching elements of the two models. To also copy the elements for which no equivalent has been found in the opposite model, the EML module imports the ETL module of Listing 9.4.

Listing 9.4: The Graphs.etl ETL transformation module

```
rule TransformGraph
 transform s : Source!Graph
 to t : Target!Graph {

 t.label = s.label;

}

@abstract
rule TransformGraphElement
 transform s : Source!GraphElement
 to t : Target!GraphElement {

 t.graph ::= s.graph;
}

rule TransformNode
 transform s : Source!Node
 to t : Target!Node
 extends TransformGraphElement {

 t.label = s.graph.label + "_" + s.label;
}

rule TransformEdge
 transform s : Source!Edge
 to t : Target!Edge
 extends TransformGraphElement {

 t.source ::= s.source;
```

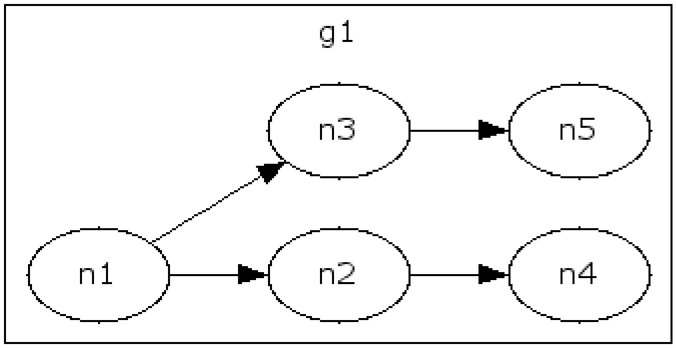

Figure 9.3: Left input model

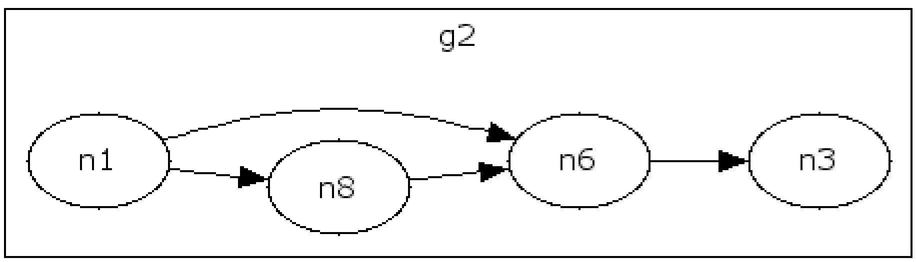

Figure 9.4: Right input model

```
31   t.target ::= s.target;
32  }
```

The rules of the ETL module apply to model elements of both the Left and the Right model as both have been aliased as Source. Of special interest is the TransformNode rule in line 17 that specifies that non-matching nodes in the two input models will be transformed into nodes in the target model the labels of which will be a concatenation of their input graph and the label of their counterparts in the input models.

Executing the ECL and EML modules of Listings 9.2 and 9.3 on the exemplar models displayed in Figures 9.3 and 9.4 creates the target model of Figure 9.5.

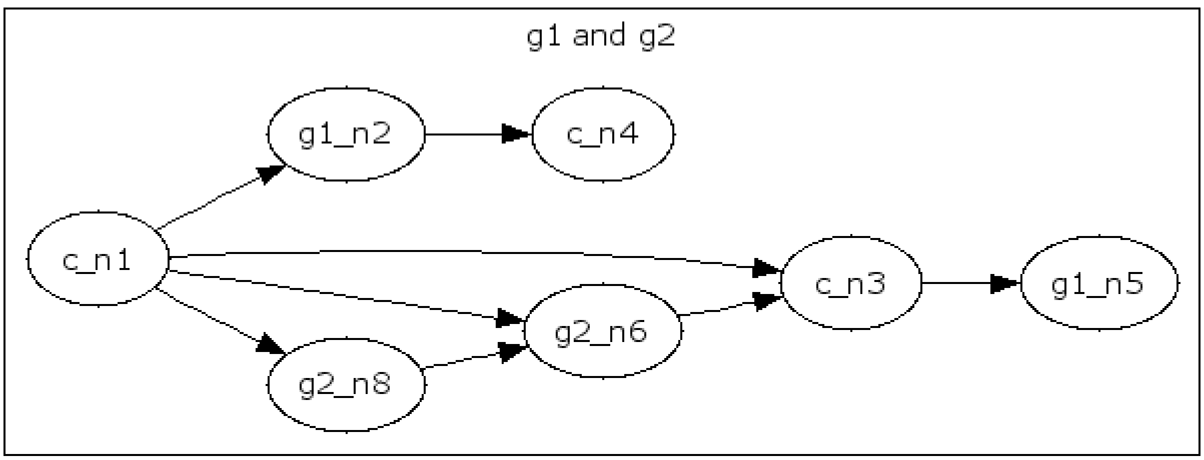

Figure 9.5: Target model derived by merging the models of Figures 9.3 and 9.4

Chapter 10

Implementing a New Task-Specific Language

Although Epsilon already provides languages for a wide range of model management tasks, additional tasks that could benefit from the convenience syntax and dedicated semantics of a task-specific language are likely to be identified in the future. Thus, this section distils the experiences obtained through the construction of existing task-specific languages to provide guidance on how to identify a task for which a dedicated language can be beneficial and develop the respective task-specific language for it atop the infrastructure provided by Epsilon.

10.1 Identifying the need for a new language

The first step of the process of constructing a new task-specific language is to identify a specific task for which a dedicated language is more appropriate than the general-purpose EOL. Typically, recurring syntactic and semantic patterns that emerge when attempting to implement the task using EOL indicate that a new task-specific language may be useful.

For example, before the introduction of the Epsilon Comparison Language, pure EOL was being used to perform model comparison. A simple comparison specification that establishes name-based matches between classes/attributes and tables/columns between two OO and DB models respectively using EOL is demonstrated in Listing 10.1.

Two patterns can be readily detected by inspecting the EOL code in Listing 10.1. First, explicit variables (*matchingCT*, *matchingAT*) are defined to capture the matching elements (class-table and attribute-column)identified during the comparison process. Also, to check all elements of one type (classes against tables and attributes against columns) repeated for statements are used in lines 3–4 and 7–8. By contrast, Listing 10.2 which is specified using the task-specific ECL language does not include such low-level information. Instead it defines only the types of elements that need to be compared and the criteria on which comparison must performed and leaves the mundane tasks of scheduling and maintaining the match trace to the execution engine.

Listing 10.1: Comparing an OO model with a DB model using EOL

```
1  var matchingCT : Sequence;
2  var matchingAC : Sequence;
3  for (c in OO!Class.allInstances) {
4   for (t in DB!Table.allInstances) {
5    if (t.name = c.name) {
6     matchingCT.add(Sequence{c,t});
7     for (att in c.attributes) {
8      for (col in t.columns) {
9       if (att.name = c.name) {
10       matchingAC.add(Sequence{att, col});
11      }
12     }
13    }
14   }
15  }
16 }
```

Listing 10.2: Comparing an OO model with a DB model using ECL

```
rule ClassTable
 match c : OO!Class
 with t : DB!Table {

 compare : c.name = t.name
}

rule AttributeColumn
 match a : OO!Attribute
 with c : DB!Column {

 compare : a.name = c.name and
   a.class.matches(c.table)
}
```

10.2 Eliciting higher-level constructs from recurring patterns

Once recurring patterns, such as those discussed above, have been identified, the next step of the process is to derive higher level constructs from them. For instance, in the previous example, the nested for loops and the explicit trace variable declaration and population have been replaced by task-specific match rules.

Introducing higher-level involves defining its abstract and concrete syntax as well as its connection points with the underlying infrastructure. For example, in the case of ECL, the types of match rules are EOL model element types, the *guard* and *check* parts of a rule are EOL expressions or statements blocks and the *pre* and *post* blocks as well as the *do* part of each rule are blocks of EOL statements.

10.3 Implement Execution Semantics and Scheduling

Once higher-level constructs (e.g. task-specific rules) have been identified and specified, their execution semantics and scheduling must be implemented similarly to what has been done for existing languages. Development of existing languages has demonstrated that task-specific constructs often need to provide more than one modes of execution (e.g. the *lazy* and *greedy* modes of ETL transformation rules discussed in Section 5.5).

A lightweight way to easily provide new execution modes and semantics for rules and user-defined operations without modifying the syntax of the language and introducing new keywords that may conflict with existing code, is through the annotations mechanism provided by EOL (see Section 3.2.1). This approach has been adopted for the definition a small unit-testing language (EUnit), which is discussed in detail in [20].

10.4 Overriding Semantics

In certain cases, it is useful to modify the semantics of certain constructs in EOL to meet the purposes of the task-specific language. An example of such a modification occurs in EVL where – as discussed in Section 4.4 – the scope of the variables defined in *guard* expression/block is extended so that variables can be reused in the context of non-nested blocks such as the *title*, and *check* parts of the invariant. Another example of overriding the semantics of EOL is the implementation of the special assignment operator (::=) by ETL which was discussed in 5.5.4.

Chapter 11

Orchestration Workflow

The previous chapter has provided a detailed discussion on a number of task-specific languages, each one addressing an individual model management task. However, in practice, model management tasks are seldom carried out in isolation; instead, they are often combined together to form complex workflows. Therefore, of similar importance to the existence of individual task-specific management languages is the provision of a mechanism that enables developers to compose modular and reusable tasks into complex automated processes. In a broader context, to facilitate implementation of seamless workflows, an appropriate MDE workflow mechanism should also support mainstream development tasks such as file management, version control management, source code compilation and invocation of external programs and services.

11.1 Motivation

As a motivating example, an exemplar workflow that consists of both MDD tasks (1-4, 6) and mainstream software development tasks (5, 7) is displayed below.

1. Load a UML model
2. Validate it
3. Transform it into a Database Schema model
4. Generate Java code from the UML model
5. Compile the Java code
6. Generate SQL code from the Database model
7. Deploy the SQL code in a Database Management System (DBMS)

In the above workflow, if the validation step (2) fails, the entire process should be aborted and the identified errors should be reported to the user. This example demonstrates that to be of practical use, a task orchestration framework needs to be able to coordinate both model management and mainstream development tasks and provide mechanisms for establishing dependencies between different tasks.

This chapter presents such a framework for orchestrating modular model management tasks implemented using languages of the Epsilon platform. As the problem of task coordination is common in software development, many technical solutions have been already proposed and are widely used by software practitioners. In this context, designing a new general-purpose workflow management solution was deemed inappropriate. Therefore, the task orchestration solution discussed here has been designed as an extension to the robust and widely used ANT [21] framework. A brief overview of ANT as well as a discussion on the choice to design the orchestration workflow of Epsilon atop it is provided below.

11.2 The ANT Tool

ANT, named so because it is *a little thing that can be used to build big things* [22], is a robust and widely-used framework for composing automated workflows from small reusable activities. The most important advantages of ANT, compared to traditional build tools such as *gnumake* [23], is that it is platform independent and easily extensible. Platform independence is achieved by building atop Java, and extensibility is realized through a lightweight binding mechanism that enables developers to contribute custom tasks using well defined interfaces and extension points.

Although a number of tools with functionality similar to ANT exist in the Java community, only Maven [24] is currently of comparable magnitude in terms of user-basis size and robustness. Outlining the discussion provided in [25], ANT is considered to be easier to learn and to enable low-level control, while Maven is considered to provide a more elaborate task organization scheme. Nevertheless, the two frameworks are significantly similar and the ANT technical solution discussed in this chapter can easily be ported to work with the latter.

This section provides a brief discussion of the structure and concrete syntax of ANT workflows, as well as the extensibility mechanisms that ANT provides to enable users contribute custom tasks.

11.2.1 Structure

In ANT, each workflow is captured as a *project*. A simplified illustration of the structure of an ANT project is displayed in Figure 11.1. Each ANT project consists of a number of *targets*. The one specified as the *default* is executed automatically when the project is executed. Each *target* contains a number of *tasks* and *depends* on other targets that must be executed before it. An ANT task is responsible for a distinct activity and can either succeed or fail. Exemplar activities implemented by ANT tasks include file system

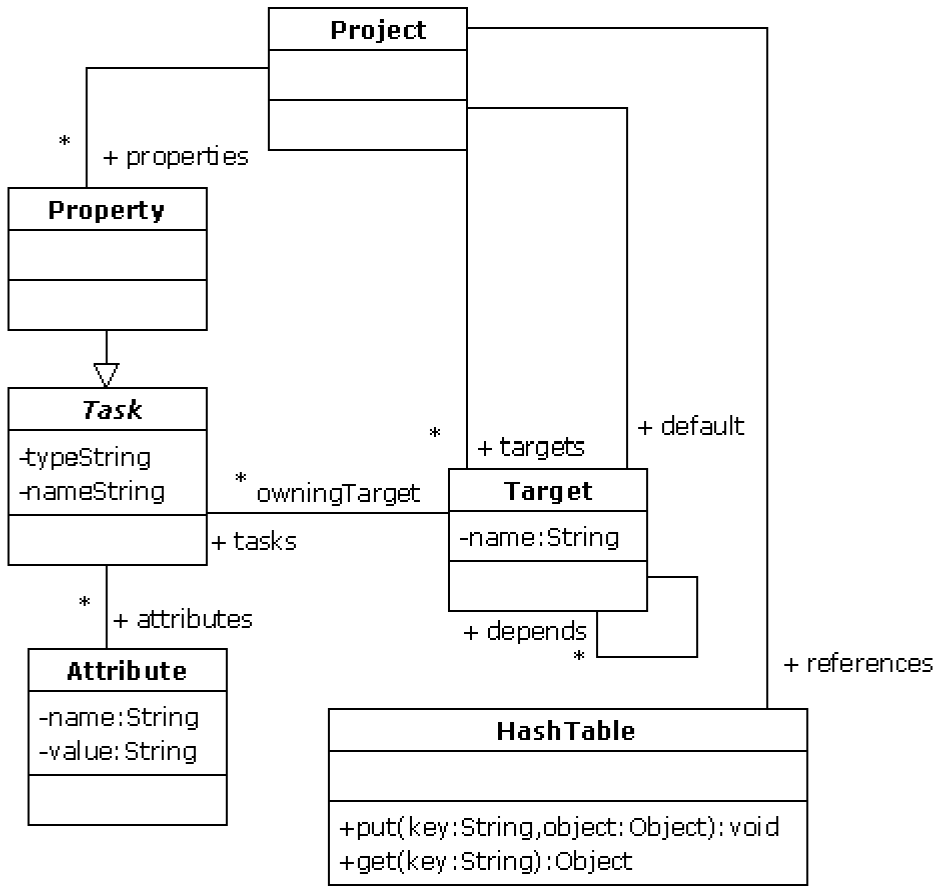

Figure 11.1: Simplified ANT object model

management, compiler invocation, version management and remote artefact deployment.

11.2.2 Concrete Syntax

In terms of concrete syntax, ANT provides an XML-based syntax. In Listing 11.1, an exemplar ANT project that compiles a set of Java files is illustrated. The project contains one target (*main*) which is also set to be the *default* target. The *main* target contains one *javac* task that specifies attributes such as *srcdir*, *destdir* and *classpath*, which define that the Java compiler will compile a set of Java files contained into the *src* directory into classes that should be placed in the *buiild* directory using *dependencies.jar* as an external library.

Listing 11.1: Compiling Java classes using the javac task

```
<project default="main">
      <target name="main"/>
        <javac srcdir="${src}"
             destdir="${build}"
             classpath="dependencies.jar"
             debug="on"
             source="1.4"/>
      </target>
</project>
```

11.2.3 Extending ANT

Binding between the XML tags that describe the tasks and the actual implementations of the tasks is achieved through a light-weight mechanism at two levels. First, the tag (in the example of Listing 11.1, *javac*) is resolved to a Java class that extends the *org.apache.ant.Task* abstract class (in the case of *javac*, the class is *org.apache.tools.ant.taskdefs.Javac*) via a configuration file. Then, the attributes of the tasks (e.g. *srcdir*) are set using the reflective features that Java provides. Finally, the *execute()* method of the task is invoked to perform the actual job.

This lightweight and straightforward way of defining tasks has rendered ANT particularly popular in the Java development community and currently there is a large number of tasks contributed by ANT users [26], ranging from invoking tools such as code generators and XSLT processors, to emulating logical control flow structures such as *if* conditions and *while* loops. The AMMA platform [27] also provides integration of model driven engineering tools such as TCS [28] and ATL [6] with ANT.

ANT also supports more advanced features including nested XML elements and *filesets*, however providing a complete discussion is beyond the scope of this paper. For a definitive guide to ANT readers can refer to [22].

11.3 Integration Challenges

A simple approach to extending ANT with support for model management tasks would be to implement one standalone task for each language in Epsilon. However, such an approach demonstrates a number of integration and performance shortcomings which are discussed below.

Since models are typically serialized in the file system, before a task is executed, the models it needs to access/modify must be parsed and loaded in memory. In the absence of a more elaborate framework, each model management task would have to take responsibility for loading and storing the models it operates on. Also, in most workflows, more than one task operates on the same models sequentially, and needlessly loading/storing the same models many times in the context of the same workflow is an expensive operation both time and memory-wise, particularly as the size of models increases.

Another weakness of this primitive approach is limited inter-task communication. In the absence of a communication framework that allows model management tasks to exchange information with each other, it is often the case that many tasks end up performing the same (potentially expensive) queries on models. By contrast, an inter-task communication framework would enable time and resource intensive calculations to be performed once and their results to be communicated to all interested subsequent tasks.

Having discussed ANT, Epsilon and the challenges their integration poses, the following sections presents the design of a solution that enables developers to invoke model management tasks in the context of ANT workflows. The solution consists of a core framework that addresses the challenges discussed in Section 11.3, a set of specific tasks, each of which implements a distinct model management activity, and a set of tasks that enable developers to initiate and manage transactions on models using the respective facilities provided by the model connectivity layer discussed in Section 2.6.

11.4 Framework Design and Core Tasks

The role of the core framework, illustrated in Figure 11.2, is to provide model loading and storing facilities as well as runtime communication facilities to the individual model management tasks that build atop it. This section provides a detailed discussion of the components it consists of.

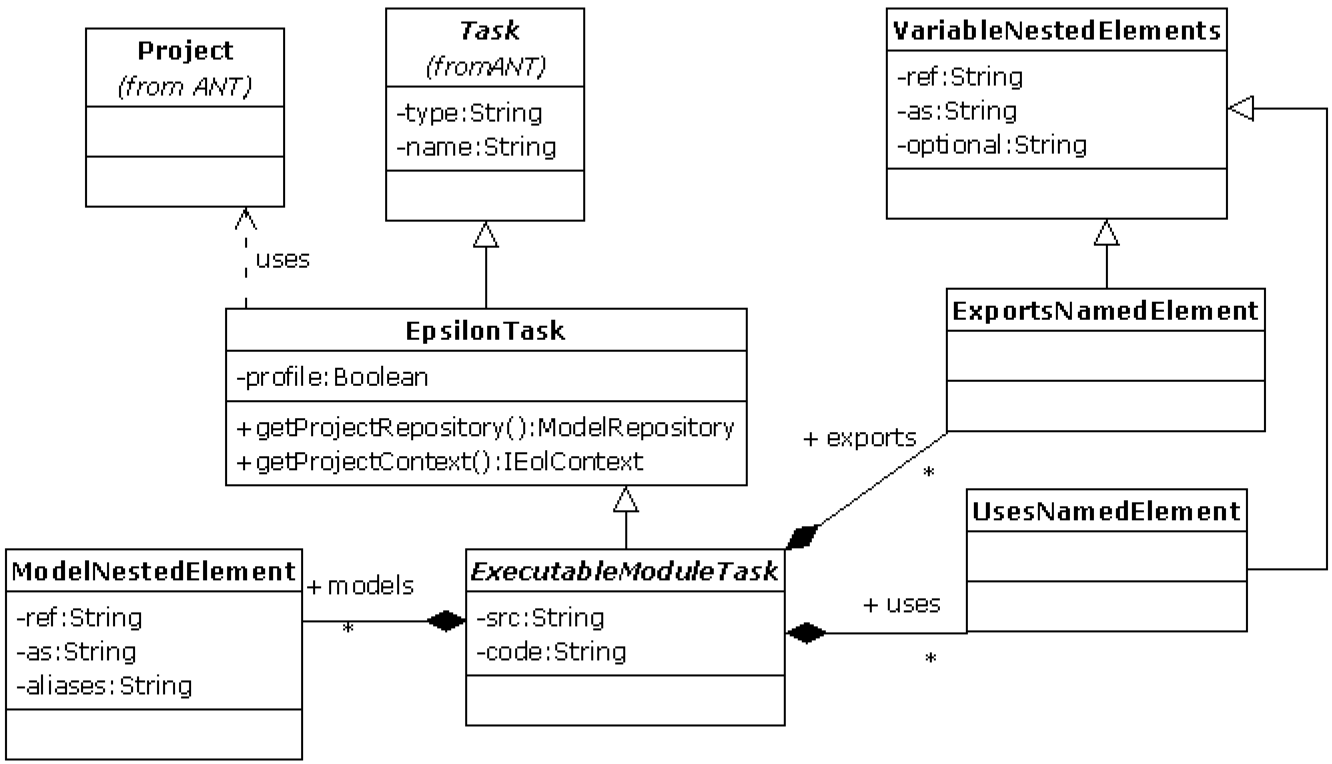

Figure 11.2: Core Framework

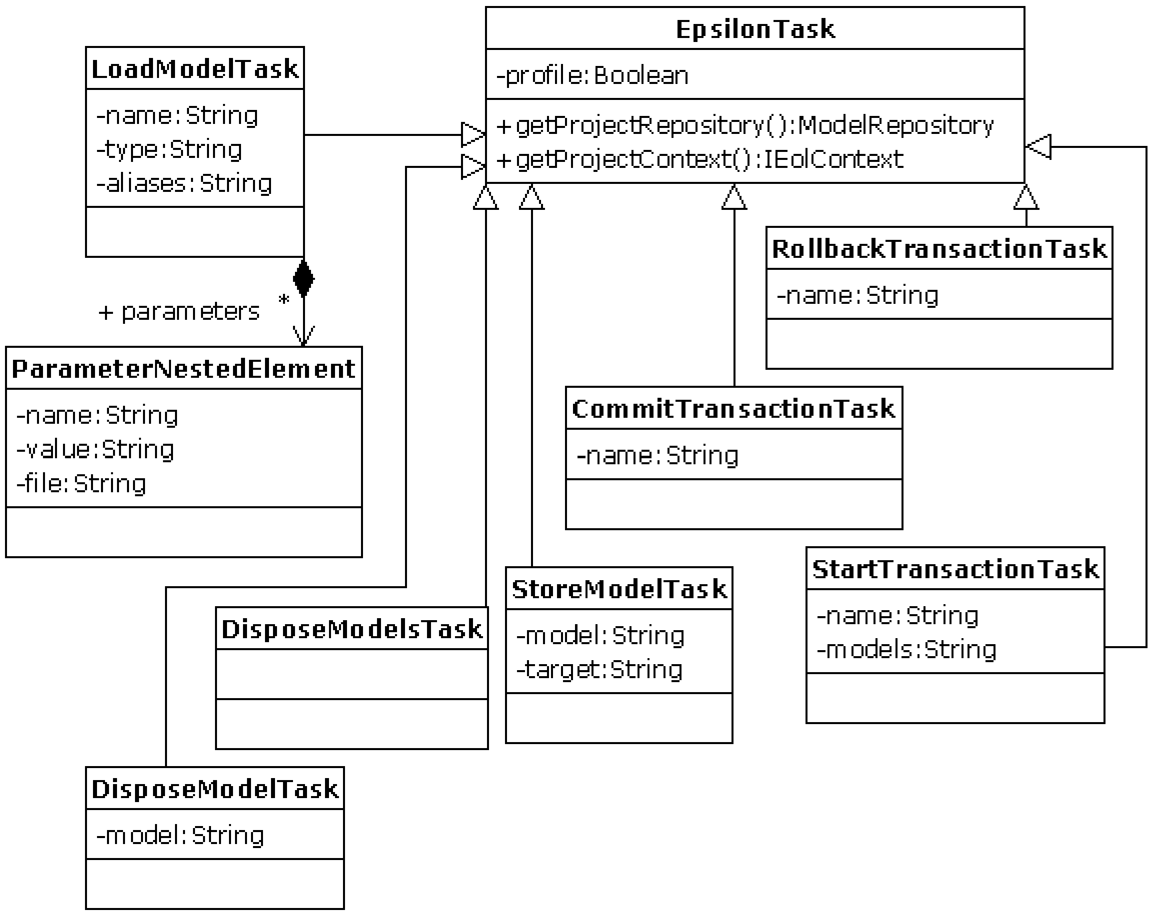

Figure 11.3: Core Models Framework

11.4.1 The EpsilonTask task

An ANT task can access the project in which it is contained by invoking the *Task.getProject()* method. To facilitate sharing of arbitrary information between tasks, ANT projects provide two convenience methods, namely *addReference(String key, Object ref)* and *getReference(String key) : Object*. The former is used to add key-value pairs, which are then accessible using the latter from other tasks of the project.

To avoid loading models multiple times and to enable on-the-fly management of models from different Epsilon modules without needing to store and re-load the models after each task, a reference to a project-wide model repository has been added to the current ANT project using the *addReference* method discussed above. In this way, all the subclasses of the abstract *EpsilonTask* can invoke the *getProjectRepository()* method to access the project model repository.

Also, to support a variable sharing mechanism that enables inter-task communication, the same technique has been employed; a shared context, accessible by all Epsilon tasks via the *getProjectContext()* method, has been added. Through this mechanism, model management tasks can export variables to the project context (e.g. traces or lists containing results of expensive queries) which other tasks can then reuse.

EpsilonTask also specifies a *profile* attribute that defines if the execution of the task must be profiled using the profiling features provided by Epsilon. Profiling is a particularly important aspect of workflow execution, especially where model management languages are involved. The main reason is that model management languages tend to provide convenient features which can however be computationally expensive (such as the *allInstances()* EOL built-in feature that returns all the instances of a specific metaclass in the model) and when used more often than really needed, can significantly degrade the overall performance.

11.4.2 Model Loading Task

The *LoadModelTask (epsilon.loadModel)* loads a model from an arbitrary location (e.g. file-system, database) and adds it to the project repository so that subsequent Epsilon tasks can query or modify it. Since Epsilon supports many modelling technologies (e.g. EMF, MDR, XML), the *LoadModelTask* defines only three generic attributes. The *name* attribute specifies the name of the model in the project repository. The *type* attribute specifies the modelling technology with which the model is captured and is used to resolve the technology-specific model loading functionality. Finally, the *aliases* attribute defines a comma-separated list of alternative names by which the model can be accessed in the model repository.

The rest of the information needed to load a model is implementation-specific and is therefore provided through *parameter* nested elements, each one defining a pair of *name-value* attributes. As an example, a task for loading an EMF model that has a file-based ECore metamodel is displayed in Listing 11.2.

Listing 11.2: Loading an EMF model using the epsilon.loadModel task

```
<epsilon.loadModel name="Tree1" type="EMF">
      <parameter name="modelFile" value="TreeInstance.ecore"/>
      <parameter name="metamodelFile" path="Tree.ecore"/>
      <parameter name="isMetamodelFileBased" value="true"/>
      <parameter name="readOnLoad" value="true"/>
</epsilon.loadModel>
```

11.4.3 Model Storing Task

The *StoreModelTask (epsilon.storeModel)* is used to store a model residing in the project repository. The *StoreModelTask* defines two attributes. The *name* attribute specifies the name of the model to be stored and the *target* attribute specifies the location where the model will be stored. The *target* attribute is optional and if it is not defined, the model is stored in the location from which it was originally loaded.

11.4.4 Model Disposal Tasks

When a model is no longer required by tasks of the workflow, it can be disposed using the *epsilon.disposeModel* task. The task provides the *model* attribute that defines the name of the model to be disposed. Also, the attribute-less *epsilon.disposeModels* task is provided that disposes all the models in the project model repository. This task is typically invoked when the model management part of the workflow has finished.

The workflow leverages the model-transaction services provided by the model connectivity framework of Epsilon by providing three tasks for managing transactions in the context of workflows.

11.4.5 The StartTransaction Task

The *epsilon.startTransaction* task defines a *name* attribute that identifies the transaction. It also optionally defines a comma-separated list of model names (*models*) that the transaction will manage. If the *models* attribute is not specified, the transaction involves all the models contained in the common project model repository.

11.4.6 The CommitTransaction and RollbackTransaction Tasks

The *epsilon.commitTransaction* and *epsilon.rollbackTransaction* tasks define a *name* attribute through which the transaction to be committed/rolled-back is located in the project's active transactions. If several active transactions with the same name exist the more recent one is selected.

The example of Listing 11.3 demonstrates an exemplar usage of the *epsilon.startTransaction* and *epsilon.rollbackTransaction* tasks. In this example, two empty models Tree1 and Tree2 are loaded in lines 1,2. Then, the EOL task of line 4 queries the models and prints the number of instances of the *Tree* metaclass in each one of them (which is 0 for both). Then, in line 13, a transaction named T1 is started on model Tree1. The EOL task of line 15, creates a new instance of Tree in both Tree1 and Tree2 and prints the number of instances of Tree in the two models (which is 1 for both models). Then, in line 26, the T1 transaction is rolled-back and any changes done in its context to model Tree1 (but not Tree2) are undone. Therefore, the

EOL task of line 28, which prints the number of instances of Tree in both models, prints 0 for Tree1 but 1 for Tree2.

Listing 11.3: Exemplar usage of the *epsilon.startTransaction* and *epsilon.rollbackTransaction* tasks

```
1 <epsilon.loadModel name="Tree1" type="EMF">...</epsilon.loadModel>
2 <epsilon.loadModel name="Tree2" type="EMF">...</epsilon.loadModel>
3
4 <epsilon.eol>
5   <![CDATA[
6   Tree1!Tree.allInstances.size().println(); // prints 0
7   Tree2!Tree.allInstances.size().println(); // prints 0
8   ]]>
9   <model ref="Tree1"/>
10  <model ref="Tree2"/>
11 </epsilon.eol>
12
13 <epsilon.startTransaction name="T1" models="Tree1"/>
14
15 <epsilon.eol>
16  <![CDATA[
17  var t1 : new Tree1!Tree;
18  Tree1!Tree.allInstances.size().println(); // prints 1
19  var t2 : new Tree2!Tree;
20  Tree2!Tree.allInstances.size().println(); // prints 1
21  ]]>
22  <model ref="Tree1"/>
23  <model ref="Tree2"/>
24 </epsilon.eol>
25
26 <epsilon.rollbackTransaction name="T1"/>
27
28 <epsilon.eol>
29  <![CDATA[
30  Tree1!Tree.allInstances.size().println(); // prints 0
31  Tree2!Tree.allInstances.size().println(); // prints 1
```

```
  ]]>
  <model ref="Tree1"/>
  <model ref="Tree2"/>
</epsilon.eol>
```

11.4.7 The Abstract Executable Module Task

This task is the base of all the model management tasks presented in Section 11.5. Its aim is to encapsulate the commonalities of Epsilon tasks in order to reduce duplication among them. As already discussed, in Epsilon, specifications of model management tasks are organized in executable modules. While modules can be stored anywhere, in the case of the workflow it is assumed that they are either stored as separate files in the file-system or they are provided inline within the worfklow. Thus, this abstract task defines an *src* attribute that specifies the path of the source file in which the Epsilon module is stored, but also supports inline specification of the source of the module. The two alternatives are demonstrated in Listings 11.4 and 11.5 respectively.

Listing 11.4: External Module Specification

```
<project default="main">
  <target name="main">
    <epsilon.eol src="HelloWorld.eol"/>
  </target>
</project>
```

Listing 11.5: Inline Module Specification

```
<project default="main">
  <target name="main">
    <epsilon.eol>
      <![CDATA[
        "Hello world".println();
      ]]>
```

```
    </epsilon.eol>
  </target>
</project>
```

The task also defines the following nested elements:

0..n *model* **nested elements** Through the *model* nested elements, each task can define which of the models, loaded in the project repository it needs to access. Each *model* element defines three attributes. The *ref* attribute specifies the name of the model that the task needs to access, the *as* attribute defines the name by which the model will be accessible in the context of the task, and the *aliases* defines a comma-delimited sequence of aliases for the model in the context of the task.

0..n *parameter* **nested elements** The *parameter* nested elements enable users to communicate String parameters to tasks. Each *parameter* element defines a *name* and a *value* attribute. Before executing the module, each *parameter* element is transformed into a String variable with the respective name and value which is then made accessible to the module.

0..n *exports* **nested elements** To facilitate low-level integration between different Epsilon tasks, each task can export a number of variables to the project context, so that subsequent tasks can access them later. Each *export* nested element defines the three attributes. The *ref* attribute specifies the name of the variable to be exported, the *as* string attribute defines the name by which the variable is stored in the project context and the *optional* boolean attribute specifies whether the variable is mandatory. If *optional* is set to *false* and the module does not specify such a variable, an ANT *BuildException* is raised.

0..n *uses* **nested elements** The *uses* nested elements enable tasks to import variables exported by previous Epsilon tasks. Each use element supports three attributes. The *ref* attribute specifies the name of the variable to be used. If there is no variable with this name in the project context, the ANT project properties are queried. This enables Epsilon modules to access ANT parameters (e.g. provided using command-line arguments). The *as* attribute specifies the name by which the variable is accessible in the context of the task. Finally, the *optional* boolean paramter specifies if the variable must exist in the project context.

To better illustrate the runtime communication mechanism, a minimal example is provided in Listings 11.6 - 11.8. In Listing 11.6, *Exporter.eol* defines a String variable named *x* and assigns a value to it. The workflow of Listing 11.8 specifies that after executing *Exporter.eol*, it must export a variable named *x* with the new name *y* to the project context. Finally, it defines that before executing *User.eol* (Listing 11.7), it must query the project context for a variable named *y* and in case this is available, add the variable to the module's context and then execute it. Thus, the result of executing the workflow is *Some String* printed in the output console.

Listing 11.6: Source code of the Exporter.eol module

```
var x : String = "Some string";
```

Listing 11.7: Source code of the User.eol module

```
z.println();
```

Listing 11.8: ANT Workflow connecting modules 11.6 and 11.7 using the epsilon.eol task

```
<epsilon.eol src="Exporter.eol">
      <exports ref="x" as="y"/>
</epsilon.eol>

<epsilon.eol src="User.eol">
      <uses ref="y" as="z"/>
</epsilon.eol>
```

11.5 Model Management Tasks

Having discussed the core framework, this section presents the model management tasks that have been implemented atop it, using languages of the Epsilon platform.

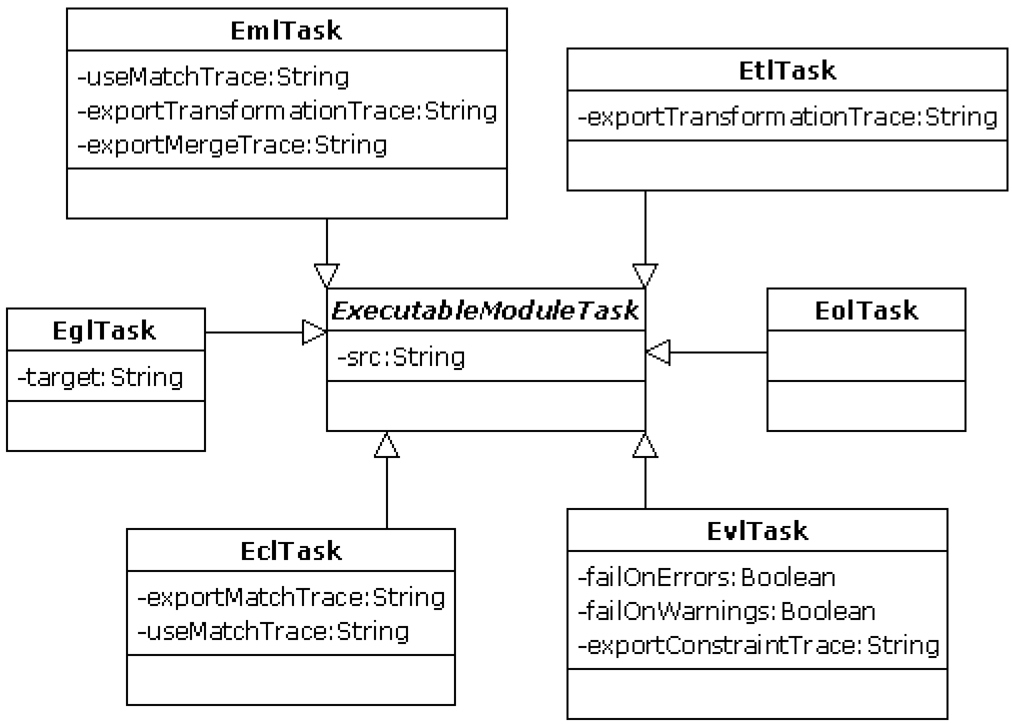

Figure 11.4: Model Management Tasks

11.5.1 Generic Model Management Task

The *epsilon.eol* task executes an EOL module, defined using the *src* attribute on the models that are specified using the *model* nested elements.

11.5.2 Model Validation Task

The *epsilon.evl* task executes an EVL module, defined using the *src* attribute on the models that are specified using the *model* nested elements. In addition to the attributes defined by the ExecutableModuleTask, this task also provides the following attributes:

- *failOnErrors* : Errors are the results of unsatisfied constraints. Setting the value of this attribute to *true* (default is *false*) causes a *BuildException* to be raised if one or more errors are identified during the validation process.

- *failOnWarnings* : Similarly to errors, warnings are the results of unsatisfied critiques. Setting the value of this atrribute to *true* (default is also *false*) causes a *BuildException* to be raised if one or more warnings are identified during the validation process.

- *exportConstraintTrace* : This attribute enables developers to export the internal constraint trace constructed during model validation to the project context so that it can be later accessed by other tasks - which could for example attempt to automatically repair the identified inconsistencies.

11.5.3 Model-to-Model Transformation Task

The *epsilon.etl* task executes an ETL module, defined using the *src* attribute to transform between the models that are specified using the *model* nested elements. In addition to the attributes defined by the ExecutableModuleTask, this task also provides the *exportTransformationTrace* attribute that enables the developer to export the internal transformation trace to the project context. In this way this trace can be reused by subsequent tasks; for example another task can serialize it in the form of a separate traceability model.

11.5.4 Model Comparison Task

The *epsilon.ecl* task executes an ECL module, defined using the *src* attribute to establish matches between elements of the models that are specified using the *model* nested elements. In addition to the attributes defined by the ExecutableModuleTask, this task also provides the *exportMatchTrace* attribute that enables users to export the match-trace calculated during the comparison to the project context so that subsequent tasks can reuse it. For example, as discussed in the sequel, an EML model merging task can use it as a means of identifying correspondences on which to perform merging. In another example, the match-trace can be stored by a subsequent EOL task in the form of an stand-alone weaving model.

11.5.5 Model Merging Task

The *epsilon.eml* task executes an EML module, defined using the *src* attribute on the models that are specified using the *model* nested elements. In addition to the attributes defined by the ExecutableModuleTask, this task also provides the following attributes:

- *useMatchTrace* : As discussed in 9, to merge a set of models, an EML module needs an established match-trace between elements of the models. The *useMatchTrace* attribute enables the EML task to use a match-trace exported by a preceeding ECL task (using its *exportMatchTrace* attribute).

- *exportMergeTrace, exportTransformationTrace* : Similarly to ETL, through these attributes an EML task can export the internal traces calculated during merging for subsequent tasks to use.

11.5.6 Model-to-Text Transformation Task

To support model to text transformations, *EglTask (epsilon.egl)* task is provided that executes an Epsilon Generation Language (EGL) module[1]. In addition to the attributes defined by *ExecutableModuleTask*, *EglTask* also defines a *target* attribute that defines where the path of the file where the generated text will be stored.

11.5.7 Adding a new Model Management Task

As discussed in Section 10, additional task-specific languages are likely to be needed in the future for tasks that are not effectively supported by existing task-specific languages. In addition to designing and implementing the syntax and execution semantics of a new language, it is also important to provide integration with the workflow – if the nature of the language permits execution within a workflow. As a counter-example, no workflow task has been provided for EWL since its execution semantics is predominately user-driven and as such, it makes little sense to execute EWL in the context of an automated workflow.

To implement support for a new task-specific language to the workflow, a new extension of the abstract *ExecutableModuleTask* needs to be provided (similarly to what has been done for existing task-specific languages). By extending *ExecutableModuleTask*, the task is automatically provided with access to the essential features of the workflow such as the shared model repository, and runtime context. Additional configuration options for the task need to specified as new ANT *attributes* and/or *nested elements*, similarly to what has been done for the tasks presented in Sections 11.5.1–11.5.6.

[1] As discussed in Section 7 EGL has been built atop Epsilon with a minimal contribution of the author

11.6 Chapter Summary

This chapter has presented the detailed design of an ANT-based framework for integrating and orchestrating mainstream software development tasks with model management tasks implemented using model management languages in Epsilon. In Section 11.4, the core framework that provides features such centralized model loading/storing facilities, a shared model repository and a mechanism through which individual tasks can communicate at runtime has been illustrated. Then, Section 11.5 has provided a discussion on the integration of the task specific languages with the framework and also provided guidance for adding support for additional languages that are likely to be developed in the future atop Epsilon.

Bibliography

[1] Object Management Group. UML 2.0 OCL Specification. http://www.omg.org/docs/ptc/03-10-14.pdf.

[2] Craig Larman. *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall PTR, 3rd edition, October 2004.

[3] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2 edition, 1997.

[4] Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In *Proc. Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis*, 2008.

[5] Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack. The Epsilon Transformation Language. In *Proc. 1st International Conference on Model Transformation*, Zurich, Switzerland, July 2008.

[6] Frédéric Jouault and Ivan Kurtev. Transforming Models with the ATL. In Jean-Michel Bruel, editor, *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, volume 3844 of *LNCS*, pages 128–138, Montego Bay, Jamaica, October 2005.

[7] Object Management Group. MOF QVT Final Adopted Specification. http://www.omg.org/cgi-bin/doc?ptc/05-11-01.pdf.

[8] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *OOPSLA '03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[9] Jack Herrington. *Code Generation in Action*. Manning, 2003. ISBN: 1930110979.

[10] Dimitrios S. Kolovos, Richard F.Paige and Fiona A.C. Polack. The Epsilon Object Language. In *Proc. European Conference in Model Driven Architecture (EC-MDA) 2006*, volume 4066 of *LNCS*, pages 128–142, Bilbao, Spain, July 2006.

[11] George A. Miller. WordNet: a lexical database for English. *Communications of ACM*, 38(11):39–41, 1995.

[12] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys (CSUR)*, 33(1):31–88, 2001.

[13] SimMetrics Similarity Metrics Library. http://www.dcs.shef.ac.uk/~sam/simmetrics.html.

[14] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[15] Object Management Group, Jishnu Mukerji, Joaquin Miller. MDA Guide version 1.0.1, 2001. http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf.

[16] Rachel A. Pottinger and Philip A. Bernstein. Merging Models Based on Given Correspondences. Technical Report UW-CSE-03-02-03, University of Washington, 2003. Technical report.

[17] C. Batini, M. Lenzerini, S.B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[18] Kim Letkeman. Comparing and merging UML models in IBM Rational Software Architect. IBM Developerworks, July 2005. http://www-128.ibm.com/developerworks/rational/library/05/712_comp.

[19] S. Melnik, E. Rahm and P. A. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *Proc. SIGMOD*, pages 193–204, 2003.

[20] Dimitrios S. Kolovos, Richard F. Paige, Louis M. Rose, Fiona A.C. Polack. Unit Testing Model Management Operations. In *Proc. 5th Workshop on Model Driven Engineering Verification and Validation (MoDeVVa), IEEE ICST*, Lillehammer, Norway, April 2008.

[21] The Apache Ant Project. http://ant.apache.org.

[22] Steve Holzner. *Ant: The Definitive Guide, Second Edition*. O'Reilly, April 2005. ISBN 0-596-00609-8.

[23] GNU Make, Official Web-Site. http://www.gnu.org/software/make/.

[24] Apache Maven Project. http://maven.apache.org.

[25] Julien Dubois. Master and Commander. Mastering J2EE Application Development Series. http://www.oracle.com/ technology/pub/articles/masterj2ee/files/j2ee2.pdf.

[26] ANT External Tools and Tasks. http://ant.apache.org/external.html.

[27] Atlas Model Management Architecture. http://www.sciences.univ-nantes.fr/lina/atl/AMMAROOT/.

[28] Fréderic Jouault, Jean Bézivin, and Ivan Kurtev. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *Proc GPCE'06: Proceedings of the fifth international conference on Generative programming and Component Engineering*, pages 249–254, 2006.