ATL Transformation Examples

The ATL to Problem ATL transformation

- version 0.1 -

October 2005

by

ATLAS group

LINA & INRIA

Nantes

Content

Figures

1 Introduction

The ATL to Problem example describes a transformation from an ATL model [1] into a Problem model. The generated Problem model contains the list of non-structural errors (along with additional warnings) that have been detected within the input ATL model. The transformation assumes the input ATL model is structurally correct, as those that have passed a syntactic analysis (for instance, a reference defined with cardinality [1-1] should not be undefined).

The input metamodel is based on the ATL metamodel. The output model is based on the Problem metamodel.

2 The ATL to Problem ATL transformation

2.1 Transformation overview

The KM3 to Metrics transformation is a single step transformation that produces a Metrics model from a KM3 model.

Users of the ATL Development Tools (ADT) [3] can easily produce their own ATL input model by 1) entering a textual ATL transformation (e.g. an ".atl" file) and, 2) injecting the produced textual ATL transformation into an ATL model by means of the *Inject ATL-0.2 file to ATL-0.2 model* contextual menu option.

2.2 Metamodels

The ATL to Problem transformation is based on both the ATL and Problem metamodel. The KM3 descriptions [2] of these metamodels can respectively be found in Appendix A: and Appendix B:. They are further described in the following subsections.

2.2.1 The ATL metamodel

The ATL metamodel provides semantics for the definition of ATL transformations [1]. A description of a subset of the ATL metamodel can be found in Figure 1, Figure 2 and Figure 3. The corresponding complete textual description of the ATL metamodel in the KM3 format is also provided in Appendix A:.

Figure 1 describes a subset of the core of the ATL metamodel (elements relative to the imperative part of ATL, as well as those related to rule inheritance, have been omitted in the figure). The root element of an ATL metamodel is the ATL Unit. An ATL Unit is either a transformation Module, a Library or a Query. A Unit contains a number of references to ATL libraries (LibraryRef).

Libraries and queries contain a number of Helper elements (which extends the abstract ModuleElement entity). A query also has a body which is an OclExpression. An ATL module, as for it, is composed of ModuleElements, which are either Helper or Rule elements. A module has input and output OclModels. Each OclModel has a metamodel (which is also an OclModel), and is composed of OclModelElements (OclModelElement extends the abstract OclType entity, see Figure 3 for further details).

A Rule is an abstract entity. In the scope of this transformation example, we only consider the concrete MachtedRule elements. A Rule has an optional OutPattern element along with a set of RuleVariableDeclarations (which extend the VariableDeclaration entity). The matched rule optionally defines, as for it, an InPattern. The InPattern contains a non-empty set of abstract InPatternElements, while an OutPattern contains a similar set of abstract OutPatternElements. They both extend the abstract PatternElement entity which is a VariableDeclaration.

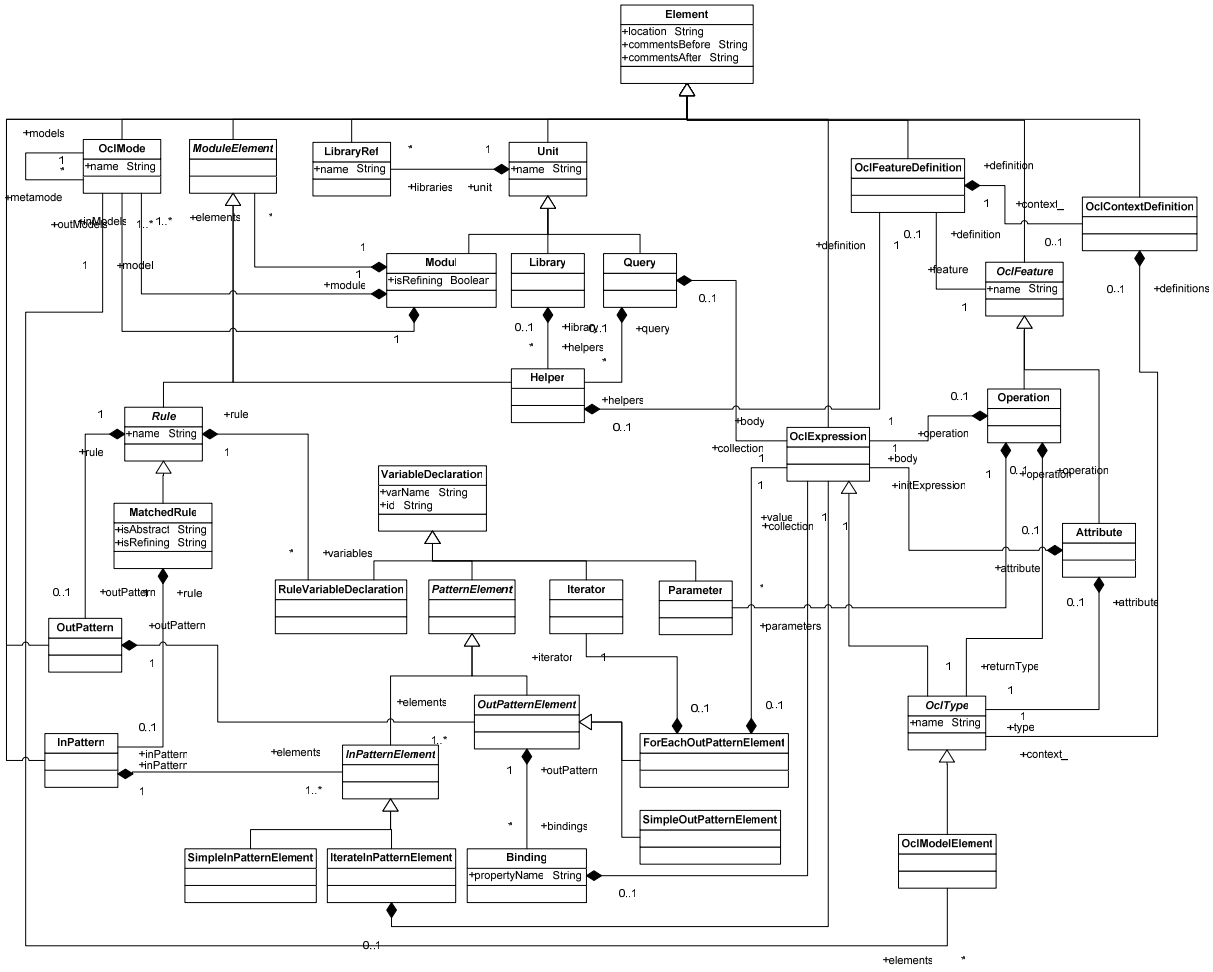

Figure 1. The ATL Core metamodel

An InPatternElement is either a SimpleInPatternElement or an IterateInPatternElement. The IterateInPatternElement contains a reference collection of type OclExpression.

An OutPatternElement is either a SimpleOutPatternElement or a ForEachoutPatternElement. The ForEachoutPatternElement contains a reference collection of type OclExpression, as well as an Iterator (that extends the VariableDeclaration entity). Each OutPatternElement contains a set of bindings. A binding associates a value with a model element property. This value is contained by the binding and encoded by an OclExpression element.

A helper contains an OclFeatureDefinition that corresponds to the helper definition. This definition is composed of an OclFeature and an optional OclContextDefinition. This last encodes the context of the helper. The default context (the ATL Module) applies when no context is associated with a helper. An OclContextDefinition contains an OclType that specifies the type of the helper context. An OclFeature is an abstract entity that is either an Attribute or an Operation. An attribute contains an OclType encoding its type, as well as an OclExpression that corresponds to its initialization expression. An Operation also contains both an OclType (its return type) and an OclExpression (its body), but also contains a set of Parameter elements (inheriting from VariableDeclaration).

Figure 2 describes the Expression part of the ATL metamodel. An OclExpression is an abstract entity that is extended by a number a different expression types: MapExp, EnumLiteralExp, PrimitiveExp, OclUndefinedExp, CollectionExp, TupleExp, VariableExp, LetExp, IfExp and PropertyCallExp.

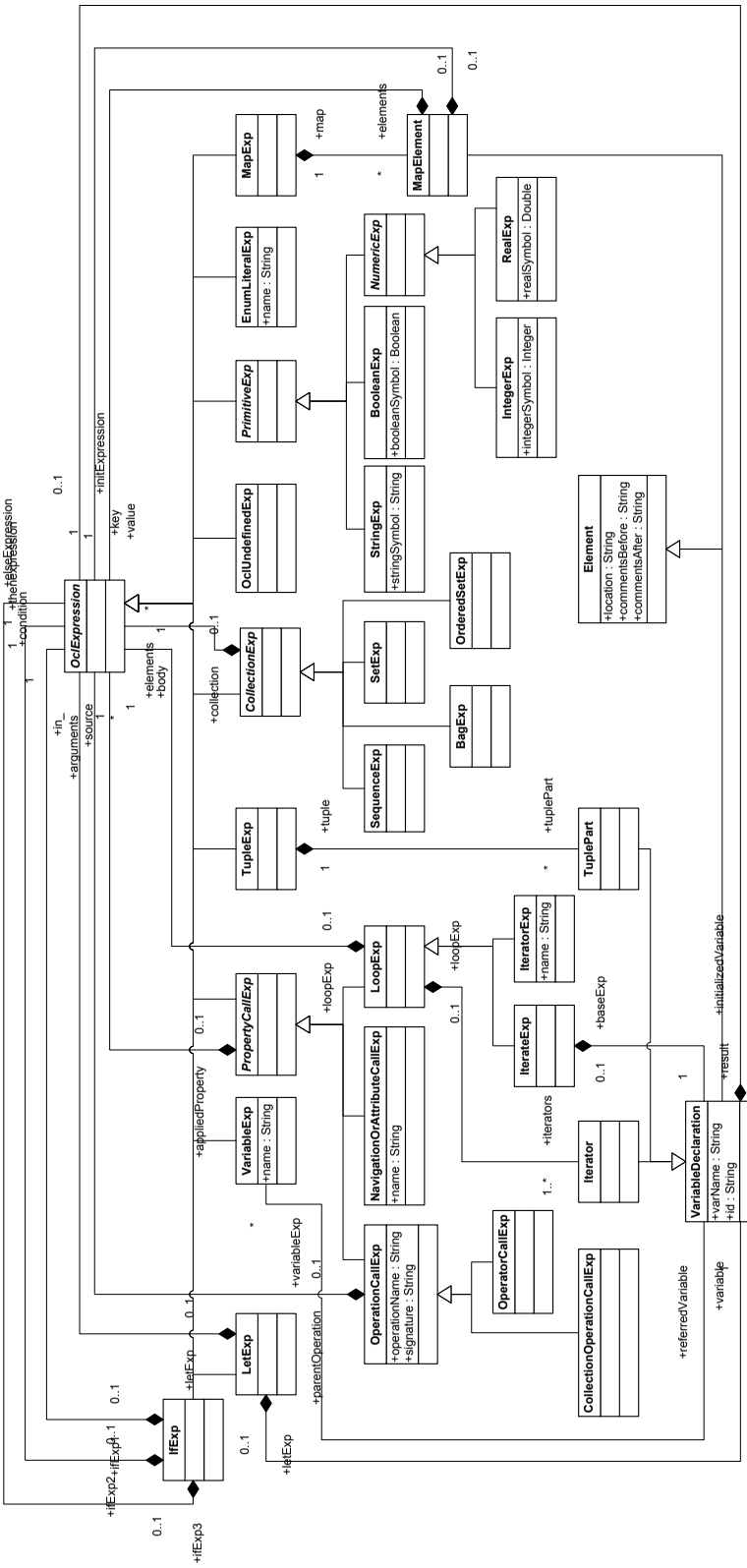

Figure 2. The ATL Expression metamodel

A MapExp contains a sequence of MapElements. Each MapElement itself contains two new OclExpression corresponding to its key and its associated value.

A PrimitiveExp is an abstract entity that is extended by StringExp, BooleanExp and the two NumericExp, IntegerExp and RealExp. Note that each PrimitiveExp entity defines its own symbol attribute encoding a value of its corresponding data type.

An abstract CollectionExp is either a SequenceExp, a BagExp, a SetExp or an OrderedSetExp. Each CollectionExp contains a sequence of OclExpression entities that correspond to the elements of the collection.

A TupleExp contains a sequence of TuplePart elements. TuplePart extends VariableDeclaration.

A VariableExp is associated with its referred VariableDeclaration.

A LetExp enables to define a new variable. It contains both a VariableDeclaration and an OclExpression that corresponds to the in statement of the let expression.

The conditional expression IfExp contains three distinct OclExpressions: one for the condition, one for the then statement and one for the then else statement.

An abstract PropertyCallExp can be extended by either a LoopExp, a NavigationOrAttributeCallExp or an OperationCallExp. Each PropertyCallExp contains an OclExpression representing the source element of the property call. An OperationCallExp is identified by its name and its signature. It is extended by both the OperatorCallExp and CollectionOperationCallExp elements. A NavigationOrAttributeCallExp is simply identified by its name. Finally, a LoopExp contains a number of Iterators (at least one) and an OclExpression representing its body. The Iterator entity extends the VariableDeclaration element. A LoopExp is either an IterateExp or an IteratorExp. The IteratorExp is simply identified by its name. The IterateExp contains a VariableDeclaration that corresponds to the result of the iterate instruction.

An OclExpression may be contained by many different elements: an IfExp (as its condition, its then statement or its else statement), a LetExp (as its in statement), a PropertyCallExp (as its source), a LoopExp (as its body), a CollectionExp (as its elements), a MapElement (as its key or its value), a VariableDeclaration (as its initialization expression), but also (see Figure 1) by a Query (as its body), an Operation (as its body), an Attribute (as its initialization expression), a Binding (as its value), an IterateInPatternElement or an ForEachOutPatternElement (as their reference collection).

Figure 3 describes the Type structure of the ATL metamodel. The base type element is represented by the abstract OclType entity. OclType extends the Oclexpression element.

The root OclType element is extended by 6 kinds of types: the abstract Collection type, the Tuple type, the OclModelElements, the OclAny type, the Primitive types (Primitive is an abstract entity) and the Map type.

A collection type is either a concrete Sequence type, a Set type, a Bag type or an OrderedSet type. Each collection type element can contain an OclType entity that encodes the type of the elements contains in the collection. A TupleType contains a set of TupleTypeAttribute elements. Each of these attributes contains an OclType encoding its own type.

An abstract Primitive type can be either a BooleanType, a StringType or an abstract NumericType (which is itself either a concrete Realtype or IntegerType). Finally, the MapType contains two distinct OclType respectively encoding its key and its value types.

An OclType is either contained by an Operation (as its return type), an attribute (as its type), an OclContextDefinition (as its context), an OclExpression (as its type), a VariableDeclaration (as its type), a TupleTypeAttribute (as its type) or a MapType (either as its key or value type).

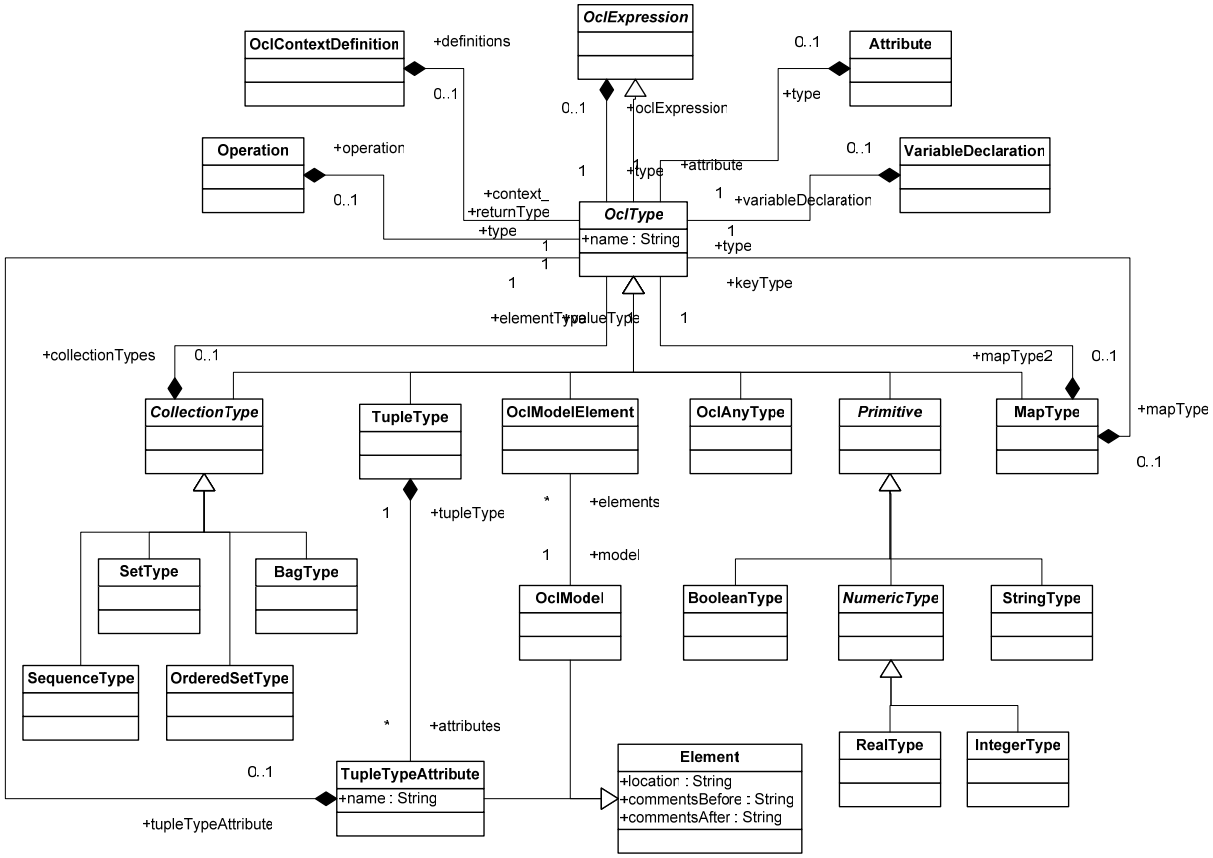

Figure 3. The ATL Type metamodel

2.2.1.1 Additional constraints

Figure 1, Figure 2 and Figure 3 define a number of structural constraints on ATL models. However, in the same way additional constraints can be specified on a MOF metamodel [4] by means of the OCL language [5], ATL models have to respect some non-structural additional constraints.

We describe here a set of non-structural constraints that have to be respected by ATL models:

- A VariableExp has to be associated with a variable declaration of its namespace.
- A model name has to be unique.
- A rule name has to be unique.
- A helper signature has to be unique.
- A binding name has to be unique in its pattern.
- A pattern name has to be unique in its rule.
- A rule variable name has to be unique in its rule.
- A helper should not have a context with a collection type.
- A declared variable should be called neither “self” nor “thisModule”.

- The use of a “self” variable is prohibited in rules.
- A parameter name has to be unique within an operation definition.
- A loop variable name (including both loop iterators and result) has to be unique within the loop definition.
- The “thisModule.resolveTemp” operation should be called neither in attribute helpers, nor in source patterns.
- Due to the current ATL implementation, an IteratorExp should not have more than one iterator. This constraint should be associated with two distinct problems since, according to the OCL specification [5], some IteratorExp elements (“exists” and “forAll”) should accept several iterators whereas the other IteratorExps are limited to a single one.

Moreover, we consider here an additional lightweight constraint:

- A variable name should be unique (but do not have to) within the namespace it belongs to (that is, a variable declaration should not hide a previously declared variable).

2.2.2 The Problem metamodel

The Problem metamodel provides semantics for the description of different kinds of problems. Figure 4 provides a description of the Problem metamodel. Its corresponding complete textual description in the KM3 format is also provided in Appendix B:.

A Problem model corresponds to a set of Problem elements. Each Problem is characterized by a *severity*, a *location* and a *description*. *severity* is of the Severity enumeration type, and can accept “error”, “warning”, and “critic” as value. The *location* and the *description* are both string attributes. The *location* attribute aims to encode the localisation of the Problem in the source file, whereas *description* provides a textual and human-readable description of the Problem.

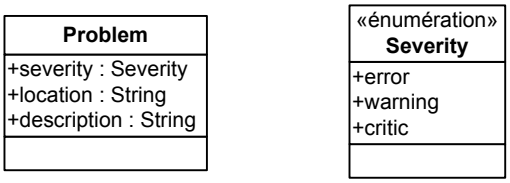

Figure 4. The Problem metamodel

2.3 Rules specification

Here are the rules used to generate a Problem model from an ATL model:

- An `error` Problem is generated for each VariableDeclaration which has no container and which name is neither “self” nor “thisModule”;
- An `error` Problem is generated for each OclModel for which another model with the same name exists in the same ATL module;
- An `error` Problem is generated for each Rule for which another Rule with the same name exists in the same ATL module;
- An `error` Problem is generated for each Helper for which another Helper with the same signature exists in the same ATL module. Note that with the current implementation, the signature of a helper is limited to its name and its context (neither the parameters nor the return value are considered);

- An `error` Problem is generated for each Binding for which another Binding with the same name exists in the same rule;
- An `error` Problem is generated for each Pattern for which another named element (either an InPatternElement, an OutPatternElement, or a RuleVariableDeclaration) with the same name exists in the same rule;
- An `error` Problem is generated for each RuleVariableDeclaration for which another named element (either an InPatternElement, an OutPatternElement, or a RuleVariableDeclaration) with the same name exists in the same rule;
- An `error` Problem is generated for each Helper which defined context is of collection type. Note that this error is due to the limitations of the current ATL implementation;
- An `error` Problem is generated for each VariableDeclaration named either “self” or “thisModule” having a non-undefined container. Such VariableDeclarations correspond to the declarations explicitly specified within an ATL transformation;
- An `error` Problem is generated for each VariableExp pointing to a variable named “self” that is contained (directly or indirectly) by a rule element;
- An `error` Problem is generated for each OperationCallExp corresponding to the “thisModule.resolveTemp” call that is contained by a source pattern of a rule;
- An `error` Problem is generated for each OperationCallExp corresponding to the “thisModule.resolveTemp” call that is contained by an ATL module attribute;
- An `error` Problem is generated for each IteratorExp of kind “isUnique”, “any”, “one”, “collect”, “select”, “reject”, “collectNested” or “sortedBy” for which several iterators are defined;
- An `error` Problem is generated for each IteratorExp of kind “exists” or “forAll” for which several iterators are defined. Although the OCL specification enables to declare several iterators for these IteratorExp, this is not supported by the current ATL implementation;
- An `error` Problem is generated for each Parameter for which another Parameter of the same name is defined in the same operation declaration;
- An `error` Problem is generated for each Iterator for which either another Iterator or a result VariableDeclaration (in case of IterateExp loops) of the same name is declared in the same iterate loop definition;
- An `error` Problem is generated for each result VariableDeclaration of an IterateExp for which an Iterator of the same name is declared in the same iterate loop definition;
- A `warning` Problem is generated for each VariableDeclaration that hides another VariableDeclaration previously defined in the same namespace. See the code of the getDeclarations helper (Appendix C:, line 270) for further information on the variable namespace definition. Note however that the variables declared in the InPattern of a rule can not collide with the other variables that may be declared in the rule.

2.4 ATL code

The ATL code for the ATL to Problem transformation is provided in Appendix C:. It consists of 20 helpers and 18 rules.

2.4.1 Helpers

The **singleIteratorExps** helper provides a set of String encoding the names of the IteratorExp that accept a single iterator according to the OCL specification [5].

The **multiIteratorExps** helper provides a set of String encoding the names of the IteratorExp that accept several iterators according to the OCL specification [5].

The **collectionTypes** helper computes the set of all CollectionType elements contained in the input ATL model.

The **allModels** helper computes the set of all OclModel elements contained in the input ATL model.

The **queryElt** helper returns the Query entity contained by the input ATL model, or undefined if this input model does not describe a query.

The **allBindings** helper computes a sequence containing all the Binding elements of the input ATL model.

The **allInPatterns** helper computes a sequence containing all the InPattern elements of the input ATL model.

The **allInPatternElts** helper computes a sequence containing all the InPatternElement entities of the input ATL model.

The **allOutPatternElts** helper computes a sequence containing all the OutPatternElement entities of the input ATL model.

The **allRules** helper computes a sequence containing all the Rule entities of the input ATL model.

The **allHelpers** helper computes a sequence containing all the Helper entities of the input ATL model.

The **allLoopExps** helper computes a sequence containing all the LoopExp entities of the input ATL model.

The **allIterateExps** helper computes a sequence containing all the IterateExp entities of the input ATL model.

The **namedElts** helper computes a sequence of VariableDeclaration corresponding to the named elements of a rule: the InPatternElements, the OutPatternElements and the RuleVariableDeclarations. For this purpose, the helper builds a sequence containing the InPatterElements of the rule in case it is a MatchedRule along with its own RuleVariableDeclarations and its OutPatternElements.

The **rule** helper returns the Rule element in which the contextual PatternElement is defined. If the contextual PatternElement is an OutPatternElement, its OutPattern is accessed through the *outPattern* property. Otherwise, if the contextual PatternElement is an InPatternElement, the PatternElement is contained by an InPattern that can be accessed through the *inPattern* property. In both case, the rule can then be accessed by means of the *rule* property.

The VariableDeclaration **immediateComposite** helper aims to return the Element that directly contains the contextual variable declaration. To this end, the helper successively tests the potential containers of the variable declaration. If the variable declaration has a defined *letExp* element, the helper returns this LetExp. Otherwise, if is has a defined *baseExp* element, the helper returns this IterateExp. If the VariableDeclaration is an InPatternElement, the helper returns its InPattern. If the VariableDeclaration is an OutPatternElement, the helper returns its OutPattern. If the VariableDeclaration is one of the Iterators of a LoopExp, the helper returns this LoopExp. If the VariableDeclaration is the result of an IterateExp, the helper returns this IterateExp. If the VariableDeclaration is a RuleVariableDeclaration, the helper returns the Rule in which it is defined. Otherwise, the helper returns undefined as default value.

The OclExpression **immediateComposite** helper aims to return the Element that directly contains the contextual OCL expression. To this end, the helper successively tests the potential containers of the

contextual OCL expression. If the contextual expression has a defined *ifExp1*, *ifExp2* or *ifExp3* element, the helper returns the corresponding IfExp. Otherwise, if the contextual expression has a defined *attribute* element, the helper returns this Attribute. If the contextual expression has a defined *operation* element, the helper returns this Operation. If the contextual expression has a defined *initializedVariable* element, the helper returns this VariableDeclaration. If the contextual expression has a defined *parentOperation* element, the helper returns this OperationCallExp. If the contextual expression has a defined *loopExp* element, the helper returns this LoopExp. If the contextual expression has a defined *letExp* element, the helper returns this LetExp. If the contextual expression has a defined *collection* element, the helper returns this CollectionExp. If the contextual expression has a defined *appliedProperty* element, the helper returns this PropertyCallExp. If the contextual expression is the *filter* of an InPattern entity, the helper returns this InPattern. If the contextual expression is the *value* of a Binding entity, the helper returns this Binding. If the input ATL model is a Query definition and if the contextual expression corresponds to the *body* of this query, the helper returns the Query. Otherwise, the helper returns undefined as default value.

The **getDeclarations()** helper aims to compute a sequence containing all the variable declarations that belong to the same namespace that the contextual variable declaration (including the contextual declaration itself). To this end, the helper first checks whether the contextual declaration is PatternElement. In this case, it simply returns a sequence composed of this only variable declaration. Otherwise, the helper computes the direct container of the contextual variable declaration by means of the appropriate immediateComposite helper. If this container is a LetExp, the helper returns a sequence composed of the contextual declaration along with the result of the call of the getUpD helper on its calculated direct container. If this container is an IteratorExp, the helper returns a sequence composed of the contextual declaration along with the result of the call of the getUpD helper on its calculated direct container. If this container is an IterateExp, the helper returns a sequence composed of the contextual declaration, along with the result of the call of the getUpD helper on its calculated direct container. Otherwise, the helper returns a sequence containing the only contextual variable declaration as default value.

The **getUpD()** helper aims to compute a sequence containing all the VariableDeclaration elements that are declared before the contextual OclExpression in its namespace. For this purpose, the helper computes the direct container of the contextual OclExpression by means of the appropriate immediateComposite helper. If this container is undefined, the helper returns an empty sequence. Otherwise, if the container is not an OclExpression, it is either a Binding, a RuleVariableDeclaration or an InPattern. In the two first cases, the helper returns a sequence composed of the named elements (in and out pattern elements with rule variable declarations) of the rule in which the Binding/ RuleVariableDeclaration is defined. In the last case (InPattern), the helper returns an empty sequence since variables declared in the InPattern do not hide the variables declared in the other parts of a rule. If the container is an OclExpression, if it is a LetExp, the helper returns a sequence composed of the *variable* of the LetExp along with the result of its recursive call on the LetExp. If the container is an IfExp, the helper returns a sequence composed of the result of its recursive call on the IfExp. If the container is an IteratorExp, the contextual expression is either the *source* or the *body* of the IteratorExp. If the expression is the *source* of the IteratorExp, the helper returns a sequence composed of the result of its recursive call on the IteratorExp. Otherwise (if the expression is the *body* of the IteratorExp), the helper returns a sequence composed of the *iterators* of the IteratorExp along with the result of its recursive call on the IteratorExp. If the container is an IterateExp, the contextual expression is either the *source* or the *body* of the IteratorExp. If the expression is the *source* of the IteratorExp, the helper returns a sequence composed of the result of its recursive call on the IteratorExp. Otherwise (if the expression is the *body* of the IteratorExp), the helper returns a sequence composed of the *iterators* of the IteratorExp, its *result* property and the result of its recursive call on the IteratorExp. Otherwise, the helper returns an empty sequence as default value.

The **getRootComposite()** helper aims to compute the root composite (e.g. which is not of type OclExpression) of the contextual OclExpression. For this purpose, the helper first computes the direct container of the contextual expression. If this container is undefined, the helper returns the

OclUndefined value. Otherwise, if this value is an OclExpression, the helper returns the result of its recursive call on the computed container. Otherwise, the helper returns the calculated direct container.

2.4.2 Rules

The **FreeVariableIsSelfOrThisModule** rule generates an `error` Problem for each VariableDeclaration that has been integrated into the model for non-declared VariableExp. This corresponds to VariableDeclarations without any defined direct container whose name is different from “self” and “thisModule”.

The **ModelNameIsUnique** rule generates an `error` Problem for each OclModel for which there exits another model with the same name.

The **RuleNameIsUnique** rule generates an `error` Problem for each Rule for which there exits another rule with the same name.

The **HelperSignatureIsUnique** rule generates an `error` Problem for each Helper for which there exits another helper with the same signature. Note that with current ATL implementation, only the name and the context of the helper are considered as part of its signature.

The **BindingNameIsUniqueInPattern** rule generates an `error` Problem for each Binding for which there exits another binding with the same name in the same OutPatternElement.

The **PatternNameIsUniqueInRule** rule generates an `error` Problem for each PatternElement for which there exits another named element (either an InPatternElement, an outPatternElement or a RuleVariableDeclaration) with the same name in the same Rule.

The **VariableNameIsUniqueInRule** rule generates an `error` Problem for each RuleVariableDeclaration for which there exits another named element (either an InPatternElement, an outPatternElement or a RuleVariableDeclaration) with the same name in the same Rule.

The **NoHelperWithCollectionAsContext** rule generates an `error` Problem for each Helper whose declared context corresponds to a CollectionType.

The **NoSelfOrThisModuleVariableDeclaration** rule generates an `error` Problem for each explicit VariableDeclaration whose name is either “self” or “thisModule”. The input ATL model may contain two implicit variable declarations, corresponding to the default “self” and “thisModule” variables. However, these two declarations do not have any immediate composite. As a consequence, the helper only matches those VariableDeclarations whose name is either “self” or “thisModule” that have a defined direct container.

The **NoSelfVariableInRule** rule generates an `error` Problem for each VariableExp whose name is “self” and which appears within the scope of a Rule. The rule therefore matches the “self”. VariableExps whose root composite is either a Binding or an InPattern.

The **NoResolveTempInSourcePattern** rule generates an `error` Problem for each OperationCallExp whose name is “resolveTemp”, whose source is the “thisModule” variable and which is contained within a rule InPattern (e.g. whose root composite is an InPattern).

The **NoResolveTempInModuleAttribute** rule generates an `error` Problem for each OperationCallExp whose name is “resolveTemp”, whose source is the “thisModule” variable and which is contained within a module Attribute (e.g. whose root composite is an Attribute).

The **ProhibitedMultiIteratorCollectionOperation** rule generates an `error` Problem for each IteratorExp that includes more than one Iterator while it should accept a single Iterator according to the OCL specification [5] (see Section 2.3 for the list of concerned IteratorExp).

The **UnsupportedMultiIteratorCollectionOperation** rule generates an `error` Problem for each IteratorExp that includes more than one Iterator, but for which multi-Iterator is still not supported by the current ATL implementation (see Section 2.3 for the list of concerned IteratorExp).

The **ParameterNameIsUniqueInOperation** rule generates an `error` Problem for each Parameter for which there exists another Parameter of the same name declared in the same Operation.

The **VariableNameIsUniqueInLoop** rule generates an `error` Problem for each Iterator for which there exists either another Iterator or a result VaribaleDeclaration (in case the considered loop is an IterateExp) of the same name declared in the same loop.

The **ResultNameIsUniqueInIterate** rule generates an `error` Problem for each VariableDeclaration encoding the result of an IterateExp for which there exists an Iterator variable of the same name declared in the same IterateExp loop.

The **VariableNameIsUniqueInContainer** rule generates a `warning` Problem for each VariableDeclaration for which there exits another VariableDeclaration with the same name in the same namespace (see Section 2.3 for further details on the namespace definitions). For this purpose, the set of VariableDeclarations of the namespace of the matched VariableDeclaration is computed by the getDeclarations() helper.

3 References

[1] ATL User Manual. The Eclipse Generative Model Transformer (GMT) project, http://eclipse.org/gmt/.

[2] KM3 User Manual. The Eclipse Generative Model Transformer (GMT) project, http://eclipse.org/gmt/.

[3] The ATL Development Tools (ADT). The Eclipse Generative Model Transformer (GMT) project, http://eclipse.org/gmt/.

[4] OMG/MOF. *Meta Object Facility (MOF)*, v1.4. OMG Document formal/02-04-03, April 2002. Available from www.omg.org.

[5] OMG/OCL Specification, ptc/03-10-14. October 2003. Available from www.omg.org.

Appendix A: The ATL metamodel in KM3 format

```
1   package OCL {
2           abstract class OclFeature extends Element {
3                   reference definition[0-1] : OclFeatureDefinition oppositeOf feature;
4                   attribute name : String;
5           }
6
7           class Attribute extends OclFeature {
8                   reference initExpression container : OclExpression oppositeOf "attribute";
9                   reference type container : OclType oppositeOf "attribute";
10          }
11
12          class Operation extends OclFeature {
13                  reference parameters[*] ordered container : Parameter oppositeOf "operation";
14                  reference returnType container : OclType oppositeOf "operation";
15                  reference body container : OclExpression oppositeOf "operation";
16          }
17
18          class Parameter extends VariableDeclaration {
19                  reference "operation" : Operation oppositeOf parameters;
20          }
21
22          class OclModel extends Element {
23                  reference metamodel : OclModel oppositeOf model;
24                  reference elements[*] : OclModelElement oppositeOf model;
25                  reference model[*] : OclModel oppositeOf metamodel;
26                  attribute name : String;
27          }
28
29          class OclContextDefinition extends Element {
30                  reference definition : OclFeatureDefinition oppositeOf context_;
31                  reference context_ container : OclType oppositeOf definitions;
32          }
33
34          class OclFeatureDefinition extends Element {
35                  reference feature container : OclFeature oppositeOf definition;
36                  reference context_[0-1] container : OclContextDefinition oppositeOf definition;
37          }
38  }
39
40  package Core {
41          class Element {
42                  attribute location : String;
43                  attribute commentsBefore[*] ordered : String;
44                  attribute commentsAfter[*] ordered : String;
45          }
46  }
47
48  package ATL {
49          class DerivedInPatternElement extends InPatternElement {
50                  reference value container : OclExpression;
51          }
52
53          class Query extends Unit {
54                  reference body container : OclExpression;
55                  reference helpers[*] ordered container : Helper oppositeOf query;
56          }
57
58          class Module extends Unit {
59                  attribute isRefining : Boolean;
60                  reference inModels[1-*] ordered container : OclModel;
61                  reference outModels[1-*] container : OclModel;
62                  reference elements[*] ordered container : ModuleElement oppositeOf module;
```

```
63          }
64
65          class ActionBlock extends Element {
66                  reference rule : Rule oppositeOf actionBlock;
67                  reference statements[*] ordered container : Statement;
68          }
69
70          abstract class Statement extends Element {
71          }
72
73          class ExpressionStat extends Statement {
74                  reference expression container : OclExpression;
75          }
76
77          class BindingStat extends Statement {
78                  reference source container : OclExpression;
79                  attribute propertyName : String;
80                  reference value container : OclExpression;
81          }
82
83          class IfStat extends Statement {
84                  reference condition container : OclExpression;
85                  reference thenStatements[*] ordered container : Statement;
86                  reference elseStatements[*] ordered container : Statement;
87          }
88
89          class ForStat extends Statement {
90                  reference iterator container : Iterator;
91                  reference collection container : OclExpression;
92                  reference statements[*] ordered container : Statement;
93          }
94
95          class Unit extends Element {
96                  reference libraries[*] container : LibraryRef oppositeOf unit;
97                  attribute name : String;
98          }
99
100         class Library extends Unit {
101                 reference helpers[*] ordered container : Helper oppositeOf library;
102         }
103
104         abstract class Rule extends ModuleElement {
105                 reference outPattern[0-1] container : OutPattern oppositeOf rule;
106                 reference actionBlock[0-1] container : ActionBlock oppositeOf rule;
107                 reference variables[*] ordered container : RuleVariableDeclaration oppositeOf
108 rule;
109                 attribute name : String;
110         }
111
112         abstract class OutPatternElement extends PatternElement {
113                 reference outPattern : OutPattern oppositeOf elements;
114                 reference sourceElement[0-1] : InPatternElement oppositeOf mapsTo;
115                 reference bindings[*] ordered container : Binding oppositeOf outPatternElement;
116         }
117
118         class InPattern extends Element {
119                 reference elements[1-*] container : InPatternElement oppositeOf inPattern;
120                 reference rule : MatchedRule oppositeOf inPattern;
121                 reference filter[0-1] container : OclExpression;
122         }
123
124         class OutPattern extends Element {
125                 reference rule : Rule oppositeOf outPattern;
126                 reference elements[1-*] ordered container : OutPatternElement oppositeOf
127 outPattern;
128         }
129
130         abstract class ModuleElement extends Element {
131                 reference module : Module oppositeOf elements;
```

```
132         }
133
134         class Helper extends ModuleElement {
135                 reference query[0-1] : Query oppositeOf helpers;
136                 reference library[0-1] : Library oppositeOf helpers;
137                 reference definition container : OclFeatureDefinition;
138         }
139
140         class SimpleInPatternElement extends InPatternElement {
141         }
142
143         class IterateInPatternElement extends InPatternElement {
144                 reference collection container : OclExpression;
145         }
146
147         abstract class InPatternElement extends PatternElement {
148                 reference mapsTo : OutPatternElement oppositeOf sourceElement;
149                 reference inPattern : InPattern oppositeOf elements;
150         }
151
152         abstract class PatternElement extends VariableDeclaration {
153         }
154
155         class CalledRule extends Rule {
156                 reference parameters[*] container : Parameter;
157                 attribute isEntrypoint : Boolean;
158         }
159
160         class Binding extends Element {
161                 reference value container : OclExpression;
162                 reference outPatternElement : OutPatternElement oppositeOf bindings;
163                 attribute propertyName : String;
164         }
165
166         class ForEachOutPatternElement extends OutPatternElement {
167                 reference collection container : OclExpression;
168                 reference iterator container : Iterator;
169         }
170
171         class RuleVariableDeclaration extends VariableDeclaration {
172                 reference rule : Rule oppositeOf variables;
173         }
174
175         class LibraryRef extends Element {
176                 reference unit : Unit oppositeOf libraries;
177                 attribute name : String;
178         }
179
180         class MatchedRule extends Rule {
181                 reference inPattern[0-1] container : InPattern oppositeOf rule;
182                 reference children[*] : MatchedRule oppositeOf superRule;
183                 reference superRule[0-1] : MatchedRule oppositeOf children;
184                 attribute isAbstract : Boolean;
185                 attribute isRefining : Boolean;
186         }
187
188         class LazyMatchedRule extends MatchedRule {
189                 attribute isUnique : Boolean;
190         }
191
192         class SimpleOutPatternElement extends OutPatternElement {
193         }
194 }
195
196 package Expressions {
197         class CollectionOperationCallExp extends OperationCallExp {
198         }
199
200         class VariableExp extends OclExpression {
```

```
201                 reference referredVariable : VariableDeclaration oppositeOf variableExp;
202                 attribute name : String;
203         }
204
205         class MapExp extends OclExpression {
206                 reference elements[*] ordered container : MapElement oppositeOf map;
207         }
208
209         class MapElement extends Element {
210                 reference map : MapExp oppositeOf elements;
211                 reference key container : OclExpression;
212                 reference value container : OclExpression;
213         }
214
215         class RealExp extends NumericExp {
216                 attribute realSymbol : Double;
217         }
218
219         abstract class PrimitiveExp extends OclExpression {
220         }
221
222         class OclUndefinedExp extends OclExpression {
223         }
224
225         class IterateExp extends LoopExp {
226                 reference result container : VariableDeclaration oppositeOf baseExp;
227         }
228
229         abstract class PropertyCallExp extends OclExpression {
230                 reference source container : OclExpression oppositeOf appliedProperty;
231         }
232
233         class TuplePart extends VariableDeclaration {
234                 reference tuple : TupleExp oppositeOf tuplePart;
235         }
236
237         abstract class OclExpression extends Element {
238                 reference ifExp3[0-1] : IfExp oppositeOf elseExpression;
239                 reference appliedProperty[0-1] : PropertyCallExp oppositeOf source;
240                 reference collection[0-1] : CollectionExp oppositeOf elements;
241                 reference letExp[0-1] : LetExp oppositeOf in_;
242                 reference loopExp[0-1] : LoopExp oppositeOf body;
243                 reference parentOperation[0-1] : OperationCallExp oppositeOf arguments;
244                 reference initializedVariable[0-1] : VariableDeclaration oppositeOf
245     initExpression;
246                 reference ifExp2[0-1] : IfExp oppositeOf thenExpression;
247                 reference "operation"[0-1] : Operation oppositeOf body;
248                 reference ifExp1[0-1] : IfExp oppositeOf condition;
249                 reference type container : OclType oppositeOf oclExpression;
250                 reference "attribute"[0-1] : Attribute oppositeOf initExpression;
251         }
252
253         class IntegerExp extends NumericExp {
254                 attribute integerSymbol : Integer;
255         }
256
257         class EnumLiteralExp extends OclExpression {
258                 attribute name : String;
259         }
260
261         class OperatorCallExp extends OperationCallExp {
262         }
263
264         class IteratorExp extends LoopExp {
265                 attribute name : String;
266         }
267
268         class StringExp extends PrimitiveExp {
269                 attribute stringSymbol : String;
```

```
270         }
271
272         class BooleanExp extends PrimitiveExp {
273                 attribute booleanSymbol : Boolean;
274         }
275
276         class LetExp extends OclExpression {
277                 reference variable container : VariableDeclaration oppositeOf letExp;
278                 reference in_ container : OclExpression oppositeOf letExp;
279         }
280
281         class Iterator extends VariableDeclaration {
282                 reference loopExpr[0-1] : LoopExp oppositeOf iterators;
283         }
284
285         class VariableDeclaration extends Element {
286                 reference letExp[0-1] : LetExp oppositeOf variable;
287                 reference type container : OclType oppositeOf variableDeclaration;
288                 reference baseExp[0-1] : IterateExp oppositeOf result;
289                 reference variableExp[*] : VariableExp oppositeOf referredVariable;
290                 reference initExpression[0-1] container : OclExpression oppositeOf
291 initializedVariable;
292                 attribute varName : String;
293                 attribute id : String;
294         }
295
296         class OperationCallExp extends PropertyCallExp {
297                 reference arguments[*] ordered container : OclExpression oppositeOf
298 parentOperation;
299                 attribute operationName : String;
300                 attribute signature[0-1] : String;
301         }
302
303         abstract class NumericExp extends PrimitiveExp {
304         }
305
306         class BagExp extends CollectionExp {
307         }
308
309         abstract class CollectionExp extends OclExpression {
310                 reference elements[*] ordered container : OclExpression oppositeOf collection;
311         }
312
313         class IfExp extends OclExpression {
314                 reference thenExpression container : OclExpression oppositeOf ifExp2;
315                 reference condition container : OclExpression oppositeOf ifExp1;
316                 reference elseExpression container : OclExpression oppositeOf ifExp3;
317         }
318
319         class LoopExp extends PropertyCallExp {
320                 reference body container : OclExpression oppositeOf loopExp;
321                 reference iterators[1-*] container : Iterator oppositeOf loopExpr;
322         }
323
324         class TupleExp extends OclExpression {
325                 reference tuplePart[*] ordered container : TuplePart oppositeOf tuple;
326         }
327
328         class SequenceExp extends CollectionExp {
329         }
330
331         class NavigationOrAttributeCallExp extends PropertyCallExp {
332                 attribute name : String;
333         }
334
335         class SetExp extends CollectionExp {
336         }
337
338         class OrderedSetExp extends CollectionExp {
```

```
339             }
340     }
341
342     package Types {
343             abstract class CollectionType extends OclType {
344                     reference elementType container : OclType oppositeOf collectionTypes;
345             }
346
347             abstract class OclType extends OclExpression {
348                     reference definitions[*] : OclContextDefinition oppositeOf context_;
349                     reference oclExpression[*] : OclExpression oppositeOf type;
350                     reference "operation"[0-1] : Operation oppositeOf returnType;
351                     reference mapType2[0-1] : MapType oppositeOf valueType;
352                     reference "attribute" : Attribute oppositeOf type;
353                     reference mapType[0-1] : MapType oppositeOf keyType;
354                     reference collectionTypes[0-1] : CollectionType oppositeOf elementType;
355                     reference tupleTypeAttribute[*] : TupleTypeAttribute oppositeOf type;
356                     reference variableDeclaration[*] : VariableDeclaration oppositeOf type;
357                     attribute name : String;
358             }
359
360             class StringType extends Primitive {
361             }
362
363             abstract class Primitive extends OclType {
364             }
365
366             class RealType extends NumericType {
367             }
368
369             class OclAnyType extends OclType {
370             }
371
372             class TupleType extends OclType {
373                     reference attributes[*] container : TupleTypeAttribute oppositeOf tupleType;
374             }
375
376             class SequenceType extends CollectionType {
377             }
378
379             class BooleanType extends Primitive {
380             }
381
382             class OclModelElement extends OclType {
383                     reference model : OclModel oppositeOf elements;
384             }
385
386             class SetType extends CollectionType {
387             }
388
389             class BagType extends CollectionType {
390             }
391
392             class OrderedSetType extends CollectionType {
393             }
394
395             abstract class NumericType extends Primitive {
396             }
397
398             class TupleTypeAttribute extends Element {
399                     reference type container : OclType oppositeOf tupleTypeAttribute;
400                     reference tupleType : TupleType oppositeOf attributes;
401                     attribute name : String;
402             }
403
404             class IntegerType extends NumericType {
405             }
406
407             class MapType extends OclType {
```

```
408                 reference valueType container : OclType oppositeOf mapType2;
409                 reference keyType container : OclType oppositeOf mapType;
410         }
411 }
```

Appendix B: The Problem metamodel in KM3 format

```
1   package Diagnostic {
2
3           enumeration Severity {
4                   literal error;
5                   literal warning;
6                   literal critic;
7           }
8
9           class Problem {
10                  attribute severity: Severity;
11                  attribute location: String;
12                  attribute description: String;
13          }
14  }
```

Appendix C: The ATL to Problem ATL code

```
1   module ATL_WFR;
2   create OUT : Problem from IN : ATL;
3
4
5   ------------------------------------------------------------------------------
6   -- HELPERS -----------------------------------------------------------------
7   ------------------------------------------------------------------------------
8
9   -- This helper provides a set containing the name of the IteratorExp elements
10  -- that accepts a single Iterator.
11  -- CONTEXT:    thisModule
12  -- RETURN:     Set(String)
13  helper def: singleIteratorExps : Set(String) =
14          Set{
15                  'isUnique', 'any', 'one', 'collect', 'select',
16                  'reject', 'collectNested', 'sortedBy'
17          };
18
19
20  -- This helper provides a set containing the name of the IteratorExp elements
21  -- for which several Iterators may be declared according to the OCL spec.
22  -- CONTEXT:    thisModule
23  -- RETURN:     Set(String)
24  helper def: multiIteratorExps : Set(String) = Set{'exists', 'forAll'};
25
26
27  -- This helper computes the set of existing CollectionType elements within the
28  -- input ATL Unit.
29  -- CONTEXT:    thisModule
30  -- RETURN:     Set(ATL!CollectionType)
31  helper def: collectionTypes : Set(ATL!CollectionType) =
32          ATL!CollectionType.allInstances();
33
34
35  -- This helper computes a sequence containing all the OclModel elements that
36  -- are used in the input ATL Unit.
37  -- CONTEXT:    thisModule
38  -- RETURN:     Sequence(ATL!OclModel)
39  helper def: allModels : Sequence(ATL!OclModel) =
40          let atlModule : ATL!Module =
41                  ATL!Module.allInstances()->asSequence()->first()
42          in
43          Sequence{
44                  atlModule.inModels,
45                  atlModule.outModels
46          }->flatten();
47
48
49  -- This helper computes the Query element that corresponds to the input ATL
50  -- Unit. If the input ATL Unit corresponds to a Module (eg a transformation),
51  -- the computed value is OclUndefined.
52  -- CONTEXT:    thisModule
53  -- RETURN:     ATL!Query
54  helper def: queryElt : ATL!Query =
55          ATL!Query.allInstances()->asSequence()->first();
56
57
58  -- This helper computes a sequence containing all the Binding elements that
59  -- are defined in the input ATL Unit.
60  -- CONTEXT:    thisModule
61  -- RETURN:     Sequence(ATL!Binding)
62  helper def: allBindings : Sequence(ATL!Binding) =
63          ATL!Binding.allInstances()->asSequence();
```

```
64
65
66    -- This helper computes a sequence containing all the Pattern elements that
67    -- are defined in the input ATL Unit.
68    -- CONTEXT:    thisModule
69    -- RETURN:     Sequence(ATL!InPattern)
70    helper def: allInPatterns : Sequence(ATL!InPattern) =
71           ATL!InPattern.allInstances()->asSequence();
72
73
74    -- This helper computes a sequence containing all the InPatternElement elements
75    -- that are defined in the input ATL Unit.
76    -- CONTEXT:    thisModule
77    -- RETURN:     Sequence(ATL!InPatternElement)
78    helper def: allInPatternElts : Sequence(ATL!InPatternElement) =
79           ATL!InPatternElement.allInstances()->asSequence();
80
81
82    -- This helper computes a sequence containing all the OutPatternElement
83    -- elements that are defined in the input ATL Unit.
84    -- CONTEXT:    thisModule
85    -- RETURN:     Sequence(ATL!OutPatternElement)
86    helper def: allOutPatternElts : Sequence(ATL!OutPatternElement) =
87           ATL!OutPatternElement.allInstances()->asSequence();
88
89
90    -- This helper computes a sequence containing all the Rule elements that are
91    -- defined in the input ATL Unit. If the input Unit is a query, the computed
92    -- sequence is empty.
93    -- CONTEXT:    thisModule
94    -- RETURN:     Sequence(ATL!Rule)
95    helper def: allRules : Sequence(ATL!Rule) =
96           ATL!Rule.allInstances()->asSequence();
97
98
99    -- This helper computes a sequence containing all the Helper elements that are
100   -- defined in the input ATL Unit.
101   -- CONTEXT:    thisModule
102   -- RETURN:     Sequence(ATL!Helper)
103   helper def: allHelpers : Sequence(ATL!Helper) =
104          ATL!Helper.allInstances()->asSequence();
105
106
107   -- This helper computes a sequence containing all the LoopExp elements that are
108   -- defined in the input ATL Unit.
109   -- CONTEXT:    thisModule
110   -- RETURN:     Sequence(ATL!LoopExp)
111   helper def: allLoopExps : Sequence(ATL!LoopExp) =
112          ATL!LoopExp.allInstances()->asSequence();
113
114
115   -- This helper computes a sequence containing all the IterateExp elements that
116   -- are defined in the input ATL Unit.
117   -- CONTEXT:    thisModule
118   -- RETURN:     Sequence(ATL!IterateExp)
119   helper def: allIterateExps : Sequence(ATL!IterateExp) =
120          ATL!IterateExp.allInstances()->asSequence();
121
122
123   -- This helper computes a sequence containing all the VariableDeclaration
124   -- elements that are associated with the contextual Rule. These declarations
125   -- can be of 3 different kinds:
126   --  * the variables declared for the rule;
127   --  * the OutPatternElements of the rule;
128   --  * the InPatternElements of the rule if this last is a MatchedRule.
129   -- CONTEXT:    ATL!Rule
130   -- RETURN:     Sequence(ATL!VariableDeclaration)
131   helper context ATL!Rule
132          def: namedElements : Sequence(ATL!VariableDeclaration) =
```

```
133         Sequence{
134                 if self.oclIsTypeOf(ATL!MatchedRule)
135                 then
136                         self.inPattern.elements->asSequence()
137                 else
138                         Sequence{}
139                 endif,
140                 self.variables->asSequence(),
141                 self.outPattern.elements->asSequence()
142         }->flatten();
143
144
145   -- This helper computes the Rule element in which the contextual PatterElement
146   -- is declared. This is achieved by returning the Rule referred by the "rule"
147   -- reference of the Pattern that conatins the contexual PatternElement. This
148   -- last one is accessed through the "outPattern" reference if the contextual
149   -- PatternElement is an OutPatternElement, throught the "inPattern" if it is
150   -- an InPatternElement.
151   -- CONTEXT:    ATL!PatternElement
152   -- RETURN:     ATL!Rule
153   helper context ATL!PatternElement def: "rule" : ATL!Rule =
154         if self.oclIsKindOf(ATL!OutPatternElement)
155         then
156                 self.outPattern."rule"
157         else
158                 self.inPattern."rule"
159         endif;
160
161
162   -- This helper returns the immediate composite (container) of the contextual
163   -- VariableDeclaration.
164   -- If the "letExp" reference of the contextual VariableDeclaration is not
165   -- undefined, the helper returns the pointed LetExp.
166   -- Otherwise, if the "letExp" reference of the contextual VD is not undefined,
167   -- the helper returns the pointed IterateExp.
168   -- Otherwise, if the contextual VD is an InPatternElement, the helper returns
169   -- the InPattern in which it is contained.
170   -- Otherwise, if the contextual VD is an OutPatternElement, the helper returns
171   -- the OutPattern in which it is contained.
172   -- Otherwise, if there exists a LoopExp element that contains the contextual VD
173   -- as an iterator, the helper returns this LoopExp.
174   -- Otherwise, if there exists an IterateExp element that contains the contextual
175   -- VD as its result, the helper returns this IterateExp.
176   -- Otherwise, if there exists a Rule element that contains the contextual VD
177   -- as a rule variable iterator, the helper returns this Rule element.
178   -- Otherwise, the helper returns OclUndefined as a default value.
179   -- CONTEXT:    ATL!VariableDeclaration
180   -- RETURN:     ATL!Element
181   helper context ATL!VariableDeclaration def: immediateComposite : ATL!Element =
182         if not self.letExp.oclIsUndefined() then
183                 self.letExp
184         else if not self.baseExp.oclIsUndefined() then
185                 self.baseExp
186         else if thisModule.allInPatternElts->exists(e | e = self) then
187                 thisModule.allInPatternElts->select(e | e = self)->first().inPattern
188         else if thisModule.allOutPatternElts->exists(e | e = self) then
189                 thisModule.allOutPatternElts->select(e | e = self)->first().outPattern
190         else if thisModule.allLoopExps
191                                 ->exists(l | l.iterators->exists(e | self = e))     then
192                 thisModule.allLoopExps
193                         ->select(l | l.iterators->exists(e | self = e))->first()
194         else if thisModule.allIterateExps->exists(e | self = e.result) then
195                 thisModule.allIterateExps->select(e | self = e.result)->first()
196         else if thisModule.allRules
197                                 ->exists(r | r.variables->exists(e | self = e)) then
198                 thisModule.allRules
199                         ->select(r | r.variables->exists(e | self = e))
200                         ->first()
201         else OclUndefined
```

```
202             endif endif    endif endif endif endif endif;
203
204
205     -- This helper returns the immediate composite (container) of the contextual
206     -- OclExpression.
207     -- If the one of the "ifExp1", "ifExp2" and "ifExp3" references of the
208     -- contextual OclExpression is not undefined, the helper returns the pointed
209     -- IfExp.
210     -- Otherwise, if its "attribute" is not undefined, the helper returns the
211     -- pointed Attribute.
212     -- Otherwise, if its "operation" is not undefined, the helper returns the
213     -- pointed Operation.
214     -- Otherwise, if its "initializedVariable" is not undefined, the helper returns
215     -- the pointed VariableDeclaration.
216     -- Otherwise, if its "parentOperation" is not undefined, the helper returns the
217     -- pointed OperationCallExp.
218     -- Otherwise, if its "loopExp" is not undefined, the helper returns the pointed
219     -- LoopExp.
220     -- Otherwise, if its "letExp" is not undefined, the helper returns the
221     -- pointed LetExp.
222     -- Otherwise, if its "collection" is not undefined, the helper returns the
223     -- pointed CollectionExp.
224     -- Otherwise, if its "appliedProperty" is not undefined, the helper returns the
225     -- pointed PropertyCallExp.
226     -- Otherwise, if its "operation" is not undefined, the helper returns the
227     -- pointed Operation.
228     -- Otherwise, if there exists an InPattern that has the contextual OclExp as
229     -- filter, the helper returns this InPattern.
230     -- Otherwise, if there exists a Binding that has the contextual OclExp as
231     -- value, the helper returns this Binding.
232     -- Otherwise, if there exists a Query that has the contextual OclExp as body,
233     -- the helper returns this Query.
234     -- Otherwise, the helper retuns OclUndefined as default value.
235     -- CONTEXT:    ATL!OclExpression
236     -- RETURN:     ATL!Element
237     helper context ATL!OclExpression def: immediateComposite : ATL!Element =
238             if not self.ifExp1.oclIsUndefined() then self.ifExp1
239             else if not self.ifExp2.oclIsUndefined() then self.ifExp2
240             else if not self.ifExp3.oclIsUndefined() then self.ifExp3
241             else if not self."attribute".oclIsUndefined() then self."attribute"
242             else if not self."operation".oclIsUndefined() then self."operation"
243             else if not self.initializedVariable.oclIsUndefined()
244                     then self.initializedVariable
245             else if not self.parentOperation.oclIsUndefined() then self.parentOperation
246             else if not self.loopExp.oclIsUndefined() then self.loopExp
247             else if not self.letExp.oclIsUndefined() then self.letExp
248             else if not self.collection.oclIsUndefined() then self.collection
249             else if not self.appliedProperty.oclIsUndefined() then self.appliedProperty
250             else if thisModule.allInPatterns->exists(e | e.filter = self)
251                     then thisModule.allInPatterns->select(e | e.filter = self)->first()
252             else if thisModule.allBindings->exists(e | e.value = self)
253                     then thisModule.allBindings->select(e | e.value = self)->first()
254             else
255                     if not thisModule.queryElt.oclIsUndefined()
256                     then
257                             if thisModule.queryElt.body = self
258                             then
259                                     thisModule.queryElt
260                             else
261                                     OclUndefined
262                             endif
263                     else
264                             OclUndefined
265                     endif
266             endif endif endif endif endif endif endif
267             endif endif endif endif endif endif;
268
269
270     -- This helper computes a sequence containing the VariableDeclarations that
```

```
271    -- precede the contextual VariableDeclaration in its namespace.
272    -- If the contextual VariableDeclaration is a PatternElement, the helper only
273    -- returns this VD.
274    -- Otherwise, it computes the container of the contextual VD. If the container
275    -- is a LetExp, it returns a Sequence composed of the VD, and the results of
276    -- the calls of the getUpD helper on the calculated container.
277    -- If the container is an IteratorExp, the helper returns a Sequence composed
278    -- of the VD and the results of the call of getUpD on the computed container.
279    -- If the container is an IterateExp, the helper a Sequence containing the same
280    -- elements that the one computed for an IteratorExp.
281    -- Otherwise, the helper returns the only contextual VD as default value.
282    -- CONTEXT:    ATL!VariableDeclaration
283    -- RETURN:     Sequence(ATL!VariableDeclaration)
284    helper context ATL!VariableDeclaration
285            def: getDeclarations() : Sequence(ATL!VariableDeclaration) =
286            if self.oclIsKindOf(ATL!PatternElement)
287            then
288                    Sequence{self}
289            else
290                    let container : ATL!Element = self.immediateComposite in
291                    if container.oclIsTypeOf(ATL!LetExp)
292                    then
293                            Sequence{
294                                    self,
295                                    container.getUpD()
296                            }->flatten()
297                    else
298                            if container.oclIsTypeOf(ATL!IteratorExp)
299                            then
300                                    Sequence{
301                                            self,
302                                            container.getUpD()
303                                    }->flatten()
304                            else
305                                    if container.oclIsTypeOf(ATL!IterateExp)
306                                    then
307                                            Sequence{
308                                                    self,
309                                                    container.getUpD()
310                                            }->flatten()
311                                    else
312                                            Sequence{
313                                                    self
314                                            }->flatten()
315                                    endif
316                            endif
317                    endif
318            endif;
319
320
321    -- This helper computes a sequence containing the VariableDeclarations that are
322    -- defined higher than the contextual OclExpression in its namespace tree.
323    -- The helper first computes the container of the contextual OclExp. If this
324    -- container is undefined, it retuns an empty sequence.
325    -- Otherwise, if this container is not an OclExpression:
326    --  * If the container is a RuleVariableDeclaration, the helper returns a
327    --    sequence containing all the named elements of the rule that contains this
328    --    InPattern.
329    --  * If the container is a Binding, the helper returns a sequence containing
330    --    all the named elements of the rule that contains this Binding.
331    -- Otherwise, if the computed container is an OclExpression:
332    --  * If the container is a LetExp, the helper returns a sequence composed of
333    --    the LetExp variable and the result of its recursive call on the LetExp.
334    --  * If the container is an IfExp, the helper returns a sequence composed of
335    --    the result of its recursive call on the IfExp.
336    --  * If the container is an IteratorExp, if the contextual OclExp is the
337    --    source of the IteratorExp then the helper returns the result of its
338    --    recursive call on the IteratorExp, else it returns this result with the
339    --    "iterators" elements of the IteratorExp.
```

```
340     --  * If the container is an IterateExp, the helper returns the same sequences
341     --    that for an IteratorExp, with the additional "result" element in case the
342     --    contextual OclExp is not the source of the IterateExp.
343     -- Otherwise, the helper returns an empty sequence as default value.
344     -- CONTEXT:    ATL!OclExpression
345     -- RETURN:     Sequence(ATL!VariableDeclaration)
346     helper context ATL!OclExpression
347             def: getUpD() : Sequence(ATL!VariableDeclaration) =
348             let container : ATL!Element = self.immediateComposite in
349             if container.oclIsUndefined() then
350                     Sequence{}
351             else if not container.oclIsKindOf(ATL!OclExpression) then
352                     if container.oclIsTypeOf(ATL!RuleVariableDeclaration)
353                     then
354                             Sequence{
355                                     container."rule".namedElements
356                             }->flatten()
357                     else
358                             if container.oclIsTypeOf(ATL!Binding)
359                             then
360                                     Sequence{
361                                             container.outPatternElement."rule".namedElements
362                                     }->flatten()
363                             else
364                                     Sequence{}
365                             endif
366                     endif
367             else if container.oclIsTypeOf(ATL!LetExp) then
368                     Sequence{
369                             container.variable,
370                             container.getUpD()
371                     }->flatten()
372             else if container.oclIsTypeOf(ATL!IfExp) then
373                     Sequence{
374                             container.getUpD()
375                     }->flatten()
376             else if container.oclIsTypeOf(ATL!IteratorExp) then
377                     if container.source = self
378                     then
379                             Sequence{
380                                     container.getUpD()
381                             }->flatten()
382                     else
383                             Sequence{
384                                     container.iterators,
385                                     container.getUpD()
386                             }->flatten()
387                     endif
388             else if container.oclIsTypeOf(ATL!IterateExp) then
389                     if container.source = self
390                     then
391                             Sequence{
392                                     container.getUpD()
393                             }->flatten()
394                     else
395                             Sequence{
396                                     container.iterators,
397                                     container.result,
398                                     container.getUpD()
399                             }->flatten()
400                     endif
401             else Sequence{}
402             endif endif endif endif endif endif;
403
404
405     -- This helper returns the root composite (container) of the contextual
406     -- OclExpression. For this purpose, the helper first computes the immediate
407     -- composite of the contextual OclExpression.
408     -- If this container is undefined, the helper returns OclUndefined.
```

```
409     -- Otherwise, if it is a kind of OclExpression, the helper returns the value
410     -- provided by its recursive call on the computed container.
411     -- Finally, if this container is not an OclExpression, the root composite has
412     -- been reached (Binding/InPattern/Operation/Query/Attribute) and is returned.
413     -- CONTEXT:    ATL!OclExpression
414     -- RETURN:     ATL!Element
415     helper context ATL!OclExpression def: getRootComposite() : ATL!Element =
416             let container : ATL!Element = self.immediateComposite
417             in
418             if container.oclIsUndefined()
419             then
420                     OclUndefined
421             else
422                     if container.oclIsKindOf(ATL!OclExpression)
423                     then
424                             container.getRootComposite()
425                     else
426                             container
427                     endif
428             endif;
429
430
431     ----------------------------------------------------------------------------
432     -- RULES -------------------------------------------------------------------
433     ----------------------------------------------------------------------------
434
435     -- Rule 'FreeVariableIsSelfOrThisModule'
436     -- This rule generates an 'error' Problem for each VariableDeclaration that has
437     -- no composite, and whose name is different from both 'self' and 'thisModule'.
438     -- The VariableExps that have not been previously declared in an ATL file are
439     -- associated with a new VariableDeclaration without any composite in the
440     -- correspoding ATL model.
441     rule FreeVariableIsSelfOrThisModule {
442             from
443                     s : ATL!VariableDeclaration (
444                             s.immediateComposite.oclIsUndefined() and
445                             s.varName <> 'self' and s.varName <> 'thisModule'
446                     )
447             to
448                     t : Problem!Problem (
449                             severity <- #error,
450                             location <-
451                                     if s.variableExp->isEmpty()
452                                     then
453                                             s.location
454                                     else
455                                             s.variableExp->first().location
456                                     endif,
457                             description <- 'variable \'' + s.varName + '\' undefined'
458                     )
459     }
460
461     -- Rule 'ModelNameIsUnique'
462     -- This rule generates an 'error' Problem when there exists models that have
463     -- the same name that the checked model.
464     rule ModelNameIsUnique {
465             from
466                     s : ATL!OclModel (
467                             thisModule.allModels->exists(e | e.name = s.name and e <> s)
468                     )
469             to
470                     t : Problem!Problem (
471                             severity <- #error,
472                             location <- s.location,
473                             description <- 'model \'' + s.name + '\' already defined'
474                     )
475     }
476
477     -- Rule 'RuleNameIsUnique'
```

```
478   -- This rule generates an 'error' Problem when there exists rules that have
479   -- the same name that the checked rule.
480   rule RuleNameIsUnique {
481           from
482                   s : ATL!Rule (
483                           thisModule.allRules->exists(e | e.name = s.name and e <> s)
484                   )
485           to
486                   t : Problem!Problem (
487                           severity <- #error,
488                           location <- s.location,
489                           description <- 'rule \'' + s.name + '\' already defined'
490                   )
491   }
492
493   -- Rule 'HelperSignatureIsUnique'
494   -- This rule generates an 'error' Problem when there exists helpers that have
495   -- the same signature that the checked helper.
496   -- Note that in current implementation, the helper signature corresponds to the
497   -- name and the context of the helper.
498   rule HelperSignatureIsUnique {
499           from
500                   s : ATL!Helper (
501                           thisModule.allHelpers
502                                   ->exists(e |
503                                           e <> s and
504                                           s.definition.feature.name = e.definition.feature.name and
505                                           (
506                                            if not s.definition.context_.oclIsUndefined()
507                                            then
508                                                   if not e.definition.context_.oclIsUndefined()
509                                                   then
510                                                   if not
511   s.definition.context_.context_.name.oclIsUndefined()
512                                                           then
513                                                           if not
514   e.definition.context_.context_.name.oclIsUndefined()
515                                                                   then
516
517           s.definition.context_.context_.name = e.definition.context_.context_.name
518                                                                   else
519                                                                           false
520                                                                   endif
521                                                           else
522
523           e.definition.context_.context_.name.oclIsUndefined()
524                                                           endif
525                                                   else
526                                                           false
527                                                   endif
528                                            else
529                                                   e.definition.context_.oclIsUndefined()
530                                            endif
531                                            )
532                                   )
533                   )
534           to
535                   t : Problem!Problem (
536                           severity <- #error,
537                           location <- s.location,
538                           description <- 'helper \'' + s.definition.feature.name
539                                                   + '\' already defined'
540                   )
541   }
542
543   -- Rule 'BindingNameIsUniqueInPattern'
544   -- This rule generates an 'error' Problem when there exists, in a same pattern,
545   -- bindings that have the same name that the checked binding.
546   rule BindingNameIsUniqueInPattern {
```

```
547             from
548                     s : ATL!Binding (
549                             s.outPatternElement.bindings
550                                     ->exists(e | e.propertyName = s.propertyName and e <> s)
551                     )
552             to
553                     t : Problem!Problem (
554                             severity <- #error,
555                             location <- s.location,
556                             description <-
557                                     'binding \'' + s.propertyName + '\' already defined in pattern'
558                     )
559     }
560
561     -- Rule 'PatternNameIsUniqueInRule'
562     -- This rule generates an 'error' Problem when there exists, in a same rule,
563     -- some named elements (InPatternElement/OutPatternElement/
564     -- RuleVariableDeclaration) that have the same name that the checked pattern.
565     rule PatternNameIsUniqueInRule {
566             from
567                     s : ATL!PatternElement (
568                             s."rule".namedElements
569                                     ->exists(e | e.varName = s.varName and e <> s)
570                     )
571             to
572                     t : Problem!Problem (
573                             severity <- #error,
574                             location <- s.location,
575                             description <-
576                                     'pattern or variable named \''
577                                     + s.varName    + '\' already defined in rule'
578                     )
579     }
580
581     -- Rule 'VariableNameIsUniqueInRule'
582     -- This rule generates an 'error' Problem when there exists, in a same rule,
583     -- some named elements (InPatternElement/OutPatternElement/
584     -- RuleVariableDeclaration) that have the same name that the checked rule
585     -- variable declaration.
586     rule VariableNameIsUniqueInRule {
587             from
588                     s : ATL!RuleVariableDeclaration (
589                             s."rule".namedElements
590                                     ->exists(e | e.varName = s.varName and e <> s)
591                     )
592             to
593                     t : Problem!Problem (
594                             severity <- #error,
595                             location <- s.location,
596                             description <-
597                                     'pattern or variable named \'' + s.varName
598                                     + '\' already defined in rule'
599                     )
600     }
601
602     -- Rule 'NoHelperWithCollectionAsContext'
603     -- This rule generates an 'error' Problem for each Helper defined with a
604     -- collection type as context.
605     -- Note that this problem is due to the limitations of the current
606     -- implementation
607     rule NoHelperWithCollectionAsContext {
608             from
609                     s : ATL!Helper (
610                             if s.definition.context_.oclIsUndefined()
611                             then
612                                     false
613                             else
614                                     thisModule.collectionTypes
615                                             ->exists(e | s.definition.context_.context_ = e)
```

```
616                         endif
617                 )
618         to
619                 t : Problem!Problem (
620                         severity <- #error,
621                         location <- s.location,
622                         description <-
623                                 'helper \'' + s.definition.feature.name
624                                         + '\': current implementation does not '
625                                         + 'support helpers with collection context'
626                 )
627 }
628
629 -- Rule 'NoSelfOrThisModuleVariableDeclaration'
630 -- This rule generates an 'error' Problem for each declaration of a variable
631 -- named 'self' or 'thisModule' in the ATL program.
632 -- Considered variable declarations must have a non-undefined immediate
633 -- composite since the input ATL model may already include a 'self' and a
634 -- 'thisModule' VD without any immediate composite that correspond to the
635 -- global declarations of the 'self' and 'thisModule' variables.
636 rule NoSelfOrThisModuleVariableDeclaration {
637         from
638                 s : ATL!VariableDeclaration (
639                         not s.immediateComposite.oclIsUndefined() and
640                         (s.varName = 'self' or s.varName = 'thisModule')
641                 )
642         to
643                 t : Problem!Problem (
644                         severity <- #error,
645                         location <- s.location,
646                         description <-
647                                 'helper \'' + s.varName      + '\' is not valid variable name'
648                 )
649 }
650
651 -- Rule 'NoSelfVariableInRule'
652 -- This rule generates an 'error' Problem for each 'self' variable expression
653 -- that is contained by a rule element.
654 rule NoSelfVariableInRule {
655         from
656                 s : ATL!VariableExp (
657 --                      s.referredVariable.varName = 'self' and
658 --                      (
659 --                              let rComp : ATL!Element = s.getRootComposite() in
660 --                              rComp.oclIsTypeOf(ATL!Binding) or
661 --                              rComp.oclIsTypeOf(ATL!InPattern)
662 --                      )
663                         if s.referredVariable.oclIsUndefined()
664                         then
665                                 false
666                         else
667                                 s.referredVariable.varName = 'self' and
668                                 (
669                                         let rComp : ATL!Element = s.getRootComposite() in
670                                         rComp.oclIsTypeOf(ATL!Binding) or
671                                         rComp.oclIsTypeOf(ATL!InPattern)
672                                 )
673                         endif
674                 )
675         to
676                 t : Problem!Problem (
677                         severity <- #error,
678                         location <- s.location,
679                         description <-
680                                 'rule \'' + s.referredVariable.varName
681                                 + '\': use of the \'self\' variable prohibited in rules'
682                 )
683 }
684
```

```
685  -- Rule 'NoResolveTempInSourcePattern'
686  -- This rule generates an 'error' Problem for each call of the
687  -- 'thisModule.resolveTemp()' operation within a source pattern of a rule.
688  rule NoResolveTempInSourcePattern {
689          from
690                  s : ATL!OperationCallExp (
691                          s.operationName = 'resolveTemp' and
692                          (
693                                  if s.source.oclIsTypeOf(ATL!VariableExp)
694                                  then
695                                          if s.source.referredVariable.oclIsUndefined()
696                                          then
697                                                  false
698                                          else
699                                                  s.source.referredVariable.varName = 'thisModule'
700                                          endif
701                                  else
702                                          false
703                                  endif
704                          ) and
705                          s.getRootComposite().oclIsTypeOf(ATL!InPattern)
706                  )
707          to
708                  t : Problem!Problem (
709                          severity <- #error,
710                          location <- s.location,
711                          description <-
712                                  'rule \'' + s.getRootComposite()."rule".name
713                                  + '\': use of \'thisModule.resolveTemp()\' function '
714                                  + 'is prohibited in source patterns'
715                  )
716  }
717
718  -- Rule 'NoResolveTempInModuleAttribute'
719  -- This rule generates an 'error' Problem for each call of the
720  -- 'thisModule.resolveTemp()' operation within a model attribute.
721  rule NoResolveTempInModuleAttribute {
722          from
723                  s : ATL!OperationCallExp (
724                          s.operationName = 'resolveTemp' and
725                          (
726                                  if s.source.oclIsTypeOf(ATL!VariableExp)
727                                  then
728                                          if s.source.referredVariable.oclIsUndefined()
729                                          then
730                                                  false
731                                          else
732                                                  s.source.referredVariable.varName = 'thisModule'
733                                          endif
734                                  else
735                                          false
736                                  endif
737                          ) and
738                          s.getRootComposite().oclIsTypeOf(ATL!Attribute)
739                  )
740          to
741                  t : Problem!Problem (
742                          severity <- #error,
743                          location <- s.location,
744                          description <-
745                                  'attribute \'' + s.getRootComposite().name
746                                  + '\': use of \'thisModule.resolveTemp()\' function '
747                                  + 'is prohibited in attributes'
748                  )
749  }
750
751  -- Rule 'ProhibitedMultiIteratorCollectionOperation'
752  -- This rule generates an 'error' Problem for each IteratorExp of the
753  -- singleIteratorExps set that is associated with several Iterators.
```

```
754   rule ProhibitedMultiIteratorCollectionOperation {
755           from
756                   s : ATL!IteratorExp (
757                           thisModule.singleIteratorExps->exists(e | s.name = e) and
758                           s.iterators->size() > 1
759                   )
760           to
761                   t : Problem!Problem (
762                           severity <- #error,
763                           location <- s.location,
764                           description <-
765                                   'iterator \'' + s.name
766                                   + '\' may have at most one iterator variable'
767                   )
768   }
769
770   -- Rule 'UnsupportedMultiIteratorCollectionOperation'
771   -- This rule generates an 'error' Problem for each IteratorExp of the
772   -- multiIteratorExps set that is associated with several Iterators.
773   -- Note that this problem is due to limitations of the current implementation.
774   rule UnsupportedMultiIteratorCollectionOperation {
775           from
776                   s : ATL!IteratorExp (
777                           thisModule.multiIteratorExps->exists(e | s.name = e) and
778                           s.iterators->size() > 1
779                   )
780           to
781                   t : Problem!Problem (
782                           severity <- #error,
783                           location <- s.location,
784                           description <-
785                                   'with current implementation, iterator \'' + s.name
786                                   + '\' may have at most one iterator variable'
787                   )
788   }
789
790   -- Rule 'ParameterNameIsUniqueInOperation'
791   -- This rule generates an 'error' Problem for each parameter for which there
792   -- exists another parameter of the same name in the operation declaration.
793   rule ParameterNameIsUniqueInOperation {
794           from
795                   s : ATL!Parameter (
796                           s.operation.parameters
797                                   ->exists(e | s.varName = e.varName and s <> e)
798                   )
799           to
800                   t : Problem!Problem (
801                           severity <- #error,
802                           location <- s.location,
803                           description <-
804                                   'a parameter named \'' + s.varName
805                                   + '\' is already declared in this operation'
806                   )
807   }
808
809   -- Rule 'IteratorNameIsUniqueInLoop'
810   -- This rule generates an 'error' Problem for each Iterator declaration for
811   -- which there exists either another Iterator or a result variable declaration
812   -- (for Iterate loop only) of the same name within the same loop definition.
813   rule VariableNameIsUniqueInLoop {
814           from
815                   s : ATL!Iterator (
816                           s.loopExpr.iterators
817                                   ->exists(e | s.varName = e.varName and s <> e)
818                           or
819                           if s.loopExpr.oclIsTypeOf(ATL!IterateExp)
820                           then
821                                   s.loopExpr.result.varName = s.varName
822                           else
```

```
823                             false
824                     endif
825             )
826     to
827             t : Problem!Problem (
828                     severity <- #error,
829                     location <- s.location,
830                     description <-
831                             'a variable named \'' + s.varName
832                             + '\' is already declared in this loop'
833             )
834 }
835
836 -- Rule 'ResultNameIsUniqueInIterate'
837 -- This rule generates an 'error' Problem for each 'result' variable
838 -- declaration of an IterateExp for which there exists an Iterator variable of
839 -- the same name in the Iterate loop definition.
840 rule ResultNameIsUniqueInIterate {
841         from
842                 s : ATL!VariableDeclaration (
843                     if s.baseExp.oclIsUndefined()
844                     then
845                             false
846                     else
847                             s.baseExp.iterators
848                                     ->exists(e | s.varName = e.varName and s <> e)
849                     endif
850             )
851         to
852                 t : Problem!Problem (
853                     severity <- #error,
854                     location <- s.location,
855                     description <-
856                             'a variable named \'' + s.varName
857                             + '\' is already declared in this loop'
858             )
859 }
860
861 -- Rule 'VariableNameIsUniqueInContainer'
862 -- This rule generates a 'warning' Problem for each declaration of a variable
863 -- for which there exists another variable declaration of the same name in the
864 -- same namespace (except multiple intances of an Iterator name in a same loop
865 -- which handle 'error' Problems).
866 rule VariableNameIsUniqueInContainer {
867         from
868                 s : ATL!VariableDeclaration (
869                     s.getDeclarations()->exists(e | s.varName = e.varName and s <> e)
870             )
871         to
872                 t : Problem!Problem (
873                     severity <- #warning,
874                     location <- s.location,
875                     description <-
876                             'a variable named \'' + s.varName
877                             + '\' is already declared in this container'
878             )
879 }
```