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.

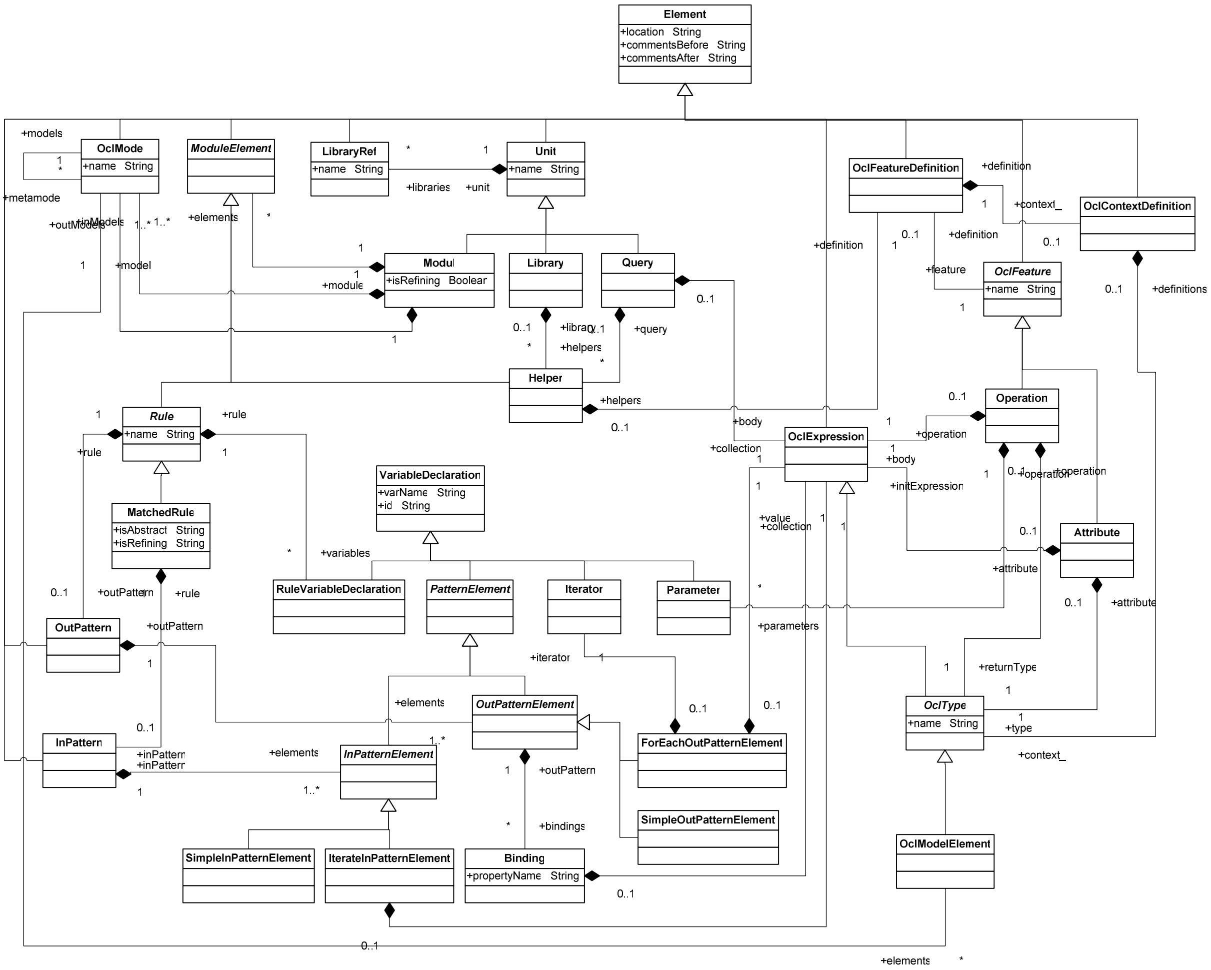

Figure 1. The ATL Core metamodel

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.

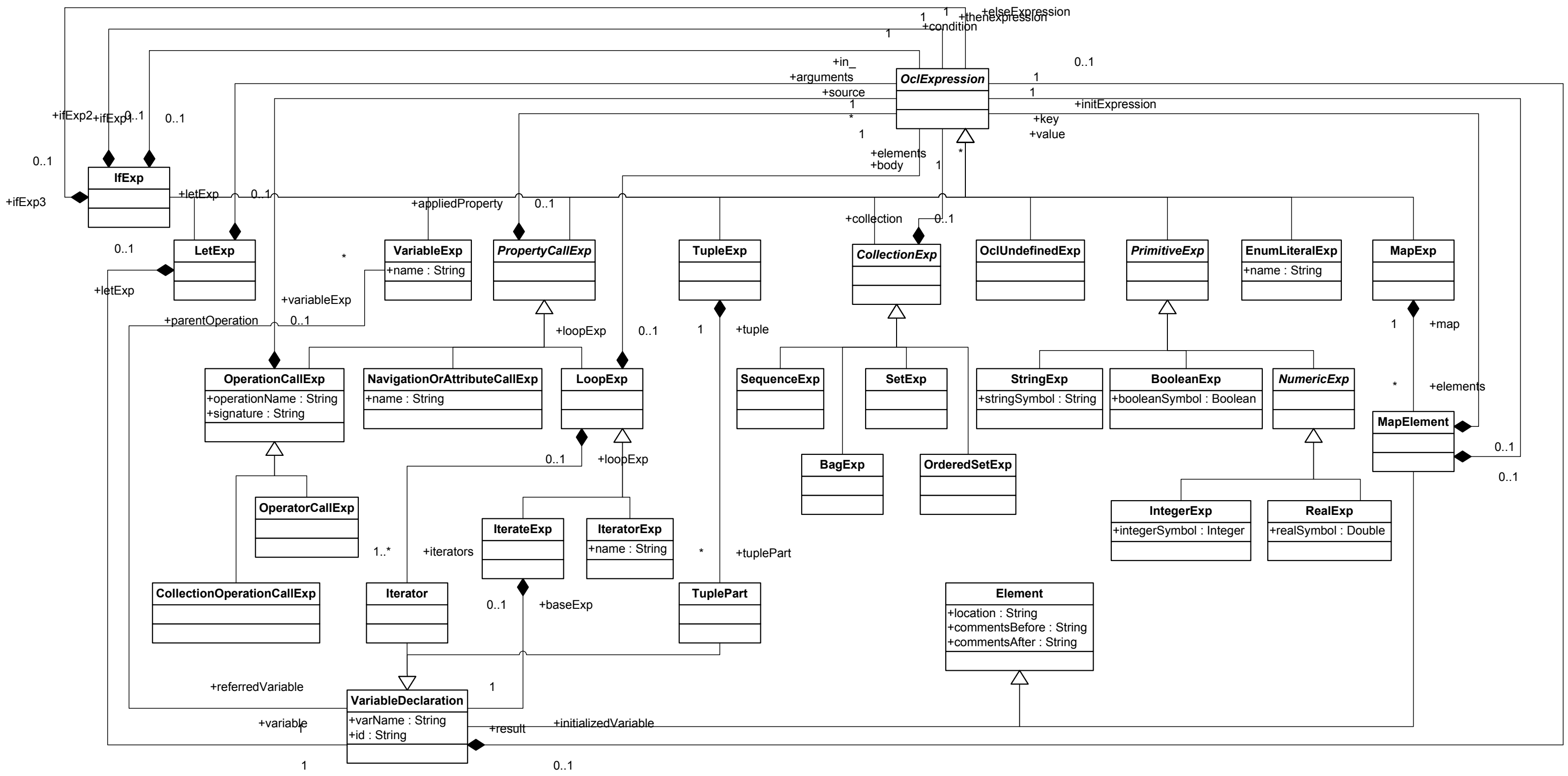

Figure 2. The ATL Expression metamodel

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).

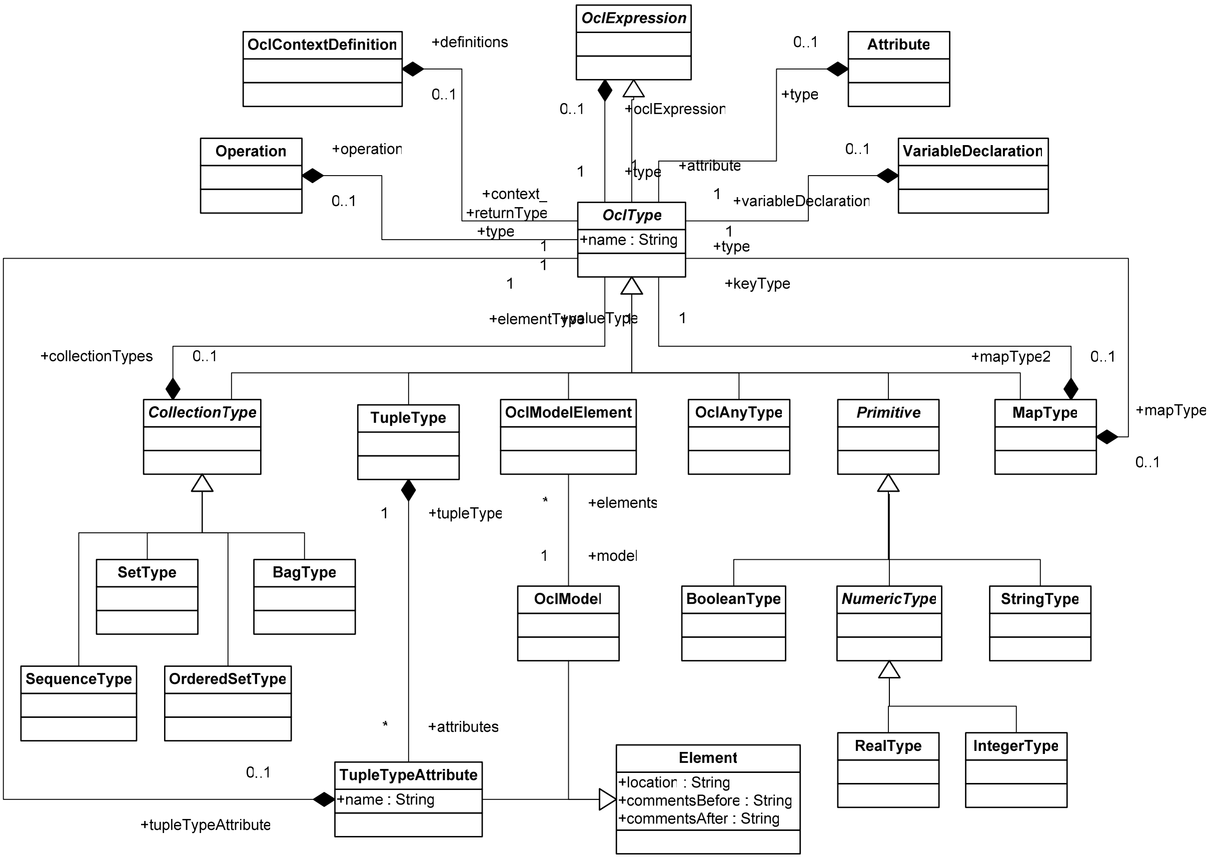

Figure 3. The ATL Type metamodel

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.

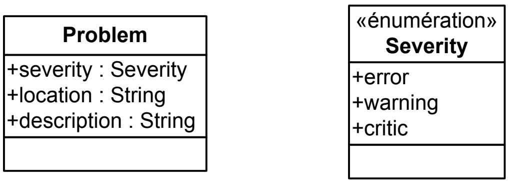

Figure 4. The Problem metamodel

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

```
package OCL {
        abstract class OclFeature extends Element {
                reference definition[0-1] : OclFeatureDefinition oppositeOf feature;
                attribute name : String;
        }

        class Attribute extends OclFeature {
                reference initExpression container : OclExpression oppositeOf "attribute";
                reference type container : OclType oppositeOf "attribute";
        }

        class Operation extends OclFeature {
                reference parameters[*] ordered container : Parameter oppositeOf "operation";
                reference returnType container : OclType oppositeOf "operation";
                reference body container : OclExpression oppositeOf "operation";
        }

        class Parameter extends VariableDeclaration {
                reference "operation" : Operation oppositeOf parameters;
        }

        class OclModel extends Element {
                reference metamodel : OclModel oppositeOf model;
                reference elements[*] : OclModelElement oppositeOf model;
                reference model[*] : OclModel oppositeOf metamodel;
                attribute name : String;
        }

        class OclContextDefinition extends Element {
                reference definition : OclFeatureDefinition oppositeOf context_;
                reference context_ container : OclType oppositeOf definitions;
        }

        class OclFeatureDefinition extends Element {
                reference feature container : OclFeature oppositeOf definition;
                reference context_[0-1] container : OclContextDefinition oppositeOf definition;
        }
}

package Core {
        class Element {
                attribute location : String;
                attribute commentsBefore[*] ordered : String;
                attribute commentsAfter[*] ordered : String;
        }
}

package ATL {
        class DerivedInPatternElement extends InPatternElement {
                reference value container : OclExpression;
        }

        class Query extends Unit {
                reference body container : OclExpression;
                reference helpers[*] ordered container : Helper oppositeOf query;
        }

        class Module extends Unit {
                attribute isRefining : Boolean;
                reference inModels[1-*] ordered container : OclModel;
                reference outModels[1-*] container : OclModel;
                reference elements[*] ordered container : ModuleElement oppositeOf module;
        }

        class ActionBlock extends Element {
                reference rule : Rule oppositeOf actionBlock;
                reference statements[*] ordered container : Statement;
        }

        abstract class Statement extends Element {
        }

        class ExpressionStat extends Statement {
                reference expression container : OclExpression;
        }

        class BindingStat extends Statement {
                reference source container : OclExpression;
                attribute propertyName : String;
                reference value container : OclExpression;
        }

        class IfStat extends Statement {
                reference condition container : OclExpression;
                reference thenStatements[*] ordered container : Statement;
                reference elseStatements[*] ordered container : Statement;
        }

        class ForStat extends Statement {
                reference iterator container : Iterator;
                reference collection container : OclExpression;
                reference statements[*] ordered container : Statement;
        }

        class Unit extends Element {
                reference libraries[*] container : LibraryRef oppositeOf unit;
                attribute name : String;
        }

        class Library extends Unit {
                reference helpers[*] ordered container : Helper oppositeOf library;
        }

        abstract class Rule extends ModuleElement {
                reference outPattern[0-1] container : OutPattern oppositeOf rule;
                reference actionBlock[0-1] container : ActionBlock oppositeOf rule;
                reference variables[*] ordered container : RuleVariableDeclaration oppositeOf
rule;
                attribute name : String;
        }

        abstract class OutPatternElement extends PatternElement {
                reference outPattern : OutPattern oppositeOf elements;
                reference sourceElement[0-1] : InPatternElement oppositeOf mapsTo;
                reference bindings[*] ordered container : Binding oppositeOf outPatternElement;
        }

        class InPattern extends Element {
                reference elements[1-*] container : InPatternElement oppositeOf inPattern;
                reference rule : MatchedRule oppositeOf inPattern;
                reference filter[0-1] container : OclExpression;
        }

        class OutPattern extends Element {
                reference rule : Rule oppositeOf outPattern;
                reference elements[1-*] ordered container : OutPatternElement oppositeOf
outPattern;
        }

        abstract class ModuleElement extends Element {
                reference module : Module oppositeOf elements;
        }

        class Helper extends ModuleElement {
                reference query[0-1] : Query oppositeOf helpers;
                reference library[0-1] : Library oppositeOf helpers;
                reference definition container : OclFeatureDefinition;
        }

        class SimpleInPatternElement extends InPatternElement {
        }

        class IterateInPatternElement extends InPatternElement {
                reference collection container : OclExpression;
        }

        abstract class InPatternElement extends PatternElement {
                reference mapsTo : OutPatternElement oppositeOf sourceElement;
                reference inPattern : InPattern oppositeOf elements;
        }

        abstract class PatternElement extends VariableDeclaration {
        }

        class CalledRule extends Rule {
                reference parameters[*] container : Parameter;
                attribute isEntrypoint : Boolean;
        }

        class Binding extends Element {
                reference value container : OclExpression;
                reference outPatternElement : OutPatternElement oppositeOf bindings;
                attribute propertyName : String;
        }

        class ForEachOutPatternElement extends OutPatternElement {
                reference collection container : OclExpression;
                reference iterator container : Iterator;
        }

        class RuleVariableDeclaration extends VariableDeclaration {
                reference rule : Rule oppositeOf variables;
        }

        class LibraryRef extends Element {
                reference unit : Unit oppositeOf libraries;
                attribute name : String;
        }

        class MatchedRule extends Rule {
                reference inPattern[0-1] container : InPattern oppositeOf rule;
                reference children[*] : MatchedRule oppositeOf superRule;
                reference superRule[0-1] : MatchedRule oppositeOf children;
                attribute isAbstract : Boolean;
                attribute isRefining : Boolean;
        }

        class LazyMatchedRule extends MatchedRule {
                attribute isUnique : Boolean;
        }

        class SimpleOutPatternElement extends OutPatternElement {
        }
}

package Expressions {
        class CollectionOperationCallExp extends OperationCallExp {
        }

        class VariableExp extends OclExpression {
                reference referredVariable : VariableDeclaration oppositeOf variableExp;
                attribute name : String;
        }

        class MapExp extends OclExpression {
                reference elements[*] ordered container : MapElement oppositeOf map;
        }

        class MapElement extends Element {
                reference map : MapExp oppositeOf elements;
                reference key container : OclExpression;
                reference value container : OclExpression;
        }

        class RealExp extends NumericExp {
                attribute realSymbol : Double;
        }

        abstract class PrimitiveExp extends OclExpression {
        }

        class OclUndefinedExp extends OclExpression {
        }

        class IterateExp extends LoopExp {
                reference result container : VariableDeclaration oppositeOf baseExp;
        }

        abstract class PropertyCallExp extends OclExpression {
                reference source container : OclExpression oppositeOf appliedProperty;
        }

        class TuplePart extends VariableDeclaration {
                reference tuple : TupleExp oppositeOf tuplePart;
        }

        abstract class OclExpression extends Element {
                reference ifExp3[0-1] : IfExp oppositeOf elseExpression;
                reference appliedProperty[0-1] : PropertyCallExp oppositeOf source;
                reference collection[0-1] : CollectionExp oppositeOf elements;
                reference letExp[0-1] : LetExp oppositeOf in_;
                reference loopExp[0-1] : LoopExp oppositeOf body;
                reference parentOperation[0-1] : OperationCallExp oppositeOf arguments;
                reference initializedVariable[0-1] : VariableDeclaration oppositeOf
initExpression;
                reference ifExp2[0-1] : IfExp oppositeOf thenExpression;
                reference "operation"[0-1] : Operation oppositeOf body;
                reference ifExp1[0-1] : IfExp oppositeOf condition;
                reference type container : OclType oppositeOf oclExpression;
                reference "attribute"[0-1] : Attribute oppositeOf initExpression;
        }

        class IntegerExp extends NumericExp {
                attribute integerSymbol : Integer;
        }

        class EnumLiteralExp extends OclExpression {
                attribute name : String;
        }

        class OperatorCallExp extends OperationCallExp {
        }

        class IteratorExp extends LoopExp {
                attribute name : String;
        }

        class StringExp extends PrimitiveExp {
                attribute stringSymbol : String;
        }

        class BooleanExp extends PrimitiveExp {
                attribute booleanSymbol : Boolean;
        }

        class LetExp extends OclExpression {
                reference variable container : VariableDeclaration oppositeOf letExp;
                reference in_ container : OclExpression oppositeOf letExp;
        }

        class Iterator extends VariableDeclaration {
                reference loopExpr[0-1] : LoopExp oppositeOf iterators;
        }

        class VariableDeclaration extends Element {
                reference letExp[0-1] : LetExp oppositeOf variable;
                reference type container : OclType oppositeOf variableDeclaration;
                reference baseExp[0-1] : IterateExp oppositeOf result;
                reference variableExp[*] : VariableExp oppositeOf referredVariable;
                reference initExpression[0-1] container : OclExpression oppositeOf
initializedVariable;
                attribute varName : String;
                attribute id : String;
        }

        class OperationCallExp extends PropertyCallExp {
                reference arguments[*] ordered container : OclExpression oppositeOf
parentOperation;
                attribute operationName : String;
                attribute signature[0-1] : String;
        }

        abstract class NumericExp extends PrimitiveExp {
        }

        class BagExp extends CollectionExp {
        }

        abstract class CollectionExp extends OclExpression {
                reference elements[*] ordered container : OclExpression oppositeOf collection;
        }

        class IfExp extends OclExpression {
                reference thenExpression container : OclExpression oppositeOf ifExp2;
                reference condition container : OclExpression oppositeOf ifExp1;
                reference elseExpression container : OclExpression oppositeOf ifExp3;
        }

        class LoopExp extends PropertyCallExp {
                reference body container : OclExpression oppositeOf loopExp;
                reference iterators[1-*] container : Iterator oppositeOf loopExpr;
        }

        class TupleExp extends OclExpression {
                reference tuplePart[*] ordered container : TuplePart oppositeOf tuple;
        }

        class SequenceExp extends CollectionExp {
        }

        class NavigationOrAttributeCallExp extends PropertyCallExp {
                attribute name : String;
        }

        class SetExp extends CollectionExp {
        }

        class OrderedSetExp extends CollectionExp {
        }
    }

    package Types {
        abstract class CollectionType extends OclType {
            reference elementType container : OclType oppositeOf collectionTypes;
        }

        abstract class OclType extends OclExpression {
            reference definitions[*] : OclContextDefinition oppositeOf context_;
            reference oclExpression[*] : OclExpression oppositeOf type;
            reference "operation"[0-1] : Operation oppositeOf returnType;
            reference mapType2[0-1] : MapType oppositeOf valueType;
            reference "attribute" : Attribute oppositeOf type;
            reference mapType[0-1] : MapType oppositeOf keyType;
            reference collectionTypes[0-1] : CollectionType oppositeOf elementType;
            reference tupleTypeAttribute[*] : TupleTypeAttribute oppositeOf type;
            reference variableDeclaration[*] : VariableDeclaration oppositeOf type;
            attribute name : String;
        }

        class StringType extends Primitive {
        }

        abstract class Primitive extends OclType {
        }

        class RealType extends NumericType {
        }

        class OclAnyType extends OclType {
        }

        class TupleType extends OclType {
            reference attributes[*] container : TupleTypeAttribute oppositeOf tupleType;
        }

        class SequenceType extends CollectionType {
        }

        class BooleanType extends Primitive {
        }

        class OclModelElement extends OclType {
            reference model : OclModel oppositeOf elements;
        }

        class SetType extends CollectionType {
        }

        class BagType extends CollectionType {
        }

        class OrderedSetType extends CollectionType {
        }

        abstract class NumericType extends Primitive {
        }

        class TupleTypeAttribute extends Element {
            reference type container : OclType oppositeOf tupleTypeAttribute;
            reference tupleType : TupleType oppositeOf attributes;
            attribute name : String;
        }

        class IntegerType extends NumericType {
        }

        class MapType extends OclType {
```

```
        reference valueType container : OclType oppositeOf mapType2;
        reference keyType container : OclType oppositeOf mapType;
    }
}
```

Appendix B: The Problem metamodel in KM3 format

```
package Diagnostic {

        enumeration Severity {
                literal error;
                literal warning;
                literal critic;
        }

        class Problem {
                attribute severity: Severity;
                attribute location: String;
                attribute description: String;
        }
}
```

Appendix C: The ATL to Problem ATL code

```
module ATL_WFR;
create OUT : Problem from IN : ATL;


--------------------------------------------------------------------------------
-- HELPERS ---------------------------------------------------------------------
--------------------------------------------------------------------------------

-- This helper provides a set containing the name of the IteratorExp elements
-- that accepts a single Iterator.
-- CONTEXT:    thisModule
-- RETURN:     Set(String)
helper def: singleIteratorExps : Set(String) =
        Set{
                'isUnique', 'any', 'one', 'collect', 'select',
                'reject', 'collectNested', 'sortedBy'
        };


-- This helper provides a set containing the name of the IteratorExp elements
-- for which several Iterators may be declared according to the OCL spec.
-- CONTEXT:    thisModule
-- RETURN:     Set(String)
helper def: multiIteratorExps : Set(String) = Set{'exists', 'forAll'};


-- This helper computes the set of existing CollectionType elements within the
-- input ATL Unit.
-- CONTEXT:    thisModule
-- RETURN:     Set(ATL!CollectionType)
helper def: collectionTypes : Set(ATL!CollectionType) =
        ATL!CollectionType.allInstances();


-- This helper computes a sequence containing all the OclModel elements that
-- are used in the input ATL Unit.
-- CONTEXT:    thisModule
-- RETURN:     Sequence(ATL!OclModel)
helper def: allModels : Sequence(ATL!OclModel) =
        let atlModule : ATL!Module =
                ATL!Module.allInstances()->asSequence()->first()
        in
        Sequence{
                atlModule.inModels,
                atlModule.outModels
        }->flatten();


-- This helper computes the Query element that corresponds to the input ATL
-- Unit. If the input ATL Unit corresponds to a Module (eg a transformation),
-- the computed value is OclUndefined.
-- CONTEXT:    thisModule
-- RETURN:     ATL!Query
helper def: queryElt : ATL!Query =
        ATL!Query.allInstances()->asSequence()->first();


-- This helper computes a sequence containing all the Binding elements that
-- are defined in the input ATL Unit.
-- CONTEXT:    thisModule
-- RETURN:     Sequence(ATL!Binding)
helper def: allBindings : Sequence(ATL!Binding) =
        ATL!Binding.allInstances()->asSequence();
```

```


-- This helper computes a sequence containing all the Pattern elements that
-- are defined in the input ATL Unit.
-- CONTEXT:    thisModule
-- RETURN:     Sequence(ATL!InPattern)
helper def: allInPatterns : Sequence(ATL!InPattern) =
        ATL!InPattern.allInstances()->asSequence();


-- This helper computes a sequence containing all the InPatternElement elements
-- that are defined in the input ATL Unit.
-- CONTEXT:    thisModule
-- RETURN:     Sequence(ATL!InPatternElement)
helper def: allInPatternElts : Sequence(ATL!InPatternElement) =
        ATL!InPatternElement.allInstances()->asSequence();


-- This helper computes a sequence containing all the OutPatternElement
-- elements that are defined in the input ATL Unit.
-- CONTEXT:    thisModule
-- RETURN:     Sequence(ATL!OutPatternElement)
helper def: allOutPatternElts : Sequence(ATL!OutPatternElement) =
        ATL!OutPatternElement.allInstances()->asSequence();


-- This helper computes a sequence containing all the Rule elements that are
-- defined in the input ATL Unit. If the input Unit is a query, the computed
-- sequence is empty.
-- CONTEXT:    thisModule
-- RETURN:     Sequence(ATL!Rule)
helper def: allRules : Sequence(ATL!Rule) =
        ATL!Rule.allInstances()->asSequence();


-- This helper computes a sequence containing all the Helper elements that are
-- defined in the input ATL Unit.
-- CONTEXT:    thisModule
-- RETURN:     Sequence(ATL!Helper)
helper def: allHelpers : Sequence(ATL!Helper) =
        ATL!Helper.allInstances()->asSequence();


-- This helper computes a sequence containing all the LoopExp elements that are
-- defined in the input ATL Unit.
-- CONTEXT:    thisModule
-- RETURN:     Sequence(ATL!LoopExp)
helper def: allLoopExps : Sequence(ATL!LoopExp) =
        ATL!LoopExp.allInstances()->asSequence();


-- This helper computes a sequence containing all the IterateExp elements that
-- are defined in the input ATL Unit.
-- CONTEXT:    thisModule
-- RETURN:     Sequence(ATL!IterateExp)
helper def: allIterateExps : Sequence(ATL!IterateExp) =
        ATL!IterateExp.allInstances()->asSequence();


-- This helper computes a sequence containing all the VariableDeclaration
-- elements that are associated with the contextual Rule. These declarations
-- can be of 3 different kinds:
--  * the variables declared for the rule;
--  * the OutPatternElements of the rule;
--  * the InPatternElements of the rule if this last is a MatchedRule.
-- CONTEXT:    ATL!Rule
-- RETURN:     Sequence(ATL!VariableDeclaration)
helper context ATL!Rule
        def: namedElements : Sequence(ATL!VariableDeclaration) =
        Sequence{
                if self.oclIsTypeOf(ATL!MatchedRule)
                then
                        self.inPattern.elements->asSequence()
                else
                        Sequence{}
                endif,
                self.variables->asSequence(),
                self.outPattern.elements->asSequence()
        }->flatten();


-- This helper computes the Rule element in which the contextual PatterElement
-- is declared. This is achieved by returning the Rule referred by the "rule"
-- reference of the Pattern that conatins the contexual PatternElement. This
-- last one is accessed through the "outPattern" reference if the contextual
-- PatternElement is an OutPatternElement, throught the "inPattern" if it is
-- an InPatternElement.
-- CONTEXT:     ATL!PatternElement
-- RETURN:      ATL!Rule
helper context ATL!PatternElement def: "rule" : ATL!Rule =
        if self.oclIsKindOf(ATL!OutPatternElement)
        then
                self.outPattern."rule"
        else
                self.inPattern."rule"
        endif;


-- This helper returns the immediate composite (container) of the contextual
-- VariableDeclaration.
-- If the "letExp" reference of the contextual VariableDeclaration is not
-- undefined, the helper returns the pointed LetExp.
-- Otherwise, if the "letExp" reference of the contextual VD is not undefined,
-- the helper returns the pointed IterateExp.
-- Otherwise, if the contextual VD is an InPatternElement, the helper returns
-- the InPattern in which it is contained.
-- Otherwise, if the contextual VD is an OutPatternElement, the helper returns
-- the OutPattern in which it is contained.
-- Otherwise, if there exists a LoopExp element that contains the contextual VD
-- as an iterator, the helper returns this LoopExp.
-- Otherwise, if there exists an IterateExp element that contains the contextual
-- VD as its result, the helper returns this IterateExp.
-- Otherwise, if there exists a Rule element that contains the contextual VD
-- as a rule variable iterator, the helper returns this Rule element.
-- Otherwise, the helper returns OclUndefined as a default value.
-- CONTEXT:     ATL!VariableDeclaration
-- RETURN:      ATL!Element
helper context ATL!VariableDeclaration def: immediateComposite : ATL!Element =
        if not self.letExp.oclIsUndefined() then
                self.letExp
        else if not self.baseExp.oclIsUndefined() then
                self.baseExp
        else if thisModule.allInPatternElts->exists(e | e = self) then
                thisModule.allInPatternElts->select(e | e = self)->first().inPattern
        else if thisModule.allOutPatternElts->exists(e | e = self) then
                thisModule.allOutPatternElts->select(e | e = self)->first().outPattern
        else if thisModule.allLoopExps
                                ->exists(l | l.iterators->exists(e | self = e))     then
                thisModule.allLoopExps
                        ->select(l | l.iterators->exists(e | self = e))->first()
        else if thisModule.allIterateExps->exists(e | self = e.result) then
                thisModule.allIterateExps->select(e | self = e.result)->first()
        else if thisModule.allRules
                                ->exists(r | r.variables->exists(e | self = e)) then
                thisModule.allRules
                        ->select(r | r.variables->exists(e | self = e))
                        ->first()
        else OclUndefined
        endif endif    endif endif endif endif endif;


-- This helper returns the immediate composite (container) of the contextual
-- OclExpression.
-- If the one of the "ifExp1", "ifExp2" and "ifExp3" references of the
-- contextual OclExpression is not undefined, the helper returns the pointed
-- IfExp.
-- Otherwise, if its "attribute" is not undefined, the helper returns the
-- pointed Attribute.
-- Otherwise, if its "operation" is not undefined, the helper returns the
-- pointed Operation.
-- Otherwise, if its "initializedVariable" is not undefined, the helper returns
-- the pointed VariableDeclaration.
-- Otherwise, if its "parentOperation" is not undefined, the helper returns the
-- pointed OperationCallExp.
-- Otherwise, if its "loopExp" is not undefined, the helper returns the pointed
-- LoopExp.
-- Otherwise, if its "letExp" is not undefined, the helper returns the
-- pointed LetExp.
-- Otherwise, if its "collection" is not undefined, the helper returns the
-- pointed CollectionExp.
-- Otherwise, if its "appliedProperty" is not undefined, the helper returns the
-- pointed PropertyCallExp.
-- Otherwise, if its "operation" is not undefined, the helper returns the
-- pointed Operation.
-- Otherwise, if there exists an InPattern that has the contextual OclExp as
-- filter, the helper returns this InPattern.
-- Otherwise, if there exists a Binding that has the contextual OclExp as
-- value, the helper returns this Binding.
-- Otherwise, if there exists a Query that has the contextual OclExp as body,
-- the helper returns this Query.
-- Otherwise, the helper retuns OclUndefined as default value.
-- CONTEXT:    ATL!OclExpression
-- RETURN:     ATL!Element
helper context ATL!OclExpression def: immediateComposite : ATL!Element =
        if not self.ifExp1.oclIsUndefined() then self.ifExp1
        else if not self.ifExp2.oclIsUndefined() then self.ifExp2
        else if not self.ifExp3.oclIsUndefined() then self.ifExp3
        else if not self."attribute".oclIsUndefined() then self."attribute"
        else if not self."operation".oclIsUndefined() then self."operation"
        else if not self.initializedVariable.oclIsUndefined()
                then self.initializedVariable
        else if not self.parentOperation.oclIsUndefined() then self.parentOperation
        else if not self.loopExp.oclIsUndefined() then self.loopExp
        else if not self.letExp.oclIsUndefined() then self.letExp
        else if not self.collection.oclIsUndefined() then self.collection
        else if not self.appliedProperty.oclIsUndefined() then self.appliedProperty
        else if thisModule.allInPatterns->exists(e | e.filter = self)
                then thisModule.allInPatterns->select(e | e.filter = self)->first()
        else if thisModule.allBindings->exists(e | e.value = self)
                then thisModule.allBindings->select(e | e.value = self)->first()
        else
                if not thisModule.queryElt.oclIsUndefined()
                then
                        if thisModule.queryElt.body = self
                        then
                                thisModule.queryElt
                        else
                                OclUndefined
                        endif
                else
                        OclUndefined
                endif
        endif endif endif endif endif endif endif
        endif endif endif endif endif endif;


-- This helper computes a sequence containing the VariableDeclarations that
-- precede the contextual VariableDeclaration in its namespace.
-- If the contextual VariableDeclaration is a PatternElement, the helper only
-- returns this VD.
-- Otherwise, it computes the container of the contextual VD. If the container
-- is a LetExp, it returns a Sequence composed of the VD, and the results of
-- the calls of the getUpD helper on the calculated container.
-- If the container is an IteratorExp, the helper returns a Sequence composed
-- of the VD and the results of the call of getUpD on the computed container.
-- If the container is an IterateExp, the helper a Sequence containing the same
-- elements that the one computed for an IteratorExp.
-- Otherwise, the helper returns the only contextual VD as default value.
-- CONTEXT:    ATL!VariableDeclaration
-- RETURN:     Sequence(ATL!VariableDeclaration)
helper context ATL!VariableDeclaration
        def: getDeclarations() : Sequence(ATL!VariableDeclaration) =
        if self.oclIsKindOf(ATL!PatternElement)
        then
                Sequence{self}
        else
                let container : ATL!Element = self.immediateComposite in
                if container.oclIsTypeOf(ATL!LetExp)
                then
                        Sequence{
                                self,
                                container.getUpD()
                        }->flatten()
                else
                        if container.oclIsTypeOf(ATL!IteratorExp)
                        then
                                Sequence{
                                        self,
                                        container.getUpD()
                                }->flatten()
                        else
                                if container.oclIsTypeOf(ATL!IterateExp)
                                then
                                        Sequence{
                                                self,
                                                container.getUpD()
                                        }->flatten()
                                else
                                        Sequence{
                                                self
                                        }->flatten()
                                endif
                        endif
                endif
        endif;


-- This helper computes a sequence containing the VariableDeclarations that are
-- defined higher than the contextual OclExpression in its namespace tree.
-- The helper first computes the container of the contextual OclExp. If this
-- container is undefined, it retuns an empty sequence.
-- Otherwise, if this container is not an OclExpression:
--  * If the container is a RuleVariableDeclaration, the helper returns a
--    sequence containing all the named elements of the rule that contains this
--    InPattern.
--  * If the container is a Binding, the helper returns a sequence containing
--    all the named elements of the rule that contains this Binding.
-- Otherwise, if the computed container is an OclExpression:
--  * If the container is a LetExp, the helper returns a sequence composed of
--    the LetExp variable and 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, if the contextual OclExp is the
--    source of the IteratorExp then the helper returns the result of its
--    recursive call on the IteratorExp, else it returns this result with the
--    "iterators" elements of the IteratorExp.
--  * If the container is an IterateExp, the helper returns the same sequences
--    that for an IteratorExp, with the additional "result" element in case the
--    contextual OclExp is not the source of the IterateExp.
-- Otherwise, the helper returns an empty sequence as default value.
-- CONTEXT:    ATL!OclExpression
-- RETURN:     Sequence(ATL!VariableDeclaration)
helper context ATL!OclExpression
        def: getUpD() : Sequence(ATL!VariableDeclaration) =
        let container : ATL!Element = self.immediateComposite in
        if container.oclIsUndefined() then
                Sequence{}
        else if not container.oclIsKindOf(ATL!OclExpression) then
                if container.oclIsTypeOf(ATL!RuleVariableDeclaration)
                then
                        Sequence{
                                container."rule".namedElements
                        }->flatten()
                else
                        if container.oclIsTypeOf(ATL!Binding)
                        then
                                Sequence{
                                        container.outPatternElement."rule".namedElements
                                }->flatten()
                        else
                                Sequence{}
                        endif
                endif
        else if container.oclIsTypeOf(ATL!LetExp) then
                Sequence{
                        container.variable,
                        container.getUpD()
                }->flatten()
        else if container.oclIsTypeOf(ATL!IfExp) then
                Sequence{
                        container.getUpD()
                }->flatten()
        else if container.oclIsTypeOf(ATL!IteratorExp) then
                if container.source = self
                then
                        Sequence{
                                container.getUpD()
                        }->flatten()
                else
                        Sequence{
                                container.iterators,
                                container.getUpD()
                        }->flatten()
                endif
        else if container.oclIsTypeOf(ATL!IterateExp) then
                if container.source = self
                then
                        Sequence{
                                container.getUpD()
                        }->flatten()
                else
                        Sequence{
                                container.iterators,
                                container.result,
                                container.getUpD()
                        }->flatten()
                endif
        else Sequence{}
        endif endif endif endif endif endif;


-- This helper returns the root composite (container) of the contextual
-- OclExpression. For this purpose, the helper first computes the immediate
-- composite of the contextual OclExpression.
-- If this container is undefined, the helper returns OclUndefined.
-- Otherwise, if it is a kind of OclExpression, the helper returns the value
-- provided by its recursive call on the computed container.
-- Finally, if this container is not an OclExpression, the root composite has
-- been reached (Binding/InPattern/Operation/Query/Attribute) and is returned.
-- CONTEXT:     ATL!OclExpression
-- RETURN:      ATL!Element
helper context ATL!OclExpression def: getRootComposite() : ATL!Element =
        let container : ATL!Element = self.immediateComposite
        in
        if container.oclIsUndefined()
        then
                OclUndefined
        else
                if container.oclIsKindOf(ATL!OclExpression)
                then
                        container.getRootComposite()
                else
                        container
                endif
        endif;


--------------------------------------------------------------------------------
-- RULES -----------------------------------------------------------------------
--------------------------------------------------------------------------------

-- Rule 'FreeVariableIsSelfOrThisModule'
-- This rule generates an 'error' Problem for each VariableDeclaration that has
-- no composite, and whose name is different from both 'self' and 'thisModule'.
-- The VariableExps that have not been previously declared in an ATL file are
-- associated with a new VariableDeclaration without any composite in the
-- correspoding ATL model.
rule FreeVariableIsSelfOrThisModule {
        from
                s : ATL!VariableDeclaration (
                        s.immediateComposite.oclIsUndefined() and
                        s.varName <> 'self' and s.varName <> 'thisModule'
                )
        to
                t : Problem!Problem (
                        severity <- #error,
                        location <-
                                if s.variableExp->isEmpty()
                                then
                                        s.location
                                else
                                        s.variableExp->first().location
                                endif,
                        description <- 'variable \'' + s.varName + '\' undefined'
                )
}

-- Rule 'ModelNameIsUnique'
-- This rule generates an 'error' Problem when there exists models that have
-- the same name that the checked model.
rule ModelNameIsUnique {
        from
                s : ATL!OclModel (
                        thisModule.allModels->exists(e | e.name = s.name and e <> s)
                )
        to
                t : Problem!Problem (
                        severity <- #error,
                        location <- s.location,
                        description <- 'model \'' + s.name + '\' already defined'
                )
}

-- Rule 'RuleNameIsUnique'
-- This rule generates an 'error' Problem when there exists rules that have
-- the same name that the checked rule.
rule RuleNameIsUnique {
    from
        s : ATL!Rule (
            thisModule.allRules->exists(e | e.name = s.name and e <> s)
        )
    to
        t : Problem!Problem (
            severity <- #error,
            location <- s.location,
            description <- 'rule \'' + s.name + '\' already defined'
        )
}

-- Rule 'HelperSignatureIsUnique'
-- This rule generates an 'error' Problem when there exists helpers that have
-- the same signature that the checked helper.
-- Note that in current implementation, the helper signature corresponds to the
-- name and the context of the helper.
rule HelperSignatureIsUnique {
    from
        s : ATL!Helper (
            thisModule.allHelpers
                ->exists(e |
                    e <> s and
                    s.definition.feature.name = e.definition.feature.name and
                    (
                     if not s.definition.context_.oclIsUndefined()
                     then
                        if not e.definition.context_.oclIsUndefined()
                        then
                        if not
s.definition.context_.context_.name.oclIsUndefined()
                            then
                            if not
e.definition.context_.context_.name.oclIsUndefined()
                                then

    s.definition.context_.context_.name = e.definition.context_.context_.name
                                else
                                    false
                                endif
                            else

    e.definition.context_.context_.name.oclIsUndefined()
                            endif
                        else
                            false
                        endif
                     else
                        e.definition.context_.oclIsUndefined()
                     endif
                    )
                )
        )
    to
        t : Problem!Problem (
            severity <- #error,
            location <- s.location,
            description <- 'helper \'' + s.definition.feature.name
                            + '\' already defined'
        )
}

-- Rule 'BindingNameIsUniqueInPattern'
-- This rule generates an 'error' Problem when there exists, in a same pattern,
-- bindings that have the same name that the checked binding.
rule BindingNameIsUniqueInPattern {
        from
                s : ATL!Binding (
                        s.outPatternElement.bindings
                                ->exists(e | e.propertyName = s.propertyName and e <> s)
                )
        to
                t : Problem!Problem (
                        severity <- #error,
                        location <- s.location,
                        description <-
                                'binding \'' + s.propertyName + '\' already defined in pattern'
                )
}

-- Rule 'PatternNameIsUniqueInRule'
-- This rule generates an 'error' Problem when there exists, in a same rule,
-- some named elements (InPatternElement/OutPatternElement/
-- RuleVariableDeclaration) that have the same name that the checked pattern.
rule PatternNameIsUniqueInRule {
        from
                s : ATL!PatternElement (
                        s."rule".namedElements
                                ->exists(e | e.varName = s.varName and e <> s)
                )
        to
                t : Problem!Problem (
                        severity <- #error,
                        location <- s.location,
                        description <-
                                'pattern or variable named \''
                                + s.varName    + '\' already defined in rule'
                )
}

-- Rule 'VariableNameIsUniqueInRule'
-- This rule generates an 'error' Problem when there exists, in a same rule,
-- some named elements (InPatternElement/OutPatternElement/
-- RuleVariableDeclaration) that have the same name that the checked rule
-- variable declaration.
rule VariableNameIsUniqueInRule {
        from
                s : ATL!RuleVariableDeclaration (
                        s."rule".namedElements
                                ->exists(e | e.varName = s.varName and e <> s)
                )
        to
                t : Problem!Problem (
                        severity <- #error,
                        location <- s.location,
                        description <-
                                'pattern or variable named \'' + s.varName
                                + '\' already defined in rule'
                )
}

-- Rule 'NoHelperWithCollectionAsContext'
-- This rule generates an 'error' Problem for each Helper defined with a
-- collection type as context.
-- Note that this problem is due to the limitations of the current
-- implementation
rule NoHelperWithCollectionAsContext {
        from
                s : ATL!Helper (
                        if s.definition.context_.oclIsUndefined()
                        then
                                false
                        else
                                thisModule.collectionTypes
                                        ->exists(e | s.definition.context_.context_ = e)
                        endif
                )
        to
                t : Problem!Problem (
                        severity <- #error,
                        location <- s.location,
                        description <-
                                'helper \'' + s.definition.feature.name
                                        + '\': current implementation does not '
                                        + 'support helpers with collection context'
                )
}

-- Rule 'NoSelfOrThisModuleVariableDeclaration'
-- This rule generates an 'error' Problem for each declaration of a variable
-- named 'self' or 'thisModule' in the ATL program.
-- Considered variable declarations must have a non-undefined immediate
-- composite since the input ATL model may already include a 'self' and a
-- 'thisModule' VD without any immediate composite that correspond to the
-- global declarations of the 'self' and 'thisModule' variables.
rule NoSelfOrThisModuleVariableDeclaration {
        from
                s : ATL!VariableDeclaration (
                        not s.immediateComposite.oclIsUndefined() and
                        (s.varName = 'self' or s.varName = 'thisModule')
                )
        to
                t : Problem!Problem (
                        severity <- #error,
                        location <- s.location,
                        description <-
                                'helper \'' + s.varName      + '\' is not valid variable name'
                )
}

-- Rule 'NoSelfVariableInRule'
-- This rule generates an 'error' Problem for each 'self' variable expression
-- that is contained by a rule element.
rule NoSelfVariableInRule {
        from
                s : ATL!VariableExp (
--                      s.referredVariable.varName = 'self' and
--                      (
--                              let rComp : ATL!Element = s.getRootComposite() in
--                              rComp.oclIsTypeOf(ATL!Binding) or
--                              rComp.oclIsTypeOf(ATL!InPattern)
--                      )
                        if s.referredVariable.oclIsUndefined()
                        then
                                false
                        else
                                s.referredVariable.varName = 'self' and
                                (
                                        let rComp : ATL!Element = s.getRootComposite() in
                                        rComp.oclIsTypeOf(ATL!Binding) or
                                        rComp.oclIsTypeOf(ATL!InPattern)
                                )
                        endif
                )
        to
                t : Problem!Problem (
                        severity <- #error,
                        location <- s.location,
                        description <-
                                'rule \'' + s.referredVariable.varName
                                + '\': use of the \'self\' variable prohibited in rules'
                )
}

```

```
-- Rule 'NoResolveTempInSourcePattern'
-- This rule generates an 'error' Problem for each call of the
-- 'thisModule.resolveTemp()' operation within a source pattern of a rule.
rule NoResolveTempInSourcePattern {
        from
                s : ATL!OperationCallExp (
                        s.operationName = 'resolveTemp' and
                        (
                                if s.source.oclIsTypeOf(ATL!VariableExp)
                                then
                                        if s.source.referredVariable.oclIsUndefined()
                                        then
                                                false
                                        else
                                                s.source.referredVariable.varName = 'thisModule'
                                        endif
                                else
                                        false
                                endif
                        ) and
                        s.getRootComposite().oclIsTypeOf(ATL!InPattern)
                )
        to
                t : Problem!Problem (
                        severity <- #error,
                        location <- s.location,
                        description <-
                                'rule \'' + s.getRootComposite()."rule".name
                                + '\': use of \'thisModule.resolveTemp()\' function '
                                + 'is prohibited in source patterns'
                )
}

-- Rule 'NoResolveTempInModuleAttribute'
-- This rule generates an 'error' Problem for each call of the
-- 'thisModule.resolveTemp()' operation within a model attribute.
rule NoResolveTempInModuleAttribute {
        from
                s : ATL!OperationCallExp (
                        s.operationName = 'resolveTemp' and
                        (
                                if s.source.oclIsTypeOf(ATL!VariableExp)
                                then
                                        if s.source.referredVariable.oclIsUndefined()
                                        then
                                                false
                                        else
                                                s.source.referredVariable.varName = 'thisModule'
                                        endif
                                else
                                        false
                                endif
                        ) and
                        s.getRootComposite().oclIsTypeOf(ATL!Attribute)
                )
        to
                t : Problem!Problem (
                        severity <- #error,
                        location <- s.location,
                        description <-
                                'attribute \'' + s.getRootComposite().name
                                + '\': use of \'thisModule.resolveTemp()\' function '
                                + 'is prohibited in attributes'
                )
}

-- Rule 'ProhibitedMultiIteratorCollectionOperation'
-- This rule generates an 'error' Problem for each IteratorExp of the
-- singleIteratorExps set that is associated with several Iterators.
```

```
rule ProhibitedMultiIteratorCollectionOperation {
        from
                s : ATL!IteratorExp (
                        thisModule.singleIteratorExps->exists(e | s.name = e) and
                        s.iterators->size() > 1
                )
        to
                t : Problem!Problem (
                        severity <- #error,
                        location <- s.location,
                        description <-
                                'iterator \'' + s.name
                                + '\' may have at most one iterator variable'
                )
}

-- Rule 'UnsupportedMultiIteratorCollectionOperation'
-- This rule generates an 'error' Problem for each IteratorExp of the
-- multiIteratorExps set that is associated with several Iterators.
-- Note that this problem is due to limitations of the current implementation.
rule UnsupportedMultiIteratorCollectionOperation {
        from
                s : ATL!IteratorExp (
                        thisModule.multiIteratorExps->exists(e | s.name = e) and
                        s.iterators->size() > 1
                )
        to
                t : Problem!Problem (
                        severity <- #error,
                        location <- s.location,
                        description <-
                                'with current implementation, iterator \'' + s.name
                                + '\' may have at most one iterator variable'
                )
}

-- Rule 'ParameterNameIsUniqueInOperation'
-- This rule generates an 'error' Problem for each parameter for which there
-- exists another parameter of the same name in the operation declaration.
rule ParameterNameIsUniqueInOperation {
        from
                s : ATL!Parameter (
                        s.operation.parameters
                                ->exists(e | s.varName = e.varName and s <> e)
                )
        to
                t : Problem!Problem (
                        severity <- #error,
                        location <- s.location,
                        description <-
                                'a parameter named \'' + s.varName
                                + '\' is already declared in this operation'
                )
}

-- Rule 'IteratorNameIsUniqueInLoop'
-- This rule generates an 'error' Problem for each Iterator declaration for
-- which there exists either another Iterator or a result variable declaration
-- (for Iterate loop only) of the same name within the same loop definition.
rule VariableNameIsUniqueInLoop {
        from
                s : ATL!Iterator (
                        s.loopExpr.iterators
                                ->exists(e | s.varName = e.varName and s <> e)
                        or
                        if s.loopExpr.oclIsTypeOf(ATL!IterateExp)
                        then
                                s.loopExpr.result.varName = s.varName
                        else
```

```
                                false
                        endif
                )
        to
                t : Problem!Problem (
                        severity <- #error,
                        location <- s.location,
                        description <-
                                'a variable named \'' + s.varName
                                + '\' is already declared in this loop'
                )
}

-- Rule 'ResultNameIsUniqueInIterate'
-- This rule generates an 'error' Problem for each 'result' variable
-- declaration of an IterateExp for which there exists an Iterator variable of
-- the same name in the Iterate loop definition.
rule ResultNameIsUniqueInIterate {
        from
                s : ATL!VariableDeclaration (
                        if s.baseExp.oclIsUndefined()
                        then
                                false
                        else
                                s.baseExp.iterators
                                        ->exists(e | s.varName = e.varName and s <> e)
                        endif
                )
        to
                t : Problem!Problem (
                        severity <- #error,
                        location <- s.location,
                        description <-
                                'a variable named \'' + s.varName
                                + '\' is already declared in this loop'
                )
}

-- Rule 'VariableNameIsUniqueInContainer'
-- This rule generates a 'warning' Problem for each declaration of a variable
-- for which there exists another variable declaration of the same name in the
-- same namespace (except multiple intances of an Iterator name in a same loop
-- which handle 'error' Problems).
rule VariableNameIsUniqueInContainer {
        from
                s : ATL!VariableDeclaration (
                        s.getDeclarations()->exists(e | s.varName = e.varName and s <> e)
                )
        to
                t : Problem!Problem (
                        severity <- #warning,
                        location <- s.location,
                        description <-
                                'a variable named \'' + s.varName
                                + '\' is already declared in this container'
                )
}
```