Static Semantics Checking Tool for jUCMNav

Jun Biao Yan

Final project report for Intensive Graduate Project (CSI6900)

Supervisor: Daniel Amyot

University of Ottawa
Ottawa, Ontario, Canada
March 2008

Table of Contents

List of Figures

List of Acronyms

Acronym	Definition
API	Application Programming Interface
GRL	Goal-oriented Requirements Language
GUI	Graphical User Interface
ITU	International Telecommunication Union
MDT OCL	Model Development Tools OCL component
OCL	Object Constraint Language
OMG	Object Management Group
UCM	Use Case Maps
URN	User Requirements Notation

Chapter 1. Introduction

jUCMNav is an Eclipse plug-in that supports the graphical editing and analysis of User Requirements Notation (URN) models [10][11]. URN itself is composed of two complementary languages: the Use Case Map (UCM) scenario notation and the Goal-oriented Requirements Language (GRL) [8][9]. jUCMNav supports some analysis features for executing UCM models (scenario definitions and traversal algorithms) and evaluating GRL models (strategies and propagation algorithms). jUCMNav already reports some issues found during execution/propagation in Eclipse's Problems view (see snapshot below Figure 1). Although the tool prevents users from creating syntactically invalid models, there are still many rules related to the static semantics of the URN language that are not enforced, especially when considering the various modeling styles required by export filters to other notations (Message Sequence Charts, Core Scenario Model, etc.) or by advanced concepts (Key Performance Indicators, Aspect-oriented URN, etc.).

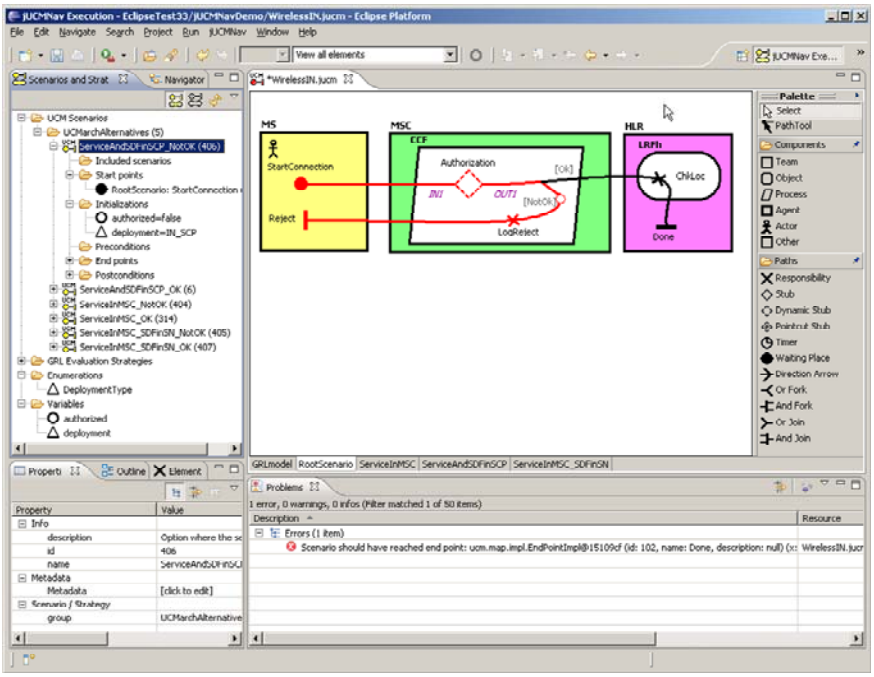

Figure 1 Snapshot of jUCMNav

Rules should refer to the underlying representation of URN models in jUCMNav, which is based on the metamodel partly described in Appendix A. Some of the main UCM and GRL element definitions in this metamodel can be found in Annex A. Rules could, to some extent, be seen as OCL constraints on this metamodel that could be switched on or off and evaluated upon demand.

Sample static semantics and verification rules could include:

- There should not be containment cycles in UCM components
- There should not be containment cycles in GRL actors
- There should not be unknown contributions in GRL models
- All UCM responsibility and component definitions should have a non-empty description
- All GRL tasks should have a URN link to at least one UCM element

The major requirements can be summarized as the following:

1. The system shall allow users to enable static semantics checking rules.
2. The system shall allow users to disable static semantics checking rules.
3. The system shall check, upon the user's request, all enabled static semantics checking rules on the URN model that is being opened in jUCMNav.
4. The system shall report rules violations to the user.
5. The system should allow defined rules to be changed without the need to recompile the jUCMNav tool.
6. The system should allow new rules to be added without the need to recompile the jUCMNav tool.
7. The system shall allow a user to save a rule or rules into a file.
8. The system shall allow a user to load a rule or rules saved in a file into the system.
9. The system shall allow users to create groups to organize rules.
10. The system shall allow users to delete groups.
11. The system shall allow users to put a rule in any group.
12. The system shall allow users to remove a rule from a group.

In our proposed solution, OMG's Object Constraint Language (OCL) [1][2] will be used to express user-defined rules. We will take advantage of an existing Eclipse plug-in that supports OCL [6][7] in our solution for the verification of URN models against these rules.

The rest of this document is structured as follows. Section 2 gives an overview of the architecture of our extension to jUCMNav, and in particular to how the static semantic rules are created, edited, grouped, stored, and checked, with appropriate menu items and dialog windows. Section 3 presents the main issues faced during the design of our tool and the solutions retained. Section 4 validates the tool with a set of sample rules. Section 5 follows with our conclusions.

Chapter 2. Architecture

This tool is composed of three packages:

- seg.jUCMNav.staticSemantic: the core part. It provides rule definition, rule group definition, evaluation of rules, rules import/export and rules/groups persistence.
- seg.jUCMNav.actions.staticSemantic: contains an Eclipse menu item action delegate, forwarding the request of checking rules.
- seg.jUCMNav.views.preferences.staticSemantic: contains GUI components on Preference View, allowing users to use all functionalities in seg.jUCMNav.staticSemantic

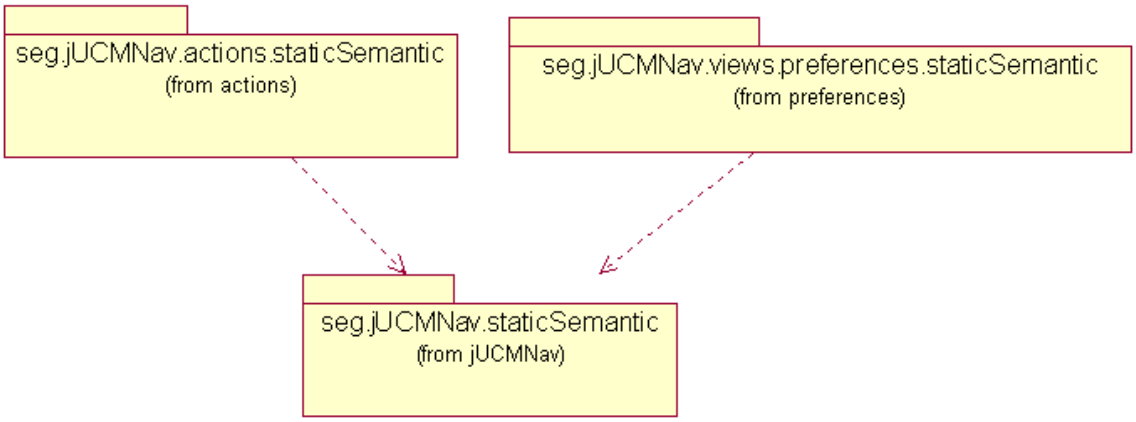

Figure 2 Class Diagram – Top Level

2.1. seg.jUCMNav.actions.staticSemantic

seg.jUCMNav.actions.staticSemantic contains an Eclipse Editor Action delegate VerifyStaticSemanticDelegate which responds to a select event of the Eclipse menu item "Verify Static Semantics"(see **Error! Reference source not found.**) and then invokes the check() method of class StaticSemanticChecker to achieve the checking work.

In the class VerifyStaticSemanticDelegate, the method run() does the delegation:

```
public void run(IAction action) {
   if (editor!=null) {
      Vector problems = new Vector();
      StaticSemanticChecker.getInstance().check(
               editor.getModel(),problems);
      refreshProblemView(problems);
   }
}
```

Another important method is refreshProblemView(), which shows checking result in the Eclipse Problem view.

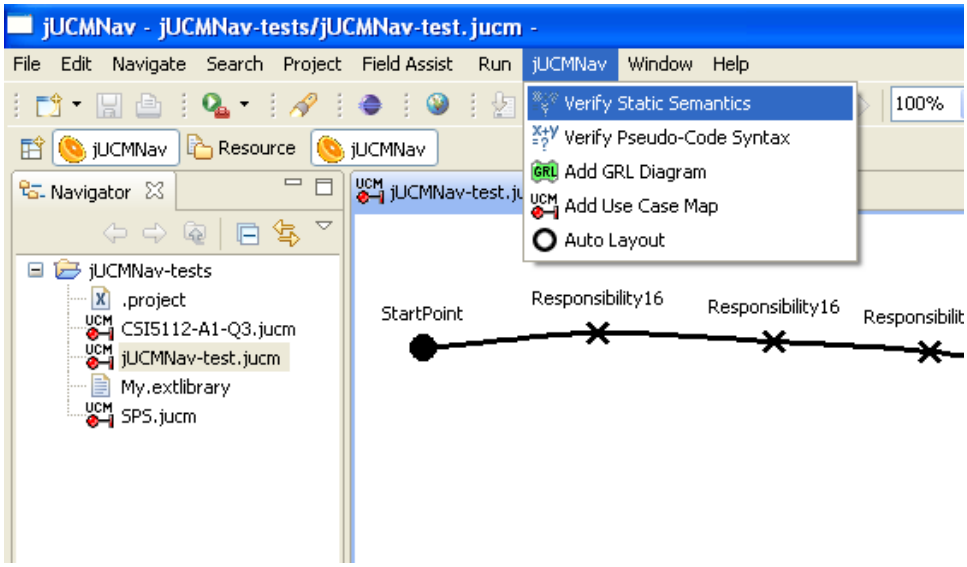

Figure 3 Menu Item- Verify Static Semantics

2.2. seg.jUCMNav.views.preferences.staticSemantic

This package contains GUI components on the Preference View, allowing users to use all functionalities in seg.jUCMNav.staticSemantic. It has four classes showed in the following Class Diagram (Figure 4). When the preference page of Static Semantic Checking Preferences under jUCMNav preferences in Eclipse Preferences view is selected, the class StaticSemanticPreferencePage is instantiated to show the preference page (see Figure 5). In this page, users can enable or disable rules, create a new rule or group, edit or delete an existing rule or group, import rules from a file and export selected rules into a file.

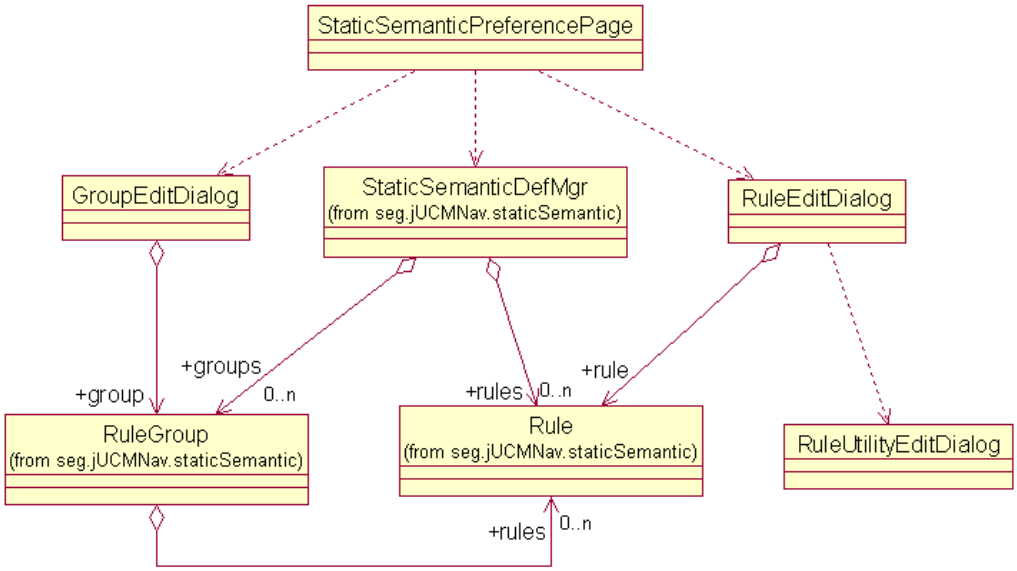

Figure 4 Class Diagram – preferences.staticSemantic

When users choose to create or edit a rule, an instance of class RuleEditDialog is created to show the rule editor (see Figure 6). To edit a rule, a rule object must be attached to the editor by the method setRule(). Another method called getRule() is also provide to return the rule object that is newly created. The GroupEditDialog (Figure 7) and RuleUtilityEditDialog work in the same way.

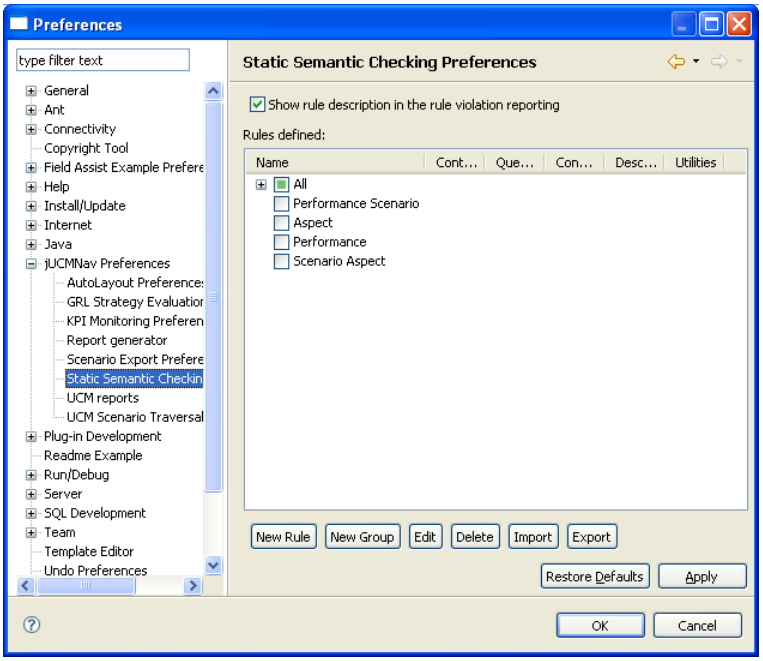

Figure 5 Static Semantic Checking Preferences page

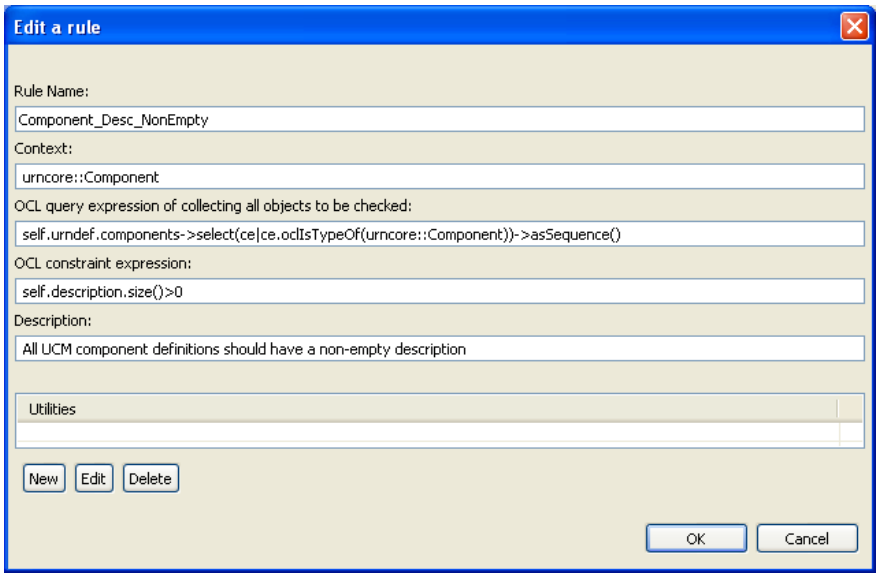

Figure 6 Rule Editor

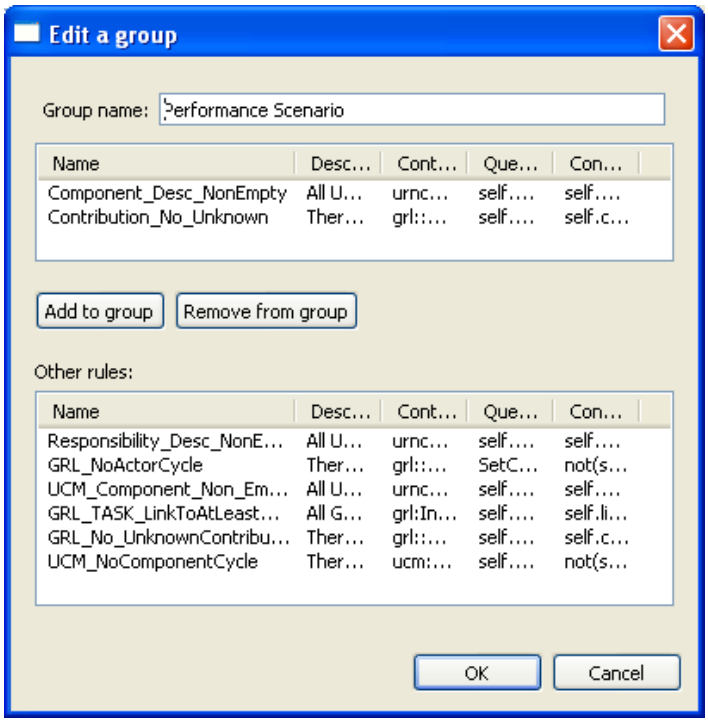

Figure 7 Group Editor

2.3. seg.jUCMNav.staticSemantic

This package is the core part of the checking tool. It provides rule definition, rule group definition, evaluation of rules, rules import/export and rules/groups persistence.

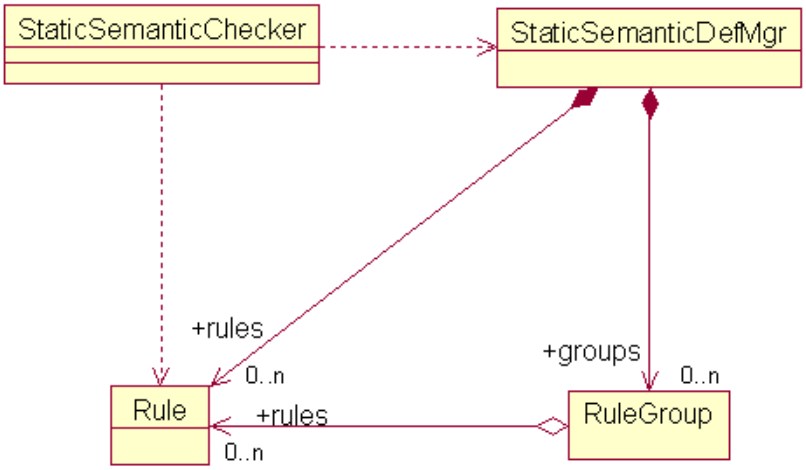

Figure 8 Class Diagram – seg.jUCMNav.staticSemantic

2.3.1 Rule class

The Rule class models a static semantic rule, including the following properties:

1) Rule name
2) Rule classifier
3) OCL context expression
4) OCL invariant expression
5) Rule description
6) Rule utilities
7) Rule enabled/disabled indicator

We will discuss properties 1, 2, 4 and 5 in section 3.1, property 3 in section 3.3 and property 6 in section 3.2. In class Rule, there is a significant method we need to pay more attention, namely the isValid() method. This method checks if properties 2, 3, 4, and 6 are valid or not under the URN metamodel (Appendix A) and standard library (section 3.4). It is important that checking the validation of a rule with this isValid() method before the rule is put into the system.

2.3.2 Group class

The Group class is designed to organize a group of rules in which a particular user is interested. Groups are not partitions of rules. In another word, a rule could be in more than one group. An object of class Group can test if a rule is in the group, add or remove rules into or from the group and return all rules in the group. Another important method of the Group class is enable(boolean), which enables or disables all rules in the group.

2.3.3 StaticSemanticDefMgr

This class is the controller of rules manipulation. Only one instance of this class can be created. This instance contains all rules and groups in the system. It also contains some other configuration information related to rules evaluation.

One of the functionalities of this class is persistence of rules, groups and other configuration information. All information is stored in the Eclipse PreferenceStore.

Another important functionality is importing and exporting rules. Rules exported are saved in XML files. Before rules are imported from XML files, format validation of XML files is performed.

StaticSemanticDefMgr also checks the uniqueness of the name of rules and groups before rules and groups are added to the system.

2.3.4 StaticSemanticChecker

The only purpose of this class is to encapsulate the behaviour of rules checking on a URN model. Therefore, only one instance of this class can be created by invoking the method getInstance(). We can execute rule checking simply by invoking the check() method.

Chapter 3. Issues & Solutions

3.1. Why OCL & MDT OCL

For various applications of URN, although the metamodel is the same, different concerns must be taken into considerations. How to allow users of jUCMNav to describe their own static semantic rules is significant. We need a rule describing language which helps jUCMNav users to specify their static semantics easily. OCL is a good choice because it has some benefits:

- OCL is standardized. This allows developers and customers to share a common language.
- OCL has no side effect. When an OCL expression is evaluated, it simply returns a value. It cannot change anything in the model.
- OCL is a precise and unambiguous language that can be easily read by developers and customers.
- OCL is able to refer to model elements easily.

Take the following static semantic rule as an example. *All UCM responsibility definitions should have a non-empty description.* With OCL, this can be expressed as the following OCL expressions:

```
package urncore
context Responsibility
-- All UCM responsibility definitions should have a non-empty
description
inv not_empty_desc: self.description.size()>0
endpackage
```

This semantic rule has five parts: the package name: *urncore*, the class name: *Responsibility*, the comment, the invariant name *not_empty_desc* and the invariant expression: *self.description.size()>0* . In our design, we combine the package name and the class name into one property called Rule Classifier in the form of Packagename::Classname. As a result, a rule has the following four properties:

- Rule Name: not_empty_desc
- Rule Classifier: urncore::Responsibility
- Rule Invariant Expression: *self.description.size()>0*
- Rule Description: All UCM responsibility definitions should have a non-empty description

In addition, a rule shall be allowed to be enabled or disabled upon the user's request. This results in a fifth property, called Rule Enabled/Disabled indicator, to be put into a rule definition.

Another advantage of OCL is that many implementations of the language have been developed. We can start work from these implementations instead of from scratch. There exist several OCL implementations developed such as MOMENT-OCL [3], EMFOCL [4], RoclET [5] and MDT-OCL [6]. Among these OCL implementations, MDT-OCL is the only one that provides APIs. With APIs, we can more easily incorporate MDT-OCL into the jUCMNav project.

The MDT OCL implementation is defined in plug-ins for convenient deployment in Eclipse, but as is the case for EMF, it can also be used stand-alone. The plug-in is partitioned into the following packages:

- org.eclipse.ocl: the core parsing, evaluation, and content assist services. Definition of the OCL Abstract Syntax Model and Environment API. These APIs are generic, independent of any particular metamodel (though using Ecore/EMF as the meta-meta-model).
- org.eclipse.ocl.ecore: implementation of the Ecore metamodel environment, binding the generic Environment and AST APIs to the Ecore language. Provides support for working with OCL constraints and queries targeting Ecore models.
- org.eclipse.ocl.uml: implementation of the UML metamodel environment, binding the generic Environment and AST APIs to the UML language. Provides support for working with OCL targeting UML models.

In our project, we only use the first two packages because the jUCMNav is based on Ecore model.

To use the MDT OCL, we must create an OCL instance like in the following code snippet:

```
// create an OCL instance for Ecore
OCL ocl = OCL.newInstance(EcoreEnvironmentFactory.INSTANCE);
```

After the OCL instance is created, we need to parse OCL expressions. There are two ways of parsing OCL. We can parse one entire OCL file. In MDT OCL, the OCLInput class encapsulates an OCL file or stream. An input can also be created from a string or an input stream. Given an OCLInput, simply ask an OCL to parse it.

```
OCLInput lib = new
   OCLInput(getClass().getResourceAsStream("library.ocl"));
ocl.parse(lib);
```

We also can parse a separate OCL expression. In MDT OCL, the OCLHelper class is primarily designed for this purpose, providing some APIs. In our project, we use three methods among them. The method **createQuery()** is used to create an OCL query expression.

```
OCLHelper helper = ocl.createOCLHelper();
helper.setContext(UrnPackage.Literals.UR_NSPEC);
OCLExpression query = helper.createQuery(r.getContext());
Query queryEval = ocl.createQuery(query);
```

The method **createInvariant()** is convenient to create an OCL invariant expression.

```
List name = r.getClassifierAsList();
EClassifier e = (EClassifier)
     ocl.getEnvironment().lookupClassifier(name);
helper.setContext(e);
Constraint invariant =
    (Constraint) helper.createInvariant(r.getQuery());
Query constraintEval = ocl.createQuery(invariant);
```

The method **defineOperation()** is useful for defining an additional operation.

```
String op = (String) r.getUtilities().get(k);
helper.setContext(e);
helper.defineOperation(op);
```

The final step is to evaluate the OCL expressions just parsed. For an OCL query expression, this can be done by invoking the method evaluate() of the class Query.

```
List objects = (List) queryEval.evaluate(urn);
```

For an OCL constraint expression, this is done by invoking the method reject() of OCL instance on some objects. As a result, some objects that violate the constraint are returned.

```
List violatedObjs = constraintEval.reject(objects);
```

3.2. Utilities

As discussed in the previous section, a rule has five properties used to describe a particular static semantic aspect of URN. However, in some complex situations, this is not enough. Here is an example in which the invariant expression is difficult to express with a short string: *There should not be containment cycles in GRL actors.*

In the GRL metamodel, more than one actor references can refer to one same actor definition. In Figure 9, there are three actor references which refer to actor definition Actor1, Actor2 and Actor1 from inside to outside respectively. In this case, Actor1 is contained by itself.

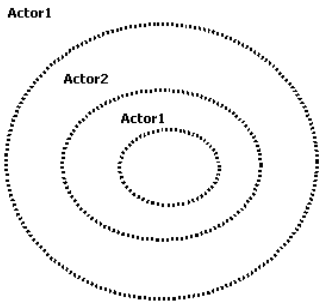

Figure 9 Actor Cycle Case 1

In Figure 10, Actor 1 is also contained by itself.

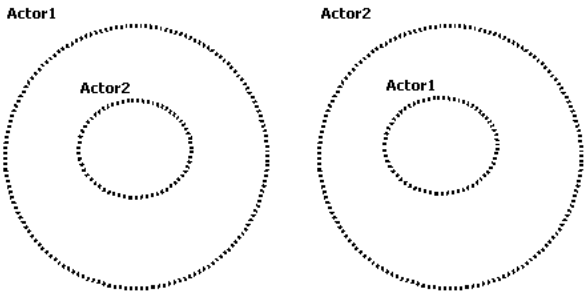

Figure 10 Actor Cycle Case 2

This static semantic rule can be expressed as the following OCL expression, in which two additional operations are defined to simplify the logic of invariant expression No_Actor_ Cycle.

```
package grl
context ActorRef
inv No_Actor_Cycle:
not(self.ancestors()->exists(ar|ar.contDef.oclAsType(Actor)
.id=self.contDef.oclAsType(Actor).id))

def:ancestorSet(current:Set(ActorRef)):Set(ActorRef)=
     let oneStep:Set(ActorRef)=
       current->iterate(ar; result : Set(ActorRef) = Set{} |
          ar.contDef.contRefs->union(
              if ar.parent->notEmpty() then ar.parent->asSet()
              else Set{}
              endif
              )
               ->collect(o|o.oclAsType(ActorRef))->asSet()
           )
       in
       if current->size() < current->union(oneStep)->size()
-- The set gets bigger
       then ancestorSet(current->union(oneStep))
       else current
       endif

def: ancestors():Set(ActorRef)= ancestorSet(
    if self.parent->notEmpty()
    then
       self.parent.contDef.contRefs
             -> union(  self.parent->asSet() )
             -> collect(o|o.oclAsType(ActorRef))->asSet()
    else
       Set{}
    endif
)
endpackage
```

We call the two additional operations in the above example *Utilities*. A rule may have any number of utilities depending on user's will. Utilities are associated with a rule. In other words, a utility is defined under the scope of a rule. This is mainly designed to resolving name conflicting in rules sharing, which is discussed in more details in section 3.6.

3.3. allInstance() of OCL

Creating a rule on a particular type means that each instance of this type must be conform to the constraint that the rule implies. Therefore, to ensure the rule is not violated, we must check if every instance of the type conforms to the constraint. In the OCL

specification [1], there is a convenient way to get all instantiated objects of a type. This is a predefined feature on classes, interfaces and enumerations called allInstances(). However, the implementation of this feature is challenging. To achieve this feature, each construction and destruction of each type must be monitored and recorded. This is as heavy price to pay. Consequently, various implementations of OCL, including MDT OCL, do not implement this feature perfectly.

Fortunately, in our application, we can provide a cheap and flexible way to achieve the same purpose. Based on the metamodel of URN (0), all instances of all types except the URNspec are contained in one instance of URNspec. This instance of URNspec represents an application model. With this structure and an instance of URNspec, we can collect all instances of a particular type by using an OCL navigation path. For example, to collect all instances of the type Responsibility in UCM, from the instance of URNspec, we specify the navigation path *self.urndef.responsibilities*. We call this navigation path Rule Context Expression, which must return a sequence of objects with a type specified by Rule Classifier. Sometimes, rule context expressions are not as simple as the example above but section 3.4 will introduce a way to handle this issue.

3.4. Common utilities

As discussed in section 3.3, a navigation path can and must be specified in a rule definition. However, the metamodel of URN is complicated. For some classes, navigation paths are not easily obtained and even if they can be found, they could be prone to errors for most users. Consequently, it is necessary to provide a mechanism such that some complex navigation paths can be predefined and users can then simply use them. For example, we can define the following utility in the context of class URNspec, where all instances of actor references are collected.

```
package urn
context URNspec
def: SetContextActorRef():Sequence(grl::ActorRef) =
       self.urndef.specDiagrams
          ->select(d|d.oclIsTypeOf(grl::GRLGraph)).contRefs
          ->select(r|r.oclIsTypeOf(grl::ActorRef))
          ->collect(o|o.oclAsType(grl::ActorRef))
          ->asSequence()
endpackage
```

This utility definition is put in the standard library file *library.ocl*, which is actually an OCL file. This kind of utility is available for all rules in the system. Therefore, we call them Common Utilities. With the common utility *SetContextActorRef*, the navigation path to ActorRef can be expressed as *self.SetContextActorRef()*.

3.5. Rule grouping

Rules are created globally. In another word, rules are associated with the jUCMNav tool as a whole, not with a particular application of URN (specifically a .jucm file). Consequently, enabling and disabling rules related to a particular application of URN is a heavy task, especially if the system has hundreds of rules. The feature of rule grouping is provided to solve this problem.

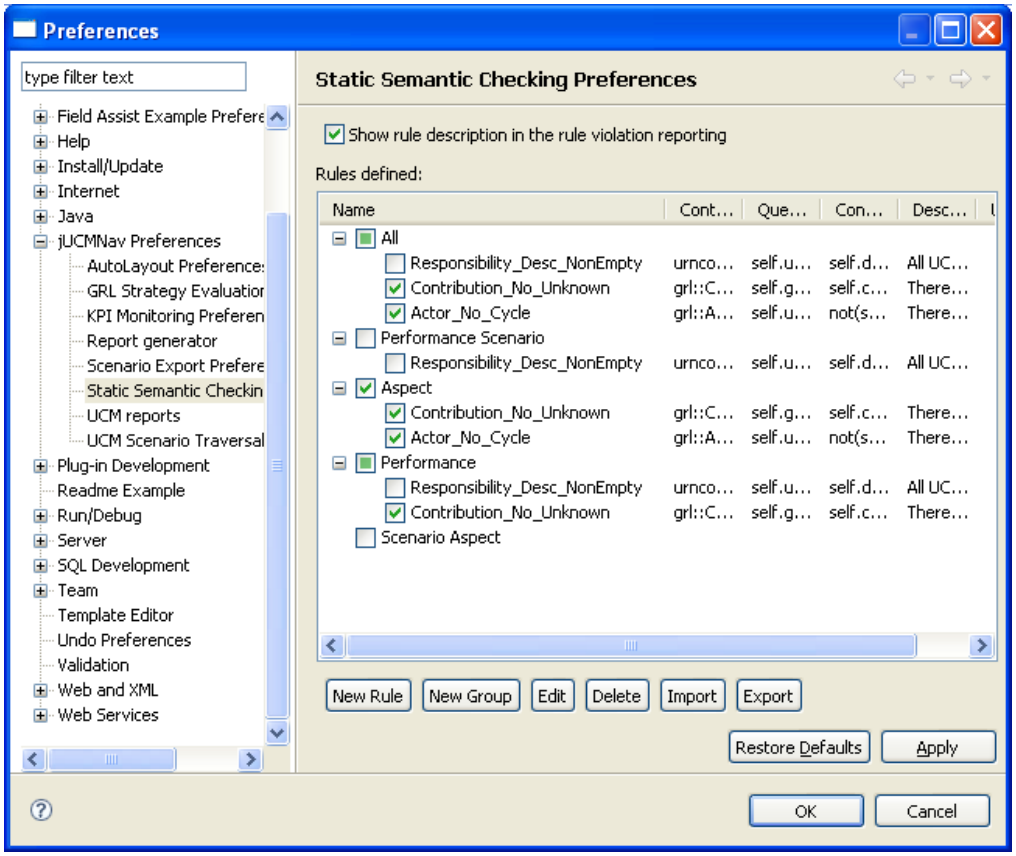

Figure 11 Enable/disable rules by groups

A rule group can be created to contain some rules. All rules in a group can be enabled/disabled easily by enabling/disabling this group. A rule can be included in many

groups. There is a special rule group called All. As the name indicates, this group includes all rules in the system and cannot be deleted.

3.6. Rule sharing

Not every user of jUCMNav knows OCL or the URN metamodel well. Therefore, there is a need that a normal user can easily use an interesting rule created by an OCL expert. Furthermore, a user may want to save his/her rules such that he/she can restore those rules after the jUCMNav crashes or is reinstalled.

Sharing rules is a kind of data exchange between users. Naturally, determining the data format is significant. XML is the best choice as it is standard and mainly designed for data exchange purpose. Furthermore, there are many existing tools that can manipulate XML files, for example, JAXP in Java.

Another concern is what content must be saved in a XML file when exporting a rule. In fact, among the properties of a rule definition, all properties except the Rule Enabled/Disabled Indicator are needed. The following is the XML schema that defines the format of the rule XML file.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
<xs:element name="Rules">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Rule" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
           <xs:element name="Name" type="xs:string"
                     maxOccurs="1" minOccurs="1"/>
           <xs:element name="Description" type="xs:string"
                     maxOccurs="1" minOccurs="1"/>
           <xs:element name="Classification" type="xs:string"
                     maxOccurs="1" minOccurs="1"/>
           <xs:element name="Query" type="xs:string"
                     maxOccurs="1" minOccurs="1"/>
           <xs:element name="Constraint" type="xs:string"
                     maxOccurs="1" minOccurs="1"/>
           <xs:element name="Utilities" maxOccurs="unbounded"
                      minOccurs="0">
              <xs:complexType>
                 <xs:sequence>
                    <xs:element name="Utility" type="xs:string"
```

```
                    minOccurs="0" maxOccurs="unbounded" />
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element></xs:schema>
```

3.7. Name conflict in rule sharing

When we are importing a rule into the system, the system might already have a rule with the same name as the rule being imported. These two rules may actually be the same one, but also may be different. If such name conflict occurs, the rule being imported is renamed to make sure its name remains unique in the system and a warning message is showed. All rules imported are put in a group called *Imported*. Users then can manipulate all rules in a group properly.

Another concern is the conflict of utility names. For example, the system has a rule with a name *RuleA*, which has a utility named *UtilityA*. Now, we are importing a rule with a name *RuleB* which has a utility named *UtilityA* as well. Should this result in a name conflict? It depends. If we treat the *UtilityA* as a common utility, it definitely results in a name conflict. However, if we put *UtilityA* in a scope of a rule rather than in the standard library file, the name conflict does not exist. This is because the two *UtilityA* in the above example are different and are not related with each other. They are only visible for the rule that contains them. To achieve this goal, we place this kind of utilities with their owner rules. From the implementation perspective, when each rule is applied, new instances of OCL and OCLHelper are created to ensure utilities fall under the scope of their owner rule.

Chapter 4. Experiments

To demonstrate and validate the tool, we considered several situations initially expressed as informal rules and then formalized them as OCL rules checked against URN models in our experiments. This section also illustrates how to do these tasks step by step.

4.1. All UCM responsibility definitions should have a non-empty description

First, we go to the Preferences dialog of Eclipse and then choose the Static Semantic Checking Preferences page under the node of jUCMNav Preferences (see Figure 12).

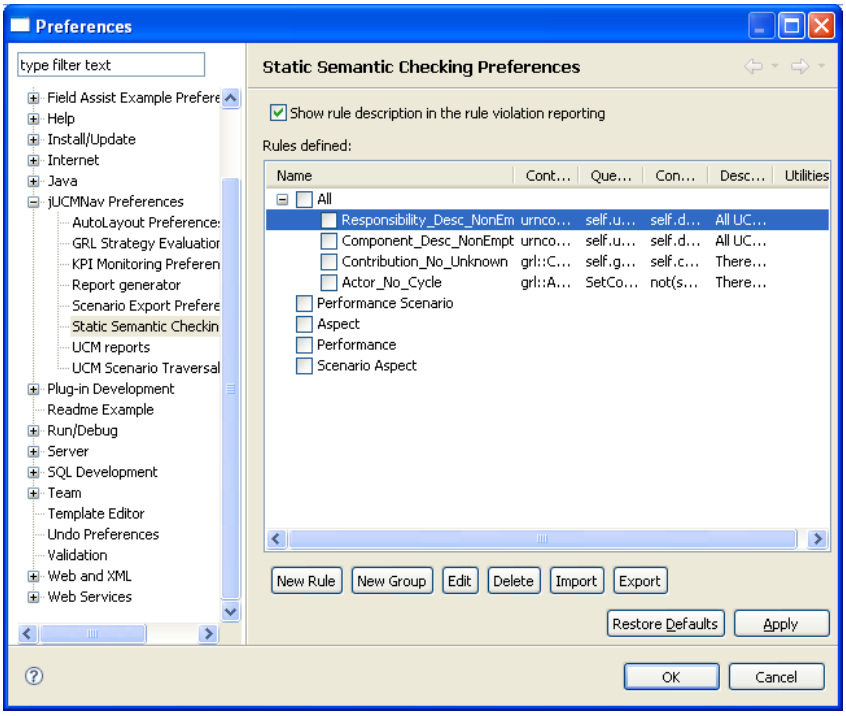

Figure 12 Static Semantic Checking Preferences page

Next, click the button of New Rule and input all properties of a rule definition (see Figure 13).

Next, click OK button and enable the rule under group of All showed in Figure 14.

Finally, we verify the rule by clicking the menu item of Verify Static Semantic under the menu jUCMNav. The result is showed in the Eclipse Problem view (Figure 15).

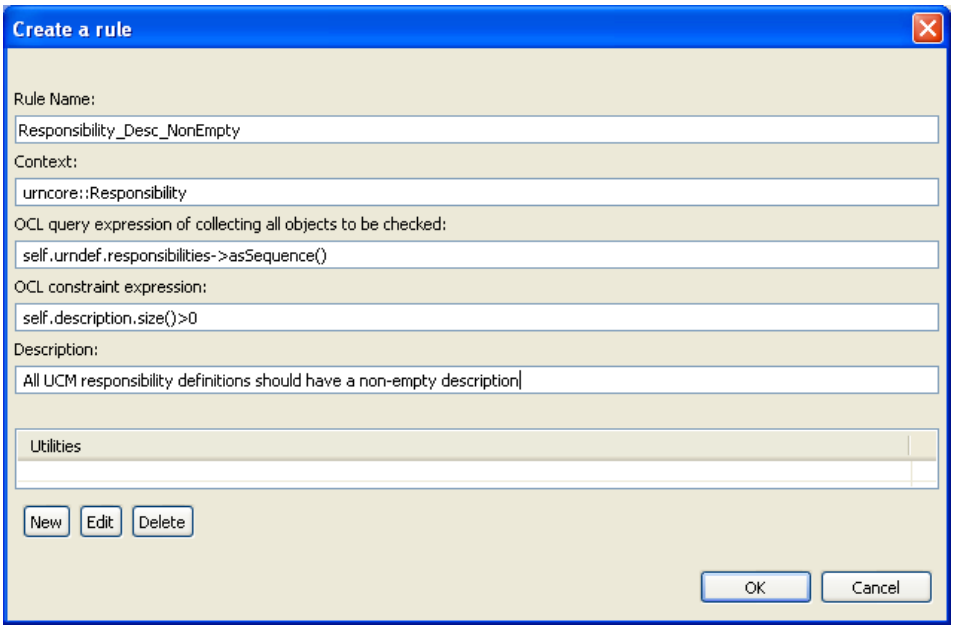

Figure 13 Creating the rule Responsibility_Desc_NonEmpty

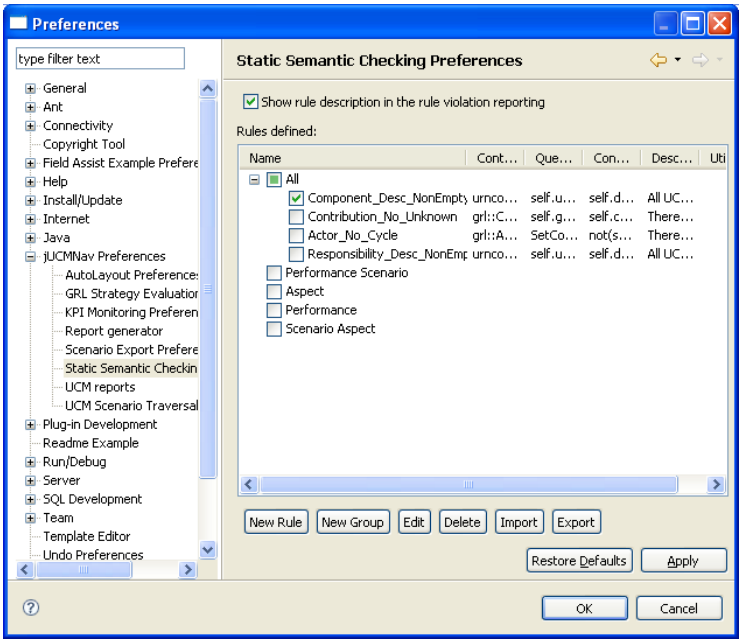

Figure 14 Enable the rule of Responsibility_Desc_NonEmpty

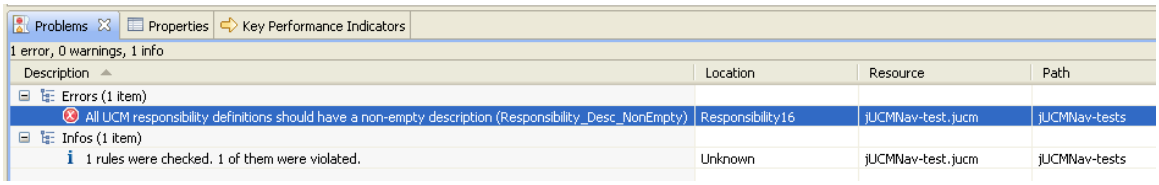

Figure 15 Checking result of Responsibility_Desc_NonEmpty

Double-clicking on the error reported brings us to the responsibility definition that does not have a description (Responsibility16 in this example), in the Outline window.

4.2. All UCM component definitions should have a non-empty description

The rule definition is showed in Figure 16. The checking result is showed in Figure 17.

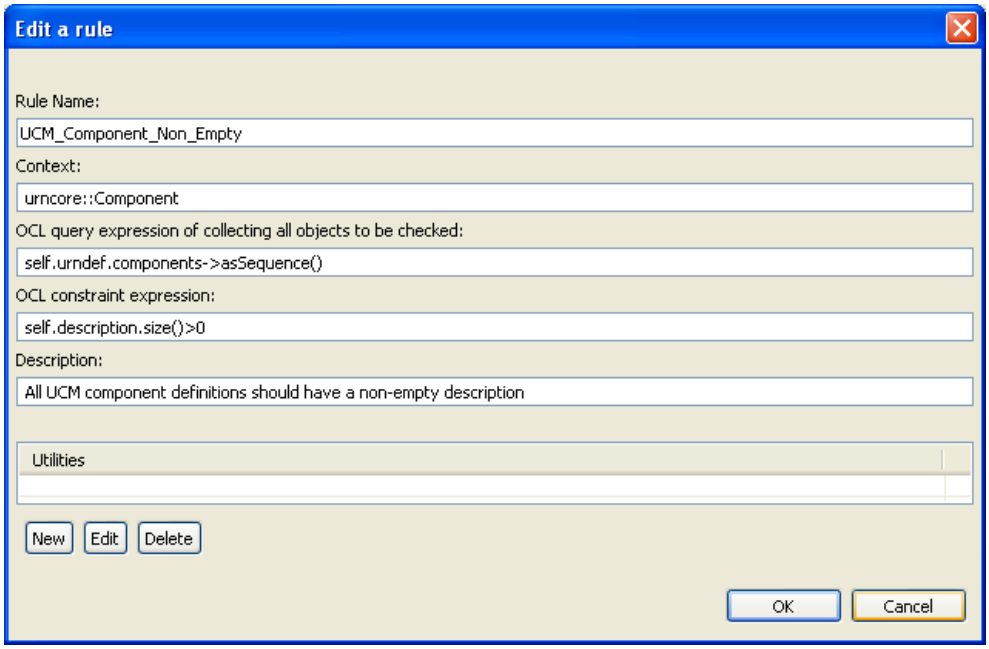

Figure 16 Rule definition of UCM_Component_Non_Empty

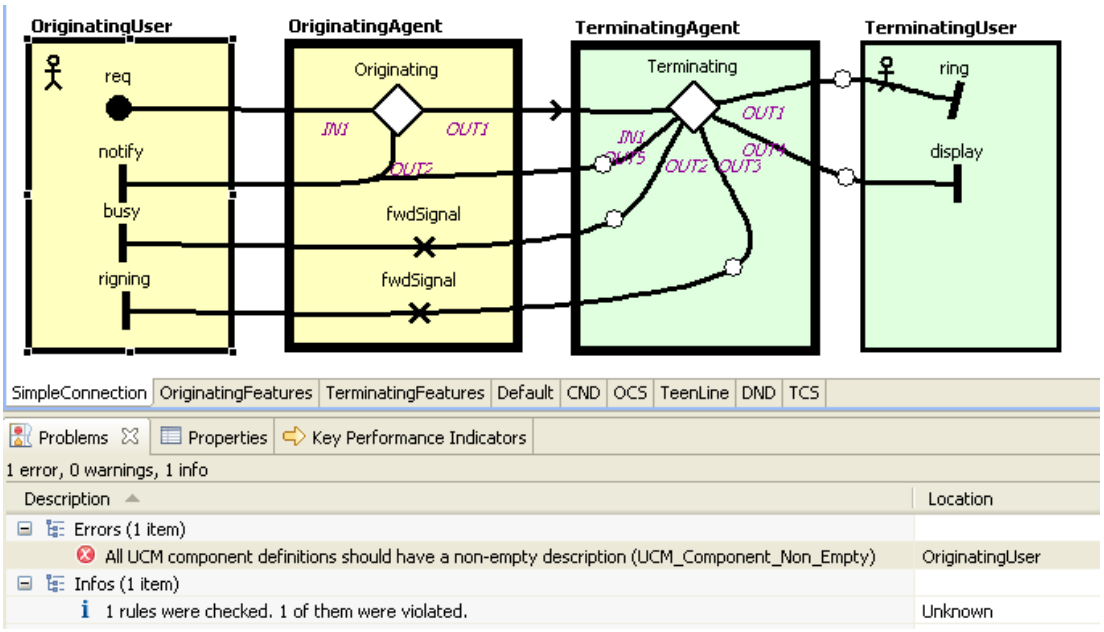

Figure 17 Checking result of UCM_Component_Non_Empty

4.3. There should not be containment cycles in GRL actors

First, as describe in previous sections, we need to open a creation rule dialog window and input the first five properties (see Figure 18).

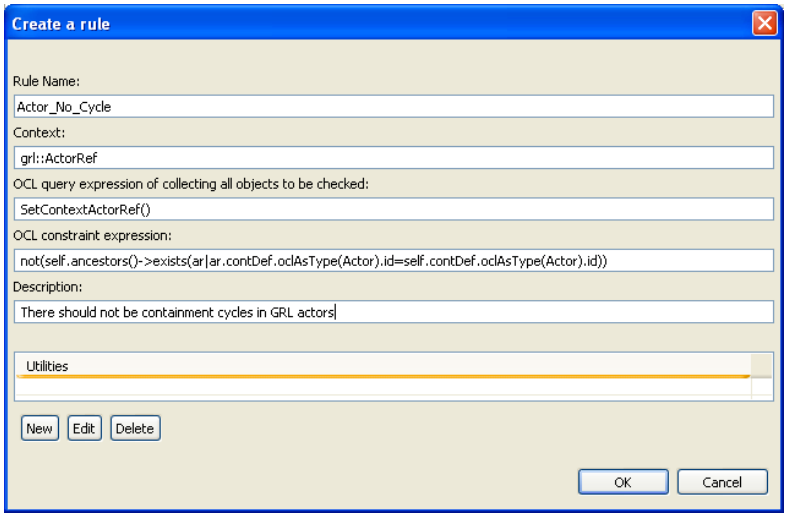

Figure 18 Creating the rule Actor_No_Cycle

Next, click the button New to create two utilities (see Figure 19, Figure 20). Finally, running the verification engine leads to the result showed in Figure 21.

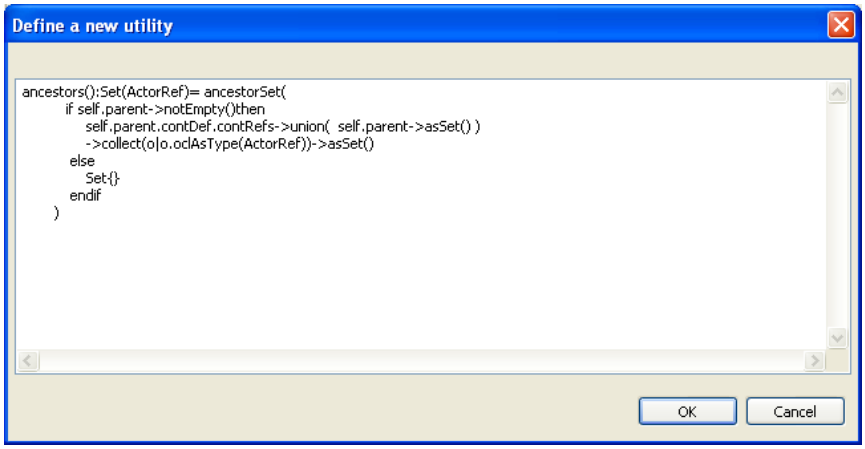

Figure 19 Defining the utility of ancestors

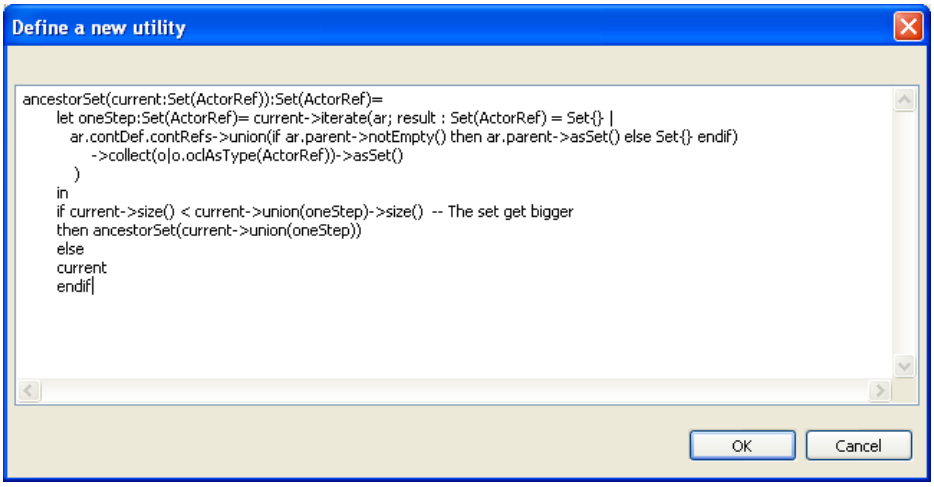

Figure 20 Defining the utility of ancestorSet

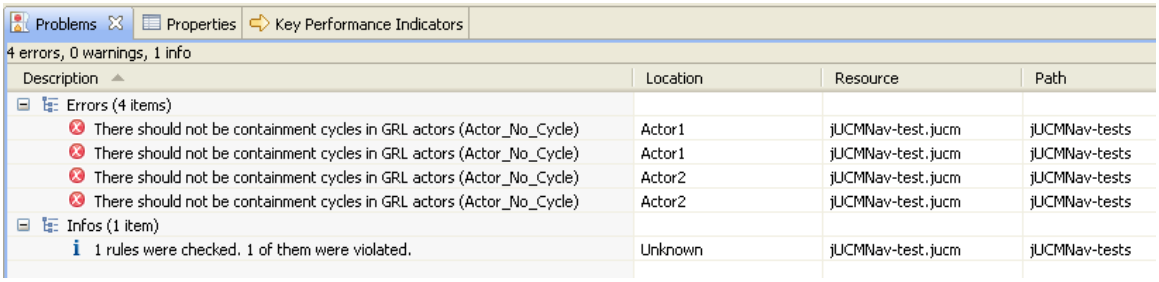

Figure 21 Checking result of Actor_No_Cycle

4.4. All GRL tasks should have a link from at least one GRL element

The rule definition is showed in Figure 22. The checking result is showed in Figure 23.

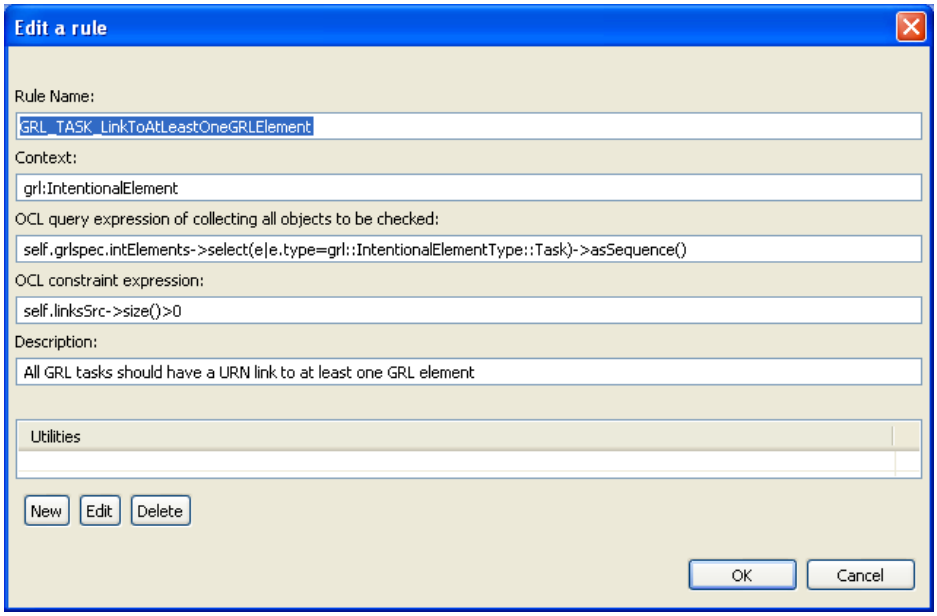

Figure 22 Rule definition of GRL_TASK_LinkToAtLeastOneGRLElement

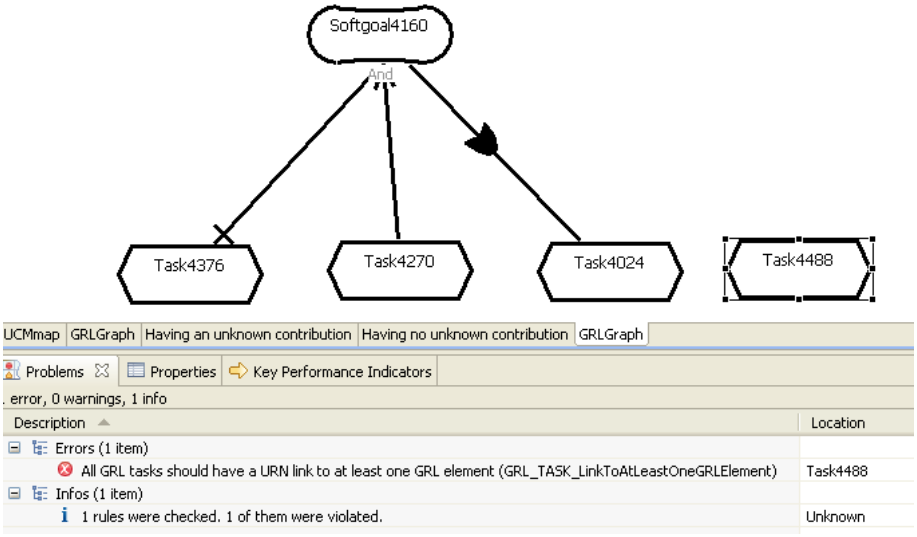

Figure 23 Checking result of GRL_TASK_LinkToAtLeastOneGRLElement

4.5. There should be no unknown contributions in GRL models

The rule definition is showed in Figure 24. The checking result is showed in Figure 25, Figure 26.

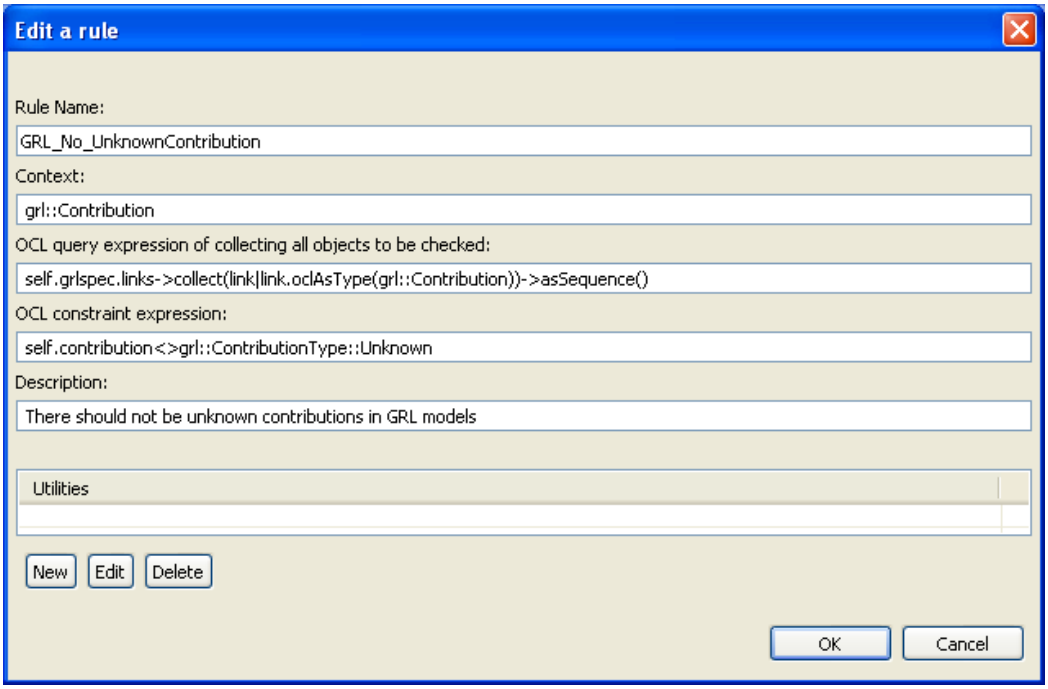

Figure 24 Rule definition of GRL_No_UnknownContribution

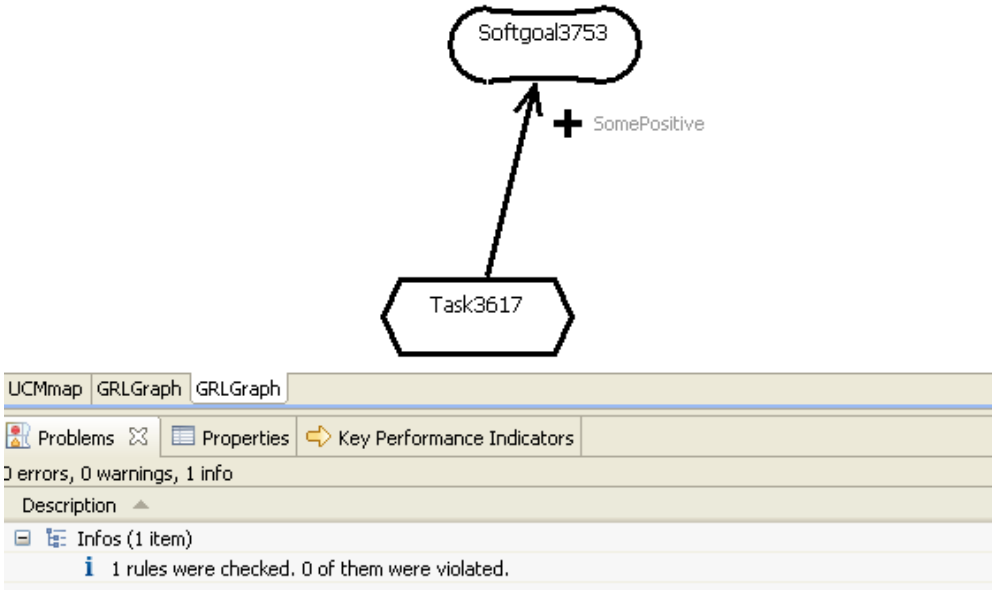

Figure 25 Result without violations

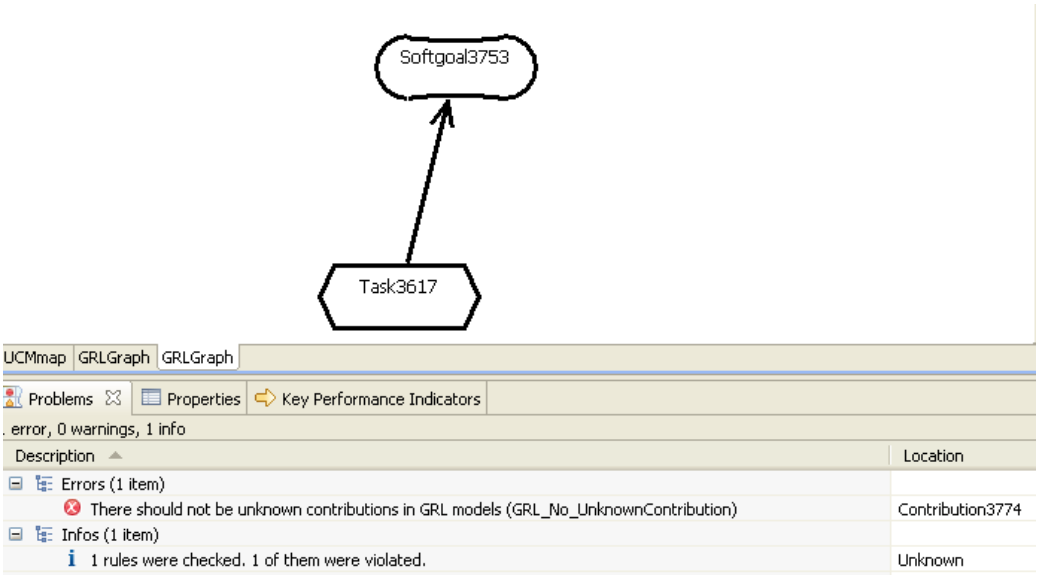

Figure 26 Result with violations

4.6. There should be no containment cycles in UCM components

This case is similar to the case in section 4.3. The following is the definition:

Rule classifier: ucm::map::ComponentRef

Rule Context Expression:

```
self.urndef.specDiagrams->
   select(d|d.oclIsTypeOf(ucm::map::UCMmap)).contRefs->
   select(r|r.oclIsTypeOf(ucm::map::ComponentRef))->
   collect(o|o.oclAsType(ucm::map::ComponentRef))->asSequence()
```

Rule Invariant Expression:

```
not(self.ancestors()->
   exists(ar|ar.contDef.oclAsType(urncore::ComponentElement).id
      =self.contDef.oclAsType(urncore::ComponentElement).id))
```

Utility One:

```
ancestorSet(current:Set(ComponentRef)):Set(ComponentRef)=
       let oneStep:Set(ComponentRef)= current->
         iterate(ar; result : Set(ComponentRef) = Set{} |
         ar.contDef.contRefs->union(
            if ar.parent->notEmpty()
               then ar.parent->asSet() else Set{}
            endif)
            ->collect(o|o.oclAsType(ComponentRef))->asSet()
          )
```

```
in
if current->size() < current->union(oneStep)->size()
   -- The set get bigger
then ancestorSet(current->union(oneStep))
else
current
endif
```

Utility Two:

```
ancestors():Set(ComponentRef)= ancestorSet(
         if self.parent->notEmpty()then
         self.parent.contDef.contRefs->
            union(  self.parent->asSet() )->
            collect(o|o.oclAsType(ComponentRef))->asSet()
          else
              Set{}
          endif
        )
```

The checking result is showed in Figure 27.

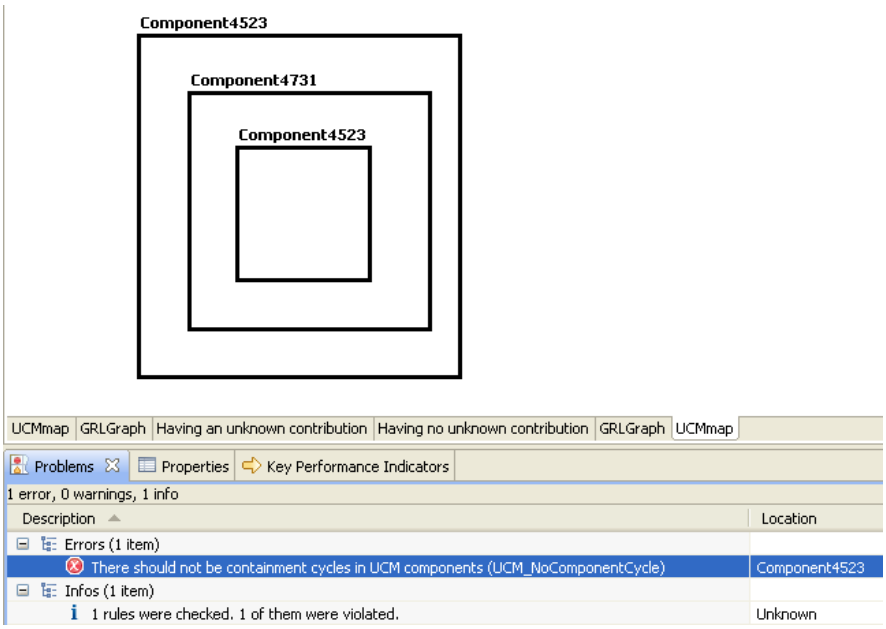

Figure 27 Checking result of UCM_NoComponentCycle

Chapter 5. Conclusions

To support different model styles in various applications of jUCMNav, a static semantics checking tool was designed and implemented. Users of jUCMNav can now regulate some special model styles described by OCL rules. In this report, we explained why the OCL language is chosen to describe the static semantic rules and we explored several OCL implementations and evaluated the MDT OCL component in details. Furthermore, we discussed some issues and corresponding solutions in designing and implementing the checking tool based on the MDT OCL component. Validation of the tool was done based on several experiments with different rules.

In the future, some potential features could be considered. For example, MDT OCL provides a feature called Content Assistant Support which parses partial OCL expressions and then supplies completion suggestions. With this feature, many helpful tips can be given in the rule definition editor. We can also provide another functionality to allow advanced users to inspect internal objects data. For example, we could show all objects that are returned by the Rule Context Expression. This would enable the support of models queries and metrics based on the same OCL engine.

References

[1] OMG – Object Management Group (2006). Object Constraint Language Specification, version 2.0, May 2006. http://www.omg.org/cgi-bin/doc?formal/2006-05-01.

[2] IBM Corporation, OCL Developer Guide: Programmer's Guide - OCL Overview, 2007

[3] The MOMENT Project, MOMENT OCL, June 11, 2007, http://moment.dsic.upv.es/content/view/32/75/

[4] Vanwormhoudt, Gilles , EMF OCL Plugin, March, 2008, http://www.enic.fr/people/Vanwormhoudt/siteEMFOCL/index.html

[5] Roclet Website, RoclET, March 2008, http://www.roclet.org/

[6] MDT OCL Wiki, MDT OCL, March 2008, http://wiki.eclipse.org/index.php/MDT-OCL

[7] Eclipse.org, MDT OCL SDK 1.1.2, 28 Nov 2007, MDT OCL developer guide, http://www.eclipse.org/modeling/mdt/downloads/index.php?project=ocl&showAll=0&showMax=5

[8] Amyot, D. (2003). Introduction to the User Requirements Notation: Learning by Example. Computer Networks, 42(3), 285-301, 21 June. http://www.usecasemaps.org/pub/ComNet03.pdf

[9] ITU-T – International Telecommunications Union (2003). Recommendation Z.150 (02/03), User Requirements Notation (URN) – Language Requirements and Framework. Geneva, Switzerland.

[10] jUCMNav 3.0, http://jucmnav.softwareengineering.ca/twiki/bin/view/ProjetSEG/WebHome

[11] Roy, J.-F. Kealey, and Amyot, D.: Towards Integrated Tool Support for the User Requirements Notation (2006). SAM 2006: Language Profiles - Fifth Workshop on System Analysis and Modelling, Kaiserslautern, Germany. LNCS 4320, pp. 198-215, Springer (2006).

Appendix A: URN Metamodel 0.19

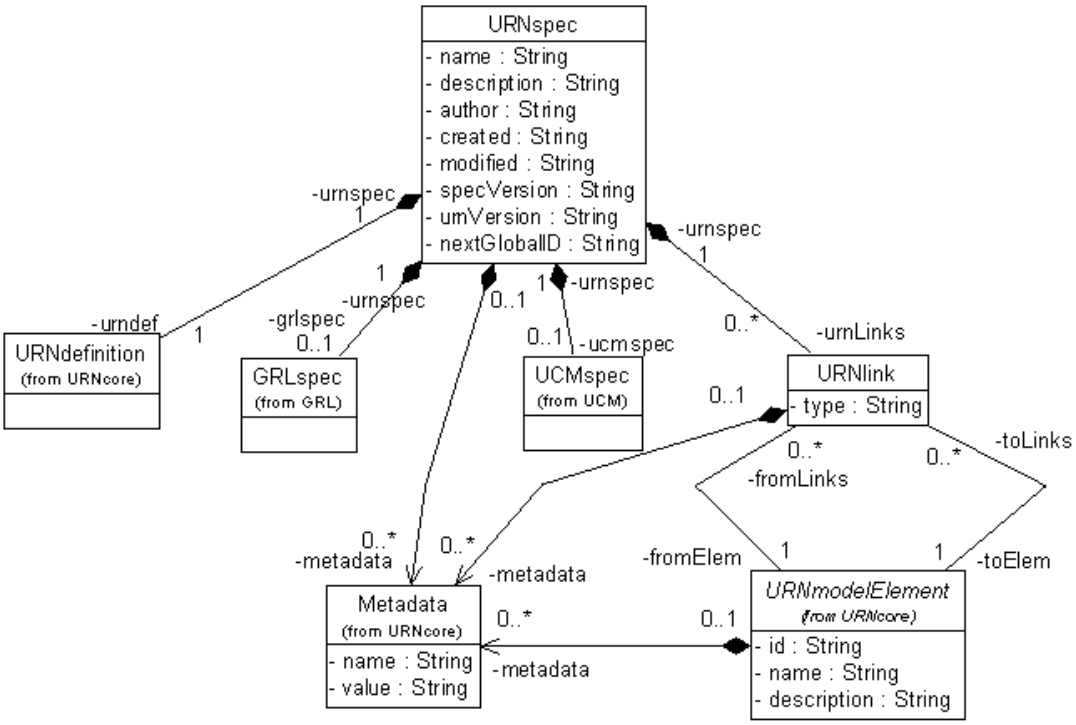

Figure 28 URN Metamodel 0.19 – Top

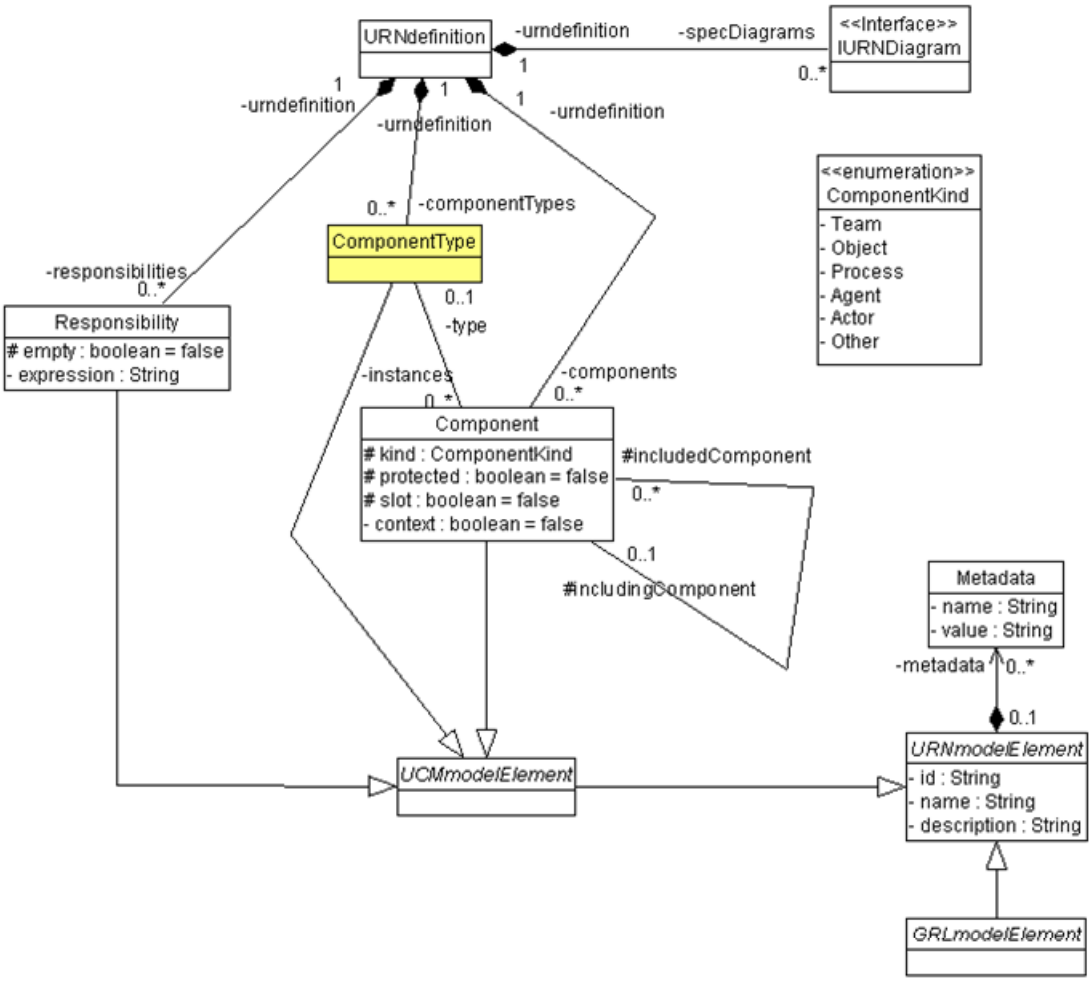

Figure 29 URN Metamodel 0.19 – URNcore

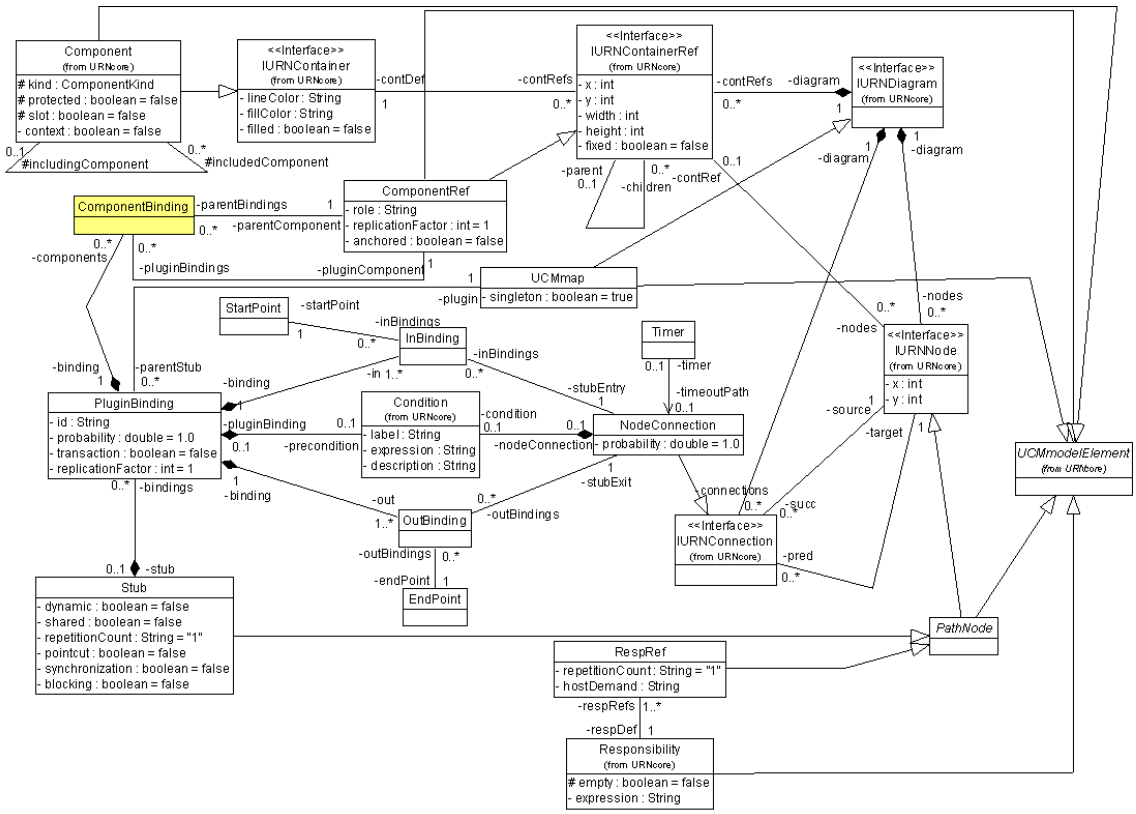

Figure 30 URN Metamodel 0.19 – UCM – Map

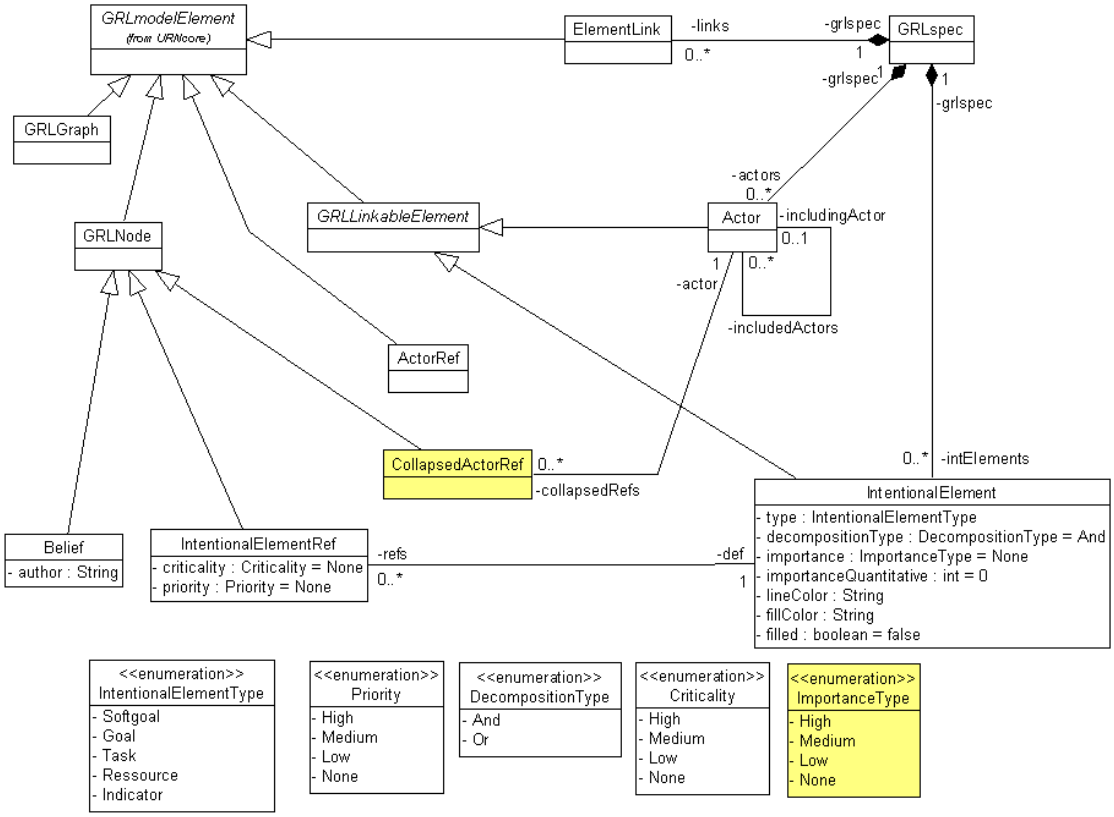

Figure 31 URN Metamodel 0.19 – GRL – Overview