Flexible Verification of User-Defined Semantic Constraints in Modelling Tools

Daniel Amyot and Jun Biao Yan

SITE, University of Ottawa, Canada

Abstract

Many modelling tools embed verification rules that are checked against user-defined models to ensure they satisfy the static semantic constraints of the modelling language. However, there are many other contexts where required constraints vary with the intended purpose of the model, and not just the modelling language used. In this paper, we propose a flexible and practical approach for users to define, select, store, group, exchange, enable, and verify custom semantic constraints on metamodels with the Object Constraint Language. We illustrate the benefits of this approach with extensions to an Eclipse-based modelling tool, called jUCMNav, and applications to various contexts such as style compliance, analysis, and transformations that involve chains of tools. We believe this approach to be easily adaptable to other Eclipse-based modelling tools, which could then enjoy similar benefits.

1 Introduction

Modelling tools are nowadays essential to many software engineering activities and business process modelling tasks. Many of these tools embed verification rules that are checked against user-defined models to ensure they satisfy the static semantic constraints of the modelling language. Constraints on language metamodels are often formalized with OMG's Object Constraint Language (OCL) [20].

OCL constraints can also be used to increase precision in user models, especially when described using the Unified Modeling Language (UML) [21]. Results from an experiment [3] indicate that, provided sufficient training, using OCL has the potential to significantly improve an engineers' ability to understand, inspect and modify a system modelled with UML.

There are however many other situations where required constraints vary with the intent of the model, and not just the modelling language used. For instance, if the model is meant to be used as input to an analysis algorithm or for a particular transformation whose output would be used by another tool, then additional constraints may be required to insure interoperability. Analysis and modelling tools often have restrictions on their expected input, and these evolve with each new release. Stylistic constraints in terms of content, structure, and even graphical representations of models could also be enforced within a project or a company. UML profiles can help tailor UML to particular domains and impose additional constraints, but not with the flexibility required to address the situations above, especially as these constraints usually vary from model to model, or even for different versions of a same model.

In this paper, we propose a flexible way for users to define, select, store, group, exchange, enable, and verify custom semantic constraints. To illustrate the benefits of such an approach, we have extended an Eclipse-based tool called jUCMNav [14][26], which supports goal and scenario modelling with the User Requirements Notation (URN) [2][12][13]. Constraints are OCL rules that are managed via preference pages in Eclipse and then selectively verified against URN models upon request. Violations are reported to the modeller in Eclipse's Problems view.

In section 2, we give an overview of semantic constraints and present requirements for their support in modelling tools. Our experiment involves the jUCMNav environment for URN, which are both introduced in section 3. In section 4, we present our extensions to jUCMNav for supporting constraints. We illustrate their benefits in section 5 with examples of applications in various contexts related to style compliance, model analysis, and model transformations. Related work is discussed in section 6, followed by our conclusions.

2 Semantic Constraints

2.1 Overview

A static semantic constraint is a rule that must be satisfied by a model. Informal examples of constraints include:

- Elements of type X should have unique names within a model.
- Element Y must contain at least one element of type Z.
- There must not be any cycle in containers (that is, a container cannot contain itself directly or indirectly).

Constraints can be provided by the developers of a modelling tool, its users, or third-parties. Many such constraints (in the hundreds) can co-exist, but not all of them are applicable to all models. The presence and utility of semantic constraints vary with the purpose of a model (informal drawing, formal documentation, input to analysis features, input to transformations, etc.) but also with time, as the same model can start as an informal representation that evolves to be used in a more formal way, e.g., for transformations.

2.2 Tool Requirements

In order for semantic constraint verification to be flexible (i.e., adaptable to multiple situations) while remaining usable, we have elicited minimal requirements for tool support:

R1. The tool shall allow users to create, modify, and delete constraints, without having to recompile or restart the tool.
R2. The tool shall report constraints syntax errors.
R3. The tool shall allow users to enable and disable individual constraints.
R4. The tool shall verify, upon the user's request, all enabled constraints on the model.
R5. The tool shall report constraint violations to the user.
R6. The tool shall allow users to save constraints to a file and load constraints from a file (hence enabling sharing and updates).
R7. The tool shall allow users to create, modify, and delete groups of constraints.
R8. The tool shall allow users to add/remove a constraint to/from one or many groups.
R9. The tool shall allow users to enable and disable groups of constraints.

These requirements have driven the development of extensions to an existing modelling tool (jUCMNav) in order to enable the flexible verification of user-defined static constraints on URN models.

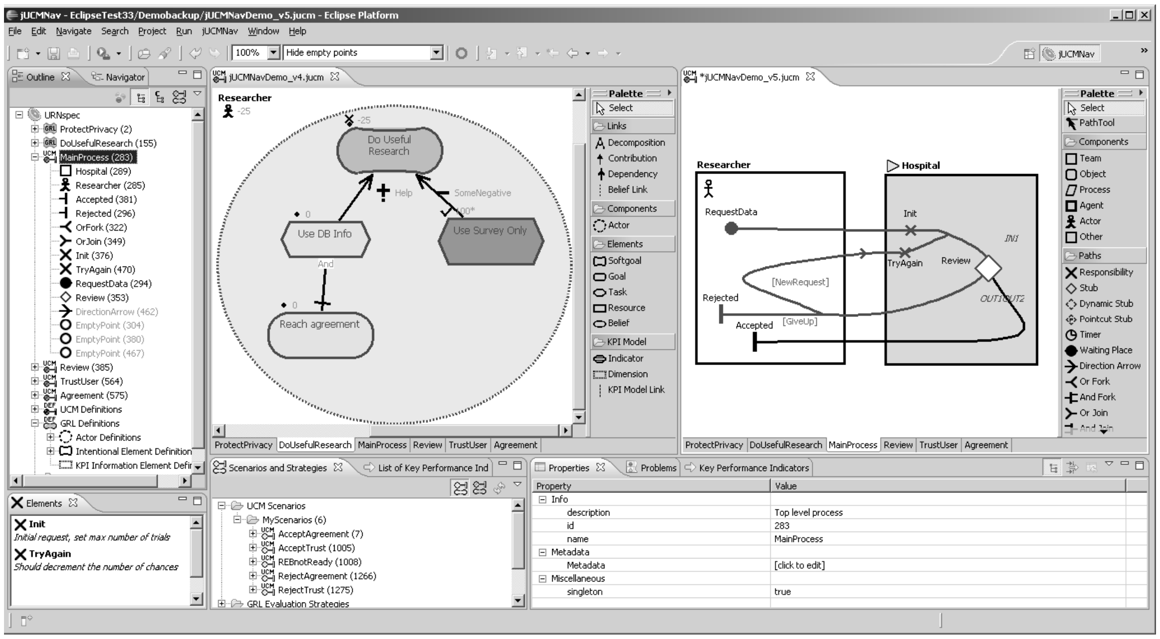
3 URN and jUCMNav

The User Requirements Notation is a modelling language for specifying and analyzing requirements described with goals and scenarios. URN is being standardized by the International Telecommunications Union (ITU-T) [12]. An overview of the notation is given in [2].

A URN model can contain multiple diagrams capturing two complementary views expressed with the Goal-oriented Requirements Language (GRL) for goals, rationales, and qualities, and with Use Case Maps (UCM) for scenarios and architectural alternatives.

URN is defined using a metamodel, described in draft Recommendation Z.151 [13]. The complete description of the language is outside the scope of this paper, but several parts will be introduced later to support examples of semantic constraints.

jUCMNav is an open-source, Eclipse-based tool for URN modelling and transformations. It supports analysis features for executing UCM models (scenario definitions and traversal algorithms) and evaluating GRL models (strategies and propagation algorithms) [26]. This tool implements the URN metamodel using the Eclipse Modeling Framework (EMF) [4]. Figure 1 shows the jUCMNav editor, with goal and scenario views of a URN model.



Property	Value
Info	
description	Top level process
id	283
name	MainProcess
Metadata	
Metadata	[click to edit]
Miscellaneous	
singleton	true

Figure 1: Goal and Scenario Modelling with URN in jUCMNav

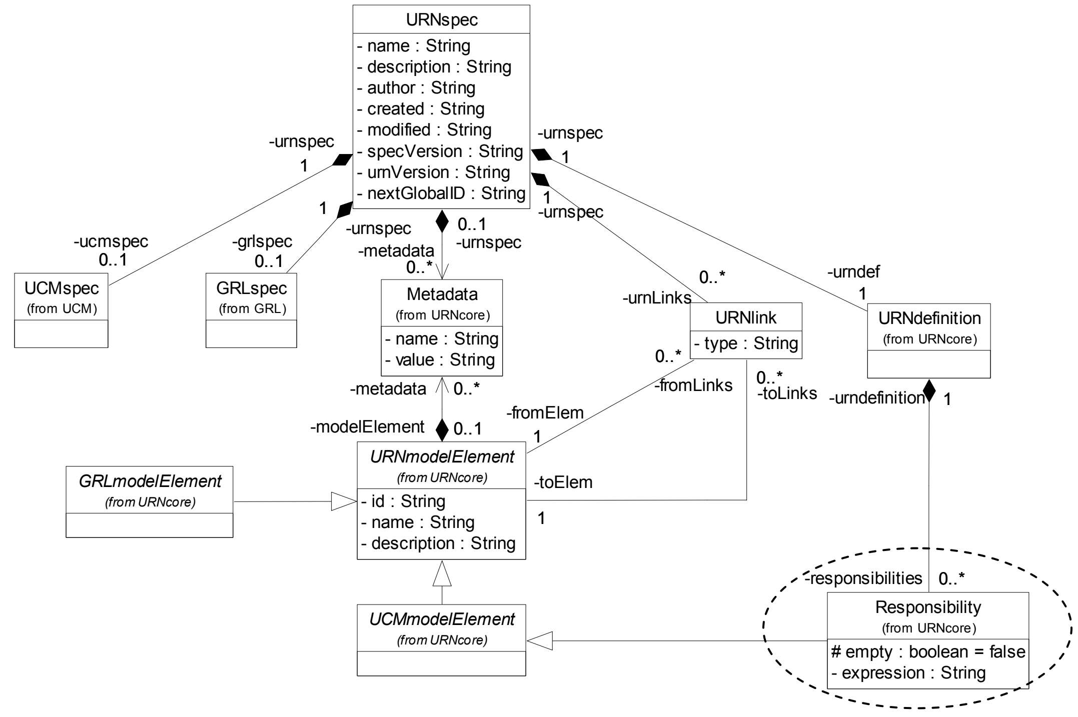


Figure 2: Extract of jUCMNav's Metamodel, with URNspec as the Top Element

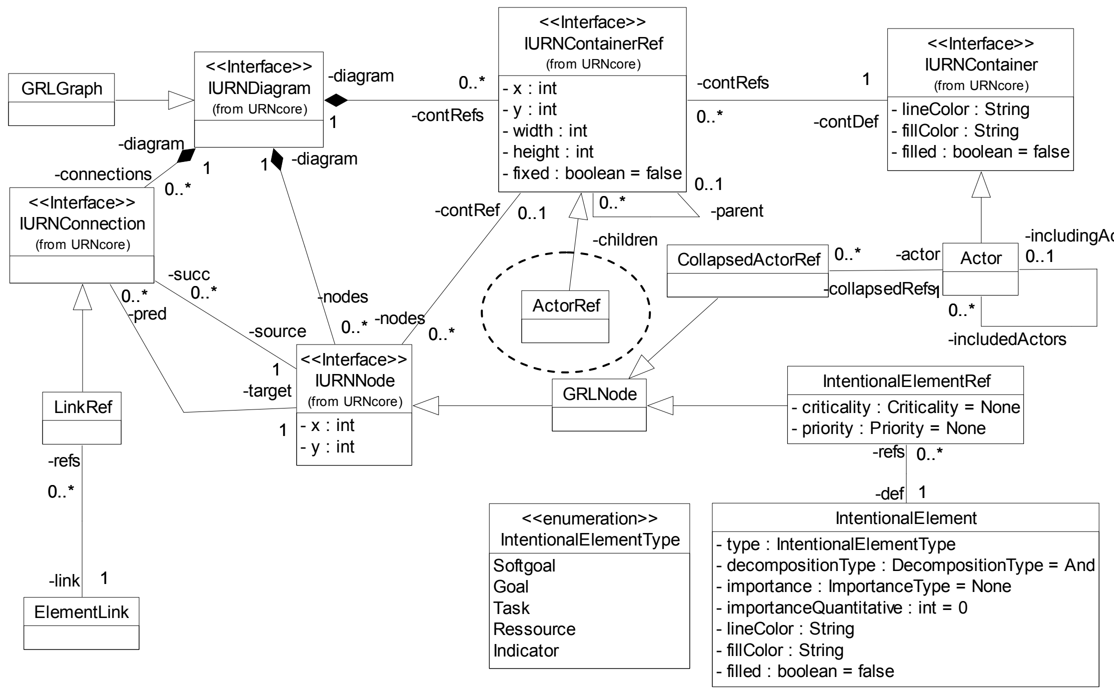


Figure 3: Extract of jUCMNav's Metamodel, Focusing on Part of GRL

Figure 2 shows part of the URN metamodel as implemented in jUCMNav. A URN model or specification may contain a GRL specification, a UCM specification, and URN elements that can be linked and have metadata. One type of UCM elements is called *responsibility*. Figure 3 shows another extract with an emphasis on how jUCMNav handles GR diagrams. Note that *actors* may contain sub-actors. These two concepts will be used later in our examples. The complete metamodel contains about 100 classes and is fully described in [14].

Although jUCMNav 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 modelling styles required by export filters (transformations) to other notations such as Message Sequence Charts (MSC) [11] for design or the Core Scenario Model (CSM) for software performance engineering [22]. The tool also deviates from the proposed Z.151 standard on several occasions and provides extensions for exploratory concepts such as Key Performance Indicators [24] and Aspect-oriented URN [19].

4 Support for Constraints

This section presents our extensions to jUCMNav enabling users to create/delete, group, load/save, enable/disable, and verify customized constraints upon demand, without restarting the tool, as required in section 2.2 (see R1 to R9). Each of these functionalities is described, but first we present the underlying OCL technology used here to support our approach as well as our data structure for constraints.

4.1 OCL Constraints and EMF

Most modelling tools nowadays use metamodels in the background, which are often based on EMF in the Eclipse community. jUCMNav is no exception. In our context, selecting OCL rather than another constraint language was a simple choice. OCL is a standardized and precise language meant to describe constraints on models and metamodels. The evaluation of OCL constraints has no side effect, and the language offers navigation features enabling constraint writers to refer to model elements easily. OCL is also well supported in general, and particularly in Eclipse.

There exist several OCL implementations developed such as MOMENT-OCL [18], RoclET [25], and the MDT OCL from Eclipse's Model Development Tools project [7]. However, MDT OCL is the only one that provides an application programming interface (API), enabling better integration with existing modelling tools.

The MDT OCL plug-in is partitioned into the following packages:

- `org.eclipse.ocl`: the core parsing, evaluation, and content assist services.
- `org.eclipse.ocl.ecore`: provides support for working with OCL constraints and queries targeting Ecore models. Ecore is Eclipse's equivalent of UML's core Meta Object Facility (MOF) and is used to create metamodels.
- `org.eclipse.ocl.uml`: provides support for working with OCL constraints and queries targeting UML models.

In our plug-in, we only use the first two packages because the jUCMNav is based on an Ecore model, not on UML.

4.2 Constraint Structure

A constraint definition, also called *rule* in our tool, contains the following attributes:

- The *rule name* identifies the constraints.
- The *rule context* is a package name followed by the class name from the metamodel, e.g., `Package::ClassName`.
- The *OCL query expression* is an expression under the context of `URNspec` that returns a sequence of objects of the type specified in the rule context.
- The *OCL constraint expression* is the invariant to be verified on the objects collected by the OCL query expression.
- The *rule description* gives the human readable semantics of the rule. This is also used as an error message in the Problems view when the rule is violated.
- The *rule enabling indicator* is a Boolean value that is true when the rule is enabled.
- The *rule utility definitions* are optional, user-defined operations used to simplify the logic of OCL invariant expressions. They are similar to the `def` part of an OCL file. Utilities are defined under the rule context.

To illustrate this structure, we will use a simple example applied to the class Responsibility in the URNcore package, as highlighted in Figure 2. Suppose that, as a stylistic constraint, we require all responsibilities to have non-empty descriptions. In OCL, this could be expressed as follows:

```
package URNcore
context Responsibility
inv non_empty_desc:
    self.description.size() > 0
endpackage
```

With our constraint structure, we would have:

- Name: `non_empty_desc`
- Context: `URNcore::Responsibility`
- Constraint expression: `self.description.size() > 0`
- Description: All UCM responsibility definitions should have a non-empty description.
- Utility definitions: none.

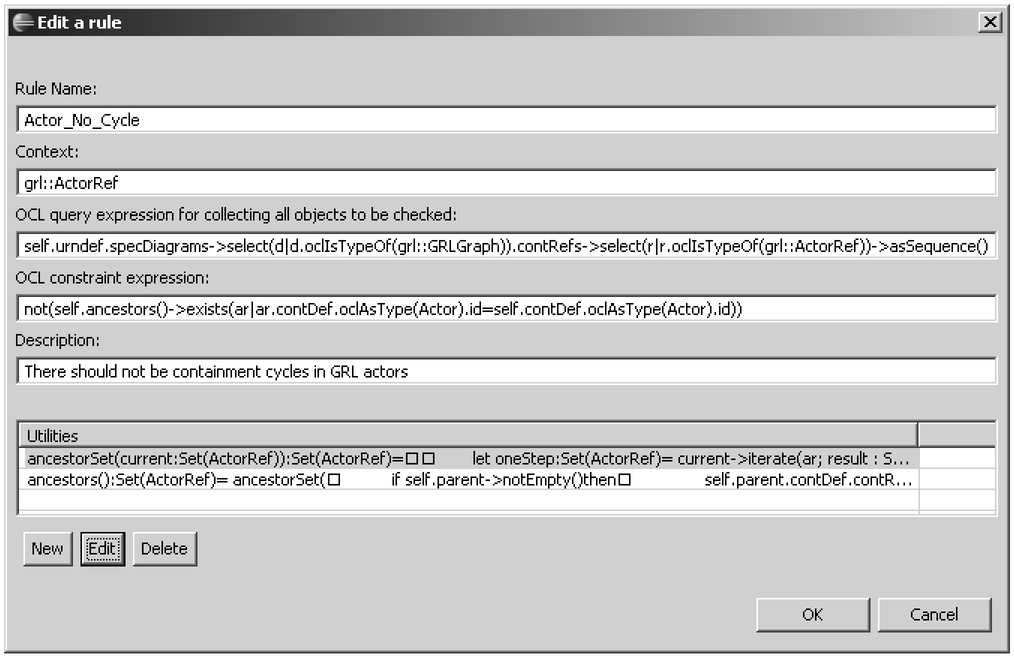
Since we are dealing with Java applications and Eclipse's MDT, the OCL `allInstance()` operator cannot be used in the expressions for queries and invariants. In fact, we cannot even get all the instances of the Responsibility class unless we provide an explicit way of collecting them at runtime. This is the purpose of the OCL query expression, which must be provided for all constraints. In jUCMNav's metamodel, all classes can be reached from the URNspec singleton through direct or indirect aggregation relationships (Figure 2). Hence, in our example, the query expression that produces the collection of all Responsibility objects is:

```
self.urndef.responsibilities
```

Such query expressions can become more complicated if one needs to collect objects from multiple locations. For example, in Figure 3, instances of the ActorRef class (actor references) are contained in different diagrams (GRLGraph class) of a model, hence they must be collected explicitly.

4.3 Creation and Deletion

Rules and groups of rules are managed through the jUCMNav preference page (Figure 8). This enables the sharing of constraints across models as well as their verification without restarting or recompiling the tool, as required by R1.



Rule Name:

Actor_No_Cycle

Context:

grl::ActorRef

OCL query expression for collecting all objects to be checked:

```
self.urndef.specDiagrams->select(d|d.oclIsTypeOf(grl::GRLGraph)).contRefs->select(r|r.oclIsTypeOf(grl::ActorRef))->asSequence()
```

OCL constraint expression:

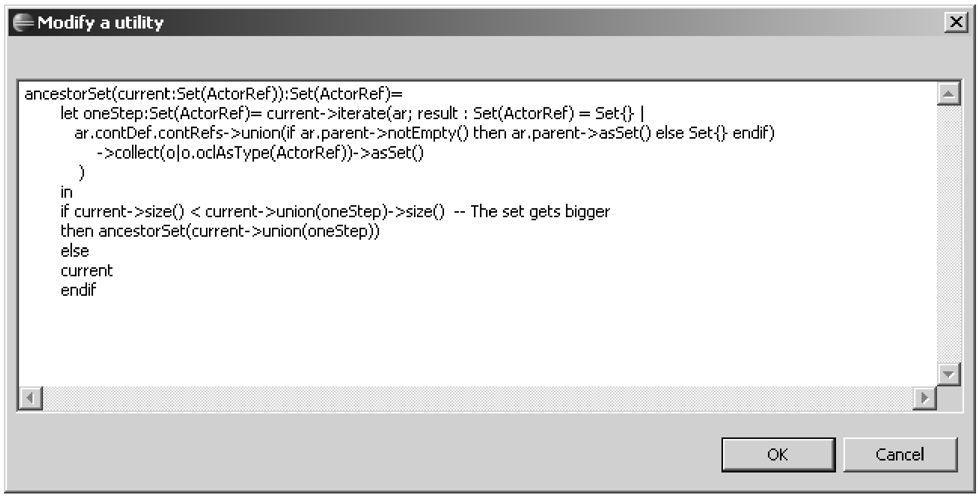
```
not(self.ancestors()->exists(ar|ar.contDef.oclAsType(Actor).id=self.contDef.oclAsType(Actor).id))
```

Description:

There should not be containment cycles in GRL actors

Utilities
ancestorSet(current:Set(ActorRef)):Set(ActorRef)=□□ let oneStep:Set(ActorRef)= current->iterate(ar; result : S...
ancestors():Set(ActorRef)= ancestorSet(□ if self.parent->notEmpty()then□ self.parent.contDef.contR...

Figure 4: Dialog for Constraint Definition



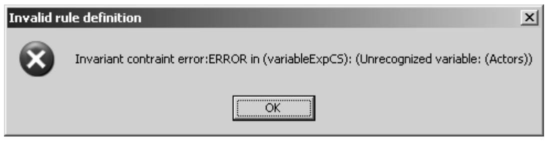
```
ancestorSet(current:Set(ActorRef)):Set(ActorRef)=
      let oneStep:Set(ActorRef)= current->iterate(ar; result : Set(ActorRef) = Set{} |
        ar.contDef.contRefs->union(if ar.parent->notEmpty() then ar.parent->asSet() else Set{} endif)
            ->collect(o|o.oclAsType(ActorRef))->asSet()
 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

Figure 5: Dialog for Utility Definition

Adding or editing a rule invokes a dialog box, shown in Figure 4, where the content of the constraint structure can be provided. The rule `Actor_No_Cycle` in this example checks that the model does not contain inconsistencies in the containment of GRL actor references. For example, it would detect that actor A is contained in actor B in one diagram and that actor B is contained in actor A in another diagram of the same model. As explained in the previous section, the corresponding OCL query expression is not trivial because it has to collect all instances of actor references (a subclass of IURNContainerRef, see Figure 3) in all instances of GRL graphs (a subclass of IURNDiagram).

This rule also makes use of two utility definitions because, for this rule to work properly, we need to compute the set of ancestors of each actor reference. This is done in OCL with additional recursive functions invoked by the constraint expression. In Figure 4, buttons are provided to add, edit, or remove utility definitions. One of them is shown in Figure 5, which illustrates the other dialog used to edit OCL utility definitions.

This is one of the most complex rules we have defined in our validation experiments. In terms of complexity, most rules are in fact more similar to the `non_empty_desc` constraint described in the previous section. Nevertheless, we realized quickly that it is easy to make mistakes in OCL expressions, simple or complex. In order to mitigate this problem, our tool (through the MDT OCL plug-in functionalities) parses modifications to OCL expressions and reports any error before committing changes to the preferences repository, hence satisfying R2. For example, if we write Actors instead of Actor in the constraint expression, the error dialog shown in Figure 6 is displayed.



```
Invariant contraint error:ERROR in (variableExpCS): (Unrecognized variable: (Actors))
```

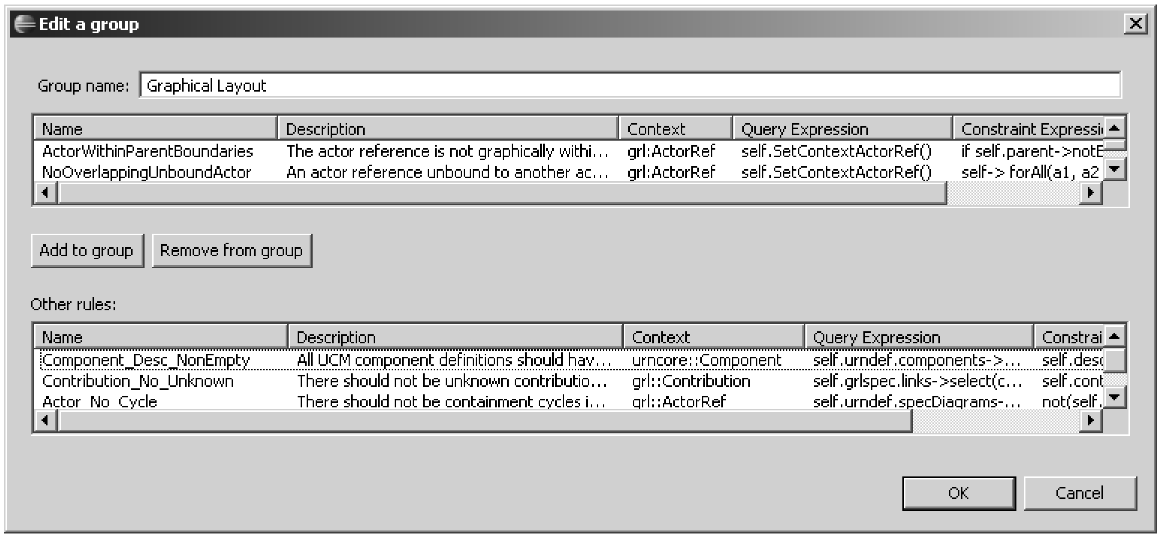
Figure 6: Example OCL Parsing Error

To further help users define constraints more easily while avoiding duplication, several common utilities are predefined in a file (*library.ocl*) and can be used by any of the rules. They currently target the navigation of the somewhat complex URN metamodel. For instance, the OCL query expression for the `Actor_No_Cycle` rule can be simplified to `self.SetContextActorRef()`. This standard library can be edited by users to add further common utility definitions. This is currently done manually but it would be simple to add GUI support for this via the preferences page.

4.4 Grouping and Selection

In order to address requirements R7 and R8, our tool's preference page was extended with facilities to create, edit, and delete named groups of rules. As shown in Figure 7, users can add existing rules to a group, or remove rules from a group. As rules are declared globally, the same rule can be part of multiple groups.

The new preference page used to manage rules and groups is illustrated in Figure 8. Checkboxes are used to set the value of the *enabling indicator* (discussed in section 4.2) for each rule, hence addressing R3. In addition, the rules that are part of a group can be enabled/disabled all at once, satisfying R9, simply by enabling/disabling this group. Note that multiple rules and groups can also be selected and deleted all at once.



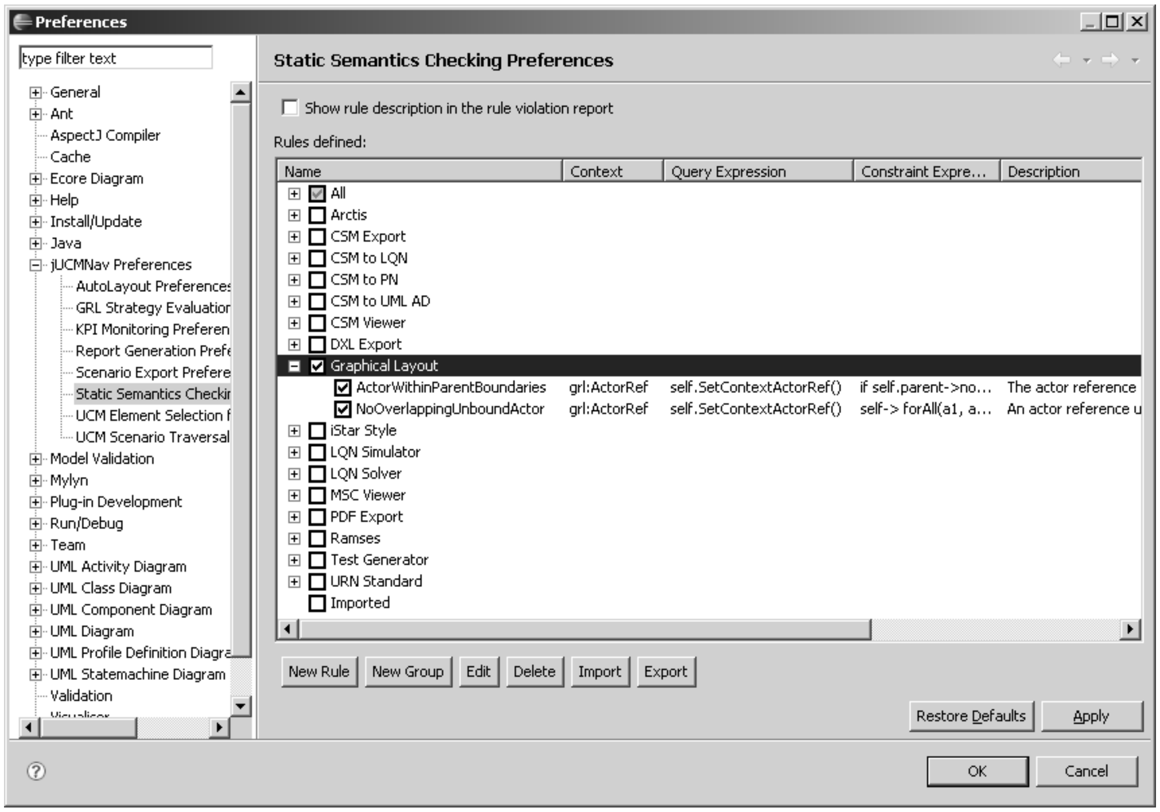
Group name: Graphical Layout

Name	Description	Context	Query Expression	Constraint Expressi
ActorWithinParentBoundaries	The actor reference is not graphically withi...	grl:ActorRef	self.SetContextActorRef()	if self.parent->notE
NoOverlappingUnboundActor	An actor reference unbound to another ac...	grl:ActorRef	self.SetContextActorRef()	self-> forAll(a1, a2

Other rules:

Name	Description	Context	Query Expression	Constrai
Component_Desc_NonEmpty	All UCM component definitions should hav...	urncore::Component	self.urndef.components->...	self.desc
Contribution_No_Unknown	There should not be unknown contributio...	grl::Contribution	self.grlspec.links->select(c...	self.cont
Actor_No_Cycle	There should not be containment cycles i...	grl::ActorRef	self.urndef.specDiagrams-...	not(self.

Figure 7: Dialog for Adding/Removing Rules in a Group



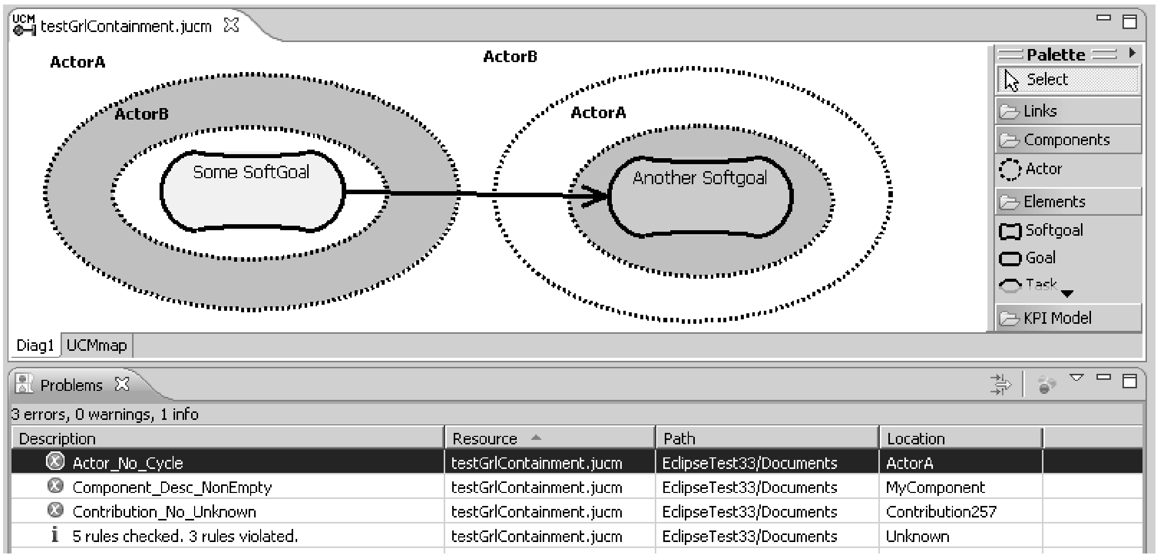
Static Semantics Checking Preferences

☐ Show rule description in the rule violation report

Rules defined:

Name	Context	Query Expression	Constraint Expre...	Description
☑ All				
☐ Arctis				
☐ CSM Export				
☐ CSM to LQN				
☐ CSM to PN				
☐ CSM to UML AD				
☐ CSM Viewer				
☐ DXL Export				
☑ Graphical Layout				
☑ ActorWithinParentBoundaries	grl:ActorRef	self.SetContextActorRef()	if self.parent->no...	The actor reference
☑ NoOverlappingUnboundActor	grl:ActorRef	self.SetContextActorRef()	self-> forAll(a1, a...	An actor reference u
☐ iStar Style				
☐ LQN Simulator				
☐ LQN Solver				
☐ MSC Viewer				
☐ PDF Export				
☐ Ramses				
☐ Test Generator				
☐ URN Standard				
☐ Imported				

Figure 8: Preferences and Rule/Group Enabling



3 errors, 0 warnings, 1 info

Description	Resource	Path	Location
Actor_No_Cycle	testGrlContainment.jucm	EclipseTest33/Documents	ActorA
Component_Desc_NonEmpty	testGrlContainment.jucm	EclipseTest33/Documents	MyComponent
Contribution_No_Unknown	testGrlContainment.jucm	EclipseTest33/Documents	Contribution257
5 rules checked. 3 rules violated.	testGrlContainment.jucm	EclipseTest33/Documents	Unknown

Figure 9: Example of Rule Violations Reported in Eclipse's Problems View

There are two special groups of rules. The first one, named All, includes all the rules in the system and cannot be deleted. It allows users to enable or disable all rules at once. The second special group, named Imported, is used to store rules imported from an external file. This will be discussed further in section 4.6.

4.5 Verification

jUCMNav already adds a menu to the Eclipse environment. A new item was added to this menu to allow users to verify all the rules enabled in the preferences against the current URN model, as per requirement R4.

Verification results are reported to the user in Eclipse's Problems view, satisfying requirement R5. As shown in Figure 9, the information contains the number of rules checked and the number of violations. For each violation, the rule name (or the rule description, if the checkbox at the top of the preference page is selected, see Figure 8), the file name of the model, and the location of the violation (i.e., a model element involved) are displayed. Double-clicking on the violation brings the user to the violating element in the editor.

Although we have not done any performance evaluation per se, so far verifying a large number of rules on a large URN model is very quick thanks to the efficiency of the underlying MDT OCL engine. It is rapid enough that live verification of the model (at every change, with a verification mechanism) could be considered instead of the on-demand approach implemented here. Unsurprisingly however, the duration of the verification depends on the complexity of the OCL rules (e.g., when recursion is used or when large collections are involved).

4.6 Saving and Loading

Not every user of jUCMNav knows OCL or the URN metamodel well. Therefore, there is a need to be able to import rules created by OCL experts. Furthermore, a user may want to save some of her rules to migrate them to another Eclipse environment or to share them with the community at large. Third-parties providing transformations based on jUCMNav's outputs can also provide rules specifying the expectations of their tools, to improve interoperability and to minimize bad surprises.

Our tool allows users to save and load rules, as required by requirement R6. Saving to a file is done first by selecting one or many rules on the preference page and then pressing the export button (Figure 8). The import button allows users to load rules from a file, which are then put in the Imported group. Imported rules are renamed (using a suffix) if their names clash with the names of existing rules. The file format is XML, and a simple schema is used to ensure the validity of the files. This schema covers the structure discussed in section 4.2, except the enabling indicator.

5 Application Examples

The previous section described the tool and demonstrated how it meets the general requirements discussed in section 2.2. This section focuses on applications of this tool to various domains. Many of these domains are still speculative and require additional empirical evidence to support their benefits, but listing them at this point is sufficient to demonstrate the flexibility of the approach and to show where it adds value to software development tool chains. Several of the rules discussed here have already been explored in [27].

5.1 Language/Style Compliance

Many modelling tools deviate from standards, either because they support only a subset of the modelling language or because they extend it to provide additional functionalities and features. Furthermore, modelling languages also evolve over time. URN and jUCMNav are no exception. There is a need for flexibility and evolveability in the support of semantic constraint verification, and our tool helps addressing these issues. For instance:

- Even though jUCMNav is a mature requirements engineering tool, it is also used as a research laboratory for the evolution of the URN language. As such, it explores new additions to the language such as Key Performance Indicators (KPIs) for business process monitoring [24] or aspect-oriented extensions to UCM and GRL views [19]. To ensure that some models are compliant with the draft URN standard [13] we can define rules that verify the absence of KPIs and aspect-oriented elements in the model. These rules can also be revisited as the URN standard evolves.

- URN itself borrows concepts from other languages for which there are existing communities of users. For instance, the GRL view of URN is based in part on the i* framework, for which many modelling guidelines have been specified [1]. In many ways, GRL is more permissive than i*, for instance on the types of intentional elements that can be linked together, or in terms of the structure of dependency links. Yet, jUCMNav is powerful and usable enough to attract users from the i* community. Therefore, to use GRL in an i*-style, additional constraints similar to those mentioned in [1] can be specified and enforced through our tool.
- Compliance to styles could even include constraints on the graphical representation of the models. For instance, to avoid confusion, GRL actors (and similarly for UCM components) may not be allowed to partially overlap. That is, an actor should be entirely inside another (parent) actor, or entirely outside of it. The use of colours in models could also be restricted, e.g. because the model is meant for a publication that publishes in black and white only or because some colours are reserved to convey some analysis results (like the green, yellow and red used for GRL elements in jUCMNav's strategy analysis [26]). OCL can be used to capture these constraints, and our tool used to manage them.

5.2 Analysis

Many modelling tools provide analysis features that embed some verification mechanisms to ensure the model under study conforms to assumptions used by the analysis algorithm. In many cases, it might be cheaper and more efficient to do such verification externally, via semantic constraints, rather than in an analysis feature. Also, pragmatic analysis rules can be described by expert users who may not want to delve into the source code of the tool, assuming it is available at all. For instance, in jUCMNav:

- Although this is allowed by URN (to explore behavioural and structural alternatives), some users want to make sure there are no inconsistent uses of UCM components across diagrams. For instance, in a way similar to the cyclic containment relationships between actors discussed in section 4.3, it might be desirable to ensure that a component does not contain itself directly or indirectly.
- GRL actor definitions that are not referenced in any diagram or that do not contain any intentional element may not be a cause for concern in the early life of a model, but they might become one when the model is considered mature. Again, this can easily be spotted with a user-selectable semantic constraint.
- A UCM stub is a container for sub-diagrams that enables hierarchical decomposition and dynamic behaviour selection. However, if a stub does not contain any sub-diagram, then this might cause some analysis feature (like the UCM scenario highlight in jUCMNav) to fail. At the moment, this would not be detected until this particular stub is visited, under a specific context (like a bug can only be found if a test case exercises it). A semantic constraint would flag these empty stubs before the invocation of the scenario highlight analysis feature.
- Some completeness aspects of a model could be analysed directly with OCL semantic constraints, without the need for other analysis algorithms. For instance, jUCMNav allows modellers to create traceability links between GRL intentional elements (e.g., goals and tasks) and UCM scenarios. One could easily define a rule to verify that all tasks are linked to some UCM element, and another rule to check that specific types of UCM elements are also linked to GRL elements. In this way, there would not be unaddressed goals or superfluous scenarios/features in the model.

5.3 Transformations

Modelling tools are often rich in transformation capabilities. Here again, semantic constraints can act as large-scale preconditions on a model to ensure that it will be of the format expected by the transformation algorithm. What is even more interesting in this context is when we start combining transformations in sequence, as constraints usually accumulate. Transformations can also be implemented outside the modelling tool. Then, this external tool has to be used to assess the quality of the input model, outside the editing environment, which is very inconvenient.

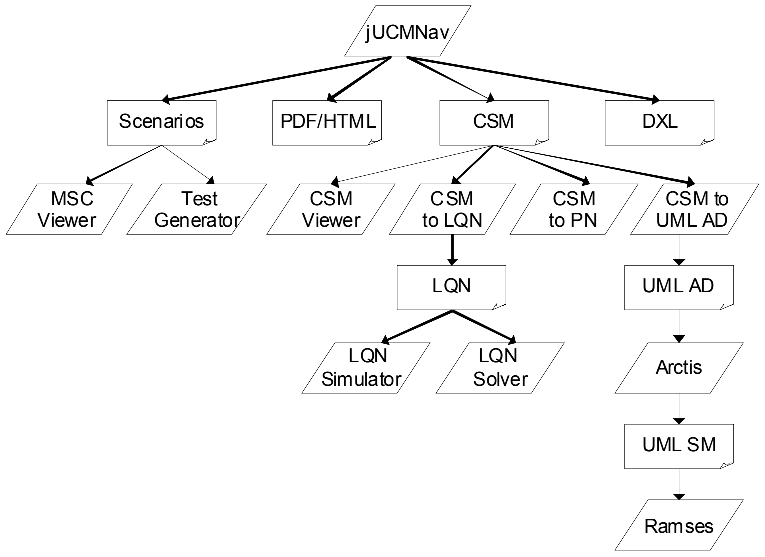


Figure 10: Examples of Sequences of Transformations from jUCMNav

jUCMNav is a good example of modelling tool that supports many export mechanisms whose outputs are themselves fed to other transformation tools. In this context, having rules that can be imported, grouped, and then enabled and disabled easily brings much value to the modeller.

Figure 10 shows some of the transformations supported by jUCMNav, together with some further typical or envisioned usages by other tools. Output file formats are represented by the document icon whereas the parallelograms represent tools.

- In a URN model, the UCM part can be "executed" (traversed) in the context of a scenario definition. Resulting execution traces can be output to an XML file that can be further converted. One application is a Message Sequence Chart (MSC [11]) viewer, which renders the execution trace in the form of an MSC, similar to a UML 2.0 sequence diagram. Messages are synthesized by the transformation engine along the way. Scenario files can also be used to generate test cases or test skeletons (FitNesse and TTCN-3 were explored in a previous version of the tool). In both cases, typical constraints would involve the proper naming of elements (e.g., by making sure UCM start/end points that represent external events are not the default names provided by jUCMNav), that guarding conditions labels are formed in a certain way (instead of free text), etc. A previously discussed rule that checks the absence of containment cycles in UCM components can be used here as well.
- Report generation to PDF or HTML formats is also a transformation. Ensuring that key model elements (e.g. UCM responsibilities and components, and GRL actors and intentional elements) have non-empty descriptions is important in that context. Moreover, existing rules targeting the naming of elements (default names must have been changed by the modeller) can be reused here.
- jUCMNav integrates with the Telelogic DOORS requirements management system via the generation of a Doors eXtensible Language (DXL) script [9]. URN models can then be imported into DOORS and linked to other types of requirements. Rules to ensure that proper traceability links exist between UCM and GRL elements are helpful in this situation. However, what is even more interesting here is that DOORS generates reports on requirements that have changed which can impact the URN model. It would be possible to have DOORS export semantic constraints (to be imported in jUCMNav) that would flag the modelling elements deserving a re-evaluation (for modification or deletion) in view of the changes in other parts of the re-

quirements. The rule import facility hence enables many such opportunities.

- The transformation from UCM to the Core Scenario Model (CSM) is particularly interesting. CSM is an intermediate format accepted by many other tools [22]. There are many assumptions about the URN model for this transformation to work [28], and most can be encoded as OCL semantic constraints. An interesting example relates to elements in CSM that do not exist in UCM, and for which URN metadata (Figure 2) can be used. Among others, resource allocation and resource release can be expressed explicitly as metadata information attached to UCM responsibilities, and this information will be considered during the transformation. As jUCMNav does not know in advance how metadata is intended to be used, appropriate rules can take over and verify their content in a particular transformation context. This is the case here.
- As we start to interoperate with other tools, the role of semantic constraints becomes more important. CSM files can be read by an Eclipse-based viewer, but also by other converters targeting Layered Queueing Networks (LQN) for performance analysis [23], Petri Nets (PN) [17], and UML activity diagrams (AD) [21]. Obviously, each of these transformations impose different constraints on the input CSM file, and therefore also on the input URN model. Nevertheless, none of these constraints can be easily implemented in the URN to CSM transformation engine as they all evolve and are often conflicting. Leaving the rules decoupled from all these tools facilitates tool development and maintenance and helps modellers to deal more easily with single-purpose and multi-purpose (URN) models.
- The same ideas can be extended to more than two consecutive transformations. In the LQN world [8], LQN models can be input to a solver or to a simulator. These two tools have different assumptions about the format of the LQN model (generated from CSM, generated from URN). We have observed the same issue in a new project where we intend to connect four transformations in sequence, from URN to CSM and then to UML activity diagrams, UML state machines (SM), and then Java code. The Arctis tool focuses on abstract, reusable service specifications that are composed from UML 2 collaborations and activities. It supports the analysis of service specifications by model checking via a temporal logic. A consistent specification can be transformed into UML state machines and components. For their implementation, Ramses contributes code generators to create executable systems [15]. With our approach, the URN modeller can determine from the editor whether her models will successfully pass through a long sequence of transformations, leading to tremendous gains in time, to models of higher quality, and to much less frustration.

6 Related Work

The idea of enabling and disabling user-defined rules is not new and has been explored in the past, both for programming languages and for modelling environments. We report here on related work for modelling.

6.1 GME and GEMS

The *Generic Modeling Environment (GME)* is both a modeling tool and a metamodelling tool used to create domain-specific modelling environments [16]. OCL constraints can be used at the model and metamodel levels. GME offers a sophisticated constraint manager where the user may disable some constraints predefined at the paradigm level. To some extent, user constraints can be added to a model or removed from it, but they cannot be imported/exported or centralized as in Eclipse preferences, and their context definition is limited. In addition, GME is a fairly standalone tool that does not integrate with Eclipse.

The recent *Generic Eclipse Modeling System (GEMS)* project attempts to bridge the gap between GME and the Eclipse technologies [6], but this work is still in progress. GEMS supports constraints on metamodels that are expressed in Java only, but the GEMS Intelligence extension supports OCL and other constraint languages.

6.2 MOMENT Project

In *MOMENT-OCL* [18], OCL queries and invariants can be executed over instances of EMF models in the Maude algebraic language. An interesting feature of this algebraic specification of the OCL 2.0 is the use of parameterization to reuse the OCL specification for any meta-model or model. It is also worth noting the simulation of higher-order functions enabling the reuse of collection operator definitions. This is somewhat similar in spirit with the common library we proposed in section 4.3. However, few of the requirements we identified in section 2.2 would be satisfied by this environment in its current state, but an open API would improve the situation and help integrate this OCL engine with other tools.

6.3 EMF Validation Framework

The Eclipse *EMF Validation Framework* [5] shares many goals with our approach and could have been used to implement several of the functionalities offered by our tool. It was not considered mature enough at the time we started this work, but it has evolved substantially since then. Among other capabilities, this framework provides:

- An API for defining on-demand and live constraints for any EMF meta-model.
- Support for parsing the content of constraint elements defined in specific languages, with predefined support for Java and OCL.
- API support to define user contexts that describe the objects that need to be validated and to bind them to constraints that need to be enforced on these objects.

Support for Java constraints is not very useful in our context. Constraints can be provided statically by editing the plugin.xml file (which would violate requirement R1) or dynamically from another source. Rule selection can be achieved through programmable constraint filters.

Our approach is more fluid and usable by end-users as everything is driven from properties and dialog boxes, as illustrated in section 4. Also, our groups can replace this framework's filters and user contexts (in fact, we could have a group defined for a specific model if really needed). Our approach handles utilities in a better way, and we offer more usable reporting through clickable messages in the Properties view.

Still, our approach might benefit from this framework's support for "live" verification (ongoing, as the model gets changed) and for localization of constraint violation messages (e.g. in French). Even if jUCMNav is a multilingual application (English and French are fully supported), semantic constraint descriptions are currently unilingual.

Hence, this EMF validation framework should be considered as a good starting point for plug-in developers who would like to add semantic constraint management functionalities similar to those we implemented for jUCMNav.

7 Conclusions

In order to better support different modelling styles and standards, the checking of the quality of inputs to analysis algorithms, and the verification of cumulative assumptions on chains of transformation tools, it is imperative that modelling tools offer flexible and usable mechanisms for the verification of customizable semantic constraints.

We have identified minimal requirements (section 2) for the management of constraints in modelling tools. We have described issues related to OCL constraints and EMF, and then proposed solutions to the creation, verification, deletion, and import/export of constraints and groups thereof. Through the use of the User Requirements Notation and the jUCMNav tool, we have illustrated these solutions, which led to tool extensions now available as of release 3.1 (April 2008), with online demonstrations [14]. Typical applications and the value of our solutions were discussed in section 5, followed by a comparison to closely-related technologies. The approach proposed in this paper is mainly beneficial to modellers, but also to tool builders (especially if they create transformations) and language developers. Validation of the tool was done based on several experiments with different rules [27], but more empirical evidence of the usefulness and benefits of the approach is required.

The approach itself is independent of the modelling notation, constraint language, and tools used. The capabilities presented here should be considered for any modelling tool (e.g. for UML, Petri Nets, Business Process Modeling Notation, etc.), especially if implemented with EMF. They should improve the life of modellers as well as the quality of the models produced. Other constraint languages could replace OCL too.

One major obstacle to the adoption of such an approach is the need for a critical mass of people to know the constraint language (e.g., OCL) and the tool's underlying metamodel. Yet, the import/export mechanism we support can help mitigate this. Although in its infancy, we also expect the library of semantic constraints for jUCMNav to grow and become quite valuable over the next few years.

In the future, some potential features could be considered. For example, MDT OCL provides a "content assistant support" which parses partial OCL expressions and then supplies completion suggestions. With this feature, many helpful tips could be given in the rule definition editor. We could 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 OCL query expression. This technology could also enable the support of more general model queries and metrics based on the same OCL engine. An interesting challenge would be to reduce the burden of constraint writers by generating OCL query expressions automatically from an EMF metamodel.

Acknowledgements

The authors would like to thank Jason Kealey and Gunter Mussbacher for their help in developing this extension to jUCMNav. This work was supported in part by NSERC (Discovery grant) and the Ontario Research Network on Electronic Commerce.

About the Authors

Daniel Amyot is Associate Professor at SITE, University of Ottawa. His research interests include requirements engineering and business process modelling. He leads the development and standardization of the User Requirements Notation at ITU-T as well as the evolution of the jUCMNav Eclipse plug-in. He can be reached at damyot@site.uottawa.ca.

Jun Biao Yan completed his Masters in Computer Science at the University of Ottawa in April 2008. He is now Chief Technical Officer at Mesada Technology Co., in China. His email address is byrne.yan@yahoo.com

References

[1] S. Abdulhadi, G. Grau, J. Horkoff, and E. Yu. *i* Guide*. V. 3.0, August 2007. http://istar.rwth-aachen.de/tiki-ndex.php?page_ref_id=67

[2] D. Amyot. *Introduction to the User Requirements Notation: Learning by Example*. Computer Networks, 42(3), pages 285-301, June 2003.

[3] L.C. Briand, Y. Labiche, M. Di Penta, and H.-D. Yan. A Controlled Experiment on the Impact of the Object Constraint Language in UML-Based Development. In *IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 380-389, Chicago, USA, September 2004.

[4] Eclipse.org. *Eclipse Modeling Framework Project (EMF)*. 2008. http://www.eclipse.org/modeling/emf/

[5] Eclipse.org. *EMF Validation Framework Developer Guide*. 2007. http://help.eclipse.org/help33/nav/25

[6] Eclipse.org. *Generic Eclipse Modeling System (GEMS)* 2008. http://www.eclipse.org/gmt/gems/

[7] Eclipse.org. *MDT OCL SDK 1.1.2*, 28 November 2007.

[8] G. Franks, P. Maly, M. Woodside, D.C. Petriu, and A. Hubbard. *Layered Queueing Network Solver and Simulator User Manual*. Carleton University, Dec. 2005. http://www.sce.carleton.ca/rads/lqns/LQNSUserMan.pdf

[9] S. Ghanavati, D. Amyot, L. Peyton, and G. Mussbacher. A Compliance Framework for Business Processes Based on URN and DOORS. *2007 Telelogic User Group Conference*, Atlanta, USA, October 2007.

[10] IBM Corporation and others, *OCL Developer Guide*, 2007. http://help.eclipse.org/help33/nav/35

[11] ITU-T – International Telecommunications Union. *Recommendation Z.120 (04/04), Message Sequence Chart (MSC)*. Geneva, Switzerland, 2004.

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

[13] ITU-T – International Telecommunications Union. *Draft Recommendation Z.151, User Requirements Notation (URN)*. Geneva, Switzerland, April 2008.

[14] jUCMNav 3.1, April 2008. http://jucmnav.softwareengineering.ca/jucmnav/

[15] F.A. Kraemer. Arctis and Ramses: Tool Suites for Rapid Service Engineering. *Proceedings of NIK 2007 (Norsk informatikkonferanse)*, Tapir Akademisk Forlag, Oslo, Norway, November 2007.

[16] A. Ledeczi. *A Generic Modeling Environment, GME 5 User's Manual, Version 5.0.* Institute for Software Integrated Systems, Vanderbilt University, 2005.

[17] S. Maqbool. *Transformation of a Core Scenario Model and Activity Diagrams into Petri Nets*. M.Sc. thesis, SITE, University of Ottawa, September 2005.

[18] The MOMENT Project, MOMENT OCL, June 2007, http://moment.dsic.upv.es/

[19] G. Mussbacher, D. Amyot, and M. Weiss. Visualizing Early Aspects with Use Case Maps. *Transactions on Aspect-Oriented Software Development III*, Springer, pages 105-143, 2007.

[20] OMG – Object Management Group. *Object Constraint Language Specification, version 2.0*, May 2006.

[21] OMG – Object Management Group. *Unified Modeling Language (OMG UML): superstructure version 2.1.2*, November 2007.

[22] D.B. Petriu and C.M. Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Software and Systems Modeling*, 6(2), Springer, pages 163-184, June 2007.

[23] D.B. Petriu and C.M. Woodside. Software performance models from system scenarios. *Performance Evaluation*, 61(1), Elsevier B.V., pages 65-89, June 2005.

[24] A. Pourshahid, P. Chen, D. Amyot, A.J. Forster, S. Ghanavati, L. Peyton, and M. Weiss. Toward an integrated User Requirements Notation framework and tool for Business Process Management. In *3rd Int. MCeTech Conference on eTechnologies*. IEEE Computer Society, pages 3-15, Montréal, Canada, January 2008.

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

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

[27] J.B. Yan. *Static Semantics Checking Tool for jUCMNav*. Master's project, SITE, University of Ottawa, April 2008.

[28] Y.X. Zeng. *Transforming Use Case Maps to the Core Scenario Model Representation*. M.Sc. thesis, SITE, University of Ottawa, June 2005.