D4.5.1-Model Views Prototype

Virtual EMF – a Model Virtualization Tool

	NAME	PARTNER	DATE
WRITTEN BY	CLASEN C.	ATLANMOD	28/11/2011
REVIEWED BY	BERNARD Y.	Airbus	

RECORD OF REVISIONS

ISSUE	***DATE***	***EFFECT ON***		***REASONS FOR REVISION***
		PAGE	***PARA***	
1.0	28/10/2011			Document creation

TABLE OF CONTENTS

TABLE OF APPLICABLE DOCUMENTS

N°	TITLE	REFERENCE	ISSUE	DATE	SOURCE	
					SIGLUM	NAME
A1						
A2						
A3						
A4						

TABLE OF REFERENCED DOCUMENTS

N°	TITLE	REFERENCE	ISSUE
R1	D1.2.2 Galaxy glossary		
R2	D3.1 Model Views Conceptual Approach		
R3			
R4			

ACRONYMS AND DEFINITIONS

Except if explicitly stated otherwise the definition of all terms and acronyms provided in [R1] is applicable in this document. If any, additional and/or specific definitions applicable only in this document are listed in the two tables below.

Acronymes

ACRONYM	DESCRIPTION

Definitions

TERMS	DESCRIPTION

1. INTRODUCTION

MDE systems can rely on a large number of heterogeneous models. These models can be very large, and highly interrelated, resulting in growing complexity. Frequently, one needs to manipulate and combine information scattered in several models in order to create a specific view of the system, more manageable and targeting specific scenarios. An overview of this problem and the subsequent conceptual research proposing a solution based on model virtualization is described in the Galaxy deliverable D.3.1.

The current document presents the next step: the realization of the conceptual approach in the form of a prototype tool, i.e., Virtual EMF, a model virtualization engine to support the creation of model views through virtual models. Virtual EMF has been developed on top of the Eclipse Modeling Framework (EMF) and is available as an open-source project in Eclipse Labs at https://code.google.com/a/eclipselabs.org/p/virtual-emf/.

This document is organized as follows: Section 2 describes how to install Virtual EMf. Section 3 provides details on its implementation and architecture. Section 4 presents a brief user guide to defining virtual modes. Finally, Section 5 introduces some performance tests used to validate our tool.

2. INSTALLATION PROCESS

Virtual EMF (tool and source code) can be retrieved online from Eclipse Labs at:
https://code.google.com/a/eclipselabs.org/p/virtual-emf/

Two ways are available to install Virtual EMF:

1. Install from source:
    a. Check out plugins from Subversion repository using the following URL:
        i. https://svn.codespot.com/a/eclipselabs.org/virtual-emf/
    b. Add projects to Eclipse workspace
    c. Launch a runtime Eclipse instance
2. Add the plugins as Jar files to the Eclipse 'dropins' folder
    a. Jar files are available in the download section in the 'Virtual_EMF_plugins.zip' file

An update site for Virtual EMF is coming soon.

3. VIRTUAL EMF - IMPLEMENTATION

This section presents a brief recapitulation of virtual models concepts, as well as Virtual EMF's architecture, to guide the reader throughout the rest of this document. This has been discussed in further detail in deliverable D.3.1.

3.1 VIRTUAL MODELS OVERVIEW

In short, a virtual model is a non-materialized model whose (virtual) model elements are proxies to elements contained in other models. A virtual model does not hold concrete data. Modeling tools and users perceive and manipulate it as a regular materialized model, but in fact the actual elements being accessed are directly retrieved from its contributing models. This allows seamless integration and reuse of original input data into virtual models, without need for duplicating it.

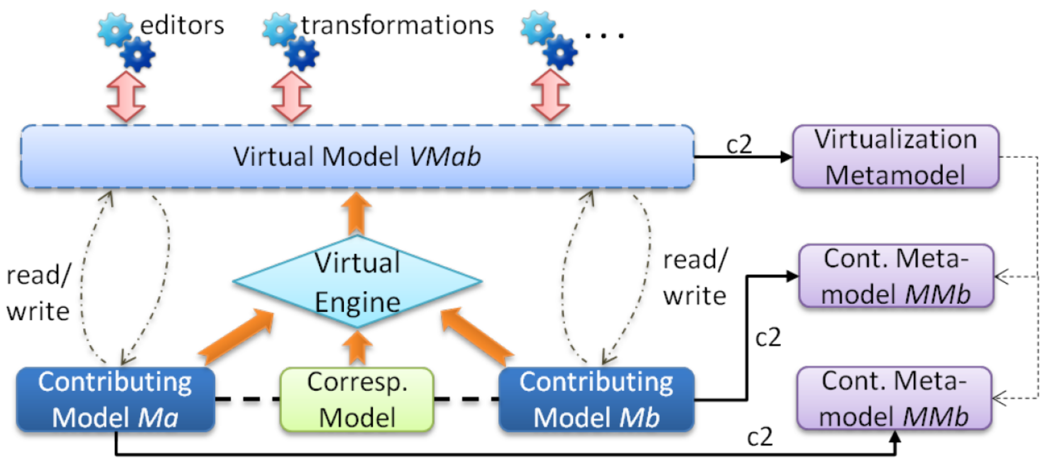


Figure 3-1. Model Virtualization Overview.

Figure 3-1 depicts the idea. A virtual model conforms to a *virtualization metamodel* and aggregates information from different *contributing models*. Based on the relationships between them, a virtualization engine combines its elements in different ways according to *translation rules*. The translation rule to be applied to each element is defined in a *correspondence model* that contains *virtual links* pointing to contributing elements. Figure 3-2 showcases a sample translation process.

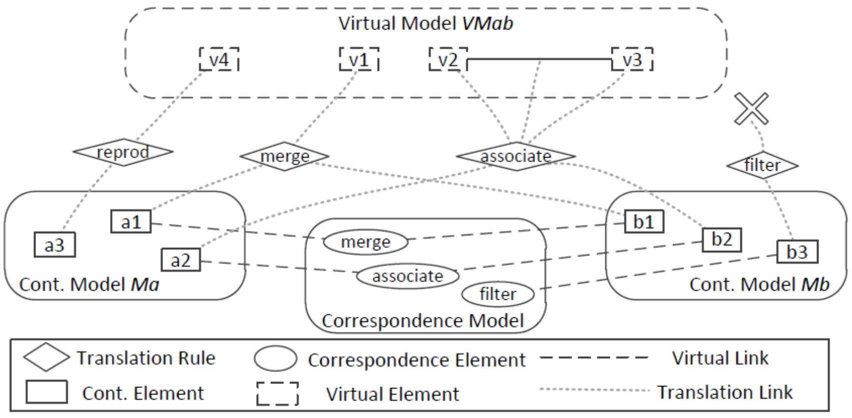


Figure 3-2. Translation of contributing elements into virtual elements.

3.2 ARCHITECTURAL OVERVIEW

Virtual models must appear as normal materialized models to users to guarantee their full compatibility with existing EMF-based modelling tools. This means the tool must redefine the behaviour of all EMF modelling operations to provide an adequate, but transparent, support for virtual models.

This is done by provide an alternative implementation of EMF's interfaces (mainly *Resource* and *EObject*, and *EList*) and corresponding methods. Examples of methods to be implemented are *Resource.load()*, *Resource.save()*, EObject.eGet(), *EObject.eSet()*, EList<*EObject*>*.add()*, and so on. This is accomplished by the two main components:

1. Virtualization API: The main component. In charge of managing the resources (models) involved in the virtualization process. It refines the behaviour of EMF's standard API to provide an adequate support for access and manipulation of virtual models.

2. Translation API: A sub-component that implements the execution semantics of translation rules.

Fig. 3 presents the relationship between our APIs with the other components of EMF. Tools access the virtual model by using EMF's interfaces, without concern about the nature of the underlying model they are accessing (e.g. XMIResource is used for XMI models; CDO for database models, and so on). Calls to virtual models are intercepted by the Virtualization API, which automatically identifies referenced element/s and with the aid of the Translation API redirects the operation to contributing models and elements. Virtualization and Translation APIs, in their turn, interact with contributing model elements through a regular implementation of the Model Access API, according to their nature.

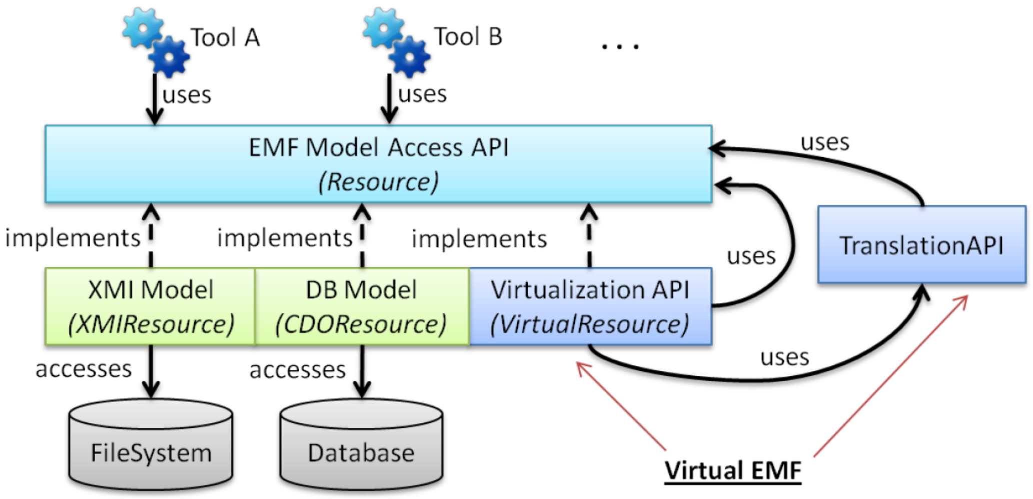


Figure 3-3. API relationship in Virtual EMF

4. USER GUIDE

4.1 CREATING A VIRTUAL MODEL

To initialize a virtualization process, users must provide the following input artifacts:

1. Contributing models: the models to be virtualized;
2. Contributing metamodels: the metamodels of the contributing models;
3. Virtualization metamodel: the metamodel of the virtual model;
4. Correspondence model: a (weaving) model containing the relationships between contributing models.

4.1.1 The Correspondence Model

The correspondence model is a (weaving) model whose elements identify relationships between contributing elements. I.e., a virtual link may relate two contributing elements that should be merged. These relationships are internally processed by Virtual EMF to transparently manipulate the composed resulting element.

Three types of correspondence links are currently supported:

1. **Virtual associations**: associations between elements in different contributing models. Navigated as normal internal references. All normal reference properties are supported (opposites, multi-valued, etc...);
2. **Merge**: merging of two contributing elements into a single merged element. All contributing attributes and properties are available in the merged element. If single-value properties overlap, the merged element uses the one from the preferred element. If properties are multi-value, they are combined into a single list;
3. **Filter**: discard this element from the virtual model.

In the case no virtual link points to a given contributing element, it is simply reproduced with the same properties as a virtual element. The correspondence model conforms to a Virtual Links metamodel (simplified diagram in Figure 4-1.).

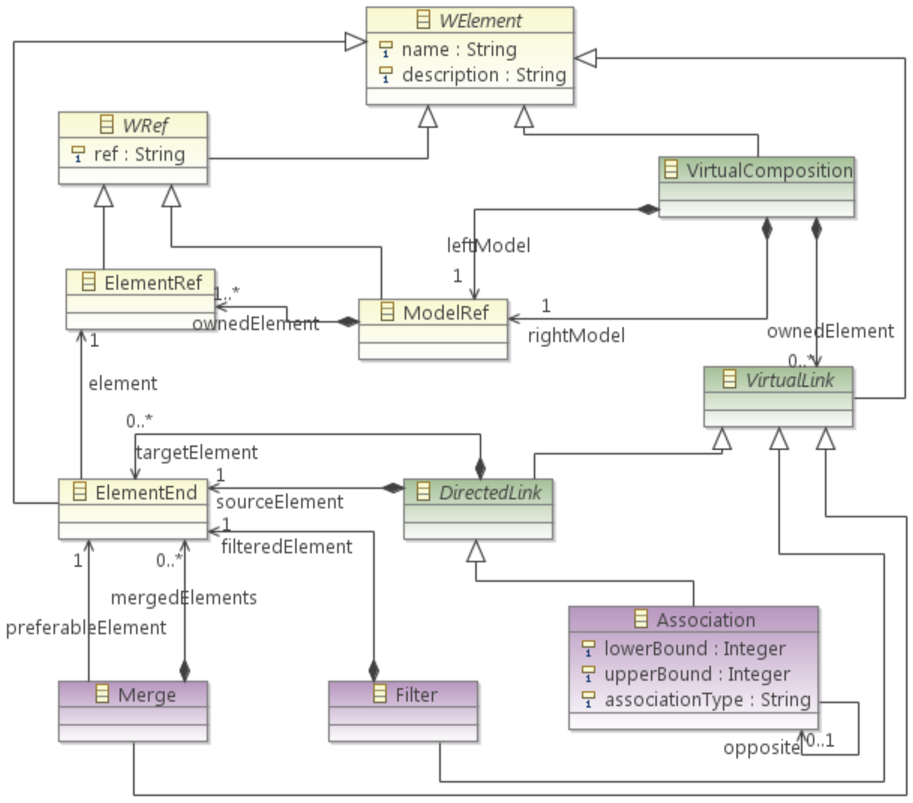
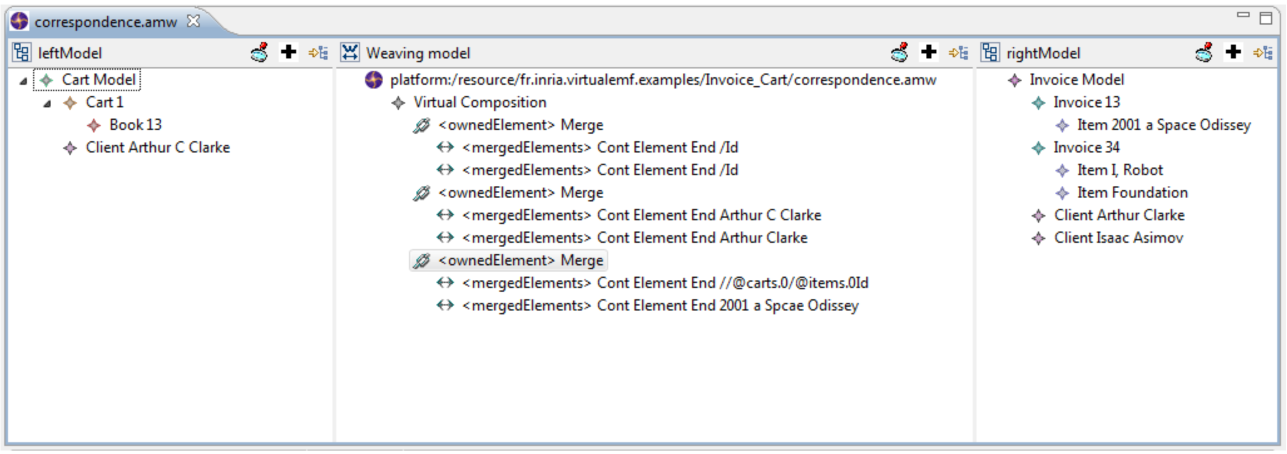


Figure 4-1. Simplified Virtual Links metamodel.

There are several ways to create a correspondence model. It can be automatically generated from a matching process that compares contributing models. Several works have been already conducted on discovering these relationships (e.g. composition languages, matching model transformations, etc.). Virtual EMF is agnostic to these methods as it only expects the result of the matching. Different matching techniques can then be plugged in Virtual EMF to achieve a fully automated virtualization process. Alternatively, it can be manually defined with the aid of a model editor (figure below display the creation of a correspondence model with the AMW tool).


correspondence.amw

leftModel
- Cart Model
  - Cart 1
    - Book 13
  - Client Arthur C Clarke

Weaving model
- platform:/resource/fr.inria.virtualemf.examples/Invoice_Cart/correspondence.amw
  - Virtual Composition
    - <ownedElement> Merge
      - <mergedElements> Cont Element End /Id
      - <mergedElements> Cont Element End /Id
    - <ownedElement> Merge
      - <mergedElements> Cont Element End Arthur C Clarke
      - <mergedElements> Cont Element End Arthur Clarke
    - <ownedElement> Merge
      - <mergedElements> Cont Element End //@carts.0/@items.0Id
      - <mergedElements> Cont Element End 2001 a Spcae Odissey

rightModel
- Invoice Model
  - Invoice 13
    - Item 2001 a Space Odissey
  - Invoice 34
    - Item I, Robot
    - Item Foundation
  - Client Arthur Clarke
  - Client Isaac Asimov

Figure 4-2. Use of AMW for defining the correspondence model.

4.1.2 The Virtualization Metamodel

As any model, a virtual model conforms to a metamodel that defines the concepts that can populate it. In the case of virtual models, this is a virtualization metamodel that essentially captures the concepts of the contributing metamodels, with the addition of concepts deriving specifically from its composition. For instance, if a reference is created between elements in different contributing models, it must conform to a reference in the virtualization metamodel. Or, if contributing elements are to be merged into a single element, it must conform to a *EClass* capturing the merged concepts.

The virtualization metamodel can be defined by hand by the user. But, as this can be a tedious task, an ATL transformation is available (in the *fr.inria.virtualemf.composition-mm* ATL project) to partially automate this task (by providing an additional weaving model with the relationships between contributing metamodels).

4.1.3 Launching a Virtual Model

There are two ways to launch the virtualization process:

1. Programmatically, by providing virtualization resources as arguments to the constructor of VirtualResource (Figure 4-3).
2. Via a virtual model file (with the .virtualmodel extension) containing the location paths of the resources (Figure (4-4).

```
VirtualResource virtualModel = new VirtualResource(Resource[]
     contributingModels, Resource[] contributingMetamodels, Resource
     virtualizationMetamodel, Resource correspondenceMetamodel);
```

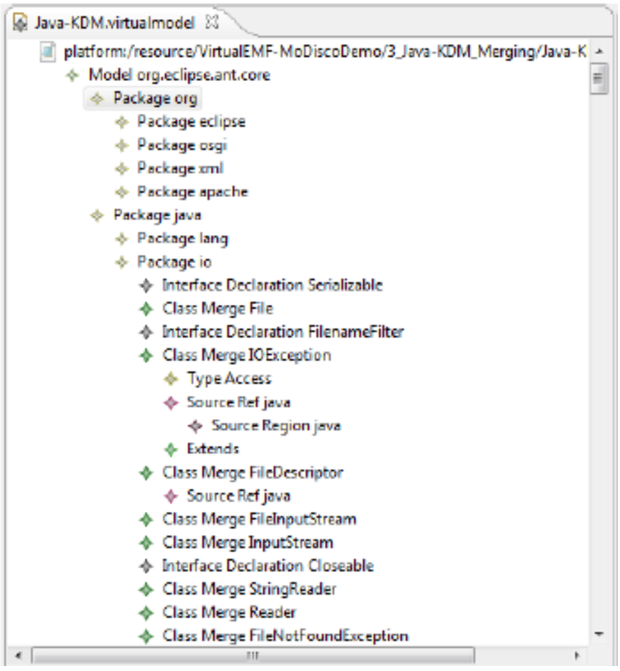
Figure 4-3. Creating a virtual model by Java code.

```
contributingModels = /DemoModels/org.eclipse.ant.core_java.xmi, /DemoModels/
     org.eclipse.ant.core.uml
contributingMetamodels = /DemoModels/java.ecore, uri:http://www.eclipse.org/
     uml2/3.0.0/UML
correspondenceModel = /DemoModels/Java-UML_traces.amw
virtualizationMetamodel = /DemoModels/Java-UML.ecore
```

Figure 4-4. A sample virtual model file

Virtual EMF automatically associates within the Eclipse environment the *.virtualmodel* extension with our tool. This association informs Eclipse that the Virtualization API has to be used when handling, loading or saving models "stored" in a *.virtualmodel* file. Therefore, a virtual model file can be simply given as input to any EMF-based modeling tool (e.g. double-clicking a *.virtualmodel* file will automatically load the virtual model in the standard EMF model editor).

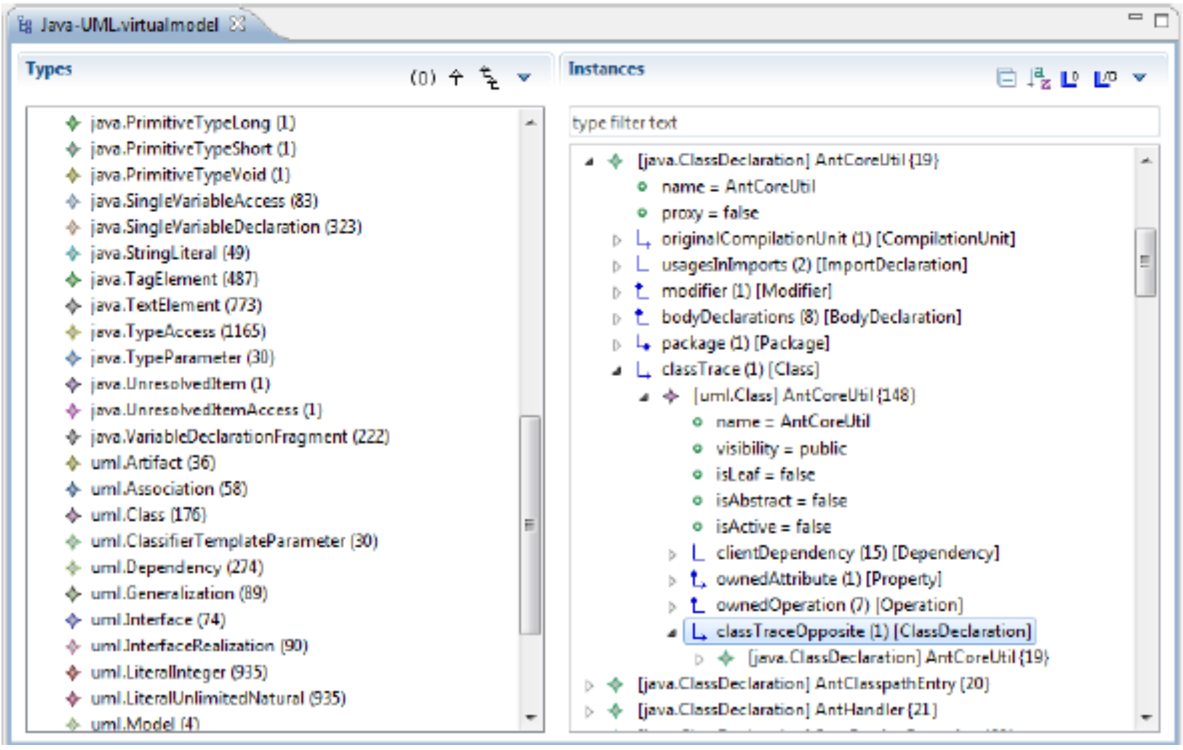
As an example Figure 4-5 showcases a virtual model generated from the composition of a Java model with a KDM model (and merged elements between them) opened from a virtual model file in the standard Ecore Reflexive Model Editor.



```
Java-KDM.virtualmodel
platform:/resource/VirtualEMF-MoDiscoDemo/3_Java-KDM_Merging/Java-K
  Model org.eclipse.ant.core
    Package org
      Package eclipse
      Package osgi
      Package xml
      Package apache
    Package java
      Package lang
      Package io
        Interface Declaration Serializable
        Class Merge File
        Interface Declaration FilenameFilter
        Class Merge IOException
          Type Access
          Source Ref java
            Source Region java
          Extends
        Class Merge FileDescriptor
          Source Ref java
        Class Merge FileInputStream
        Class Merge InputStream
        Interface Declaration Closeable
        Class Merge StringReader
        Class Merge Reader
        Class Merge FileNotFoundException
```

Figure 4-5. Java-KDM virtual model viewed in the Ecore Reflexive Model Editor.

Similarly, Figure 4-6 shows a virtual model generated from the composition of a Java model with a UML model (with inter-model traceability links between its elements).



```
Java-UML.virtualmodel

Types (0)
  java.PrimitiveTypeLong (1)
  java.PrimitiveTypeShort (1)
  java.PrimitiveTypeVoid (1)
  java.SingleVariableAccess (83)
  java.SingleVariableDeclaration (323)
  java.StringLiteral (49)
  java.TagElement (487)
  java.TextElement (773)
  java.TypeAccess (1165)
  java.TypeParameter (30)
  java.UnresolvedItem (1)
  java.UnresolvedItemAccess (1)
  java.VariableDeclarationFragment (222)
  uml.Artifact (36)
  uml.Association (58)
  uml.Class (176)
  uml.ClassifierTemplateParameter (30)
  uml.Dependency (274)
  uml.Generalization (89)
  uml.Interface (74)
  uml.InterfaceRealization (90)
  uml.LiteralInteger (935)
  uml.LiteralUnlimitedNatural (935)
  uml.Model (4)

Instances
type filter text
  [java.ClassDeclaration] AntCoreUtil {19}
    name = AntCoreUtil
    proxy = false
    originalCompilationUnit (1) [CompilationUnit]
    usagesInImports (2) [ImportDeclaration]
    modifier (1) [Modifier]
    bodyDeclarations (8) [BodyDeclaration]
    package (1) [Package]
    classTrace (1) [Class]
      [uml.Class] AntCoreUtil {148}
        name = AntCoreUtil
        visibility = public
        isLeaf = false
        isAbstract = false
        isActive = false
        clientDependency (15) [Dependency]
        ownedAttribute (1) [Property]
        ownedOperation (7) [Operation]
        classTraceOpposite (1) [ClassDeclaration]
          [java.ClassDeclaration] AntCoreUtil {19}
  [java.ClassDeclaration] AntClasspathEntry {20}
  [java.ClassDeclaration] AntHandler {21}
```

Figure 4-6. Java-UML virtual model viewed in MoDisco's Model Browser.

Virtual EMF has been also tested within other modelling tools (e.g. ATL transformations). Examples of virtual models are available online and can be found on Virtual EMF's website.

5. PERFORMANCE TESTS

This section focuses on the performance of Virtual EMF.

We have conducted tests measuring the memory usage, creation time, and manipulation time of Virtual EMF when compared to a traditional composition approach. As a representative baseline approach for the comparison we use a composition technique based on the execution of ATL model transformations to translate the elements from contributing to composed models.

As a composition scenario, we have adopted one of the simplest possible:

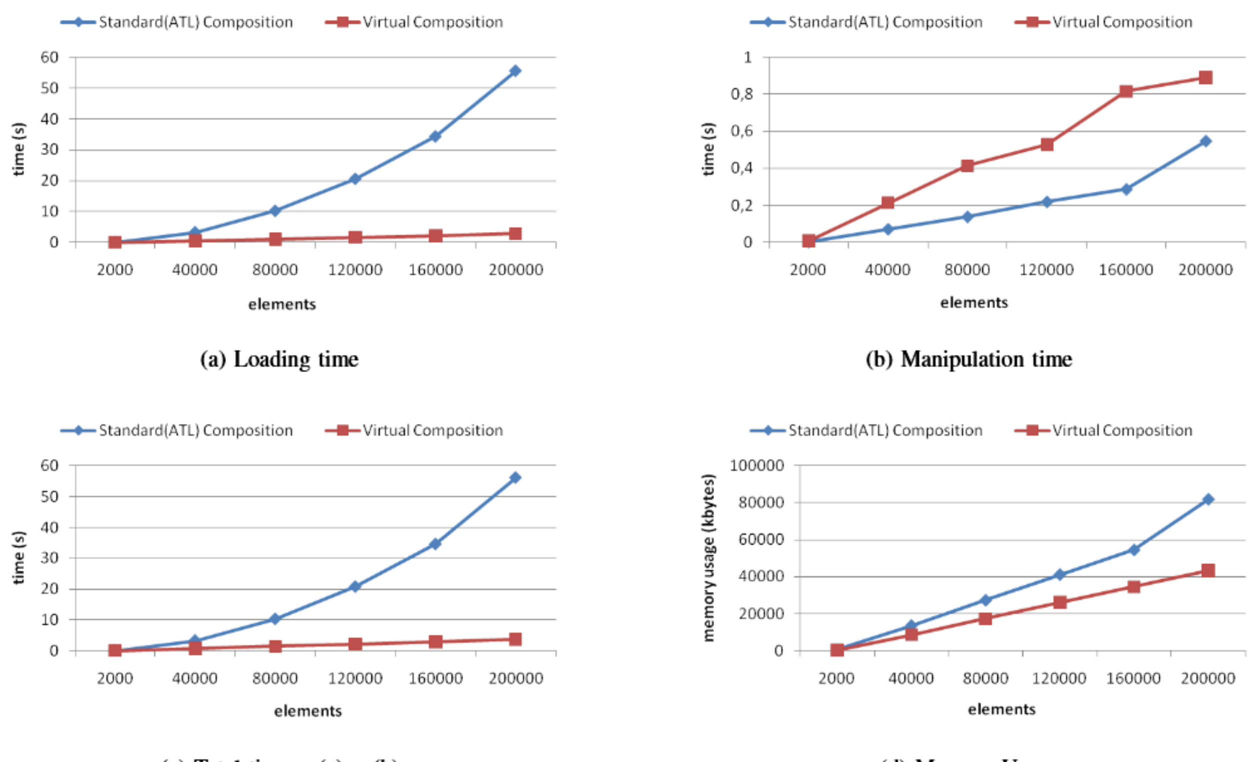
1. All elements from two contributing models are included in the composed model
2. Virtual relationships are defined between elements of both models (more specifically, randomly one association per each ten model elements).

We believe this scenario allows a fair comparison of the performances of both approaches. We have tested this scenario for different contributing models sizes (between 1000 and 100.000 elements for each contributing model). All tests were executed in the same machine and OS (Intel Core 2 Duo E6850 3.0GHz, 4GB RAM. Windows 7 Ultimate x64). Each test was repeated twenty times with the average value used as final value for that test.

The tests consisted in:

1. Measuring the time to create the composed model;
2. Measuring the time to manipulate the composed model assuming that we (1) read (*eGet()*) all contributing elements and its properties and (2) modify (*eSet()*) the values of all elements. This may be seen as a worst case scenario for our approach, since in a vast majority of cases, only a subset of the elements in the composed model would be accessed, and even less modified;
3. Measuring the total memory usage when creating the composed model (considering all models involved in the virtualization process). We consider the memory usage of both the contributing and composed model since at least at some point (e.g., during the creation of the composed model) all these models would coexist in memory, with this moment becoming a bottleneck (besides, the contributing models could continue to be used in different modeling scenarios).

A summary of the results is shown in Figure 5-1. As it can be seen, our approach behaves far better when creating the composed model and the extra time required when manipulating it can be disregarded. Regarding the memory usage, since we do not duplicate model elements, the usage is essentially equal to the sum of the size of the contributing models (with a small addition coming from the virtual associations).Clearly, there are also situations in which materialized composed model could perform better (e.g. when the composed model is very small in comparison with the contributing ones; here being able to work only with the small composed model would offer some advantages) but we believe that in most application scenarios our approach behaves much better.



(a) Loading time

(b) Manipulation time

(c) Total time = (a) + (b)

(d) Memory Usage

Figure 5-1. Comparison of two view generation approaches: ATL transformation and Virtual Composition (Virtual EMF). The x-axis indicates the total number of contributing elements (sum of the size of each contributing model).