xUMLi: Towards a Tool-independent UML Processing Platform

Jani Airaksinen, Kai Koskimies, Johannes Koskinen,
Jari Peltonen, Petri Selonen, Mika Siikarla, Tarja Systä
email: {jairaksi, kk, jomppa, jpe, pselonen, miksi, tsysta}@cs.tut.fi
Institute of Software Systems
Tampere University of Technology, Box 553, FIN-33101 Tampere, Finland

Abstract

Being a modeling language rather than a method, UML does not inherently define a design process. Consequently there is no common basis for building design process support for UML tools. They often offer extension interfaces, which allow external applications to access and handle the UML models constructed using that particular tool. Although they support extensibility and customizability, such tool-dependent solutions are typically suitable only for performing single operations or tasks.

In this paper, we propose a tool-independent software platform, which allows the user to build and combine various kinds of UML processing facilities and use them from integrated CASE-tools. The proposed solution conforms to the UML standard metamodel and hides the individual conventions and complexities of the CASE-tool APIs. We discuss our solution, its architecture and API, and also illustrate their usage by giving an example.

1. INTRODUCTION

UML [OMG02] has become an industrial standard for the presentation of software design models, and is supported by all major CASE-tool vendors. The success of UML is largely based on its conservative approach: the most important diagram types in UML (like class diagrams, sequence diagrams, and statechart diagrams) have already been used in software engineering for a long time in different variations. Therefore, UML should be understood essentially as a standardization effort, rather than yet another graphical design notation.

UML is a standard for a modeling language, not for a design method. This gives considerable freedom for tool developers that may have rather different assumptions of the roles played by UML diagrams in a software development process. Tool developers may also have different solutions for the internal representation of UML models, and for accessing them from external applications. Further, tools may support UML only partially, or they may support different versions of UML, or they may customize UML with their own notations and conventions. Finally, both the tools and UML itself are under constant evolution. All this implies that although in principle UML would be an ideal conceptual basis for developing general techniques to support

the software design process, tool support for these techniques must be implemented separately for each CASE-tool.

Recently, XMI [Cov01] (XML Metadata Interchange) has been introduced as a standard for representing MOF-based (Meta Object Facility) models in XML. Hence XMI can be used as an exchange format for UML models as well, and indeed many UML tools support importing and exporting XMI. While XMI could in principle be used as a standard interface between UML applications, in practice this approach does not work well for model processing actions that are performed frequently during design sessions. The reason is the heavy interpretation and generation process implied by XMI. Especially actions, which need to access many UML model elements spread out in the model, become far too slow for practical purposes. XMI is a solution for porting UML models from one tool environment to another, but it is not suitable (or even intended) for an interactive working mode. Another drawback of XMI is that it does not cover visual information but only logical model data.

Many available UML-based CASE-tools provide an application programming interface (API) for extending the tool with auxiliary, possibly company-specific facilities. These interfaces allow any external component to fairly easily access and modify the model data, but they fall short in providing stronger support for typical UML model processing activities. For example, there is often a need to navigate within the model and to find model elements with certain properties. It would be obviously advantageous to implement this functionality only once in the platform implementing the API.

To summarize, we need a tool-independent API for processing UML models created by various kinds of tools, providing not only primitive access to the model but also more high-level support for implementing model processing actions. In the ideal case such a standard API would be directly supported by all tool vendors. Since this situation cannot be realized in the foreseeable future, we propose separate software platform that implements this interface in a tool-independent way. Such a platform (xUMLi, executable UML interface) has been developed by the authors as part of a research project investigating advanced tool support for UML modeling ([Kos01], [MS01], [Pel00], [PS01], [SKS01], [SSK01]). In this project, we have developed (i) transformation techniques between different UML diagram types, (ii) model combination techniques for merging the information contents of two UML models represented with diagrams of the same type, and (iii) a general scripting mechanism for constructing more complicated operations, based on the primitive model transformation and combination operations. The scripting mechanism is implemented within xUMLi itself, while the model operations are implemented by components written against the xUMLi API, called UMLi (UML interface). xUMLi can accept script descriptions and execute them, calling external model operation components which, in turn, access the model data through UMLi.

To actually make use of xUMLi, it is assumed to be integrated with some UML-based CASE-tool which allows the creation and graphical editing of UML models. Currently, xUMLi is integrated with Rational Rose [Ros01]. The integration is based on importing and exporting the UML models making use of the Rose API. Hence all existing (and future) model operation components developed on top of the xUMLi platform will be available in Rose as well, and can be used as parts of scripts.

This paper is a short introduction to xUMLi and its rationale. In Section 2 we discuss the basic architecture of xUMLi. In Section 3 we present an example of using xUMLi from the viewpoint of a composite script calling model operations, and in Section 4 we show how primitive model operations can be implemented against UMLi. In Section 5 we discuss related work and finally some concluding remarks are presented in Section 6. We assume the knowledge of UML throughout the paper.

2. ARCHITECTURE

The xUMLi platform can be conceptually understood as a layered architecture. The lowest layer consists of the script engine (VISIOME) that eventually executes scripts. VISIOME defines a domain-independent, general data model that can be specialized for a variety of purposes. VISIOME layer also implements OCL as a general mechanism for navigating and querying within a data structure, independently of the domain of the data. OCL (Object Constraint Language) is a standard notation included in UML, intended for writing constraints and queries over object models expressed in UML.

The second lowest layer specializes the general data model, providing access to UML models according to the UML metamodel. This layer is used through UMLi, which serves as the basic API for xUMLi.

We have developed components that exploit UMLi and that are included in the xUMLi platform. These components fall into two categories: (i) export/import components for other UML tools and XMI, and (ii) primitive UML model processing operations that are assumed to be used as parts of more complex operations. The latter operations include UML diagram type transformation operations and model combination operations [SKS01]. Currently there are export/import components for Rational Rose. These components constitute the third layer.

Finally, the topmost layer consists of the UML-based CASE-tools (and XMI) supported by export/import components and user-defined scripts that call the existing components on the lower layer. The actual xUMLi platform consists of the lowest three layers (Figure 1).

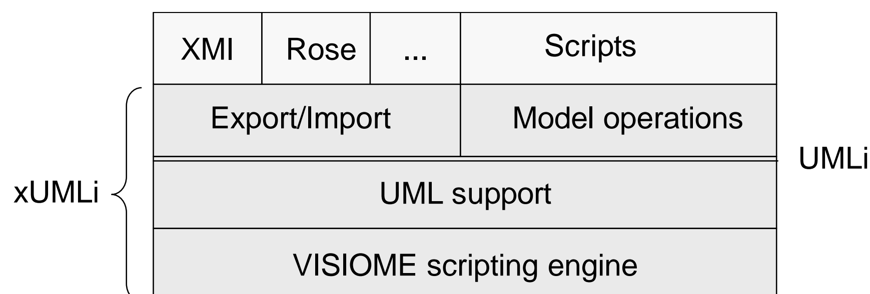


Figure 1. The layered architecture of xUMLi

The xUMLi platform can be extended in many ways: one can introduce new UML processing components on top of the UMLi layer, write new export/import components for another UML tool, port the UML models from xUMLi to another tool

using the XMI bridge, or write new scripts supporting certain (say, company-specific) UML-based design tasks.

Since UMLi has been implemented using the COM component model with automation support, writing components on top of UMLi becomes very easy. Any language supporting COM automation can be used for those components. We have currently used Python and C++ in writing the components. This kind of general interface has turned out be a valuable asset especially for exploratory, research-oriented implementation of various advanced UML processing capabilities. In particular, the Python components can be written on a high conceptual level, without deeper understanding of the underlying implementation.

The scripts are assumed to be given in a visual form through the graphical UML tool used (for example, Rose). Additional components are then needed to transform the visual representation into the form required by the script engine and back. Currently the script is transmitted simply in XML form to the engine. Hence, in principle any tool that can produce this XML representation can be used as a scripting tool.

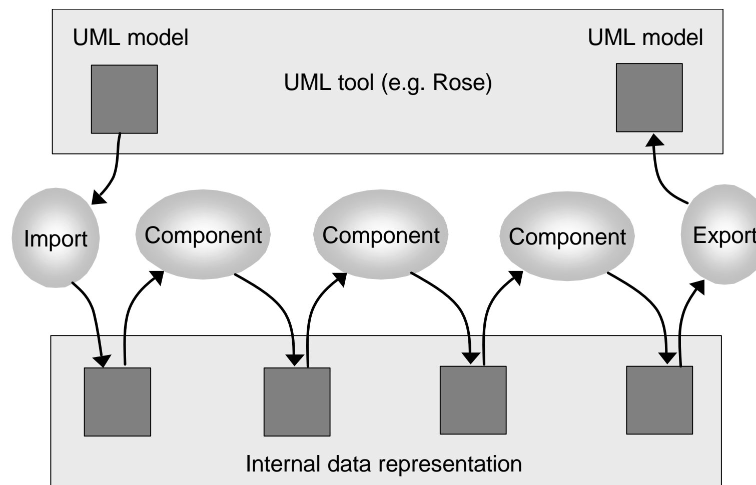


Figure 2. Execution of a script. Components implement various model processing operations that are combined by the script.

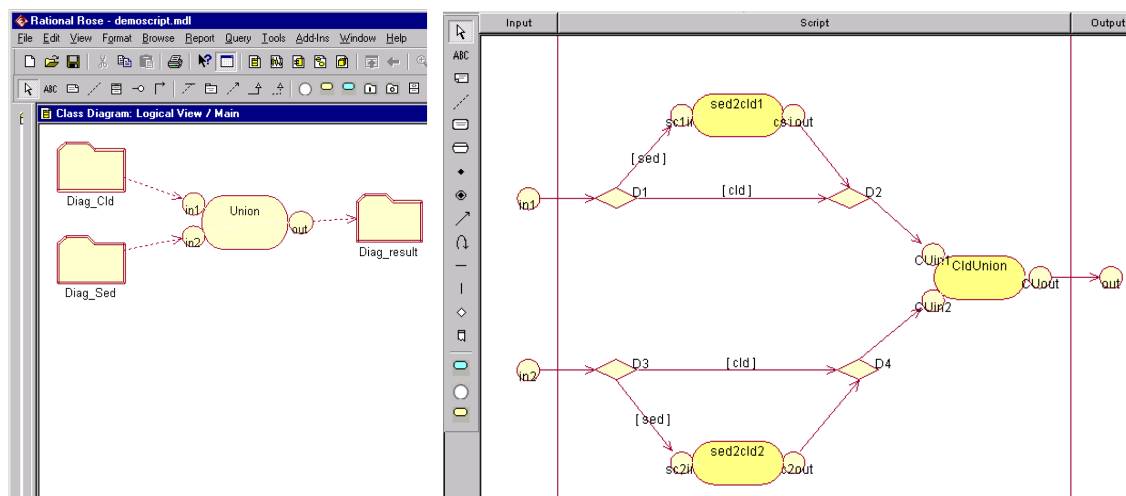
From a functional viewpoint, xUMLi can be understood as a pipes-and-filters architecture. In a typical usage scenario, the designer activates a VISIOME script from a graphical UML tool (e.g., Rose). The script is transformed to an internal representation and handed over to the script engine. The script engine then takes care of transmitting the data (UML models) between the components of the script. Initially, the input data for the script is extracted from the UML tool and imported into xUMLi. When the script is run to a completion, the output data is again exported from xUMLi to the UML tool as a new UML model, as specified in the script. Each component used by the script can be implemented separately from other components, based on its inputs and outputs only. Note that all intermediate results in the script are represented within xUMLi only, without transmitting them back to the UML tool. In this way complex UML processing operations can be carried out fairly efficiently, based on the internal data representation of xUMLi. The process is illustrated in Figure 2.

3. USING THE XUMLI PLATFORM: AN EXAMPLE

In this chapter we show how the xUMLi platform can be used with a CASE-tool. We give an example where Rational Rose is used to create models, and a script that combines operations implemented on top of the xUMLi platform. In the example we perform a chain of operations to combine the structural information of two input diagrams and show the result as a class diagram. Figure 3 shows the corresponding script as it is presented in Rational Rose.

Figure 3a shows the high level view of the script, where the input diagrams (Diag_Cld, Diag_Sed) are tied to the script and a place for the result diagram is given. The script elements are visualized using stereotyped classes with modified appearance. As such they can be used anywhere in Rational Rose.

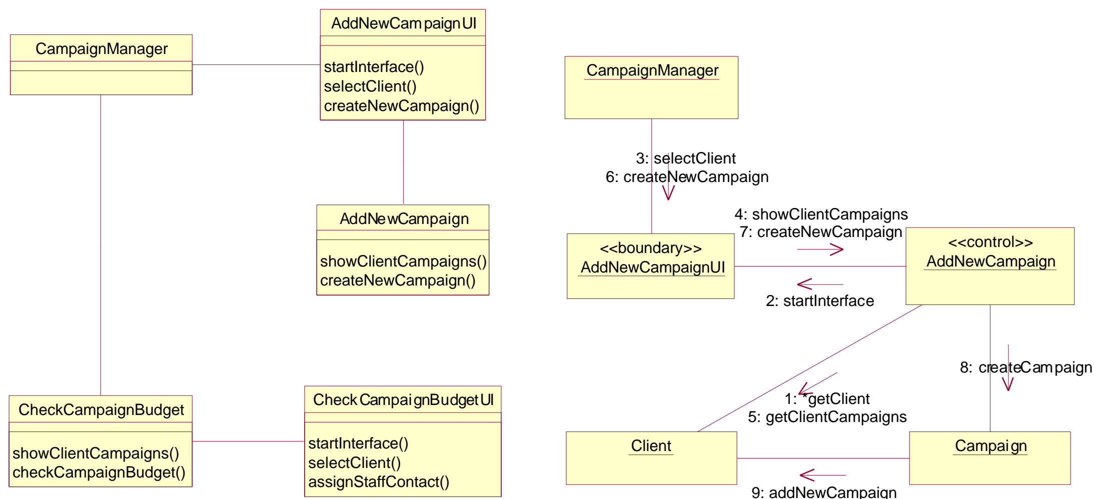
Figure 3b shows the description of the script *Union*. VISIOME notation is based on the UML activity diagram notation, so the actual script descriptions are presented as stereotyped activity diagrams in Rose. A VISIOME script has three areas, presented as swimlanes. Input area describes the input interfaces of the script, output area describes the output interfaces, and script area describes the actual script execution.



a. high level script *b. actual script description*

Figure 3. A Visiome script in Rational Rose.

The *Union* script presented in Figure 3b takes two diagrams as its inputs. If a diagram is an interaction diagram (decisions *D1* and *D3*), i.e. a sequence diagram or a collaboration diagram, the operation *Sed2Cld* [SKS01] is used to transform it into a class diagram. If a diagram is already a class diagram, no transformation is needed. If either of the diagrams is something else than an interaction or a class diagram, this particular script does not do anything. Execution ends, because there are no activities that can be executed. When both of the input diagrams are in the form of a class diagram, operation *CldUnion* is executed on them. *CldUnion* performs a union operation for two class diagrams, producing a class diagram, combining information from both of the input diagrams. Finally, the result is passed to the *out* interface of the script. The input diagrams, one class diagram and one collaboration diagram, used in the example are shown in Figure 4.



a. The class diagram given as input Diag_Cld. *b. The collaboration diagram given as input Diag_Sed.*

Figure 4. The input diagrams for the script. The diagrams are adapted from [BMF02].

The activities in a VISIOME script can be either other VISIOME scripts or model processing operations written against UMLi (e.g., Python components). In this particular script, all the activities are of the latter type. Guards can be used to set conditions on the flows in a VISIOME script. They are normally in the form of OCL expressions, like the one below. The flows seen in Figure 3b are combined control and data flows. The data is an object that hierarchically consists of other objects according to UML metamodel. The data content of a flow can be navigated starting from the object representing the data. This object is referred to with keyword *self*.

```
self.ownedElement.metaclass->exists(z|z='Collaboration')
```

This expression checks whether there is an interaction diagram in the given data by searching for metaclass “Collaboration” from it. If a collaboration is found, there is an interaction diagram in the data. For convenience, we have named the complete OCL expressions to find interaction and class diagrams as *Sed* and *Cld* (see Figure 3b), correspondingly.

One possible execution path of the script is shown as a sequence diagram in Figure 5. When the user wants to run the script, she opens the context sensitive menu of *Union* script, and selects *Start Script*. The Rose specific script parser is launched and the script diagrams in Rose are translated into the internal script format of VISIOME. VISIOME interprets the script and starts the execution by importing the required data separately for each input diagram. Data is passed to the activities according to the flows and conditions in the script. For each activity, VISIOME enquires whether the activity can be performed with the given data or not. If all the required data is available for an activity, it is performed; if not, the execution of the activity is postponed. If there are no activities that can be executed, the script execution ends.

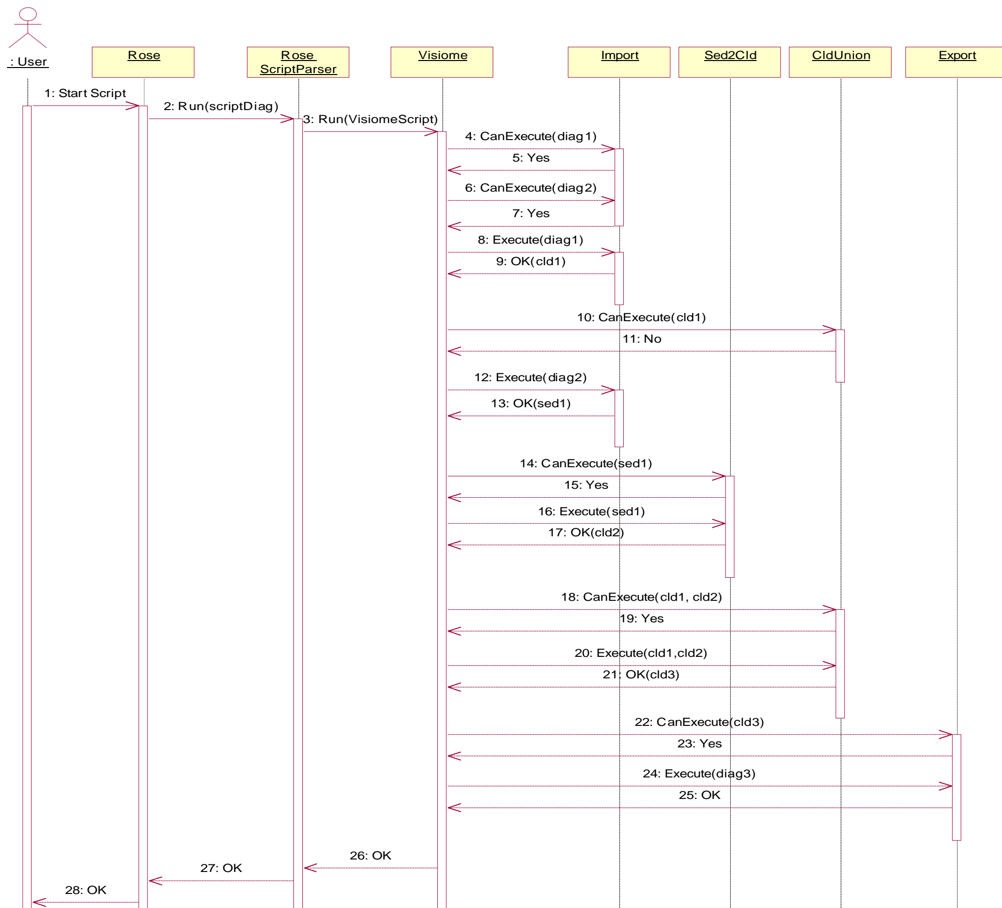


Figure 5. A sequence diagram describing a script execution path.

Let us assume that the script is interpreted "top down" according to Figure 3b. In this particular case, the first input is a class diagram, which is passed directly to *CldUnion* operation after the import. VISIOME then enquires if the operation can be executed. Because the operation does not have all the needed data, it answers no. Since the other input is a collaboration diagram, it is translated into a class diagram after the import operation. The resulted class diagram is then passed to *CldUnion* operation. At this point, VISIOME enquires again if the operation can be executed. Now all the needed data is available, and the operation can be performed. As the last operation, the resulting class diagram is exported to Rational Rose. The result of the script is shown in Figure 6.

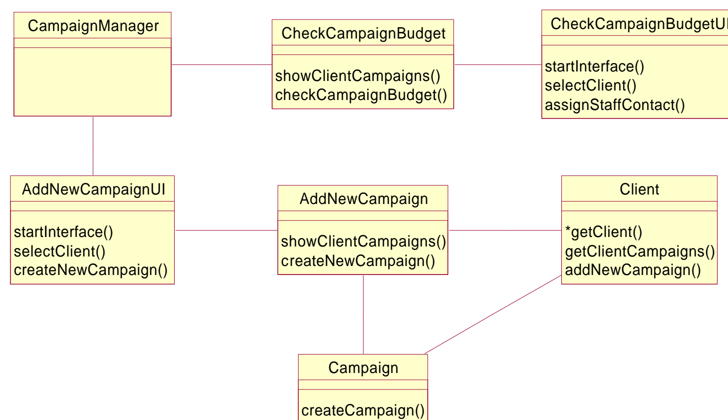


Figure 6. The result of the script. The layout has been modified by hand.

By looking at Figure 5, we can see the connections between the layers of Figure 1. The topmost layer is represented by Rational Rose and VISIOME scripts. The script parser is the mapping of the VISIOME scripts to the internal format of VISIOME. The script is actually run by VISIOME script engine in the bottom layer. Import and export operations actually use Rose API to transfer data to and from Rational Rose, but this has been left out of Figure 5. Another thing missing from the figure is the role of UML interface (UMLi) as the means for operations to manipulate data. We explain how this is done in the next chapter.

4. UML INTERFACE FOR MODEL PROCESSING

In this chapter we will discuss the UML interface in more detail from a programmer's point of view. We introduce the UMLi Schema that defines the UML data model, and give an overview on the interface itself together with a high-level example of using UMLi with Python.

While many of the current UML-based CASE-tools offer proprietary extensibility interfaces for accessing UML models, they usually fall short on giving sufficient support for producing general UML model manipulation operations. The tools have different interpretations of the UML metamodel and they support it only partially. Moreover, writing individual model operations would require each operation to access and interpret the CASE-tool specific data model separately. The UMLi import and export components lift this burden off the designer, thus allowing her to concentrate on the actual functionality of the operations on UML. UMLi itself provides a uniform, high-level platform for working on the UML domain.

In order to have a uniform platform, a common data model must be defined. This data model, the *UMLi Schema*, is used both by UMLi for maintaining the integrity of the UML models and verifying the validness of invoked operations, and by the programmer for understanding, navigating, and manipulating the models. The UMLi Schema conforms to the UML metamodel version 1.4 [OMG02], and extends it with view and custom data. View data makes it possible to perform general view-related operations, such as layout operations, on the models. Custom data is a mechanism for the user to embed her own proprietary non-UML data into the models. The UMLi Schema, given in a text format, can be freely modified and extended to meet the requirements of a particular domain, and to evolve with forthcoming UML versions.

A simplified example of the UMLi Schema together with a corresponding subset of the UML metamodel as a class diagram is given in Figure 7. The example describes how an interaction consists of a set of messages with sending and receiving classifiers.

```
element Interaction inherits ModelElement
{
  (1..*) message       Message
}

relation Message inherits ModelElement
{
  (1..1) sender        --> ClassifierRole
  (1..1) receiver      --> ClassifierRole
  (1..1) interaction   within [message] Interaction
}

element ClassifierRole inherits Classifier
{
  (0..1) multiplicity  Multiplicity
}
```

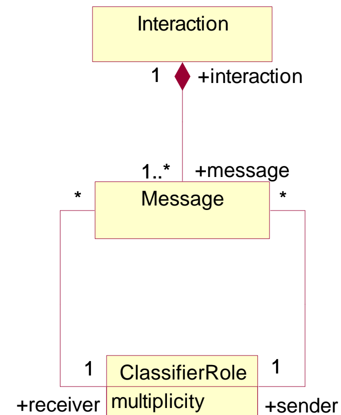


Figure 7. Example of the UMLi Schema

Every entity in the UMLi Schema is either an element or a relation. Roughly, every metaclass denoting a relationship between other metaclasses of the UML metamodel produces a relation, such as *Message* in Figure 7, while other metaclasses produce elements (e.g. *Interaction*, *ClassifierRole*). The inherited metaclasses are given in the header of each entity (e.g. *ClassifierRole* inherits the *Classifier* metaclass). Every metaclass attribute is mapped to an attribute (e.g. the *multiplicity* attribute of a *ClassifierRole*). Attribute types can either be predefined UML types, or in the case of composition relationship (e.g. *Interaction* owning a set of *Messages*), another UML metaclass. In the latter case, the part class has a "within" tag, together with the name of the composition role in square brackets. Meta-associations to other metaclasses are denoted with an arrow "-->" (e.g. *sender* and *receiver* associations of *Message*). A multiplicity is defined for every attribute inside parenthesis (e.g. (1..1) for *sender* attribute of *Message*). Note that if the lower bound of a multiplicity is zero, the attribute is not required to exist. In this case, the value is undefined. The export module can create suitable default values for attributes when required by a given CASE-tool.

The interface offers a variety of operations for manipulating, navigating, and examining the model. Each operation performs checks to ensure the requested changes do not violate schema-dictated rules and thus compromise model integrity. For example, trying to attach more than one *ClassifierRole* as a *sender* to a *Message* results in an error. The schema is also used by some operations to update one end of an association when the other end is changed. For example, moving a message under an *Interaction* would update the *interaction* attribute in the *Message*, as well as the *message* attribute of the *Interaction*.

UMLi Interface consists of *UMLiObjects* and *UMLiVectors*. UMLiObjects act as high-level wrappers for VISIOME internal data, adding operations aware of the constraints and dependencies described in the schema. UMLiVectors are collections of UMLiObjects, usually used as operands or result sets for operations.

Metaclass instances can be created, moved, copied, attached, and detached within a model, and their data and metadata manipulated freely, as long as the changes comply with the constraints. It is also possible to use UMLiObjects and UMLiVectors in OCL expressions to locate metaclass instances meeting some criteria.

As an example, consider a simple Python script implementing horizontal compression of a sequence diagram. The script searches for classifier roles with a given tagged value (a standard UML extension mechanism) and merges them together into a new classifier role. This kind of an operation can be used for abstracting a sequence diagrams showing detailed interaction of objects into a more compact higher-level view. Figure 8 shows an example of the original and compressed sequence diagrams. The classifier roles with tagged value {network} have been compressed into a new classifier role called Network.

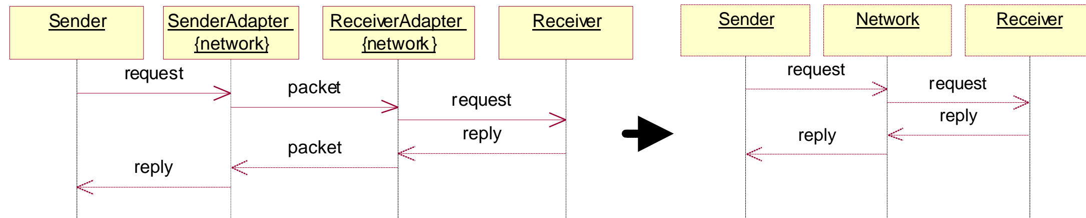


Figure 8. Sequence diagram horizontal compression example

```
def compress_horizontal( collaboration, tagged_value, compressed_name ):
    """
    The script finds the classifier roles marked with a given tagged value and
    redirects their incoming and outgoing messages to a new classifier role
    with name compressed_name.
    """
    # Gain access to UMLi services by dispatching a new UMLiObject
    u_object = win32com.client.Dispatch("UMLilib.UMLiObject")

    # Create a new classifier role and set its name
    compressed = u_object.CreateObject( "ClassifierRole" )
    compressed.SetProperty( "name", compressed_name )

    # Fetch all the messages of interactions belonging to the collaboration
    messages = collaboration.GetChildren( "interaction.messages" )

    # An OCL expression for searching the classifier roles with tagged_value
    ocl_expression = "element.type->exists(x|x='" + tagged_value + "')"

    # Iterate over the messages and redirect them when necessary
    for msg in messages.ToList():
        # Find the tagged values of the sender and the receiver
        sender_tvalues = msg.GetConnections("sender.taggedValue")
        receiver_tvalues = msg.GetConnections("receiver.taggedValue")

        # If the sender (a Classifier) of the message has the given tagged value,
        # redirect it to the compressed classifier role
        if sender_tvalues.Select(ocl_expression).Length>0:
            msg.SetConnection("sender",compressed)

        # If the receiver of the message has the given tagged value,
        # redirect it to the compressed classifier role
        if receiver_tvalues.Select(ocl_expression).Length>0:
            msg.SetConnection("receiver",compressed)

    # Remove classifier roles with the given tagged value from the collaboration.
    # These classifier roles are the ones that were compressed, and have
    # become obsolete.
    included_elements = GetChildren("ownedElement").Reject(ocl_expression)
    collaboration.SetChildren("ownedElement",included_elements)

    # Append the new compressed classifier role under the collaboration
    collaboration.AddChild("ownedElement",compressed)
    return
```

Figure 9. Example of using UMLi with Python

A Python script implementing the compression operation is given in Figure 9 with the UMLi specific operations shown in boldface. The script takes as its input parameters a *collaboration* containing a set of interactions, a name of tagged value (*tagged_value*), and the name for the new compressed classifier role to be created (*compressed_name*). The script first gains access to the UMLi services by dispatching a new UMLiObject. It then creates a new classifier role with name *compressed_name*. The names used by the UMLi interface conform to those defined in the UMLi Schema and consequently the UML metamodel.

Next, the script collects all the messages under the interactions belonging to the given *collaboration* using OCL expressions for accessing and manipulating the model data. Another OCL expression body is defined for collecting the model elements with an attribute *type* conforming to *tagged_value*. The *Messages* are iterated over and their *sender* and *receiver ClassifierRole*s are tested against the *tagged_value* tag. When found, the corresponding attribute is redirected to the newly created compressed C*lassifierRole*. Finally, the classifier roles without the given *tagged_value* are kept while the ones with the value are removed from the interaction.

This simple Python example demonstrates the reasonably high conceptual level that UMLi operates on, thus allowing the programmer to concentrate on the UML domain rather than the obscurities of any particular UML CASE-tool API.

5. RELATED WORK

Some CASE tools give only modest support on tool interoperability, limited to providing XMI import and export facilities. According to our and others' experiences, XMI is not, however, working properly for that purpose. First, the current XMI standard by OMG does not define how representational information (e.g., layouts) is to be stored in XMI. Further, the extensibility of XMI allows the tool vendors to define their own way of attaching representational information with the model data. Second, different XMI dialects developed vary from each other considerably enough to make tool interoperability impossible in most cases. Some research has been carried out to define alternative exchange formats. One interesting attempt is Graph Exchange Format (GXL), which has been developed to support interoperability among graph-based tools [Win01]. Even though several groups are involved in the development of GXL, no experiences on its applicability to serve as an exchange format among UML-based CASE tools have been reported.

Extensibility and more enhanced interoperability are supported in many current UML CASE-tools by providing an API or by implementing an interpreter for a scripting language. In the former case, the API typically conforms to UML metamodel (e.g., in TED [Wik98]) or is an extension of it. Together Control Center [Tog02], for instance, can be extended by creating Java applets that interact with the tool via its Java-based API. Rational Rose [Ros01], in turn, implements an interpreter for a BASIC dialect and provides Rose Extensibility Interface (REI) that can be used to access or import/export the models. Although supporting extensibility and customizability, such tool-dependent solutions are not well-suited for performing a chain of transactions or queries on the models.

In [Por02], Porres discusses the requirements and most important features of a stand-alone UML-aware programming environment, relying on a scripting language that can be used to manipulate and extract information from any UML model. The language used is meant to serve as a complement to UML editors, not as an alternative choice for them. The textual scripting language and the supporting environment built is used for loading, querying, modifying, and saving UML models. The actual power of the xUMLi platform, in turn, is to allow the CASE-tool user to create operations and to combine existing ones to perform more complicated (chains of) tasks. Further, while a specific scripting language is used to write the operations desired in the approach proposed by Porres, the xUMLi platform allows the usage of independent operations that are not tied with a specific programming language or a platform. As the UMLi Schema in our approach, the scripting language proposed by Porres is compliant with the UML metamodel. In [Por02], the models are stored in memory for desired processing. Also in our solution, the internal data representation of xUMLi is used to store all intermediate results of performing UML processing operations. The UML metamodel contains a series of OCL expressions, named well-formedness rules, which define validity of UML models. In the approach by Porres, the well-formedness rules must currently be translated from OCL to Python manually. The actual interoperability with existing tools is enabled by supporting XMI serialization.

Marder *et al.* propose a UML repository and an API, based on UML metamodel, for managing and querying UML models [Marder99]. The repository is implemented by exploiting an object-relational database management system (ORDBMS). OCL constraints can be used to specify and check commands on the UML models. The constraint can, for instance, hold design guidelines or semantic invariants to enforce validity of UML models. The OCL constraints are parsed and translated into an internal graph representation, which is then translated into SQL constraints used for manipulating the models in the ORDBMS database. An OCL interpreter is also used in our approach. While the database used by Marder *et al.* in [MRS99] is a persistent data storage, we propose an interface to transient data. We believe that the usage of a persistent data storage can be a heavy and slow solution, especially if constructing chains of operations is desired. Further, our implementation is not dependent on a specific persistent data storage.

Various implementations of UML are offered as libraries available for CASE-tool developers. Currently, a popular one is Novosoft UML (NSUML) [NS00] Java library, which offers an implementation of a complete UML metamodel, a reflective API, XMI support, etc. NSUML lacks OCL-like queries but it can be combined with the Dresden OCL toolkit [HDF00]. As in our UMLi Schema and in the approach proposed by Porres [Por02], some parts of NSUML are generated automatically from a UML metamodel. NSUML is used with various CASE-tools, e.g., in ArgoUML [Arg02]. ArgoUML offers OCL support as well, allowing additions of OCL constraints, for which syntax and type checks are implemented. While we aim at supporting the CASE-tool user in making enhanced manipulations of UML diagrams using a tool independent platform, NSUML is rather a UML implementation, to be used by tool developers.

6. DISCUSSION

We have presented the design of our tool-independent UML processing platform. Our motivation has been to explore advanced UML processing capabilities. Currently the platform has been integrated with Rational Rose, but we assume that the required export/import components can be fairly easily constructed for a UML tool with a reasonable API. After a tool has been integrated with the platform, all the processing facilities included in the platform are readily usable from the tool, and on the other hand new UML processing operations implemented on the platform become available for all the tools integrated with the platform.

A key feature of our architecture is its support for independently written COM-based UML processing components (implemented, say, in Python or C++) and for combining these components with high-level scripts. In this way complex UML model processing operations can be specified conveniently by the software designer, without making them overly inefficient due to the heavy repository traffic.

The xUMLi platform currently supports the major UML diagram types (class, interaction, statechart, and component diagrams). The scripting interface has been so far implemented in Rose using specially stereotyped activity diagrams, interpreted as a visual script. This nicely demonstrates the benefits of having a scripting capability presented in the UML world itself: no new scripting language (and its editor) is needed, and scripts can be conveniently stored together with the models they are manipulating.

So far our practical experiences are limited to more or less toy examples of UML models. However, our initial experiences suggest that the platform can process chains of UML processing operations significantly faster than performing separate operations in sequence directly on top of a tool API. This is due to the fact that intermediate results need not be stored back to the tool repository. The repository connection is often the bottleneck in this type of operations.

We are currently developing the platform together with an industry partner, aiming at a general software architect's workbench. What kind of operations are actually needed in such a workbench is largely an open question. However, our platform allows us to easily experiment with various kinds of operations that are assumed to be helpful for a software architect, and combine them with the scripting mechanism. We hope that the industry case studies give us new insight on the software architecting process in general and guide us in finding useful UML processing facilities.

Acknowledgements
This work is funded by the Academy of Finland and by Finnish National Technology Agency.

References

[Arg02] Project Home Page, http://argouml.tigris.org/, 2002.

[BMF02] Bennett, S., McRobb, S., and Farmer, R., Object-Oriented Systems Analysis And Design Using UML, 2nd edition, McGraw-Hill, 2002.

[Cov01] Cover R.: XML Metadata Interchange (XMI). http://www.oasis-open.org/cover/xmi.html. November 2001.

[HDF00] Hussmann H., Demuth B., and Finger F.. Modular architecture for a toolset supporting OCL. In Andy Evans, Stuart Kent, and Bran Selic, eds., UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Vol 1939 of LNCS, pp. 278-293. Springer, 2000.

[Kos01] Koskinen J., Peltonen J., Selonen P., Systä T., Koskimies K.: Model processing tools in UML. Proc. ICSE 2001, Toronto, May 2001, 819-820 (Formal research demo).

[MRS99] Marder U., Ritter N., Steiert H-P., A DBMS-based approach for automatic checking of OCL constraints.

[Mic02] Microsoft: http://www.microsoft.com/com/tech/com.asp, 2002.

[MS01] Mäkinen E. and Systä T.: MAS - An Interactive Synthesizer to Support Behavioral Modeling in UML. In Proc of ICSE 2001, Toronto, May 2001, 15-24.

[NS00] Novosoft. Novosoft metadata framework and UML library. Available at nsuml.sourceforge.net/.

[OMG02] Object Management Group, http://www.omg.org/uml/, 2002

[Pel00] Peltonen J.: Visual Scripting for UML-Based Tools. In: Proceedings of ICSSEA 2000: Paris, France, December 2000.

[PS01] Peltonen J., Selonen P.: Processing UML Models with Visual Scripts. In Proc. of HCC'01, Stresa, Italy, 2001. 264-271.

[Por02] Porres I., A Toolkit for Manipulating UML Models, TUCS Technical Report No 441, University of Turku, January 2002.

[Ros01] Rational Software Corporation, Rational Rose, http://www.rational.com, 2001.

[SKS01] Selonen P., Koskimies K., Sakkinen M.: How to Make Apples from Oranges in UML. In Proc. of HICCS 34, IEEE Computer Society 2001.

[SSK01] Selonen P., Systä T., Koskimies K.: Generating Structured Implementation Schemes from UML Sequence Diagrams. In Proc. of TOOLS USA 2001, Santa Barbara, California, 2001. 317-328.

[Tog02] TogetherSoft Corporation, Together Control Center, http://www.togethersoft.com/, 2002.

[Wik98] Wikman J., Evolution of a distributed repository-based architecture, Dept. of Software Engineering and Computer Science, Research Report 1998:14, Blenkinge Institute of Technology, Sweden. Electoric Proceeding of the First Nordic Software Architecutre Workshop NOSA'98, http://www.hk-r.se/fou/forskinfo.nsf/.

[Win01] Winter A., Exchanging Graphs with GXL, in *Graph Drawing – 9th International Symposium,* Vienna, Austria, September 2001.