

An Approach and a Platform for Building UML Model Processing Tools

Jari Peltonen and Petri Selonen

Tampere University of Technology, Institute of Software Systems,
P.O.Box 553, FIN-33101 Tampere, Finland
{jari.peltonen, pertri.selonen}@tut.fi

Abstract

In this paper we present a model processing approach and an accompanying platform for building software engineering tools. The approach is based on small model processing tasks that can be combined to gain higher level functionality, and eventually complete tools. The model processing platform can be seen as middleware that can be integrated with any other tool having an appropriate application programming interface or data transform file format like XMI. As an example of applying the platform, we introduce a tool for validating software architectures against UML architectural profiles, embodying domain, product-line, or platform specific conventions and constraints. The tool and the platform have been used as a part of a maintenance process of real-life mobile terminal software.

1. Introduction

Unified Modeling Language (UML) is an industry standard for visualizing, specifying, constructing and documenting the artifacts of software-intensive systems [12]. UML models are presented, constructed, and manipulated as diagrams that can be used to view a system from different perspectives, on different levels of abstraction, and at different stages of the design process. Hence, the various UML models are dependent and overlapping.

UML is a standard for a modeling language, not for a design method. This gives considerable freedom for implementing arbitrary method support. Consequently, the dependencies between the models are not static: there are different methods for e.g. different organizations, product families, and projects. The interpretation of the UML diagrams and their inter-relationships is specified individually for each context. This calls for strong semantic tool support for each context using UML.

Object-oriented software development can be seen as a series of modeling assignments. Besides automation of development tasks, issues like model comprehension and validity should be addressed. There are several

opportunities for tool support: model comprehension can be supported by generating more abstract, or otherwise different views of the model, validation can be supported by checking a model against predefined rules, profiles or other models, and a new diagram can be generated automatically by combining other diagrams. All of these operations can be seen as tasks for querying or processing models.

Since system developers want to use UML for different purposes, and the methods that best fit their purposes, there should be means to provide customizable tool support for UML. This paper presents one such approach, together with the implementation of the accompanied model processing environment. The application of the approach and the environment is also illustrated by a concrete model processing tool built for validating software architectures against UML architectural profiles. The tool has been used as a part of a maintenance process of a large-scale real-life mobile terminal system.

2. The Model Processing Approach

The main goal of our research is to provide automated tool support for various aspects in software engineering, each introducing a set of model processing tasks. Instead of creating several high-level tools for different tasks, we create smaller *model operations* and build higher level functionality by combining them. We refer to the usage of these operations with the term *model processing*.

2.1 UML Model Operations

UML model operations can be used for various purposes during software development. These purposes include checking, abstracting, merging, slicing, and synthesizing UML models [10]. A fundamental example of a UML model operation is searching for, and filtering of, information in given UML models. When accompanied with a mechanism for evaluating and enforcing constraints, the operations can provide side-effect free checking and validation of models. An obvious

example is enforcing the standard UML well-formedness rules [12]. To add expressiveness, though, model operations usually modify existing diagrams or produce new UML diagrams based on them.

Model operations include *transformation operations* [18] that take a UML diagram as an input operand and produce a diagram of another type as the result, *set operations* [17] that apply set theoretical operations for two diagrams of the same type, *projection operations* [3] that take a UML diagram and yield a view of that diagram with respect to some criteria, *conformance operations* [19] that together with UML profiles define and enforce domain-specific constraints and conventions, and *visual operations* that involve mechanisms for coloring and arranging UML diagrams.

Expected benefits of using the model operations include faster creation of models through synthesis and merging, improved quality of models following increased understanding of them through generation of transient views, enhanced tool support and customizability, and support for incremental and iterative model development.

2.2 Combining Model Operations

The model operations are combined together for more complex model processing functionality with *VISIOME* (Visual Scripting In Object Modeling Environments) [13], a high-level visual programming language. The language emphasizes the usage of the Object Constraint Language (OCL) with a set of fundamental programming constructs, and its notation resembles the UML activity diagram notation. The language is dynamically typed. The scripts are built using *objects* as sources of data and *activities* that act on the data. The elements are put together with *object flows* and *control flows*. The activities can be executables like components (e.g. COM, CORBA or Java beans) or other scripts.

An example script, "Paint Difference" is shown in Figure 1. The script takes as its input operands two or more diagrams. Taken the asymmetric nature of the described operation, the reference input is marked with '1'. The script makes a *decision*, marked as a diamond with *guard conditions*, based on the diagram type and converts all input diagrams class diagrams. Similarly to UML activity diagrams, decisions indicate alternative paths of control. *Synchronization* is marked with a horizontal line. In the Paint Difference script, all concurrent branches have to be completed before the difference operation is applied to the diagrams. The differences to the original diagram are highlighted in the resulting diagram.

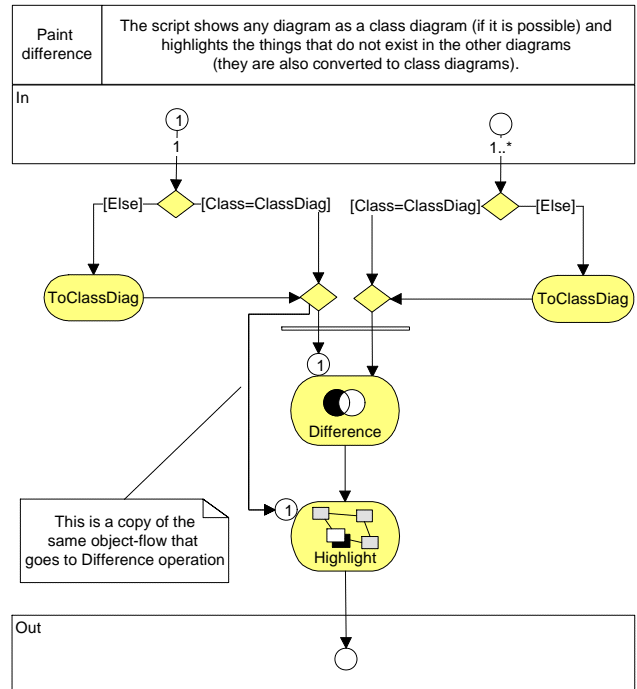


Figure 1. Paint Difference script

3. The Model Processing Mechanisms

In order to support model processing in practice, feasible mechanisms are needed for implementing UML domain specific model operations, for combining the operations together, and accessing to commercial UML CASE-tools.

3.1 Visiome Implementation

We have an implementation of the *VISIOME* interpreter for Windows platforms. In this implementation the executables used as *VISIOME* activities are COM (Common Object Model) [11] components that can use the *VISIOME* application programming interface (*VISIOME* API) to query and manipulate the data provided for them. In addition, *VISIOME* includes an OCL interpreter whose services are also offered through the API. The interpreter and the classes in the data model are implemented as COM automation classes. Instances of these classes can be accessed from programming languages that support COM as if the instances were native objects of the language.

Although *VISIOME* was designed for object modeling environments – especially UML tools – in mind, the implementation of *VISIOME* is domain independent. *VISIOME* does not make assumptions on the manipulated data. Thus, the API does not include any functionality for tasks like integrity checking. If needed, a specialization

layer with a domain dependent data model has to be provided.

Originally, we planned to use any UML compliant CASE-tool as the VISIOME editor. Prototype implementations for Nokia TED [22] and Rational Rose [Rat03] suggested, however, that the VISIOME editor required capabilities beyond those provided by commercial UML tools. Therefore, a separate editor for VISIOME has been implemented.

3.2 Implementing Model Operations for UML Domain

In order to get UML awareness to VISIOME, we have implemented a specialization layer for UML providing a UML metamodel compliant data model, and a higher level API for UML model processing. Since they are also implemented as COM automation classes, a person implementing operations can use both the UML API and the lower level VISIOME API with its OCL interpreter, in the same way, and at the same time.

Typically, the individual model operations are constructed using Python and then combined together with VISIOME to form scripts of more complex functionality. We have also used C++ and Java to implement model operations. Other languages with COM support include Perl and Visual Basic.

As an example of using the UML interface, assume that an architecture model is only allowed to contain classifiers representing valid domain-specific concepts. Further, assume that these concepts are described in UML as domain-specific stereotypes, defined by a conceptual profile. This checking of stereotype conformance [19] can be expressed as follows (taken from [20]):

```
st = profile.find( \
    "element.oclAsType(Class).clientDependency->exists(cd |" \
    "cd.stereotype.name->includes('stereotype')" \
    "and cd.supplier.stereotype.name->include('metaclass'))")
```

```
for cls in model.find("element.oclIsKindOf('Class')"):
    for cst in cls.stereotype:
        if len(st.select("element.name = " + cst.name + ""))==0:
            # handle class with wrong stereotype
```

The first part of the operation queries for all properly defined stereotypes in the stereotype definition profiles ([12, Fig. 4-1], also see left-hand side of Figure 3 in Section 4.1), while the latter iterates over all classes in the view and checks whether they have a valid stereotype.

The UML interface offers relatively high-level support for Python and other COM-compliant programming languages by providing access to the UML models through getters, setters, and OCL-based queries, aiming at

allowing the user to concentrate on the UML-related problem at hand on a high conceptual level.

3.3 xUMLi – The Model Processing Platform

xUMLi is a software platform for UML model processing [1] consisting of VISIOME, the UML specialization layer, an a set of standard components. Standard components are mainly meant for importing and exporting models between *xUMLi* and other tools, but also some of the model operations, as well as general operations for selecting and showing data can be seen as standard components. Besides of the standard components, a user of *xUMLi* can use self made components with arbitrary functionality, making it possible to e.g. support context specific interaction between the final user and the tool.

Figure 2 shows the high-level architecture of *xUMLi*. In general terms, *xUMLi* can be seen as a middleware meant for model processing purposes. The environment is not dependent of any specific CASE-tool, but offers a plug-in interface for components that transfer models between a tool repository and the data model. It is therefore possible to support several different CASE-tools or UML model repositories. We have built such import/export plug-ins for Rational Rose, XMI, and some proprietary file formats.

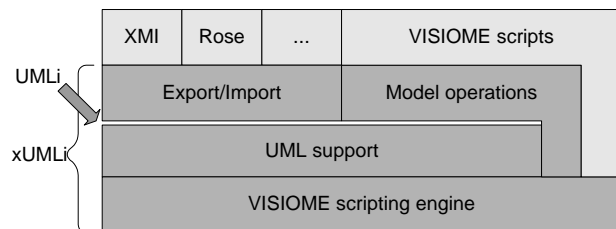


Figure 2. The Architecture of *xUMLi*

4. Exploiting the Platform: A Tool for Validating Software Architectures

The model processing approach, together with the platform, is ultimately meant for building software engineering tools. As a concrete example of tool implementation, *artDECO* is used for validating architecture views against architecture profiles, expressed in UML. The tool has been developed as part of an ongoing joint research project with Nokia Research Center targeting the architecture validation of mobile terminal systems.

4.1 Overview and Motivation

With its well-established position in software engineering, UML is a credible choice to act also as an architecture description language. As a modeling language, UML lacks methodological support for describing software architectures. artDECO provides tool support for defining domain, product line, and platform specific architectural conventions for guiding the designers' work.

To meet this purpose, a technique exploiting UML *architectural profiles* [19] is proposed. In this approach, architectural profiles play a key role in materializing the work context of software architects. They can be seen as visual shortcuts, or intuitive examples, of what kinds of structures the architectural style in question allows. The profiles are in turn interpreted as standard UML profiles defining new architectural concepts and their inter-relationships.

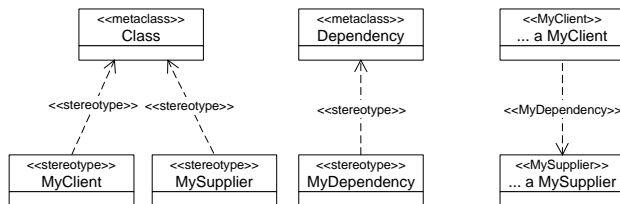


Figure 3. Example of an architectural profile

An example of an architectural profile is shown in Figure 3. The left-hand side of the figure, a stereotype definition part of the profile, introduces three new user-defined architectural types, derived from the standard UML Class and Dependency metaclasses: MyClient, MySupplier, and MyDependency. The right-hand side of the figure, a constraint part, shows a new constraint imposed on the instances of MyClient and MySupplier: there can only be a dependency of type MyDependency from a MyClient to a MySupplier. In addition, only the defined stereotypes are allowed to be used in the architectural views.

Since the conventions and requirements vary from domain to domain, there can hardly exist a general and complete set of architectural profiles. Instead, a set of *conformance rules* gives the interpretation for the profiles. The profiles and the operations implementing the rules, together effectively define an architectural language to which the architectural views must conform to.

While the example usage scenario to be given later in Section 4.3 will concentrate on a reverse engineered model, the profile-centric approach can be applied both on forward engineering and reverse engineering models. With the former, when integrated into a suitable software engineering process, the technique allows the architects to freely experiment with ideas while still providing the

architectural profiles as a means for embodying the necessary conventions, rules, and restrictions. When a model is completed, it can be validated against the profiles and remedied if necessary.

4.2 The Tool Implementation

artDECO is strictly speaking not a single tool but a set of conformance operations, represented by xUMLi Python components, and domain specific VISIOME scripts describing *configurations* of the conformance operations. An artDECO script imports the architectural profiles and views, and then runs the conformance operations one by one, each validating the views against the profiles. The resulting incidents are then reported to the user, either in the form of an XML error report, or by providing a visual browser that can be used with the selected UML CASE tool. Conceptually, the architectural profiles can be viewed as standard UML profiles imposed on the views in a *posteriori* fashion.

Architectural profiles contain a *conceptual profile* and a set of *view profiles*. The conceptual profile defines the fundamental concepts that are used throughout the entire architecture design and description. It also specifies the architectural style and the validation rules with class diagrams, e.g. by defining component types and different logical dependency types, describing the allowed relationships between the fundamental architectural concepts, the interfaces, their realizers, and the classifiers that depend on them.

The components specific to the artDECO tool – the conformance operations – typically query the profile model for relevant structures, then iterate over the views and checks whether or not the particular constraints hold. As an example, consider the relationship conformance rule stated in [19]:

Every relationship (i.e. association, dependency) in a view is required to have a corresponding relationship in a profile belonging to the profile hierarchy.

This rule, together with the profile given in Figure 3, effectively implies the following constraint imposed on all instances of MyDependency:

```
context MyDependency inv:
  self.client->forAll(c |
    c.stereotype.name->includes('MyClient') ) and
  self.supplier->forAll(s |
    s.stereotype.name->includes('MySupplier') )
```

The invariant states that for each MyDependency instance, all its client elements must have stereotype MyClient and all its supplier elements must have stereotype MySupplier. A generic version (i.e., not tied to

a fixed context) of this invariant can be enforced using Python and xUMLi interface as follows:

```
for d in view.find( "element.oclIsKindOf(Dependency)" ):
  pd = profile.find( \
    "element.oclIsKindOf(Dependency) and " \
    "element.stereotype.name->includes("'" + \
    d.stereotype.name + "'" ) and " \
    "element.client.stereotype.name->includes("'" + \
    d.client.stereotype.name + "'" ) and " \
    "element.supplier.stereotype.name->includes("'" + \
    d.supplier.stereotype.name + "'" )" )

  if pd.Length==0:
    # report non-conformant dependency
```

The example iterates over all dependencies in the given architectural model and checks whether or not there exists a corresponding dependency in the profiles (i.e., with the same stereotype, and with supplier and client classifiers having the same stereotypes). For simplicity, the example assumes binary dependencies with single stereotypes.

Figure 4 shows an example of an artDECO configuration opened in the VISIOME editor. The script has two inputs, one for architectural profiles and one for architectural views. Both models are imported from a selected Rational Rose repository by a xUMLi component RoseImporter. The architectural views are filtered by the ISADepFilter component (i.e., unwanted dependency types are removed). The first synchronization bar ensures that both the profiles and views are ready before they are forwarded to the two conformance operations, Stereotype Conformance and Relationship Conformance. After both operations have worked on the input profiles and views (ensured by the second synchronization bar), the incident data is forwarded to the Error To XML component that produces a summary incident report.

4.3 Using the Tool for Architecture Validation

artDECO and xUMLi have been used as a part of a maintenance process targeting a large-scale product platform architecture and products built on top of this platform. The target system, hereafter referred to as ISA, consists of product platform architecture and products built on top of it. The domain is embedded mobile terminal system. The target architecture model has roughly 150 subsystems, 1000 components, and 15000 dependencies.

The maintenance process relies on two subprocesses: a reverse engineering process and a model analysis process. The former is guided by the profile descriptions constructed for the ISA product platform, gradually abstracting the implementation-level artifacts and mapping them to the high-level concepts, resulting in new UML architecture views. The latter utilizes artDECO for validating architectural views against the profiles to indicate whether the given architectural rules have been followed when maintaining and further developing the subject system. During the evolution of the software architecture, the procedure is iterated for individual releases of the subject system. The overall maintenance process is described in more detail in [16].

Table 1 gives examples of the architectural concerns, together with their UML interpretations, that can be used for automatic validation in the artDECO tool. The first two items reflect the rules shown in Figure 3. The last one describes the most interesting mechanism: validation of architecture styles, expressed as profiles.

An artDECO validation script imports the profiles and the views (from Rational Rose), performs a set of required model transformations on them e.g. by filtering of unwanted modeling elements, performs a set of conformance operations, and finally outputs the results

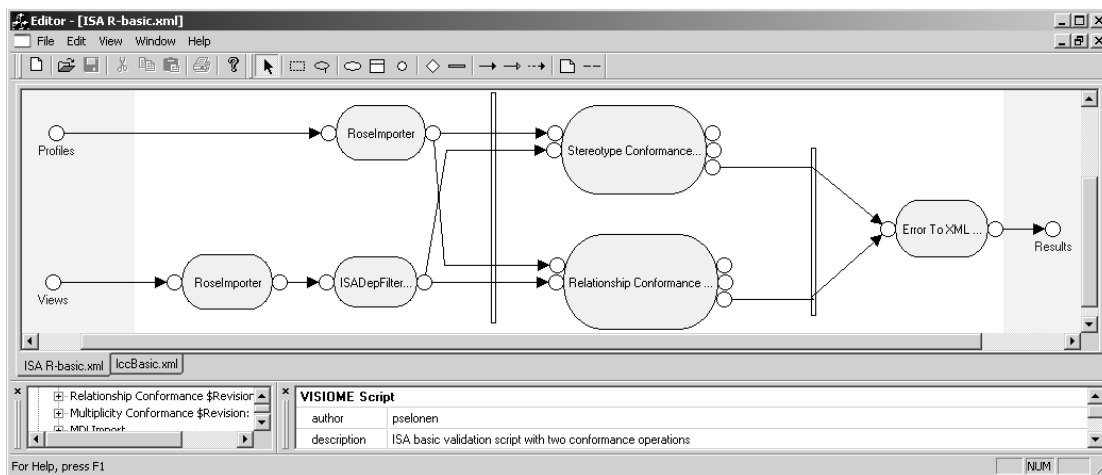


Figure 4. Example artDECO VISIOME configuration

from the validation in XML format. Alternatively, the user can choose to browse the reported incidents with an error browser (as shown in Figure 5). The error browser is implemented as a xUMLi component that filters out the incident data and selects them in Rational Rose. The component is an example of a xUMLi-compliant component dedicated to user interaction.

Table 1. Examples of ISA architectural concerns and their respective UML interpretations

Architectural concern	UML interpretation
Validate that only the architectural concepts allowed by the domain are being used	Check that all classes, packages, and dependencies have stereotypes defined in the architecture profiles
Validate that only allowed relationships are being used	Check that all used dependencies have been defined in the architecture profiles
Validate that the system conforms to a three-layer architecture style	Check that all the dependencies between classes belonging layers defined in the architecture profiles (realized as namespace hierarchies) are directed from a higher-level layer to a lower-level layer

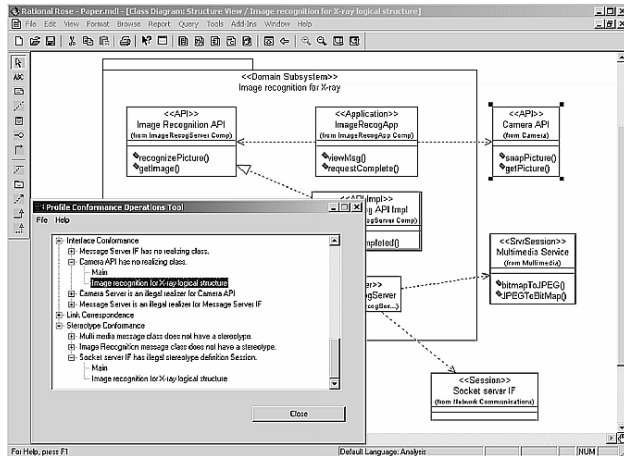


Figure 5. artDECO error browser screenshot [19, Fig. 16]

artDECO has been successfully applied on a series of reverse engineered ISA release architecture models. Taken the inherent characteristics of the models into account, the need for tool support becomes evident: to analyze and validate such large-scale models manually would be error-prone and tedious at best, and taken the existing tool support into account, virtually impossible at times.

5. Related Work

Many current UML CASE-tools, both commercial (e.g. Rational Rose [15], Together [21], Poseidon [6]) and non-commercial (e.g. ArgoUML [2]) offer extensibility and interoperability capabilities, for example, by providing a proprietary API for model repository access, by introducing a scripting language, or by providing libraries for tool developers. However, these CASE-tool dependent solutions are not generally well-suited for performing a chain of transactions or queries on the models. One of the main goals of xUMLi is to support the combining of small model operations to achieve higher-level functionality, customizable for a given process, domain, or a platform.

Of the existing UML model processing platforms, the System Modeling Workbench (SMW) [14], the UML all purpose transformer (UMLAUT) [7], the OCL4Java library of the Kent Modeling Framework (KMF) [9], and FUJABA [5] come close to our approach. In comparison, xUMLi supports COM automation and is thus not restricted to any particular programming language, allowing the development of practically any kind of components (e.g. user interaction, connection to an external tool). In addition, it involves a dedicated OCL interpreter for querying the models. The platform does not involve a CASE tool itself, but is intended to be integrated with arbitrary CASE tools.

Effort has been put on mapping UML and architecture description languages (ADLs) (e.g. Zarras *et al.* [23], Hudaib and Motangero [8]). Egyed and Medvidovic [4] discuss mapping ADL specifications into UML models and defines conformance and consistency relationships, but between UML models, not between profiles and models. The artDECO tool does not aim at defining a single profile, but focuses on giving support for configurable validation processes that can be customized to support the conventions of a given product line or domain.

6. Discussion

There is a growing need for building highly customizable and dynamic tool support to meet the requirements of a particular process, domain, product line, or product platform. We argue that the model processing approach presented in this paper places the necessary emphasis on facilitating tool support, and xUMLi as a customizable platform aims at providing such support. The platform is being actively developed and a new major release is expected during summer 2004. Likewise, the model processing components are being studied and developed further. In addition, artDECO configuration will be developed and taken into use on an industrial software system analogous to the ISA example given in

this paper. The strong co-operation with both the industry and academia has been a key driver in the research and development process. The experiences gained while applying the approach on large-scale industrial software systems have been promising.

Acknowledgements

The authors wish to thank Jianli Xu, Claudio Riva and Yaojin Yang at Nokia Research Center for their work on architectural profiles and ISA models, Jani Airaksinen, Johannes Koskinen, Mika Maununmaa, Mika Siikarla, and Sami Tolvanen for their efforts on implementing the platform and artDeco, and Kai Koskimies and Tarja Systä for their valuable support for the work as well as the comments for this paper. This research has been financially supported by Nokia and the Academy of Finland (project 51528).

References

- [1] J. Airaksinen, K. Koskimies, J. Koskinen, J. Peltonen, P. Selonen, M. Siikarla, and T. Systä, "xUMLi: Towards a Tool-Independent UML Processing Platform", In K. Østerbye (Ed.), *Proceedings of the Nordic Workshop on Software Development Tools and Techniques*, Copenhagen, Denmark, August 2002, pp. 1-15.
- [2] ArgoUML Project Home Page, 2002. On-line at <http://argouml.tigris.org/>
- [3] F. Dósa Rácz, and K. Koskimies, "Tool-Supported Compression of UML Class Diagrams", In *Proceedings of the Second International Conference on the Unified Modeling Language, UML'99*, Springer, Fort Collins, Colorado, USA, October 1999, pp. 172-187.
- [4] A. Egyed, and N. Medvidovic, "Extending Architectural Representation in UML with View Integration", In *Proceedings of the Second International Conference on the Unified Modeling Language, UML'99*, Springer, Fort Collins, CO, USA, October 1999, pp. 2-16.
- [5] FUJABA Tool Suite Developer Team, University of Paderborn, 2004. On-line at <http://www.fujaba.de/>
- [6] Gentleware AG, Poseidon, 2004. On-line at <http://www.gentleware.com/>
- [7] W.-M. Ho, F. Pennaneac'h, and N. Plouzeau, "UMLAUT: A Framework for Weaving UML-based Aspect-Oriented Designs", In *Proceedings of 33rd International Conference on Technology of Object-Oriented Languages*, Mont. St. Michel & St. Malo, France, June, 2000, pp. 324-334.
- [8] A. Hudaib, and C. Montagero, "A UML Profile to Support the Formal Presentation of Software Architecture", In *Proceedings of COMPSAC 2002*, IEEE Computer Society, Oxford, England, August 2002, pp. 217-223.
- [9] Kent Modelling Framework home page, 2004. On-line at <http://www.cs.kent.ac.uk/projects/kmf/index.html>
- [10] J. Koskinen, J. Peltonen, P. Selonen, T. Systä, and K. Koskimies, "Towards tool assisted UML development environments", In *7th Symposium on Programming Language and Software Tools*, Szeged, Hungary, June 2001.
- [11] Microsoft Corporation, "Microsoft COM Technologies – Information and Resources for the Component Object Model-based technologies", 2004. On-line at <http://www.microsoft.com/com>
- [12] Object Management Group, "OMG Unified Modeling Language Specification", version 1.5, 2003. On-line at <http://www.omg.org/uml>
- [13] J. Peltonen, "Visual Scripting for UML-Based Tools", In *Proceedings of ICSSEA 2000*, Paris, France, December 2000.
- [14] I. Porres, "A Toolkit for Manipulating UML Models", In *TUCS Technical Report No. 441*, Turku, Finland, 2002.
- [15] Rational Software Corporation, Rational Rose, 2004. On-line at <http://www.rational.com>
- [16] C. Riva, P. Selonen, T. Systä, and J. Xu, "Combining Top-down and Bottom-up Approaches for UML-based Software Architecture Maintenance", unpublished manuscript, March 2004. Submitted.
- [17] P. Selonen, "Set Operations for Unified Modeling Language", In *Proceedings of the Eight Symposium on Programming Languages and Tools, SPLST'2003*, Kuopio, Finland, June 2003, pp. 70-81.
- [18] P. Selonen, K. Koskimies, and M. Sakkinen, "Transformations Between UML Diagrams", *Journal of Database Management*, 14(3), Idea Group Publishing, 2003, pp. 37-55.
- [19] P. Selonen, and J. Xu, "Validating UML Models Against Architectural Profiles", In *Proceedings of ESEC 2003*, Helsinki, Finland, September 2003, pp. 58-67.
- [20] M. Siikarla, J. Peltonen, and P. Selonen, "Combining OCL and Programming Languages for UML Model Processing", In *Electric Notes in Theoretical Computer Science (ENTCS) dedicated to the UML 2003 workshops*, Elsevier publishing, San Fransisco, CA, USA, October 2003. Accepted for publication.
- [21] TogetherSoft Corporation, Together Control Center, 2002. On-line at <http://www.togethersoft.com/>
- [22] J. Wikman, "Evolution of a distributed repository-based architecture", In *Electric Proceeding of the First Nordic Software Architecture Workshop NOSA'98*, Sweden, 1998. On-line at <http://www.hk-r.se/fou/forskinfor/nsf/>
- [23] A. Zarras, V. Issarny, C. Kloukinas, and V. K. Kgyuen, "Towards a Base UML Profile for Architecture Description", In *Proceedings of ICSE 2001 Workshop on Describing Software Architecture with UML*, IEEE Computer Society, Toronto, Ontario, Canada, May 2001, pp. 22-26.