

Extending OpenTool/UML Using Metamodeling: An Aspect Oriented Programming Case Study

Jean Marie Lions, Didier Simoneau, Gilles Pitette, Imed Moussa

TNI-Valiosys, France
Technopole Brest-Iroise BP 70801
F – 29608 BREST CEDEX

Email: {jean-marie.lions, didier.simoneau, gilles.pitette, imed.moussa}@tni-valiosys.com

Abstract: Unified Modeling Language (UML) is currently being used as the universal technique for modeling object-oriented applications across a wide range of domains. Developing a truly adequate uniform modeling technique in the face of diverse domains requirements and activities can be challenging. Recently, many adaptations and extensions to UML have been made to reflect a domain's world view. As a technique, domain-specific UML metamodeling has gained in importance, e.g., in the context of code generation. However, the support for metamodeling within UML is weak and while many CASE tools do support manifold extensions they do not offer inherent metamodeling. We propose in this paper a pragmatic presentation of how to extend OpenTool/UML using the tool's built-in metamodeling capabilities to support Aspect Oriented Programming (AOP) in UML. Using a full metalanguage, called OTScript, we will show how the AOP concepts can be added and modified in a fully integrated UML environment, supporting consistent viewing at both the semantic model level and the user interface level.

Keywords: UML, MOF, Metamodeling, OpenTool, OTScript, CASE Tools.

1 - Introduction

In today's global market, the ability to manage and exploit knowledge, provides a very important competitive advantage. UML, as defined by the Object Management Group (OMG) [1, 2] may be seen as a means for to capturing, communicating, and leveraging knowledge within an organization, and thus for obtaining such an advantage. UML is an evolutionary general-purpose, standardized, and tool-supported, modeling language for specifying, visualizing, constructing, and documenting the artifacts in a system intensive process. UML is applicable to different types of system, domain, method, and process. It enables an object oriented and component based development process that is use-case-driven, architecture-centric, iterative and incremental. Therefore UML fundamentally supports and promotes industry engineering best practices. As a compromise between the requirements for a standard notation and for domain-specific modeling languages, the UML was designed as an extensible modeling language. In this way, UML users would be able to tailor the language to their specific requirements by introducing domain-specific model elements. On the other hand, such extensions would be performed in a way that conforms with the UML standard.

With the advent of UML, the consensus on a common notation helps designers, as well as tool vendors, to concentrate on issues much more relevant to them than whether to represent classes as rectangles or clouds, or the direction of arrows when visualizing a relationship. While many of the early CASE tools tried to cover the whole development process, practice has shown that such a generic approach has trouble competing with a series of individual specialized tools. Consequently, CASE tools are becoming more and more open, permitting developers to assemble their favorite development environment from different tools, purchased from different vendors, but yet co-operating via a single interoperability standard [5]. The OMG has anticipated this tool interoperability evolution by encouraging and adopting the Meta Object Facility (MOF) [4]. The goal of the MOF is "to provide the specification of a rich semantics to enable two systems or applications to meaningfully share information. This goal is achieved by providing domain-specific metamodels that conform to the MOF metamodeling architecture".

In this paper, we present a new technology called OpenTool/UML [7] which is usable to realize CASE tools. This technology is based mainly on metamodeling techniques and presents many powerful features. We use our OTScript language to easily extend UML properties and structures to match a specific domain. This extension approach is illustrated through the introduction of AOP concepts in UML

The remainder of this paper is organized as follows. Section 2 describes the four layers of the OMG metamodeling architecture. Section 3 presents an overview of TNI-Valiosys' OpenTool and gives some hints on the OTScript language. Section 4 describes the extension mechanism in UML metamodel. The extension process and results with OpenTool/UML are illustrated through an example in Section 5. Finally we conclude and discuss future perspectives with the arrival of UML 2.0 in Section 6.

Note: It should be noticed that the aim of this paper is not to discuss how best to introduce AOP concepts in UML. The intent is rather, given an appropriate extension schema to support AOP in UML, to show how to provide support to such an extension in a tool. In this context, the approach to AOP in UML proposed in [10] is only used as an example.

2 – UML metamodel hierarchy

It is important to understand the architecture that the OMG has chosen for its modeling standards. The OMG architecture has four layers, called M0, M1, M2 and M3 [3, 6]. Figure 1 shows these levels.

M0: user object layer is where the actual runtime objects reside. This layer is composed of the information that we wish to describe. This information is typically referred to as “data.” e.g. instances in a running program. This level of abstraction is used to formalize specific expressions regarding a given subject.

M1: user model layer is the layer where the UML models, developed by UML modelers, live. This is the level at which modeling of problems, solutions, or systems occur. The user model describes an information domain. This model is typically a UML model describing the structure of the M0 layer.

M2: metamodel layer is the level where the UML metamodel is defined. The concepts used by a UML modeler, such as Class, Attribute, Message, etc., are defined at this level. It also includes concepts from the object-oriented and component-oriented paradigms that constitute the UML.

M3: meta-metamodel layer is the top level and is an instance of itself. It consists of the most basic elements on which the UML is based, e.g. the concept of a Thing, representing anything that may be defined. It also defines the language for specifying metamodels. This layer is called MOF in the OMG terminology.

An instance at a certain level is always an instance of something defined at one level higher. An actual runtime object at M0 is an instance of a class defined at M1. The classes defined in UML models at M1 are instances of the Class concept defined at M2. The UML metamodel itself is an instance of M3. Other meta-models that define other modeling languages are also instances of M3 [6].

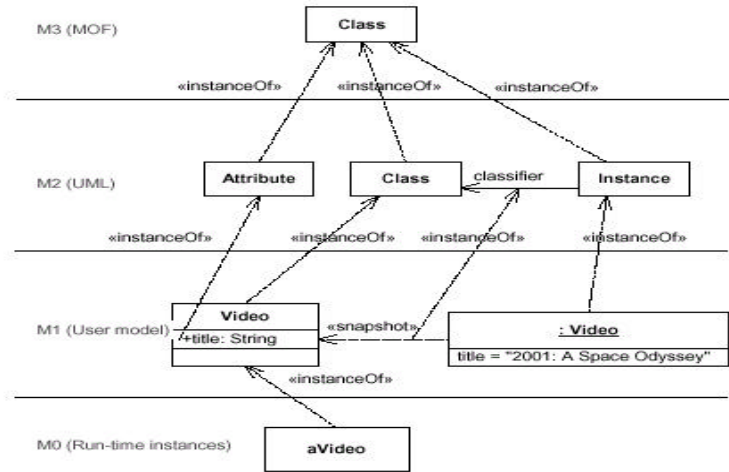


Figure 1: UML Metamodel Layer Architecture

3 - OpenTool: a technology based on metamodeling

OpenTool provides support for M3 to M1 layers. At M3 layer, it provides a full meta-metalanguage, called OTScript, which is going to be explained in the next subsection. At M2 layer, OpenTool/UML provides full implementation of the UML meta-model. At M1 layer, OpenTool/UML offers the tools to create and edit user UML models. OpenTool can be viewed as a generic Virtual Machine (VM) that has the capability to execute a M2 level meta-model specified using OTScript, which is the M3 layer language.

Figure 2 depicts the OpenTool VM environment. At start-up time, OpenTool reads a `.mn` file (mn for main) specifying the names of the OTScript modules to be loaded. For each module, OpenTool looks for a `module.ml` file (ml for model) and/or a `module.br` file (br for behavior). Model files contain fragments of metamodel descriptions. Behavior files contain methods and user interface description operating on the metamodel.

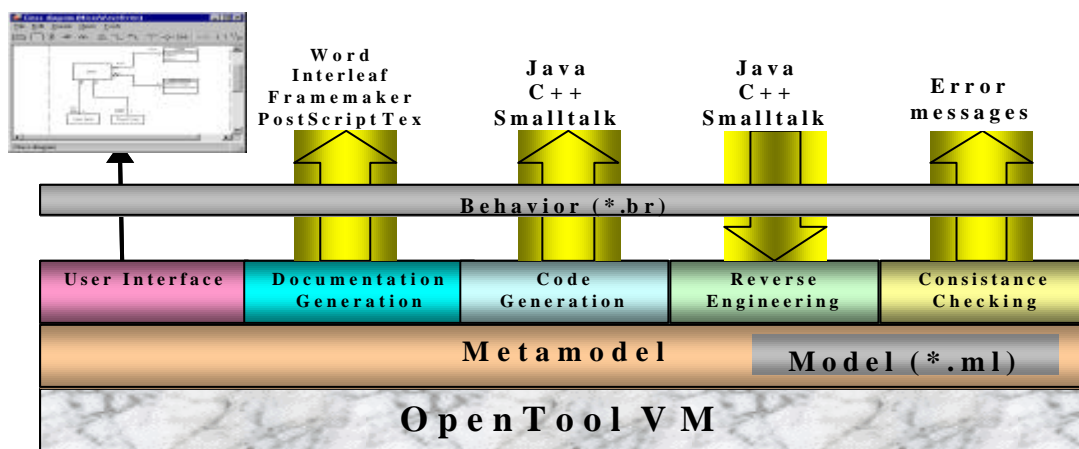


Figure 2: OpenTool VM Environment

OTScript language

OTScript is a high level language, with list manipulation capabilities, native handling of binary associations, very concise syntax, rule-checker and user interface description language.

OTScript is a class-based object-oriented language. Classes are shared descriptions of structure and the behavior of objects. Every object belongs to a class and each class is a refinement of a root class called `Any_`.

Beyond its classical OO aspects, OTScript is also list-oriented: data items are held by lists, which are kinds of collections. Facilities are provided by the language to avoid hard list manipulations: for example, it is possible to iterate upon them or to apply a function on each element of them without involving the use of classical nested loop structures.

OTScript is a strongly typed language. This means that the type of each expression must be known at compile-time. Some predefined types are provided by the language. Objects are distinct entities characterized by their own internal state containing attributes and/or roles, and their class. An attribute is valued by a primitive type such as a Boolean, an Integer, a String or an enumerated type, while a role is a part of an association, which is a bi-directional reference between two entities of the language. Associations are guaranteed to be consistent. A class is a set of objects (its instances) that share the same behavior. Behavior is stored into the classes, which also provide information to create new instances.

OTScript allows multiple inheritance. Subclasses inherit any attributes and any behavior defined in the superclass. A subclass may also define additional attributes and behaviors. Interactions with objects are performed by sending them a message *à la* Java, i.e. by using a dot notation. When an object receives a message, a method associated with this message is executed. The result of the execution is returned as an answer of the object to the message.

Development environment

The OpenTool technology provides tools helping the development of OTScript code:

- *UML metamodeler*: is used to describe in UML the metamodel., and provides automatic generation of the model configuration (.ml files).
- *OTScript compiler*: performs syntax and type checking. Produces concise messages helping the developer to find errors in the behavior configuration (.br files).
- *OTScript source browser*.
- *Interactive evaluation of OTScript requests*
- *Code profiler*: fine tuning of OTScript methods and rules
- *Source level debugger*

4- Extension mechanism in UML Metamodeling

Extension mechanisms allow to customize and extend the UML. A domain-specific metamodel serves as a formal definition of an extension to the UML for the modeling domain. It adds *more semantic depth* to the standard metamodel and thus builds a *foundation* for *model analysis* and *code generation*. Providing good metamodeling support within the UML enables solution developers to seamlessly integrate new extensions into CASE tools and build special-purpose solutions outside of the standard UML applications [8]. However, the support for metamodeling within the current version of UML (i.e. via stereotypes, tagged values and constraints, possibly gathered in profiles) is constrained. While many CASE tools are open to the implementation of manifold extensions they do not have an inherent metamodeling facility.

OpenTool/UML extension mechanism

The extension mechanism of OpenTool is based on a real metamodeling technique. It has therefore no limitations compared to the UML extension mechanism, which is limited to only adding new concepts to UML. In fact, OpenTool allows for adding easily new concepts in the UML metamodel implemented in the tool, as well as modifying already existing concepts. Furthermore, the concepts that have been added/modified are fully integrated in the tool and are made available in the model editing environment: in the browser, in property sheets, as well as in the graphical notation environment (e.g. in the diagrams). This is enabled by the powerful metamodelisation language OTScript, described in more detail in previous sections.

5 – An extension for Aspect Oriented Programming

In this section, we illustrate our approach through the introduction of an extension to UML 1.4 in OpenTool/UML providing concepts for supporting AOP. That extension will enable an OpenTool/UML user to use AOP techniques in UML models using the approach defined in [10].

Metamodeling

The AOP extension is implemented as an OpenTool module named `aop`, and coded in an `aop.ml` and an `aop.br` files. This new module supplements the existing modules supporting UML 1.4. A fragment of the `uml_aop.mn` file is shown below. It shows the list of modules loaded when launching OpenTool/UML+AOP.

```
UMLAOP :
LABEL "UML 1.4 + AOP WP"
VERSION "3.3.011"
{
  kernel
  extensions
  types
  systems
  ...
  aop
};
```

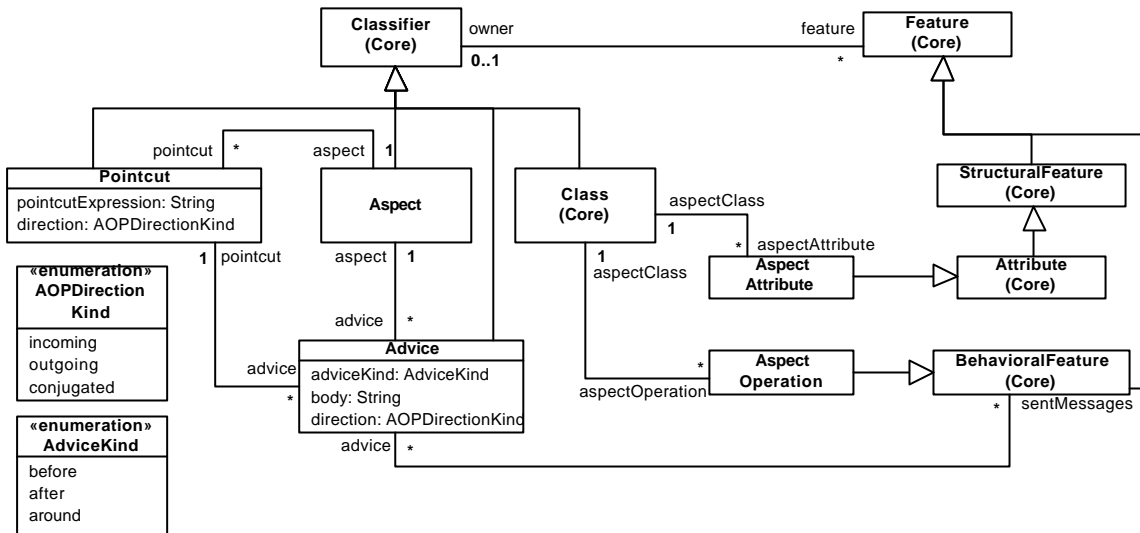


Figure 3: UML AOP Extension Metamodel

The next step is to define the new concepts required by this extension. This is done by describing a metamodel for AOP in a UML 1.4 class diagram. Basic concepts for classes, multiple inheritance, binary associations and class attributes are reused. Existing metaclasses are imported, from the Core package defined in the UML1.4 metamodel, into a new package, named `aop`, which contains the AOP extension to the UML1.4 metamodel.

Such a package in the metamodel corresponds to an independent module that can be added to, or removed from, OpenTool/UML1.4. A package is generally mapped to a pair of `.ml` and `.br` files. This preserves the modularity resulting from the package structure. Such modularity allows for incremental definitions of metaclasses. For instance, in the AOP model, some primitives UML1.4 metaclasses (`Class`, `BehavioralFeature`) are added new associations, without modifying their former definition.

The core concept of the AOP extension is modeled by the `Aspect` metaclass. An aspect is a classifier since it can declare attributes and operations, modeled by the `AspectAttribute` and `AspectOperation` metaclasses. Instances of `AspectAttribute` and `AspectOperation` are associated to the aspect using the inherited `owner-feature` association defined in the UML core. They are also associated to some other classes, to which they are added, using the `aspectClass-aspectAttribute` and `aspectClass-aspectOperation` associations. An aspect also owns pointcuts and advices, modeled by the `Pointcut` and `Advice` metaclasses and the `pointcut-aspect` and `pointcut-advice` associations. A pointcut specifies a pattern of message interception, specified with the `pointcutExpression`. Finally, each advice uses exactly one pointcut, through the `advice-pointcut` association, and specifies some kind of action to be performed when the pointcut condition occurs, using the `body` attribute. This metamodel is then automatically translated to OTScript code, stored in the `aop.ml` file.

Building the user interface

We will now describe the design of the user interface for creating, viewing and editing the new concepts introduced by the AOP package.

The function of the user interface framework is to expose to the user different views or projections of the semantic model. The OpenTool UI framework provides different types of view: property sheets, tree explorers, structured browsers and graphical editors for diagrams. It is also possible to define free windows using a standard widget library. These views are described using the UI subset of OTScript in the behavior part of the configuration (`.br` files). An entity in the model can be viewed in both a property sheet or a graphical editor. The consistency between different views created using this viewing mechanisms is automatic: a modification of an entity appearing in one of the views existing at a given time is instantly propagated in all the other views displaying that entity.

The standard UML property sheet for a classifier is shown in Figure 4.



Figure 4: Standard UML Classifier property sheet

The property sheet for any metaclass is defined in OTScript code by a FORM declaration in the corresponding OTScript class. In the AOP model, attributes `pointcutExpression` and `direction` have been defined in the `Pointcut` class. The `Pointcut` class has been defined as a subclass of the `Classifier` class, so the property sheet associated to the

Classifier class is inherited by the Pointcut class. We simply need to add in its OTScript code a FORM declaration creating a dedicated tab named Pointcut, with two fields corresponding to the two additional attributes.

The Pointcut property sheet has now a new tab named "Pointcut", as shown in Figure 5.

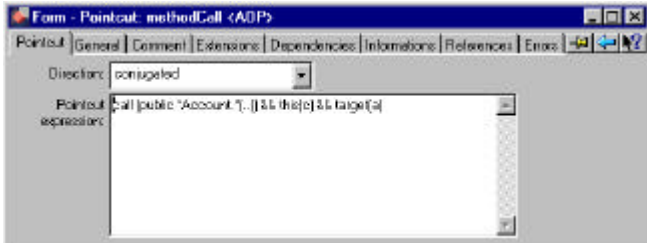


Figure 5: AOP extension in the standard UML Pointcut property sheet

Let's now design a more complex editor such as a diagram editor. A diagram is a tree of graphical elements. Each graphical element is associated to a semantic entity. The root of the tree is associated to the diagram entity. Our need is to extend the UML class diagram graphical editor with the AOP concepts proposed by [10].

The graphical editor for class diagrams is defined in OTScript code by a DIAGRAM declaration in the UMLClassDiagram. This declaration specifies precisely the entities to be displayed in the diagram, the look of these entities on the screen (DISPLAY statement). So means for creating new instances of Aspect, Pointcut, Advice, ... must simply be added to that declaration.

Figure 6 shows the original UML class diagram editor.

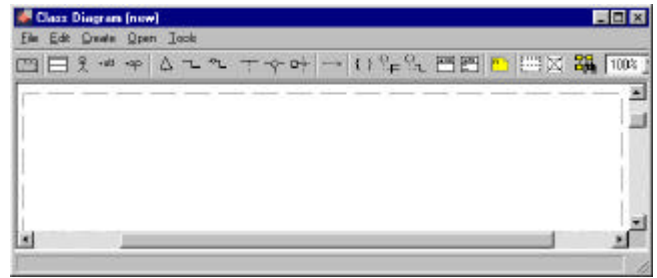


Figure 6: UML class diagram editor

New creation buttons for Aspect and Pointcut classes are introduced by adding CREATE Aspect and CREATE Pointcut clauses in the UMLClassDiagram. Creation buttons for aspect attributes (AspectAttribute class), aspect operations (AspectOperation class) and advice (Advice class) have been introduced in the same way. Figure 7 shows this editor with these AOP extension.

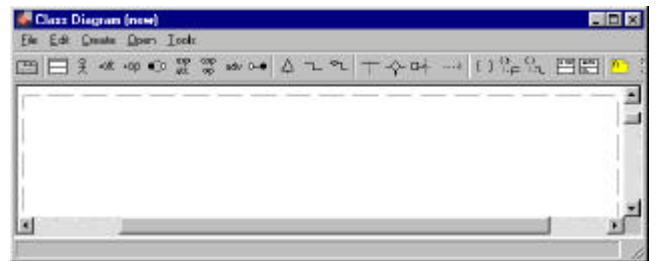


Figure 7: UML + AOP class diagram editor

With the extended class diagram editor, and the property sheets associated to UML and AOP modeling elements, we can now model the Logging Aspect Model as found in [10]. The result is shown by Figure 8.

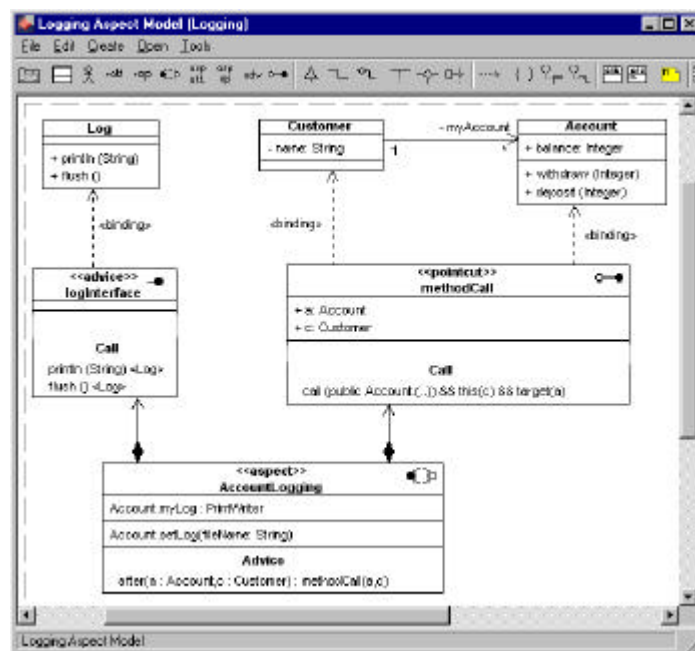


Figure 8: The Logging Aspect Model on the UML + AOP class diagram editor

Consistency checking

The basic concepts used in metamodeling (inheritance, associations and attributes) guaranty a level of consistency on the semantic model: associations specify the types of objects that can be related and cardinalities add more constraints on these relations. Most of the time, such structural consistency is sufficient. However, there is sometime a need for a more sophisticated constraint expressiveness.

OTScript language provides a powerful consistency checker, based on the definition of rules. A rule is a special kind of method associated to a class, whose result must evaluate to a boolean. It specifies a property that must be verified by the semantic entities of that class. A rule can be tagged as “immediate”; in that case, the checking of the rule is triggered automatically when any entity of that class is modified. When a rule is tagged as “deferred” the checking is triggered on user’s request.

If the property specified in the rule is not true for a specific entity, the checker creates an error associated to the entity. The presence of the error usually changes the color display of the entity in diagrams to red. At any time, the list of errors on the selected entity can be viewed in the “Errors” tab of its property sheet. Also, the “Error browser” presents an index of all the errors detected in the model. An error is presented to the user with its name, and a description text explaining the cause of the error. That description is provided through the EXPLAIN statement associated to a rule. If, during the subsequent edition of the model, the condition that raised the error disappears, the error is immediately deleted.

Another type of rule is the repair rule, which attempts to correct the cause of the error. A repair rule is provided through the REPAIR statement associated to a rule.

In the AOP metamodel, one could write a rule for checking that a pointcut is used at least once by an advice. With such a rule, a pointcut is created, named "pointcut1", by default, it appears errored (in red) as no advice yet indicates that it uses the new pointcut. This results in the production of an error. This error is notified to the user by displaying “ pointcut1” in red. This error can be viewed by clicking on the “Errors” tab of the property sheet of “ pointcut1”. This is illustrated in Figure 9.

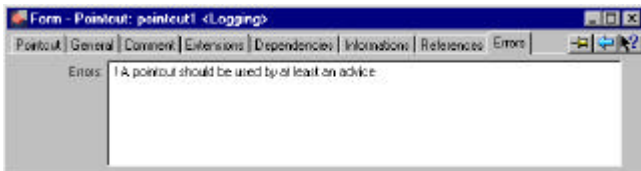


Figure 9: Errors list property sheet

By double-clicking the error, its specific property sheet opens, providing more information to the user (Figure 10).

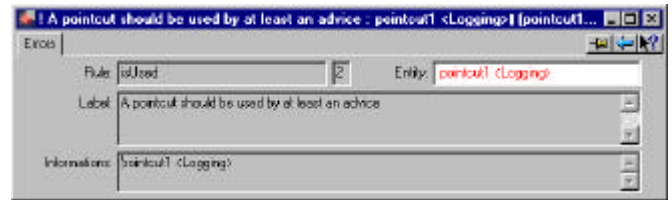


Figure 10: Detailed error list property sheet

The bottom field provides an explanation of the error based on the EXPLAIN statement of the rule.

The global list of all errors can be viewed at any time using the error browser. This browser presents the entities for which errors have been detected (list on the left), and for each entity, the errors linked to it (list on the right).

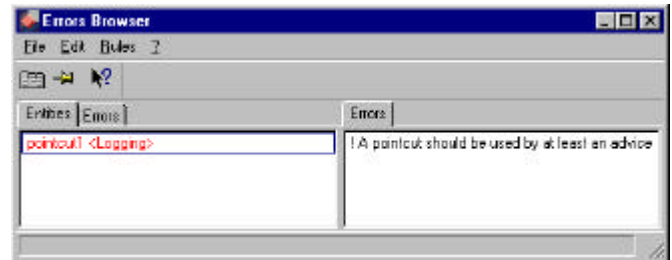


Figure 11: Errors Browser

If we create a new advice referencing "pointcut1", the error will automatically disappear.

6 – Conclusion and perspectives

We have illustrated in this paper how OpenTool’s metamodeling capabilities allows one to easily extend the UML metamodel implemented in OpenTool/UML and provide tool support to such extensions. This approach has been evaluated through the Aspect Oriented Programming case study.

Using its native metamodeling facilities, OpenTool can also be used successfully to design tools based on any metamodel, not necessarily derived from the UML metamodel.

The metamodeling facilities offered by OpenTool, together with the organization of metamodels developed with OTScript, may help to prefigure what could be tools supporting the upcoming UML 2.0 [11].

Mapping packages in the architecture of the UML 2.0 metamodel to a concept such as OpenTool modules might allow an incremental implementation of UML 2.0, as well as an easy configuration of future UML tools, e.g. to support well defined subsets of the language.

In the same way such mechanisms offer a good candidate for managing properly extensions to UML, be they dialects defined using profiles, or UML-related languages augmenting UML metamodel with appropriate metaclasses and metarelationships.

7 - References

- [1] Booch, G., Jacobson, I., and Rumbaugh, J.: *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA (1999)
- [2] OMG: *OMG Unified Modeling Language Specification*, <http://www.omg.org>.
- [4] OMG. *Meta-Object Facility (MOF) Specification*, <http://www.omg.org>.
- [5] Jean Bezivin. *On Different Interoperability Modes in Software Engineering: the Case of Modeling Activities at OMG*. In Proceedings Software Engineering'98, Paris, December 1998.
- [6] Marie-Noelle Terrasse and Marinette Savonnet. *Formalization of the UML metamodel: An Approach based upon the four-layer metamodeling architecture*. In Proceedings Software Engineering'00, April 2000.
- [7] <http://www.tni-valiosys.com>
- [8] James O. Gillian. *Improving the open source software model with UML Case Tools*. Published in Issue 67 of Linux Gazette, June 2001.
- [9] Sinan Si Alhir. *Unified Modeling Language: Extension Mechanisms*. Published in Distributed Computing, December 2000.
- [10] Mohamed Mancona Kandé, Jörg Kienzle and Alfred Strohmeier: *From AOP to UML – A Bottom-up Approach*, Aspect-Oriented modeling with UML Workshop in conjunction with the 1st International Conference on Aspect-Oriented Software Development, April 2002, Enschede, The Netherlands
- [11] U2 Partners: *Unified Modeling Language: Infrastructure*, OMG TC. Document ad/2002-09-01, <http://www.u2-partners.org>