Adopting Software Engineering Trends in AI

Dragan Đurić, Vladan Devedžić, Dragan Gašević

FON – School of Business Administration, University of Belgrade
POB 52, Jove Ilića 154, 11000 Belgrade, Serbia and Montenegro
Tel. +381-11-3950853, Fax. +381-11-461221
Email: dragandj@gmail.com, devedzic@fon.bg.ac.yu, dgasevic@acm.org

In order to design and develop a reliable, robust, well-architectured, and easy-to-extend application or tool in any field, it is important to conform with sound principles and rules of software engineering. Intelligent systems are no exception to that rule. It is especially important for AI development *tools* to be designed closely following SE practices.

Keeping an eye on current SE developments and trends can help design AI tools to remain stable over a longer period of time. Some trends in SE are general and span many, if not all fields and application domains. Others are more specific, but can be relevant for AI. For example, a general trend in SE is the use of tailored versions of Unified Modeling Language (UML) to alleviate design of system specifics. In practice, this usually means defining UML stereotypes and profiles to facilitate modeling of a specific domain (see Sidebar 1 for a concise explanation of such UML mechanisms). Another such trend is that of plug-in architectures, that allow for easy and modular extension of tools and systems by features targeting specific groups of users. Such a plug-in architecture is effectively used in Protégé-2000 AI tool (http://protege.stanford.edu/). An example of AI-relevant but more specific trends in SE is agent-oriented software engineering [Wooldridge and Jennings, 1999]. It uses software agents as metaphors and modeling elements in system analysis and design.

An emerging SE trend, with intensive support from Object Management Group (OMG), is application development based on Model-Driven Architecture (MDA) [Miller and Mukerji, 2003]. MDA is a generally applicable idea, but is simultaneously of specific interest to AI developers since it has much in common with ontology modeling and development [Devedžić, 2002]. Essentially, MDA defines three levels of abstraction in system modeling. *Computation Independent Model* (CIM) corresponds to the system's domain model and is similar to the domain ontology. It does not show details of the system structure. *Platform Independent Model* (PIM) is computationally dependent, but not aware of specific computer platform details. In other words, PIM shows how a technology-neutral virtual machine runs the system. *Platform Specific Model* (PSM) introduces platform-specific issues and implementation details. The goal of MDA modeling is to shift the designer's focus from PSM towards PIM and CIM and use automated tools to transform PIM to PSM.

We have developed AIR, an integrated AI development environment based on MDA modeling concepts. Using MDA philosophy in AIR makes possible to employ mainstream software technologies that users are familiar with, and expand them with new functionalities.

AIR framework

AI developers may need to use a number of knowledge representation, reasoning, communication, and learning paradigms in their systems. Moreover, they may want to do parts of their system modeling with CASE and other tools they are used to. When the project is already underway, designers might like to switch to a new tool as well. In all such cases, easy and seamless integration of different formats, tools, and techniques within the same project is highly desirable.

In order to support such integration, the AIR framework uses MDA metamodeling principles (see Sidebar 2 for details). The central part of the framework is a model base, represented as a metadata repository, Figure 1. It can include models of different kinds of intelligent systems, as well as models of any other domains of interest in a specific project. Java Metadata Interchange (JMI) compliant APIs enable access to these models . The mechanism for exchanging the models with other applications, agents, and tools is XML Metadata Interchange (XMI), which is an open W3C standard.

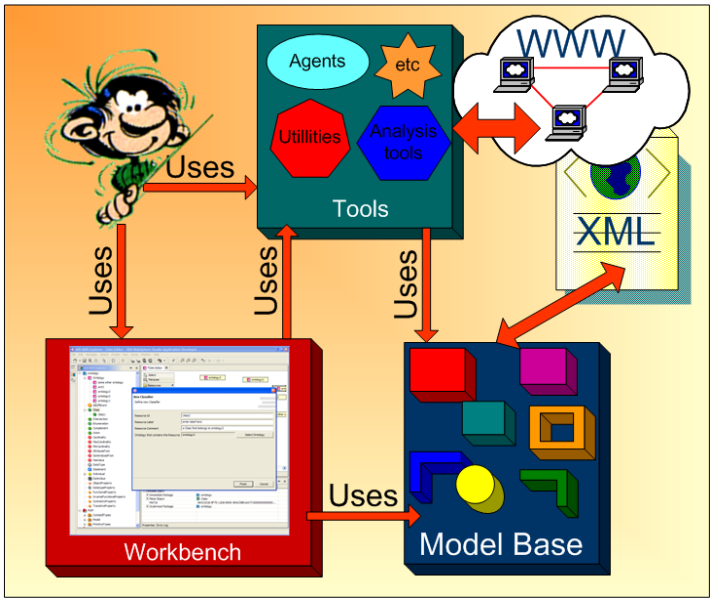

Figure 1 – Model-based development using the AIR framework

The other important part of AIR is an integrated development environment with a rich GUI for specifying and manipulating the models – AIR Workbench. It implements the MDA metamodeling architecture illustrated in Figure 2. Typically, developers use UML to represent their domain models (M1 layer). Specific

instances of domain model concepts are databases, objects, programs, and other concrete entities (M0 layer). However, MDA also provides means for defining modeling languages themselves (such as UML and UML profiles). They are defined in the form of metamodels (M2 layer). Specifying metamodels of modeling languages is done using *Meta-Object Facility* (MOF). MOF is also an OMG standard like UML, and is the meta-metamodel (M3 layer). It defines an abstract language and framework for specifying, constructing and managing technology-neutral metamodels. Any modeling language, such as UML (or even MOF itself!) can be defined in MOF. That is exactly what AIR Workbench is used for – it lets the users specify the XML-based metamodel of a desired representational format, language, or paradigm, put the metamodel in the repository, and make possible for tools and applications to use it along with the other metamodels for integration purposes.

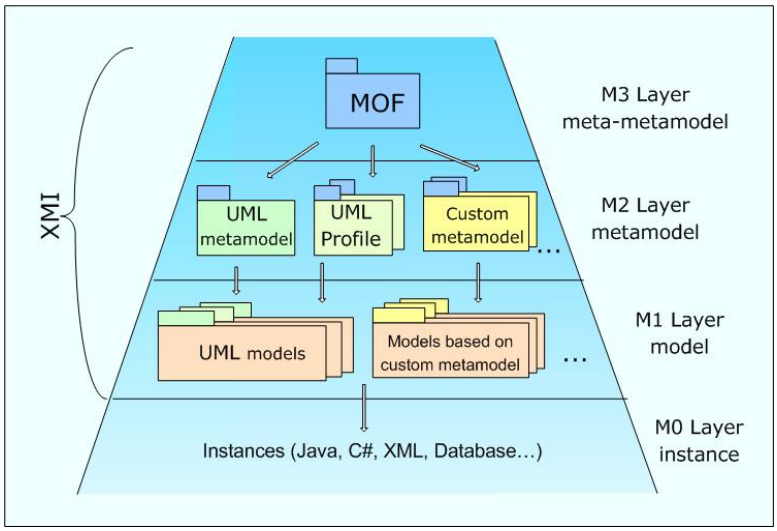

Figure 2 – Four-layer MDA modeling framework

The AIR framework and AIR Workbench are developed with several goals in mind:

- To provide a general modeling and metamodeling *infrastructure* for analysis, design, and development of AI systems.
- To make the infrastructure, the corresponding tools, and the resulting metamodels Semantic Web-ready.
- To be able to instantiate/specialize the general framework, i.e. to define more specific AI frameworks (starting from the general one) to support developments in specific domains, such as manufacturing, medicine, and education.

AIR metadata repository

The MOF Specification also defines a framework for implementing repositories that hold metadata (e.g., models) described by metamodels. Standard XML technology is used to transform metamodels into metadata API, giving the framework an implementation. Figure 3 shows an overview of a MOF repository and its implementation in Java.

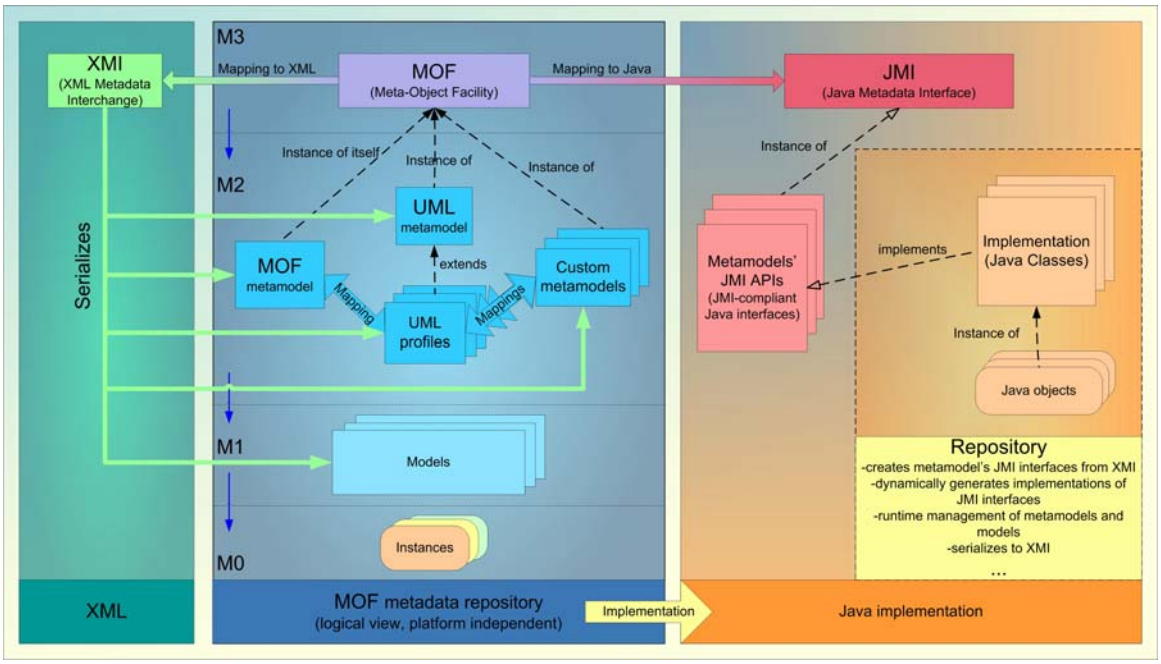

Figure 3 – Metadata repository structure (Java-based implementation)

A conceptual view of a MOF-based repository is shown in the center of Figure 3. It reflects the four-layer MOF-based MDA architecture. Custom metamodels, specified using MOF as a meta-metamodel, can define mappings to UML and UML profiles. This enables the use of UML tools to manipulate the metamodels. XMI serializes MOF-based metamodels and models into plain text (XML), thus making such data ready to exchange in a standard way and to be read by any platform-specific implementation.

Java-based implementation of the repository uses JMI, the Java metadata API. Starting from any MOF-based metamodel (serialized to XMI), JMI-compliant metamodel-specific JMI interfaces can be generated. These interfaces are used to access Java metadata repository, which is implemented by Java classes. All data from repository can be serialized into XMI and then exchanged with other repositories, regardless of their implementation. It is only required that they support MOF-based metadata (i.e. that they can “understand” MOF XMI format).

The reference implementation for JMI metadata repository is Unisys’ CIM (www.unisys.com), but it seems that it has not been updated recently. The other implementation is NetBeans MDR (mdr.netbeans.org), a part of open source NetBeans project. NetBeans MDR is used in AIR as the metadata repository due

to its generic implementation of JMI interfaces and frequent upgrades [Đurić et al., 2005]. Any metamodel can be loaded from XMI and instantly implemented using Java reflection. Other metadata repositories can be used with AIR as well, provided that they support the features needed.

Ontology definition metamodel

OMG has recently announced a Request for Proposals for an ontology modeling architecture. In our approach to ontology modeling in the context of MDA, shown in Figure 4, several specifications are defined:

- Ontology Definition Metamodel (ODM);
- Ontology UML Profile – a UML Profile that supports UML notation for ontology definition (OUP);
- Two-way mappings between OWL and ODM, ODM and Ontology UML Profile, and from Ontology UML Profile to other UML profiles.

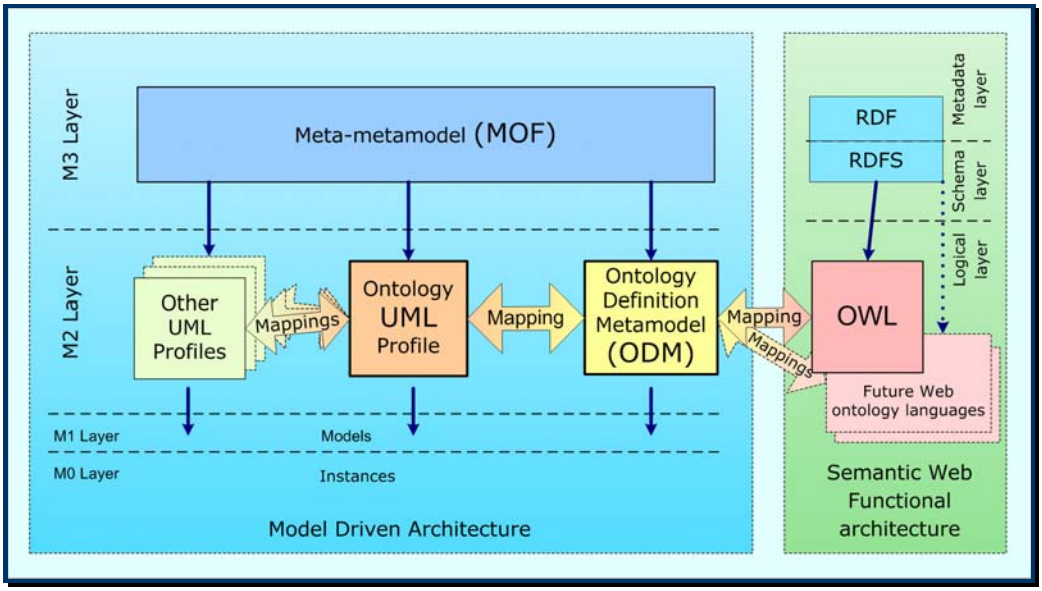

Figure 4 – Ontology modeling in the context of MDA and Semantic Web

Ontology Definition Metamodel (ODM) is designed to cover common ontology concepts. A good starting point for ODM construction is OWL since it is the result of the evolution of existing ontology representation languages, and is a W3C recommendation. It is at the Logical layer of the Semantic Web, on top of RDF Schema (Schema layer). In order to make use of graphical modeling capabilities of UML, an ODM should have a corresponding UML Profile. This profile enables graphical editing of ontologies using UML diagrams as well as other benefits of using mature UML CASE tools. Both UML models and ODM models are serialized in XMI format so the two-way transformation between them can be done using XSL Transformation. OWL also has its representation in the XML

format, so another pair of XSL Transformations should be provided for two-way mapping between ODM and OWL. For mapping from the Ontology UML Profile into other, technology-specific UML Profiles, additional transformations can be added to support the use of ontologies in modeling the other domains and vice versa.

Ontlogy UML profile

UML is a metamodel that also defines a graphical representation of its concepts and standard extensions that enable other metamodels to use it as their graphical representation, which is very useful for other MOF-complant modeling languages. Ontology UML profile (OUP) is an UML extension for graphical ontology modeling that enables ontology developers to use mainstream, well-known software-development tools. Such tools have good support (user base, literature…) and let the developers comfortably model ontologies. The resulting can be transformed into a standard format, ODM, shared, and used by tools that support that standard.

Figure 5 shows an example ontology modeled using OUP. Person is a class that represents a human that can have a name, almost always has a nationality, social security number, colleagues that work with her/him, and many other properties not shown in this simplified ontology.

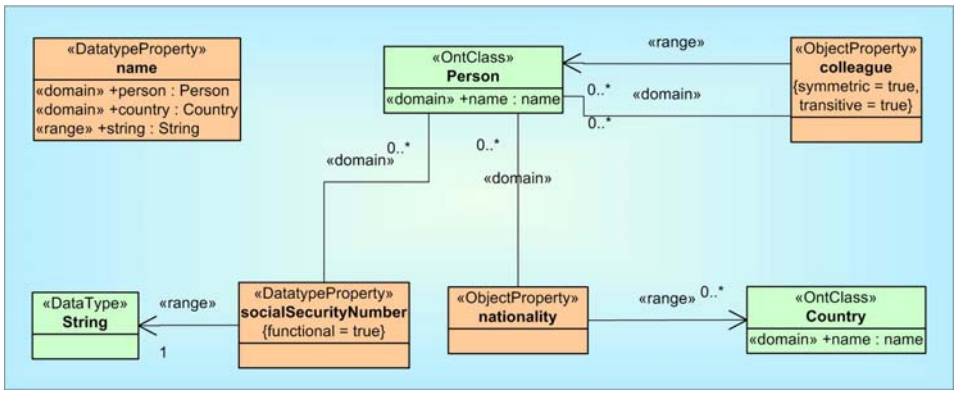

Figure 5 – An ontology modeled in OUP using UML design tool

The diagram shown in Figure 5 is nothing more than a graphical representation of some UML model. However, it can be transformed in a standard way into an ODM model, which does not have its own graphical representation but is ontology-aware. When an ontology that we just created reaches the ODM representation, it can be further used by various software tools (agents, reasoners, analyzers…) that do not care about how people see graphical representations of ontologies, but know about what is an ontology class, property, or instance, so they can work with that data.

As we can see, these concepts are clearly represented visually in an intuitive way that most people can easily understand. More importantly, it can be done

with minimal additional effort in software design, development and maintenance, since OUP is based on a standard that is going to be fully supported by modeling tools within a few years with the adoption of UML2 (UML 1.4 standard is almost fully implemented by most vendors, but proprietary formats and extensions make some difficulties).

The role of XML technologies

The importance of XML technologies is well-known to the AI community, especially after the introduction of the Semantic Web. The Semantic Web architecture itself is based on XML. The standard Semantic Web knowledge model (RDF), as well as the language for specifying simple ontology vocabularies (RDFS) are defined on the top of XML. These two standards are the basis for the current W3C Web Ontology Language (OWL) recommendation. Of course, ontology languages are not an isolated example of applying XML in AI. For example, a language for sharing rules on the Semantic Web (RuleML) is based on XML as well. Moreover, there are several AI development tools that define their own XML formats for sharing their knowledge bases (e.g. JessGUI tool [Jovanović et al, 2004] creates XML knowledge bases for Jess, a well-known expert system shell).

In the AIR framework, we use XMI for sharing metadata in MDA. In fact, XMI is not a specific XML format; it is rather a set of production rules that specify how one transforms a MOF-compliant model (i.e. metamodel and meta-metamodel) into a corresponding XML Schema and an XML document. Using this production principle we have a standard way of sharing MDA metadata by XML. Of course, there are a few standard XML Schemas for MOF-compliant models like the XMI schema for UML as well as the XML schema for MOF. However, it is necessary to define XML schemas for every new custom model or metamodel.

Knowing that these two different communities (AI and MDA) both employ XML, we should bridge the gap between them using XML. Since they use different XML formats we should define transformations between them. XSL Transformation (XSLT) is coming as a natural solution to this problem. XSLT is a standard language for transforming XML documents into other documents, either XML or regular text. Figure 6 illustrates how we support model sharing between a UML tool (Poseidon for UML that uses UML XMI) and an ontology editor (Protégé, i.e. its OWL plug-in). Applying this XSLT principle we do not have to change (i.e. reprogram and recompile) an existing tool, but we just develop an auxiliary tool (i.e. an XSLT) that extends the existing functionalities.

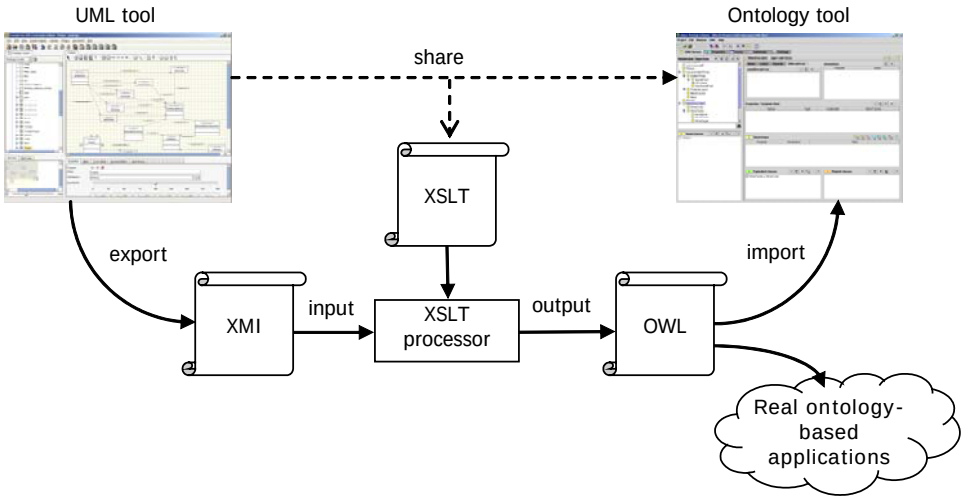

Figure 6 – The XSLT principle: extensions of present UML tools for ontology development

An important shortcoming of XSLT is its sensitivity to changes of the input format, so maintenance can be difficult. Note that XSLT-based solutions can be unsuitable for ontology languages since they can use different syntactic forms to express the same semantics [Decker et al, 2000]. This problem can be overcome using some of RDF Query languages (e.g. RDQL, TRIPLE, etc.) as these languages are also transformation languages. Their main advantage is that they do not depend on an XML document structure as they are based on RDF triplets.

Finally, note that transformations can be implemented using the results of the OMG's ongoing effort called MOF Query/View/Transformation (MOF QVT), which is a language for querying, viewing, and transforming MOF-compliant models and metamodels [OMG, QVT, 2002]. However, MOF QVT is not based on XMI, so we can only use it for transformations between MOF-compliant models (e.g. we can not transform ODM to OWL using MOF QVT).

AIR Workbench

AIR Workbench provides various tools with rich GUI that make the entire workbench user friendly. This workbench is built on top of the Eclipse plug-in architecture and Eclipse IDE (www.eclipse.org), today's leading extensible platform [Gamma and Beck, 2003]. The main difference between Eclipse and other extensible IDEs is that Eclipse consists entirely of plug-ins that work on a tiny platform runtime, whereas other IDEs are monolithic tools with some extensions. Thus, Eclipse core plug-ins are of equal importance as any other plug-in, including the AIR plug-ins.

Figure 7 depicts the Eclipse-based AIR plug-in architecture. Although only the Eclipse Core is mandatory here, there is no reason not to utilize Eclipse UI (SWT, JFace, and Workbench), help and team support, so they are not discarded. Using the entire Eclipse IDE, AIR adds the plug-ins related to MDR and Intelligent Systems – generic MDR support (AIR Framework, AIR NetBeans MDR,

AIR MDR Core), specific metamodel support (ODM, RDM, UML, CWM, etc.), and GUI-related (AIR MDR Explorer). These plug-ins are inserted at extension points defined by plug-ins that are parts of Eclipse IDE. Being treated in exactly the same way as Eclipse native plug-ins, the AIR plug-ins also extend each other and offer future plug-ins to extend them.

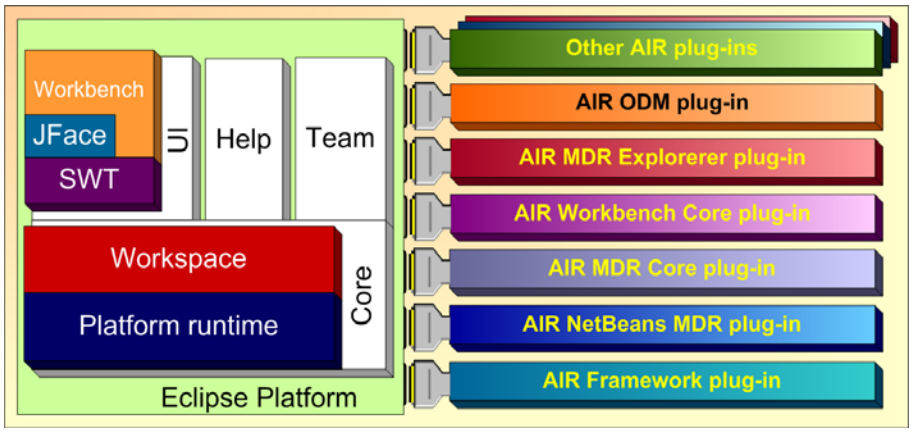

Figure 7 – Eclipse-based AIR plug-in architecture

A screenshot from the AIR Workbench is shown in Figure 8. The Explorer shows MOF-based models and metamodels graphically and serves as a starting point for model manipulation. Selecting any element, the user can reach menus specific for that element and perform various actions. These actions span from usual ones (instantiating, deleting, viewing properties etc.) to more specific (opening various metamodel specific editors, starting transformations etc.). Due to the underlying Eclipse architecture, these menus can be easily extended by new items that can initiate new actions.

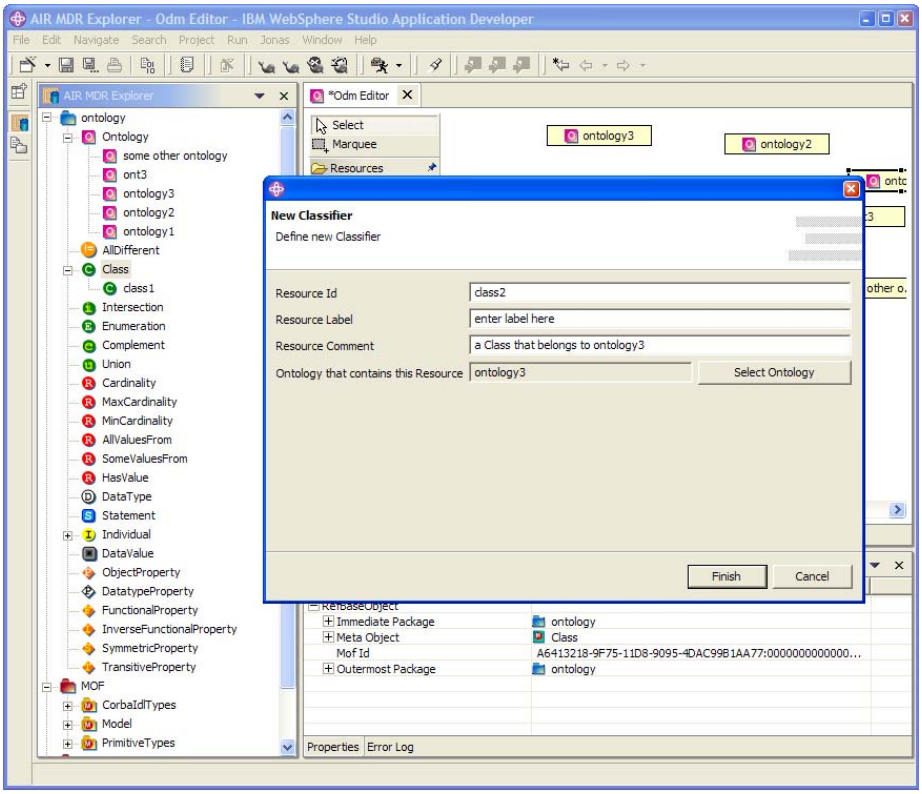

Figure 8 – An ontology in AIR MDRExplorer

Related work

Our work on the AIR framework coincided in time and thematically with the OMG request for proposals to define a specification of a MOF2 Metamodel, UML2 profile, and any additional information needed to support development of ontologies in the context of MDA, using UML modeling tools, the OWL language, and orward and reverse engineering for ontologies (http://www.omg.org/cgi-bin/doc?ad/2003-03-40). Both the MDA concept and the AIR framework are instances of hierarchical modeling, and we already had extensive experience with hierarchical modeling of AI systems. For example, we have developed a fairly general hierarchical framework for modeling AI systems, called OBOA [Devedžić and Radović, 1999], and have recently specialized it for development of fuzzy systems [Šendelj and Devedžić, 2004].

The AIR framework and AIR Workbench differ from traditional general-purpose languages and environments for constructing intelligent applications, such as Parka (http://www.cs.umd.edu/projects/plus/Parka/parka-db.html) and Loom (http://www.isi.edu/isd/LOOM/LOOM-HOME.html) in that AIR provides a highly modular development infrastructure, rather than a complex development environment. AIR is also different from specific AI tools, such as expert system shells or fuzzy system development environments, since it provides flexible extension mechanisms (MOF-based metamodel specifications) for representing

different paradigms as the designer decides. On the other hand, the AIR framework is similar in a way to component-based system frameworks and their plug-and-play design approach (http://www.cbseng.com/). Just like CBS frameworks enable component integration, AIR provides means for an easy integration of different models.

There are also other efforts to enable integration of different knowledge representation paradigms, but none of them relies on standard MOF-based metamodels. Well-known examples include Open Knowledge Base Connectivity (OKBC), which is an application programming interface for accessing knowledge bases stored in different knowledge representation systems (http://www.ai.sri.com/~okbc/), and Protégé-2000 graphical tool for ontology editing and knowledge acquisition (http://protege.stanford.edu/), that has a number of plugins for different formats and tools.

AIR Workbench screen layout is designed after typical GUIs of CASE tools like Together, Rational Rose, and Posseidon, as well as after Protégé-2000 GUI.

Conclusions

Bringing AI and SE close together results in well-engineered AI systems with a firm SE backbone. There are a number of possibilities for such a synergy. On one end there are disciplined approaches with low "coupling", such as developing and using an API for building AI systems (like in OKBC) or merely using UML-based CASE tools in designing AI systems. On the other end, there are integrated AI development tools. In between the two extremes, there are still other opportunities to add more SE flavor to AI systems development. One can use a suite of tools instead of a complicated integrated tool, or can extend the basic tool with a number of useful plug-ins (as in Protégé-2000), possibly with an idea of evolving the basic tool into a framework. Using MDA, UML, and MOF standards from OMG is yet another possibility. True, it does take some time for AI developers to get used to it. On the long run, it does pay off as well. At its core are standard SE tools and XML technologies that many developers are familiar with. Due to MOF, it enables integration at the metamodeling level, which is related to ontological engineering. It also enables smooth and gradual transition between traditional and emerging modeling styles and paradigms.

References

V. Devedžić, "Understanding Ontological Engineering," *Communications of the ACM*, Vol.45, No.4ve, pp. 136-144, April 2002.

V. Devedžić and D. Radović, "A Framework for Building Intelligent Manufacturing Systems," *IEEE Trans. on Systems, Man, and Cybernetics, Part C - Applications and Reviews*, Vol.29, No.3, pp. 422-439, 1999.

J. Miller and J. Mukerji (Eds.) (2003, May), "MDA Guide Version 1.0," OMG Document: omg/2003-05-01. [Online]. Available: http://www.omg.org/mda/mda_files/MDA_Guide_Version0.pdf (current Nov. 2004).

R. Šendelj and V. Devedžić, "Fuzzy Systems Based on Component Software", *Fuzzy Sets and Systems*, Vol.141, No.3, 2004, pp. 487-504.

M. Wooldridge and N. Jennings, "Software engineering with agents: pitfalls and Pratfalls," *IEEE Internet Computing*, Vol.3, pp. 20-27, May/June 1999.

E. Gamma, K. Beck, "Contributing to Eclipse: Principles, Patterns and Plug-Ins", Addison-Wesley, 2003.

D. Đurić, D. Gašević, & V. Devedžić, "Ontology Modeling and MDA," Accepted for publication in *Journal on Object Technology*, Vol. 4, No.1. 2005. Forthcoming.

J. Jovanović, D. Gašević, & V. Devedžić, "A GUI for Jess," *Expert Systems with Applications*, Vol. 26, No.4, pp. 625-637, 2004.

S. Decker, S. Melnik, F. van Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Ederman, & I. Horrocks, "The Semantic Web: The Roles of XML and RDF," *IEEE Internet Computing*, Vol. 4, No. 5, pp. 63-74, Sep/Oct 2000.(2002). "MOF 2.0 Query/Views/Transformations Request for Proposal," OMG Document ad/2002-04-10,) [Online]. Available: http://www.omg.org/docs/ad/02-04-10.pdf

Sidebar 1 – UML Profile Basics

UML Profile is a concept used for adapting the basic UML constructs to some specific purpose. Essentially, this means introducing new kinds of modeling elements by extending the basic ones, and adding them to the modeler's tools repertoire. Also, free-form information can be attached to the new modeling elements.

The basic UML constructs (model elements) can be customized and extended with new semantics by using four *UML extension* mechanisms defined in the UML Specification [1]: stereotypes, tag definitions, tagged values, and constraints. *Stereotypes* enable defining virtual subclasses of UML metaclasses, assigning them additional semantics. For example, we may want to define the «`OntClass`» stereotype, Figure A, by extending the UML `Class` metaclass to denote the modeling element used to represent ontologies (and not other kinds of concepts).

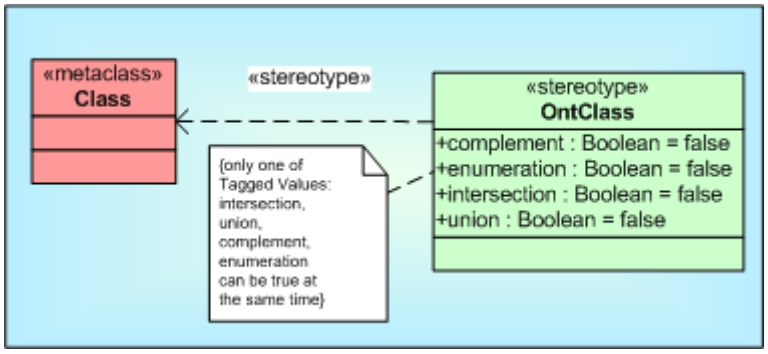

Figure A – New stereotype definition

Tag definitions can be attached to model elements. They allow for introducing new kinds of properties that model elements may have and are analogous to metaatribute definitions. Each tag definition specifies the actual values of properties of individual model elements, called *tagged values*. Tag definitions can be attached to a stereotype to define its virtual metaattributes. For example, the «`OntClass`» stereotype in Figure A has a tag definition specifying 4 tagged values (for enumeration, intersection, etc.).

Constraints make possible to additionally refine the semantics of the modeling element they are attached to. They can be attached to each stereotype using OCL (Object Constraint Language) [1] or English language (i.e. spoken language) in order to precisely define the stereotype's semantics (see the example in Figure A).

More details about UML extension mechanisms can be found in [1] and [2].

A coherent set of extensions of the basic UML model elements, defined for specific purposes or for a specific modeling domain, constitutes a UML profile.

References

1. Object Management Group (2003, March). OMG Unified Modeling Language Specification. [Online]. Available: http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.zip (current Mar. 2004).
2. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1998.

Sidebar 2 – Metamodeling with MOF

Those familiar with UML and object-oriented modeling should easily understand MOF, since it is basically the core of UML2. Being a subset of UML2, MOF contains only concepts that are needed for metmodeling (modeling of modeling languages) – UML, XML, ODM, or even MOF itself.

An example of a metamodel defined by MOF concepts is shown in Figure B, which depicts a simplified metamodel of XML.

As most readers have been using XML, they are already familiar with those concepts when thez are coded in text. An XML document consists of a number of *Elements*, that have their names. *Elements* are further specialized into *Nodes* and *Attributes*. *Element*, *Node*, and *Attribute* are instances of MOF *Class*, whereas a name is an instance of MOF *Attribute*. Each node can contain its child *Nodes* and *Attributes*, which is modeled by the MOF Association *Contains*. Using MOF as a metamodeling language (M3 layer), we have just described the structure of XML documents – we defined the XML metamodel (M2 layer).

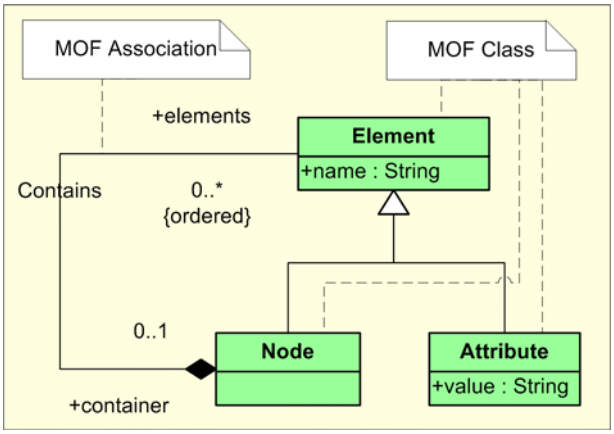

Figure B – Simplified metamodel of XML

Using this metamodel, we can describe specific XML documents, for example a CD collection catalogue. An entry in such a catalogue contains the artist name and the album name. *Catalogue* is the root *Node* that contains multiple entries (also an instance of *Node*). Each one of them contains the *artistName* attribute (not a MOF *Attribute*) and the *albumName* attribute. We still do not have a standard XML document – we have its model (M1 layer), in the repository or serialized in XMI format. When we transform this document to plain XML, we get its instance, shown in the following code snippet.

```
<!-- ...-->
<catalogue>
   <entry artistName=”Deep Purple”
         albumName=”Machine Head”/>
   <entry artistName=”Dire Straits”
         albumName=”Sultans of Swing”/>
   <entry artistName=”The Clash”
         albumName=”Combat Rock”/>
   <entry artistName=”The Ramones”
         albumName=”End of The Century”/>
     <!-- ...-->
</catalogue>
```

This model is an XML model, so it describes only the hierarchy of some nodes that have some elements. It does not know that “The Clash” is a rock band; it is simply an instance of value (see Figure A) of type String. If we model our

collection in some semanticaly richer metamodel, Ontology Definition Metamodel for example, our model will make distinction between *artistName* and *albumName*. If, on the other hand, we parse it with an XML parser, we will get just a bunch of elements – nodes and attributes.

Do not let the diagram in Figure B confuse you – although it is a UML diagram (MOF uses UML diagrams for representation), it represents a metamodel that is modeled in MOF (M3 layer), not a UML model that will be used to generate plain Java or C++ classes. Although MOF is at the core of UML (a MOF model is also a UML2 model), it is used at M3 layer to model metamodels that are at M2 layer. If we use the same concepts at M2 layer to develop models that are at M1 layer, then we are using UML2. This can be a little confusing, particulary if we model UML2 itself as a metamodeling language; its core concepts are self-described in this case. Also note that a UML diagram is only a visualization of a UML model – thus they are not the same.