Coral: A Metamodel Kernel for Transformation Engines

Marcus Alanen and Ivan Porres

TUCS Turku Centre for Computer Science
Department of Computer Science,
Åbo Akademi University
Lemminkäisenkatu 14, FIN-20520 Turku, Finland
e-mail:{marcus.alanen,ivan.porres}@abo.fi

Abstract. A metamodel kernel is a program library or application framework that is used to manage models described in user-defined modelling languages. Metamodel kernels provide the basic functionality to create models, add, delete and update elements in an existing models and to store and retrieve models from a XMI document.
Coral is a metamodel kernel that is used to try in practice new research ideas in modelling technology. In this short paper, we describe Coral, our own implementation of a modelling tool and some discoveries related to modelling and metamodelling that we have found.
Keywords: Modelling Frameworks, Model Driven Engineering, Metamodelling, Modelling

1 Introduction

The advance of modelling techniques both in academia and industry has lead to the development of several commercial modelling tools. However, the research area of modelling tools is still relevant as solid frameworks are required to empower software developers to actually use the benefits of a model driven architecture. In this paper, we present the idea of a metamodel kernel and our work on Coral, a generic open source metamodel kernel.

A metamodel kernel is a program library or application framework that is used to manage models described in user-defined modelling languages. Metamodel kernels provide the basic functionality to create models, add, delete and update elements in an existing models and to store and retrieve models from a XMI document. Examples of metamodel kernels are the Eclipse EMF project and the Netbeans Metadata Repository.

All metamodel kernels are based on a specific metamodelling language. This metamodelling language defines the building blocks of all modelling languages support by the kernel. An example of a metamodelling language is the OMG Meta Object Facility (MOF) [4]. All the existing metamodel kernels implement at least a subset of MOF. However, MOF is not a metamodel kernel as such since it is not a software tool. Eclipse uses EMF as its metamodelling language that is comparable to EMOF, a simplified version of MOF. Coral uses its own Simple Metamodel Description (SMD) language. SMD is quite similar to EMOF but contains some extensions to deal with models described in multiple modelling languages.

A metamodel kernel can be used to implement model transformations as defined in the Model Driven Architecture (MDA) but, as such, the features provided by a kernel are too basic. In many cases, it is necessary to implement a transformation engine that can be used to execute model transformations defined in a high-level model transformation language. Examples of transformation engines are the implementation of rule-based transformations presented in [6] or the transformation tool for relational mappings presented in [1]. Although these tools implement different transformation languages, they share many common features related with basic model management that could be implemented by an independent kernel.

The discussion about model transformations is quite often centred on model transformation languages. However, we consider that is equally important to discuss the features and development of model transformation engines and metamodel kernels that support interesting model transformation languages. Coral is an attempt to seek practical and theoretical issues in these topics and provide a working solution for researchers.

In the rest of the paper, we present the more important features of Coral, and how it manages to create a flexible approach to querying and manipulating models that can be used to implement a transformation engine.

2 A Dynamic Metamodelling and Modelling Tool

The Coral kernel is based on few but important principles. The most fundamental is the notion of being metamodel-independent, i.e., metamodels and models can be created at runtime. In several other modelling tools, there is only one or a few static metamodels to choose from; in Coral, metamodels are first-class citizens. Large parts of Coral try to be as ignorant of the underlying metamodel as possible, and several interesting algorithms and problems arise from this.

Even though Coral can create any metamodel at runtime, there are still some predefined metamodel elements (*metaelements*) for primitive datatypes such as integers, strings and floating-point values.

A Coral modelling language is represented as a model in a language called the Simple Metamodel Description language. SMD can be seen as analogous to MOF. Whenever Coral needs the definition of a modelling language, the SMD model for this language is loaded dynamically and converted to a metamodel internally. Naturally this arrangement creates a chicken-and-egg problem in practice with respect to the SMD language in itself. This is circumvented by bootstrapping Coral with a hand-written SMD metamodel which is statically linked into Coral.

An interesting feature of the dynamic nature of the metamodelling layer is the concept of importing the contents of one metamodel into the namespace of another metamodel. This allows us to form hierarchies of metamodels. For example, a tool vendor uses its own namespace as the combination of UML 1.4 and XMI-DI 2.0. In Coral, this compatibility is achieved by creating the metamodels *separately* and then importing their contents into a third metamodel.

2.1 Mutually Independent Property Characteristics

In our opinion, the expressive power of metamodels does not come from the actual metaelements, but rather from the different characteristics of the interconnections between metaelements. An element's possible connections (*slots*) are described by its metaelement's *properties*. Two properties can be connected together to form a bidirectional meta-association.

In Coral, a property consists of several characteristics and describes the restrictions for each slot. Using a combination of characteristics several common constructs can be modelled, as well as more esoteric ones. It is important to notice that this part is static in Coral, i.e. users cannot change what characteristics are available. The various characteristics are:

- a *name* for convenience
- a *multiplicity* range $[l,u]$ defining how many connections to instances of the target the slot (instantiated property) should have to be well-formed. Common values are $[0..1]$ for an optional element, $[1..1]$ for exactly one, $[0..*]$ for any amount and $[1..*]$ for at least one element
- a *target*, telling what the type (metaelement) of every element in the slot must be
- a boolean *ordered* telling if the order of the elements in the slots is important and must be kept
- a boolean flag *bag* telling if the same element can occur several times in a slot
- a boolean flag *unserialisable* telling if corresponding slots should not be serialised when saving a model
- an optional *opposite*, giving the opposite property for bidirectional connections
- a link *type* enumeration value {association, composition } describing an ordinary connection or describing ownership, respectively.

The characteristics *unserialisable* and *anonymous* need more careful explanation. An unserialisable property means that the contents of the corresponding slots are not saved to an output stream. This is useful when elements in file A reference elements in file B, but without the elements in B having to know anything about file A. This occurs when creating models that resemble "plugins"; we are not sure what plugins are available and we do not want to change the main file every time something is added or removed. Instead, the available plugins are loaded at runtime and bidirectional connections are created, even though they are not serialised at both ends. Arguably the usefulness of the characteristic in this case is specific to the way current filesystems work using files as independent streams of bytes. A filesystem acting more like a database would not share the benefits from the unserialisable characteristic.

Anonymous properties provide fully bidirectional meta-associations between two metaelements, even though the meta-association was unidirectional at first. This is useful in cases where a language was not designed to be used together with another language. An example is a project management language (PML) keeping track of developers, bugs, timelines and several UML models. Since UML models do not know about PML, only unidirectional connections from PML to UML would be feasible, thus rendering any navigation from UML models to PML models impossible. But Coral automatically creates an anonymous property (with a private, nonconflicting name) at

run-time from UML models to PML models, and thus it is indeed possible to navigate from any UML model to the corresponding PML model(s). The PML models can be saved in an XMI file separate from the UML file. Using the support from XMI for interconnecting model elements across files, the PML elements can still reference the UML elements.

Anonymous properties are necessarily also unserialised, since otherwise ordinary UML tools would not be able to read the model file with nonstandard slots. Anonymous properties provide an excellent way to combine models without changing the languages.

Most notably, the list is currently missing new characteristics from MOF 2.0, property *subsetting* and *derived unions*. These are important characteristics but have not been added to Coral yet. Otherwise, it is worth noting that the characteristics aim to be as mutually independent as possible. This has the benefit that very complex definitions can be modelled.

3 Metamodel Kernel API

These features of the Coral kernel can be accessed using a programming interface or API. The Coral kernel is implemented in C++, but we have created an interface for the Python programming language. It should not be difficult to support other programming languages such as Java, since the bindings for the specific programming languages are created using the SWIG tool.

Python is a highly dynamic expressive language which is easy to learn. Using Python, the interface to query models is very close to OCL [2], but with several methods added to also modify the model. Notably, model transformations can be written as Python programs with separate phases for preconditions, query and modification and postconditions. Support for transactions as well as checking of well-formed rules means that an illegal transformation can be rolled back, leaving the user with the original model. Examples of a rule-based model transformer can be found in [6].

Arbitrary scripts and well-formedness checks can be used to keep the design and evolution of a system within a predefined process or methodology. A success story is Dragos Truscan's work [7] on relations between data flow diagrams and object diagrams. It presents "an approach to combine both data-flow and object-oriented computing paradigms to model embedded systems." The work is fundamental for designing complex embedded systems since there is often a need to switch between the two paradigms. The design relies on an SA/RT metamodel for the data flow and the UML 1.4 metamodel for object and class diagrams. Python scripts are heavily used for the transformations between the domains.

In the future, using models also as the primary artefact for transformations using e.g. the upcoming OMG Query-View-Transform (QVT) [3] standard could be possible.

The scripting interface provides a highly flexible environment for automatic model generation, querying and transformation. Metamodels support predefined operations on specific elements, and there is no need to explicitly compile any scripts as they can be loaded on-the-fly from within Coral.

4 Conclusions

We have presented the Coral kernel, a metamodel-independent tool. In Coral, metamodels are first-class objects that can be created at runtime. This allows us to support new modelling languages without recompiling or even restarting Coral. Furthermore, we have made interesting progress in the dynamic combination of metamodels. Effort has been placed into making a Python-friendly interface to facilitate easy scripting for model transformation. However, it is possible to support other programming languages.

There are two ways to evaluate the Coral kernel: as a research tool or as a development tool. As a research tool, we are interested in a flexible software library that can be used to quickly create small prototypes to test new ideas in modelling technology. If we consider Coral a development tool we are interested in a robust and efficient library based on current standards.

Currently, Coral is definitely geared towards a research tool. The decision of using its own metamodelling language instead of MOF is an example of this. However, Coral is able to manage large models created using commercial tools efficiently. It supports the UML 1.1, UML 1.3, UML 1.4 and UML 1.5 metamodels. Model interchange can be accomplished using XMI 1.x, XMI 2.0 and XMI-DI [5] for diagram interchange.

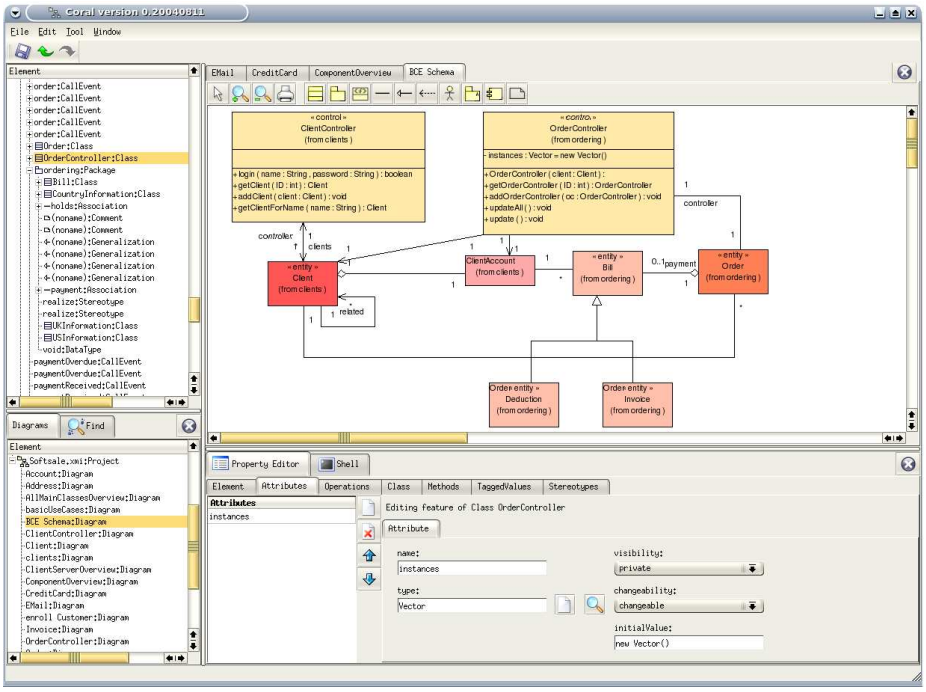

Fig. 1. Coral can render diagrams stored in XMI-DI 2.0, for example models created with Gentleware's Poseidon. This screenshot shows the Softsale example model imported from Poseidon.

As an example of the compatibility and scalability of Coral, Figure 1 shows Coral rendering a UML 1.4 model with more than 20.000 model elements created with a commercial tool. The figure also reveals one of the interesting discoveries we had while developing Coral: Although the tool does not follow the OMG standards at the Metamodelling level, it is fully compatible at the modelling level. That is, it is possible to develop a fully UML compliant editor and transformation tool without using MOF or the UML 2.0 infrastructure as basis for the tool.

Coral source code is available under an open source license at http://mde.abo.fi.

References

1. D. Akehurst, S. Kent, and O. Patrascoiu. A relational approach to defining and implementing transformations between metamodels. *Software and Systems Modeling*, 2(4), December 2003.
2. OMG. Object Constraint Language Specification, version 1.1, September 1997. Available at `http://www.omg.org/`.
3. OMG. MOF 2.0 Query / Views / Transformations RFP. OMG Document ad/02-04-10. Available at www.omg.org, 2002.
4. OMG. Meta Object Facility, version 2.0, April 2003. Document ad/03-04-07, available at `http://www.omg.org/`.
5. OMG. Unified Modeling Language: Diagram Interchange version 2.0, July 2003. OMG document ptc/03-07-03. Available at `http://www.omg.org`.
6. Ivan Porres. Model Refactorings as Rule-Based Update Transformations. In *Proceedings of the UML 2003 Conference*. Springer-Verlag, October 2003.
7. Dragos Truscan, João Miguel Fernandes, and Johan Lilius. Tool Support for DFD-UML Model-based Transformations. In *Proceedings of the ECBS 2004 Conference*, May 2004.