The Coral Modelling Framework

Marcus Alanen, 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

The technology to fully support model-based software development approaches such as OMG's model driven architecture is still in its infancy. There is still a lot to be learned about how a modelling framework should be constructed and used to enable using models as the only description of a software under construction and model transformation as the basic step in software development. 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. In this paper, we present our work on Coral, a generic open source tool for modelling, and how we have been able to experiment with novel ideas in modelling. The research area of modelling tools is important as solid frameworks are required to empower software developers to actually use the benefits of a model driven architecture.

In the next section, we go through the more important features of Coral, and how it manages to create a flexible approach to querying and manipulating models. We finally conclude with some remarks on what features we consider important in a modelling framework.

2 Coral Features

Todays de facto modelling environment adhere to the specifications defined by the Object Management Group (OMG) more or less closely simply because of practicality; they have the broadest audience. OMG defines a modelling environment

in four layers which are, from highest to lowest, the metametamodel, the metamodel, the model and the runtime layers. Each layer serves as a description of what can be accomplished in the layer immediately below it, akin to class definitions providing which objects can be created in an object-oriented language. The metametamodel is fixed to the Meta Object Facility (MOF) [6]. The most widely known metamodel is the Unified Modelling Language (UML) [3].

While UML has created useful ways to describe software systems in a visual language, the semantic preciseness of metamodelling and modelling constructs has been lacking. Unfortunately the various specifications do not help. Coral is an attempt to seek practical and theoretical issues in metamodelling and modelling standards and outside standards and provide working solutions. In the next subsections various aspects of Coral are brought forward that show how it fits as a researcher's tool.

2.1 A Dynamic Metamodelling and Modelling Tool

The Coral framework is based on few but important principles. The most fundamental is the notion of being metamodel-independent, i.e., Coral positions itself at the top of OMG's layers creating a metametamodel interface. Using it, 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. The Coral metametamodelling layer is static, which has the implication that users cannot experiment with new metametamodelling techniques. While the OMG standards are self-referencing and self-defining, this is usually not possible to do in a software program, so there is a limitation to the level of flexibility that can be constructed.

Even though Coral can create any metamodel at runtime, there are still some fixed metamodel elements (*metaelements*) for primitive datatypes such as integers, strings and floating-point values. To represent diagrams, the XMI-DI [7] metamodel is supported.

As the de facto serialisation format for *models* is the XML Metadata Interchange (XMI) [5], no metamodels (as defined in Coral) per se can be loaded or saved. Trivially this is rectified by noting that every metamodel can be interpreted as a model which therefore must have a metamodel. This metamodel is called the Simple Metamodel Description language (SMD) in Coral, and then metamodels can be represented as SMD models. SMD can be seen as analogous to MOF. The SMD metamodel is used to load models, from which metamodels can be created using a special routine, *model2metamodel*. This arrangement can be seen in Figure 1. Similarly, the metamodel can be transformed back to a model (which has SMD as its metamodel) using *metamodel2model*. This is portrayed in Figure 2.

Naturally this arrangement creates a chicken-and-egg problem in practice with respect to the SMD language. This is circumvented by bootstrapping Coral with

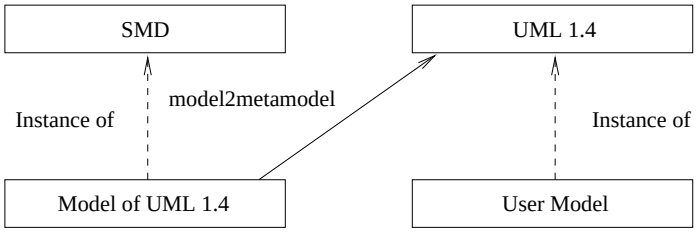

Figure 1: Lifting a model to the metamodel layer. The Simple Metamodel Description language (SMD) is statically linked into Coral. Using it, other metamodels can be loaded as models and then transformed into metamodels.

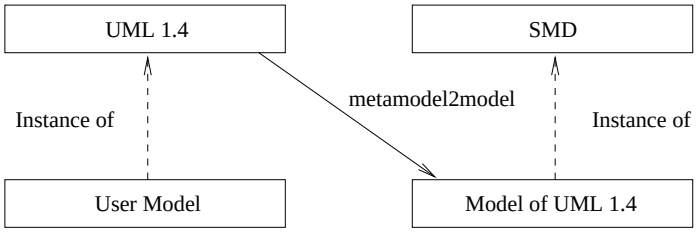

Figure 2: Lowering a metamodel to the model layer, the opposite operation of Figure 1.

a hand-written SMD metamodel which is statically linked into Coral.

At the moment, there is no explicit support for the lowest layer in Coral. However, this is not a problem, since a user can transform (parts of) their model into an SMD model, which can in turn be transformed into a metamodel *M*. Thus, objects in the runtime layer can be simulated by a model with a suitable metamodel. Having generic support for two layers (the model and the metamodel) seems to be enough, although we do not have the empirical evidence to support this argument yet.

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 the current metamodel. This allows us to form hierarchies of metamodels. For example, a tool vendor uses its own namespace for 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.

The modelling layer provides support for transactions by recording changes to models. Each transaction consists of an ordered list of commands. Trivially, a transaction can be undoed and redoed by traversing the list backwards unexecuting the commands or by traversing the list forwards executing the commands. Independent transaction observers can be attached (in a subject-observer design pattern). The graphical subsystem uses the transaction facility to automatically

update the graphics when the model is changed by e.g. a script.

2.2 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, but are free to combine them in arbitrary ways. The various characteristics are described below.

- 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 *anonymous* telling if the property is anonymous (described later)
- 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
- an 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 characteric.

Anonymous properties provide fully bidirectional meta-associations between any metaelements even though the meta-association was unidirectional at first. This is useful in cases where a metamodel was not designed to be used together with another metamodel. An example is a project management metamodel (PMM) keeping track of developers, bugs, timelines and several UML models. Since UML models do not know about PMM, only unidirectional connections from PMM to UML would be feasible, thus rendering any navigation from UML models to PMM models impossible. But Coral automatically creates an anonymous property (with a private, nonconflicting name) at runtime from UML models to PMM models, and thus it is indeed possible to navigate from any UML model to the corresponding PMM model(s). The UML models can then be saved in one file, and the PMM models in another; the XMI standard for model interchange contains facilities for linking across files. Anonymous properties are necessarily also unserialised, since otherwise ordinary UML tools would not be able to read the UML model file with nonstandard slots.

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.

2.3 Python Scripting Interface

An important feature of a modelling framework is its ability to query and modify models at runtime, preferably both interactively and using a script. In Coral, this has been achieved by creating Python wrappers around the Coral C++ core. 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 [9].

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 [10] 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) [4] standard could be possible.

The scripting interface provides a highly flexible environment for automatic model generation, querying and transformation. SMD 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.

2.4 Miscellaneous

Coral supports XMI 1.x and XMI 2.0 input and XMI 1.2 and XMI 2.0 output well. It also has support for more esoteric features such as interfile relationships although these are not too heavily tested. In practice it has good support for reading XMI generated by other common commercial tools.

Coral currently comes with the UML 1.1, UML 1.3, UML 1.4 and UML 1.5 metamodels, but interactive graphical support is lacking. How the presentation (graphics) of a metamodel is defined and drawn has not been as thoroughly developed by OMG as the abstract metamodels, which unfortunately is reflected in Coral as well; support for every diagram must be written explicitly, although there is work-in-progress to use models to generate graphical interfaces.

Coral has support for difference calculation between two models, based on [1], but this has not been integrated into the graphical environment.

Currently, Coral and its predecessor SMW [8] have been used as modelling and metamodelling tools by two PhD students and several Master's Theses have been written on their plugins and subsystems.

3 Conclusions

We have presented the modelling tool Coral, its main features and primary principles, as well as some novel concepts that it uses to address practical problems in the creation of such a tool. Coral is still work-in-progress, but is used by other members of our model driven engineering group more and more.

Coral works as a metamodel-independent tool. For this to be possible, metamodels must be treated as first-class citizens that can be created at runtime. Furthermore, we have made interesting progress on the characteristics of metaelements' properties in the form of anonymous properties, which greatly aid linking

together metamodels. Effort has been placed into making a Python-friendly interface to facilitate easy scripting. However, there is a lot of on-going work with e.g. the interactive part of Coral.

References

[1] Marcus Alanen and Ivan Porres. Difference and Union of Models. In *Proceedings of the UML 2003 Conference*, October 2003.

[2] OMG. Object Constraint Language Specification, version 1.1, September 1997. Available at `http://www.omg.org/`.

[3] OMG. OMG Unified Modeling Language Specification, version 1.4, September 2001. Available at `http://www.omg.org/`.

[4] OMG. MOF 2.0 Query / Views / Transformations RFP. OMG Document ad/02-04-10. Available at www.omg.org, 2002.

[5] OMG. XML Metadata Interchange, version 1.2, January 2002. Available at `http://www.omg.org/`.

[6] OMG. Meta Object Facility, version 2.0, April 2003. Document ad/03-04-07, available at `http://www.omg.org/`.

[7] OMG. Unified Modeling Language: Diagram Interchange version 2.0, July 2003. OMG document ptc/03-07-03. Available at `http://www.omg.org`.

[8] Ivan Porres. A Toolkit for Manipulating UML Models. Technical Report 441, Turku Centre for Computer Science, January 2002. Available at `http://www.tucs.fi/`.

[9] Ivan Porres. Model Refactorings as Rule-Based Update Transformations. In *Proceedings of the UML 2003 Conference*, October 2003.

[10] 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.