A toolkit for model manipulation

Ivan Porres

TUCS Turku Centre for Computer Science, Åbo Akademi University, Department of Computer Science, Lemminkäisenkatu 14, FIN-20520 Turku, Finland; E-mail: ivan.porres@abo.fi

Received: 1 February 2002/Accepted: 2 May 2003
Published online: 30 October 2003 – © Springer-Verlag 2003

Abstract. We present a toolkit to develop scripts to process software models. It can be used to create applications to check, transform and generate derived artifacts from a model. The toolkit is based on the current OMG standards and it can be used with the Unified Modeling Language (UML) and other user-defined languages based on the Meta Object Facility.

Keywords: UML – Transformation – Metamodeling – Tools – Scripting

1 Introduction

Software models can be created just by jotting on a piece of paper or a whiteboard. This is a great way to represent and communicate our thoughts during a discussion or a brainstorming session. However, thanks to recent modeling languages such as the UML, we can now represent the most relevant information in a software project in a format that can be easily processed and transformed by a computer system.

The Model Driven Architecture (MDA) initiative [3] has increased the interest on methods and tools to transform software models. The goal is to use models as the main artifact in software development. Using advanced tools, a model describing the requirements and problem domain of a system to be built can be stepwise transformed into a detailed implementation model that can be executed. Another application of model transformation is model refactoring [27]. In this case, we do not transform a model to make it more concrete but to simplify and improve its design.

Actually, model transformation is just one of the possible uses of a model-driven software development tool. There are many other applications that do not transform an existing model but generate a derived text-based artifact. This is the case of generation of program code [26], tests cases [18], metrics [28], a specification in a formal language [11] or an input model for a verification tool [15] or a performance analysis tool [5]. There are other applications that do not even generate new artifacts. An example is to check if a model is consistent with respect to the rules of its modeling language [24] or with respect to user-defined rules such as architectural, design or implementation guidelines.

In the recent years, many UML tools have appeared in the market providing all-in-one solutions for model-driven software development. Nevertheless, there is still a need to extend a development toolset and create small tools customized to our particular application domain, software development process, target platform or testing procedures. This has been the case in source code-based development projects and we expect the same need will arise in model-driven development projects.

Some UML tools offer as an extension mechanism a low-level application programming interface (API) or a high-level scripting language. For example, Borland’s Together Control Center can be extended by creating Java applets that interact with the tool via its Java-based API. Rose from Rational implements an interpreter for a dialect of the BASIC programming language and provides the Rose Extensibility Interface to manipulate models within the tool. These approaches are satisfactory for advanced users that want to create add-ins to enhance the graphical CASE tool. However, we think that in many other scenarios, using the extension mechanism of a full-featured interactive tool can present some disadvantages such as high license costs, partial support for the UML standard or interoperability issues with other tools and programming languages.

An alternative is to use a stand-alone scripting language to create our own tools to implement model transformations, generate derived artifacts and check model consistency. Popular scripting languages such as Perl [30], Python [29] or Tcl [22] have proven extremely useful...
in source code-based development projects to develop short utility programs in a short time. These languages are often interpreted, have a dynamic type system or no type system at all and promote a fast development cycle. In many cases, the interpreter and development tools are open source and there is a large library of contributed modules and utilities freely available and freely distributable.

This article discusses what are the relevant features in a scripting language to process models and presents our own implementation of a scripting toolkit based on these features.

We proceed as follows. The next section is an introduction to the basic concepts of metamodeling and describes how a model can be stored in a computer system. Section 3 describes what are the main features that we consider relevant in a scripting language to process models and discusses related work. Sections 4 and 5 describe SMW, our own toolkit, while Sect. 6 describes a small example application of SMW. Finally, the last section contains some concluding remarks. Sections 3 and 5 are intended for those readers interested in the study and development of a scripting language, while those who are interested in developing tools using a scripting language will find Sects. 4 and 6 the most interesting.

2 Modeling languages and metamodels

In order to understand how to construct programs that process and transform models we need first to understand how models are organized in a computer system. For this purpose, we will use a small modeling language of our invention that is much simpler than UML.

Our example language is called FSM and is a language for describing finite state machines. A state machine has a finite number of states and transitions. Each transition connects two states and it can be triggered by a token. The set of tokens in a state machine is called the alphabet. One or more states may be marked as accepting states, while one of the states is marked as initial. These concepts are described as a class model in Fig. 1. We call this kind of diagram a metamodel. This diagram is similar to the standard UML metamodel shown in [20].

A metamodel describes the abstract syntax of a modeling language. Each class in a metamodel describes a model element, i.e. a concept or abstraction in our modeling language. Each class may have a number of attributes. An association connecting two classes represents a symmetric relation between these elements. In our example language, the fact that each state machine has an initial state is represented by the association named initial. We use the generalization relationship to define a model element as a specialization of other model elements. In our metamodel, an accepting state is a specialization of state.

Figure 2 shows an example model in the FSM language. The model is represented using three different notations. The diagram at the left uses a syntax that is specific for our language for finite state machines. Most designer would prefer this notation since it is a fully visual language where each concept is described using a different icon.

The diagram at the right shows the same model, this time represented as an object graph. Each node of this graph represents an instance of a model element and each arc an instance of an association. Using this representation, there is a one-to-one mapping from the concepts described in the metamodel and the object graph representing a model.

Finally, we can also represent the same model as a XMI document. XMI is an OMG standard [21] for model interchange. It is based on XML and can be used to represent any modeling language, i.e. it is not limited to UML. XMI is the preferred notation to exchange models between programs since XMI documents are portable and easy to parse. Some XML parsers, such as those following the DOM interface, can generate an object graph based on the information stored in a XML document and recreate a XML document based on an object graph.

In the rest of the article, we will represent a model as an object graph since this simplifies the construction and description of programs and algorithms that process models. We will use the terms class, attribute and association to describe classes in a metamodel, attributes in a metamodel and associations in a metamodel. These concepts are also called meta-classes, meta-attribute and meta-association and should not be confused with the