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 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 an 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
3 Scripting for model manipulation

A scripting language is a programming language that is either interpreted or compiled quickly on the fly and that can be used interactively via a command prompt. The design of most scripting languages sacrifices the execution speed provided by system programming languages in favor of a more friendly language that is often tailored to a specific task.

In our case, we are interested in a language that can be used to develop quickly small applications to process and transform models. We consider that there are two fundamental choices in such language: the overall programming paradigm and the model interface level. We explain these concepts in this section while discussing related work.

**Model Interface Level.** The OMG standards follow a four-layer metamodel architectural approach to define the UML. In this approach, actual user data is defined as an instance of model, a model is defined as an instance of a metamodel and a metamodel is defined as an instance of a meta-metamodel. These concepts are explained in [20]. We can use the same classification to define how a script can access the features of a model.

In a meta-metamodel or M3-based language, there is one data type in the programming language that can represent all the model elements in the modeling language. Both the features of the modeling language and the actual models are represented as data. An example of this approach can be found in the xUMLi scripting tool [1]. In contrast, a metamodel or M2-based programming language has a different type or class for each model element in the modeling language. The features of the modeling language are represented using the structural features of the programming language. For example, in an object-oriented programming language a model is represented using classes and objects. The Java Metadata Interface (JMI) follows this approach [10].

As an example, Fig. 3 shows a code fragment in a M3-based language (left), a M2-based language using the Python programming conventions (center) and a M2-based language using the Java programming conven-
Ivan Porres: A toolkit for model manipulation

Using an object-oriented programming language seems a natural choice since the UML metamodel is an object model and UML is mostly used to describe object-oriented systems. JMI is an example of such approach. There are at least two implementations of JMI, the Netbeans Metadata Repository [9] and Novosoft NSUML [17]. These libraries are probably best suited to build large tools, including an interpreter for a scripting language for model transformation, but not to construct the actual scripts.

3.1 Language features

We have seen that there are many different approaches to define model transformations, sometimes as different as Perl and MAUDE, all of them viable in a certain context. The question now is what features we should require in a language for model manipulation to simplify the construction of new scripts and ensure their quality. We review these features in the rest of this section.

Model Transformation. Our scripting language should be able to modify the information contained in a model: update the attributes of a model element, create new elements and remove existing ones.

The Object Constraint Language (OCL) [31] is not a good language for describing model transformations since OCL is declarative and free of side effects. Although it is possible to describe a transformation in a model as a precondition and postcondition pair, this kind of specification is not always deterministic and directly executable. However, OCL is an integral part of UML. It has excellent facilities to query and navigate UML models and therefore we think that a scripting language for model transformation should embrace many of the ideas behind OCL.

We can transform a model by direct manipulation of a XMI document. In this case, modifying an attribute of an existing element or creating a new element is a straightforward operation. However, direct manipulation of XMI files can easily lead to malformed models. For example, changing an association requires updating both association ends simultaneously. In addition, deleting a model element from a XMI document requires traversing the complete document and removing all references to the original model element. The complexity of this operation is linear to the size of the model and not the number of associations of the element.

Code Generation. Many applications involve the creation of text artifacts from a model. This imposes another re-
requirement in our language: deterministic and predictable execution. This can be an intrinsic feature of the language, e.g. it is an imperative programming language, or it can be provided using a special mechanism, e.g. the monadic input/output in Haskell.

Model Navigation. Model transformation is an important topic in the MDA initiative but our experience is that a large part of the code in a script is dedicated to find information in a model and not to change it. Indeed, there are applications such as metric extraction that are built using only a combination of model queries, aggregation operations and output operations.

A scripting language should have a mechanism to find information in a model easily. It should allow finding an element by type, by the value of its attributes or by its associations. This can be implemented using a pattern matching mechanism like in XSLT or a query language such as OCL.

Support for different Modeling Languages. The most important requirement for our scripting language is that it should be able to represent and transform any UML model as defined in the UML standard. It should also support extensions to the UML standard, called profiles and our own modeling languages such as the FSM language used as example. This implies that the scripting language should provide a mechanism to easily define new modeling languages or profiles.

XMI Compliance. A script should be able to retrieve and store models using the XMI exchange format since this ensures the interoperability with other XMI-compliant tools.

However, XMI compliance goes beyond the input and output operations. Each XMI element has a globally unique identifier that should be taken into account during model transformations. Our scripting language should ensure that each model element has only one identifier during its lifetime and that this identifier is unique.

Well-Formed Models and User-Defined Constraints. The language should include a mechanism that ensures that the models constructed and transformed using a script are syntactically and semantically correct. We consider a model correct if:

- The contents of the attributes are of the correct type.
- Associations are consistent. Associations may only relate elements of the correct types. In addition, associations are symmetric, i.e. if element a is associated with b, b is also associated with a. Most programming languages only support simple references to objects, but our language should have a mechanism that ensures that both ends in an association are always synchronized.
- All well-formed rules hold. Well-formed rules are additional semantic constraints over a model. For example, there is a well-formed rule in UML that says that two attributes in the same class cannot have the same name. A transformation should fail if these rules are breached.

There is also a need to define our own constraints in a model besides the predefined well-formed rules of the modeling language. User-defined constraints can be used e.g. to enforce design guidelines such as “All class names start with a capital letter” or to describe features of the target platform or programming language, such as a constraint that forbids multiple implementation inheritance in a Java design.

We can also define the postcondition or goal of a transformation script as user-defined constraints. The execution of a script will be successful only when the constraints representing the postcondition hold after running the script.

Reflection. Reflection allows a script to obtain information about the features of a given modeling language: the name and parents of each model element and the names, types, multiplicities and well-formed rules of their attributes and associations. Reflection can be used to write generic scripts that can process models in different modeling languages or different versions of a modeling language. Example applications of reflection are the implementation an algorithm for model duplication presented in [23] and a multi-user model repository presented in [2]. These tools can work with any model since they query their structure via reflection. Reflection can also be used to create scripts that query or transform a metamodel instead of a model. For example, the mmdiff tool, distributed with SMW, compares two metamodels and prints their differences. It can be used to show what has changed from UML 1.3 to UML 1.4.

In order to provide this functionality, a scripting language should keep all the relevant information about a given modeling language and make it accessible via a M3 or meta-metamodel interface. In any case, a XML DTD cannot be used to implement a reflection interface since it does not contain all the required information about data types, well-formed rules, multiplicities and inheritance of model elements.

Transactions. A transaction is a logical sequence of changes in a model. All changes in a transaction are stored in a transaction history. This history can be reviewed, rolled back and reexecuted again. Transactions have three important applications: Define check-points to verify model correctness, implement user friendly applications that offer an “undo & redo” feature to correct mistakes from the user and record a trace of the transformations in a model. The transaction history may be used to know which model elements are created, removed and modified in each transformation. Traceability of transformations is a key requirement in the MDA initiative.
Transactions are also important when we use multiple profiles and scripts to develop a model. In this context, it is not possible to know a priori if a transformation will be successful since the model may contain arbitrary constraints defined in a profile after the script was developed. There is no general solution to this problem but at least the transaction mechanism ensures model consistency. If the execution of a script violates a constraint, the transaction mechanism may roll back the changes and leave the model in its previous state.

An implementation of a transaction mechanism may or may not preserve the atomicity, consistency, isolation, and durability properties usually associated to database transactions. A single-user model editor may not deal explicitly with isolation and durability of transactions while a multi-user model repository should definitely implement these four properties.

**Simple Programming Model and Clean Syntax.** Finally, we want the scripts to be as simple, easy to understand and concise as possible, so they can be reviewed and maintained easily. This is a common requirement in any program, but it is especially true in one usage scenario of the scripting language.

Researchers and developers of new model-based development methods have the need to describe model transformations in a concise and self-explanatory notation. In this case, is not enough to present an algorithm describing a transformation since there is the question of how to ensure that the implementation follows the algorithm. Our long-term goal is to create a scripting language that is so concise and easy to understand that we can use an actual executable script instead of an algorithm to explain a transformation in full detail.

Summarizing, a scripting language for model manipulation should be able to represent and manipulate models using a complete interface defined by the standard metamodels. It can be used to create applications for model transformation and code generation. It should also be possible to group a sequence of changes into a transaction, keep a history of transactions and it should enforce the typing, well formed and consistency rules of the modeling language after each transaction.

In the next section, we describe SMW, our own toolkit that has been implemented following these requirements and features. It is not our intention to discuss in full detail how SMW has been constructed but to show how its main features are integrated in a simple programming model.

### 4 The SMW toolkit

SMW is a collection of tools to manipulate models based on the requirements defined in the previous section. We have decided to create an extension module for the object-oriented programming language Python. The objective is to provide an implementation of the main features needed in model transformation while maintaining all the benefits provided by an easy to use and well-supported scripting language.

Since the SMW toolkit is based on the Python programming language we will first introduce this language and then show how we can use it to represent software models. We only present the most basic features of the language that are needed to understand the examples in the article. Readers interested in the language can find a complete introduction in [16].

Python is an interpreted programming language developed by van Rossum [29]. Python programs are compiled on the fly into bytecode and its interpreter is portable and has been ported to all major platforms. It has an elegant syntax and it is easy to learn and read. Python is a class-based object-oriented language so it is well suited to represent a metamodel in terms of classes and attributes. Python syntax is slightly different from C++ or Java, albeit simpler and clearer. Statements are terminated by new lines. Code blocks are denoted by the indentation, as in the Occam language. This reduces the number of special symbols in the code and improves readability.

The # symbol is used to add remarks. The assignment is denoted by one equal symbol, while equality is denoted by two. None is the null reference. Boolean expressions are similar to the C programming language: 0, None or an empty list or dictionary evaluates to false, while everything else evaluates to true. `assert(b)` raises an exception if `b` is false.

If `A` is a Python class, the expression `A()` returns a new instance of `A`. Function application is also denoted with parenthesis and parameters may be denoted by position or by name.

Python code can be organized into modules that are conceptually similar to Java packages. We can import a complete module using the `import` keyword or just a specific definition inside of a module by using the construction `from module import name`. A module can contain other modules. In this case, we use a dot to create the full name of a module.

A list is denoted with brackets: `l=["alpha", "beta", "gamma"]` assigns a list containing three strings to `l`. A dictionary is an associative array. We can use any immutable value as an index. Dictionaries are denoted with curly brackets: `colors={"red": "rojo", "green": "verde", "blue": "azul"}` defines a dictionary that maps color names from English to Spanish. We have extended these basic data structures to support other kinds of containers needed to represent a model in SMW.

#### 4.1 Collections

The OCL defines several container types or collections to represent associations and the result of model queries. SMW implements similar collections and supports all the OCL queries such as `forall`, `exist`, `select` or `collect`. The SMW collections are called `MMSequence`, `MMBag`, and
MMSets. These classes implement collections with ordered bag, unordered bag and set semantics.

The following example creates a set containing the integers from 1 to 5 and then selects only the odd ones. We define the predicate used to select elements in the collection using a lambda function. In the example, we use the Python prompt symbol to indicate that these commands are typed and executed interactively.

```python
>>> c = MMSet([1, 2, 3, 4, 5])
>>> c.select(lambda n: n % 2 == 1)
[1, 3, 5]
```

OCL does not provide operations to update the contents of a collection since this language is free of side effects. SMW provides these operations to allow model transformations. The following table includes all the operations for collections with side effects. In the table, `c` stands for a collection (an instance of `MMCollection`), and `o` for any other object.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Side-Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c.insert(o)</code></td>
<td>Inserts <code>o</code> in the collection</td>
</tr>
<tr>
<td><code>c.append(o)</code></td>
<td>Inserts <code>o</code> at the end of the collection</td>
</tr>
<tr>
<td><code>c.sort()</code></td>
<td>Sorts the elements in the collection</td>
</tr>
<tr>
<td><code>c.remove(o)</code></td>
<td>Removes <code>o</code> from the collection</td>
</tr>
</tbody>
</table>

These operations are available for all kind of collections. However, the results of `sort` are only observable in a sequence and `insert` and `append` have the same effect in a set and a bag.

SMW supports all the OCL queries for collections, but in some cases, the syntax is different. The most important change is that there is no direct support for the OCL `->` operator, but there is a work-around. For model elements, we can use the method `asSet()` that returns a set containing that element. For collections, we can use the `->` operator implicitly by using a dot, as in OCL. Another important change is the logical implication, which is implemented as a function instead of as an operand. These changes have been introduced to accommodate the existing OCL constructions to the Python syntax. They are summarized in the following table:

<table>
<thead>
<tr>
<th>OCL</th>
<th>SMW</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a implies b</code></td>
<td><code>implies(a, b)</code></td>
</tr>
<tr>
<td><code>o -&gt; query(p1,...,pn)</code></td>
<td><code>asSet().query(p1,...,pn)</code></td>
</tr>
<tr>
<td><code>c -&gt; query(p1,...,pn)</code></td>
<td><code>c.query(p1,...,pn)</code></td>
</tr>
</tbody>
</table>

Once we know how SMW represents collections of objects we can study how to represent a complete model.

4.2 Representing models in Python

SMW scripts can access the features of a model using a M2 or metamodel-based interface. SMW represents each element in a modeling language, i.e. each metaclass in a metamodel, as a Python class. Each element in a model is represented as an instance of the corresponding Python class.

The following code shows how to create the model from Fig. 2 as Python objects. The script requires a Python module containing the definition of the FSM metamodel. We describe how to create this kind of module in Sect. 5.2.

```python
from FSM1 import *
A=Token(name="A")
s1=State(name="S1")
s2=AcceptingState(name="S2")
s3=State(name="S3")
t1=Transition(source=s1,target=s2,trigger=A)
t2=Transition(source=s2,target=s3,trigger=A)
sm=StateMachine(name="Example",
    alphabet=[A],
    states=[s1,s2,s3], initial=s1,
    transitions=[t1,t2])
```

The first line of the script imports a module containing the FSM language. There is a mandatory version number in each metamodel module. The module representing the first version of our language for state machines is called FSM1. The rest of the script instantiates the model elements one by one using the classes defined in the metamodel and standard Python idioms.

Since we store a model using Python objects we can use all the features of this programming language to traverse, extract information and transform a model. We can iterate through the elements in an association end like a usual Python container. This allows us to write code in an imperative style using the Python for statement. In the following example, we show how to print a description of each transition in a state machine. The for loop in line 13 iterates through all the transitions in a machine. In each iteration of the loop, the variable `t` will contain a different element in the association end.

```python
for t in sm.transition:
    print "Tr. from", t.source.name,"to", t.target.name
```

A model transformation is also implemented using the standard Python idioms. The following script reverses all the transitions in our model: it swaps the source and target state of all the transitions owned by the state machine:

```python
for t in sm.transition:
    t.source, t.target = t.target, t.source
```

The toolkit maintains the consistency of the associations. We can see in Fig. 1 that there is an association between Transition and State with role names incoming and target. If state `s` has an incoming transition `t`, `s` is the target of `t`. After executing the previous script, `t1` is an incoming transition of state `s1`.

```python
>>> s1.incoming==[t1]
```

This query returns one, which is true in Python. Since an association end is a collection, we can query and navigate the associations using OCL-like queries. For example, we can check if all states in the state machine from Fig. 2 are named “S1”.

```python
>>> s1.name=="S1"
```
4.3 Well-formed rules

SMW has two mechanisms to define model correctness. First, the classes and associations in a given metamodel define a type system that is enforced by the toolkit. An attribute or an association end can only contain elements of a given type and its specializations and in a number specified in its multiplicity constraint.

If this basic type system is not enough, we can use predicates to introduce additional constraints in a model. A constraint or well-formed rule is defined in SMW just as a normal Python function. This function should accept a model element as a parameter and return true if the rule holds for that element. We can use the `addWellFormedRule` function to define the new rule. This function accepts as parameters a model element to bind the rule to, a name or a human-readable description for the rule and the function representing the rule.

As an example we can define two new well-formed rules for the FSM language. The first rule says that all tokens in the alphabet should be unique.

```python
>>> sm.transition.trigger.name
['A', 'B']
```

The second rule checks that the name of each state is unique.

```python
>>> sm.state forall(lambda s: s.name=='S1')
0
```

It is possible to define as many rules as needed. Often, well-formed rules are used to define a profile of an existing modeling language. In our example, we can add a second rule to limit our language to model only deterministic state machines. A state machine is non-deterministic if there are two transitions triggered by the same token emanating from the same source state.

```python
addWellFormedRule(
    StateMachine, "Deterministic",
    lambda sm : sm.transition forall(    
        lambda t1,t2:t1==t2 or t1.name==t2.name)
)
```

It is possible to define new rules by adding them to the model, e.g.,

```python
addWellFormedRule(
    StateMachine, "UniqueTokens",
    lambda t1,t2: t1!=t2 or t1.name!=t2.name)
```

We can combine queries so the result of a query is used as the input of another query. For example, the next query returns the names of the tokens that may trigger a transition in the state machine.

```python
addWellFormedRule(
    StateMachine, "Deterministic",
    lambda sm : sm.transition forall(    
        lambda t1,t2:t1==t2 or t1.name==t2.name)
)
```

4.4 Transaction manager

A transaction groups a sequence of updates in a model. A model update may be the instantiation of a new model element or an update of an attribute or an association end in an existing model element. A transaction defines a logical transformation in a model.

Using the transaction manager is completely transparent for the script developer. A transaction starts with a call to the `beginTransaction` method of the transaction manager and ends with a call to the `endTransaction` method. The model can be modified in between these calls as usual. A model element reports all changes in its state to a transaction manager as soon as is created. After the transaction is finished, the transaction manager stores all the updates in the model in a transaction history object.

The SMW class `Project` keeps a history of transactions as described in [12]. The communication between the transaction manager and the model elements is implemented using the metaprogramming features of Python. In the following example, we create a FSM State element and we change its name inside a transaction. Then, we revert and commit again the transaction so we can assert that the name changes.

```python
from smw.Project import *  
from smw.metamodel.FSM1 import *
prj=Project()     # Create a project
s=State()    # It is possible to create objects and modify them
prj.history.beginTransaction() # Begin a transaction
s.name="Open" # out of a transaction
prj.history.endTransaction()   # Finish the transaction
assert(s.name=="Open")        # We get the previous name
prj.history.undo()             # Undo the last transaction
assert(s.name=="Close")       # Voila!
```

Usually, SMW checks whether a model is well formed when the `isWellformed` method is invoked explicitly. This is done by the script developer. To break the well-formed rules during a transformation. In the example in Sect. 4.2, the objects in the model are created by one by one. Just after the creation of `S1` (line 3), the state is not connected to any state machine but the FSM metamodel shows that each state should always have one owner. At this point of the execution of the script the model is not well formed. However, we should require that the model is
well formed once the execution of the script is completed. Therefore, it is often preferable to check the well-formed rules after a series of changes, e.g. a transaction, and not after each individual update.

To enforce well-formed rules after a series of changes we can use strict transactions. In this case, the transaction manager checks if the elements that appear in the transaction history are well formed before committing the changes. If a transformation breaks a well-formed rule, the transaction manager undoes all updates performed during the offending transaction. This mechanism is transparent for the programmer and executed automatically when calling the endTransaction method. The transaction manager only verifies the well-formed rules of the elements that have been updated during the transaction so this approach scales up to large models.

4.5 XMI

SMW implements the XMI standard to load and save models from a file. Unfortunately, we have discovered that XMI does not ensure perfect interoperability between tools. There are several versions of the XMI standard and several versions of the UML metamodels. Different tools and even different versions of the same tool generate different XMI documents for the same model. As an example, the Together Control Center tool version 6.0 can generate seven different kinds of XMI documents from the same model.

However, the main drawback of XMI as an exchange format for UML is that the UML interchange metamodel does not define the format to represent presentations elements. That is, the exchange metamodel defines how to store the model elements, but not their graphical representation in a diagram, position, size, color, etc... This is an important drawback if we want to use our toolkit to modify the presentation of a UML model, e.g. to implement a graph layout algorithm. However, in many applications, the appearance of the diagrams is irrelevant. This is the case of scripts for e.g. code generation, test case generation, metrics extraction and model analysis. In these applications, all the needed information is contained in the XMI document.

The XMI standard states that each model element may have a globally unique and permanent identifier. This identifier can be used e.g. to track an element in a repository. SMW automatically maintains the XMI identifiers of the model elements following these rules: Each new element is assigned a unique identifier as soon as it is created. Updating the attributes of an existing element does not alter its identifier. If we copy an existing element, the copy gets a new identifier.

Loading and saving XMI documents in SMW is accomplished by the two main functions of i/o module. loadModel(url) loads a model from url and returns its root element (the first element that appear in the XMI.content section) and saveModel(url,model) stores the model. The XMI file shown in Fig. 2 was generated from the model created in Sect. 4.2 by executing the following script:

```python
13 from smw import io
14 io.saveModel("example.xml", sm)
```

SMW does not use a DTD document to generate a XMI document since it extracts all the necessary information about the structure of the XMI document from the metamodel module. In the next section we describe how to create new metamodel modules, i.e. how to add support to new modeling languages.

5 SMW metamodels

A SMW metamodel module defines a modeling language. SMW supports multiple modeling languages and all the SMW features such as transactions, well-formed rules and XMI support are available for all languages. This is possible since all the SMW modeling languages are defined using the same metamodeling language. A metamodeling language is used to define a modeling language and defines the structure and semantics of a metamodel.

The OMG standards use the Meta Object Facility (MOF), defined in [19], and OCL as metamodeling languages. MOF is used to define the UML metamodel while OCL is used to define well-formed rules in a metamodel.

5.1 Simple metamodel description

The Simple Metamodel Description (SMD) is the subset of MOF understood by our toolkit. It only contains the most basic features that we consider essential in the definition of a language: packages, elements, attributes, generalizations, associations, compositions, enumerations and constraints. SMD cannot represent all MOF models, but it can be used to describe modeling languages as complex as UML 1.4. SMD is implemented by the MetaMM module that represents the meta-metamodel layer in SMW. (Fig. 4).

A SMD model is defined as a text file containing a sequence of definitions using this syntax:

- `Package(name: String)`
- `Enumeration(package:String,name:String, elements:Sequence of String)`
- `Element(package: String, name: String, parents: Sequence of String, attributes: Sequence of Attribute)`
- `Attribute(name: String, type: String)`
- `Association(element1:String,role1: String, m1:Multiplicity, element2:String,role2: String, m2:Multiplicity)`
- `Composition(element1:String,role1: String, m1:Multiplicity, element2:String,role2: String, m2:Multiplicity)`
- `Constraint(name: String, element: String, rule: String)`

A name is a non-empty string containing alphanumeric characters only. A sequence is denoted using brackets as in Python while a multiplicity is denoted using a string as in UML, e.g. “*” or “1..5”.

SMW does not use a DTD document to generate a XML document since it extracts all the necessary information about the structure of the XML document from the metamodel module. In the next section we describe how to create new metamodel modules, i.e. how to add support to new modeling languages.
Packages are used to organize a metamodel. A package may contain other packages and model elements. A model element can inherit the features of one or more parent elements. Each element may have a number of attributes. The type of an attribute that can be defined as a metamodel element or a basic data types such as integer or string.

The basic data types are not an element of the modeling language but of the language used to define the modeling languages. For example, in FSM the type of the name of a State is string even when the FSM language does not define what a string is. The elemental data types in SMD are integer, unlimited integer, string, boolean and enumeration. These data types are mapped into the basic Python data types. The multiplicity of an attribute may be 0..1 or 1.

An association is a symmetric relationship between two elements. It is represented as two association ends. Each participant in the association has an attribute named after the association end. Each end has a multiplicity define as a range, e.g. 0..1 or 0..*. SMW supports a special kind of associations named compositions that represent a containment relationship. However, for the purposes of this paper we will treat compositions as normal associations.

A constraint is defined as a Python boolean function. We can only attach a constraint to a model element. If we want to define a constraint over an attribute or an association we should attach it to a model element owning the attribute or participating in the association.

The following example shows how we can represent our language for describing finite state machines shown in Fig. 1 using SMD:

```
1 Package("FSM")
2 Element("FSM","StateMachine", [],
3   [Attribute("name","String")])
4 Element("FSM","Token",[],[Attribute("name","String")])
5 Element("FSM","Transition",[],[])
6 Element("FSM","State", [],[Attribute("name","String")])
7 Element("FSM","AcceptingState",["State"],[])
8 Composition("StateMachine","alphabet","1..*",
9   "Token","stateMachine","0..*")
10 Composition("StateMachine","transition","0..*", 
11   "Transition","stateMachine","1")
12 Composition("StateMachine","state","1..*", 
13   "State","owner","1")
14 Composition("StateMachine","initial","1", 
15   "State","stateMachine","0..1")
16 Association("Transition","trigger","1", 
17   "Token","transition","0..*")
18 Association("Transition","source","1", 
19   "State","outgoing","0..*")
20 Association("Transition","target","1", 
21   "State","incoming","0..*")
```

A SMD metamodel cannot be used directly in SMW for performance reasons. We compile the SMD elements into executable Python code that can be used directly in any Python interpreter. Model elements, such as State or Transition, are represented in SMW as a Python class with same name as the model element. Specialization of model elements is implemented using inheritance. Metamodel attributes are represented as Python attributes with the same name and associations are represented using the SMW collections. Constraints are represented as boolean functions. In the next section we describe this process with more detail.

### 5.2 Generation of a metamodel module

A SMW metamodel language is created automatically from a data file. This has several advantages. First, it saves us from the tedious work of implementing these classes by hand (UML 1.4 contains 130 different model elements). It allows final users of SMW to support their own profiles or extensions to the UML language without
learning the internals of the toolkit. Automatic generation ensures that a SMW metamodel module has the same elements with the same properties as a standard metamodel. Beside this, UML is an evolving standard and the OMG has published new revisions at a steady pace since 1997. Automatic generation of metamodel modules will help us to keep the toolkit up-to-date with the UML standard.

The generation of a metamodel module is performed by a stand-alone tool, named mmgen. Currently, the tool can read a metamodel description in three different formats: An XMI document containing a metamodel expressed using MOF, an XMI document containing a UML class model and a SMD model.

The MOF-based document is the kind of metamodel provided by the OMG to define the UML interchange metamodel (a chapter in [20]). The metamodel provided in that document is clean and contains a definition of the enumerations and data types used in the model.

We can also create a metamodel by drawing a UML class diagram. The metamodel is then generated based on the UML subset that is equivalent to MOF. That is, based on the information stored in the packages, classes, associations and generalizations of the UML model.

Some old versions of the UML metamodels provided by the OMG were created using UML class diagrams. We have found two problems with some of these old UML metamodels. First, the definition of the enumerations contained in the models is not complete. For example, OMG document ad/99-10-16 contains an enumeration called AggregationKind. However, the enumeration is empty; it does not define the different types of aggregation. The solution to this problem is to provide our own data types manually. The second problem is that these models contain some elements that were deleted from the diagrams but not from the model. The solution was to edit and clean the models by hand before generating a metamodel.

The SMD input format is not based on a standard but is more amenable for a human than a XML document. It can be used to define our own profiles if we do not have a UML tool that can create XMI files. Actually the mmgen tool uses the SMD as an intermediate representation during the generation process. That is, the mmgen tool converts first a MOF or UML document into a SMD and then it generates the metamodel module.

The compilation of a SMD metamodel into an executable SMW metamodel module in Python is a simple process since the semantics for model elements and associations are already implemented in the MetaMM module.

Each generated Python class has a dictionary that describes the types and multiplicities of the meta attributes and meta associations. This dictionary is used in the constructor to initialize the association ends of the object and to maintain model consistency every time that an attribute is updated or element is added or removed from an association end. It also used to implement the reflection interface.

Finally, a well-formed rule is added to the Python class to check the multiplicity constraints in the associations ends of the class. This implies that multiplicity constraints are only checked at the end of a transaction or when the method isWellFormed is called, but our experience shows that this approach simplifies the construction of the scripts.

As an example, the following code shows the Python definition of the class StateMachine of the FSM module. Lines 9–11 contain the definition of a well-formed rule that constrains the multiplicities of the association ends. The class MMClass is defined in the MetaMM module and implements the consistency rules for model elements.

```python
1 class StateMachine(MMClass):
2     __mm__={
3         'name': (kind__Attribute, String, Many, None, None),
4         'state': (kind__Composition, State, Many, 'owner', One),
5         'initial': (kind__Composition, State, Many,
6                      'stateMachine', One),
7         'transition': (kind__Composition, Transition, Many,
8                                             'stateMachine', One),
9         'alphabet': (kind__Composition, Token, Many,
10                     'stateMachine', Many)
11     };
12 def wfrMetaModelMultiplicity(self):
13     return self.state.size()==1 and self.initial and \
14     self.alphabet.size()==1
```

### 5.3 From UML to SMD

The SMD format is useful to describe small modeling languages. To convert larger metamodels or the metamodels provided by OMG we need to be able to extract a metamodel from a UML class diagram. This process is performed in two steps. First, the UML model is converted into SMD and then converted into Python using the process described previously.

We have used our own toolkit to generate the UML to SMD compiler. An abridged version of the compiler is described below. There are three main methods: one to extract information from a UML class, another to extract information from an association or a composition and a third one to process UML packages.

The process starts by reading the UML model describing the metamodel from a XMI document. Then, we process the main package in the model. In UML, a model is also a package, so we do not need a special method to process the root element in a model. The method that processes a package is straightforward. A SMD package is defined with the same name as the UML package. Then we iterate through all the elements owned by the UML package and process those of type Package, Class and Association. We ignore UML datatypes as well as associations with more than two ends.

```python
1 class UML2SimpleMetamodelLanguage:
2     def extractUMLPackage(self,p,path):
```
To compile a UML class into SMD we just need to obtain a collection with the names of the parents of the UML class (line 15) and a collection with the names of the attributes and their types (lines 16-19).

```python
# define a package
path=path+":"+"*p.name"
SMD.Package(path)
# process all elements inside the package
for e in p.ownedElement:
    if e.oclIsKindOf(UML.Package):
        self.extractUMLPackage(e,path)
    if e.oclIsKindOf(UML.Class) and \
        not e.oclIsKindOf(UML.Datatype):
        self.extractUMLClass(e,path)
    if e.oclIsKindOf(UML.Association) and \
        e.connection.size()==2:
        self.extractUMLAssociation(e)
```

Converting a UML association or composition to SMD is a more complex process. In the case of a composition, SMD assumes that the first association end is the composite and the second the part. If this is not the case, lines 22-26 swap the association ends. SMD also requires that all association ends be named. If an end does not have a name, it is generated from the participant. Lines 34-35 convert a UML Multiplicity element into a string.

```python
def extractUMLClass(self,c,path):
    parents=c.generalization.parent.name
    attrs=[]
    for a in c.feature:
        if a.oclIsKindOf(UML.Attribute) and a.type:
            attrs.append(SMD.Attribute(a.name,a.type.name))
    SMD.Element(path,c.name,parents,attrs)
```

5.4 Limitations

There are some limitations built-in the mmgen tool and the SMD language. We assume that all the association ends are ordered. We also assume that there are no two elements with the same name in a metamodel even if they are in different packages. The tool generates only one Python module for the complete SMD metamodel instead of generating a different Python module for each package. Package names have changed in different versions of the UML, therefore scripts are more portable between different metamodels if they do not use package names. So far, there are no name conflicts in the UML metamodels.

There are also other limitations that are inherited by MOF and the metamodels provided by OMG, such as the lack of a standard definition of well-formed rules and undefined datatypes. The toolkit enforces automatically the types and multiplicities of attributes and associations. However, other well-formed rules must be translated from OCL to Python manually.

Even with its limitations, we have used the mmgen to generate a metamodel module for UML 1.1, 1.3 and 1.4 based on the metamodels downloaded from the OMG web site and for other modeling languages such MML [4, 6].

In the next section we describe how to use SMW and the UML metamodel modules to create an example application of model transformation.

6 Example: encapsulate attributes

As an example of a simple application, we show a script that implements a model transformation: We want to convert all public attributes in a class to private. This requires adding to each attribute two new public operations to set and get its value. For brevity, the script does not check certain conditions, for example, if a class already has a get and set operation for an attribute, neither updates the body of the operations in the model so they use the new set and get features instead of accessing the attributes directly.

Figure 5 shows a simplification of a part of the UML metamodel that describes how UML classes, UML attributes and UML operations are stored in a UML package. A package may own one or more model elements, including classes and other packages. This is represented by the association between Namespace and ModelElement.
A UML model is also a package and therefore inherits all its features. Each UML class has a number of structural features, i.e. attributes, or behavioral features, e.g. operations. A UML attribute is described by a name, a visibility and a type. A UML operation has a name, a visibility, a specification describing its behavior and a list of parameters. Each parameter has a name, a type and a direction. A direction may be in, out and return. The types of attributes and parameters are described by a class.

We can illustrate our script with the following example: Let’s assume that we want to transform a UML class named Point that has two public attributes named x and y of type Integer. We can represent these concepts in the object graph shown in Fig. 6.

To transform the x attribute, we will change its visibility to private and create two new features in the class Point named getx and setx. The operation getx has one parameter called result of type Integer and kind return. The specification of getx is the string ”return x”. The operation setx has one parameter named newValue. It is an in parameter of type Integer. The specification of setx is the string ”x=newValue”. Figure 7 shows the resulting object graph after the transforming the attribute x.

The construction of a SMW script that implements this transformation is rather simple once we understand how a class model is represented in the UML metamodel and how we actually want to transform it. We define the function encapsulateAttribute as follows:

```
import string

def encapsulateAttribute(a,metamodel):
    a.visibility=metamodel.VisibilityKind.vk_private
    # add a public accessor (getter)
    a.owner.feature.insert(
        metamodel.Operation(
            name="get"+string.capitalize(a.name),
            visibility=metamodel.VisibilityKind.vk_public,
            parameter=[ metamodel.Parameter(
                name="result",type=a.type,
                kind=metamodel.ParameterDirectionKind.pdk_return) ],
            specification="return "+a.name
        )
    )
    # add a public setter
    a.owner.feature.insert(
        metamodel.Operation(
            name="set"+string.capitalize(a.name),
            visibility=metamodel.VisibilityKind.vk_public,
            parameter=[ metamodel.Parameter(
                name="newValue",type=a.type,
                kind=metamodel.ParameterDirectionKind.pdk_in)
            ],
            specification="x=newValue"
        )
    )
```

Fig. 6. The class Point before transformation

Fig. 7. The class Point after transformation
The function `encapsulateClass` will apply this transformation only to the public attributes of a given class. It is implemented using a for loop that iterates over all the elements returned by the `select` query that chooses all the public attributes in the class.

```python
def encapsulateClass(c, metamodel):
    for a in c.feature.select(lambda f:
        f.oclIsKindOf(metamodel.Attribute) and
        f.visibility==metamodel.VisibilityKind.vk_public):
        encapsulateAttribute(a, metamodel)
```

We can now define a function that transforms all classes in a model. The recursive function `transformAllClasses` traverses the complete model and transforms each class as soon as it is found. Termination is guaranteed by the fact that a package cannot directly or transitively be owned by itself.

```python
def transformAllClasses(package, metamodel):
    for e in package.ownedElement:
        if e.oclIsKindOf(metamodel.Class):
            encapsulateClass(e, metamodel)
        if e.oclIsKindOf(metamodel.Package):
            transformAllClasses(e, metamodel)
```

Once we have defined the function `transformAllClasses` we can explain the main body of the script. Line 34 loads a model from a file whose name has been passed as the first command line parameter. The next statement obtains the metamodel used to define the model. This is necessary since the script can transform models written in UML 1.3 and UML 1.4. Then, we call the `transformAllClasses` to perform the transformation. Finally, if all the elements in the transformed model are well formed, we save the model into a new file, passed as the second command line parameter.

```python
import sys, smw.io
# load model passed as an argument
model=smw.io.loadModel(sys.argv[1])
# obtain its metamodel
metamodel=model.getMetamodel()
# transform the model
transformAllClasses(model, metamodel)
# if the model is fine, save the results in a file
if model.isWellFormedRecursive():
    smw.io.saveModel(sys.argv[2], model)
```

This script is a complete stand-alone Python program that can be used to transform any model produced by a UML and XMI-compliant CASE tool. Figure 8 shows how we used Rational Rose to create a simple model containing the class `Point`. We generated a XMI file by using the “Tools/Export Model to UML” option from Rose and we ran the script. The script loaded the generated XMI document but all information about the diagrams such as the size and position of the class `Point` was lost, since our toolkit does not support the proprietary extensions to XMI used by Rose. However, all relevant model information was there, so the transformation was performed successfully and a new model was generated. Finally, we imported the model generated by the script back into Rose. At this point, the class diagram was empty (the class `Point` was not shown), but we were able to drag and drop it from the model outline view into the class diagram. The final result is shown in the right side of Fig. 8. We expect that the problems associated with the interchange of diagram information will disappear when a standard such as [8] is approved and widely implemented.

7 Conclusions
In order to use models as the main artifact in software engineering we need model-based tools for every
task in software development. In this article, we have enumerated the main features of a scripting language that can be used to construct small tools to process models. These tools can be used to check model consistency, implement model transformations and to create derived artifacts from a model such as code or test cases.

This kind of scripting tools should support the existing standards such as UML and XMI. Scripts should be independent of the modeling tools used to construct the models since in many cases, scripts will be use to integrate different tools from different vendors. Another equally important requirement is the need of a mechanism to ensure model correctness. The scripting language should enforce the well-formed rules of the modeling language and support user-defined constraints.

We expect that most scripts will be developed in-house by designers or programmers specialized in the task since the development of a script requires an advanced knowledge of the scripting tool, the modeling language and also of the other development tools and methods used in a given project.

In this article, we have also presented SMW, our scripting toolkit. We have extended an existing scripting language using as main abstraction the metamodel module. A metamodel module represents a modeling language and supports all the operations required to transform and query a model while ensuring model consistency and correctness. Our approach to comply with the UML standard is based on the automatic generation of the metamodel modules based on the official definitions posted by the OMG in its web site. The SMW toolkit supports UML version 1.4 and it is being used by its authors to develop new modeling languages. The source code for the toolkit is freely available from the author under an open source license.

The current implementation of the toolkit is open to many improvements. The metamodel modules do not support all the features of MOF. Mainly, we have found some issues on how the elemental data types are used in the MOF models. Also, one of the main design assumptions of the toolkit is that it is always possible to store complete models in memory. An alternative that we are exploring is to adapt the toolkit so it can retrieve model elements seamlessly from an XMI-based repository in a remote machine.

Although this work is eminently practical, we like to think that it can influence the way the vast theoretical research on UML and other modeling languages is performed. It facilitates the implementation of new model transformations, algorithms to analyze models and generate derived artifacts. It allows us to and experiment with the internals of existing modeling languages and to validate new ones. It also facilitates the creation of experiments to validate and demonstrate our research and it allows us to freely distribute these experiments and all the tools needed to repeat them.

References

29. van Rossum, G.: The Python programming language. Available at: www.python.org

Ivan Porres received in 1997 his MS degree in Computer Science at the Polytechnic University of Valencia, Spain and in 2001 his PhD in Computer Science at Åbo Akademi University in Turku, Finland. His thesis, "Modeling and Analyzing Behavior in UML", studies how to use different formal methods to ensure the correctness of software modeled using UML. Currently, he works as an assistant professor at the Department of Computer Science at Åbo Akademi University where he studies how to use models and modeling in all aspects of software engineering.