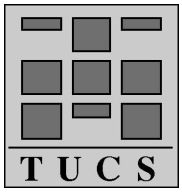
A Generic Deep Copy Algorithm for MOF-Based Models

Ivan Porres
Marcus Alanen

TUCS Turku Centre for Computer Science
Åbo Akademi University, Department of Computer Science
Lemminkäisenkatu 14A
FIN-20520 Turku, Finland
e-mail:{iporres,maalanen}@abo.fi



Turku Centre for Computer Science
TUCS Technical Report No 486
November 2002

ISBN 952-12-1073-7
ISSN 1239-1891

Abstract

This paper discusses the problem of copying a subset of a model based on the Meta Object Facility (MOF), such as a UML model. We show how the usual shallow and deep copy operators implemented in many programming languages can not be used to copy MOF models. As a solution to this problem, we propose a new algorithm that works at the metamodel level and produces the expected results. This algorithm can be used to implement tools that transform models, e.g., to implement the copy and paste feature or a template instantiation mechanism in a UML editor.

Keywords: MOF, Metamodeling, Model Transformation, UML, Deep Copy

TUCS Laboratory
Software Construction Laboratory

1 Introduction

Software models have become a primary artifact in software development thanks to the advent of the Unified Modeling Language (UML) [5]. Models are created, reviewed and updated along the whole project life in many software development processes. However, in order to be practical, this requires the construction of specialized tools to manipulate the models.

The UML language is defined using the Meta Object Facility (MOF) [4]. MOF is used to define metadata, such as the structure of an object repository or a modeling language. The UML language, including each UML model element such as class, attribute or dependency, is defined in a document called the UML metamodel. The UML metamodel is described using MOF. Actually, MOF is similar to the core subset of UML, what is usually called UML class diagrams. Tools that understand the MOF standard can manipulate any MOF-based modeling language, including UML and future versions and extensions to UML.

In this paper we discuss the problem of the efficient duplication of a subset of a model. This task is needed while performing many model transformations. Its implementation may seem trivial: most object-oriented programming languages have a deep copy operator that can copy arbitrary data structures, including data structures that contain circular references. However, a generic deep copy operator will ignore the special semantics of a MOF-based model. In many situations, the standard deep copy operator will copy too much, while the simple or shallow copy operator will copy too little. In the article, we will define a new modelcopy operator that performs these operations as expected.

We proceed as follows: Section 2 describes the purpose and requirements of a copy operator for MOF-based models. Section 3 defines how we would like the modelcopy operator to work, and gives examples of common situations while Section 4 shows the modelcopy algorithm in Python pseudocode [10], and explains the inner workings of the algorithm. We also show how we meet the requirements stated in Section 3, and discuss the implementation details. We describe examples of usage in Section 5, particularly referring to our SMW [8] toolkit. SMW is a collection of tools for manipulating software models. All the figures and code examples in this article have been created using SMW. Finally, we conclude in Section 6 by stating the importance of useful and correct tools in manipulation models in UML (and other languages). In this context, we believe the modelcopy operator is a basic algorithm that can be used in many tools for model transformation.

1.1 The UML Metamodel

In order to understand an algorithm that transforms a UML model we must understand the internal representation of UML, i.e., the UML metamodel. We will use the model in Figure 1 as an example. This figure shows a simple UML class diagram. It contains two classes, Company and Person, and an association between the classes named work. In UML, associations are bidirectional by default. The as-

sociation has two ends, named employee and employer. An instance of Company can be associated with many instances of Person while a Person may be associated with just one Company. The class Person also contains two attributes, Name and Address of type String. The diagram does not contain the class String, so we assume that its defined somewhere else.

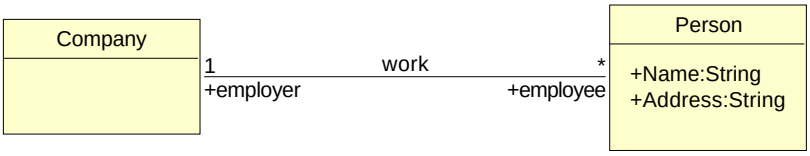


Figure 1: Example Model

All the elements that can appear in a UML diagram, such as Class, Association and Attribute, are defined in a document called the UML metamodel. Figure 2 shows (a simplification of) a part of the UML metamodel that describes UML class diagrams. The UML metamodel is defined using UML itself. It describes the abstract syntax of a UML class diagram, e.g., what is a UML class or a UML association and how they are related.

The metamodel contains metaclasses, such as ModelElement and generalizations between metaclasses, e.g., Feature is a subclass of ModelElement. Metaclasses may contain attributes, such as name in ModelElement and can be connected using associations, such as the line between Package and ModelElement.

Associations describe how we can connect and compose different model elements. The metamodel shown in Figure 2 shows that a UML Package may own many other model elements, for example classes and associations. Associations in the metamodel are also bidirectional. if a Package owns a ModelElement, the Package is the namespace of the ModelElement.

The association between Package and ModelElement is a composition. This is represented in by the black diamond next to Package. A composition represents a whole / part relationship. An ModelElement is a part of a Package and a Package owns the ModelElement.

Some elements of the UML metamodel are not always represented directly in a UML diagram. An example is the UML AssociationEnd that contains information about the name or multiplicities of an end of an association. This model element is not represented as such, but as one or more labels attached to the association. In our example, there is an AssociationEnd with the name employee that connects the Person class with the work association.

Figure 3 shows how the model described in Fig. 1 fits into the UML metamodel described in Fig. 2. This figure is a collaboration diagram. Each rectangle in the diagram is an instance of a metaclass where each link between two objects

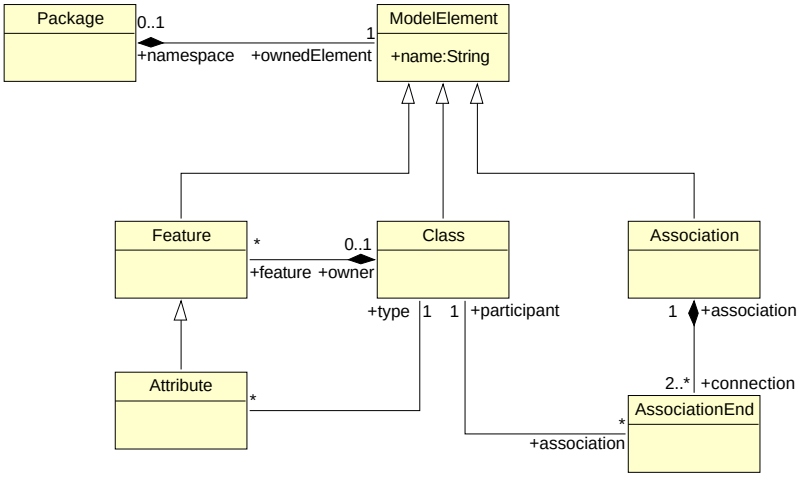


Figure 2: Example Metamodel

represent an instance of a metamodel association. This is also how a UML model is stored into an XMI [6] file or a repository according to the UML standard. This diagram also makes explicit some information from our example: the Name and Address features of Person are linked explicitly with String.

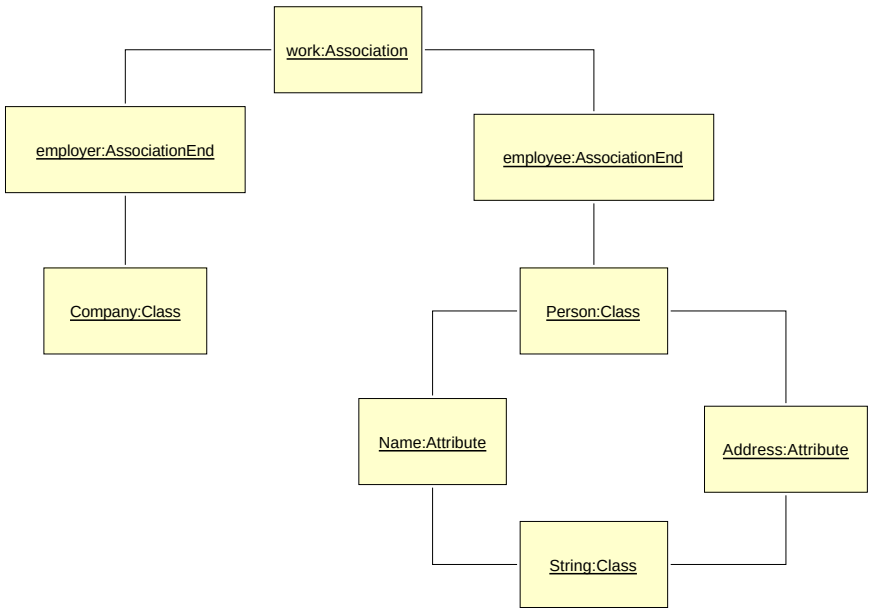


Figure 3: Metamodel Representation of the Example Model

2 The Problem

Once we know how we can represent UML models as a collection of objects and links between objects, we can proceed to explain the copy operation in more detail. As an example, let us assume that we want to copy the object that represents the class Person.

Most modern programming languages implement several copy operators. Shallow copy usually copies a single object while deep copy copies a whole data structure recursively, e.g., the object passed as a parameter and all other objects that are transitively connected to the object.

However, we cannot apply the standard shallow or deepcopy operators as such to copy a UML class in a model. If we use the shallow copy operator, we will obtain a copy of Person, but the copy will not have any features. The Name and Address attributes will not be copied since, by definition, shallow copy only copies one object, and features can belong to at most one class. Also, the original Name and Address attributes cannot and should not be shared between the original and the copy of Person. This is probably not the result expected intuitively. More rigorously, we can argue that we have violated the semantics of the composition association between a class and its features, since we have split a composite from its parts.

The solution may seem to use deep copy to copy recursively the class Person and all its features. However, all elements in the model are connected together since all associations are bidirectional. Deep copy will create a copy of Person, but also of the rest of the model. We will obtain a complete copy of the model. This again is not acceptable, we want to duplicate a subset of a model, not the complete model.

One possible workaround would be to break the link from the class and the other elements, perform a deep copy operation and then set the links back again. This approach does not work in practice since other model elements, for example an Operation or an Attribute, may contain associations to other parts of the model. Also, it is not generic. It may work in the case of copying a class in a UML class diagram, but it does not generalize to any MOF-based model. Another erroneous approach is to implement a deep copy operation that only considers composite associations. Unfortunately, this will ignore valid associations between elements to be copied and the rest of the model. In the example, we want to preserve the association between the attribute Name and the class String,even if String should not be copied.

Figure 4 shows the expected result for copying class Person in our example model. Modelcopy should duplicate the class and its attributes. The new class, named Copy of Person in the figure, is not associated to Company since that would also require to duplicate the association. Therefore not all associations between the objects in the model are preserved. However, the attributes of the class Copy of Person should have a type. As we can see in the figure, this association should be preserved in the copy.

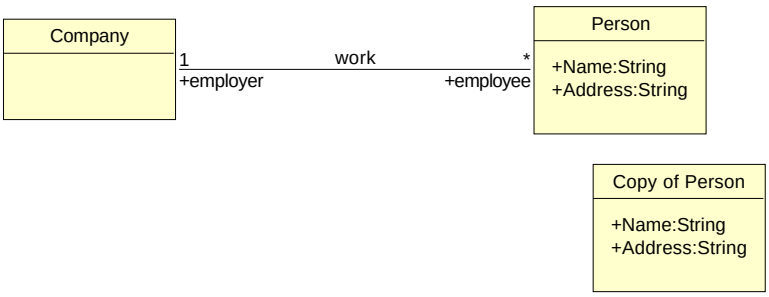


Figure 4: Copy of Person

A second example is when we want to copy Person and work simultaneously. Modelcopy should create a copy of both elements. However, Copy of work should be connected to the Copy of Person in one side but keep its original connection to Company in the other side. This example is shown in Fig. 5.

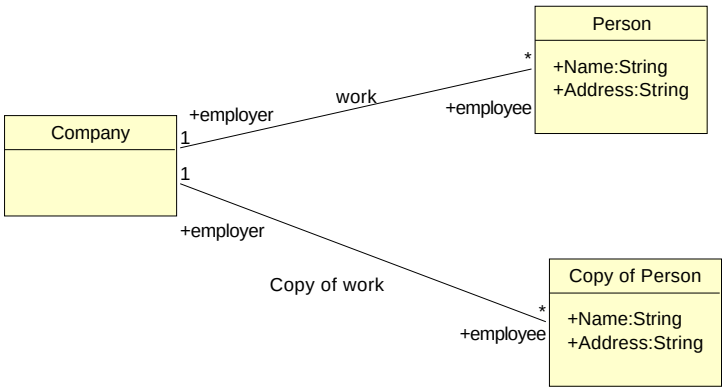


Figure 5: Copy of work and Person

Figure 6 shows our third example where we want to copy Person, Company and work. In this case we should get two new copies of the classes and an association connecting the copies. An alternative outcome to this operation is shown in Fig. 7. We consider this result to be incorrect since it is not consistent with the second example.

We should note that copying an element in a model is a different action that adding a graphical representation of an element into several diagrams. For example, the class Person could appear again in a different class diagram in the same model, e.g., describing the payroll system in a company. However, in this case, both diagrams represent the same class Person. This is supported explicitly in

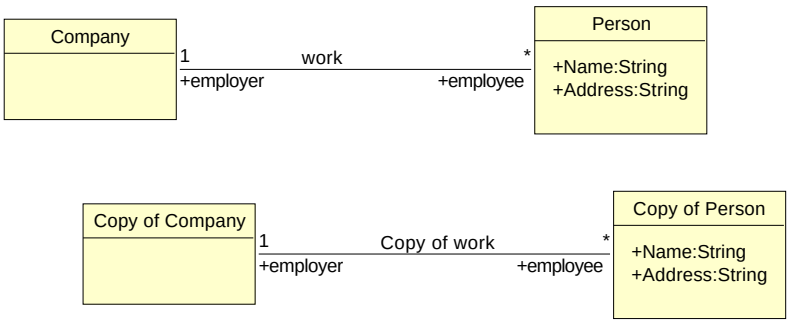


Figure 6: Copy of Company, Person and work

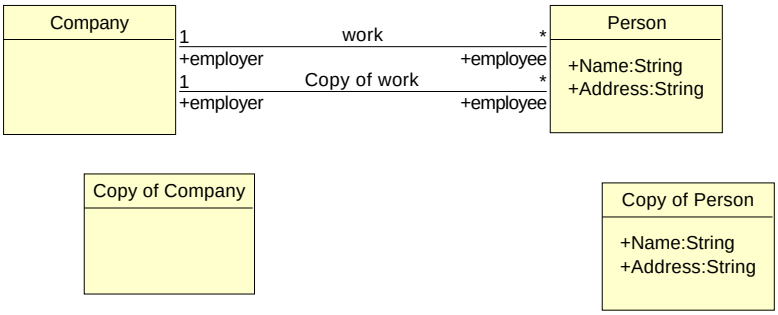


Figure 7: An Alternative Result for the Copy of Company, Person and work

the UML metamodel (Section 3.5.1 of [5]). Any model element may have many (graphical) representations and a graphical representation may describe many graphical elements. Adding the class Person into another class diagram does not involve duplicating the class, but creating a new presentation for it.

3 The Modelcopy Operator

Thanks to the observations from the previous examples we are now ready to define the modelcopy operator more rigorously: given a ordered collection of elements $C=[c_1,..,c_n]$ that belong to model M, the modelcopy operator returns a new ordered collection $C'=[c_1',..,c_n']$ where c_i' belongs to M and is a well-formed and connected copy of c_i.

1. The elements $c_1,..,c_n$ to be copied should belong to the same model. We say that two elements belong to the same model if they are transitively connected.

2. All copied elements should be well-formed. This implies:

   (a) If element e is copied all parts of e (elements participating transitively in a composition association of e) should also be copied.

   (b) Copies should preserve the ordering and multiplicity constraints of the original associations.

3. All copied elements should be connected: Each association between a copied element and an element in the original model should be preserved except when:

   (a) This would break a multiplicity constraint of the association.

   (b) The element in the original model is also being copied. In this case the association should occur between the copied elements.

We can see an example of the application of rules 2.a and rules 2.b in the Copy of Person. The copy also contains a copy of the attributes Name and Address, and in the same order as the original. These attributes only have one type (String), this is an application of rule 3.a. We can observe the application of rule 3.b in Fig. 6. If we do not consider this rule we obtain the result shown in Fig. 7.

We should note that a modelcopy algorithm as described above works at the MOF level. It does not guarantee that the copied models will be well-formed at the modeling language level. For example, there is a well-formed rule in UML that says that a region in a Statechart may have only one initial state. If we use the modelcopy algorithm to duplicate an initial state the model will not be well-formed.

In the next section we will give a detailed algorithm for the modelcopy operator.

4 The Modelcopy Algorithm

In order to describe an algorithm that performs a modelcopy operation in a generic way, we need a mechanism to query the structure of the modeling language, i.e., to query the metamodel used in a given model.

We assume that there is a reflection interface that, given a model element, returns its metaclass and the attributes and associations of the metaclass. Examples of such an interface can be found in [8] and [1].

In this article, we assume that the function classOf(e) returns the metaclass of model element e. featuresOf(c) returns a list of all features of the metaclass c. A feature can be an attribute, or an association end. Each feature has an attribute with its name, an attribute named kind that contain one of these values: Attribute, Association or Composition .

If the feature is an Association or a Composition, there will be an attribute name multiplicity that contains the multiplicity constraint of the association. Valid values

are either one, for exactly zero or one child element, or unlimited, for any amount of child elements. The attributes otherName and otherMultiplicity will describe the name and multiplicity of the other association end.

The modelcopy algorithm assumes a property of the metamodel. If we consider each metaclass as a node in a graph and each composition association a directed arc from the composite to the part, the created graph should be a tree (a directed acyclic graph). This assumption is implicit in the semantics of MOF, holds for all versions of UML and it is also an implicit requirement in the XMI Specification.

In the UML metamodel there are some associations that are marked as ordered. The order of the elements of these associations should be preserved trough model transformations. We have designed the modelcopy algorithm so it preserves the ordering of the association ends if the methods that manipulate the association ends also preserve the order

The following clarifications of the pseudocode might be in order. If c is a collection, the command for o in c binds o successively to each element in c. Also, c.append(o) adds a new element to the collection. If the collection is ordered, the element is placed at the end of the collection. If d is a dictionary, d[k]=o adds a new object o with the key k to the dictionary, deleting any previous value of d[k]. The expression k in d returns true if the key k is present in the dictionary. The expression d[k] returns the current mapping of k, and d.getValue(k,default) returns the value d[k] if it is defined, otherwise it returns default. The statement get(o, f) returns a collection with the values of field f of object o, if that field does not have a multiplicity constraint, and otherwise it returns the single value (object) of field f in object o, or None, meaning there is no object in field f.

The function setValueNoUpdate(o, f, v) is used to set the value v of field f of an object o, without updating the other end, i.e., without updating the object v. The function insertValueNoUpdate(o, f, v) is used to append the value v of field f of an object o, also without updating the other end.

The algorithm works in two passes, modelcopy1 and modelcopy2. In the first pass, it creates a new instance of each element where needed, and in the second pass, it sets each new element's associations correctly.

The function modelcopy1 creates a mapping between an element in the original model to a newly-created element in the copied1 dictionary. The created elements have default values without any connections to each other. We assume the ordinary UML containment policy, that compositions and attributes are owned by exactly one element, and that all elements except the root of the tree are owned by an element. Thus, modelcopy1 is guaranteed to create all objects in the subtrees, and only those objects.

The mapping is created by visiting each element in C transitively under the composition or attribute association (lines 4–8), and creating a new instance of each element encountered (line 3). This satisfies our requirement that if an element is copied, all parts of it are copied. Termination is guaranteed by the copied1 dictionary and the test on line 2 in modelcopy1.

The function modelcopy2 sets the connections between elements. This is ac-

```
def modelcopy(Collection c) : Collection
    copied1, copied2 = new Dictionary, new Dictionary
    result = new Collection
    for e in c:
        result.append(modelcopy1(e, copied1))
    for e in copied1:
        modelcopy2(e, copied1, copied2)
    return result
```

```
def modelcopy1(Element e, Dictionary copied1): Element
    if e not in copied1:
        copied1[e] = new getClassOf(e)
        for f in getFeaturesOf(classOf(e)):
            if f.kind != Association:
                for value in get(e, f.name):
                    modelcopy1(value, copied1)
    return copied1[e]
```

```
def modelcopy2(Element old, Dictionary copied1, Dictionary copied2): Element
    if old not in copied2:
        copied2[old] = 1
        newE = copied1[old]
        for f in getFeaturesOf(class(old)):
            if f.kind == Association:
                if f.multiplicity == ZeroOrOne:
                    v = get(old, f.name)
                    if f.otherMultiplicity == ZeroOrOne and not v in copied1:
                        setValueNoUpdate(newE, f.name, None)
                    else:
                        setValueNoUpdate(newE, f.name,
                                         copied1.getValue(v, v)
                        if v and not v in copied1:
                            insertValueNoUpdate(v, f.otherName,
                                                newE)
                else:
                    for v in get(old, f.name):
                        if f.otherMultiplicity == ZeroOrOne
                           and not v in copied1:
                            skip
                        else:
                            insertValueNoUpdate(newE, f.name,
                                                copied1.getValue(v, v)
                            if v and not v in copied1:
                                insertValueNoUpdate(v, f.otherName,
                                                    newE)
            else:
                # It is an attribute or a composition
                if f.multiplicity == ZeroOrOne:
                    value = get(old, f.name)
                    setValueNoUpdate(newE, f.name,
                                     modelcopy2(value, copied1, copied2))
                else:
                    for value in get(old, f.name):
                        insertValueNoUpdate(newObj, f.name,
                                       modelcopy2(value, copied1, copied2))
    return copied1[old]
```

complished by going trough the dictionary of newly created objects and setting their connections to point to the same elements as the original element's connections point to, or to new elements in the mapping copied1 if they exist.

The most important part of this pass is to not update the other end of a bidirectional association automatically. The reason for this is rather subtle. The appending of new associations must be done in the same order as they were defined, separately at both ends. Otherwise, the other automatically updated end would have its elements in the wrong order.

An example of this is shown in Figure 8. We copy the elements in pass modelcopy1, and start adding the associations in modelcopy2. Now, if we would update both ends (middle picture) at the same time, the wrong result is bound to happen in ordered associations; the result obtained is not our original model.

Thus, we update only one end of an association, at lines 12, 20, 27 and 30, for associations and compositions alike. This is no problem for copied elements that reference other copied elements since we are guaranteed to visit both ends of an association anyway, by lines 24–30.

In practice, the modelcopy2 pass is not as straightforward. An association with a multiplicity of exactly one can not connect to the original and the copied element. In this case all we can do is skip this connection for the copied element. This is done at lines 9–10 and 17–18. How big a problem this is depends highly on the metamodel.

When associations between a non-copied and copied element are valid, copied elements must immediately update the associations of non-copied elements, since we will never visit the non-copied elements in modelcopy2. This is done at lines 13–14 and 21–22.

An additional small caveat is that non-copied elements have their new associations in a non-deterministic order. This can be fixed by yet a third pass, which is not included in this paper.

Figure 9 shows the correct result. First, all associations of the copy of element 1 is updated, then the copy of element 2, resulting in a complete copy.

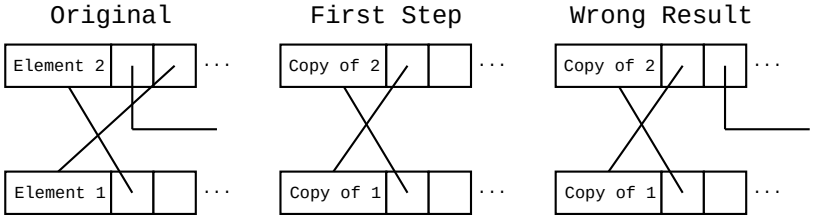


Figure 8: The Wrong Result

In the worst case the algorithm terminates after copying all elements in the model. This results in an exact copy of the original model, in which the original and copy do not share any associations between each other.

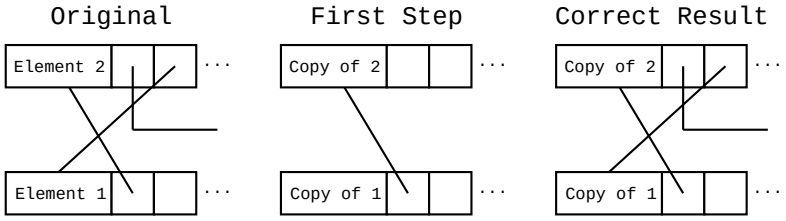


Figure 9: The Correct Result

4.1 Implementation

The modelcopy algorithm has been implemented in the SMW toolkit. SMW is a collection of tools for manipulating software models. It is implemented in Python. The SMW toolkit is divided in several components: the kernel, a networked repository, the model transformation framework and the graphical modeler.

The SMW kernel implements the reflection interface needed to implement the modelcopy algorithm as described above. It also supports OCL-like queries, bidirectional associations, well-formed rules and XMI input and output. The kernel contains metamodel modules for UML 1.1, UML 1.3, UML 1.4 and SA/RT. Users can define new metamodel modules from a MOF model or a UML class diagram. The modelcopy algorithm has been implemented in the SMW kernel and therefore is available in all SMW applications.

The SMW kernel supports strict transactions. A transaction is a sequence of transformations in a model. The kernel keeps a lists with the most recent transactions in the model and it is possible to undo and redo a transaction at will. A strict transaction is like a normal transaction but it can only be committed if the model is well-formed. If we try to commit a strict transaction that violated a well-formed rule, the kernel rolls back the transformation and throws an exception. This feature is used in the SMW tools in the conjunction with the modelcopy operator in order to ensure that the copy operations are also well-formed with respect to the constraints of the modeling language.

In the following section we show two applications of the modelcopy algorithm that have been implemented in the SMW toolkit

5 Examples

Copy and Paste

One of the most obvious application of the modelcopy algorithm is to implement the cut, copy and paste commands in a UML model editor.

The cut and copy commands use the selection, the items highlighted in the editor, as the subset of the model to copy. The copy command uses modelcopy to copy the current selection and places it into a clipboard. This is implemented

just as a single call to modelcopy. The clipboard is actually just the collection of model elements returned by modelcopy. The cut command is implemented by first copying the selection and then deleting it.

Paste uses modelcopy again to place the elements in the clipboard again in the current diagram. This requires five steps, described in the algorithm shown in Figure 10. First, we must decide if the contents of the clipboard can be pasted in the current diagram. For example, we should not allow the user to paste a state from a Statechart diagram into a class diagram. (lines 2–4). This step depends on the type of diagram. If this check is passed, we use modelcopy to duplicate the clipboard (line 5). This step is needed since it is possible to paste the contents of the clipboard several times.

```
1 def paste(clipboard,diagram):
2     for e in clipboard:
3         if not diagram.canContain(e):
4             raise "Diagram cannot contain element",e
5     new=modelcopy(clipboard)
6     for e in new:
7          diagram.connect(e)
8          diagram.addPresentation(e)
9          e.name="Copy of"+e.name
```

Figure 10: Paste Algorithm

Then we need to connect the copied elements to the main element shown in the current diagram. If we are pasting elements into a class diagram, we should add the elements to the ownedElement association of the Namespace shown in the diagram (Page 5–4 of [5]). If we are pasting into an Statechart, we should add the elements to the subvertex association of the top state of the StateMachine shown in the diagram (Page 5–18 of [5]). This step is performed in line 7.

Once an element is connected to the model element shown in the diagram, we should create a new graphical presentation in the diagram (line 8). Finally, we may need to rename the element. The elements in the collection returned by the copy of the clipboard are renamed by adding the prefix "Copy of" as shown in the figures in this article (line 9).

The algorithm in Fig. 10 uses three methods that we assume are defined in the object representing the diagram. The method canContain(e) returns true if the diagram can contain an element of type e. The method connect(e) connects the element e to the main element shown in the diagram, while the method addPresentation(e) adds a new graphical representation of the element in the diagram. We have observed that these methods are also needed in many other command of a UML editor, such as the drag and drop feature. The actual implementation of methods depends of the kind of diagram we are manipulating.

In all, the implementation of the copy and paste feature in the SMW editor is just a few hundred lines of Python code. By using the previous methods to

manipulate the diagrams, our implementation can handle any MOF-based model, including all types of UML diagrams.

Template Instantiation

Another application of the modelcopy operator is template instantiation. A template is a (part of a) model that we want to reuse in other models. Template instantiation is the introduction of a template in a model. Each element in the template should be copied into the current diagram. Optionally, it is possible to rename some of the elements in the template based on one or more template parameters. Figure 11 shows a simple algorithm for template instantiation. This algorithm accepts a parameter that it is used to rename the copied elements. This step is performed in lines 9–10 of the algorithm. We show an example of this mechanism in Figure 12. The leftmost side of the figure shows a template for the Subject-Observer pattern [2], while the right side shows an instance of the template where the «name» parameter has been replaced with the string "Temperature".

```
 1 def instanciate(template,newName,diagram):
 2     new=modelcopy(template)
 6     for e in new:
 7          diagram.connect(e)
 8          diagram.addPresentation(e)
 9          if e.name:
10             e.name=string.replace(e.name,"<<name>>",newName)
```

Figure 11: Template Instanciation Algorithm

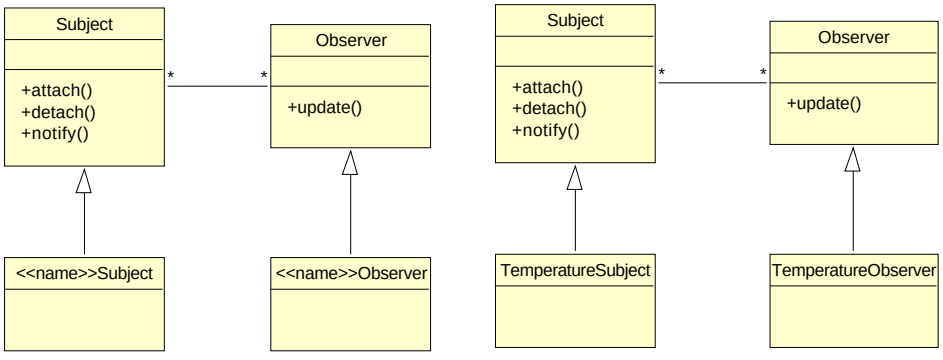


Figure 12: Template Instantiation Example

6 Conclusions

This article presents an algorithm that duplicates subsets of models described using UML or any other MOF-based modeling language. Although the problem may seem trivial, we have shown that its solution is rather complex and it requires an extensive knowledge of the MOF standard. The algorithm requires two phases and two temporary data structures while standard deep copy algorithms need only one phase and one temporary data structure.

We like to think that this algorithm is a useful building block to construct advanced tools that can manipulate any UML and MOF-based model. The Model Driven Architecture [7] approach can only be realized if we are able to construct sophisticated tools that can manipulate models in a generic way. MDA tools should support many different profiles or extensions to the original modeling languages. They should also be able to transform models from one profile to another. We consider that a generic modelcopy operator is necessary in many of these transformations.

Many researches have studied the properties of the UML language, but the construction of tools to manipulate UML models is seldom discussed. The UML standard itself does not discuss the construction or design of UML tools. Other implementation standards and libraries, such as Java Metadata Interface [1], or Novosoft NSUML [3] are targeted to tool builders but they do not discuss the need of a copy operator. This fact is surprising since we consider this a common operation in model manipulation and refactoring.

Most UML editors are commercial products and their source code is not available, so we cannot compare our approach with these tools. On the other side, Argo/UML [9] is an open source UML editor. Argo can copy the presentation of an element and place it in a new diagram, but it lacks a general copy and paste feature, i.e., it cannot duplicate existing model elements. Poseidon from Gentleware, a proprietary tool originally based on the Argo codebase, also presents the same behavior.

The SMW Toolkit, including the implementation of the modelcopy operator and its application in a UML model editor is available under an open source license at `http://www.abo.fi/~iporres/Projects/SMW`.

References

[1] S. Iyengar et al. Java metadata interface (JMI) specification. Available at `java.sun.com`.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[3] Novosoft. Novosoft metadata framework and UML library. Available at `nsuml.sourceforge.net`.

[4] OMG. OMG Meta-Object Facility (MOF). OMG Document formal/01-11-02. Available at www.omg.org.

[5] OMG. OMG Unified Modeling Language Specification. Version 1.4, September 2001, available at www.omg.org.

[6] OMG. OMG XML metadata interchange (XMI) specification. OMG Document formal/00-11-02. Available at www.omg.org.

[7] OMG. OMG Model Driven Architecture. OMG Document ormsc/2001-07-01. Available at www.omg.org, July 2001.

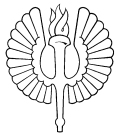
[8] Ivan Porres. A Toolkit for Manipulating UML Models. Technical Report 441, Turku Centre for Computer Science, January 2002. Available at www.tucs.fi.

[9] J. Robbins and D. Redmiles D. Hilbert. Software architecture critics in Argo. In *The 1998 International Conference on Intelligent User Interfaces, San Francisco, CA, USA, January 6-9*, 1998.

[10] G. van Rossum. The Python programming language. Available at www.python.org.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.fi



University of Turku
- **Department of Information Technology**
- **Department of Mathematics**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**