Copyng Subgraphs within Model Repositories

Pieter Van Gorp, Hans Schippers, Dirk Janssens

Department of Mathematics and Computer Science
University of Antwerp
Middelheimlaan 1, B-2020 Antwerp
{pieter.vangorp,hans.schippers,dirk.janssens}@ua.ac.be

Abstract

The set of operations in state-of-the-art graph transformation tools allows one to conditionally create and remove nodes and edges from input graphs. Node attributes can be initialized or updated with information from other attributes, parameters or constants. These operations appear to be too restricted for expressing model refinements in a concise manner. More specifically, graph transformation lacks an operation for copying subgraphs (multiple connected nodes, including their attributes) to a new location in the host graph. This paper presents a case study that illustrates the need, a syntax and an informal semantics for such an operation. It also discusses how the operation was integrated in an existing graph transformation language. Finally, it indicates how our ongoing effort towards the implementation of a model transformation language based on graph transformation makes optimal reuse of evaluation code for existing language constructs.

Keywords: graph transformation, copy, subgraph, model management, repository, UML, profile, SDM, MOF, JMI

Introduction

A model can be defined as a simplified representation of a part of the world, named the system [17]. Model repositories are databases with specialized support for storing and retrieving models. Their main functionality consists of serializing their data into standard model exchange formats (like XMI [13]), and exposing a query and transformation API (like OCL [11] and JMI [5]). Any program with the purpose of creating or changing models can be called a model transformation. The purpose of this paper is to extend graph transformation such that model transformations can be programmed at a high level of abstraction while the low-level APIs of mainstream model repositories are interfaced by means of compilers.

The data definition languages (like MOF [10] and E Core [8]) for modern model repositories (like MDR [6] and EMF [8]) are object-oriented. Consequently, model repositories can be perceived as object-oriented databases. The data instances in a repository can be perceived as graphs with objects taking the role of attributed nodes. Association, containment, inheritance and other relationships take the role of edges. Transforming data in repositories can thus be perceived as a graph transformation activity.

This paper is structured as follows: Section 1 presents two models of a meeting scheduler system. These models are expressed in two different UML profiles and a part of one...
model should be generated from the other one. When using graph transformation to formalize the model transformation that defines this generation process in Section 2, the need for a copy operator becomes obvious. Section 3 presents the syntax and semantics of the proposed copy operator as an extension to Story Diagrams [15]. Additionally, the section briefly compares two approaches for extending an existing Story Diagram engine. The next section refers the reader to related work while the paper concludes with a summary of the contributions and lessons learned.

1 Motivating Example Models

Figure 1 shows a conceptual model (CM) of a Meeting Scheduler application, specified in UML syntax [12]. At the conceptual level, analysts are free to use constructs such as association classes, views, and other language features. Such features may not be supported directly by the implementation language but they allow one to represent the problem domain as one perceives it in reality as good as possible.

A complete conceptual model contains all relevant nouns and verbs from a problem domain as classes and operations. In order to localize changes to the problem domain, many architectures hide the conceptual model by means of layers. To design such architectures, Rosenberg and Scott [14] propose to model user interface screens as interfaces and user interface flow as services. Only services are allowed to access entities, which are based on the classes in a conceptual model. Figure 2 shows a robustness model (RM [14]) of the application under study. Note that the entity Schedule corresponds to the class Schedule from Figure 1.

Figure 3 clarifies how the elements from the conceptual modeling diagram shown in Figure 1 relate to a typed and attributed graph in the underlying model repository. The tree on the left represents the “containment hierarchy view” from the Meeting Scheduler sample in the MagicDraw UML tool. Node n1 is of type Model and represents the UML model that contains both the application examples and the definitions of the profiles used within these examples. All examples reside in node n2 of type UmlPackage and with name “Examples”.

Figure 1. Conceptual Model of a Meeting Scheduler application.
Node \( n3 \) represents the actual Meeting Scheduler sample. This \textit{UmlPackage} contains node \( n4 \) which represents the conceptual model of the Meeting Scheduler. All its contained classes (like \textit{Attendee}, \textit{Flexibility}, ...) map directly to concepts in the problem domain. The containment relationship between \( n1, n2, n3 \) and \( n4 \) is realized by means of links \( l1, l2 \) and \( l3 \) with label “ownedElement”. These links can be traversed in the other direction (from contained element to container) as well by means of the “namespace” label. Therefore, the underlying graph is not a directed graph. Moreover, it contains cycles: node \( n4 \) (\textit{CM}, the conceptual model of the Meeting Scheduler) is decorated with the “Conceptual Model” stereotype by means of link \( l4 \). This link can be edited by means of the context-sensitive menu shown in the bottom right corner of Figure 3. The “Conceptual Model” stereotype is defined by node \( n7 \) which is contained in node \( n6 \), representing the package defining the robustness modeling profile. Due to space limitations, \( n7 \) is not shown in Figure 3. However, the figure does show a node defining another stereotype: node \( n5 \) represents the definition of the “Foreign Key” stereotype from the profile for physical data modeling.

In the following, it will be shown how the entities in the robustness model can be created automatically from the classes in the conceptual model by means of the subgraph copy operator. The idea is to integrate the approach into model editors such that software engineers can focus on design decisions in the model refinement process rather than performing low-level copy operations manually.

2 \textbf{The CM2RM Transformation}

This section discusses the nature of the “Conceptual Model to Robustness Model” (CM2RM) transformation by presenting a structural and a behavioral model. The structural model will illustrate how the transformation is related to data from the input and output repositories. The subsection discussing the behavioral model will focus on the application of the subgraph copy operator.
2.1 Structural model of the transformation

As stated in the introduction, the structure of a modern model repository is defined by an object-oriented model. More specifically, such a “metamodel” represents the language of the models that can be stored in the repository. Since such metamodels define the input and output types of model transformations, they are discussed for the modeling languages used in the running example. Both the conceptual and the robustness models are expressed in the UML. Since the UML profiles that decorate the standard diagrams with a domain specific syntax are defined as UML models as well, the transformation under discussion only needs to interact with UML repositories.

Figure 4 shows a structural model of the CM2RM transformation. The interesting fact about this diagram is that one does not have to reason about the distinction between transformations, models, metamodels and metametamodels (as defined in [10]) to understand its meaning. It is a traditional class diagram that happens to be used in the context of model transformations but that does not presume any knowledge about the platform-specific repository code that is generated from it.

The CM2RM transformation contains a reference to one Model (defined in package org.omg.uml.modelmanagement) while such a Model can be transformed by many CM2RM transformations. The Model class, its association to the contained UML Model Elements (UmlClass, UmlPackage, State, ...) and other concepts from the UML are defined in the UML specification [12]. Since the repositories from popular UML tools are derived from
(often even generated from) this specification, the class implementing the Model concept in MagicDraw does not define a collection of CM2RM. Therefore, the UMLmodelOfTransformation association can only be traversed from CM2RM to Model. In order to apply the CM2RM transformation to the example from Section 1, the applicationModel reference needs to be initialized with the “Data” model (node n1 from Figure 3).

The CM2RM transformation can be parameterized with its applicationName attribute. This attribute determines what package inside the UML model will be looked up in order to transform the classes in its contained conceptual model to entities in its contained robustness model. When the CM2RM transformation would be applied to the example from Section 1 then applicationModel would be set to node n1 from Figure 3 while applicationName would be set to “Meeting Scheduler”. This would configure CM2RM for execution on n3.

While the CM2RM transformation could contain more methods for more complete case studies, this paper requires only one which is called “cmClasses2rm-Entities”. The method does not take any arguments and returns true or false based on the success of the transformation. The complete behavior of cmClasses2rmEntities has already been discussed before [3] but this paper provides a more comprehensive discussion of the copy operation used there.

2.2 Behavioral model of the transformation

The behavior of the cmClasses2rm-Entities method can be modeled in two phases. Firstly, the transformation needs to look up some meta-information for robustness modeling in the UML. Secondly, the classes are copied from the conceptual model to the robustness model and they are marked as entities by decorating them with the proper meta-information. Each of these steps can be implemented as a primitive graph transformation while the order between the primitives needs to be enforced by a controlled graph transformation rule. When using the story diagram syntax, such controlled graph transformation rules are specified as activity diagrams [15].

Figure 5 shows the primitive graph transformation rule for phase two. The rule is written in the UML profile for Story Driven Modeling (SDM) [16], into which the new copy operator is integrated. Unlike the Story Diagram syntax in Fujaba, the UML profile for SDM is based on class diagrams instead of object diagrams. This is primarily motivated by syntactical support for the visualization of attribute assignments. Moreover, using class diagrams to model rewrite rules allows one to show the cardinalities of link ends. This assists one to identify sources of multiple matches without looking at the type graph. The following subsections discuss the meaning of the rule in three steps.
2.2.1 Finding a Match

The nodes and edges that do not have a <<create>> stereotype in a primitive story specify a pattern that needs to be found in the input model. The pattern on Figure 5 starts from a node representing CM2RM’s applicationModel property. As stated, this property represents a handle to the input and output UML model of the discussed transformation (like node n1 from Figure 4). Just like the stereotypeOnCM, stereotypeOnRM and entityStereotype nodes, the applicationModel node is already bound: in fact, attributes of transformation classes are bound during the construction of the transformation object while the stereotype nodes are bound by the first primitive graph transformation rule of cmClasses2rmEntities.

From the applicationModel node, the rule searches for each recursively contained package with its name equal to the applicationName property of CM2RM. Such a UmlPackage is called wodnApplication and it represents the application containing the model that needs to be copied. Variable wodnApplication would be bound to node n3 from Figure 4. Note that all nodes and edges are typed and map directly to the class diagrams defining the UML metamodel. The UmlPackage nodes that can be reached from the applicationModel by recursively traversing outgoing links with association end name ownedElements are only bound to the wodnApplication node if they in turn contain a specific cm node in their outgoing ownedElement links. A node is bound to cm if it is of type Model and contains the already bound stereotypeOnCM node in its outgoing stereotype links. By specifying that cm contains zero or more nodes of type UmlClass with zero or more nodes of type Attribute, one does not constrain the search for cm any further.

2.2.2 Copying the Subgraph

The cm node needs to be copied since it is decorated with the <<copy>> stereotype. Apart from the cm node, all nodes and links on its outgoing composition path need to be copied as well. Note that all matches on this path are handled since the controlled graph transformation rule that executes this primitive rule marks it as <<loop>>. Without this directive,
the primitive rule shown on Figure 5 would copy only one matched class and attribute.

Implicitly, all attributes from a copied node are copied along. For example, since it is of type `Model`, the `name`, `isSpecification`, `isRoot`, `isLeaf`, and `isAbstract` attributes of node `classInCM` are copied implicitly. For the definition of the `Model` class, its attributes and superclasses, please refer to the metamodel in the UML 1.5 specification [12].

2.2.3 Using the Copy

When copying a subgraph, one should always store a reference to the copy. Otherwise, it wouldn’t become a subgraph of the host graph but just a standalone graph which may be inaccessible in subsequent graph transformations. The undesirable result would be an output model that would not contain the copy.

Creating a link is a standard graph transformation operation. In the UML profile for SDM one needs to specify a link between the nodes that need to be connected and label it with the `<create>` stereotype. Obviously the name of the link and the name and cardinality of the association ends need to conform to an association between the types of the node. Otherwise, the resulting graph would not conform to the output metamodel. In order to create a link from the `wodnApplication` node to the copy of the `cm` node, one needs an explicit notion of node copies in the graph transformation language.

Instead of representing the copy as a node in the transformation rule, the UML profile for SDM is extended with an `<onCopy>` stereotype. By specifying it on the `ownedElement` association end of the `<create>` link that connects `wodnApplication` with `cm`, one expresses that the link should be created to the copy of `cm` instead of to `cm` itself. When the `<onCopy>` stereotype would not be specified on `ownedElement` end, one would erroneously specify that the conceptual model needs to be added to the package it already resides in. The robustness model would be missing from the output model.

The `<onCopy>` instruction is also defined in the context of attribute assignments. This allows one to specify that the name of the robustness model, that is a copy of the `cm` node, needs to be changed to “RM”: the attribute assignment on the `cm` node is decorated with the `<onCopy>` stereotype. Without this stereotype one would change the name attribute of the conceptual model.

The `<onCopy>` instruction for `<create>` links is also applied to decorate all classes in the robustness model with the `<entity>` stereotype: the association end at the `classInCM` side of the stereotypes link is decorated with the `<onCopy>` stereotype while the association end at the `entityStereotype` side is left undecorated. The class in the robustness model is indeed a part of the copied subgraph while `entityStereotype` is a node in the original host graph.

The outgoing `type` link of node `a` (of type UML `Attribute`) needs to be copied to the target subgraph as well. A detailed discussion thereof is outside the scope of this paper. In summary, the rule on Figure 5 needs to be extended and an additional loop story is needed. By using multi-objects in combination with the `<onCopy>` instruction one can first create and then query the required traceability data.

3 Subgraph Copy operator

This section presents a syntax and an informal semantics for the proposed copy operator as an extension to the UML profile for Story Diagrams. It also compares two implementation
approaches to motivate the direction of the ongoing effort.

3.1 What

The proposed copy operator consists of the following syntactical constructs:

**copy** The `<copy>` construct allows one to specify what node represents the entry point to the subgraph that needs to be copied.

**composition** Starting from the `<copy>` node one can specify that a particular match path has composition semantics. Each node and link on this path will be copied.

**onCopy** The `<onCopy>` construct can be used to indicate that a particular instruction needs to be executed on the copy of an element instead of on the element itself. The construct is defined on (1) association ends of `<create>` links and (2) attribute assignments.

(i) By specifying `<onCopy>` on the source (or target) end of a `<create>` link, one specifies that the link needs to be created from (or to) the copy of the node at that association end.

(ii) An assignment on an attribute from a node on the composition path, that is marked as `<onCopy>`, is executed on the attribute from the copy of this node instead of on that from the node itself.

Not every application of these directives results in a valid use of the copy operator. Therefore, the following new well-formedness rules (WFRs) are defined for the UML profile for SDM:

- At least one link should be created from the host graph to a node from the copied subgraph. More specifically, at least one link should be created to the `<copy>` node or a node on its outgoing composition path.
- The `<onCopy>` instruction should only be applied (1) on attributes inside a copied node, or (2) on association ends connected to a copied node.
- A node should be part of at most one composition. Otherwise, it would be ambiguous what should be the container of such nodes’s copy. (see Figure 6).

Appendix A formalizes the first WFR in OCL. The specification is defined within the context of the Class class from the Foundation::Core package of the UML metamodel. Every instance of that metaclass needs to respect the invariant defined from line 56 onwards. One can use the OCLE tool [7] to confirm that the “cm: Model” node from the transformation rule in Figure 5 respects this invariant. The constraint makes use of three OCL helper attributes defined on line 43 to 49 and 50. The `trfoPkgNodes` attribute represents all nodes from the copy transformation rule under study. The `copiedNodes` and `nonCopiedNodes` attributes divide this set of nodes into the nodes that will or will not be copied respectively. These attributes are defined using the helper operations specified on line 10 and 30. The OCL specification of the latter two WFRs is left out due to space considerations but can be obtained from the authors.

3.2 How

Two implementation approaches have been investigated: a direct model-to-code transformation approach and a model-to-model transformation approach. All related artifacts are
publically available in the MoTMoT project [9]. MoTMoT (Model driven, Template based
Model Transformer) is a “model transformation” code generator based on the AndroMDA
3.x framework. It uses Freemarker templates to translate UML models (conforming to the
profile for SDM [16]) into Java code conforming to the JMI standard.

The straightforward approach for adding support for the copy operator is to extend
the Freemarker templates that handle the code generation for existing SDM constructs.
At a very high level of abstraction, the generated code should implement the following
algorithm:

(i) collect all nodes matching the composition path specified in a copy rule,
(ii) in the case of a complete match: (a) copy these nodes, including all their attributes,
and (b) execute \(<\text{onCopy}>\) attribute assignments,
(iii) maintain a map of traceability links between nodes and their copies,
(iv) use the traceability map to create the composition links between the copies as soon as
all of the copy nodes have been created,
(v) create \(<\text{onCopy}>\) links using the same approach.

In practice, the complexity of the Freemarker templates reached an unacceptable level after
implementing step (iv).

Therefore, current development is focussed on a model-to-model transformation ap-
proach that leaves the code templates unchanged. Story Diagrams are used to transform
models conforming to the profile discussed in Section 3.1 into models conforming to the
SDM profile without the copy operator. The generated Story Diagrams realize the be-
havior of the copy operator by means of a traceability metamodel and the introduction of
additional stories and control structures. The complete transformation is still complex but
thanks to the use of an intermediate layer and the modularity mechanisms of Story Dia-
grams, the complexity can be decomposed into manageable parts. Apart from the facilities
for manageing the transformation complexity, the model-to-model transformation approach
is promising due to portability opportunities:

• It does not involve a further investment into code specific to the MDR/JMI platform.
  Migrating the Freemarker templates to platforms such as EMF does not become harder
  than before.
• With reasonable effort, it should be possible to deploy the story diagrams that are gener-
  ated by the model transformation on other SDM platforms such as Fujaba.

An upcoming article will discuss this model transformation in more detail.
Related Work

Subgraph copying was first investigated in the context of hierarchical graph transformation. This work assumes that one can decompose the transformed graphs into “frames” where edges are not allowed to cross frame boundaries. Drewes, Hoffmann and Plump acknowledge that nested visual languages like the UML require a more flexible decomposition mechanism but require the assumption for proving that rewrite rules do not violate grammatical constraints [2].

Although the hierarchical approach presents the interesting idea of automatically copying all edges between the nodes in a frame, it should be extended for performing copy operations in a more general sense. An \texttt{<onCopy>} instruction such as the one presented in this paper could be defined to specify that, for example, the copy of a subgraph should not contain particular edges while including others that do not originate from the source subgraph. Another limitation of the hierarchical approach is that frames are not proposed to be defined on a rule by rule basis. Hoffmann et al. tackled this issue by allowing “shape grammars” to define the structure of a frame variable in the scope of a rewrite rule instead of in the scope of the complete rewriting system [1].

This paper presents a very specific model refinement case study. However, the copy operator can be used for transforming any typed graph with edge labels and attributed nodes. More specifically, it can be used for implementing refactorings. Van Eetvelde et al. have proposed the use of graph variables and cloning for raising the abstraction level of graph transformation rules in this context [18]. Applying the copy operator on the Push Down Method refactoring defined on a metamodel for Java appears to be promising but the validation of this work is still in progress. This work builds upon the case study from Hoffmann [4] by considering the attributes and links from syntax nodes within method bodies in more detail. We are evaluating whether or not the use of control structures such as a Story Diagram \texttt{<loop>} leads to more complex rules than those making use of graph variables.

Conclusion

This paper introduces a graph transformation operator for subgraph copying. The operator allows one to define refinements on models conforming to UML profiles in a concise manner. More specifically: copying model elements from one domain specific model to another one, changing attribute values of copied elements and attaching links to the copied elements can be done in one rewrite rule. The operator has been integrated in Story Diagrams, a controlled graph transformation language with a wide user base. The extension has been implemented in the UML profile for SDM such that any UML 1.5 compliant editor can be used to model model transformations. The implementation effort for the transformation engine is focussed on an SDM model transformation from the extended SDM profile to the profile version without the operator. The operator appears to be applicable in the context of model refactoring as well but more validation is required to get a better understanding of its applicability and limitations.
References


Appendices

A  OCL for Well-Formedness Rule

```
context Class
-- Return transitive closure of the "ownedElement" links starting from s
def: let ownedElementTC(s: Set(ModelElement)): Set(ModelElement) =
if s->includesAll(
  s->select (me| me.oclIsKindOf(Namespace))
)->collect (me2) me2.oclAsType(Namespace)
).ownedElement->asSet()
) then s
else ownedElementTC(
  s->union(
    s->select (me| me.oclIsKindOf(Namespace))
)->collect (me2) me2.oclAsType(Namespace)
).ownedElement->asSet()
)
)

-- Return from a primitive story all nodes that will be copied
def: let allCopiedNodes(s: Set(Classifier)): Set(Classifier) =
s->select (c| -- Return all classes
  c.oclAsType(Classifier)
)->collect (class | class.oclAsType(Classifier))
).asSet in
let trfoPkgNodes: Set(Classifier) =
ownedElementTC(Set(self.namespace))->select (element |
  element.oclAsType(Classifier))
)->collect (class | class.oclAsType(Classifier))
)->asSet in
let copiedNodes: Set(Classifier) = allCopiedNodes(trfoPkgNodes) in
let nonCopiedNodes: Set(Classifier) = -- trfoPkgMEs minus copiedNodes
  trfoPkgNodes->reject (el |
    copiedNodes->exists (copiedNode |
      el=copiedNode -- Reject elements that are copied (set 'minus').
    )
  )
)
in
hasStereotype(self, "copy") implies -- When applying the copy instruction,
  nonCopiedNodes.association->exists (end | -- the non-copied nodes should be
    hasStereotype (end.association,
      hasStereotype (end.association, "create") and
      -- representing a "create" link
      end.oclAsType(Namespace)->asSet()->size = 0)
    )
  )
  )
inv:
let trfoPkgNodes: Set(Classifier) =
ownedElementTC(Set(self.namespace))->select (element |
  element.oclAsType(Classifier))
)->collect (class | class.oclAsType(Classifier))
)->asSet in
```

Actual WFR: as soon as the "copy" instruction is issued, the copied sub-graph needs to be connected to the host graph by means of a "create" link.

---

Van Gorp, Schippers and Janssens

12