Copying Subgraphs within Model Repositories

Pieter Van Gorp, Hans Schippers, Dirk Janssens

Formal Techniques in Software Engineering
University of 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 the ongoing implementation effort makes optimal reuse of evaluation code for existing language constructs.

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 [14]), and exposing a query and transformation API (like OCL [12] and JMI [5]). Any program with the purpose of creating or changing models starting from at least one model 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 and interfacing the low-level API of mainstream model repositories by means of a compiler.

The data definition languages (like MOF [11] and ECORE [9]) for modern model repositories (like MDR [7] and EMF [9]) 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.

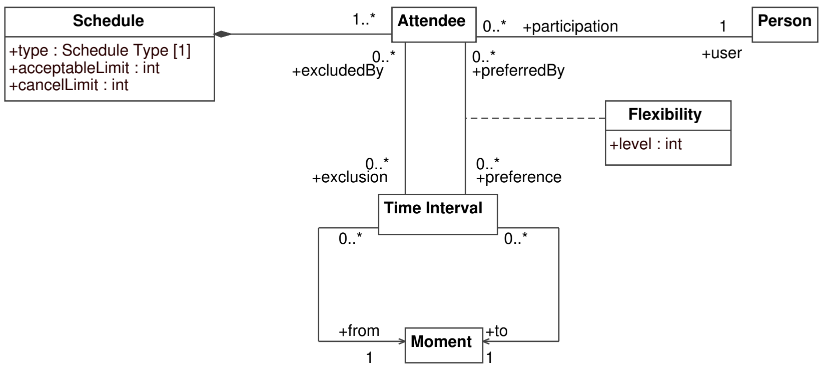


Figure 1. Conceptual Model of a Meeting Scheduler application.

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 copy operator as an extension to Story Diagrams [6]. 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 learnt.

1 Motivating Example Models

Figure 1 shows a conceptual model (CM) of a Meeting Scheduler application, specified in UML syntax [13]. 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 in 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 verbs and nouns 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 *[15]* 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 *[15]*) 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

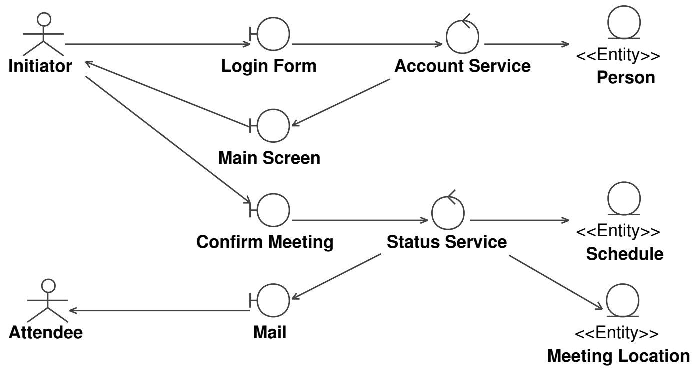


Figure 2. Robustness Model of a Meeting Scheduler application.

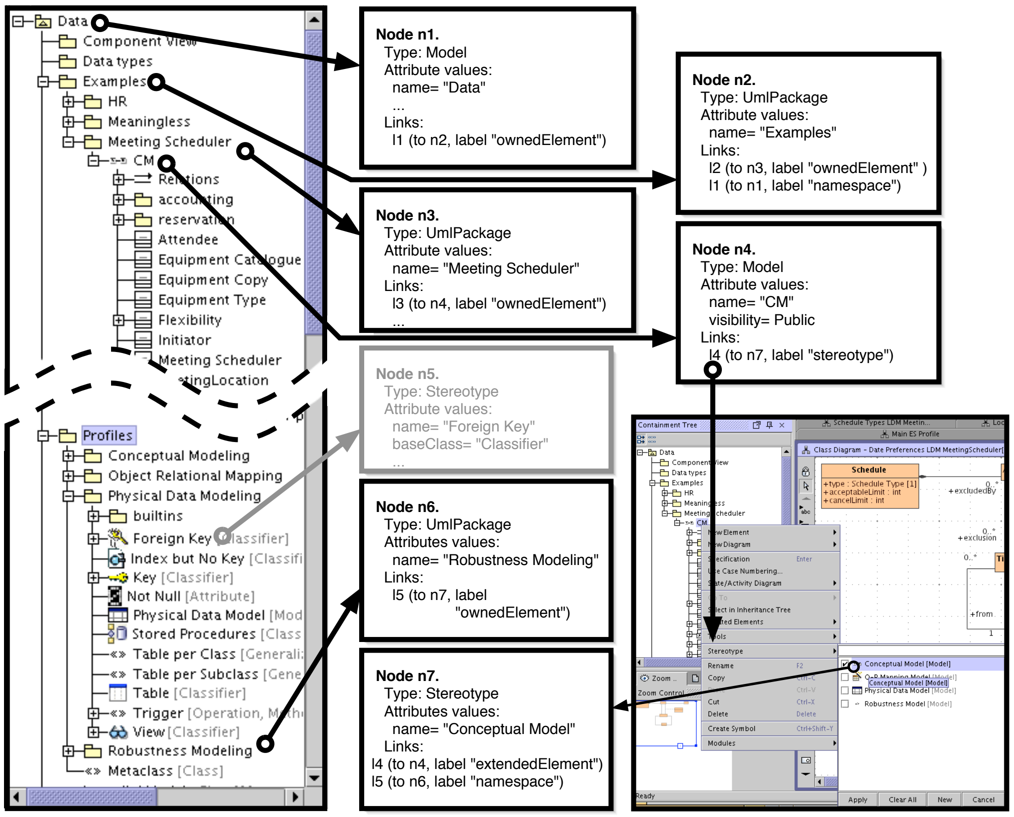
and the definitions of the profiles used within these examples. All examples reside in node *n2* of type *UmlPackage* and with name "Examples".

Node *n3* represents the actual Meeting Scheduler sample. This *UmlPackage* contains node *n4* which represents the conceptual model of the Meeting Scheduler. All its contained classes (like *Attendee*, *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 (*CM*, the conceptual model of the Meeting Scheduler) is decorated with the "Conceptual Model" stereotype by means of link *l4*. In MagicDraw, 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 The *CM2RM* transformation system

This section discusses the nature of the "Conceptual Model to Robustness Model" (*CM2RM*) transformation system 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.



Node n1.
Type: Model
Attribute values:
name= "Data"
...
Links:
l1 (to n2, label "ownedElement")

Node n2.
Type: UmlPackage
Attribute values:
name= "Examples"
Links:
l2 (to n3, label "ownedElement")
l1 (to n1, label "namespace")

Node n3.
Type: UmlPackage
Attribute values:
name= "Meeting Scheduler"
Links:
l3 (to n4, label "ownedElement")
...

Node n4.
Type: Model
Attribute values:
name= "CM"
visibility= Public
Links:
l4 (to n7, label "stereotype")

Node n5.
Type: Stereotype
Attribute values:
name= "Foreign Key"
baseClass= "Classifier"
...

Node n6.
Type: UmlPackage
Attributes values:
name= "Robustness Modeling"
Links:
l5 (to n7, label "ownedElement")

Node n7.
Type: Stereotype
Attributes values:
name= "Conceptual Model"
Links:
l4 (to n4, label "extendedElement")
l5 (to n6, label "namespace")

Figure 3. Relation between the UML editor and the underlying model graph.

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 tranformation system under discussion only needs to interact with UML repositories.

Figure 4 shows a structural model of the *CM2RM* transformation system. 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 [11]) 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

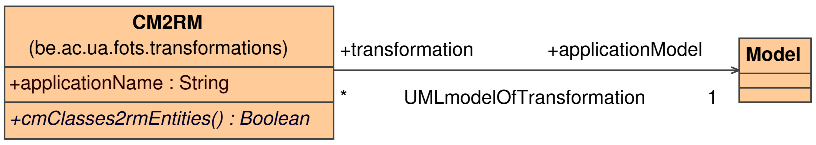

CM2RM
(be.ac.ua.fots.transformations)
+applicationName : String
+cmClasses2rmEntities() : Boolean
+transformation
+applicationModel
*
UMLmodelOfTransformation
1
Model

Figure 4. Metamodel defining a repository that allows one to store the CM2RM transformation with a direct reference to the UML model it transforms. The class "Model" is imported from the UML metamodel.

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 [13]. 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*s. Therefore, the *UMLmodelOfTransformation* association can only be traversed from *CM2RM* to *Model*. In order to apply the *CM2RM* transfomation 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* transfomation 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 "cmClasses2rmEntities". 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 *cmClasses2rmEntities* 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 [6].

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

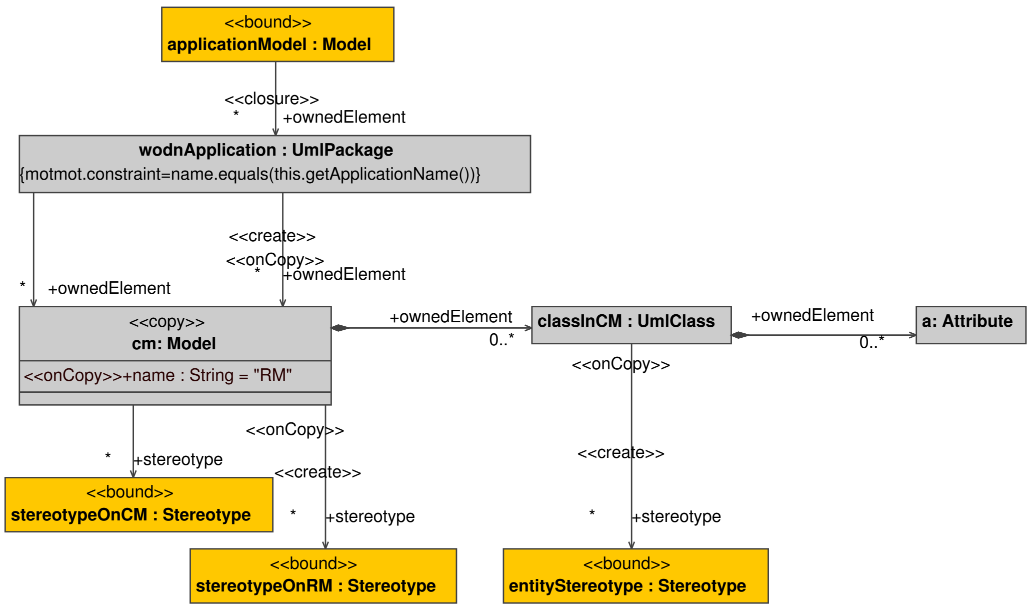


Figure 5. Primitive graph transformation rule applying the new copy operator.

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 system (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 already 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 further constrain the search for *cm*.

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 [13].

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 unaccessible in subsequent graph transformations. The undesired result would be an output model that does 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 the node *a* (of type UML *Attribute*) needs to be copied to the target subgraph as well. Again, one cannot use a conventional <<create>> operation from standard graph transformation since it would create a new link between nodes in the source subgraph. In the example given, one wants to create a *type* link between the copy of the *a* node and the copy of the *classInCM* node. For this purpose, one decorates both association ends of the link to be copied with the <<onCopy>> stereotype.

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 just 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:

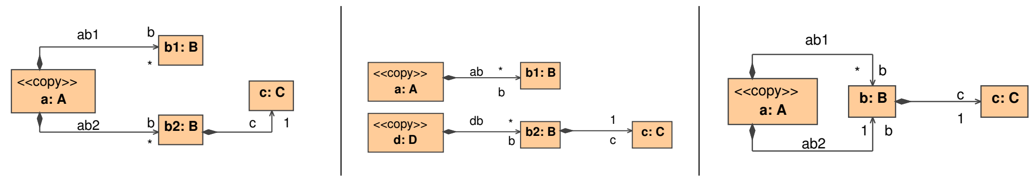


Figure 6. From left to right: two valid rewrite rules and an invalid one. In the rightmost rule, it is unclear whether the *c*'s contained by the *b*'s from *ab1* should be copied, or those contained by the *b*'s from *ab2*, or both. The leftmost rewrite rule illustrates how one can unambiguously specify that for the *b*'s from *ab2* one should copy the contained *c*'s while this is not the case for those from *ab1*. The middle rewrite rule illustrates that within one rewrite rule one can use multiple <<copy>> nodes as long as their composition paths do not overlap.

- One should create a link from the host graph to a node from the copied subgraph. More specifically, one should create a link to the <<copy>> node or a node on its outgoing composition path. Appendix A formalizes this 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 [8] 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 *tfroPkgNodes* attribute represents all nodes from the copy transformation rule under study. The *copiedNodes* and *nonCopiedNodes* attributes devide 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 <<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).

The OCL specification of the latter two WFRs 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 [10]. 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 <<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 <<onCopy>> links using the same approach.

In practice, the complexity of the Freemarker templates reached an unacceptable level after implementing step (4).

Therefore, current development is focussed on a model-to-model transformation approach 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 behavior 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 (1) the use of an intermediate layer and (2) the modularity mechanisms of Story Diagrams, 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 generated 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 prooving that rewrite rules do not break 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 <<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 presented 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 further on the case study from Hoffmann [4] by considering the attributes and links from syntax nodes within method bodies in more detail. One is evaluating whether the use of control structures such as a Story Diagram <<loop>> actually 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 the general purpose MagicDraw UML 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 discover its complete applicability and its limitations more precisely.

References

[1] Berthold Hoffmann. Abstraction and Control for Shapely Nested Graph Transformation. *Fundamenta Informaticae*, 58(1):39–65, 2003.

[2] Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, 64:249–283, 2002.

[3] Pieter Van Gorp and Dirk Janssens. CAViT: a consistency maintenance framework based on visual model transformation and transformation contracts. In J. Cordy, R. Lämmel, and A. Winter, editors, *Transformation Techniques in Software Engineering*, number 05161 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[4] Berthold Hoffmann, Dirk Janssens, and Niels Van Eetvelde. Cloning and expanding graph transformation rules for refactoring. In *International Workshop on Graph and Model Transformation*, Tallinn, Estonia, 2005. A satellite event of GPCE'05.

[5] Java Community Process. Java metadata interface (JMI) specification – JSR 000040, June 2002.

[6] Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp, and Daniel Varro. *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*, chapter Using Graph Transformation for Practical Model Driven Software Engineering. Springer-Verlag, 2005.

[7] Sun Microsystems. NetBeans Metadata Repository, 2002. http://mdr.netbeans.org/.

[8] D. Chiorean A. Carcu M. Pasca C. Botiza H. Chiorean S. Moldovan. *Studia Informatica*, volume XLVII, chapter UML Model Checking, pages 71–88. Babes-Bolyai University, 400084 Cluj-Napoca, Romania, 2002.

[9] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks. International Business Machines, January 2004.

[10] Olaf Muliawan, Hans Schippers, and Pieter Van Gorp. Model driven, Template based, Model Transformer (MoTMoT). http://motmot.sourceforge.net/, 2005.

[11] Object Management Group. *Meta Object Facility (MOF) specification*. Object Management Group, 2002. Version 1.4. Available for download at url http://cgi.omg.org/cgi-bin/doc?formal/2002-04-03.

[12] Object Management Group. UML 2.0 OCL Final Adopted specification. ptc/03-10-14, 2003.

[13] Object Management Group. Unified Modeling Language (UML), March 2003. version 1.5. document ID formal/03-03-01.

[14] Object Management Group. XML Metadata Interchange (XMI), v2.0. formal/03-05-02, 2003.

[15] Doug Rosenberg and Kendall Scott. *Use case driven object modeling with UML: a practical approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[16] Hans Schippers, Pieter Van Gorp, and Dirk Janssens. Leveraging UML profiles to generate plugins from visual model transformations. *Software Evolution through Transformations (SETra). Satellite of the 2nd Intl. Conference on Graph Transformation*, 127(3):5–16, 2004.

[17] E Seidewitz. What models mean. *IEEE Software*, 20, Sept.-Oct. 2003.

[18] N. van Eetvelde and D. Janssens. Extending graph rewriting for refactoring. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *2nd Int'l Conference on Graph Transformation*, volume 3256 of *Lecture Notes in Computer Science*, pages 399–415. Springer-Verlag, 2004.

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(me1|
     me1.oclIsKindOf(Namespace)
   )->collect(me2|
     me2.oclAsType(Namespace)
   ).ownedElement->asSet()
 ) then s
 else ownedElementTC(
   s->union(
     s->select(me1|
       me1.oclIsKindOf(Namespace)
     )->collect(me2|
       me2.oclAsType(Namespace)
     ).ownedElement->asSet()
   )
 )
 endif

 -- 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
    hasStereotype(c, "copy") or -- that have a <<copy>> stereotype
    c.association->exists(end| -- or connected to
      end.association.connection->exists(end2| -- an association
        end2<>end and -- of which the other end
        end2.aggregation=AggregationKind::composite -- is of type composite.
      )
    )
  )
 -- 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
inv:
 let trfoPkgNodes: Set(Classifier) =
  ownedElementTC(Set{self.namespace})->select(element |
    element.oclIsKindOf(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,  -- connected to an association
                "create") and -- representing a <<create>> link
      end.association.connection->select(end2| -- and containing
        end<>end2 -- another end that
      ).participant->exists(copiedNode| -- is connected to a node
        copiedNodes->includes(copiedNode) -- that *is* copied.
      )
  )
```