Transformation Language Integration based on Profiles and Higher Order Transformations

Pieter Van Gorp, Anne Keller and Dirk Janssens

University of Antwerp
{pieter.vangorp,anne.keller,dirk.janssens}@ua.ac.be

Abstract. For about two decades, researchers have been constructing tools for applying graph transformations on large model transformation case studies. Instead of incrementally extending a common core, these competitive tool builders have repeatedly reconstructed mechanisms that were already supported by other tools. Not only has this been counter-productive, it has also prevented the definition of new language constructs independently of a specific transformation tool. Moreover, it has complicated the comparison of transformation languages. This paper describes a light-weight solution to this integration problem. The approach is based on executable transformation modeling using a small UML profile and on higher order transformations. It enables the integration of graph transformation tools such as Fujaba, VMTS and GReAT. The paper illustrates the approach by discussing the contribution of a Copy operator to any of these tools. Other language constructs can be realized similarly, without locking into specific tools.

1 Problem: Lack of Portability and Reuse

This paper tackles the integration problem of model transformation tools in general but focuses on graph transformation tools. Graph transformation gained industrial credibility in the nineties, thanks to the application of the Progres language (and tool) within industrial tool integration projects [1]. One of Progres' shortcomings was that it relied on a rather limited language for metamodeling. Moreover, some of its control flow constructs were rather ad-hoc compared to standard notations such as activity diagrams. After working within the Progres team, Zündorf showed how these two limitations could be overcome by defining "Story Diagrams" as a new graph transformation language based on the UML [2]. Due to some interoperability issues with the C implementation of the Progres tool [3], a new Java based tool called Fujaba was implemented for these Story Diagrams [4]. Unfortunately, no mechanisms for reusing the advanced pattern matching libraries of Progres were available. Consequently, a lot of existing functionality had to be implemented again. In fact, some of Progres' language constructs are not yet supported by any other tool. In parallel to Fujaba, another graph transformation tool called GReAT was implemented for the Windows platform [5]. Similar to Fujaba and Progres, GReAT also supported pattern matching, rewriting, rule iteration, etc. However, one did not yet define metamodels for integrating the different tools' editors, analyzers or compilers.

Although this kind of prototyping was natural for the first generation of tools, the disadvantages became clearly visible when even more tools were constructed. With the advent of the MDA, a new generation of tools (such as MOLA [6] and VMTS [7]) was constructed by competing research groups. Unfortunately, these tools have proposed yet another syntax for graph transformation constructs with the same underlying semantics. Moreover, none of the graph transformation tools rely on libraries to reuse existing infrastructure for pattern matching, control flow, etc. Although the existence of multiple tools ensures healthy competition, today's tool landscape suffers from (1) the lack of portability of transformation models across different editors, and (2) the lack of reuse of compiler/interpreter infrastructure. This paper presents language engineering techniques to solve these two problems.

The remainder of the text is structured as follows: Section 2 explains how standard UML profile syntax enables editor and repository independence, Section 3 describes how transformation language "behavior" can be shared across tools, Section 4 describes how the approach has been applied to contribute a new transformation language construct in a tool-independent manner, Section 5 discusses how the approach still applies for quite advanced and seemingly tool-specific language constructs, Section 6 presents related work, Section 7 presents an outlook for future work and Section 8 concludes.

2 Extensible Transformation Language: Standard Profile

The first step to transformation tool integration is the agreement on a metamodel for all mainstream transformation language constructs. The metamodel should meet the concerns of transformation writers, while these may be in conflict with those of transformation tool builders. Since transformation writers need to write and debug transformation models, they want a transformation language with a human-readable concrete syntax. Moreover, they want to be able to use the editor of the most user-friendly transformation tool (automatic completion, interactive debugging, ...). Finally, they may want to rely on the optimizer of one tool and potentially deploy the result using integration components of yet another transformation framework (EMF compatible Java, JMI compatible Java, Windows compatible C++, Unix compatible C++, ...). Since transformation tool builders need to provide such functionality, they want a simple metamodel and reuse existing software whenever possible.

2.1 Metamodel: UML 1.5

It turns out that a small subset of the UML 1.5 metamodel reconciles the requirements of transformation writers with those from tool builders. More specifically, UML class and activity diagrams resemble the concrete syntax of popular graph transformation languages such as Story Diagrams, MOLA and VMTS quite closely.

UML 1.5 relates to an ISO standard and is supported by quite a number of mature tools (commercial as well as open source). In contrast, UML 2 based tools tend to differ more at the metamodel level. Moreover, the UML 2 metamodel is designed to integrate several diagram types that are irrelevant for transformation modeling. This complicates the metamodel structure and thus unnecessarily increases the implementation effort for tool builders. Transformation writers that insist on using a UML 2 editor can apply a standard converter for mapping UML 2 profile, class and activity models back to UML 1.5.

At the other end of the *universal-to-domain-specific* spectrum, the Graph Transformation eXchange Language (GTXL [8]) does not suffer from the metamodel complexity overhead associated with the UML standards. However, besides more problems discussed in Section 6, GTXL models have no standard *concrete syntax* beyond XML. Therefore, it does not satisfy the readability concerns of transformation writers. Section 6 also discusses the drawbacks of the Queries/Views/Transformations (QVT [9]) languages. Among other problems, QVT standardizes even more than is supported by mainstream transformation tools. Therefore, it requires much more implementation effort than the lightweight UML approach presented here.

2.2 Extensions for Transformation Modeling

Among the domain-specific extensions, a UML based transformation modeling language needs a means to relate an activity in a control flow to a particular rewrite rule. Such extensions could be realized using MOF extensions of the UML metamodel. However, this approach is undesirable for several reasons. First of all, such heavy-weight extensions break the compatibility of the standard transformation language with general purpose UML editors. More specifically, at the implementation level there would be several incompatibilities in MOF repository interfaces, XMI schema's, etc. Similarly, transformation tools that would use MOF extensions to realize language constructs beyond the core transformation standard would break compatibility with tools that only support that core.

Fortunately, the UML metamodel is designed for enabling language extensions without the need for a metamodel change. Essentially, there is a mechanism to introduce (1) the equivalent of a new metaclass (MOF M2) as a library element (MOF M1), and (2) the equivalent of a new MOF attribute (MOF M2) as another kind of library element (MOF M1). The mechanism is known as "UML Profiles". New *virtual* metaclasses are called "Stereotypes" and new *virtual* meta-attributes are called "Tagged Values".

Table 1 displays the concrete stereotypes and tag definitions from the proposed UML Profile for Transformation Modeling. The *core* profile only supports the basic graph transformation concepts: it has the notion of a rewrite rule (matched elements, created elements, deleted elements and updated elements) and control flows (iterative loops, conditionals and called transformations).

Profile Element	**Related UML Constructs**	**Meaning**
«ModelTransformation»,	Method	Callable Transformation
motmot.transformation tag	Method, Package, Activity Diagram	Link from method to controlled rewrite rules
«loop», «each time»	State Transition	Iterative loop over rule, iterative execution of nested flow
«success», «failure»	Transition	Match/Mismatch of rule
motmot.constraint tag	State, String Class, String	Application condition on state Application condition on node
«link», «code»	State, String	Call to transformation
motmot.transprimitive tag	State Package, Class Diagram(s)	Link from state in a flow to a rewrite rule
motmot.metatype tag,	Class	Type of rewrite node
«bound»,	Class	Parameter node, or node matched by previously executed rule
«create», «destroy»	Class, Association	Created/Destroyed elements
	Attribute (initial value)	Node attribute updates
«closure»,	Association	Transitive closure for link label

Table 1. A basic (*core*) UML Profile for Transformation Modeling.

These syntactical constructs already provide a large portion of the expressiveness of today's graph transformation languages. Nevertheless, we do not want to standardize all transformation tools prematurely to this common denominator. Instead, new stereotypes and tagged values can be defined in UML models that extend the core profile. Such extensions can be made publicly available to extend any other UML editor with the new transformation language constructs. In summary, the proposed standard metamodel for exchanging transformation models consists of the class diagram (*core*), activity diagram (*activities*), and profile (*extension mechanisms*) packages of the UML metamodel.

2.3 Evaluation of the Core Transformation Modeling Profile

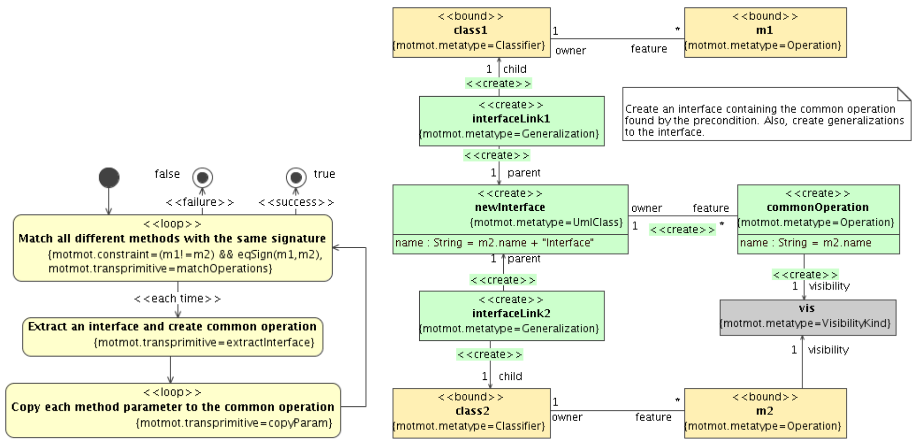
To validate whether the core transformation profile is still human-friendly, it has been used to model a variety of transformation problems using off-the-shelf UML tools. More specifically, several refactorings (*Pull Up Method*, *Encapsulate Field*, *Flatten Hierarchical Statemachine* and others) and refinements (e.g., a translation of UML to CSP) have been modeled using commercial tools, without any plugins specific to the proposed transformation modeling profile [10]. An cognitive evaluation based on the framework of Green [11] confirms that the standard transformation language is usable even without any user interface implementation effort from transformation tool builders. Obviously, in an industrial context, one would rely on auto-completion and debugging plugins. However, the effort required for constructing such plugins would still be smaller than the effort for constructing a QVT editor from scratch.

As discussed above, the proposed integration architecture enables transformation tool builders to specialize in complementary features. One tool builder (e.g., the MOLA team) can provide syntactical sugar on top of the proposed profile while another one (e.g., the VMTS team) can invest in optimizations. One tool builder (e.g., the MOFLON team) can provide integration with MOF based repositories (EMF and JMI) while another tool builder (e.g., the Progres team) can offer a C++ backend. To evaluate this architecture in practice, the MoTMoT tool provides a bridge from the proposed profile to JMI based repositories without offering a dedicated transformation model editor. Instead, MoTMoT relies on third-party editors such as MagicDraw 9 or Poseidon 2 [12].

To illustrate the core profile, Figure 1 shows the control flow and a rewrite rule for the *Extract Interface Where Possible* refactoring. Within the context of a package, the refactoring should mine for commonality between classes. More specifically, by means of an interface it should make explicit which classes implement the same method signatures. The transformation presented in this paper has been created by master-level students, after a three hour introduction to the proposed approach to transformation modeling.

The transformation flow shown on Figure 1 (a) controls the execution of three rewrite rules. The *matchOperations* rule associated with the first activity is used to check the precondition of the refactoring. More specifically, it matches only for those methods that have the same signature. Since the activity is decorated with the *«loop»* stereotype, it is executed for every possible match. Moreover, the two subsequent activities are executed for each match as well due to the *«each time»* stereotype on the outgoing transition. Due to the absence of a *«loop»* stereotype, the second activity is executed only once for each match of the first activity. The second activity is associated with the rewrite rule shown in Figure 1 (b). The *class1*, *class2*, *m1* and *m2* nodes are bound from the execution of the previous rule. The rule matches the visibility of the second method (*m2*) in order to create a copy of that method in the node *commonOperation*. The operation is embedded in the *newInterface* node that is newly created, as specified by the *«create»* stereotype. The newly created *interfaceLink1* and *interfaceLink2* elements ensure that *class1* and *class2* implement the new interface.

The third activity is executed for all possible matches of its rewrite rule. In this rewrite rule (not shown due to space considerations), all parameters from the new operation are created based on the parameters of *m1* and *m2*. To complete the discussion of this example, consider the two final states shown in the top of Figure 1 (a). The left final state is reached when the precondition (activity 1) fails. The right final state is reached otherwise. The final states are used to return a success boolean value for the transformation that is modeled by Figure 1 (a). Specific tools provide small variations on the syntactical constructs presented in this example. For example, the Fujaba and MOLA tools visually embed the rewrite rules (cfr., Figure 1 (b)) inside the activities of the control flow (cfr., Figure 1 (a)). However, for example VMTS does not rely on such an embedded representation. Therefore, the proposed concrete syntax resembles mainstream graph transformation languages in a satisfactory manner.


false
true
<<failure>>
<<success>>
<<loop>>
Match all different methods with the same signature
{motmot.constraint=(m1!=m2) && eqSign(m1,m2), motmot.transprimitive=matchOperations}
<<each time>>
Extract an interface and create common operation
{motmot.transprimitive=extractInterface}
<<loop>>
Copy each method parameter to the common operation
{motmot.transprimitive=copyParam}
<<bound>> class1 {motmot.metatype=Classifier}
<<bound>> m1 {motmot.metatype=Operation}
owner
feature
child
<<create>>
<<create>> interfaceLink1 {motmot.metatype=Generalization}
Create an interface containing the common operation found by the precondition. Also, create generalizations to the interface.
parent
<<create>> newInterface {motmot.metatype=UmlClass}
name : String = m2.name + "Interface"
<<create>> commonOperation {motmot.metatype=Operation}
name : String = m2.name
visibility
vis {motmot.metatype=VisibilityKind}
<<create>> interfaceLink2 {motmot.metatype=Generalization}
<<bound>> class2 {motmot.metatype=Classifier}
<<bound>> m2 {motmot.metatype=Operation}

(a) Control Flow (b) Rewrite Rule

Fig. 1. *Extract Interface* refactoring, modeled using the core profile.

3 Exchangeable Semantics: Higher Order Transformations

In the previous section we showed how transformation languages can be integrated at the syntactical level. This relates to the editors and repositories of transformation tools. In this section we show how the semantics of the transformation modeling profile can be extended with the help of higher order transformations. This relates to the compilers or interpreters of transformation tools. In general, higher order transformations are defined as transformations that consume and/or produce transformation models. In the context of this paper, the role of higher order transformations is to transform transformation models that conform to an extension of the profile into transformation models that conform to the profile without the new language construct. The semantics of the core profile is assumed to be well-understood: it is merely a standard syntax for what has been applied for two decades already.

A key to the proposed integration architecture is that the higher order transformations themselves are modeled using the core profile for transformation modeling. Therefore, any transformation engine that supports the core profile can execute the higher order transformations. Consequently, any such engine can normalize transformation models that apply a new language construct into more primitive transformation models. In general, a transformation tool may execute a series of publicly available higher order transformations before executing the result on its native graph transformation engine. In the case of performance problems (that we have not encountered so far), some tools might directly generate optimized code for particular transformation language constructs. In that case, the higher order transformations may be used for checking the correctness of the optimized code.

In summary, higher order transformations ensure that new graph transformation language constructs can be executed by all tools that support the core transformation modeling profile. On the one hand, this rewards the creator of the language construct with an instant adoption across a set of tools. On the other hand, this relieves tool builders from implementing constructs that have been defined by others.

4 Example Profile Extension: Copy Operator

This section explains how we contributed a *Copy* operator according to the proposed integration architecture. Subsection 4.1 describes the syntactical extension of the profile. Subsection 4.2 presents the higher order transformation that makes the operator executable in a tool-independent manner. Remark that copy operations can already be *programmed* with the core profile: one can specify match and create operations explicitly for all elements that need to be copied. For example, the rewrite rule from Figure 1 (b) explicitly enumerates which properties (*name* and *visibility*) from the *m2* node need to be copied to the *commonOperation* node. However, such transformation specifications are undesirably low-level. One would like to model declaratively which nodes should be copied without worrying about the attributes (cfr., *name*) and associations (cfr., *visibility*) of the related metaclasses (cfr., *Operation*).

4.1 Syntax

The need for a course-grained *Copy* operator has been acknowledged outside the graph transformation community as well [13]. Therefore, we propose first-class transformation language support for *modeling* the following activities:

- matching a *tree* pattern that defines the scope of the subgraphs that need to be copied,
- preserving internal edges that are not contained in that tree pattern,
- performing side-effects on the resulting copy: adding/removing nodes or edges and updating node attributes in the target subgraph.

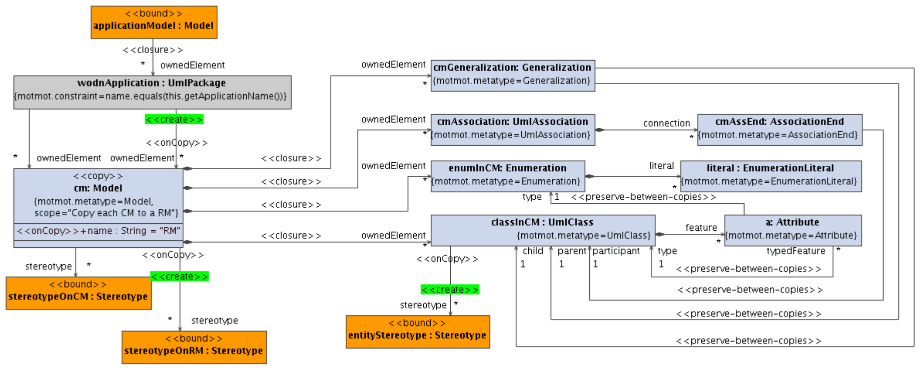
Therefore, the following language constructs need to be added:

- the «*copy*» construct allows one to specify what node represents the entry point to the subgraph that needs to be copied.
- starting from such a «*copy*» node one can specify that an underlying tree pattern has *composition* semantics. Each node and edge matched by this pattern will be copied.
- the «*onCopy*» construct can be used to indicate that a particular instruction («*create*», «*destroy*», «*update*») needs to be executed on the copy of an element instead of on the element itself.
- the «*preserve-between-copies*» construct can be used to indicate that an edge in the subgraph needs to be copied to the target subgraph.

By packaging these three stereotypes («*copy*», «*onCopy*», «*preserve-between-copies*») in a library, any generic UML editor can be used to model copy operations concisely [14].

4.2 Semantics

This section describes the higher order transformation that normalizes the constructs presented in the previous section back into more primitive graph transformation constructs. Before presenting some fine-grained mapping rules between models from the *Copy* profile and those from the *core transformation* profile, we consider an example of an input and output transformation model. The input model contains the copy specific stereotypes presented in the previous section. The output model conforms to the core transformation profile. It is up to the higher order transformation to replace the declarative copy stereotypes by operational counter-parts from the core transformation profile.



<<bound>>
applicationModel : Model
<<closure>>
ownedElement
wodnApplication : UmlPackage
{motmot.constraint=name.equals(this.getApplicationName())}
<<create>>
<<onCopy>>
ownedElement
ownedElement
<<copy>>
cm: Model
{motmot.metatype=Model, scope="Copy each CM to a RM"}
<<onCopy>>+name : String = "RM"
<<closure>>
<<closure>>
<<closure>>
<<closure>>
<<onCopy>>
stereotype
<<create>>
<<bound>>
stereotypeOnCM : Stereotype
stereotype
<<bound>>
stereotypeOnRM : Stereotype
ownedElement
cmGeneralization: Generalization
{motmot.metatype=Generalization}
ownedElement
cmAssociation: UmlAssociation
{motmot.metatype=UmlAssociation}
connection
cmAssEnd: AssociationEnd
{motmot.metatype=AssociationEnd}
ownedElement
enumInCM: Enumeration
{motmot.metatype=Enumeration}
literal
literal : EnumerationLiteral
{motmot.metatype=EnumerationLiteral}
type
<<preserve-between-copies>>
ownedElement
classInCM : UmlClass
{motmot.metatype=UmlClass}
feature
a: Attribute
{motmot.metatype=Attribute}
child
parent
participant
type
typedFeature
<<preserve-between-copies>>
<<preserve-between-copies>>
<<preserve-between-copies>>
<<preserve-between-copies>>
<<onCopy>>
<<create>>
stereotype
<<bound>>
entityStereotype : Stereotype

Fig. 2. Illustrative rewrite rule from an input first-order transformation model.

Example Input Transformation Model Figure 2 shows a rewrite rule from an example input transformation model. This rewrite rule applies the stereotypes presented in the previous section to model that a subgraph representing a *analysis model* needs to be copied into a subgraph representing a *design model*. The rule expresses that all classes, typed attributes, enumerations, inheritance links and association links need to be copied from one to the other subgraph. Classes within the design model are marked as persistent "entities". We refer the reader to [14] for a more detailed description of the context and meaning of all elements. Also remark that this rewrite rule is part of a larger transformation model in which other rules take care of association class flattening [15], etc.

Example Output Transformation Model Within the domain of the core transformation profile, there is no notion of the copy specific constructs. Therefore, the higher order transformation needs to turn the input transformation rules that apply these constructs into complex sequences of more primitive rewrite rules.

To make this more concrete, Figure 3 visualizes the sequence of rewrite rules by showing the control flow of the output transformation model. The 33 generated states are highlighted in clusters of light and dark grey. Each cluster represents rewrite rules for a specific type. States that were already contained in the input transformation model are shown in white. The layout of the diagram is designed as follows: the top-most cluster of dark-gray nodes contains all states required to copy the model element representing the analysis model. This element of type *Model* acts as a container for UML classes, enumerations, associations and generalizations. The four clusters of gray nodes that are displayed on the left of Figure 3 handle the copying of these contained elements:

– the upper cluster of light-gray nodes contains all states required to copy the contained elements of type *Generalization*,
– the following cluster of dark-gray nodes contains all states required to copy the contained elements of type *Association* and *AssociationEnd*,
– the following cluster of light-gray nodes contains all states required to copy the contained elements of type *Enumeration* and the contained *EnumerationLiteral* elements,
– the lower cluster of dark-gray nodes contains all states required to copy the contained elements of type *UmlClass* and the contained *Attribute* elements.

The order in which these clusters are executed could be altered without changing the behavior of the transformation. The diagram layout emphasizes the recursive nature of the transformation behavior. More specifically, each cluster follows a fixed pattern:

select an element from its container in the source subgraph,
check whether the element is already mapped to a copy,
generate a copy if needed,
manipulate the copied element,
recursively apply this pattern on the contained elements.

Figure 4 shows a fragment of Figure 3 in more detail. More specifically, the states related to the copying of enumerations are shown. The top right state (called *select_ enumInCM_ AsPart*) iterates over all *Enumeration* elements contained within the conceptual model. The second state checks whether such an element is already copied. The outgoing ≪failure≫ transition ensures a copy is generated when needed.

When a copy is present, the outgoing ≪success≫ transition ensures that the rewrite rule performing the side effects on the copy is triggered. As indicated by the state name, these manipulations of the copied elements relate to

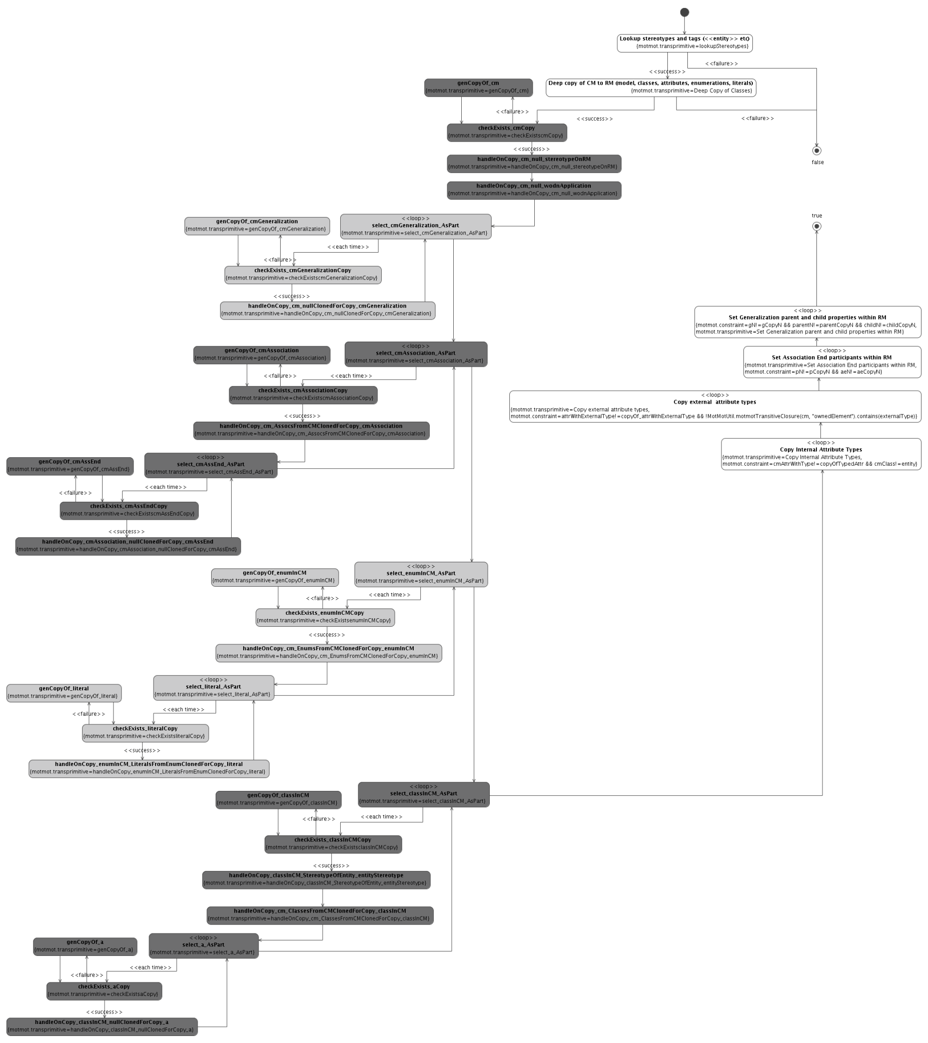

Lookup stereotypes and tags (<<entity>> etc)
{motmot.transprimitive=lookupStereotypes}
<<success>>
<<failure>>
Deep copy of CM to RM (model, classes, attributes, enumerations, literals)
{motmot.transprimitive=Deep Copy of Classes}
<<success>>
<<failure>>
false
genCopyOf_cm
{motmot.transprimitive=genCopyOf_cm}
<<failure>>
checkExists_cmCopy
{motmot.transprimitive=checkExistscmCopy}
<<success>>
handleOnCopy_cm_null_stereotypeOnRM
{motmot.transprimitive=handleOnCopy_cm_null_stereotypeOnRM}
handleOnCopy_cm_null_wodnApplication
{motmot.transprimitive=handleOnCopy_cm_null_wodnApplication}
true
genCopyOf_cmGeneralization
{motmot.transprimitive=genCopyOf_cmGeneralization}
<<loop>>
select_cmGeneralization_AsPart
{motmot.transprimitive=select_cmGeneralization_AsPart}
<<each time>>
<<failure>>
checkExists_cmGeneralizationCopy
{motmot.transprimitive=checkExistscmGeneralizationCopy}
<<success>>
handleOnCopy_cm_nullClonedForCopy_cmGeneralization
{motmot.transprimitive=handleOnCopy_cm_nullClonedForCopy_cmGeneralization}
<<loop>>
Set Generalization parent and child properties within RM
{motmot.constraint=gN!=gCopyN && parentN!=parentCopyN && childN!=childCopyN,
motmot.transprimitive=Set Generalization parent and child properties within RM}
<<loop>>
Set Association End participants within RM
{motmot.transprimitive=Set Association End participants within RM,
motmot.constraint=pN!=pCopyN && aeN!=aeCopyN}
genCopyOf_cmAssociation
{motmot.transprimitive=genCopyOf_cmAssociation}
<<loop>>
select_cmAssociation_AsPart
{motmot.transprimitive=select_cmAssociation_AsPart}
<<failure>>
<<each time>>
checkExists_cmAssociationCopy
{motmot.transprimitive=checkExistscmAssociationCopy}
<<loop>>
Copy external attribute types
{motmot.transprimitive=Copy external attribute types,
motmot.constraint=attrWithExternalType!=copyOf_attrWithExternalType && !MotMotUtil.motmotTransitiveClosure(cm, "ownedElement").contains(externalType)}
<<success>>
handleOnCopy_cm_AssocsFromCMClonedForCopy_cmAssociation
{motmot.transprimitive=handleOnCopy_cm_AssocsFromCMClonedForCopy_cmAssociation}
<<loop>>
Copy Internal Attribute Types
{motmot.transprimitive=Copy Internal Attribute Types,
motmot.constraint=cmAttrWithType!=copyOfTypedAttr && cmClass!=entity}
genCopyOf_cmAssEnd
{motmot.transprimitive=genCopyOf_cmAssEnd}
<<loop>>
select_cmAssEnd_AsPart
{motmot.transprimitive=select_cmAssEnd_AsPart}
<<failure>>
<<each time>>
checkExists_cmAssEndCopy
{motmot.transprimitive=checkExistscmAssEndCopy}
<<success>>
handleOnCopy_cmAssociation_nullClonedForCopy_cmAssEnd
{motmot.transprimitive=handleOnCopy_cmAssociation_nullClonedForCopy_cmAssEnd}
genCopyOf_enumInCM
{motmot.transprimitive=genCopyOf_enumInCM}
<<loop>>
select_enumInCM_AsPart
{motmot.transprimitive=select_enumInCM_AsPart}
<<failure>>
<<each time>>
checkExists_enumInCMCopy
{motmot.transprimitive=checkExistsenumInCMCopy}
<<success>>
handleOnCopy_cm_EnumsFromCMClonedForCopy_enumInCM
{motmot.transprimitive=handleOnCopy_cm_EnumsFromCMClonedForCopy_enumInCM}
genCopyOf_literal
{motmot.transprimitive=genCopyOf_literal}
<<loop>>
select_literal_AsPart
{motmot.transprimitive=select_literal_AsPart}
<<failure>>
<<each time>>
checkExists_literalCopy
{motmot.transprimitive=checkExistsliteralCopy}
<<success>>
handleOnCopy_enumInCM_LiteralsFromEnumClonedForCopy_literal
{motmot.transprimitive=handleOnCopy_enumInCM_LiteralsFromEnumClonedForCopy_literal}
genCopyOf_classInCM
{motmot.transprimitive=genCopyOf_classInCM}
<<loop>>
select_classInCM_AsPart
{motmot.transprimitive=select_classInCM_AsPart}
<<failure>>
<<each time>>
checkExists_classInCMCopy
{motmot.transprimitive=checkExistsclassInCMCopy}
<<success>>
handleOnCopy_classInCM_StereotypeOfEntity_entityStereotype
{motmot.transprimitive=handleOnCopy_classInCM_StereotypeOfEntity_entityStereotype}
handleOnCopy_cm_ClassesFromCMClonedForCopy_classInCM
{motmot.transprimitive=handleOnCopy_cm_ClassesFromCMClonedForCopy_classInCM}
genCopyOf_a
{motmot.transprimitive=genCopyOf_a}
<<loop>>
select_a_AsPart
{motmot.transprimitive=select_a_AsPart}
<<failure>>
<<each time>>
checkExists_aCopy
{motmot.transprimitive=checkExistsaCopy}
<<success>>
handleOnCopy_classInCM_nullClonedForCopy_a
{motmot.transprimitive=handleOnCopy_classInCM_nullClonedForCopy_a}

Fig. 3. Control flow of the output of the higher order transformation.

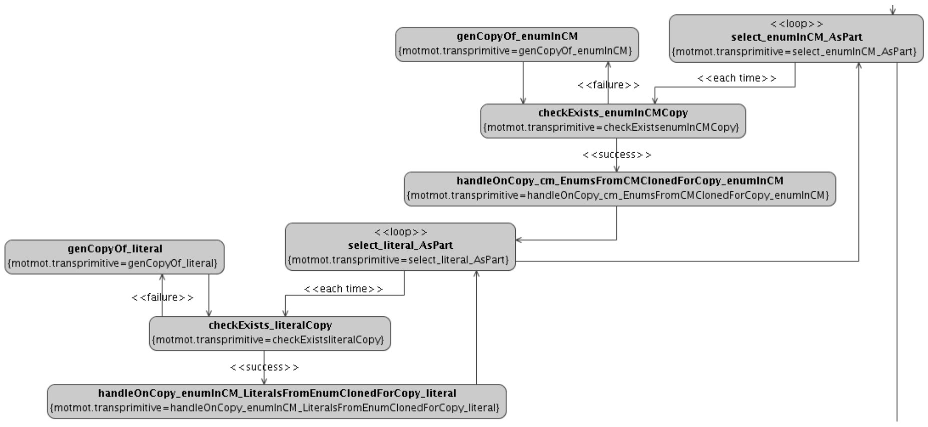

genCopyOf_enumInCM
{motmot.transprimitive=genCopyOf_enumInCM}
<<loop>>
select_enumInCM_AsPart
{motmot.transprimitive=select_enumInCM_AsPart}
<<failure>>
<<each time>>
checkExists_enumInCMCopy
{motmot.transprimitive=checkExistsenumInCMCopy}
<<success>>
handleOnCopy_cm_EnumsFromCMClonedForCopy_enumInCM
{motmot.transprimitive=handleOnCopy_cm_EnumsFromCMClonedForCopy_enumInCM}
<<loop>>
select_literal_AsPart
{motmot.transprimitive=select_literal_AsPart}
genCopyOf_literal
{motmot.transprimitive=genCopyOf_literal}
<<each time>>
<<failure>>
checkExists_literalCopy
{motmot.transprimitive=checkExistsliteralCopy}
<<success>>
handleOnCopy_enumInCM_LiteralsFromEnumClonedForCopy_literal
{motmot.transprimitive=handleOnCopy_enumInCM_LiteralsFromEnumClonedForCopy_literal}

Fig. 4. Control flow of the rules for copying *Enumeration* and *EnumerationLiteral* elements.

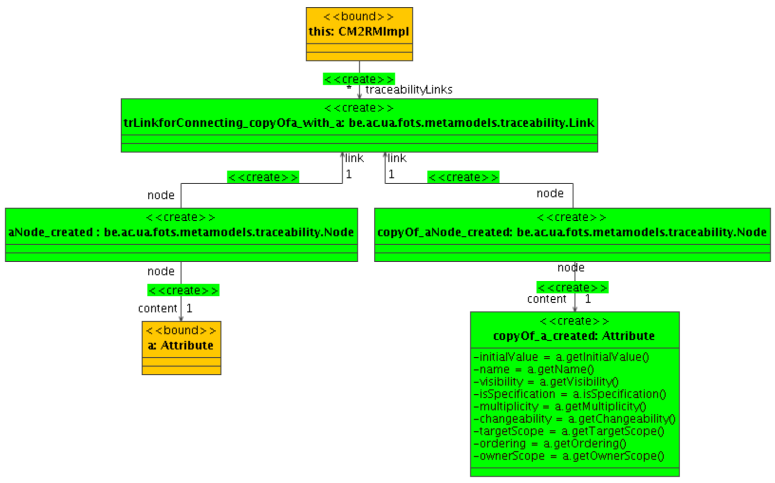
both the *cm* and the *enumInCM* nodes from the Story Pattern shown on Figure 2. More specifically, the state *handleOnCopy_ cm_ EnumsFromCMCloned-ForCopy_ enumInCM* ensures that the copied enumerations are added to the robustness model (which is a copy of the conceptual model). The automatically generated name of this state is based on the name of the edge between the *cm* and *enumInCM* nodes from the Story Pattern shown on Figure 2.

The transition to the *select_ literal_ AsPart* state realizes a recursive step. More specifically, the four states shown on the bottom left of Figure 4 realize the *"select, check, generate, manipulate"* pattern described above on all elements contained within an enumeration.

Figure 5 shows the rewrite rule for generating a copy of *Attribute* elements. The rule corresponds to a state in the lowest cluster from Figure 3. The higher order transformation generates such rewrite rules for all elements that need to be copied. It should be stressed that this generated rewrite rule conforms to the core transformation profile and can therefore be executed by any engine that implements that profile, even by those engines that have been implemented without knowledge of the *Copy* operator discussed in Section 4.1.

Since the nodes in the rewrite rules from the input transformation model (e.g., the *cm*, *classInCM*, *enumInCM*, *a*, ... nodes shown on Figure 2) do not contain these low-level attribute assignments explicitly, the higher order transformation needs to query some metamodel information. Using such metamodel information, the higher order transformation infers, for example, that the generated rewrite rule for *Attribute* nodes initializes the *copyOf_ a_ created* node with the following properties of the *a* node: *initialValue*, *name*, *visibility*, *isSpecification*, *multiplicity*, *changeability*, *targetScope*, *ordering* and *ownerScope*.

Figure 5 also illustrates that the output transformation model explicitly creates traceability links between the source and target subgraphs. Using that mechanism, the realization of the *Copy* operator supports change propagation from



<<bound>>
this: CM2RMImpl
<<create>>
*
traceabilityLinks
<<create>>
trLinkforConnecting_copyOfa_with_a: be.ac.ua.fots.metamodels.traceability.Link
link
1
link
1
<<create>>
<<create>>
node
node
<<create>>
aNode_created : be.ac.ua.fots.metamodels.traceability.Node
<<create>>
copyOf_aNode_created: be.ac.ua.fots.metamodels.traceability.Node
node
<<create>>
content
1
node
<<create>>
content
1
<<bound>>
a: Attribute
<<create>>
copyOf_a_created: Attribute
-initialValue = a.getInitialValue()
-name = a.getName()
-visibility = a.getVisibility()
-isSpecification = a.isSpecification()
-multiplicity = a.getMultiplicity()
-changeability = a.getChangeability()
-targetScope = a.getTargetScope()
-ordering = a.getOrdering()
-ownerScope = a.getOwnerScope()

Fig. 5. Generated Story Pattern for Generating nodes of type *Attribute*.

the source subgraph to its copy. Interestingly, such details are hidden by the transformation profile extension presented in Section 4.1.

4.3 Example of a general Mapping Rule

This section generalizes the example application of the higher order transformation from the previous section into a mapping rule that is independent of the input first order transformation (such as the transformation from analysis to design models). The transformation model that realizes this mapping is publicly available in the MoTMoT project [12].

A rewrite node n_{copy} carrying the ≪copy≫ construct expresses that its matched element needs to be copied. Within a rewrite rule, such a rewrite node should be preserved by the higher order transformation. Obviously, its ≪copy≫ stereotype cannot be preserved in the output transformation model, but all other properties (state or node constraints, attribute assignments, etc.) are preserved. Additionally, a *"check and generate if needed"* pattern should be present in the output:

- a *check*-state, associated with a rewrite rule that models how the presence of an existing copy of the element can be checked,
- a ≪failure≫ transition to a *generate*-state,
- a *generate*-state, whose rewrite rule models the actual creation of the copy,
- a transition back to the *check*-state.

Within the output transformation model, the rewrite rule modeling the *check*-state should contain a traceability link from a ≪bound≫ node that represents the element from n_{copy} to another node of the same type. Similarly, the derived rewrite rule representing the *generate*-state should model the creation of a traceability link from the ≪bound≫ representation of n_{copy} to the node representing the copy. Other mapping rules prescribe how the composition links, the ≪onCopy≫, and the ≪preserve-between-copies≫ constructs should be converted into constructs from the core transformation profile [10, Chapter 8].

5 General Application of the Approach

This section collects a representative set of challenges that need to be overcome for aligning the aforementioned tools (GReAT, MOFLON/TGG, VMTS, ...) with the proposed UML profile. On the one hand, these tool-specific issues should not be disregarded as trivial. On the other hand, this section illustrates that such issues are no fundamental obstacles to the adoption of the proposed approach either. The following sections focus on a well-known data-flow based language and a state-of-the-art mapping language, since such languages are sometimes perceived to be major variations on the graph transformation style that is captured by the profile presented in Section 2.

5.1 Data-Flow Constructs

The proposed transformation profile relies on activity diagrams as a control flow language. At first sight, this may seem incompatible with data-flow based scheduling structures in languages such as GReAT. Therefore, this section illustrates how a GReAT data-flow sequence can be mapped to a controlled graph transformation specification based on the profile's activity diagram language.

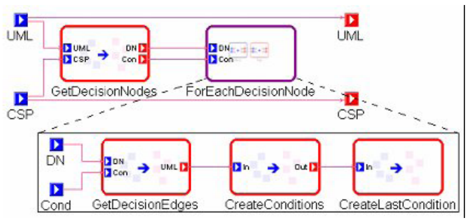


Fig. 6. GReAT data-flow for scheduling rewrite rules.

Consider for example the data-flow specification shown on Figure 6. It controls the GReAT rules for mapping UML diagrams to CSP diagrams [16]. A distinctive feature of this specification style is the use of ports to transfer matched nodes across rewrite rules *explicitly*. In the proposed transformation profile, all nodes that have been matched by previously executed rules are made available automatically to subsequent rewrite rules. Thus, the two data transfer connectors between the *GetDecisionNodes* and *ForEachDecisionNode* blocks would not be modeled explicitly. Instead, one transition link would be modeled between two such activities. Additionally, the *DN* and *Con* nodes would be represented as

≪bound≫ nodes in rewrite rule corresponding to the the *ForEachDecisionNode* activity.

A second characteristic feature shown on Figure 6 is the support for iteration using hierarchical composition. More specifically, the *ForEachDecisionNode* block is an instance of a *ForBlock*. Therefore, each packet consumed by the *ForEachDecisionNode* block will be passed sequentially through the three lower level blocks. When mapping this configuration to the proposed profile, one would map the *ForEachDecisionNode* block to an activity marked with the ≪loop≫ construct. The three hierarchically composed blocks would be contained in an embedded ≪each time≫ flow.

A third issue that needs to be resolved is that GReAT graph elements can be renamed across a data-flow. For example, the output port of the *GetDecisionEdges* block shown on Figure 6 is called *UML*. By connecting it to the *In* port from the *CreateConditions* block, it is available under another name in the rewrite rule that corresponds to the latter block. Although this is not directly supported by the core version of the transformation modeling profile, it can be realized by means of a so-called ≪alias≫ construct [17].

A fourth, final but open issue is GReAT's reliance upon the underlying C++ language for the specification of textual constraints and attribute updates. This issue is not specific to GReAT but is common for today's graph transformation languages (e.g., the Fujaba tool relies on Java for similar purposes). Interestingly, an increasing number of languages is accompanied by a metamodel (e.g., a metamodel for Java has been released in the open source *NetBeans* project). Therefore, mappings from specialized expression languages (such as Imperative OCL, supported by VMTS [7]) into general purpose languages (such as C++ and Java) can be supported by the proposed higher order transformation approach too. Before transformation tool builders generalize their expression language implementations, a limited amount of manual completion of transformation models will remain needed in practice.

In summary, although the GReAT language is based on data-flow modeling, it is close enough to the proposed profile to support the proposed integration architecture. The approach can be introduced incrementally: some language legacy constructs may initially require manual translation.

5.2 Bi-Directional Rules

Triple Graph Grammars (TGGs) were introduced in the early nineties as a formalism for maintaining *bidirectional* consistency constraints between models originating from different software engineering tools [18]. The more recently proposed QVT *relations* strongly resemble triple graph grammar rules [9].

A triple rule not only consists of a left- and a right-hand side. Additionally, it divides the rewrite nodes and links in three domains: two domains represent the models that need to be kept consistent. A third domain represents the traceability model. Essentially, a triple rule describes the relations that need to hold between elements from the right-hand side when the elements from the left-hand side are already consistent.

Although TGG rules can be executed directly by a Java interpreter, their operational semantics is usually clarified by presenting the mapping of a TGG rule to conventional rewrite rules [19]. Burmester et al., for instance, map TGG rules to six primitive graph rewriting rules [20]: three rules for adapting changes to the source model and three for adapting changes to the the target model.

Such a mapping from triple rules into operational rules has already been realized by several transformation tools (e.g., MoRTEn [21] and MOFLON [22]). Unfortunately, the syntax of the triple rules as well as that of the operational rules has always been formalized by a tool-specific metamodel. Moreover, the higher order transformation has always been implemented directly on the API of a Java or C(++) based tool. Therefore, the problems described in Section 1 have been exposed specifically in the domain of TGG tools too. As illustrated in [10], the proposed profile approach is applicable for standardizing the TGG syntax too. Moreover, the algorithm for deriving operational rules from triple rules can be realized using the proposed higher order transformation approach as well.

6 Related Work

The presented use of profiles and higher order transformations has not been proposed before for the integration of transformation languages. Nevertheless, the following references relate to particular aspects of the approach.

Standard Syntax for Rewrite Rules First of all, Graph Transformation eXchange Language (GTXL [8]) has been proposed as a standard for exchanging transformation models. Unlike the proposed profile, GTXL has no relation to a mainstream modeling language such as the UML. Therefore, there are no off-the-shelf industrial tools for editing GTXL models. Secondly, the GTXL metamodel relies on a XML DTD instead of on the MOF. Therefore, it requires more integration effort in an MDA tool integration context. Finally, GTXL only supports uncontrolled rules whereas the proposed profile supports rules that are controlled by activity diagrams. Finally, due to the lack of a profile concept, GTXL cannot be extended without breaking metamodel compatibility with its implementations. Secondly, the Queries/Views/Transformations (QVT [9]) standard presents three languages for transformation modeling. Apart from the MOF basis, it has the same limitations as GTXL. The QVT standard does promote bridges between its sublanguages by means of higher order transformations. In fact, the mapping between the relations and core language is formalized in the QVT relations language [9]. Unfortunately, the QVT relations language is not as generally applicable as the profile presented in this paper. Moreover, the complex semantics of the language requires more implementation effort and less semantical infrastructure can be reused through higher order transformations.

Higher Order Transformations Within the VIATRA tool, the transformation process from human-oriented transformation models into machine-oriented transformation code is supported by higher order transformations too [23].

Unlike the proposed approach, the transformation models do not conform to any standards. Moreover, the higher order transformation is not written in a standard transformation language either.

7 Future Work

Once Fujaba, VMTS, MOLA, GReAT and Progres support the proposed metamodel discussed in Section 2.1, these tools will not only be able to exchange transformation models that apply the language constructs they supported already («*create*», «*destroy*», ...). Instead, they will also be able to load the stereotypes related to the declarative copying that was not anticipated when building these respective tools.

Remark once more that the *Copy* operator is only treated in so much detail to make the language extension approach as concrete as possible. A *Merge* or *Diff* operator (see [13]) could be defined similarly. In fact, we are actively working on profile extensions for negative application conditions and graph grammars with uncontrolled rule applications. This work should make these popular AGG language constructs available to any transformation engine that supports the core profile [24].

In the proposed architecture, higher order transformations are realized using the same language as first-order transformations. This is enabled by using a transformation language that has a MOF metamodel. Although this is a simple technique in theory, the following practical issue still needs to be resolved: when using the MoTMoT prototype for the proposed architecture, we execute higher order transformations manually where needed. Ultimately, both the transformation profile extensions (stereotypes and tagged values) and the corresponding higher order transformations are available in an online repository. Transformation tools should be able to access these artifacts automatically and apply the higher order transformations behind the scenes. Although the MoTMoT prototype provides an online build infrastructure for managing the *compilation*, *testing* and *deployment* and *versioning* of model transformations [25], a systematic process for distributing new versions of a higher order transformation to a public online repository has not been defined yet.

8 Conclusions

This paper presented a new approach to the integration of transformation languages and tools. A realization of the proposed approach enables transformation tool builders to focus on user-oriented added value (such as editor usability, run-time performance, etc.) and new, *declarative* language constructs, instead of spending time on the implementation of evaluation code that was already been realized in other tools before. A unique characteristic of the approach is that it only requires transformation tool builders to implement a small core, and provides support for more declarative language constructs (bidirectional mapping, copying, ...) without breaking interoperability.

The approach relies on two techniques: first of all, its syntactic extensibility is based on the profile support of the metamodel of the "host" modeling language (e.g., the UML). Secondly, new language constructs are made executable by normalizing the profile extensions into the core profile by means of a higher order transformation that is modeled in the core profile itself. As a proof of concept, a core transformation profile was proposed as an OMG UML 1.5 profile.

The MoTMoT tool has already illustrated the executability and usefulness of that profile before. This prototype has now been used to contribute a *Copy* operator to the core profile, without writing any MoTMoT specific code. When somebody realizes *Merge*, *Diff*, or data-flow constructs using the same approach, any tool that supports the core profile will automatically be able to execute these constructs too.

References

1. Manfred Nagl and Andy Schürr. Summary and specification lessons learned. In Manfred Nagl, editor, *IPSEN Book*, volume 1170 of *Lecture Notes in Computer Science*, pages 370–377. Springer, 1996.
2. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A New Graph RewriteLanguage Based on the Unified Modeling Language and Java. In *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)* , volume 1764 of *LNCS*, pages 296–309. Springer Verlag, Nov 1998.
3. Jens H. Jahnke, Wilhelm Schäfer, Jörg P. Wadsack, and Albert Zündorf. Supporting iterations in exploratory database reengineering processes. *Sci. Comput. Program.*, 45(2-3):99–136, 2002.
4. University of Paderborn. Fujaba Tool Suite. http://www.fujaba.de/, 2007.
5. Aditya Agrawal. Graph Rewriting And Transformation (GReAT): A solution for the Model Integrated Computing (MIC) bottleneck. *ASE*, 0:364, 2003.
6. A. Kalnins, E. Celms, and A. Sostaks. Simple and efficient implementation of pattern matching in MOLA tool. *Databases and Information Systems, 2006 7th International Baltic Conference on*, pages 159–167, 3-6 July 2006.
7. T. Levendovszky, L. Lengyel, G Mezei, and H. Charaf. A systematic approach to metamodeling environments and model transformation systems in VMTS. *Electronic Notes in Theoretical Computer Science*, 127:65–75, March 2005.
8. Leen Lambers. A new version of GTXL : An exchange format for graph transformation systems. *Electronic Notes in Theoretical Computer Science*, 127:51–63, March 2005.
9. Object Management Group. MOF 2.0 QVT final adopted specifications. http://www.omg.org/cgi-bin/doc?ptc/05-11-01, November 2005.
10. Pieter Van Gorp. *Model-driven Development of Model Transformations.* PhD thesis, University of Antwerp, April 2008.
11. T. R. G. Green. Cognitive dimensions of notations. In Alistair Sutcliffe and Linda Macaulay, editors, *People and Computers V*, pages 443–460, New York, NY, USA, 1989. Cambridge University Press.
12. Olaf Muliawan, Hans Schippers, and Pieter Van Gorp. Model driven, Template based, Model Transformer (MoTMoT). http://motmot.sourceforge.net/, 2007.

13. Philip A. Bernstein. Generic model management: A database infrastructure for schema manipulation. In *CooplS'01: Proceedings of the 9th International Conference on Cooperative Information Systems*, pages 1–6, London, UK, 2001. Springer-Verlag.
14. Pieter Van Gorp, Hans Schippers, and Dirk Janssens. Copying Subgraphs within Model Repositories. In Roberto Bruni and Dániel Varró, editors, *Fifth International Workshop on Graph Transformation and Visual Modeling Techniques*, Electronic Notes in Theoretical Computer Science, pages 127–139, Vienna, Austria, apr 2006. Elsevier.
15. Martin Gogolla and Mark Richters. Transformation rules for UML class diagrams. In *UML'98: Selected papers from the First International Workshop on The Unified Modeling Language*, pages 92–106, London, UK, 1999. Springer-Verlag.
16. Anantha Narayanan. UML-to-CSP transformation using GReAT. In *AGTiVE 2007 Tool Contest Solutions*, 2007.
17. Pieter Van Gorp, Olaf Muliawan, Anne Keller, and Dirk Janssens. Executing a platform independent model of the UML-to-CSP transformation on a commercial platform. In Gabriele Täntzer and Arend Rensink, editors, *AGTIVE 2007 Tool Contest*, January 2008.
18. Andy Schürr. Specification of graph translators with triple graph grammars. In *Proceedings 20th Workshop on Graph-Theoretic Concepts in Computer Science WG 1994*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1995.
19. A. Königs and A. Schürr. Tool integration with Triple Graph Grammars - a survey. In R. Heckel, editor, *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 113–150, Amsterdam, 2006. Elsevier Science Publ.
20. Sven Burmester, Holger Giese, Jörg Niere, Matthias Tichy, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. Tool integration at the meta-model level: the Fujaba approach. *International Journal on STTT*, 6(3):203–218, August 2004.
21. Robert Wagner. Consistency Management System for the Fujaba Tool Suite – MoTE/MoRTEn Plugins. https://dsd-serv.uni-paderborn.de/projects/cms/, January 2008.
22. C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *Lecture Notes in Computer Science (LNCS)*, pages 361–375, Heidelberg, 2006. Springer Verlag.
23. Ákos Horváth, Dániel Varró, and Gergely Varró. Automatic generation of platform-specific transformation. *Info-Communications-Technology*, LXI(7):40–45, 2006.
24. C. Ermel, M. Rudolf, and G. Taentzer. *The AGG approach: language and environment*, volume II: Applications, Languages, and Tools of *Handbook of graph grammars and computing by graph transformation*, pages 551–603. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.
25. The Apache Software Foundation. Maven. http://maven.apache.org/, 2007.