Integrating a Declarative with an Imperative Model Transformation Language

Pieter Van Gorp, Olaf Muliawan,
Dirk Janssens

September 2006
Technical Report 2006-12

University of Antwerp
Department of Mathematics & Computer Science

Division of Computer Science

Middelheimlaan 1,
2020 Antwerpen, Belgium

Integrating a Declarative with an Imperative Model Transformation Language

Pieter Van Gorp, Olaf Muliawan, Dirk Janssens

Department of Mathematics and Computer Science
University of Antwerp
{pieter.vangorp,olaf.muliawan,dirk.janssens}@ua.ac.be

Abstract. By using a small, yet complex, case study as a model transformation language benchmark, advantages and limitations of several language paradigms can be identified. On the one hand, *declarative* languages only support the specification of constraints that need to be maintained by a transformation. This limitation enables engines to apply default transformation strategies for handling common cases of constraint violation. On the other hand, *imperative* languages support the explicit specification of model updates. This additional expressiveness comes at the cost of more verbose specifications. Therefore, this paper proposes a new, *hybrid*, transformation language that combines the advantages of two legacy languages from these two categories.

1 Introduction

In this paper, we use Model Driven Architecture (MDA) technologies such as Unified Modeling Language (UML) profiles, the Meta-Object Facility (MOF) and the Object Constraint Language (OCL) for improving the language support for the development of model transformations. More specifically, we present the use of a new, *hybrid*, transformation language in the context of a small case study. By limiting the size of the case study, we are able to focus on the issues that challenge today's state-of-the-art transformation tools. One particular challenge that challenges today's *graph rewriting* tools is developer interaction: tools supporting declarative languages such as *Triple Graph Grammars* (TGGs [15]) are usually limited to batch-transformations or only support a fixed interaction pattern [4]. Tools supporting imperative transformation languages such as *Story Diagrams* [2] can be used for implementing any kind of interaction scenario but the transformation models tend to be rather low-level [10].

This paper is organized as follows: Section 2 introduces the exemplary models that are used as a guideline throughout this paper, Section 3 explains why model management is a graph transformation problem and introduces the reader to the various transformation styles from this field. Section 4 presents an imperative and a declarative graph transformation solution to the presented case study. Based on the problems identified in the previous Section, Section 5 covers the main contribution of this paper by introducing a concrete application of a new, *hybrid*, transformation language. After pointing to related work in Section 6 and summarizing the lessons learned in Section 7, the paper concludes.

2 Case Study: Conceptual and Robustness Models

In order to compare various model transformation approaches, an established case study based on the requirements of a Meeting Scheduler system is used throughout our work. The case study was proposed by Van Lamsweerde et al. [1] as a benchmark for requirements elicitation and software specification techniques. The problem statement of the benchmark was published deliberately imprecise and incomplete.

The requirements specification distinguishes between different conflict types and describes ways of resolving them. Subsection 2.1 presents a conceptual model that formalizes the concepts, the associations and their multiplicities from the problem domain. Subsection 2.2 presents a robustness model for the “confirm meeting” use case scenario. The fragments in this paper should merely illustrate some realistic dependencies between models in different languages and should not be regarded as a complete or stable specification of a meeting scheduler.

2.1 Conceptual Model

Figure 1 shows a conceptual model (CM) of a Meeting Scheduler application, specified in UML syntax. 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 in a way as close as possible to one’s perception of reality.

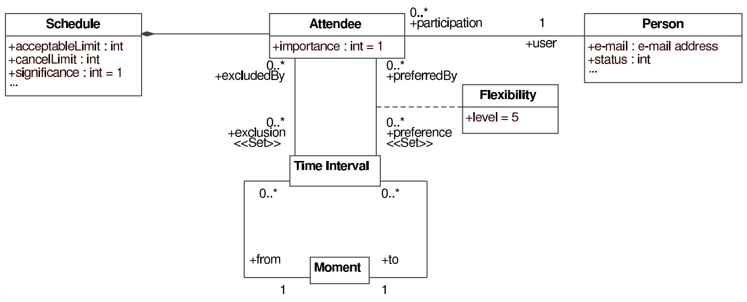


Fig. 1. Conceptual Model of a Meeting Scheduler application.

A complete conceptual model contains all relevant nouns and verbs from a problem domain as classes and operations.

2.2 Robustness Model

The model-view-controller (MVC) pattern has been found beneficial for system evolvability. Therefore, Rosenberg and Scott propose to move from analysis to design by creating *robustness model*s [12]. User interface screens are represented by *boundary objects* (or *interfaces*), persistent classes from the conceptual model are represented by *entities* while application behavior is encapsulated by *control objects* (or *services*). A set of architectural rules (like “only services are allowed to access entities”) assists developers to create an evolvable design that is *robust* by localizing changes. Figure 2 shows a robustness model (RM [12]) of the application under study. Note that the entity Schedule corresponds to the class *Schedule* from Figure 1.

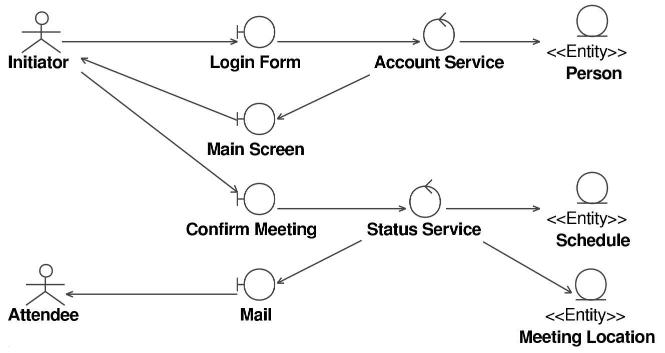


Fig. 2. Robustness Model of a Meeting Scheduler application.

The model describes the way a meeting can be confirmed by an initiator. After logging in, the initiator is directed to the main user interface screen of the application. There, he can select a particular meeting and click the “confirm meeting” button. This action triggers an update of the meeting status and a booking of the meeting location. Finally, the success of the use case is confirmed by sending a mail to all meeting attendees.

2.3 Consistency Constraint

An interesting consistency constraint between conceptual and robustness models is that all classes from the conceptual model should correspond to entities in the robustness model. These classes should “roughly” have the same attributes. Types of corresponding attributes can be contained in the conceptual and robustness model respectively or they can be imported from an external “library” model (or package).

3 Consistency Maintenance based on Graph Transformation

This paper makes extensive use of graph transformation. Therefore, this section introduces the reader to the basics and different flavors of graph transformation that are of interest to a model-driven software engineer.

3.1 Model Transformation as Graph Transformation

The data definition languages (like OMG’s MOF and Eclipse’s ECORE) for modern model repositories (like NetBeans’s MDR and Eclipse’s EMF) 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. The relation between the contents of a repository and a graph are discussed more concretely in the context of Figure 6 from [5] or Figure 3 from [18].

A basic graph transformation system is defined with a set of graph production rules, where a production rule consists of a left-hand side (LHS) graph and a right-hand side (RHS) graph. Such rules are the graph equivalent of term rewriting rules, i.e., intuitively, if the LHS graph is matched in the host graph, it is replaced by the RHS graph.

3.2 Imperative Transformations: Controlled Graph Transformation

In *controlled* graph rewriting, rules can be embedded in a control flow consisting of conditionals and loops. Controlled graph transformation rules correspond to methods that contain a state-based control flow and that are able to call each other while passing nodes as parameters.

Throughout this paper, the UML profile for Story Diagrams [13] is used. The Story Diagram language consists of a UML based syntax with the semantics of controlled graph rewriting. Graph schemata are encoded as UML class diagrams. Primitive graph rewriting rules (called *story patterns*) are encoded in UML's class diagram syntax. A class represents a node to be matched (or *object template* in QVT terminology [9]) while an association represents an edge to be matched. A colon in the class name allows one to separate the name of a node from the type to be matched.

While in PROGRES [14], a language that historically precedes Story Diagrams, the LHS and RHS are shown in distinct parts of a rewrite rule, a story pattern shows all elements in one compound figure. As an advantage, elements occuring in both the LHS and RHS have to be displayed only once. Nodes and edges marked with the $\ll destroy \gg$ stereotype appear only on the left-hand side of the corresponding graph transformation. Such elements are deleted. The stereotype $\ll create \gg$ marks elements only used on the right-handside. Such elements are created.

The control flow that embeds the story patterns is encoded in UML's activity diagram syntax. In controlled graph rewriting, *bound* nodes are nodes that are already known to the system either from previous matchings or because they are passed as parameters to the transformation rule. Thus, a bound node does not trigger the computation of a new match but it reuses its old match.

3.3 Declarative Transformations: Graph Grammars

A *graph grammar* is defined as a set of rules that are executed in parallel until a fixed point is reached. Graph grammars are a declarative formalism since they do not specify a state-based modification of one graph into another one. Therefore, one graph grammar can be used:

- not only for generating one graph from another one,
- but also for checking if an existing graph is consistent with another one.

Pair Graph Grammars were introduced in the early seventies to specify graph-to-graph translations. A pair grammar thus consists of rules which modify two participating graphs. 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 [15].

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. Burmester et al., for instance, map TGG rules to six primitive graph rewriting rules [16]: three rules for adapting changes to the source model and three for adapting changes to the the target model. To indicate the direction of the change propagation, the former three are called the left-to-right or "forward" rules while the latter are called right-to-left or "backward". Both groups of rules consist of a *creation rule*, a *deletion rule* and a *consistency rule*. The former makes sure that when an element is found in one model, a corresponding element exists in the other model. The second rule makes sure that when an element is deleted from one model, its corresponding element is deleted from the other model. The latter rule makes sure that when attribute updates on an element in one model trigger a violation of a consistency constraint related to an element in the other model, that the element in the other model is updated. In fact, an additional rule is needed to create traceability links between model elements that are consistent with one another but were not mapped to one another yet [15].

4 Balancing between Declarative and Imperative

The case study, briefly presented in section 2, challenged our use of OCL and Story Diagrams in that bidirectional consistency constraints could not be modeled concisely. A bidirectional consistency constraint can be maintained by a pair of Story Diagram transformations but this is undesirable due to the verbose specification style. The case study also challenged the triple graph grammar formalism on various aspects. For example, the implementation of a transformation that supported a realistic developer interaction process required us to add control structures between TGG rules (or implement them on a lower level of abstraction [17]).

4.1 Story Diagrams with OCL: too low-level

This subsection illustrates how Story Diagrams and OCL can be used for the implementation of a transformation that maintains the consistency constraint that was introduced informally in Subsection 2.3. Although a set of fine-grained rules with clear pre- and postconditions can properly establish the constraint in a large number of violation scenario's, such a set easily becomes hard to maintain for transformation writers.

Application to Case Study As a first example of a consistency constraint, consider the OCL helper operation *eachClassTracesToAnEntity.* It tests whether each class from the conceptual model traces to an entity in the robustness model:

```
-- Evaluate whether each class in the conceptual model traces to
-- an entity in the robustness model
let eachClassTracesToAnEntity(): Boolean=
 conceptualmodelTracesToRobustnessmodel() and -- 'rm' not Undefined
 allClassesFromModel(cm)->forAll(cmc|
  allClassesFromModel(rm)->exists(rmc|
   this.traceabilityLinks->select(oclIsKindOf(Class2Entity))->exists(l|
    l.node->contains(cmc) and
    l.node->contains(rmc)
   )
  )
 )
```

A transformation system should search for all classes unrelated to an entity and either generate the entity automatically or allow the user to link the class to an existing entity manually. Figure 3 shows how a Story Diagram can model the behavior for the second case.

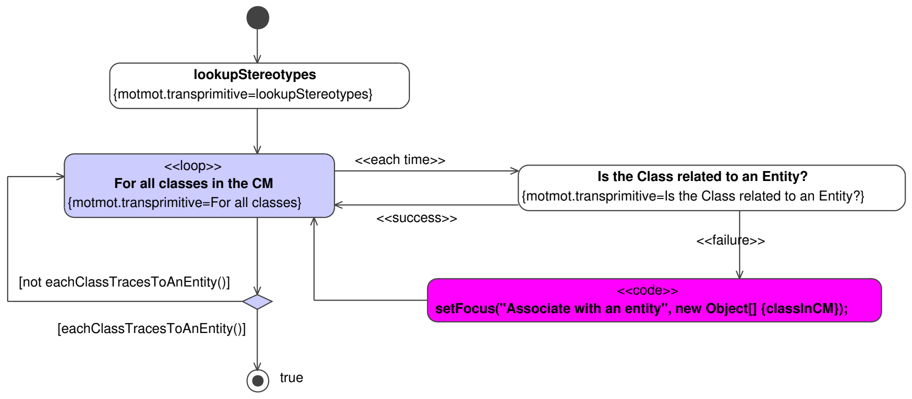

lookupStereotypes
{motmot.transprimitive=lookupStereotypes}
<<loop>>
For all classes in the CM
{motmot.transprimitive=For all classes}
<<each time>>
Is the Class related to an Entity?
{motmot.transprimitive=Is the Class related to an Entity?}
<<success>>
<<failure>>
[not eachClassTracesToAnEntity()]
<<code>>
setFocus("Associate with an entity", new Object[] {classInCM});
[eachClassTracesToAnEntity()]
true

Fig. 3. Story Diagram for a transformation that establishes the *eachClassTracesToAnEntity* constraint on existing models.

The transformation iteratively looks for violating classes in the conceptual model by iterating (due to the$\ll loop \gg$ stereotype) over all classes in the conceptual model. For every (due to the $\ll each\,time \gg$ stereotype) match, the transformation checks whether the matched class is related to an entity. If the latter "Is the Class related to an Entity?" pattern doesn't match, the transformation has found a violating class: the transition with the $\ll failure \gg$ stereotype is triggered and the code state containing a *setFocus* call highlights this problem such that the developer can solve it manually. The *setFocus* method is part of the *ICONS* framework and allows the transformation writers to interact with the modeler. More specifically, the method presents a dialog with the first parameter as a message to the modeler. The second parameter is an array of model elements that should provide the modeler some context for assessing and solving the problem. If the "Is the Class related to an Entity?" pattern does match, the transition with the $\ll success \gg$ stereotype is triggered and the transformation continues with the next class in the conceptual model. After visiting all such classes, the transformation returns *true* if it has established its postcondition, which corresponds to the *eachClassTracesToAnEntity* constraint, or resumes the iteration over violating classes.

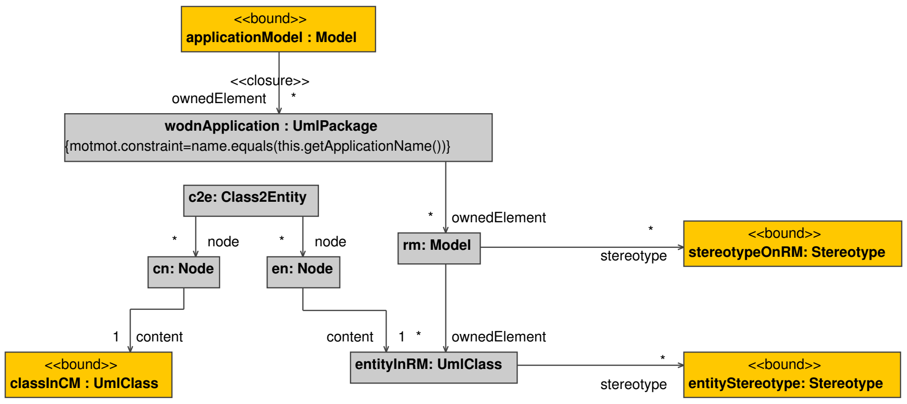


Fig. 4. Story pattern modeling the transformation behavior in state *"Is the Class related to an Entity?"*.

Figure 4 shows how the behavior of the second state can be modeled as a primitive graph rewrite rule (or *story pattern*). The pattern can be read as starting from the bound *applicationModel* node. It searches for the package *wodnApplication* containing the robustness model *rm*. The *rm* node should be connected to a stereotype that has been bound in the first state of the transformation. The *c2e* node represents a traceability link between an entity in *rm* and the class that was bound in the $\ll$ *loop* $\gg$ state called "For all classes in the CM". Figure 5 from [18] models the transformation that generates an entity in an empty robustness model from a class in an existing conceptual model.

As a second sample consistency constraint, consider *classEntity_name_match*. This OCL helper operation checks for all elements satisfying *eachClassTracesToAnEntity* whether the names of a class and its related entity correspond.

```
102 -- Evaluate whether the classifiers (including entities) of all nodes related by
103 -- a Class2Entity link have the same name
104 let classEntity_name_match(): Boolean=
105  traceabilityLinks->select(oclIsKindOf(Class2Entity))->forAll(l|
106   -- for all relevant links:
107   l.ignoreConstraints or -- user has marked that the name can be ignored, or
108   l.node->forAll(n1,n2| -- for any couple of nodes,
109    -- their content elements should have the same name
110    n1.content.name=n2.content.name
111   )
112  )
```

Note that this constraint implements the fundamental concept of "tolerated inconsistencies" as described by Balzer [3]. More specifically, on line 107 it specifies that no further checking is required if the user has set the *ignoreConstraints* property of the *Class2Entity* link to true. Figure 5 shows how the *ignoreConstraints* property is defined on the Link metaclass. By exposing this property to developers, a consistency maintenance system allows them to postpone the resolution of particular inconsistencies. To this extent, we are developing a MagicDraw plugin that can highlight model elements on demand and that provides query and update facilities based on the *ignoreConstraints* property [10].

The *Link* and *Node* metaclasses are part of a metamodel for traceability. *Link*'s four subclasses are part of a transformation model that extends the traceability metamodel. As illustrated by the above OCL constraints, these subclasses provide a context for constraints that are specific to relationships between metaclasses of the languages of the transformed models.

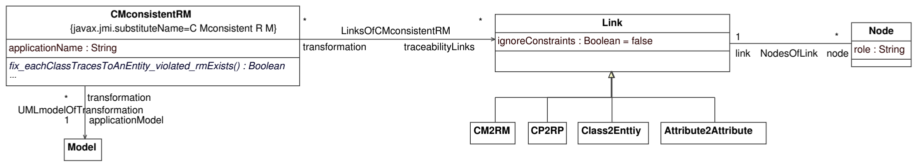


Fig. 5. MOF instance defining the structure of the *CMconsistentRM* transformation and its traceability links.

Problem Identification From *fix_ClassEntity_name_match_violated* one expects that it searches for all classes linked to an entity with a different name. For these conflicting pairs, a transformation can:

- update the name of the class automatically, or
- update the name of the entity automatically, or
- ask the user the give his explicit permission to ignore the inconsistency, or
- ask the user to change the names of class and/or entity manually.

Moreover, next to a change of name, the related class can be deleted, etc. Because these violations can occur for all metaclasses that are constrained in two directions, they should not be modeled explicitly by the transformation writer but should be part of the transformation modeling language. In the next subsection, consider a formalism that provides default reactive behavior to a lot of bidirectional contraint violation scenario's.

4.2 Triple Graph Grammars: too generic

Triple Graph Grammars are a natural alternative for Story Diagrams when bidirectional constraints need to be maintained. This subsection illustrates how the constraint defined on conceptual and robustness models can be maintained by a set of TGG rules. We will then identify where the TGG formalism needs to be extended or complemented to complete the case study in a satisfactory manner.

Application to Example The rule on Figure 6 specifies that at all times, classes contained in a package from the robustness model should be mapped to classes with the same name in a corresponding package in the robustness model. Additionally, the classes in the robustness model should be decorated with the $\ll entity \gg$ stereotype.

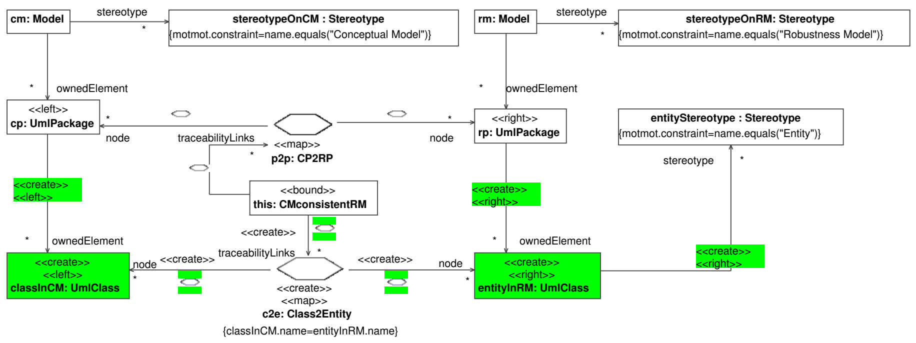


Fig. 6. TGG rule for classes and entities.

Note that in contrast to the rewrite rules from the previous subsection, the rule from Figure 6 is not embedded in a control flow. The only bound node is a *this* reference to the *CMconsistentRM* class. From this context, all other nodes are matched.

The nodes and edges that are decorated with the $\ll create \gg$ stereotype are part of the RHS of the TGG rule. TGG nodes can be devided in four groups, based on whether they carry one of the $\ll left \gg$, $\ll map \gg$ or $\ll right \gg$ stereotypes. Elements that do not carry any of the three stereotypes are part of the overall host graph. Elements carrying the $\ll left \gg$or $\ll right \gg$ stereotypes are part of subgraphs representing the two models that need to be kept consistent. Elements carrying the $\ll map \gg$ stereotype are part of the interconnection (sub)graph (or "traceability model"). They are displayed with a hexagon symbol.

As an illustration that the semantics of a TGG rule is more declarative than a conventional rewrite rule, consider the semantics of Figure 6. With conventional rewrite semantics (or without taking the $\ll left \gg$, $\ll map \gg$ and $\ll right \gg$ stereotypes into account), three new nodes would be created after finding the match described above. No more checking would be performed afterwards. With TGG semantics however, the rule will create a consistent *entityInRM* node when only the *classInCM* node is available. Vice versa: a new *classInCM* node node can be created from an existing *entityInRM* node. When both the *classInCM* and *entityInRM* nodes exist, but they are in conflict, the TGG rule will use the path over the *c2e* node to navigate between conflicting nodes in order to make them consistent again. This can involve changing the name of the *classInCM* or *entityInRM* node or changing the set of stereotypes attached to these nodes. Finally, when a pair of consistent *classInCM* and *entityInRM* nodes exist without a path over *c2e* connecting them, the TGG rule will create such a path.

The second and final conventional TGG rule presented here ensures that the types of corresponding attributes are consistent. The rule illustrates the issue of tracking the creation of edges. Since the underlying graph formalism does not support edges pointing to edges, the *at2at* node points to the *(ca,cat)* and *(ra,rat)* nodes respectively instead of pointing to the RHS edge between these nodes. Note that the management of the Link classes by the *CMconsistentRM* class is implemented in another view that is left out due to space considerations.

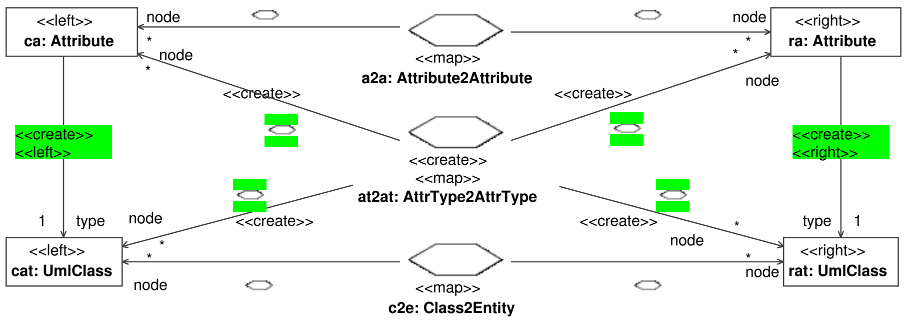


Fig. 7. TGG rule for attribute types in the context of mapped classes and entities.

Once more, mind that the presented TGG rules are not embedded in a control flow and are thus assumed to operate in parallel. Ambiguities can be circumvented by ensuring the LHS of each rule is logically exclusive with that of the other TGG rules, or resolved by offering users the opportunity to order the set of matched rules based on some predefined characteristics. The latter strategy is supported by Fujaba's *MoTE/MoRTEn* plugins and is implemented by flagging elements as soon as they have participated in the execution of one TGG rule such that other rules can be disabled for those elements [11].

Problem Identification The TGG rules presented in the previous section only define the general consistency constraints that should be maintained across conceptual and robustness models. They do not take into account that, for example, the types of attributes can be external datatypes or classes imported from a third party "library" model. Moreover, the semantics of all rules is automatically the same. This implies that, for example, all inconsistencies are resolved fully automatically in both directions. However, it may be desirable that some rules interact with developers before modifying any model element. Moreover, some inconsistencies should be resolved manually, or they should even be tolerated. Nuseibeh even argues that each inconsistency must be treated differently [8]. Without going that far, we acknowledge the need for control flow, user interaction and inconsistency tolerance.

5 A Hybrid Model Transformation Language

This section learns from the problems identified in the declarative and imperative approaches to derive a hybrid solution that allows one to apply the best features of both paradigms together.

5.1 Control Flow

A simple extension of the discussed TGG system is the addition of a rule that defines how external attribute types should be kept consistent. Figure 8 illustrates that such a rule does not introduce any new concepts as such.

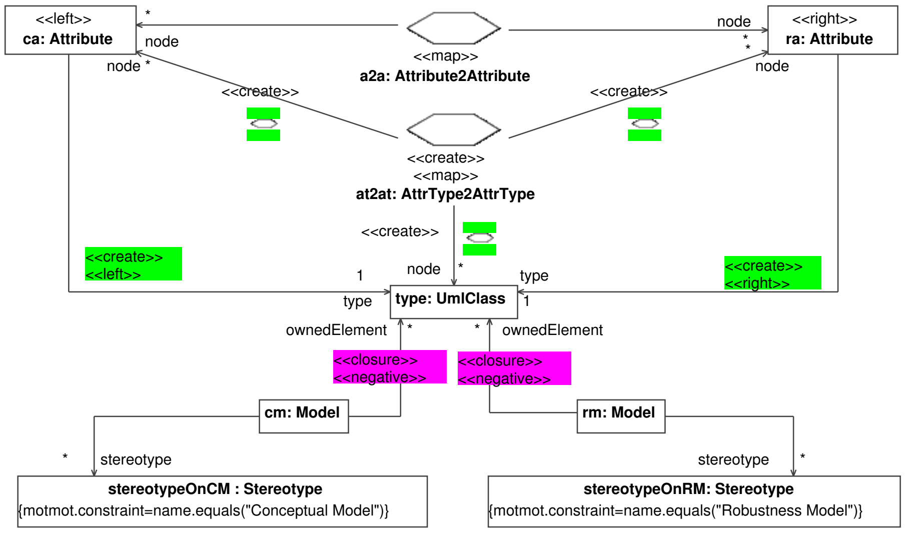


Fig. 8. TGG rule for handling external attribute types.

The major challenge however is the integration of the TGG rules shown on Figure 8 and 7.

Controlled Triple Graph Grammars In many cases, part of the control flow can be pulled out of rules by organizing them in a fine-grained manner [10], with logically exclusive preconditions. However, since the two rules from Figures 8 and 7 can establish consistency under overlapping preconditions, requesting information from the modeler is essential. A branch and TGG rule call is needed to delegate to the appropriate rule. After identifying the possible cases in which inconsistencies between attribute types can occur, the following paragraph will focus on the overlapping between the application conditions of the two TGG rules.

Table 1 presents the relevant values of the types of the attributes *ca* and *ra* from the perspective of keeping them consistent. Note that the ∈ relation can be defined as a transitive traversal of the *owner* (inverse of or *ownedElements*) association end from the UML metamodel.

1	ca=null & ra=null	*4*	ca∈cm & ra=null	*7*	ca∉cm & ra=null
2	ca=null & ra∈rm	*5*	ca∈cm & ra∈rm	*8*	ca∉cm & ra∈rm
3	ca=null & ra∉rm	*6*	ca∈cm & ra∉rm	*9*	ca∉cm & ra∉rm

Table 1. Possible inconsistencies for attribute types.

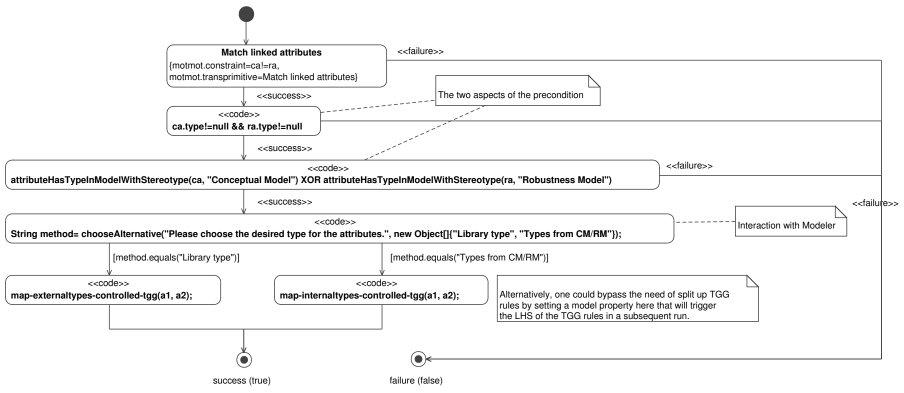
Both the rule on Figure 8 and that on Figure 7 can create a consistent type for *ca* (or *ra*) if this type is null while the type of the corresponding *ra* (or *ca*) exists already. The TGG rule from Figure 7 thus covers cases 2 and 4 while the rule from Figure 8 covers cases 3 and 7. The rule from Figure 7 also covers case 5 since it can reconcile existing types of *ca* and *ra* if they are contained in *cm* and *rm* respectively. More interestingly, case 5 does not trigger a match for the rule from Figure 8: it matches only when at least one of *ca*'s type or *ra*'s type exists in an external library. Similarly, the rule from Figure 8 covers case 9 while this case does not trigger a match of the rule from Figure 7: the latter is matched only when at least one of *ca*'s type or *ra*'s type resides in *cm* or *rm* respectively. If both the type of *ca* and *ra* are null (case 1), these types are already consistent and neither of the TGG rules need to be triggered. The challenging cases from a model reconciliation viewpoint are the ones where *ca* has a type in the conceptual model *cm* and *ra* node has an external libray type (case 6), or vice versa (case 8): in these cases, both the rule from Figure 7 and that from Figure 8 would match. Adding an additional application condition in both rules is not a feasible solution since input from the modeler is required to resolve this ambiguity.

Instead, an explicit control flow needs to be specified between the two TGG rules. More specifcally, before executing the TGG rules, the consistency system needs to look up what attribute type pairs consist of one internal and one external type. For such pairs, the system needs to ask the modeler what type gets precendence over the other one.

Figure 9 displays such a control flow specification. In the first state, all pairs of linked attributes from the conceptual and robustness model are matched. The story diagram modeling the behavior of this state is omitted due to space restrictions but it can be syntactically compared to that of Figure 4.

The second and third topmost states from Figure 9 test the precondition of this interactive transformation: does *ca* have an internal type while *ra* has an external one (case 6 from Table 1) or vice versa (case 8)? The second state tests that type of neither *ca* nor *ra* is null since those cases are handled by the conventional, uncontrolled, behavior of the two TGG rules. The third topmost state tests whether the type of *ca* is contained in the conceptual model too while the type of *ra* is not contained in the robustness model (case 6) or, vice versa, that the type of *ra* is contained in the robustness model too while the type of *ca* is not contained in the conceptual model (case 8).

The fourth topmost state contains a call to request information from the modeler. Without this human input, the transformation cannot decide whether to change the external type to an internal one or vice versa. The two subsequent states contain a call to *variants* of the presented TGG rules. These rules modify the type of *ca* or *ra*, which resolves the inconsistency.


Match linked attributes
<<failure>>
<<success>>
The two aspects of the precondition
<<code>>
<<success>>
<<code>>
<<failure>>
<<success>>
<<code>>
<<failure>>
Interaction with Modeler
[method.equals("Library type")]
[method.equals("Types from CM/RM")]
<<code>>
<<code>>
Alternatively, one could bypass the need of split up TGG rules by setting a model property here that will trigger the LHS of the TGG rules in a subsequent run.
success (true)
failure (false)

```
{motmot.constraint=ca!=ra,
motmot.transprimitive=Match linked attributes}
ca.type!=null && ra.type!=null
attributeHasTypeInModelWithStereotype(ca, "Conceptual Model") XOR attributeHasTypeInModelWithStereotype(ra, "Robustness Model")
String method= chooseAlternative("Please choose the desired type for the attributes.", new Object[]{"Library type", "Types from CM/RM"});
map-externaltypes-controlled-tgg(a1, a2);
map-internaltypes-controlled-tgg(a1, a2);
```

Fig. 9. Control flow of the interactive, hybrid, model transformation for reconciling attribute types in the case of one internal type and one external one.

Calling conventional TGG rules directly is not feasible since they do not operate within the context of particular model element tuples. Therefore, fully declarative TGG rules need to be refactored into more operational ones. For example, by transforming the TGG rule for external attribute types (presented on Figure 8) into a two-state story diagram σ, the TGG mapping can be executed on the *ca* and *ra* attributes that were already bound in the frist state of the transformation shown on Figure 9. The behavior of the second state of σ is modeled by the TGG rule on Figure 10. The main difference with Figure 8 is that the *ca* and *ra* nodes are already bound by being passed as function parameters instead of being matched from the *at2at* node.

Control flow alternative Instead of mixing Story Diagrams with TGG rules on one level of abstraction, one could keep the high level TGG rules and the low-level interaction and control flow details stricly separate. This however leads to subtle dependencies between TGG rules and derived story diagrams. Moreover, since details are added to the Story Diagrams that are generated from the TGG rules, unexpected behavior can be introduced at the Story Diagram level. Therefore, convenient navigation should be provided from TGG rules to the derived Story Diagrams and back. In [17] we proposed the use of traceability links to manage this complexity.

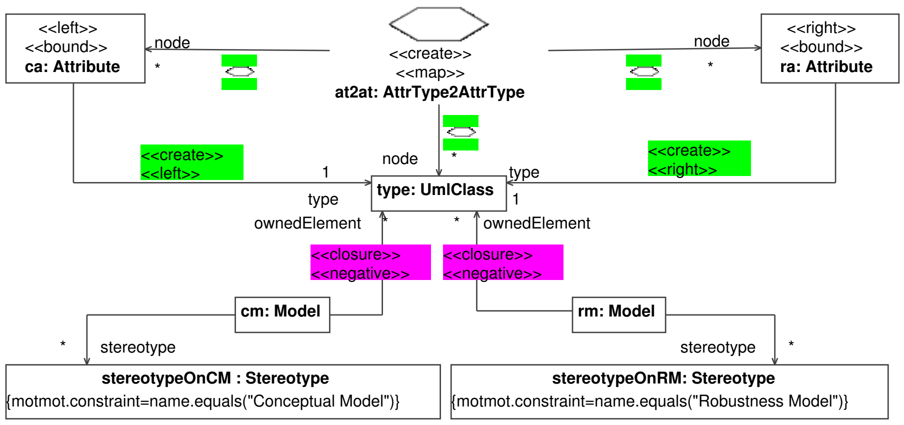


Fig. 10. Example of a controlled TGG rule: *ca* and *ra* are bound by being passed as function parameters.

6 Related Work

The emerging QVT standard [9] refers to the combination of its declarative and imperative sublanguages as a hybrid transformation language. However, the combination of these sublanguages has not been demonstrated on an actual transformation problem yet. The most concrete proposal of integrating declarative language features with imperative ones to date may have been published by Jouault and Kurtev [6]. Their *ATLAS Transformation Language* (ATL) allows one to map elements in *called rules* with the same syntax as that for *matched rules*. Matched rules can be compared with graph grammar rules. Called rules make no use at all of a matching engine. Therefore, ATL could not be employed as the hybrid transformation language introduced in this paper: in Subsection 5.1 we analyzed under what condition two declarative rules would result in an ambiguity and resolved it by defining an imperative rule with higher precedence. This imperative rule used input from the modeler (which would also be supported by ATL's *native* called rules) to delegate to the proper declarative rule (which would also be supported by ATL). ATL's limitation is that the imperative rule cannot be scheduled between (or in this case: *before*) declarative rules. It should also be noted that the main declarative power of TGG rules is that they support bidirectional consistency maintenance with very low specification effort. Future versions of declarative ATL may benefit from an evolution compared with that from pair graph grammars to triple graph grammars, as discussed in Subsection 3.3.

7 Summary and Outlook

This paper aimed to model a complex transformation problem in an intuitive manner. The complexity of the case study consisted of the variety of ways the overall consistency of two models could be violated and the need for developer interaction in the conflict resolution process. After illustrating that neither a purely imperative, nor a purely declarative approach was sufficient to solve this case study in a desirable manner, the integration of two formalisms was used as the first concrete application of a hybrid transformation language.

The advantage of the declarative features of Triple Graph Grammars are that incremental and bidirectional transformations can be specified concisely. On the other hand, adding imperative features such as method definition and calling, sequential composition, branching and looping was required for incorporating some human intelligence into the system.

In our current work, we are comparing the "merged language" approach presented in this paper with the use of the "two-level language architecture" presented in [17]. These alternative approaches for reconciling cociseness with completeness are discussed with the main international TGG tool developers and may be supported by a future version of Fujaba [16] or MoTMoT [7].

Language Integration By Metamodel Merging The advantage of the approach presented in this paper is that the semantics of a TGG rule is defined unambiguously. A potential disadvantage is that some TGG rules become slightly more difficult to reason about. Moreover, a merged language may suffer from compromises that need to be made for satisfying the expectations of developers familiar with the original languages. In the case of the proposed Controlled TGG language, we preserved the mapping of a TGG rule to its six operational rules. However, some of these cases could never be triggered in the given example. This is a possible source of confusion and runtime performance problems (due to redundant matching).

Language Integration By Metamodel Mapping The advantage of the approach from [17] is that TGG rules remain simple by handling ambiguities only at the level of the derived operational rules. The disadvantage from [17] is that without a proper traceability tool, it is hard to understand the complete semantics of a set of TGG rules. It should be noted that the emerging QVT standard [9] proposes a two-level transformation language architecture as well which makes [17] applicable in that context too.

Conclusion

This paper demonstrated the feasability of merging a declarative model transformation language with an imperative one. By combining the advantages of Triple Graph Grammars with those of Story Diagrams, compactness was reconciled with expressiveness. This experiment raised an interesting research question for the language engineering community: what are, in the context of language integration, the advantages of metamodel merging over metamodel mapping and vice versa? Although a preliminary comparison could be given, more research in this direction should be conducted for enabling a more mature tradeoff analysis.

References

1. A. v. Lamsweerde and R. Darimont and P. Massonet. The Meeting Scheduler System - Problem Statement. Technical report, Université Catholique de Louvain - Département d'Ingénierie Informatique, B-1348 Louvain-la-Neuve (Belgium), 1992.
2. Albert Zündorf. *Rigorous Object Oriented Software Development*. PhD thesis, University of Paderborn, 2001.
3. Robert Balzer. Tolerating inconsistency. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 158–165, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.
4. S. Becker, T. Haase, and B. Westfechtel. Model-based a-posteriori integration of engineering tools for incremental development processes. *Journal of Software and Systems Modeling*, 4:123–140, 2004.
5. Stephen Cranefield and Jin Pan. Bridging the gap between the model-driven architecture and ontology engineering. Technical Report 2005/12, Department of Information Science, University of Otago, Dunedin, New Zealand, November 2005.
6. Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, Jan 2006.
7. Olaf Muliawan, Hans Schippers, and Pieter Van Gorp. Model driven, Template based, Model Transformer (MoTMoT). http://motmot.sourceforge.net/, 2006.
8. Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Leveraging inconsistency in software development. *Computer*, 33(4):pp. 24–29, 2000.
9. Object Management Group. *MOF QVT Final Adopted Specification – ptc/05-11-01*, 2005. http://www.omg.org/docs/ptc/05-11-01.pdf.
10. Pieter Van Gorp, Frank Altheide, and Dirk Janssens. Traceability and Fine-Grained Constraints in Interactive Inconsistency Management. In Tor Neple, Jon Oldevik, and Jan Aagedal, editors, *Second ECMDA Traceability Workshop*, 10 July 2006.
11. Robert Wagner. Consistency Management System for the Fujaba Tool Suite – MoTE/-MoRTEn Plugins. https://dsd-serv.uni-paderborn.de/projects/cms/, 1 August 2006.
12. 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.
13. Hans Schippers, Pieter Van Gorp, and Dirk Janssens. Leveraging UML profiles to generate plugins from visual model transformations. *Electronic Notes in Theoretical Computer Science*, 127(3):5–16, 2004. Software Evolution through Transformations (SETra). Satellite of the 2nd Intl. Conference on Graph Transformation.
14. Andy Schürr. Progres: A visual language and environment for programming with graph rewrite systems. Technical Report AIB 94-11, RWTH Aachen, Fachgruppe Informatik, 1994.
15. 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.
16. 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.
17. Pieter Van Gorp, Frank Altheide, and Dirk Janssens. Towards 2D Traceability in a Platform for Contract Aware Visual Transformations with Tolerated Inconsistencies. In *Enterprise Distributed Object Computing Conference (EDOC)*, Hong Kong, 16 October 2006. IEEE.
18. 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, 1 April 2006. Elsevier.

Integrating a Declarative with an Imperative Model Transformation Language

Pieter Van Gorp, Olaf Muliawan,
Dirk Janssens

September 2006
Technical Report 2006-12

University of Antwerp
Department of Mathematics & Computer Science

Division of Computer Science

Middelheimlaan 1,
2020 Antwerpen, Belgium

Integrating a Declarative with an Imperative Model Transformation Language

Pieter Van Gorp, Olaf Muliawan, Dirk Janssens

Department of Mathematics and Computer Science
University of Antwerp
{pieter.vangorp,olaf.muliawan,dirk.janssens}@ua.ac.be

Abstract. By using a small, yet complex, case study as a model transformation language benchmark, advantages and limitations of several language paradigms can be identified. On the one hand, *declarative* languages only support the specification of constraints that need to be maintained by a transformation. This limitation enables engines to apply default transformation strategies for handling common cases of constraint violation. On the other hand, *imperative* languages support the explicit specification of model updates. This additional expressiveness comes at the cost of more verbose specifications. Therefore, this paper proposes a new, *hybrid*, transformation language that combines the advantages of two legacy languages from these two categories.

1 Introduction

In this paper, we use Model Driven Architecture (MDA) technologies such as Unified Modeling Language (UML) profiles, the Meta-Object Facility (MOF) and the Object Constraint Language (OCL) for improving the language support for the development of model transformations. More specifically, we present the use of a new, *hybrid*, transformation language in the context of a small case study. By limiting the size of the case study, we are able to focus on the issues that challenge today's state-of-the-art transformation tools. One particular challenge that challenges today's *graph rewriting* tools is developer interaction: tools supporting declarative languages such as *Triple Graph Grammars* (TGGs [15]) are usually limited to batch-transformations or only support a fixed interaction pattern [4]. Tools supporting imperative transformation languages such as *Story Diagrams* [2] can be used for implementing any kind of interaction scenario but the transformation models tend to be rather low-level [10].

This paper is organized as follows: Section 2 introduces the exemplary models that are used as a guideline throughout this paper, Section 3 explains why model management is a graph transformation problem and introduces the reader to the various transformation styles from this field. Section 4 presents an imperative and a declarative graph transformation solution to the presented case study. Based on the problems identified in the previous Section, Section 5 covers the main contribution of this paper by introducing a concrete application of a new, *hybrid*, transformation language. After pointing to related work in Section 6 and summarizing the lessons learned in Section 7, the paper concludes.

2 Case Study: Conceptual and Robustness Models

In order to compare various model transformation approaches, an established case study based on the requirements of a Meeting Scheduler system is used throughout our work. The case study was proposed by Van Lamsweerde et al. [1] as a benchmark for requirements elicitation and software specification techniques. The problem statement of the benchmark was published deliberately imprecise and incomplete.

The requirements specification distinguishes between different conflict types and describes ways of resolving them. Subsection 2.1 presents a conceptual model that formalizes the concepts, the associations and their multiplicities from the problem domain. Subsection 2.2 presents a robustness model for the “confirm meeting” use case scenario. The fragments in this paper should merely illustrate some realistic dependencies between models in different languages and should not be regarded as a complete or stable specification of a meeting scheduler.

2.1 Conceptual Model

Figure 1 shows a conceptual model (CM) of a Meeting Scheduler application, specified in UML syntax. 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 in a way as close as possible to one’s perception of reality.

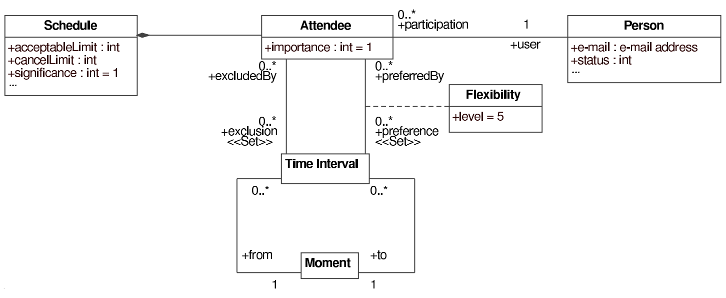


Fig. 1. Conceptual Model of a Meeting Scheduler application.

A complete conceptual model contains all relevant nouns and verbs from a problem domain as classes and operations.

2.2 Robustness Model

The model-view-controller (MVC) pattern has been found beneficial for system evolvability. Therefore, Rosenberg and Scott propose to move from analysis to design by creating *robustness model*s [12]. User interface screens are represented by *boundary objects* (or *interfaces*), persistent classes from the conceptual model are represented by *entities* while application behavior is encapsulated by *control objects* (or *services*). A set of architectural rules (like "only services are allowed to access entities") assists developers to create an evolvable design that is *robust* by localizing changes. Figure 2 shows a robustness model (RM [12]) of the application under study. Note that the entity Schedule corresponds to the class *Schedule* from Figure 1.

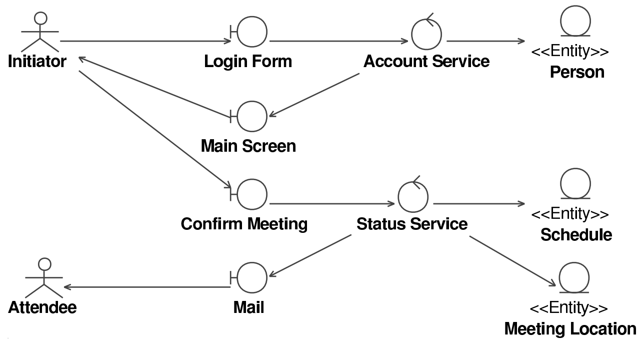


Fig. 2. Robustness Model of a Meeting Scheduler application.

The model describes the way a meeting can be confirmed by an initiator. After logging in, the initiator is directed to the main user interface screen of the application. There, he can select a particular meeting and click the "confirm meeting" button. This action triggers an update of the meeting status and a booking of the meeting location. Finally, the success of the use case is confirmed by sending a mail to all meeting attendees.

2.3 Consistency Constraint

An interesting consistency constraint between conceptual and robustness models is that all classes from the conceptual model should correspond to entities in the robustness model. These classes should "roughly" have the same attributes. Types of corresponding attributes can be contained in the conceptual and robustness model respectively or they can be imported from an external "library" model (or package).

3 Consistency Maintenance based on Graph Transformation

This paper makes extensive use of graph transformation. Therefore, this section introduces the reader to the basics and different flavors of graph transformation that are of interest to a model-driven software engineer.

3.1 Model Transformation as Graph Transformation

The data definition languages (like OMG's MOF and Eclipse's ECORE) for modern model repositories (like NetBeans's MDR and Eclipse's EMF) 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. The relation between the contents of a repository and a graph are discussed more concretely in the context of Figure 6 from [5] or Figure 3 from [18].

A basic graph transformation system is defined with a set of graph production rules, where a production rule consists of a left-hand side (LHS) graph and a right-hand side (RHS) graph. Such rules are the graph equivalent of term rewriting rules, i.e., intuitively, if the LHS graph is matched in the host graph, it is replaced by the RHS graph.

3.2 Imperative Transformations: Controlled Graph Transformation

In *controlled* graph rewriting, rules can be embedded in a control flow consisting of conditionals and loops. Controlled graph transformation rules correspond to methods that contain a state-based control flow and that are able to call each other while passing nodes as parameters.

Throughout this paper, the UML profile for Story Diagrams [13] is used. The Story Diagram language consists of a UML based syntax with the semantics of controlled graph rewriting. Graph schemata are encoded as UML class diagrams. Primitive graph rewriting rules (called *story patterns*) are encoded in UML's class diagram syntax. A class represents a node to be matched (or *object template* in QVT terminology [9]) while an association represents an edge to be matched. A colon in the class name allows one to separate the name of a node from the type to be matched.

While in PROGRES [14], a language that historically precedes Story Diagrams, the LHS and RHS are shown in distinct parts of a rewrite rule, a story pattern shows all elements in one compound figure. As an advantage, elements occuring in both the LHS and RHS have to be displayed only once. Nodes and edges marked with the $\ll destroy \gg$ stereotype appear only on the left-hand side of the corresponding graph transformation. Such elements are deleted. The stereotype $\ll create \gg$ marks elements only used on the right-handside. Such elements are created.

The control flow that embeds the story patterns is encoded in UML's activity diagram syntax. In controlled graph rewriting, *bound* nodes are nodes that are already known to the system either from previous matchings or because they are passed as parameters to the transformation rule. Thus, a bound node does not trigger the computation of a new match but it reuses its old match.

3.3 Declarative Transformations: Graph Grammars

A *graph grammar* is defined as a set of rules that are executed in parallel until a fixed point is reached. Graph grammars are a declarative formalism since they do not specify a state-based modification of one graph into another one. Therefore, one graph grammar can be used:

– not only for generating one graph from another one,
– but also for checking if an existing graph is consistent with another one.

Pair Graph Grammars were introduced in the early seventies to specify graph-to-graph translations. A pair grammar thus consists of rules which modify two participating graphs. 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 [15].

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. Burmester et al., for instance, map TGG rules to six primitive graph rewriting rules [16]: three rules for adapting changes to the source model and three for adapting changes to the the target model. To indicate the direction of the change propagation, the former three are called the left-to-right or "forward" rules while the latter are called right-to-left or "backward". Both groups of rules consist of a *creation rule*, a *deletion rule* and a *consistency rule*. The former makes sure that when an element is found in one model, a corresponding element exists in the other model. The second rule makes sure that when an element is deleted from one model, its corresponding element is deleted from the other model. The latter rule makes sure that when attribute updates on an element in one model trigger a violation of a consistency constraint related to an element in the other model, that the element in the other model is updated. In fact, an additional rule is needed to create traceability links between model elements that are consistent with one another but were not mapped to one another yet [15].

4 Balancing between Declarative and Imperative

The case study, briefly presented in section 2, challenged our use of OCL and Story Diagrams in that bidirectional consistency constraints could not be modeled concisely. A bidirectional consistency constraint can be maintained by a pair of Story Diagram transformations but this is undesirable due to the verbose specification style. The case study also challenged the triple graph grammar formalism on various aspects. For example, the implementation of a transformation that supported a realistic developer interaction process required us to add control structures between TGG rules (or implement them on a lower level of abstraction [17]).

4.1 Story Diagrams with OCL: too low-level

This subsection illustrates how Story Diagrams and OCL can be used for the implementation of a transformation that maintains the consistency constraint that was introduced informally in Subsection 2.3. Although a set of fine-grained rules with clear pre- and postconditions can properly establish the constraint in a large number of violation scenario's, such a set easily becomes hard to maintain for transformation writers.

Application to Case Study As a first example of a consistency constraint, consider the OCL helper operation *eachClassTracesToAnEntity.* It tests whether each class from the conceptual model traces to an entity in the robustness model:

```
-- Evaluate whether each class in the conceptual model traces to
-- an entity in the robustness model
let eachClassTracesToAnEntity(): Boolean=
 conceptualmodelTracesToRobustnessmodel() and -- 'rm' not Undefined
 allClassesFromModel(cm)->forAll(cmc|
  allClassesFromModel(rm)->exists(rmc|
   this.traceabilityLinks->select(oclIsKindOf(Class2Entity))->exists(l|
    l.node->contains(cmc) and
    l.node->contains(rmc)
   )
  )
 )
```

A transformation system should search for all classes unrelated to an entity and either generate the entity automatically or allow the user to link the class to an existing entity manually. Figure 3 shows how a Story Diagram can model the behavior for the second case.

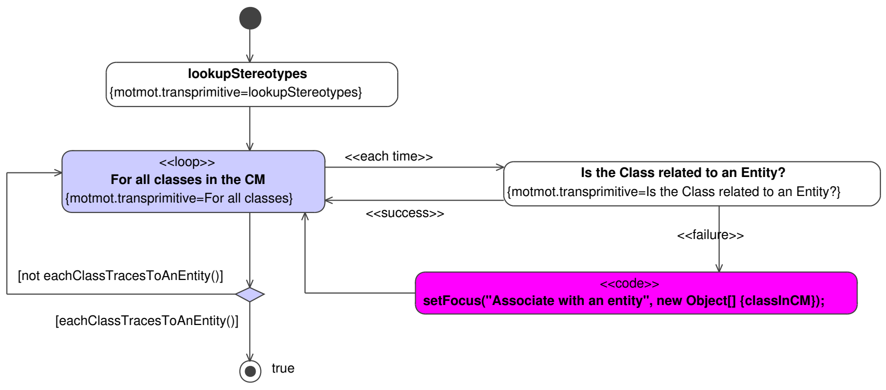

lookupStereotypes
{motmot.transprimitive=lookupStereotypes}
<<loop>>
For all classes in the CM
{motmot.transprimitive=For all classes}
<<each time>>
Is the Class related to an Entity?
{motmot.transprimitive=Is the Class related to an Entity?}
<<success>>
<<failure>>
<<code>>
setFocus("Associate with an entity", new Object[] {classInCM});
[not eachClassTracesToAnEntity()]
[eachClassTracesToAnEntity()]
true

Fig. 3. Story Diagram for a transformation that establishes the *eachClassTracesToAnEntity* constraint on existing models.

The transformation iteratively looks for violating classes in the conceptual model by iterating (due to the$\ll loop \gg$ stereotype) over all classes in the conceptual model. For every (due to the $\ll$ *each time* $\gg$ stereotype) match, the transformation checks whether the matched class is related to an entity. If the latter "Is the Class related to an Entity?" pattern doesn't match, the transformation has found a violating class: the transition with the $\ll failure \gg$ stereotype is triggered and the code state containing a *setFocus* call highlights this problem such that the developer can solve it manually. The *setFocus* method is part of the *ICONS* framework and allows the transformation writers to interact with the modeler. More specifically, the method presents a dialog with the first parameter as a message to the modeler. The second parameter is an array of model elements that should provide the modeler some context for assessing and solving the problem. If the "Is the Class related to an Entity?" pattern does match, the transition with the $\ll success \gg$ stereotype is triggered and the transformation continues with the next class in the conceptual model. After visiting all such classes, the transformation returns *true* if it has established its postcondition, which corresponds to the *eachClassTracesToAnEntity* constraint, or resumes the iteration over violating classes.

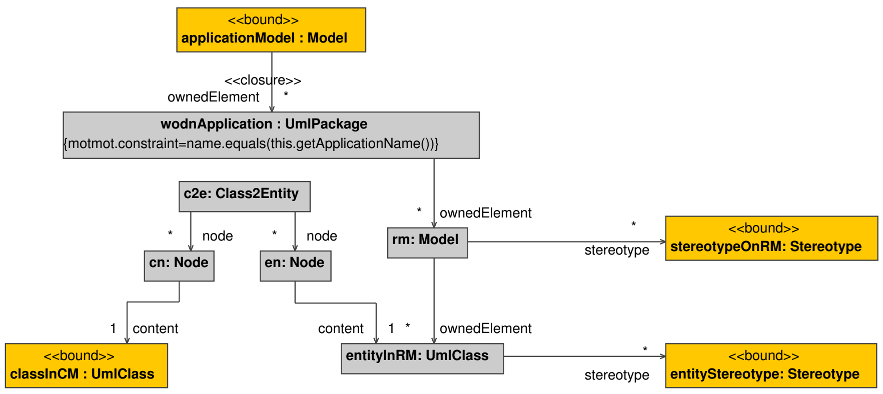


Fig. 4. Story pattern modeling the transformation behavior in state *"Is the Class related to an Entity?"*.

Figure 4 shows how the behavior of the second state can be modeled as a primitive graph rewrite rule (or *story pattern*). The pattern can be read as starting from the bound *applicationModel* node. It searches for the package *wodnApplication* containing the robustness model *rm*. The *rm* node should be connected to a stereotype that has been bound in the first state of the transformation. The *c2e* node represents a traceability link between an entity in *rm* and the class that was bound in the $\ll$ *loop* $\gg$ state called "For all classes in the CM". Figure 5 from [18] models the transformation that generates an entity in an empty robustness model from a class in an existing conceptual model.

As a second sample consistency constraint, consider *classEntity_name_match*. This OCL helper operation checks for all elements satisfying *eachClassTracesToAnEntity* whether the names of a class and its related entity correspond.

```
102  -- Evaluate whether the classifiers (including entities) of all nodes related by
103  -- a Class2Entity link have the same name
104  let classEntity_name_match(): Boolean=
105   traceabilityLinks->select(oclIsKindOf(Class2Entity))->forAll(l|
106    -- for all relevant links:
107    l.ignoreConstraints or -- user has marked that the name can be ignored, or
108    l.node->forAll(n1,n2| -- for any couple of nodes,
109     -- their content elements should have the same name
110     n1.content.name=n2.content.name
111    )
112   )
```

Note that this constraint implements the fundamental concept of "tolerated inconsistencies" as described by Balzer [3]. More specifically, on line 107 it specifies that no further checking is required if the user has set the *ignoreConstraints* property of the *Class2Entity* link to true. Figure 5 shows how the *ignoreConstraints* property is defined on the Link metaclass. By exposing this property to developers, a consistency maintenance system allows them to postpone the resolution of particular inconsistencies. To this extent, we are developing a MagicDraw plugin that can highlight model elements on demand and that provides query and update facilities based on the *ignoreConstraints* property [10].

The *Link* and *Node* metaclasses are part of a metamodel for traceability. *Link*'s four subclasses are part of a transformation model that extends the traceability metamodel. As illustrated by the above OCL constraints, these subclasses provide a context for constraints that are specific to relationships between metaclasses of the languages of the transformed models.

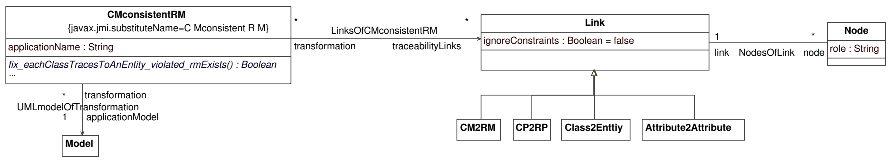


Fig. 5. MOF instance defining the structure of the *CMconsistentRM* transformation and its traceability links.

Problem Identification From *fix_ClassEntity_name_match_violated* one expects that it searches for all classes linked to an entity with a different name. For these conflicting pairs, a transformation can:

- update the name of the class automatically, or
- update the name of the entity automatically, or
- ask the user the give his explicit permission to ignore the inconsistency, or
- ask the user to change the names of class and/or entity manually.

Moreover, next to a change of name, the related class can be deleted, etc. Because these violations can occur for all metaclasses that are constrained in two directions, they should not be modeled explicitly by the transformation writer but should be part of the transformation modeling language. In the next subsection, consider a formalism that provides default reactive behavior to a lot of bidirectional contraint violation scenario's.

4.2 Triple Graph Grammars: too generic

Triple Graph Grammars are a natural alternative for Story Diagrams when bidirectional constraints need to be maintained. This subsection illustrates how the constraint defined on conceptual and robustness models can be maintained by a set of TGG rules. We will then identify where the TGG formalism needs to be extended or complemented to complete the case study in a satisfactory manner.

Application to Example The rule on Figure 6 specifies that at all times, classes contained in a package from the robustness model should be mapped to classes with the same name in a corresponding package in the robustness model. Additionally, the classes in the robustness model should be decorated with the $\ll entity \gg$ stereotype.

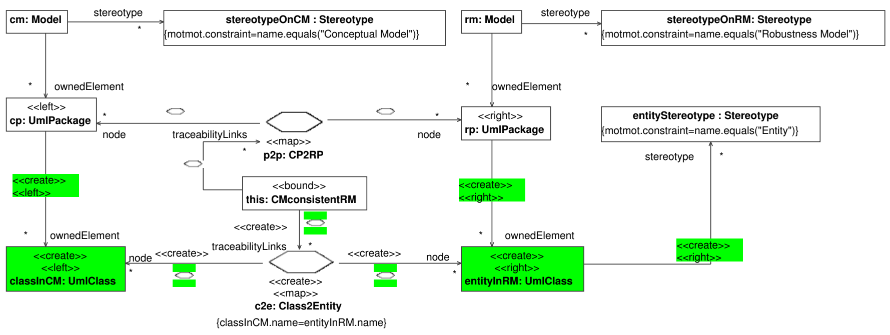


Fig. 6. TGG rule for classes and entities.

Note that in contrast to the rewrite rules from the previous subsection, the rule from Figure 6 is not embedded in a control flow. The only bound node is a *this* reference to the *CMconsistentRM* class. From this context, all other nodes are matched.

The nodes and edges that are decorated with the $\ll create \gg$ stereotype are part of the RHS of the TGG rule. TGG nodes can be devided in four groups, based on whether they carry one of the $\ll left \gg$, $\ll map \gg$ or $\ll right \gg$ stereotypes. Elements that do not carry any of the three stereotypes are part of the overall host graph. Elements carrying the $\ll left \gg$or $\ll right \gg$ stereotypes are part of subgraphs representing the two models that need to be kept consistent. Elements carrying the $\ll map \gg$ stereotype are part of the interconnection (sub)graph (or "traceability model"). They are displayed with a hexagon symbol.

As an illustration that the semantics of a TGG rule is more declarative than a conventional rewrite rule, consider the semantics of Figure 6. With conventional rewrite semantics (or without taking the $\ll left \gg$, $\ll map \gg$ and $\ll right \gg$ stereotypes into account), three new nodes would be created after finding the match described above. No more checking would be performed afterwards. With TGG semantics however, the rule will create a consistent *entityInRM* node when only the *classInCM* node is available. Vice versa: a new *classInCM* node node can be created from an existing *entityInRM* node. When both the *classInCM* and *entityInRM* nodes exist, but they are in conflict, the TGG rule will use the path over the *c2e* node to navigate between conflicting nodes in order to make them consistent again. This can involve changing the name of the *classInCM* or *entityInRM* node or changing the set of stereotypes attached to these nodes. Finally, when a pair of consistent *classInCM* and *entityInRM* nodes exist without a path over *c2e* connecting them, the TGG rule will create such a path.

The second and final conventional TGG rule presented here ensures that the types of corresponding attributes are consistent. The rule illustrates the issue of tracking the creation of edges. Since the underlying graph formalism does not support edges pointing to edges, the *at2at* node points to the *(ca,cat)* and *(ra,rat)* nodes respectively instead of pointing to the RHS edge between these nodes. Note that the management of the Link classes by the *CMconsistentRM* class is implemented in another view that is left out due to space considerations.

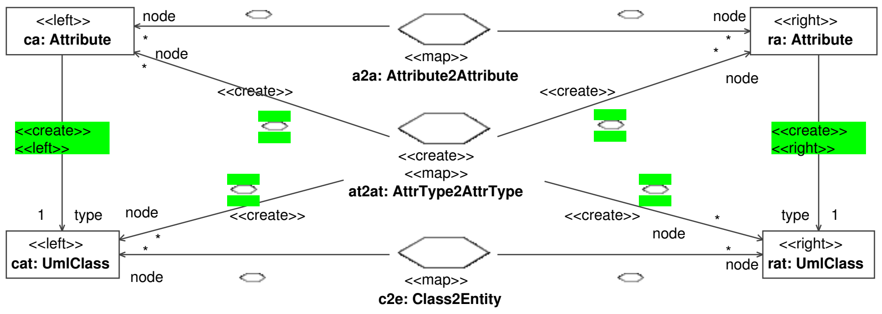


Fig. 7. TGG rule for attribute types in the context of mapped classes and entities.

Once more, mind that the presented TGG rules are not embedded in a control flow and are thus assumed to operate in parallel. Ambiguities can be circumvented by ensuring the LHS of each rule is logically exclusive with that of the other TGG rules, or resolved by offering users the opportunity to order the set of matched rules based on some predefined characteristics. The latter strategy is supported by Fujaba's *MoTE/MoRTEn* plugins and is implemented by flagging elements as soon as they have participated in the execution of one TGG rule such that other rules can be disabled for those elements [11].

Problem Identification The TGG rules presented in the previous section only define the general consistency constraints that should be maintained across conceptual and robustness models. They do not take into account that, for example, the types of attributes can be external datatypes or classes imported from a third party "library" model. Moreover, the semantics of all rules is automatically the same. This implies that, for example, all inconsistencies are resolved fully automatically in both directions. However, it may be desirable that some rules interact with developers before modifying any model element. Moreover, some inconsistencies should be resolved manually, or they should even be tolerated. Nuseibeh even argues that each inconsistency must be treated differently [8]. Without going that far, we acknowledge the need for control flow, user interaction and inconsistency tolerance.

5 A Hybrid Model Transformation Language

This section learns from the problems identified in the declarative and imperative approaches to derive a hybrid solution that allows one to apply the best features of both paradigms together.

5.1 Control Flow

A simple extension of the discussed TGG system is the addition of a rule that defines how external attribute types should be kept consistent. Figure 8 illustrates that such a rule does not introduce any new concepts as such.

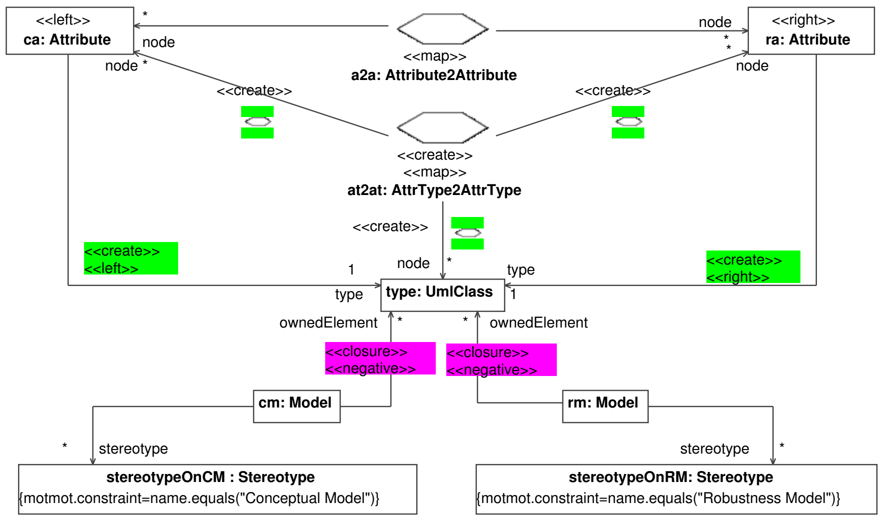


Fig. 8. TGG rule for handling external attribute types.

The major challenge however is the integration of the TGG rules shown on Figure 8 and 7.

Controlled Triple Graph Grammars In many cases, part of the control flow can be pulled out of rules by organizing them in a fine-grained manner [10], with logically exclusive preconditions. However, since the two rules from Figures 8 and 7 can establish consistency under overlapping preconditions, requesting information from the modeler is essential. A branch and TGG rule call is needed to delegate to the appropriate rule. After identifying the possible cases in which inconsistencies between attribute types can occur, the following paragraph will focus on the overlapping between the application conditions of the two TGG rules.

Table 1 presents the relevant values of the types of the attributes *ca* and *ra* from the perspective of keeping them consistent. Note that the ∈ relation can be defined as a transitive traversal of the *owner* (inverse of or *ownedElements*) association end from the UML metamodel.

1	ca=null & ra=null	*4*	ca∈cm & ra=null	*7*	ca∉cm & ra=null
2	ca=null & ra∈rm	*5*	ca∈cm & ra∈rm	*8*	ca∉cm & ra∈rm
3	ca=null & ra∉rm	*6*	ca∈cm & ra∉rm	*9*	ca∉cm & ra∉rm

Table 1. Possible inconsistencies for attribute types.

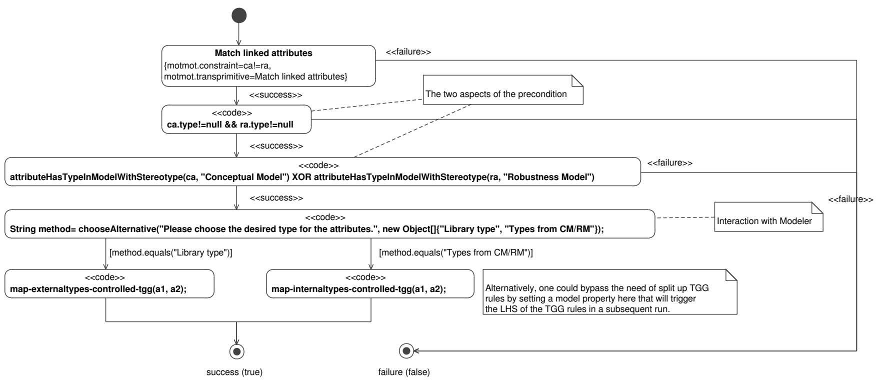
Both the rule on Figure 8 and that on Figure 7 can create a consistent type for *ca* (or *ra*) if this type is null while the type of the corresponding *ra* (or *ca*) exists already. The TGG rule from Figure 7 thus covers cases 2 and 4 while the rule from Figure 8 covers cases 3 and 7. The rule from Figure 7 also covers case 5 since it can reconcile existing types of *ca* and *ra* if they are contained in *cm* and *rm* respectively. More interestingly, case 5 does not trigger a match for the rule from Figure 8: it matches only when at least one of *ca*'s type or *ra*'s type exists in an external library. Similarly, the rule from Figure 8 covers case 9 while this case does not trigger a match of the rule from Figure 7: the latter is matched only when at least one of *ca*'s type or *ra*'s type resides in *cm* or *rm* respectively. If both the type of *ca* and *ra* are null (case 1), these types are already consistent and neither of the TGG rules need to be triggered. The challenging cases from a model reconciliation viewpoint are the ones where *ca* has a type in the conceptual model *cm* and *ra* node has an external libray type (case 6), or vice versa (case 8): in these cases, both the rule from Figure 7 and that from Figure 8 would match. Adding an additional application condition in both rules is not a feasible solution since input from the modeler is required to resolve this ambiguity.

Instead, an explicit control flow needs to be specified between the two TGG rules. More specifcally, before executing the TGG rules, the consistency system needs to look up what attribute type pairs consist of one internal and one external type. For such pairs, the system needs to ask the modeler what type gets precendence over the other one.

Figure 9 displays such a control flow specification. In the first state, all pairs of linked attributes from the conceptual and robustness model are matched. The story diagram modeling the behavior of this state is omitted due to space restrictions but it can be syntactically compared to that of Figure 4.

The second and third topmost states from Figure 9 test the precondition of this interactive transformation: does *ca* have an internal type while *ra* has an external one (case 6 from Table 1) or vice versa (case 8)? The second state tests that type of neither *ca* nor *ra* is null since those cases are handled by the conventional, uncontrolled, behavior of the two TGG rules. The third topmost state tests whether the type of *ca* is contained in the conceptual model too while the type of *ra* is not contained in the robustness model (case 6) or, vice versa, that the type of *ra* is contained in the robustness model too while the type of *ca* is not contained in the conceptual model (case 8).

The fourth topmost state contains a call to request information from the modeler. Without this human input, the transformation cannot decide whether to change the external type to an internal one or vice versa. The two subsequent states contain a call to *variants* of the presented TGG rules. These rules modify the type of *ca* or *ra*, which resolves the inconsistency.



Match linked attributes
{motmot.constraint=ca!=ra,
motmot.transprimitive=Match linked attributes}
<<failure>>
<<success>>
The two aspects of the precondition
<<code>>
ca.type!=null && ra.type!=null
<<success>>
<<code>>
attributeHasTypeInModelWithStereotype(ca, "Conceptual Model") XOR attributeHasTypeInModelWithStereotype(ra, "Robustness Model")
<<failure>>
<<success>>
<<failure>>
<<code>>
String method= chooseAlternative("Please choose the desired type for the attributes.", new Object[]{"Library type", "Types from CM/RM"});
Interaction with Modeler
[method.equals("Library type")]
[method.equals("Types from CM/RM")]
<<code>>
map-externaltypes-controlled-tgg(a1, a2);
<<code>>
map-internaltypes-controlled-tgg(a1, a2);
Alternatively, one could bypass the need of split up TGG rules by setting a model property here that will trigger the LHS of the TGG rules in a subsequent run.
success (true)
failure (false)

Fig. 9. Control flow of the interactive, hybrid, model transformation for reconciling attribute types in the case of one internal type and one external one.

Calling conventional TGG rules directly is not feasible since they do not operate within the context of particular model element tuples. Therefore, fully declarative TGG rules need to be refactored into more operational ones. For example, by transforming the TGG rule for external attribute types (presented on Figure 8) into a two-state story diagram σ, the TGG mapping can be executed on the *ca* and *ra* attributes that were already bound in the frist state of the transformation shown on Figure 9. The behavior of the second state of σ is modeled by the TGG rule on Figure 10. The main difference with Figure 8 is that the *ca* and *ra* nodes are already bound by being passed as function parameters instead of being matched from the *at2at* node.

Control flow alternative Instead of mixing Story Diagrams with TGG rules on one level of abstraction, one could keep the high level TGG rules and the low-level interaction and control flow details stricly separate. This however leads to subtle dependencies between TGG rules and derived story diagrams. Moreover, since details are added to the Story Diagrams that are generated from the TGG rules, unexpected behavior can be introduced at the Story Diagram level. Therefore, convenient navigation should be provided from TGG rules to the derived Story Diagrams and back. In [17] we proposed the use of traceability links to manage this complexity.

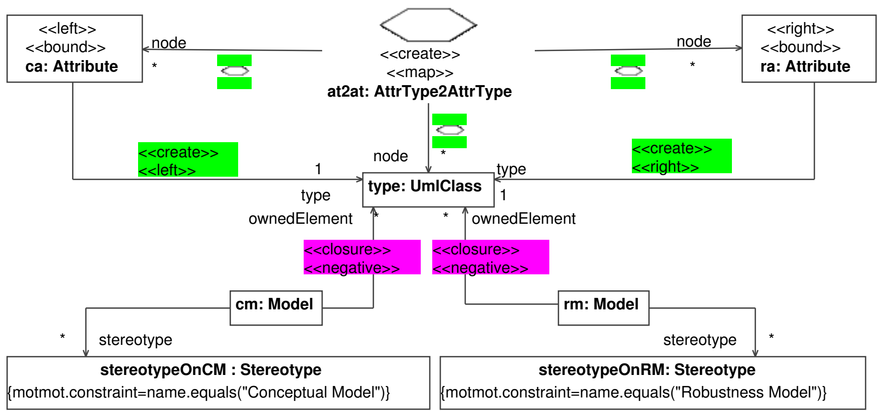


Fig. 10. Example of a controlled TGG rule: *ca* and *ra* are bound by being passed as function parameters.

6 Related Work

The emerging QVT standard [9] refers to the combination of its declarative and imperative sublanguages as a hybrid transformation language. However, the combination of these sublanguages has not been demonstrated on an actual transformation problem yet. The most concrete proposal of integrating declarative language features with imperative ones to date may have been published by Jouault and Kurtev [6]. Their *ATLAS Transformation Language* (ATL) allows one to map elements in *called rules* with the same syntax as that for *matched rules*. Matched rules can be compared with graph grammar rules. Called rules make no use at all of a matching engine. Therefore, ATL could not be employed as the hybrid transformation language introduced in this paper: in Subsection 5.1 we analyzed under what condition two declarative rules would result in an ambiguity and resolved it by defining an imperative rule with higher precedence. This imperative rule used input from the modeler (which would also be supported by ATL's *native* called rules) to delegate to the proper declarative rule (which would also be supported by ATL). ATL's limitation is that the imperative rule cannot be scheduled between (or in this case: *before*) declarative rules. It should also be noted that the main declarative power of TGG rules is that they support bidirectional consistency maintenance with very low specification effort. Future versions of declarative ATL may benefit from an evolution compared with that from pair graph grammars to triple graph grammars, as discussed in Subsection 3.3.

7 Summary and Outlook

This paper aimed to model a complex transformation problem in an intuitive manner. The complexity of the case study consisted of the variety of ways the overall consistency of two models could be violated and the need for developer interaction in the conflict resolution process. After illustrating that neither a purely imperative, nor a purely declarative approach was sufficient to solve this case study in a desirable manner, the integration of two formalisms was used as the first concrete application of a hybrid transformation language.

The advantage of the declarative features of Triple Graph Grammars are that incremental and bidirectional transformations can be specified concisely. On the other hand, adding imperative features such as method definition and calling, sequential composition, branching and looping was required for incorporating some human intelligence into the system.

In our current work, we are comparing the "merged language" approach presented in this paper with the use of the "two-level language architecture" presented in [17]. These alternative approaches for reconciling cociseness with completeness are discussed with the main international TGG tool developers and may be supported by a future version of Fujaba [16] or MoTMoT [7].

Language Integration By Metamodel Merging The advantage of the approach presented in this paper is that the semantics of a TGG rule is defined unambiguously. A potential disadvantage is that some TGG rules become slightly more difficult to reason about. Moreover, a merged language may suffer from compromises that need to be made for satisfying the expectations of developers familiar with the original languages. In the case of the proposed Controlled TGG language, we preserved the mapping of a TGG rule to its six operational rules. However, some of these cases could never be triggered in the given example. This is a possible source of confusion and runtime performance problems (due to redundant matching).

Language Integration By Metamodel Mapping The advantage of the approach from [17] is that TGG rules remain simple by handling ambiguities only at the level of the derived operational rules. The disadvantage from [17] is that without a proper traceability tool, it is hard to understand the complete semantics of a set of TGG rules. It should be noted that the emerging QVT standard [9] proposes a two-level transformation language architecture as well which makes [17] applicable in that context too.

Conclusion

This paper demonstrated the feasability of merging a declarative model transformation language with an imperative one. By combining the advantages of Triple Graph Grammars with those of Story Diagrams, compactness was reconciled with expressiveness. This experiment raised an interesting research question for the language engineering community: what are, in the context of language integration, the advantages of metamodel merging over metamodel mapping and vice versa? Although a preliminary comparison could be given, more research in this direction should be conducted for enabling a more mature tradeoff analysis.

References

1. A. v. Lamsweerde and R. Darimont and P. Massonet. The Meeting Scheduler System - Problem Statement. Technical report, Université Catholique de Louvain - Département d'Ingénierie Informatique, B-1348 Louvain-la-Neuve (Belgium), 1992.
2. Albert Zündorf. *Rigorous Object Oriented Software Development*. PhD thesis, University of Paderborn, 2001.
3. Robert Balzer. Tolerating inconsistency. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 158–165, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.
4. S. Becker, T. Haase, and B. Westfechtel. Model-based a-posteriori integration of engineering tools for incremental development processes. *Journal of Software and Systems Modeling*, 4:123–140, 2004.
5. Stephen Cranefield and Jin Pan. Bridging the gap between the model-driven architecture and ontology engineering. Technical Report 2005/12, Department of Information Science, University of Otago, Dunedin, New Zealand, November 2005.
6. Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, Jan 2006.
7. Olaf Muliawan, Hans Schippers, and Pieter Van Gorp. Model driven, Template based, Model Transformer (MoTMoT). http://motmot.sourceforge.net/, 2006.
8. Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Leveraging inconsistency in software development. *Computer*, 33(4):pp. 24–29, 2000.
9. Object Management Group. *MOF QVT Final Adopted Specification – ptc/05-11-01*, 2005. http://www.omg.org/docs/ptc/05-11-01.pdf.
10. Pieter Van Gorp, Frank Altheide, and Dirk Janssens. Traceability and Fine-Grained Constraints in Interactive Inconsistency Management. In Tor Neple, Jon Oldevik, and Jan Aagedal, editors, *Second ECMDA Traceability Workshop*, 10 July 2006.
11. Robert Wagner. Consistency Management System for the Fujaba Tool Suite – MoTE/-MoRTEn Plugins. https://dsd-serv.uni-paderborn.de/projects/cms/, 1 August 2006.
12. 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.
13. Hans Schippers, Pieter Van Gorp, and Dirk Janssens. Leveraging UML profiles to generate plugins from visual model transformations. *Electronic Notes in Theoretical Computer Science*, 127(3):5–16, 2004. Software Evolution through Transformations (SETra). Satellite of the 2nd Intl. Conference on Graph Transformation.
14. Andy Schürr. Progres: A visual language and environment for programming with graph rewrite systems. Technical Report AIB 94-11, RWTH Aachen, Fachgruppe Informatik, 1994.
15. 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.
16. 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.
17. Pieter Van Gorp, Frank Altheide, and Dirk Janssens. Towards 2D Traceability in a Platform for Contract Aware Visual Transformations with Tolerated Inconsistencies. In *Enterprise Distributed Object Computing Conference (EDOC)*, Hong Kong, 16 October 2006. IEEE.
18. 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, 1 April 2006. Elsevier.