Evaluation of Rule-based Modularization in Model Transformation Languages illustrated with ATL

Ivan Kurtev
ATLAS Group, INRIA and LINA
ivan.kurtev@univ-nantes.fr

Klaas van den Berg
Software Engineering Group, University of Twente
k.g.vandenberg@ewi.utwente.nl

Frédéric Jouault
ATLAS Group, INRIA and LINA
frederic.jouault@univ-nantes.fr

ABSTRACT

This paper studies ways for modularizing transformation definitions in current rule-based model transformation languages. Two scenarios are shown in which the modular units are identified on the base of the relations between source and target metamodels and on the base of generic transformation functionality. Both scenarios justify modularization by requiring adaptability and reusability in transformation definitions. To enable representation and composition of the identified units, a transformation language must provide proper modular constructs and mechanisms for their integration. We evaluate several implementations of the scenarios by applying different transformation techniques: declarative and imperative implementations, usage of explicit and implicit rule calls, and usage of rule inheritance. ATLAS Transformation Language (ATL) is used to illustrate these implementations. The evaluation makes software engineers aware of advantages and disadvantages of the considered techniques and possible anomalies in the transformation definitions regarding their reusability and adaptability properties. The experience with these scenarios shows that current languages provide a reasonably full set of modular constructs but may have problems in handling some composition tasks.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Specialized Application Languages – *model transformation languages.* D.3.3 [**Programming Languages**]: Language Constructs and Features – *modules, packages.* D.2.13 [**Reusable Software**]: Reusable Libraries.

General Terms

Design, Languages

Keywords

Model transformations, transformation languages, modularity, reusability, adaptability, ATL

1. INTRODUCTION

In Model Driven Engineering (MDE) model transformations executed according to transformation definitions play an important role in the development process. Transformation definitions may be a subject of reuse, adaptation and composition in the same manner as the traditional software artifacts such as classes and libraries are reused, adapted and composed. Modularizing a transformation definition into units helps in performing these tasks. Modularization requires decomposition of definitions and thus may help in reducing the complexity in the design of a transformation. Composition of existing modules promotes reusability and makes the development faster. Explicit representation of a composition provides a finer control over the modules affected by changes and therefore improves the adaptability of transformations.

Most of the current transformation languages are rule-based, that is, transformation rule is the basic modular construct [4][7][12][1]. Several research questions arise in respect to expressing transformations with rule-based languages. We may study the modularization issue at two levels: conceptual level and language level. At the conceptual level we are interested in identifying the functionality that will be encoded in modules. Generally, a transformation is built on the base of source and target metamodels. They are modularized in a certain way and it is naturally to expect that the modularization of the metamodels affects the modularization of transformation definitions. However, is that the only source for modularization of transformation definitions? Is there some transformation functionality not related to the metamodels that should be modularized? Furthermore, there are usually multiple possible modularizations. A particular modularization is often justified by certain quality properties that it brings such as reusability and adaptability. Therefore we are interested in identifying those modularizations that satisfy the required quality properties.

At the language level we are interested in expressing the identified conceptual modules by first-class language constructs. Ideally, there should be one-to-one mapping among the conceptual modules and the available language modules. Furthermore, modules should be integrated in order to bring a complete functionality. Therefore, apart from the modular constructs we need also compositional operators in order to integrate the modules. This brings the next important question: are the current transformation languages capable of expressing the required modular structure without compromising the desired quality of the transformation definitions.

The approach we take to answer these research questions explores two transformation scenarios. We study how the modularization of metamodels affects the modularization of transformation definitions and how the evolution of metamodels may affect the identified transformation units. For every scenario a set of transformation rules is identified. This is done independently from a particular transformation language. Identified transformation rules are conceptual and they specify relations between the source and target metamodels. On the base of these rules we identify pieces of functionality that needs to be modularized. Particular modularization is justified by various quality requirements imposed on transformation definitions.

The identified language independent modular units have to be mapped to the modular constructs in a given language. Instead of studying available transformation languages one by one we take a more general approach by studying transformation techniques commonly found in existing languages. For example, we have imperative and declarative languages with various forms of rule ordering (implicit and explicit), various forms of traceability support, etc. We implement the transformation scenarios with each of the techniques. To bring more concreteness to the discussion the implementations are presented in ATLAS Transformation Language (ATL) [2][7].

Two issues are studied for each implementation. The first one is the possibility to express the conceptual modules and their integration by the chosen technique. The second one is the quality of the resulting transformation definition. We believe that the result of our study is important from both practical and theoretical point of view. It makes the software engineers aware of the advantages and disadvantages of the chosen technique with respect of modularization of definitions and their quality. It also suggests directions for improvements of current transformation languages.

This paper is organized as follows. Section 2 presents the transformation scenarios. Section 3 gives an overview of implementation techniques available in current transformation languages. Section 4 provides ATL implementations of the scenarios. Section 5 evaluates the implementations. Section 6 gives conclusions and directions for future work.

2. TRANSFORMATION SCENARIOS

The first scenario studies how decompositions found in the source and the target metamodels interact with each other within a single transformation definition. The second scenario shows an example where certain modules in a transformation definition are not derived from the source and target metamodels but are related to a generic transformation functionality independent from them. A more extensive study of the subject accompanied with more scenarios can be found in [9].

For every scenario we identify some transformation rules. We use a simple notation based on constructs commonly found in current rule-based transformation languages. We assume that rules have left-hand and right-hand side that relate elements in the source and target metamodels.

2.1 Scenario 1: Decomposition of Metamodels in Multiple Dimensions

This scenario studies how decompositions in the source and target metamodels may be used to identify rules in a transformation definition. The anticipated evolution in the metamodels guides the identification of the transformation rules. The scenario illustrates that more than one possible decomposition in the metamodels have to be considered. If certain decompositions are neglected then the resulting transformation definition may expose anomalies such as tangling and scattering of transformation functionality. These anomalies reduce the reusability of the transformation definition.

Consider a simple system called *The Examination Assistant* that supports teachers in performing computer-based examinations. The system presents exam questionnaires to students in several modes: *exam*, *self-test*, and *tutorial* mode. Modes vary in the level of control the student has over navigating and answering the questions. Questionnaires are stored as XML documents. Documents are interpreted by the Examination Assistant. The interpretation is implemented as a transformation executed on XML documents to produce a set of objects.

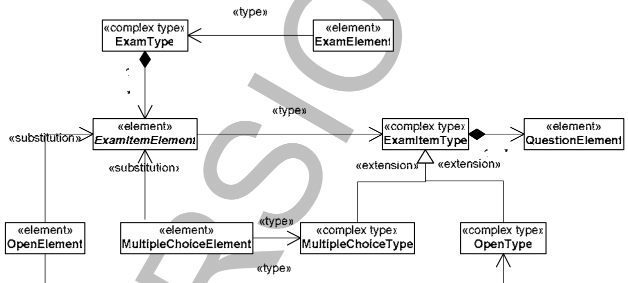

Figure 1 The XML Schema for examination documents

Figure 1 shows the XML schema of examination documents presented as an UML diagram with elements decorated with stereotypes to indicate the corresponding XML schema constructs.

In this scenario we only focus on the design of the user interface of the Examination Assistant. The design is based on the *Model-View-Controller* (MVC) design pattern [6].

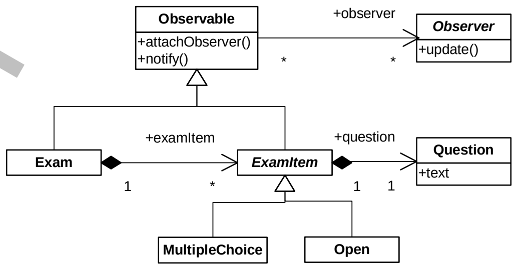

Figure 2 Model classes of Examination Assistant

Figure 2 shows the classes of the model part of the application. There are classes for the questionnaire (*Exam*) and classes for the exam items (*ExamItem*, *MultipleChoice,* and *Open*) that represent different exam item types. Every class is a specialization of class *Observable* according to the *Observer-Observable* design pattern [6].

Figure 3 shows the class hierarchy of the views and the controllers used in the application.

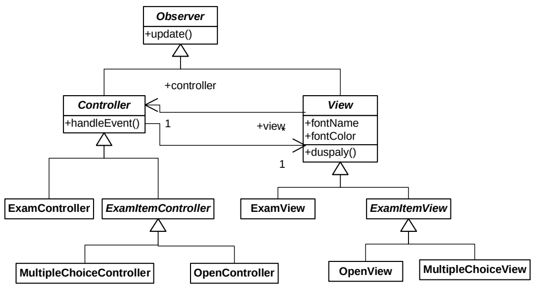

Figure 3 View and Controller classes of Examination Assistant

There are two abstract classes *Controller* and *View* that specialize class *Observer*. For every exam item type there are a concrete view and a concrete controller. An essential part of the Examination Assistant is the processing of questionnaires. We focus on this processing since it uses a transformation definition to transform XML documents to a set of application objects. The source metamodel of this definition is the XML schema shown in Figure 1 and the target metamodel is the application model parts of which are shown in Figure 2 and Figure 3[1].

How do we identify the rules in the transformation definition? Both the source and the target metamodels may be used as a starting point. For instance, for every exam item type in the source schema three classes from the application model have to be instantiated: the model, the view, and the controller. This leads to a possible decomposition of rules based on the exam item types. This approach may be regarded as a source-driven. Transformation rules are shown below:

Source-driven transformation:
openQuestionRule (**source**[OpenElement],
    target[Open, OpenView, OpenController])
multipleChoiceRule (**source**[MultipleChoiceElement],
    target[MultipleChoice, MultipleChoiceView,
        MultipleChoiceController])

There are two rules: for open question (*openQuestionRule*) and multiple choice (*multipleChoiceRule*) exam item types. The rest of the rules are skipped for simplicity. Every rule has a source and a target. The source is a tuple that contains model elements from the XML schema. The target is another tuple that contains model elements from the Examination Assistant application model. The interpretation of these rules is that for every match of the source tuple over an input model (an XML document), the model elements in the target tuple are instantiated. This basic interpretation is sufficient for our discussion and captures an essential part of the semantics of the rules found in most transformation languages.

The problem with this definition is that it involves *tangling* and *scattering* of transformation functionality. Scattering and tangling are code anomalies that often cause crosscutting [15][5]. The functionality affected by them is related to some important concerns found in the target application model. Four concerns may be identified. The first one is the type of the exam item. The rest three are related to the model, view, and controller parts of the application. These concerns, for example, may lead to a decomposition of the application classes into three sets containing the model, view, and controller classes respectively.

Transformation rules are decomposed along the first concern: the exam item type. Every rule refers to the classes for the model, view and controller in its target (*tangling*). Constructs related, for example, to the instantiation of the view classes are *scattered* across multiple rules. The same is valid for model and controller-related classes.

The scattering and tangling lead to problems if the application model evolves. Assume that the current implementation of the controller classes performs the *self-test* mode of examination in which the student is able to navigate through and to answer the exam questions. As a possible evolution of the Examination Assistant a new mode is introduced called *tutorial* mode in which students only navigate through the exam items without entering information. To implement this mode at least new controller classes have to be introduced that implement this mode of interaction. The controller classes for the self-test mode will be replaced.

[1] Note that these models are not metamodels in the sense of QVT RFP [11].

Due to the scattering the change of the exam mode leads to a series of changes in all the rules. Furthermore, the system may be switched back to the self-test mode and then the self-test mode controller classes will be included in the transformation. Therefore, these two parts of the transformation definition should be separated and reused. In the current specification, however, the controller-related transformation functionality is not separately specified. This hinders the reusability of the transformation rules.

The reason for the described anomalies is that only one dimension of decomposition is considered. The transformation is organized around the decomposition of the source schema into different exam item types. The same decomposition is possible within the target model. This ensures evolvable and reusable transformation definition along that dimension. If a new exam item type is introduced then a new transformation rule is added. However, we neglect the decompositions along other concern dimensions in the target model. The target model may be decomposed along multiple dimensions that form a multidimensional space according to the definitions given in [13] and [14].

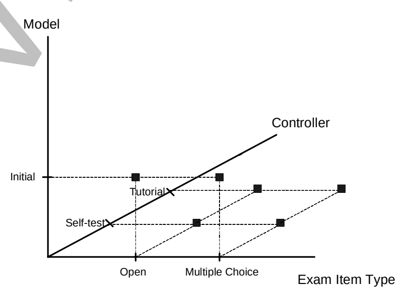

Figure 4 Decomposition of the target metamodel along multiple dimensions

Figure 4 shows three dimensions of decomposition: *Exam Item Type*, *Controller* and *Model*. The fourth dimension that corresponds to the *View* concern is not shown. Dimensions represent concerns in the target model. Each dimension has a set of coordinates that represent alternatives in the dimension. For example, *Controller* dimension has two alternatives: *Self-test* and *Tutorial*. The points in the space are classes (shown as solid rectangles).

In the current transformation definition the targets of the rules are based on only one dimension of decomposition: *Exam Item Type*. However, classes from that dimension also pertain to other two dimensions: *Model* and *Controller*. If the target model evolves along these two dimensions all the transformation rules must be changed. It is better to isolate each dimension in the target from the other dimensions. The new version of the transformation definition is the following:

Model-related part:
openQuestionModel (**source**[OpenElement], **target**[Open])
multipleChoiceModel (**source**[MultipleChoiceElement],
    target[MultipleChoice])
View-related part:
openQuestionView (**source**[OpenElement], **target**[OpenView])
multipleChoiceView (**source**[MultipleChoiceElement],
    target[MultipleChoiceView])
Controller-related part:

```
openQuestionController (source[OpenElement], target[OpenController])
multipleChoiceController (source[MultipleChoiceElement],
                          target[MultipleChoiceController])
```

This version eliminates the tangling and scattering observed in the previous version. The source of the rules is identified along the exam item type dimension. For a given exam item type the target of rules is decomposed along the other three dimensions: *Model*, *View,* and *Controller*. The benefits of the second version are the following. First, it allows adaptation along all the dimensions found in the target model. Second, the rules for different alternatives in the dimensions are reusable.

Until now we explored the impact of the decomposition in the metamodels on the transformation definition. In the remaining part of this scenario we focus on another aspect of transformations: setting property values.

Assume we want to impose a uniform layout style on the exam items by using the same font and color in all the views. This is set by the attributes *fontName* and *fontColor* of class *View*. These attributes must have the same values in all the exam item types. The part of the transformation that sets up the values is shown below. The attribute names and their values are shown as a list of comma separated assignments surrounded by curly braces.

```
View-related part:
openQuestionView (source[OpenElement],
                  target[OpenView {fontName='Times', fontColor='Red'}])
multipleChoiceView (source[MultipleChoiceElement],
                    target[MultipleChoiceView
                           {fontName='Times', fontColor='Red'}])
```

Apparently the values are repeated for every exam item type. Similarly to the previous example we have a reduced quality of the transformation definition. Changes in the layout style lead to identical changes in many rules. Furthermore, if we want to switch between different layout styles it is suitable to separate the values of the attributes in different modules that can be reused. The functionality of attribute value calculation is another example of *scattering* in the transformation definition.

The difference with the previous example is that the anomaly in the transformation definition is not caused by the decomposition of the application model. The scattering appears in the transformation definition where the concrete values are set. Of course, the example is very simple and the problem can be easily solved by providing the required values as a set of external parameters. However, in more complex cases the logic for calculating attribute values may scatter in a bigger scale.

This second example shows the need for modularization and reuse of the calculation of the attribute values for some model elements. The transformation definition should be adaptable with respect to changes in the layout style. Below we show the fragment of the view related part that sets up the values:

```
{fontName='Times', fontColor='Red'}
```

This fragment must be composed with a number of transformation rules.

2.2 Scenario 2: Trace Information

This scenario illustrates the need for modularizing and reusing part of the transformation logic that is independent of the source and target metamodels.

Assume we want to keep trace information about the correspondence between the source and target elements established during transformation execution. Trace information forms a model populated with elements every time a rule is executed over a source element. Models with trace information conform to the metamodel shown in Figure 5.

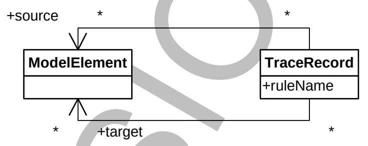

Figure 5 The metamodel for trace information

Whenever a transformation rule is executed class *TraceRecord* is instantiated and the name of the rule becomes value of the attribute *ruleName*. This instance also relates to the source and target elements of the rule. Some transformation engines may provide built-in traceability support. However, the user may need a customized structure of trace information and a better control when to generate traces. Therefore, this scenario is justified even in case of native support for traceability.

The generation of the trace information can be done by enhancing all the rules (or a selected subset) in a transformation definition with an instantiation of *TraceRecord*. For example, some of the rules in Scenario 1 will look like the following (newly added constructs are underlined):

```
Model-related part:
openQuestionModel (source[OpenElement],
                   target[Open,
                          TraceRecord{ruleName="openQuestionModel",
                          source=OpenElement, target=Open}])
multipleChoiceModel (source[MultipleChoiceElement],
                     target[MultipleChoice,
                     TraceRecord{ruleName="multipleChoiceModel",
                     source=MultipleChoiceElement, target=MultipleChoice}])
```

Apparently the logic for generating the trace information is again scattered across multiple rules. If this logic is changed this would require changes in all the rules. Also, this is a piece of generic functionality independent of a particular transformation definition and can be reused in an arbitrary transformation. Finally, we need a loosely coupled main transformation definition and trace generation code. It should be possible to add the trace generation logic to already existing transformation definitions and to eventually exclude it later.

How do we specify this functionality in a separate module that can be reused? In this scenario the transformation logic does not form a rule with source and target parts since no concrete source can be identified. Instead, the functionality should be included in the target of every rule. There may be a problem if a given transformation language does not provide a proper construct.

3. IMPLEMENTATION TECHNIQUES

The presented scenarios will be used to evaluate the modularization support in rule-based transformation languages. Instead of selecting a set of languages and implement the scenarios in every language we take a different approach by focusing on some commonly found implementation techniques. We use the work of Czarnecki and Helsen [3] who analyze the domain of transformation languages and identify commonalities and variabilities among them. We focus on two aspects of transformation languages: modular constructs and integration mechanisms.

3.1 Modular Constructs

Modularity denotes ability to develop programs as assemblies of smaller units called *modules*. Meyer [10] treats modularity as a means to achieve two quality properties in software: *extensibility* and *reusability*. Modules must satisfy the criterion for *modular composability*. This means that modules may be combined with each other to produce other modules.

For the purpose of this paper we assume that *transformation rule* is the basic module for expressing transformation functionality. We assume that rules have left-hand and right-hand side or are like procedures that receive parameters and eventually return a result. We neglect the details of directionality, parameterization, and others. Whenever necessary we will refer to other constructs and the language that provides them.

3.2 Rule Integration Mechanisms

In this paper we use the term *rule integration mechanism* to denote a means for assembling a set of rules in order to achieve new functionality beyond the functionality of each single rule. This term may denote a compositional operator such as rule inheritance, rule call mechanism that allows one rule to use the functionality of another rule, or the built-in execution algorithm used by a transformation engine that executes a set of rules. The classification of Czarnecki and Helsen contains the following categories related to rule integration.

- **Rule scheduling**. This category includes mechanisms responsible for the order in which the rules are applied. These mechanisms may vary in their form, i.e. the way the order is expressed. The form may be *implicit* and *explicit*. Implicit form of scheduling relies on implicit relations among the rules. Explicit form of scheduling uses dedicated constructs to control the order. Explicit scheduling may be *internal* and *external*. Internal scheduling uses control flow structures within rules and explicit rule invocation. External scheduling uses scheduling logic separated from the transformation rules;
- **Rule organization**. This category is concerned with relations among transformation rules. In this paper we focus only on *rule inheritance* as a mechanism to construct new rules from existing rules;
- **Traceability links**. We consider this mechanism as a way for communication among rules and therefore a mechanism for integration since a rule may access results produced by another rule;

On the base of the categories we may extract three mechanisms for rule integration:

- **Implicit rule calls**. This mechanism underlines the implicit rule scheduling and relies on rule dependencies. Rules may trigger/call the execution of other rules without necessarily using explicit references to rule names. Implicit rule calls are supported in DSTC (an early submission to QVT RFP) [4] and ATL (see next section), for example. The mechanism is more common in declarative languages. A rule is triggered when another rule needs the result produced by the rule. It should be noted that this does not mean that the rule is immediately executed. This depends on the scheduling algorithm of the transformation engine. The result may already have been produced or the rule may be executed later;
- **Explicit rule calls**. Explicit rule calls are usually found in hybrid and imperative languages. The mechanism is used in languages with explicit scheduling in both internal and external form. In the internal form, a rule may invoke another rule. In the external form, rules usually do not directly invoke each other. This is specified outside the rules where a rule application order is given (a mechanism found in some graph transformation based languages such as Great [1]);
- **Rule inheritance**. This is a basic reuse mechanism that allows one rule to be created by inheriting functionality from another rule;

We do not claim exhaustiveness of the list of mechanisms. The reason for selecting them is their presence in most of the current transformation languages.

4. IMPLEMENTATION OF TRANSFORMATION SCENARIOS

We present implementations of the two scenarios by using the integration mechanisms described in the previous section. The implementation is done in ATLAS Transformation Language (ATL) [2][7] that provides the three mechanisms. ATL is a hybrid language that employs declarative rules with left-hand and right-hand side (*matched rules*) and rules that are similar to procedures (*called rules*). These features make the language suitable to illustrate our discussion. We do not give detailed explanation of the syntax and semantics of ATL. Whenever we think that the syntax is not self-explanatory and the semantics not intuitive enough some clarification is provided. Moreover, the syntax is slightly adapted to the illustrative purposes of this part of the paper. Some features are not supported yet by the current ATL engine so this paper should not be used as an ATL reference. Our focus is mainly on general techniques rather than on a concrete language. In general, every language that supports the mechanisms described in section 3 may be used to illustrate the implementation of the scenarios.

4.1 Scenario 1

4.1.1 Using Implicit Rule Calls

The following code provides a declarative implementation of Scenario 1 without specifying the setting of the layout properties in a separate module.

```
rule OpenQuestionModel{
 from e : OpenElement
 to mOpen : Open(
        observer<-[vOpen]e
    )
}
rule MultipleChoiceModel{
 from e : MultipleChoiceElement
 to mMultipleChoice : MultipleChoice(
        observer<-[vMultipleChoice]e
    )
}
rule OpenQuestionView{
  from e : OpenElement
  to vOpen : OpenView(
16.       controller<-[cOpen]e,
17.       fontName<-'Times',
18.       fontColor<-'red'
19.     )
20. }
21. rule MultipleChoiceView {
22.  from e : MultipleChoiceElement
23.  to vMultipleChoice : MultipleChoiceView(
24.       controller<-[cMultipleChoice]e,
25.       fontName<-'Times',
26.       fontColor<-'red'
27.     )
28. }
29. rule OpenQuestionController{
30.  from e : OpenElement
31.  to cOpen : OpenController(
32.       view<-[vOpen]e
33.     )
33. }
34. rule MultipleChoiceController{
35.  from e : MultipleChoiceElement
36.  to cMultipleChoice : MultipleChoiceController(
37.       view<-[vMultipleChoice]e
38.     )
39. }
```

There are 6 rules that directly correspond to the rules identified in section 2.1. For example, rule *OpenQuestionModel* (lines 1-6) creates instances of class *Open* (line 3) from the elements instances of *OpenElement* (line 2). The source and target of the rules are indicated by the keywords *from* and *to*. The code in line 4 sets the value of the property *observer*. This value is a view object created by rule *OpenQuestionView* (lines 13-20). To obtain this object the implicit rule call mechanism is used. The expression `[vOpen]e` returns the object created from the source element *e* and assigned with the identifier *vOpen* (line 15). The rule that creates this object is not referenced. The rule is identified by the internal resolution algorithm that resolves the references to the target model elements created for a given source element. The identifiers used in the target parts of the rules form a kind of interface for communication among rules. The code in line 4 may use any rule that creates an object assigned with the identifier *vOpen*. Similar mechanism is provided by DSTC where a separate tracking class keeps a record of the correspondences between source and target elements. Such a class has a name that is used as a key. Rules do not refer to each other by name. Instead they use the tracking class names as an interface.

In this example all the requirements for reusability and adaptability of rules formulated in Scenario 1 are satisfied. Elements that belong to different dimensions of decomposition are not coupled in a single rule. New rules may be added provided that they keep the same identifiers of the target elements. This is the only coupling between the rules. ATL currently provides even looser coupling in which no identifier is specified and only a source element is passed to the resolution algorithm. In this case, however, there must be exactly one target element created from the given source element. If there are more than one element an ambiguity in the resolution may occur. In our case we cannot apply this because multiple target elements are created from a single source element.

In principle, it is possible to use the rule name and the identifier to resolve the target element. This brings additional coupling construct between rules. This means that if other rules are added for the controller classes they must have the same name as the replaced rules. This can be achieved by distributing rules across packages and only one package should be included in the transformation definition. A potential problem is if the required rules already exist and have different names.

The presented implementation does not address the problem with the scattering of layout properties. They are repeated twice: in rules *OpenQuestionView* and *MultipleChoiceView*. In the description of Scenario 1 we required that this functionality should be in a separate module for the purpose of reuse and should allow replacement by other modules that set other layout properties. In the next example we try to solve this problem by using explicit rule calls.

4.1.2 Using Explicit Rule Calls

In general, for rules *OpenQuestionView* and *MultipleChoiceView* we need to separate the assignments of the properties of the views into a new rule. This can be done in DSTC where a single target element may be initialized by multiple rules but only one instantiation is performed by the transformation engine. Separation of assignments of properties in a module is also possible in language MISTRAL described in [9] where assignments are decoupled from the instantiations. Unfortunately, in the declarative part of ATL assignments are coupled to instantiations. We will use an imperative feature named *called rule* to implement the scenario. Called rules act like procedures with input parameters and eventually a produced result.

The code below shows the new implementation. The rule *SetLayout* assigns the layout properties of the views (lines 19-24). It is called from the other two rules (lines 7 and 16). This rule is an example of a called rule in ATL. The *do* construct (lines 6, 15, and 20) is used to specify an imperative block in ATL rules.

```
1. rule OpenQuestionView{
2.  from e : OpenElement
3.  to vOpen : OpenView(
4.       controller<-[cOpen]e
5.     )
6.  do{
7.       SetLayout(vOpen);
8.  }
9. }
10. rule MultipleChoiceView {
11.   from e : MultipleChoiceElement
12.   to vMultipleChoice : MultipleChoiceView(
13.        controller<-[cMultipleChoice]e
14.   )
15.   do{
16.        SetLayout(vMultipleChoice);
17.   }
18. }
19. rule SetLayout(view : ExamItemView){
20.   do {
21.      view.fontName<-'Times';
22.      view.fontColor<-'red';
23.   }
24. }
```

The main difference with the previous implementation is that the assignment of the layout properties is located in a single rule. The requirement for reusability is achieved in this way. Achieving the requirement for adaptability to different layout styles depends on the used integration mechanism. In case of explicit rule calling, if a new rule for layout properties is to be used the invoking code must be changed to reflect the name of the new rule. To the best of our knowledge this issue is not addressed in current proposals for transformation languages. Of course, we may provide a rule with the same name in a different module but this may not be applicable if the rule exists in advance.

There is another way for integrating the rule for layout properties with the rest of the rules. It is based on rule inheritance and is explained in the next section.

4.1.3 Using Rule Inheritance

In this implementation only ATL declarative rules are used. In the code shown below rule *ExamItemView* (lines 1-7) is an abstract rule that sets only the layout properties. It is extended by the other two rules that add the other properties.

```
1. abstract rule ExamItemView{
2.  from e : ExamItemElement
3.  to vExamItem : ExamItemView(
4.        fontName<-'Times',
5.        fontColor<-'red'
6.     )
7. }
8. rule OpenQuestionView extends ExamItemView{
9.  from e : OpenElement
10. to vExamItem : OpenView(
11.        controller<-[cOpen]e
12.    )
13. }
14. rule MultipleChoiceView extends ExamItemView{
15.  from e : MultipleChoiceElement
16.  to vExamItem : MultipleChoiceView (
17.        controller<-[cMultipleChoice]e
18.  )
19. }
```

The requirement for separation of the assignment and its reusability is satisfied by this implementation. However, there are problems with the requirement for adaptability. Assume that a new rule is to be used for setting the layout properties. This means that rules *OpenQuestionView* and *MultipleChoiceView* must inherit from the new rule. Inheritance mechanism does not allow this, however. It introduces tight coupling between rules. In current languages there is no way to break this coupling with the available language constructs.

There is another way to apply rule inheritance. The rules *OpenQuestionView* and *MultipleChoiceView* may be used as base rules and other rules will add the layout properties to them through inheritance. Although the adaptability is improved, this is not a viable solution because the assignment of layout properties will be repeated as many times as the number of the extended rules. This is another code anomaly that reduces the maintainability.

4.2 Scenario 2

In this scenario the functionality that must be separated and reused is responsible for trace generation. It cannot be expressed as a standalone rule with left and right-hand sides since there is no fixed rule source. Only rule inheritance and explicit rule call mechanisms are applicable here

Rule inheritance leads to problems similar to problems described in section 4.1.3. We either cannot change the parent rule or end up with repetition of identical code. In order to save space we do not give an example here.

An example implementation based on explicit rule call is shown for only one rule that generates trace information (rule *MultipleChoiceView*). Trace generation is performed by *GenerateTrace* rule (lines 10-17).

```
1. rule MultipleChoiceView {
2.   from e : MultipleChoiceElement
3.   to vMultipleChoice : MultipleChoiceView(
4.        controller<-[cMultipleChoice]e
5.   )
6.   do{GenerateTrace(e, vMultipleChoice,
7.                   'MultipleChoiceView);
8.   }
9. }
10. rule GenerateTrace(source : OclAny, target : OclAny,
11.                   name : String){
12.   do {t : TraceRecord = new TraceRecord;
13.       t.source<-source;
14.       t.target<-target;
15.       t.ruleName<-name;
16.   }
17. }
```

Here the coupling introduced by the rule calling causes one additional problem apart from the potential need for name change. We require the ability to remove the trace generation functionality from the rule. However, this is not possible here since the rule call cannot be removed once introduced in the code.

5. EVALUATION OF IMPLEMENTATIONS

Presented implementations showed that transformation rule as a basic modular unit in transformation languages is sufficient to handle the scenarios. Integration mechanisms, however, introduce problems. In principle they allow achieving the required functionality. The problem is that the quality requirements for reusability and adaptability of transformation definitions are not met.

In Scenario 1 the adaptability is not achieved because of the coupling introduced by the explicit rule call and inheritance mechanisms. This observation is also valid for Scenario 2. In addition, the use of inheritance may introduce repetitions of code in the transformation definition if adaptability must be met. In this case achieving one quality property leads to an anomaly that reduces another quality property: maintainability.

The scenarios show that the software engineers should be careful about the level of coupling introduced by various integration mechanisms. If the mechanisms have to be ordered according to the degree of coupling the usage of declarative rules with implicit rule calls seems to be the most appropriate solution. Unfortunately it is not applicable in cases of resolution ambiguity and when the modular units are not rules with left and right hand side. Explicit rule calls and rule inheritance introduce a higher degree of coupling among the modules.

The common feature of both scenarios is that they require separation of functionality that is scattered among multiple modules. This functionality should be separated for reuse, it may be altered, and even excluded from transformation definitions. Therefore, a loose coupling among the modules is required to achieve this. That is exactly the point where current integration mechanisms fail.

The approach we took is to reason about the general features of integration mechanisms. As was mentioned in the paper, some languages provide specific mechanisms that solve some of the encountered problems. They may be used for further study in order to improve the modularity of the other languages.

6. CONCLUSIONS AND FUTURE WORK

In this paper we studied modularization techniques in rule-based transformation languages. Three main issues were addressed: the identification of modules in transformation definitions at a conceptual level, the ability to express these modules by the available language constructs, and the ability to integrate the modules. The main motivation for choosing a particular modularization is achieving certain quality properties of the transformation programs. On the basis of two scenarios we studied some ways to identify transformation rules. For instance, some of the rules can be derived from the correspondences among the elements in the source and target metamodels. Since multiple correspondences exist in the general case, software engineers must evaluate them having in mind the required quality. Scenario 1 showed that it is useful to consider more than one decomposition in the metamodels. Depicting the decompositions in a multidimensional space may help in identifying the proper rules. Scenario 2 showed that there may be transformation modules not derived from the involved metamodels.

The analysis of the scenarios implementations showed that the transformation rules in their two forms (*matched* and *called* in terms of ATL) are suitable modular units at least for these scenarios. The three integration mechanisms, however, led to problems in meeting the quality requirements for transformation programs. They introduced a degree of coupling between rules that cannot be handled by the currently available language constructs.

It should be noted that the problems may be overcome if higher-order transformations (HOTs) are employed, that is, transformations that manipulate other transformations. An example of using HOT to solve traceability issues in ATL can be found in [8]. However, this happens by using techniques not directly available in the languages. The analysis presented here should be used for introducing additional constructs that make possible to solve the problems within the employed language without resorting to external program manipulation. Identifying these constructs is an important direction for future research.

Both scenarios showed scattering and tangling: anomalies that are usually treated by aspect-oriented techniques [5]. It would be interesting to study the applicability of aspect-oriented techniques in transformation languages. Achieving modularity and reuse in the scenarios depends on the integration mechanisms. There are other situations where modularity and reuse is required. In [16] transformation rules are specified in a generic way and are later adapted for a particular metamodel by using higher-order transformations.

7. ACKNOWLEDGMENTS

This work has been partially supported by ModelWare, IST European project 511731.

8. REFERENCES

[1] Agrawal A., Karsai G., Kalmar Z., Neema S., Shi F., Vizhanyo A.The Design of a Simple Language for Graph Transformations, Journal in Software and System Modeling, in review, 2005

[2] Bézivin, J., Dupé, G., Jouault, F., Pitette, G., and Rougui, J. E., First experiments with the ATL model transformation language, 2nd OOPSLA Workshop on Generative Techniques in the context of MDA, USA, 2003

[3] Czarnecki, K., Helsen, S. Classification of model transformation approaches. OOPSLA2003 Workshop on Generative Techniques in the Context of MDA, USA, 2003

[4] DSTC, IBM, CBOP. MOF Query/Views/Transformations Submission. OMG document ad/2004-01-06, 2004

[5] Filman, R., Elrad, T., Clarke, S., and Aksit, M. Aspect-Oriented Software Development. Addison-Wesley. 2004

[6] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995

[7] Jouault, F., and Kurtev, I., Transforming Models with ATL, Workshop Model Transformations in Practice, part of the MoDELS 2005 Conference, Jamaica, 2005

[8] Jouault, F., Loosely Coupled Traceability for ATL, accepted for the ECMDA Workshop on Traceability, Nuremberg, Germany, 2005

[9] Kurtev, I. Adaptability of Model Transformations. PhD Thesis, University of Twente, ISBN 90-365-2184-X, 2005

[10] Meyer, B. Object-oriented software construction. Second edition, Prentice Hall PTR, 1997

[11] OMG. MOF 2.0 Query/Views/Transformations RFP. OMG document ad/2002-04-10, 2002

[12] OMG. Revised Submission for MOF 2.0 Query/View/Transformations RFP (ad/2002-04-10). OMG Document ad/2005-07-01, 2005

[13] Ossher, H., and Tarr, P. Multi-dimensional separation of concerns and the hyperspace approach. In M. Aksit (Ed.), Software Architectures and Component Technology (pp. 293-323). Kluwer Academic Publishers, 2002

[14] Tarr, P., Ossher, H., Sutton, S. M. Jr., and Harrison, W. NDegrees of Separation: Multi-Dimensional Separation of Concerns. In R. Filman, T. Elrad, S. Clarke, and M. Aksit (Eds.), Aspect-Oriented Software Development, (pp. 37-62), Addison-Wesley, 2004

[15] van den Berg, K., and Conejero, J.M. Disentangling Crosscutting in AOSD: A Conceptual Framework. European Interactive Workshop on Aspects in Software. Brussels, 2005

[16] Varro, D., Pataricza, A. Generic and Meta-transformations for Model Transformation Engineering. Proceedings of UML2004, LNCS 3273, Springer, 2004