Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 68 (2007) 111–127
Science of Computer Programming
www.elsevier.com/locate/scico

Rule-based modularization in model transformation languages illustrated with ATL

Ivan Kurtev[a,*], Klaas van den Berg[a], Frédéric Jouault[b,c]

[a] *Software Engineering Group, University of Twente, The Netherlands*
[b] *ATLAS Group, INRIA and LINA, University of Nantes, France*
[c] *Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL 35294-1170, United States*

Received 12 August 2006; received in revised form 20 April 2007; accepted 14 May 2007
Available online 14 July 2007

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 basis of 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: usage of explicit and implicit rule calls, and usage of rule inheritance. ATLAS Transformation Language (ATL) is used to illustrate these implementations. 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.

Keywords: Model transformation; Transformation language; 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. It is expected that transformation definitions will become an important asset and as such they should fulfill certain quality requirements, for example, reusability and adaptability. The experience with traditional software artifacts such as classes and libraries shows that a proper modular structure may help in achieving these quality properties. In the context of model transformations, 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. Explicit representation of a composition provides a finer control over the modules affected by changes and therefore improves the adaptability of transformations.

Modularization can be seen as a tradeoff between coupling and cohesion [16]. Modularity affects among others comprehensibility and maintainability [7]. It can be used for fault prediction [9]. Modularity in the context of a

* Corresponding author.
E-mail addresses: kurtev@ewi.utwente.nl (I. Kurtev), vdberg@cs.utwente.nl (K. van den Berg), frederic.jouault@univ-nantes.fr (F. Jouault).

0167-6423/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2007.05.006

programming language means ability to develop programs as assemblies of smaller units, usually called *modules*. A set of requirements may be imposed upon the modules. Meyer [15] treats modularity as a means to achieve two quality properties in software: extensibility and reusability. He considers the notion of modularity in the context of a software construction method and defines five criteria, five rules and five principles that must be met (respectively followed) in order to call a basic unit of decomposition a module. Here, we focus only on those criteria, rules, and principles that are relevant to our context. Modules must satisfy the criterion for *modular composability*. This means that modules may be combined with each other to produce other modules. Furthermore, Meyer defines the rule for *direct mapping*. The rule requires that the modular structure of a software system must be compatible with the model of the problem domain for which the system is built. Finally, we select the principle of *linguistic modular units* which states: "modules must correspond to syntactic units in the language used".

Most of the current transformation languages are rule-based, that is, transformation rule is the basic modular construct [11,1,2,13]. Rules are either structures with clearly separated left-hand and right-hand sides or are units similar to procedures in the imperative languages. It is quite straightforward to identify such a basic modular unit. A transformation usually relates two metamodels. Two-side rules provide finer granularity in relating metamodel elements. In imperative transformation languages the transformation logic is expressed in rules that may call each other. Many design decisions for current transformation languages are taken based on analogies with existing computer languages and paradigms (e.g., declarative vs. imperative, object-oriented vs. functional, etc.). In many cases such decisions would produce good results due to the available knowledge and successful practices in the area of computer languages. However, to achieve a good modularization system for a model transformation language a study of the problem domain of model transformations should be performed. To the best of our knowledge, there is no study focusing on the identification and justification of the necessary modular units in model transformation languages. It is not clear if other modules are needed with functionality different from the obvious rule functionality mentioned above. Furthermore, the modular units cannot be studied separately from the required module integration mechanisms.

In this paper, we analyze the modularity features of several model transformation languages. The goal is to evaluate the capabilities of different mechanisms to produce transformation definitions that are reusable and adaptable. The following two research questions are addressed.

First, what is the transformation functionality that needs to be encoded in modules? Generally, a transformation is defined on the basis of source and target metamodels. They are modularized in a certain way and it is natural to expect that the modularization of the metamodels affects the modularization of transformation definitions. However, is that the only source for identification of modules? Is there some functionality not related to the metamodels that should be modularized?

Second, there are usually multiple possible modularizations for a given transformation problem. A particular modularization is often justified by certain quality properties that it brings. We are interested in evaluating different modular constructs and integration mechanisms that ensure reusability and adaptability. For example, rules may be integrated by explicit rule calls or by rule inheritance. What are the properties of these mechanisms regarding the desired quality? To give an intuition for this research question we consider the degree of coupling between rules. The coupling directly affects the adaptability. Both explicit rule calls and rule inheritance introduce tight coupling. Therefore they should be ruled out and another mechanism should be applied. This brings the issue about the capability of the current transformation languages to express the required modular structure without compromising the desired quality of the transformation definitions.

The approach we take to answer the questions is based on case studies. We explore two transformation scenarios inspired by practical situations.

To answer the first research question, 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 basis of these rules we identify pieces of functionality that need 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. Such mapping reveals the adequacy of the language constructs to preserve the desired modularity. 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) [11].

To answer the second research question, we evaluate the implementations of the transformation scenarios from the point of view of reusability and adaptability. We believe that the result of this evaluation 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 to modularization of definitions and their quality. It also suggests directions for improvements of current transformation languages.

The major conclusions that we draw from the case studies can be summarized in two groups reflecting each research question. First, to achieve a proper modularization the software engineer should consider multiple possible decompositions in the source and target metamodels. This gives a larger set of alternatives for modularization of the transformation definition. Furthermore, metamodels are not the only source for identifying transformation modules. Second, to achieve reusable and adaptable transformation definitions an integration mechanism that ensures loose coupling between modules should be used whenever possible. Declarative languages with implicit ordering of rules allow more adaptable transformations in the general case. In contrast, imperative languages that rely on explicit rule calls and rule inheritance produce more tightly coupled structures. This hinders reusability and adaptability. Our support for these findings is given in the remaining part of the paper.

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

In the first scenario, we study 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 [12].

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 sides that relate elements in the source and target metamodels.

2.1. Scenario 1: Decomposition of metamodels in multiple dimensions

This scenario demonstrates 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.

Fig. 1 shows the XML schema of examination documents presented as an UML diagram, which elements are 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.

Fig. 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.

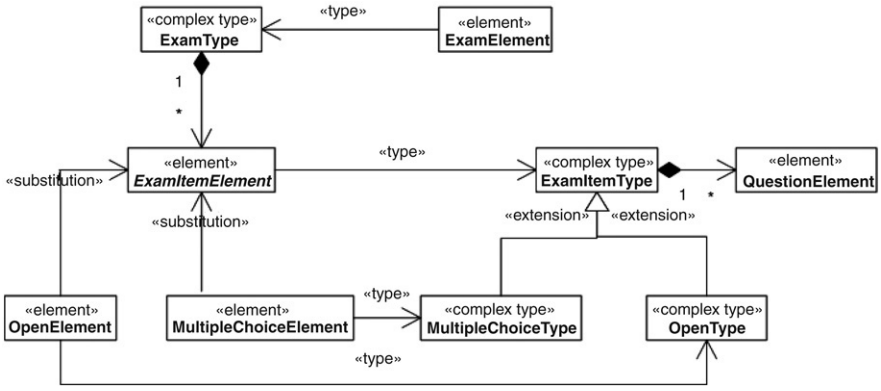

Fig. 1. The XML Schema for examination documents.

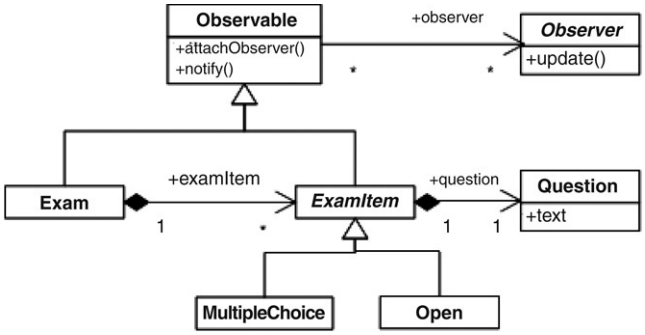

Fig. 2. Model classes of Examination Assistant.

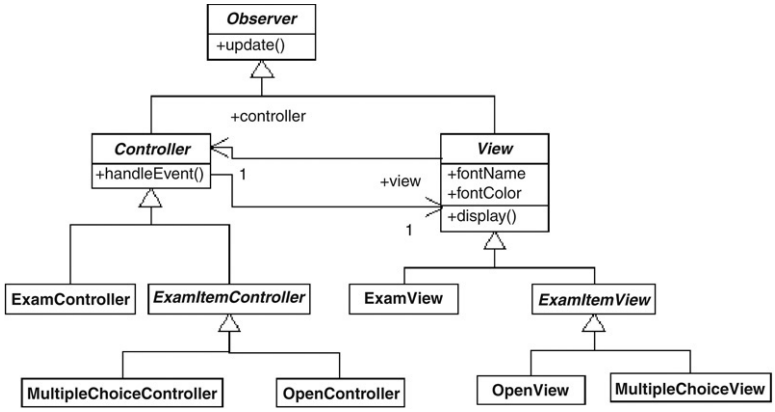

Fig. 3. View and Controller classes of Examination Assistant.

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

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 Fig. 1 and the target metamodel is the application model parts which are shown in Figs. 2 and 3.[1]

[1] Note that these models are not metamodels in the sense of QVT RFP [17]. If we use a metamodel for XML schemas located at level M2 then the schema in Fig. 1 will be at level M1 (i.e., at the level of models, not at the level of metamodels). Models in Figs. 2 and 3 are UML models residing at level M1.

How do we identify the rules in the transformation definition? Both the source and 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. 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. Tangling is mix of functionality within a single module. Scattering is spreading a single piece of functionality across several modules. Scattering and tangling are code anomalies that often cause crosscutting [4,8,20]. The functionality affected by these anomalies 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. Thus it includes units that belong to three different concerns. This is an example of tangling. The second anomaly called scattering is characterized by inclusion of units belonging to a single concern into more than one transformation rule. For example, the instantiation of the view classes is scattered across multiple rules. The same is valid for model- and controller-related classes.

Scattering and tangling lead to problems if the application model evolves. We demonstrate this by considering an evolution scenario. 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.

Due to 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 [18] and [19].

Fig. 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 *Model* dimension has currently only one coordinate called *Initial*. It is called

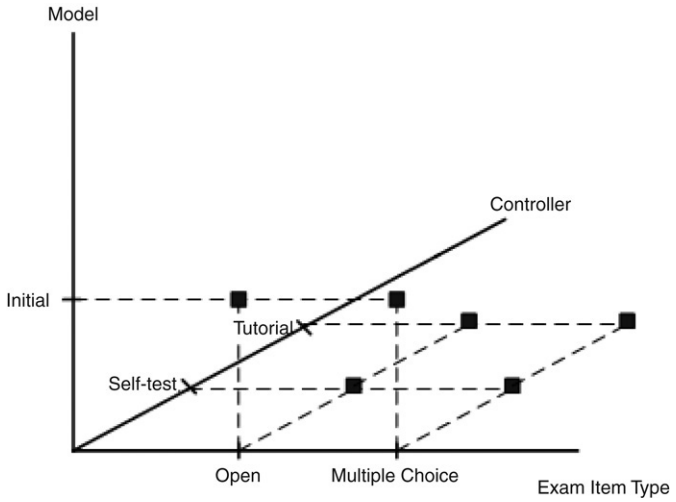

Fig. 4. Decomposition of the target metamodel along multiple dimensions.

like that because no alternatives are identified for the model part of the application yet and we keep the possibility for alternatives in the future. 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:

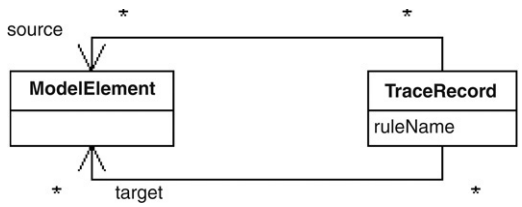

Fig. 5. The metamodel for trace information.

```
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 examples is that the anomaly in the transformation definition is not caused by neglecting some dimensions of decomposition in the target metamodel. The scattering appears because a set of concrete attribute values is repeated multiple times. It is not caused by the structure of the metamodel but by the transformation logic itself.

This second example shows the need for modularization and reuse of the functionality that calculates 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 attribute values:

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

This fragment must be embedded within a number of transformation rules.

It should be noted that this scenario does not address the possibility to have a condition that applies on the source elements and the possibility to have multiple elements in the rule source. Intuitively, it is clear that reusability of rules will be affected by this more complex structure of the rules. We need another case study to explore these issues and do not address them in this paper.

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. Such a trace information may for instance be used to perform change impact analysis if the source and the target models evolve later. 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 Fig. 5.

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 of 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.

The example shown above also demonstrates tangling. Every rule contains logic that belongs to two different concerns: the Model concern and the Traceability concern.

How do we specify the trace 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 [6] 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

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 sides 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., in 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. The latter one is often found in graph transformation languages [1,2]. In these languages rules have source and target patterns and no order of execution is presupposed. The order is specified outside rules by using some language for describing control flow;

– **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 basis 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 Tefkat [13], ATL (see next section), and VIATRA [2], 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. Explicit rule calls in internal form are supported by ATL and QVT languages. The external form is supported, for example, by the languages GReAT [1] and VIATRA [2];
– **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) [11] that provides the three mechanisms except the external specification of explicit rule scheduling. A form of this specification can be expressed in ATL since the language supports the typical control flow structures. The control flow of rule execution can be encoded in an imperative rule. In contrast, in some graph transformation languages the control flow is not encoded in the transformation rules. For example, VIATRA uses Abstract State Machine code for this purpose. GReAT provides constructs for sequencing transformation rules. In all these cases, the essential characteristic is the explicit call of rules by name. Therefore, the discussion remains valid for a graph transformation-based implementation (not presented here) and the ATL implementation presented in this section.

ATL is a hybrid language that employs declarative rules with left-hand and right-hand sides (*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.

```
1. rule OpenQuestionModel{ --Mapping of open question XML element to
2.   from e : OpenElement  --class Open
3.   to mOpen : Open(
4.         observer<-[vOpen]e --Property assignment relies on the ATL
```

```
5.      )                    --resolution mechanism
6. }
7. rule MultipleChoiceModel{        --Mapping of multiple choice XML
8.   from e : MultipleChoiceElement --element to class MultipleChoice
9.   to mMultipleChoice : MultipleChoice(
10.       observer<-[vMultipleChoice]e
11.     )
12.}
13. rule OpenQuestionView{ --Mapping of open question element to its
14.   from e : OpenElement -- view class
15.   to vOpen : OpenView(
16.        controller<-[cOpen]e,
17.        fontName<-'Times',
18.        fontColor<-'red'
19.      )
20. }
21. rule MultipleChoiceView {        --Mapping of multiple choice
22.   from e : MultipleChoiceElement --element to its view class
23.   to vMultipleChoice : MultipleChoiceView(
24.        controller<-[cMultipleChoice]e,
25.        fontName<-'Times', fontColor<-'red'
26.      )
27. }
28. rule OpenQuestionController{--Mapping of open question element
29.   from e : OpenElement      --to its controller class
30.   to cOpen : OpenController(
31.        view<-[vOpen]e --The view object is created by applying
32.      )               --rule OpenQuestionView.
33. }                    --Note that the rule name is not mentioned
34. rule MultipleChoiceController{   --Mapping of multiple choice
35.   from e : MultipleChoiceElement --element to its controller
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 *OpenQuestion-Model* (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 explicitly 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 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 with a slight difference. In this language a separate tracking class keeps a record of the correspondences between source and target elements. Tracking classes introduce a layer of indirection among rules. It is not necessary for rules to refer to each other by names. Rules may be exchanged freely provided that the same tracking classes are used by the rules. From certain point of view tracking classes form an interface layer between rules.[2]

[2] At a first glance there is circularity in the implicit calls among the rules. The ATL execution algorithm first creates target model elements and then assigns their properties. This guarantees that the transformation terminates. It is beyond the scope of this paper to give a full description of the ATL execution algorithm. For more details the reader is referred to [11].

The tracing mechanisms in QVT languages are slightly different from the ones described until now. In Relations language the execution engine handles the creation of trace records automatically and in a way transparent to the user. The Core language, in contrast, does not provide such a mechanism. Transformation developer is responsible for creating trace records. Therefore, in this language the implicit rule call technique cannot be used. Operational Mappings language provides a trace mechanism similar to the one in ATL.

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. Similarly, it is possible to have more than one rule that create target elements assigned to the same identifier, that is, we have sharing of identifiers across the rules. The resolution algorithm currently handles this as an ambiguity and raises an error. There are two solutions to this problem. The first solution uses the rule name and the identifier to resolve the target element. This brings additional coupling between rules. If new rules are used 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 second solution returns all the elements assigned to the given identifier. The rule developer is responsible to select the right element, for example, on the basis of the class of the element.

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 [12] 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 assignment of layout properties will be modularized in a called rule.

It has to be noticed that this decision breaks the principle of linguistic modular units described in the introduction. A set of assignments is not exactly a rule. The conceptual module that groups assignments together cannot be mapped to an equivalent syntactical module. All the languages that provide only rule as a modular unit do not follow the principle. Furthermore, if a language does not support a form of rules similar to ATL called rules this scenario cannot be implemented without compromising the quality of the transformation definition. The definition will expose scattering anomaly as in Section 4.1.1.

The code below shows the implementation based on called rules. 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); --Explicit rule call
8.   }
9. }
```

```
10. rule MultipleChoiceView {
11.   from e : MultipleChoiceElement
12.   to vMultipleChoice : MultipleChoiceView(
13.        controller<-[cMultipleChoice]e
14.      )
15.  do {
16.     SetLayout(vMultipleChoice); --Explicit rule call
17.  }
18. }
19. rule SetLayout(view : ExamItemView){ --An example of a called rule
20.   do {                   --Imperative section with two assignments
21.     view.fontName<-'Times'; view.fontColor<-'red';
22.   }
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 the 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{ --An example of rule
9.    from e : OpenElement                     --inheritance in ATL
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 again 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. This is another example of deviation from the principle of linguistic modular units.

4.2.1. Using explicit rule calls

An example implementation based on explicit rule calls is shown for only one rule that generates trace information (rule *MultipleChoiceView*). Trace generation is performed by *GenerateTrace* rule (lines 9–15).

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

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 since the rule call cannot be removed once introduced in the code.

4.2.2. Using rule inheritance

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. The example implementation is given below.

```
1. abstract rule MultipleChoiceView {
2.   from e : MultipleChoiceElement
3.   to vMultipleChoice : MultipleChoiceView(
4.         controller<-[cMultipleChoice]e
5.      )
6. }
7. abstract rule OpenQuestionView{
8.   from e : OpenElement
9.   to vOpen : OpenView(
10.        controller<-[cOpen]e
11.     )
```

```
12.}
13. rule TracedMultipleChoiceView extends MultipleChoiceView{
14.   from e : MultipleChoiceElement
15.   to t : TraceRecord(   --First occurrence of the trace
16.            source<-e,   --generation functionality
17.            target<-controller,
18.            ruleName<-'MultipleChoiceView'
19.      )
20. }
21. rule TracedOpenQuestionView extends OpenQuestionView{
22.   from e : OpenElement
23.   to t : TraceRecord(   --Second occurrence of the trace
24.            source<-e,   --generation functionality
25.            target<-controller,
26.            ruleName<-'OpenQuestionView'
27.      )
28. }
```

In this implementation we define the rules *TracedMultipleChoiceView* and *TracedOpenQuestionView* that extend two already existing rules by adding trace generation functionality. The trace generation logic is not included in the original rules thus overcoming the problem shown in the previous implementation. However, this logic is not modularized in a single reusable unit but is replicated twice in lines 15 and 23 (and potentially as many times as the number of the traced rules is). Therefore, potential changes in this logic are error prone because of the repetition.

4.3. Other mechanisms

In this section we comment other mechanisms that might be useful in the implementation of the scenarios. The reason to comment on them separately is that they do not fall in the classification scheme used here and are not widely present in the current transformation languages.

The first mechanism is *reflection*. Reflection can be applied in scenario 2 to specify the trace generation functionality in a generic and rule-independent way. We need at least a form of introspection that allows access to the properties of the rule that provides the execution context: rule name, currently matched source elements, and the newly instantiated target elements.

The applicability of reflection in transformation languages is not well studied. We are aware of only one language that claims support for reflection: RubyTL [5]. Unfortunately, we do not have first-hand experience with it and therefore cannot provide an evaluation.

The second mechanism is *templates* mechanism that is known as a way for improving the reusability of code. Templates are applicable in scenario 2. The application of templates in model transformations is discussed in [2] in the context of the VIATRA language. The transformation rules are specified in a generic way and are later adapted for a particular metamodel by using higher-order transformations.

It should be noted that the problems with the implementation of scenario 2 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 [10]. Other examples are shown in [21] using VIATRA language. HOTs use 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.

5. Evaluation of implementations

Presented implementations showed that transformation rule as a basic modular unit (in both forms of declarative and called rule) in transformation languages is not always sufficient to handle the scenarios. If we encode the trace generation logic and property values assignments in separate rules then we encounter twice a deviation from the

principle of linguistic modular units. In order to be fully compliant with this principle two additional unit types should be introduced. The first one groups a set of property assignments in a single unit that can be reused across transformation rules. The second one is a fragment of rule pattern. In scenario 2 this is a fragment of the target pattern. To achieve complete genericity this has to be combined with reflective features. As we mentioned these constructs are not present in the majority of the transformation languages.

The integration mechanisms that we applied also demonstrated some problems. In principle they allow achieving the required functionality in both scenarios. 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 sides. 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. Two research questions were addressed: what is the transformation functionality that needs to be modularized, and how different modularization mechanisms affect the reusability and adaptability of transformation definitions.

To address the first research question we studied the relations between source and target metamodels. Our goal was to identify the "ideal" modularization for a particular problem justified by certain quality properties. The identification of transformation modules at conceptual level is highly influenced by the decomposition in the source and target metamodels. 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 the choice of different decompositions in the target metamodel leads to different sets of rules. Therefore, it is recommended to consider more than one decomposition in the metamodels. Depicting the decompositions in a multidimensional space may help in identifying the proper transformation rules. We also found pieces of functionality not directly related to the metamodels. This functionality should be modularized regardless of the fact that it is not a complete rule. Therefore, new modules that encode only attribute value assignments should be considered.

Once a modularization is identified conceptually, it must be expressed in a given transformation language by preserving the desired properties. This leads to the second research question. To address this, we implemented two case studies using modular constructs commonly found in a set of contemporary transformation languages. We assumed that the transformation rule construct is the main modular construct at our disposal. The analysis of the implementations showed that the transformation rules being either declarative or imperative (known as *matched* and *called* in terms of ATL) are not always enough to express the desired modular structure identified conceptually. It is still possible to implement the scenarios only with rules but at the price of decreased quality caused by scattering and tangling. To overcome this, new modular constructs that represent parts of rules should be considered. We also assumed that three integration mechanisms can be used to combine rules into a complete transformation program: implicit rule calls, explicit rule call, and rule inheritance. The mechanisms that we have introduced led to problems in meeting the quality requirements for transformation definitions. Every mechanism introduces a degree of coupling

between rules. Implicit rule calls usually lead to a low coupling and therefore should be chosen when the reusability and adaptability of the transformation definitions is of a primary importance. Unfortunately this mechanism is not available in the imperative transformation languages where the execution flow is hard coded in the transformation program. Developers can use explicit rule calls and rule inheritance. The result is a higher degree of coupling between the transformation modules. This ultimately decreases the reusability of rules.

Both scenarios showed scattering and tangling: anomalies that are usually treated by aspect-oriented techniques [8]. It would be interesting to study the applicability of aspect-oriented techniques in transformation languages. We do not propose a complete aspect-oriented extension to transformation languages. We plan to experiment with a limited use of techniques that will overcome the scattering problem found in the second scenario.

The evaluation presented here does not use formal techniques. It was possible to draw interesting conclusions just by implementing two simple case studies. In the future we would like to study the modularization issues in a more formal and rigorous way. There is a significant amount of related work applied in the area of object-oriented languages. A set of empirical techniques may be applied to capture the structural complexity of code [7,14]. They include quantitative measurements of coupling and cohesion in modules. As we saw the decoupling between rules is crucial for achieving transformation definitions with good quality. Another possible technique is to apply automatic modularization by cluster identification [16]. It remains an open issue how the mentioned techniques can be adapted in the context of model transformation programs.

We studied two quality properties of transformation definitions: reusability and adaptability. In general, if one property is achieved or increased others may be lost or decreased. For example, the code in scenario 1 gets larger due to the decomposition into multiple rules aimed to increase adaptability. Furthermore, writing a transformation with the perspective of later reuse and adaptation is more difficult since related evolution scenarios should be explicitly identified. Opting for one quality property and giving lower priority to others is a matter of achieving a balance between quality factors.

Our study was performed by assuming that a transformation language possesses a set of features: it is a rule-based language and provides the described three rule integration mechanisms. Although the implementations were given in ATL, we may hypothesize that our conclusions are valid for languages that provide those or similar features. These are, for example, the QVT Operational Mappings language, which is close to imperative ATL and the language provided by DSTC. To prove this hypothesis, however, we need to provide concrete implementations of the case studies in these languages. This paper does not provide such implementations and leaves them as a follow up work.

Acknowledgments

This work was partially supported by ModelWare, IST European project 511731. We thank the anonymous reviewers for their comments that helped to improve the paper.

References

[1] A. Agrawal, Graph rewriting and transformation (GReAT): A solution for the model integrated computing (MIC) bottleneck, in: ASE, IEEE Computer Society, Los Alamitos, CA, USA, 2003.
[2] A. Balogh, D. Varró, Advanced model transformation language constructs in the VIATRA2 framework, in: SAC'06: Proceedings of the 2006 ACM Symposium on Applied Computing, ACM Press, New York, NY, USA, 2006.
[3] J.-M. Bruel (Ed.), Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2–7, 2005 Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 3844, Springer, 2006.
[4] J.M. Conejero, K. van den Berg, J. Hernàndez, Disentangling crosscutting in AOSD: Formalization based on a crosscutting pattern, Jornadas de Ingenieria del Software y Bases de Datos (2006) 325–334.
[5] J.S. Cuadrado, J.G. Molina, M.M. Tortosa, RubyTL: A practical, extensible transformation language, in: A. Rensink, J. Warmer (Eds.), ECMDA-FA, in: Lecture Notes in Computer Science, vol. 4066, Springer, 2006.
[6] K. Czarnecki, S. Helsen, Feature-based survey of model transformation approaches, IBM Syst. J. 45 (3) (2006) 621–645.
[7] D.P. Darcy, C.F. Kemerer, S.A. Slaughter, J.E. Tomayko, The structural complexity of software: An experimental test, IEEE Trans. Softw. Eng. 31 (11) (2005) 982–995.
[8] R. Filman, T. Elrad, S. Clarke, M. Aksit, Aspect-Oriented Software Development, Addison-Wesley, 2004.
[9] T. Gyimothy, R. Ferenc, I. Siket, Empirical validation of object-oriented metrics on open source software for fault prediction, IEEE Trans. Softw. Eng. 31 (10) (2005) 897–910.
[10] F. Jouault, Loosely coupled traceability for ATL, in: ECMDA Workshop on Traceability, Nuremberg, Germany, 2005.

[11] F. Jouault, I. Kurtev, Transforming models with ATL, in: Bruel [3], pp. 128–138.
[12] I. Kurtev, Adaptability of Model Transformations, Ph.D. Thesis, University of Twente, 2005.
[13] M. Lawley, J. Steel, Practical declarative model transformation with Tefkat, in: Bruel [3], pp. 139–150.
[14] C.V. Lopes, S.K. Bajracharya, An analysis of modularity in aspect oriented design, in: AOSD'05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development, ACM Press, New York, NY, USA, 2005.
[15] B. Meyer, Object-Oriented Software Construction, second edition, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
[16] B.S. Mitchell, S. Mancoridis, On the automatic modularization of software systems using the bunch tool, IEEE Trans. Softw. Eng. 32 (3) (2006) 193–208.
[17] OMG, QVT MOF 2.0 Query/Views/Transformations RFP, OMG document ad/2002-04-10. http://www.omg.org, 2002.
[18] H. Ossher, P. Tarr, Multi-dimensional separation of concerns and the hyperspace approach, in: Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, Kluwer, 2000.
[19] P.L. Tarr, H. Ossher, W.H. Harrison, S.M. Sutton Jr., *N* degrees of separation: Multi-dimensional separation of concerns, in: International Conference on Software Engineering, 1999.
[20] K. van den Berg, B. Tekinerdogan, H. Nguyen, Analysis of crosscutting in model transformations, in: J.O.J. Aagedal, T. Neple (Eds.), ECMDA-TW Traceability Workshop Proceedings 2006, No. A219 in SINTEF Report, 2006.
[21] D. Varró, A. Pataricza, Generic and meta-transformations for model transformation engineering, in: T. Baar, A. Strohmeier, A.M.D. Moreira, S.J. Mellor (Eds.), UML, in: Lecture Notes in Computer Science, vol. 3273, Springer, 2004.