On the interoperability of model-to-model transformation languages

Frédéric Jouault<sup>a,b,∗</sup>, Ivan Kurtev<sup>c</sup>

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

Received 2 August 2006; received in revised form 6 May 2007; accepted 14 May 2007
Available online 13 July 2007

Abstract

Transforming models is a crucial activity in Model Driven Engineering (MDE). With the adoption of the OMG QVT standard for model transformation languages, it is anticipated that the experience in applying model transformations in various domains will increase. However, the QVT standard is just one possible approach for solving model transformation problems. In parallel with the QVT activity, many research groups and companies have been working on their own model transformation approaches and languages. It is important for software developers to be able to compare and select the most suitable languages and tools for a particular problem. This paper compares several model-to-model transformation languages as a step in the direction of gathering knowledge about the existing model transformation approaches. The focus is on the major language components (sublanguages and their features, execution tools, etc.) and how they are related. The major goal is to motivate the need for language interoperability and to explore options and obstacles for such interoperability. We propose a set of heuristics to reason about the problems that must be addressed when translators between languages have to be developed. These heuristics are applied on several examples. The experience from these examples shows that achieving a large degree of interoperability is difficult since some languages expose incompatible features. We managed to identify, however, cases where the interoperability between languages is feasible and brings certain benefits.

© 2007 Elsevier B.V. All rights reserved.

Keywords: Model engineering; Model transformations; Transformation languages; Interoperability

1. Introduction

Model transformation is an important activity in Model Driven Engineering (MDE). OMG recognized this and accordingly issued the Query/Views/Transformations (QVT) RFP [22] to seek an answer compatible with its MDA standard suite: UML, MOF, OCL, etc. Several formal replies were given by a number of companies and research institutions. They evolved during the last three years and converged to a single proposal for the QVT standard language [25]. Some other transformational approaches and languages evolved in parallel to the OMG process. Atlas Transformation Language (ATL) [15] was initially conceived as an answer to the QVT RFP but later the language requirements evolved towards a larger set of transformational scenarios. Several languages built upon the strong
The foundation of graph transformations were elaborated independently of the OMG efforts: VIATRA2 [3], GReAT [2], AGG [27, 28]. The language submitted by DSTC [24] as a response to the QVT RFP continued as a separate initiative and was reused as a conceptual foundation of the declarative logic-based language Tefkat [19]. All these languages have some common goals and features, but also expose differences in their paradigms, constructs, underlying modeling approaches, etc. Despite the fact that they are designed as general-purpose model-to-model transformation languages, everyone has strong and weak points and demonstrates a better suitability for a certain set of problems.

The adoption of the QVT standard and the expected development of QVT tools should open a possibility for applying model transformations on a larger scale and in more non-trivial cases. The experience gained from this should be used for improving the language. Furthermore, we witness other approaches for model transformations along with the QVT. A set of different languages should provide a broader field of experimentation. This motivates us to keep working on ATL and studying other transformation approaches rather than completely adopting the QVT implementation.

As language engineering history has shown, no single language can be adapted to all application domains. We believe it is not different for model engineering, and especially for the model transformation domain. Moreover, although simple problems can often be solved using one language, more complex problems sometimes need several. Therefore, it is important to acquire knowledge about the strong and weak points of existing languages and their applicability. Furthermore, the need for multiple languages poses the problem of interoperability among them. A comparison of language features should be utilized for identifying the translation possibilities of transformation definitions from one language to another.

As a first step in this direction, we analyze and compare several transformation languages: QVT, ATL, Tefkat, VIATRA2, AGG, and GReAT. This list is not exhaustive. Our goal is to cover languages with available execution engines with a satisfactory level of maturity, and standard languages such as QVT are expected to play an important role in the future. Another goal is to cover a broad spectrum of paradigms and theoretical backgrounds.

This paper provides two main contributions. The first one is a comparison among the languages on the basis of their features that influence the possibility for language interoperability. This comparison may be used as a standalone result when transformation developers are faced with a given problem and must evaluate and choose among available solutions. The second contribution, which builds upon the first one and is the major objective of this paper, is to provide a framework for reasoning and identification for language interoperability and potential obstacles for it.

The results of our study showed that at a conceptual level, it is possible to have interoperation between pairs of languages (e.g., ATL and QVT, ATL and Tefkat) with different degrees of limitation. This interoperation is based on model transformations based on programs written in these languages. However, most languages expose complex architectures that consist of several sublanguages (e.g., ATL, QVT, VIATRA2). More than one possibility for implementing interoperability exists, and not all of them are feasible. Some are difficult (perhaps impossible) to implement due to conceptual and technical mismatches.

The paper is organized as follows. Section 2 defines the concept of interoperability in the domain of model-to-model transformation languages and motivates the need for it. Section 3 explains the approach we follow to analyze the problem. Section 4 describes languages. Section 5 summarizes the presentation and compares the languages. Section 6 presents reasoning over interoperability possibilities and obstacles. Section 7 gives conclusions.

### 2. Definition and motivation for interoperability

According to the IEEE Standard Computer Dictionary [13], interoperability is the ability of two or more systems or components to exchange information and to use the information that has been exchanged. In the context of transformation languages, we assume that the systems we are interested in are comprised of the tools that support the usage of a language: compilers, interpreters, debuggers, etc. The information produced by these systems may vary, covering artifacts like transformation programs written in a given language, traces produced after the execution of a transformation, error logs, and others. MDE uses the concept of model as a unification concept. Therefore, we state that transformation language systems produce the aforementioned artifacts in the form of models.

In this paper, we only focus on transformation programs written in a given language as artifacts that will be exchanged and used between systems. The problem of exchanging and using trace information is also interesting; however, we do not consider it here.
For the purpose of this paper, we assume the following definition of transformation language interoperability: it is an ability to execute programs written in one language with the tools designed for another language. An example is to execute ATL code on a QVT-compliant engine.

We identified the following four motivations for this kind of interoperability:

(i) **Problem-language adequacy.** A given transformation language may be especially adapted to a given set of problems, but is unable to solve different sets of problems as efficiently as another language. Some complex problems may even require the integration of solutions implemented using different languages. The ability to execute programs written in different languages on a single engine should ease such integrations.

(ii) **Execution.** A given language may have a poor engine to execute programs. The possibility to execute programs on a more mature and optimized engine originally designed for another language could improve performance. Some languages may not even have an engine. For instance, at the time of writing of this paper the QVT Core language is not supported by an engine. Moreover, a given model engineering platform may only provide a QVT engine to comply with the standard. Some users may still want to use another language on top of this platform.

(iii) **Support tools.** Debuggers, profilers, etc. may not exist for a given language. If it was executed using the environment of a language having more tools, it could benefit from them.

(iv) **Compliance to standards.** Compliance with standards may sometimes be achieved through interoperability. For instance, if (at least) one QVT language is executable on the ATL Virtual Machine, we would be able to claim QVT compliance for it.

In the next section, we outline and analyze approaches for achieving interoperability among transformation languages. Two main approaches gain more interest. The first one relies on language translation, which in the context of MDE is implemented as model transformation. The second approach relies on translation and integration of the transformation code to a common engine. Multiple languages may be compiled to the language of this engine in the same way that Java programs are compiled to bytecode.

3. Approach

According to the definition given in Section 2, we aim at executing programs written in a *source* language on an engine originally built for a *target* language. There are several possibilities for achieving this. The first one relies on language translation, which in the context of MDE is implemented as model transformation. The second approach relies on translation and integration of the transformation code to a common engine. Multiple languages may be compiled to the language of this engine in the same way that Java programs are compiled to bytecode.

**Fig. 1.** Performing language translations.
we are able to establish a set of translators then we achieve interoperability by translating a program expressed in a source language into a program that can be executed on the target engine.

Fig. 1(b) shows another approach. Programs in languages $L_1, \ldots, L_n$ are compiled to a Common Intermediate Language (CIL). An engine is available to execute programs expressed in CIL. The interoperability is possible because all the source programs are compiled to a common format and executed on a single engine. Integration between programs written in different languages is possible at the level of CIL. This approach satisfies motivation 1. This is not a new approach and is already employed in the Microsoft.NET platform where multiple languages (e.g., Visual Basic, C#, Java) are compiled to a common intermediate language [32]. The main problem with this approach is crafting the CIL. We performed an initial work in this direction by designing a layered architecture for ATL (see Section 4.2), in which ATL programs are compiled to bytecode executed by the ATL Virtual Machine (ATL VM). However, we have not performed studies on the suitability of this bytecode language as a target for compiling other transformation languages yet. Therefore, we treat the idea presented in Fig. 1(b) as a direction for future work and limit ourselves to the approach in Fig. 1(a).

We employ the following terminology to reason on transformation language interoperability.

Assume we have two model transformation languages $L_1$ and $L_2$ and two transformation programs $t_1$ and $t_2$ expressed in $L_1$ and $L_2$ respectively. Both $t_1$ and $t_2$ transform source models conforming to source metamodel $M_1$ into target models conforming to target metamodel $M_2$. If $m$ is a source model, with $t(m)$ we denote the target model obtained by applying the transformation $t$. We introduce a definition for equivalent transformation programs:

**Definition 1** (Equivalent Transformation Programs). Two transformation programs $t_1$ and $t_2$ respectively expressed in languages $L_1$ and $L_2$ are equivalent if for every source model $m$ conforming to $M_1$, we have $t_1(m) = t_2(m)$.

In this definition, we assume that we have an equivalence relation ‘=’ between two models that conform to the same metamodel (according to our assumptions this is $M_2$ here).

Now we can define the interoperability we aim at more formally:

**Definition 2** (Unidirectional Language Interoperability). Two languages $L_1$ and $L_2$ are related with unidirectional interoperability with respect to transformation $t$ iff for every transformation program $t_1$ expressed in $L_1$:

- $t(t_1)$ is a transformation program expressed in $L_2$.
- $t_1$ and $t(t_1)$ are equivalent transformation programs.

If for two languages $L_1$ and $L_2$ we establish interoperability in both directions, we have bidirectional interoperability between them. The challenge is to identify the transformation program $t$ required by Definition 2. To achieve a translation between a pair of languages, we have to establish correspondences between their language constructs. Sometimes this is straightforward, at other times difficult, and in some cases impossible. We need a framework for reasoning the possibility of having such a transformation $t$ before trying to implement it. Such a framework would help us to identify cases where defining $t$ is difficult or impossible.

During the analysis of possible language translations, we found it useful to compare language features according to different categories. An example of such a category is the programming paradigm employed by a language: imperative, declarative, and hybrid. In general, it is easier to translate from a declarative to an imperative language than vice versa. Another example is the pattern matching language. Some languages use simple patterns, whereas others use complex OCL expressions [23]. Translation in this case may be impossible. The conclusion is that we immediately may rule out some interoperability options just on the basis of a high-level reasoning on the language features.

This observation serves as the basis for our approach. We select a set of categories and classify every analyzed language along these categories. A comparison table is built as a result. It is the major input for the framework that we propose for the identification of interoperability options and obstacles.

Most categories are derived from the work of Czarnecki and Helsen [8] and Gardner et al. [11]. Whenever necessary, we introduce other categories. The categories are enumerated and explained in the following list.

- **Transformation scenarios.** We consider the following scenarios directly derived from the analyzed languages:
  - Model synchronization. In this scenario, two existing models are synchronized according to a given set of relations. Whenever necessary changes are made in the models.
  - Conformance checking. In this scenario, two models are checked if they satisfy a set of relations. No changes are made to the models.
Model transformation. In this scenario, a set of output models is produced from a set of input models.

In-place update. This is a special case of transformation in which the source and the target models are the same.

- **Paradigm.** We recognize imperative, hybrid, and declarative paradigms for defining transformations.
- **Directionality.** This category indicates the direction in which a transformation definition may be executed. We distinguish between unidirectional and multidirectional transformations;
- **Cardinality.** Cardinality indicates the number of input and output models for a transformation definition;
- **Traceability.** Traceability links keep a record of correspondences between source and target elements during (and eventually after) the execution of a transformation. In general, there are two ways to deal with traceability: automatic and user-specified. The automatic way is supported by the language constructs and the execution engine. In the user-specified approach, the software engineer is responsible for building and using the data structure for traceability.
- **Query language.** Generally, a transformation language provides a means to select elements from the source models. We refer to this means as a query language. It is not necessary that it be defined as a standalone language.
- **Rule scheduling.** This category concerns the order of rule application assuming that the transformation rule is the basic modular unit in transformation languages. There are two forms of rule scheduling: implicit and explicit. Implicit scheduling is based on the implicit dependencies among rules. It is usually found in declarative languages. Explicit scheduling uses an explicit specification of rule application order. It is of two kinds: internal and external. Explicit internal scheduling includes the control flow specification within transformation rules. In contrast, the explicit external scheduling separates the rules from the execution order specification. This form is often found in graph transformation languages.
- **Rule organization.** This category concerns the relationships among rules. Some examples are rule inheritance, rule packaging, and rule priority relations.
- **Reflection.** Some languages provide reflective capabilities. Generally, they are classified into introspection and intercession [20]. There are different forms of reflection observed in transformation languages. We elaborate on this subject later on, when describing the languages.

In this paper, languages are classified according to these categories. In order to identify possible translations among languages, we use several heuristic rules. It is a space consuming task to study all the possible pairs of languages. Since ATL is the language in which we have the best expertise, we analyze interoperability between ATL and the rest of the languages discussed here in greater detail. For a pair of languages, we compare various technical details. The goal is to identify potential obstacles and ultimately to judge the feasibility of the language interoperability.

Some identified problems may also be valid for the cases of translations not considered in the paper.

### 4. Description of transformation languages

This section presents the languages analyzed in the paper: QVT, ATL, Tefkat, VIATRA2, GReAT, and AGG. To establish a common terminology, these languages are discussed from the point of view of the model transformation pattern shown in Fig. 2. When a certain language uses its own specific terminology, we establish a correspondence with the concepts in the transformation pattern.

Tab is a transformation program in which execution results in the automatic creation of model \( M_b \) from model \( M_a \). These three entities are all models conforming to \( M_Mt, M_Mb, \) and \( M_Ma \) metamodels respectively. \( M_Mt \) corresponds to the abstract syntax of the transformation language. The metamodels are expressed in metalanguage called \( M_MM \). In the figure, one target model is created from one source model. We will assume a more general pattern in which multiple source and target models may be used.

#### 4.1. QVT

The QVT RFP called for a language capable of expressing queries, views, and transformations over models in the context of MOF 2.0 metamodeling architecture. The QVT RFP was answered by several initial submissions that converged into a single proposal for the QVT standard. In this section, we give an overview of the language by focusing on two issues: the language architecture and the conformance points for QVT tools.
4.1.1. The QVT architecture

The operational context of QVT is shown in Fig. 3. It is based on the transformational pattern. QVT provides its own metamodel defining its abstract syntax.

The abstract syntax of QVT is defined as a MOF 2.0 metamodel. This metamodel defines three sublanguages for transforming models. OCL 2.0 is used for querying models. The creation of views on models is not addressed in the proposal.

The three QVT languages collectively form a hybrid transformation language with declarative and imperative constructs. The languages are named Relations, Core, and Operational Mappings. These languages are organized in a layered architecture shown in Fig. 4.
Table 1
QVT conformance points for tools

<table>
<thead>
<tr>
<th>Language Dimension</th>
<th>Interoperability Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Syntax Executable</td>
</tr>
<tr>
<td>Core</td>
<td></td>
</tr>
<tr>
<td>Relations</td>
<td></td>
</tr>
<tr>
<td>Operational Mappings</td>
<td></td>
</tr>
</tbody>
</table>

The languages *Relations* and *Core* are declarative languages at two different levels of abstraction. The specification document defines their concrete textual syntax and abstract syntax. In addition, *Relations* language has a graphical syntax. *Operational Mappings* is an imperative language that extends *Relations* and *Core* languages.

The *Relations* language provides capabilities for specifying transformations as a set of relations among models. Relations contain a set of object patterns. These patterns can be matched against existing model elements, instantiated to model elements in new models, and may be used to apply changes to existing models. The language handles the manipulation of traceability links automatically and hides the related details from the developer.

The *Core* language is a declarative language that is simpler than the *Relations* language. Transformation definitions written in it tends to be longer than the equivalent definitions written in the *Relations* language. Traceability links are treated as ordinary model elements. The developer is responsible for creating and using the links. One purpose of the *Core* language is to provide the basis for specifying the semantics of the *Relations* language. The semantics of the *Relations* language is given as a transformation *RelationsToCore*. This transformation may be written in the *Relations* language.

Sometimes it is difficult to provide a complete declarative solution to a given transformation problem. To address this issue, the QVT proposes two mechanisms for extending the declarative languages *Relations* and *Core*: a third language called *Operational Mappings*, and a mechanism for invoking transformation functionality implemented in an arbitrary language (*Black Box* implementation).

The *Operational Mappings* language extends the *Relations* language with imperative constructs and OCL constructs with side effects. The basic idea in this language is that the object patterns specified in the relations are instantiated by using imperative constructs. In that way, the declaratively specified relations are imperatively implemented in the language. The syntax of *Operational Mappings* language provides constructs commonly found in imperative languages (loops, conditions, etc.).

The *Black Box* mechanism allows the plugging-in and execution of external code during the transformation execution. This mechanism allows complex algorithms to be implemented in any programming language and enables reuse of already existing libraries. This makes some parts of the transformation opaque, which brings a potential danger, since the functionality is arbitrary and is not controlled by the transformation engine.

4.1.2. QVT conformance points

Fig. 4 does not suggest any particular implementation of a QVT transformation engine. Tool vendors may choose different strategies. For example, the *Core* language may be supported by an execution engine and the *Relations* transformations may be transformed to equivalent programs written in *Core* language. In that way, the engine is capable of executing programs written in both languages. Another possibility is that only the *Relations* and *Operational Mappings* are supported by a tool. In this case, the *Core* language serves simply as a reference point for specifying the semantics of the *Relations* language.

These implementation options may produce tools with different capabilities. To denote the capabilities of tools, the QVT proposal defines a set of *QVT conformance points* for tools. Conformance points are organized along two dimensions and form a grid with 12 cells. Table 1 shows the dimensions and the possible conformance points.

The Language Dimension defines three levels corresponding to the three QVT languages: *Core*, *Relations*, and *Operational Mappings*. If a tool conforms to a given level, this means that it is capable of executing transformation definitions written in the corresponding language.

The *Interoperability Dimension* is concerned with the form in which a transformation definition is expressed. It defines four levels:
Fig. 5. ATL layered architecture.

- **Syntax Executable.** A tool can read and execute transformation definitions written in the concrete syntax given in the QVT proposal;
- **XMI Executable.** A tool can read and execute transformation definitions serialized according to the XMI serialization rules (recall that transformation definitions conform to the QVT metamodel and therefore are XMI serializable);
- **Syntax Exportable.** A tool can export transformation definitions in the concrete syntax of the corresponding language;
- **XMI Exportable.** A tool can export transformation definitions in XMI format.

A requirement states that if a tool is SyntaxExecutable or XMIEexecutable for a given language level, it should also be SyntaxExportable or XMIEexportable respectively.

### 4.2. ATL

ATL operates in the same context as QVT, as shown in Fig. 3. Similarly to QVT, ATL provides a set of languages that form its architecture. ATL architecture is composed of three layers, as shown in Fig. 5. They are (in decreasing abstraction level order) the ATLAS Model Weaving (AMW) [10], ATL, and the ATL Virtual Machine (ATL VM).

ATL provides both declarative and imperative constructs, and is therefore a hybrid model transformation language. Compiled ATL programs are executed by ATL VM, which uses a model-oriented instruction set. AMW may optionally be used as a higher abstraction level transformation specification language. These three layers are presented in the coming sections.

We start by presenting ATL, followed by an overview of the virtual machine, because they are both central components of ATL model transformation architecture. AMW is not limited to model transformation and can be used in other contexts. It is presented in the last subsection.

#### 4.2.1. ATLAS transformation language

The declarative part of ATL is based on the notion of matched rule. Such a rule consists of a source pattern matched over source models and of a target pattern that gets created in target models for every match. Traceability links are automatically created. Rule inheritance and polymorphic rule reference are available. Navigation is performed using OCL expressions.

ATL offers two imperative constructs: called rule and action block. A called rule is explicitly called, like a procedure, but its body may be composed of a declarative target pattern. Matched rules and called rules may be used together in a single transformation program. Action blocks are sequences of imperative instructions that can be used in either matched or called rules.

Transformation programs written in ATL are inherently unidirectional. Source models, which are only navigable (i.e., read-only), and target models, which are not navigable (i.e. write-only), are clearly identified at development time.

There are two modes in which the declarative part of an ATL program can operate: standard and refining. In the standard mode, elements are only created when a rule is matched. However, since models cannot be transformed in-place (source models are read-only), transformations that only modify small parts of a model and leave most of the rest unchanged are too complex to write in this mode. There must be roughly at least one copy rule for each type
 declared in the metamodel. This is not required in the refining mode, where unmatched elements are automatically copied by the engine.

An execution engine and development tools (code editor, compiler, debugger, etc.) are available on the GMT project web site [9].

4.2.2. Execution support: ATL VM

We chose to base the current ATL execution engine on a virtual machine architecture, as shown in Fig. 6. The VM is implemented on top of two model handlers (i.e., libraries dealing with models): the Eclipse Modeling Framework (EMF) [7] and Netbeans MetaData Repository (MDR) [21]. The VM could also be based on other model handlers, as suggested by the “etc.” box in Fig. 6. The ATL compiler works on top of the ATL VM and generates ATL programs capable of running on top of it too.

There are several advantages of this approach:

– Extending ATL mostly requires changes in the compiler only;
– Adding the ability to deal with new kinds of models only involves changes in the virtual machine;
– Compiling other languages to ATL bytecode and having them benefit from the same virtual machine tools such as the debugger and model handler drivers (e.g., MDR, EMF).

The ATL VM language is a small imperative instruction set composed of four categories of bytecodes: stack, memory, control flow, and model handling. There are currently only 21 different instructions. VM operations are especially adapted to OCL helpers implementation (see [23], Section 7.4.4).

There is also specific support for traceability, but it must be used explicitly. Traceability links are indeed not automatically created like they are in ATL. A draft specification of ATL VM is available on GMT [9].

4.2.3. ATLAS model weaver

We saw that the ATL virtual machine provides a basic set of constructs, which are sufficient to perform automatic operations on models. Although ATL provides a higher level language for transformation definition, it is sometimes necessary to express transformations in even more abstract terms. AMW provides solutions to this issue.

We believe application-independent transformation language abstract syntaxes are not enough for some applications. Some problem domains indeed require specific transformation concepts that cannot always match a general purpose syntax. Ideally, the modular structure of a software system should be compatible with the model of the problem domain for which the system is built. In the context of model transformations, this means that transformation abstract syntax should match problem domain transformation concepts.

It should first be noted that model weaving [10] is different from aspect weaving [18]. Model weaving is about establishing typed links between model elements. Links themselves form a model, and link types are therefore defined in a metamodel. Weaving links are more abstract than ATL rules because, whereas ATL and the ATL VM have fixed semantics, AMW has a user-defined semantics. Consequently, link types can be adapted to specific application domains. This provides a framework for defining solution domain concepts that match those of a given problem domain.

For instance, in [1] we presented a concrete use of AMW to represent and use mappings between metamodels and UML profiles. Several link types are used to: map metamodel classes and profile elements (i.e., UML classes and stereotypes), map their respective properties (attributes and tagged values), identify cases in which multiple elements on one side correspond to a single element on the other side, etc. These link types are specific to the problem of
mapping metamodels and UML profiles. As a consequence, the actual mapping descriptions can be relatively simple compared to their ATL equivalents. We also showed in [1] that such AMW mapping descriptions can be executed. To this end, they are transformed into ATL programs, translating between metamodel-based and profile-based models.

The adaptability of AMW to different problem domains is achieved by providing tools working on a core weaving metamodel defining only the abstract notion of link type. This core metamodel can be extended by users. Metamodel extension is a complex operation, which will not be discussed here. The basic idea is that user-defined link types have to extend the abstract link type concept defined in the core. Transformations to ATL code can then be used to implement link types’ semantics.

Furthermore, weaving semantics need not even be executable. This is because the AMW application domain is actually broader than the transformation specification. This is, however, out of the scope of this paper. More information may be found in [10]. An implementation of AMW is available on the GMT web site [9].

4.3. Tefkat

Tefkat is a declarative transformation language based on the DSTC submission to the QVT RFP. Its operational context is the same as the context of QVT and ATL. The transformation engine relies on ECore (a part of EMF) as a metamodeling language.

Tefkat transformations operate on extents. Extents are sets of model elements that represent the input and output models and the traceability relations. There are three types of extents: source, target, and tracking. Source extents are built from the source models of a transformation. They may only be queried by the transformation rules (i.e., source extents are read-only). Target extents are populated with model elements during the transformation execution. They are write-only. The tracking extent is a special type of extent that may be both queried and populated by the transformation rules. Its purpose is to maintain links between elements in the source and target extents. The elements in the tracking extents are instances of the tracking classes defined in transformations.

Declarative unidirectional transformation rules are the basic modular units in Tefkat. They have two parts formed by patterns. Patterns may be defined separately and used in multiple rules by name reference. Patterns may refer to each other and may form recursions. Patterns are either matched over the input extents and thus select elements on the base of certain constraints, or are used to create or to ensure the existence of model elements in the target extents. Transformational rules are loosely coupled: they do not refer to each other directly. A rule may use the results produced by other rules by querying the tracking extent. A tracking class may be used by multiple transformation rules.

Since source extents are read-only, the in-place update transformation scenario is not supported. Tefkat transformations are unidirectional. The language and the engine supports a scenario in which no target extent is created but only constraints checking over the source extents is performed.

Tefkat supports reflection. It has three forms. The first form is a generic access to the properties and metaclasses of the model elements. Tefkat relies on the ECore reflective API to do that (recall that the operational environment of the language is based on the ECore metamodeling architecture). The second form allows specifying expressions in places where a class or a feature is expected. The third form uses the construct AnyType that allows any object to be selected regardless its concrete type.

These three reflective constructs allow model copy transformations to be specified in a generic and concise manner. As we can see, the reflective support in Tefkat concerns only model elements. We are not aware of reflective capabilities that allow navigation over the transformation rules or changing the behavior of the transformation system at runtime.

The Tefkat engine may be used as an Eclipse plugin. Its execution algorithm resembles Prolog execution. This is in accordance with the logic-based theoretical underpinnings of the language.

4.4. Graph transformation languages

This section gives a general overview of graph transformation approaches and presents three graph transformation languages: VIATRA2, GReAT, and AGG.
Graph transformation languages describe transformations that operate on a graph by rewriting it. A transformation is performed in steps operating on a current graph. This current graph is initially the source graph that evolves after each step. When no rule can be applied any more, the current graph is considered as the result of the transformation. A step corresponds to the application of a rule.

Each rule is composed of a left-hand side to be matched in the current graph and of a right-hand side defining the changes to be perform. These changes are deletion of matched elements and the creation of new elements in the graph. More than one rule is generally applicable on the current graph. The choice of the rule to apply can be performed non-deterministically or be driven by priorities, the control flow, or an explicit rule scheduling. There may also be additional conditions to the application of a rule.

Since models may be represented as graphs, graph transformation languages are suitable for model transformation. Some of them do not use metamodels defined in MOF. Furthermore, during their evolution, parts of the current graph conform to the source metamodel, while other parts conform to the target metamodel. Often, traceability links between source and target elements are represented by elements of the current graph. This means that the current graph conforms to a common metamodel composed of the source and target metamodels, as well as of the additional elements used to link target elements to source elements.

Graph transformation rules can often be represented using a graphical notation. Some languages also offer a textual notation. Some parts of the rules (e.g., guards) may be written in a textual language.

VIATRA2 is a unidirectional transformation language based mainly on graph transformation techniques. In contrast to the languages described so far, the operational context of VIATRA2 is not based on MOF/ECore architecture. The language operates on models expressed following the VPM metamodeling approach. In this approach, an indefinite number of metalevels may exist and the instanceof relations are explicitly represented. This approach is more expressive than the metamodeling approaches that stem from MOF (e.g., ECore and KM3). It is, however, possible to represent the three-layered MOF architecture within VPM, thus enabling the execution of MOF-based transformation scenarios.

VIATRA2 integrates three sublanguages that form a coherent whole in contrast to QVT, where the three sublanguages may be used in a standalone manner. The languages are:

- **Graph pattern language.** This language is used to express patterns that are matched to select elements in the current graph. Patterns may reuse other patterns and may form recursions;
- **Graph transformation rules language.** This is a language for expressing graph rewriting rules. Every rule has a left and a right hand side which is a pattern. Rules are unidirectional. Following the classical scheme of graph transformations, a rule may delete some of the matched elements, may create new elements, and may preserve existing elements;
- **Abstract State Machine (ASM) language.** There is no predefined order of execution of the transformation rules. The order is specified by using ASM constructs. The ASM language provides a set of control flow structures: sequencing, rule call, conditionals, fixed-point iteration, etc. These constructs are used in ASM rules similar to methods in OO languages;

We may perceive VIATRA2 as a hybrid language. The transformation rules language is declarative, but the rules cannot be executed without an execution strategy specified in an imperative manner. The language also allows code generation by using code templates. This is out of the scope of this paper, since we focus only on model-to-model transformations.

It is possible to define generic template rules in VIATRA2. In these rules, the classes are parameters that may be substituted via template instantiation. The instantiation is achieved by a meta-transformation in the terms of VIATRA2. In fact, the instantiation executes a higher-order transformation (HOT) which manipulates the generic transformation.

The execution environment provides both an interpreter and a compiler. Transformation programs may be interpreted for the purposes of testing and simulation. This is based on the execution mechanism of ASMs. In addition, transformations may be compiled to a given target environment. It is possible to invoke native functions implemented in Java. This is similar to the Black Box mechanism in QVT.
4.4.3. GReAT

GReAT [17,2] (Graph Rewriting and Transformation Language) enables the specification of unidirectional translations between sets of models. These models conform to metamodels specified in UML, and can be created with the Generic Modeling Environment (GME). Each metamodel is called a domain, and traceability information conforms to a user-specified cross-domain metamodel. This metamodel may also be used to extend source or target domains with transformation-specific elements (e.g., additional classes). The current graph is called the host graph.

GReAT mainly uses a graphical notation. However, some parts are specified textually: attribute initialization expressions and guards. GReAT is composed of three sublanguages:

- **Pattern specification language.** A pattern is a graph that is recognized by pattern matching over the host graph. The GReAT pattern matcher does not work on the whole host graph but starts from a set of already matched elements, which is more efficient. The corresponding algorithm only works on connected graphs. The pattern specification language supports cardinalities and the grouping of sub-patterns.

- **Graph transformation language.** Transformation rules are specified in the graph transformation language, which is an extension of the pattern specification language. Each element of a pattern is associated to a role depending on the existence of the element before and after the application of the rule. The bind role corresponds to elements existing before and after. The delete role is used for elements that exist before, but not after (i.e., they are deleted). The new role specifies elements that exist only after (i.e., they are created). Guards can be specified as Boolean C++ expressions. Attribute mappings written in C++ are used to initialize the attributes of new elements.

- **Control flow language.** Rule application order is specified using a dedicated imperative language, which provides iterating and conditional constructs. Moreover, partial matches (called packets) are passed from rule to rule in order to set the initial bindings. This implies a specific sequencing of execution.

Depending on how the rules and the control flow are specified, a given transformation may be non-deterministic (i.e., have different targets for the same source).

4.4.4. AGG

AGG is a development environment for attributed graph transformation systems supporting an algebraic approach to graph transformation [27,29]. This approach is based on strong formal foundations. The source, target, and common metamodels are represented by type graphs. Graphs may additionally be attributed using Java code. Rule application can be guarded by negative application conditions, which specify patterns that prevent rule execution if they are present in the current graph. For instance, this may be used to prevent multiple applications of a rule. In some cases, graph transformations can be automatically reversed to produce the inverse transformation [28].

Transformations can be executed in two modes. The first one is called interpretation mode and corresponds to the automatic application of rules. The rules are applied non-deterministically until none of them can be applied any more. If an explicit application order is required, rules can be grouped in ordered layers. The rules contained in a given layer are only applied until a fixed point is reached. In the second mode, a user explicitly fires each rule individually. We refer to this as interactive mode.

This approach heavily relies on pattern matching. Moreover, since there is no determinism in the choice of several applicable rules to apply, there is a strong need for confluence checking. A set of rules is confluent if any application order leads to a deterministic result. AGG can detect conflicts between rules that could lead to a non-deterministic result. This is called critical pair analysis, because it finds pairs of conflicting rules. Once all conflicts are solved (e.g., by using layers), rule application is still non-deterministic but the final result is always the same. AGG can also check termination criteria.

5. Comparing languages

The descriptions of the languages are used to derive a comparative table that puts together the major language features. It will be used for reasoning about the possibilities and obstacles in achieving language interoperability.

Table 2 shows a summary of the discussion in Section 4. The rows in the table are the categories and the subcategories for analyzing the transformation languages. The columns represent the languages. They are also shown in case of a language with a complex architecture consisting of several sublanguages. This is the case with ATL and QVT.
<table>
<thead>
<tr>
<th>Category</th>
<th>AMW</th>
<th>ATL</th>
<th>ATL VM</th>
<th>Tefkat</th>
<th>VIATRA2</th>
<th>GReAT</th>
<th>AGG</th>
<th>QVT</th>
<th>Relations</th>
<th>OM</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformation scenarios</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model synchronization</td>
<td>N/A</td>
<td>No (see note 1)</td>
<td>No (see note 1)</td>
<td>No (see note 1)</td>
<td>No (see note 1)</td>
<td>No (see note 1)</td>
<td>No (see note 1)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Conformance checking</td>
<td>N/A</td>
<td>No (see note 1)</td>
<td>No (see note 1)</td>
<td>No (see note 1)</td>
<td>No (see note 1)</td>
<td>No (see note 1)</td>
<td>No (see note 1)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Model transformation</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>In-place update</td>
<td>N/A</td>
<td>Yes (refining mode)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Interactive transformation</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Paradigm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Declarative</td>
<td>Yes</td>
<td>Yes (via ATL)</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Hybrid</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes (see note 2)</td>
<td>Yes (see note 2)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Imperative</td>
<td>No</td>
<td>Yes (via ATL)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Directionality</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unidirectional only</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Multidirectional</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Cardinality</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M-to-N</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes (see note 3)</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>1-to-1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (see note 3)</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Traceability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Automatic</td>
<td>N/A</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>User-specified only</td>
<td>N/A</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Query language</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N/A</td>
<td>Based on OCL</td>
<td>Stack-based</td>
<td>Object patterns, recursion allowed</td>
<td>Graph patterns, recursion allowed</td>
<td>Graph patterns, C++</td>
<td>Graph patterns</td>
<td>Object patterns</td>
<td>Based on OCL</td>
<td>Object patterns</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule scheduling</td>
<td>N/A</td>
<td>Implicit, Internal explicit</td>
<td>N/A</td>
<td>Implicit</td>
<td>External explicit</td>
<td>External explicit</td>
<td>External explicit, implicit (see note 4)</td>
<td>Implicit</td>
<td>Internal explicit</td>
<td>Implicit</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category</th>
<th>AMW</th>
<th>ATL</th>
<th>ATL VM</th>
<th>Tefkat</th>
<th>VIATRA2</th>
<th>GReAT</th>
<th>AGG</th>
<th>QVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule organization</td>
<td>N/A</td>
<td>Rule inheritance, libraries</td>
<td>N/A</td>
<td>Rule inheritance</td>
<td>Rule packaging</td>
<td>Hierarchy of blocks</td>
<td>Layering</td>
<td>Rule and transformation inheritance</td>
</tr>
<tr>
<td>Reflection</td>
<td>Access to model and metamodel elements</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Unknown</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Reflection</td>
<td>Type and Property Expressions in source and target patterns</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes (template rules)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Reflection</td>
<td>Runtime access to the transformation program</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
In each given cell, we indicate if the category in the row is supported by the language. Generally, the content of cells is either ‘Yes’ or ‘No’. Whenever the support cannot be judged on the base of the available language description, we put ‘Unknown’. In some cases the cells contain more detailed information, for example, the concrete constructs used for rule organization. There are several clarifications about the table given as notes and explanations.

Note (1): Conformance checking and model synchronization are natively supported only by QVT Relations and QVT Core languages. For the rest of the languages, we indicate no support. However, it may be possible to implement these scenarios by a separate transformation.

Note (2): VIATRA2 and GReAT have a declarative rule language and an imperative language for the rule application order. As a whole, we assume these languages to be hybrid.

Note (3): The QVT specification does not clarify if the cardinality M-to-N is supported in enforce mode. The available examples are M-to-1 transformations only.

Note (4): AGG provides external explicit scheduling via layering and implicit scheduling within a layer. Most of the categories are not applicable to AMW. This is indicated as ‘N/A’. For example, it is not possible to determine what transformation scenario a concrete specialization of the AMW model will implement.

The reflective access to model and model elements is supported in ATL and Operational Mappings thanks to the usage of OCL. We consider the template rules available in VIATRA2 as a form of reflection.

Regarding the programming paradigm, we distinguish between purely declarative and purely imperative forms ATL. They are indicated as ATL_D and ATL_I respectively. Indeed, the language allows both styles in their pure form. ATL also allows mixing declarative and imperative rules in a single transformation program (i.e., it is a hybrid language).

6. Achieving interoperability

As we established earlier, our goal is to translate transformation programs from the source language to transformation programs in the target language. In this section, we first present heuristics for the initial identification of the feasibility and potential obstacles for such translation. Then we analyze several possible translations and discuss them in more detail. At the end of the section, we draw some conclusions on the base of the explored examples, and extend the initial set of heuristics.

6.1. Heuristics for interoperability between transformation languages

Some of the categories in Table 2 represent mandatory features of transformation languages. These are transformation scenarios, paradigm, directionality, cardinality, traceability, and rule scheduling. When we design a translator from one language to another, we have to deal with potential incompatibilities between these features. We can derive heuristics that tell us when a translation between features is possible and when we can expect obstacles. These heuristics are discussed below per every mandatory feature. Of course, optional features are also important and must be taken into account in every concrete case.

– Transformation scenarios. In general, every transformation language supports at least a model-to-model transformation scenario. QVT supports two additional scenarios. Some languages support in-place updates. Implementing a scenario in a language that does not directly support it may be a problem. For example, conformance checking between models is supported only in QVT Relations and QVT Core languages. Intuitively, this scenario may be implemented as a transformation that takes two models as input and generates a report about the found inconsistencies. However, it is not clear how exactly the relations that must be satisfied are translated in the transformation rules in various languages. In order to simplify the discussion, we limit ourselves only to the model transformation scenario. The handling of the special scenarios in QVT is discussed later.

– Paradigm. The common knowledge in the area of programming languages suggests that translations from declarative to hybrid and from hybrid to imperative language are easier than translations in the opposite direction. In the context of model transformations, the declarative/imperative dichotomy is mostly demonstrated in the way the flow of control is expressed. Declarative languages usually have an implicit flow of control, whereas imperative ones have an explicit flow of control. From that point of view, the mentioned heuristic is valid also for model transformation languages. Indeed, it is difficult to induce declarative transformation rules in the style of ATL or Tefkat from an imperative program that may perform multiple passes over the source model and create and initialize
the target model elements in a certain order. This is exactly the aspect that remains hidden from the developer when
she is using declarative languages. We capture the heuristic in the following form:

\[ \text{Declarative} \rightarrow \text{Hybrid} \rightarrow \text{Imperative}. \]

The arrow indicates the expected “easy” or the more possible direction for translation. It should be noted that
examples may exist in which the direction is opposite and the translation is still possible.

- **Directionality.** A multidirectional transformation definition may be converted to a set of unidirectional definitions,
  but the opposite generally does not hold. There are many examples of transformations that are not reversible, and
  therefore they are pure unidirectional. Furthermore, considering only directionality, a unidirectional transformation
definition in one language should be translatable to an equivalent unidirectional definition in another language. Of
course, there are other language features that may cause problems. We formulate the following two heuristics:

\[ \text{Bidirectional} \rightarrow \text{Unidirectional} \]
\[ \text{Unidirectional} \rightarrow \text{Unidirectional}. \]

- **Cardinality.** The heuristics regarding cardinality are easily identified on the basis of comparing the number of
  allowed input and output models. The following holds:

\[ 1:1 \rightarrow M:1 \rightarrow M:N. \]

Obviously, a transformation that transforms three input models to one output model cannot be expressed in a
language that expresses only one-to-one transformations.

- **Traceability.** We distinguish two types of traceability: user specified only, and automatic. The presence of
  automatic traceability does not generally prevent the possibility of user specified traceability. The following
heuristics may be derived:

\[ \text{User specified} \rightarrow \text{User specified} \]
\[ \text{Automatic} \rightarrow \text{User specified} \]
\[ \text{User specified} \rightarrow \text{Automatic, but only if the user handling is compatible with the automatic one} \]
\[ \text{Automatic} \rightarrow \text{Automatic, but only if the two mechanisms are compatible}. \]

The last two rules hold only in specific cases.

- **Rule scheduling.** This affects the paradigm of the language and is not considered here.

We do not consider heuristics for the remaining language features. They are taken into account in the course of the
analysis of different language translations.

### 6.2. Studying interoperability between languages

In this section, we analyze pairs of languages and make conclusions about the problems in achieving
interoperability. We do not cover all the possible combinations between the languages presented here. Our focus
is mainly on achieving interoperability between ATL and other languages.

#### 6.2.1. ATL and QVT

These two languages have several sublanguages. Some combinations are more amenable to interoperability than
others. We consider some of the combinations in this section. A previous version of this work was presented in [16]
with discussions of a couple of other combinations. We only keep the most relevant ones here.

#### 6.2.1.1. ATL\textsubscript{D} and Operational Mappings (OM)

Table 3 compares the features of the languages and takes into
account the applicability of the heuristics described above.

We can see that only one potential problem exists according to the heuristics: both the declarative ATL and OM
support automatic traceability. The ATL traceability mechanism may be translated to the OM mechanism only if they
are compatible. This is indicated by a question mark. However, it is possible to implement the ATL traceability
manually in OM. In conclusion, the application of the heuristics indicates a high possibility for interoperability
between the two languages. Now we turn to an analysis of the other features of the languages and how they can
be translated to each other.

- **Expressing the ATL execution algorithm in OM.** Although we concluded that the translation from declarative
to imperative is possible, we still need to delve into details. Declarative transformations in ATL are executed in
two phases: in the first phase, the matching of rules is performed. In the second phase, the creation of the target
elements and their initialization is performed. It is possible to encode these phases with the available control flow
structures of OM.
Table 3
Possibilities for translation from declarative ATL to OM

<table>
<thead>
<tr>
<th>ATL\textsubscript{D}</th>
<th>OM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
<td>Declarative</td>
</tr>
<tr>
<td>Directionality</td>
<td>Unidirectional</td>
</tr>
<tr>
<td>Cardinality</td>
<td>M:N</td>
</tr>
<tr>
<td>Traceability</td>
<td>Automatic</td>
</tr>
<tr>
<td></td>
<td>Automatic</td>
</tr>
</tbody>
</table>

Table 4
Possibilities for translation from OM to declarative ATL

<table>
<thead>
<tr>
<th>OM</th>
<th>ATL\textsubscript{D}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
<td>Imperative</td>
</tr>
<tr>
<td>Directionality</td>
<td>Unidirectional</td>
</tr>
<tr>
<td>Cardinality</td>
<td>M:N</td>
</tr>
<tr>
<td>Traceability</td>
<td>Automatic</td>
</tr>
<tr>
<td>User specified</td>
<td>?</td>
</tr>
</tbody>
</table>

- **Query language.** The query languages are compatible to a large extent. ATL uses OCL expressions as guards and OCL helpers for navigation over the source model. The same constructs are available in OM. We can perform a direct translation of these constructs.

- **Rule scheduling.** This is already discussed in the first bullet regarding the ATL execution algorithm.

- **Rule organization.** ATL supports rule inheritance. It is different from the notion of rule inheritance in OM. Therefore, no direct translation is possible. In order to handle this problem, we may introduce an initial step that flattens the rule inheritance hierarchy. Translations into OM will be applied on ATL programs that do not contain rule inheritance.

- **Reflection.** Both languages support the same form of reflection based on OCL. Generally, a direct translation is possible.

This discussion revealed that all the incompatibilities between the languages can be resolved. We conclude that the interoperability in that case is feasible. Of course, without implementing the actual translation, we cannot claim 100% certainty.

The opposite translation is a kind of a negative example that illustrates the capabilities of the heuristics to indicate difficulties in translations. Table 4 shows the indications implied by the heuristics. We see that one translation is indicated as a potential obstacle and two others hold under certain conditions.

Let us analyze the indicated problem: the translation from an imperative to a declarative language might be impossible. Taking into account the semantics of both languages, we can conclude that in the general case it is not possible to find a translation that works for all possible OM programs. First, a given source element may be visited multiple times by multiple rules. This is only allowed in specific contexts in the declarative ATL. Second, arbitrary complex strategies for visiting the source model may be expressed in OM. On the contrary, the execution algorithm in ATL is fixed. Third, it is not possible to infer only declarative rules from the rules in OM that may contain input parameters, a feature not supported in declarative ATL. The mentioned reasons are related to the complexity of the control flow in OM, whereas it is completely hidden from the developer in the declarative ATL. This is a consequence of the different programming paradigms employed in the languages.

6.2.1.2. **ATL\textsubscript{I} and Operational Mappings (OM).** Tables 5 and 6 respectively compare the features of the languages for the translations ATL\textsubscript{I} to OM and OM to ATL\textsubscript{I} according to the heuristics described above.

We can see that in both cases, no obstacle is identified. The only unclear point is the translation of user-specified traceability in ATL\textsubscript{I} to automatic traceability in OM. However, since user-specified traceability is available in OM, this issue is not very relevant. The issues of execution algorithm, rule scheduling, and rule organization do not cause any problems here. This is generally to be expected between imperative languages. Moreover, since both languages use OCL to navigate models, there should not be any issue in translating expressions from one to the other.
### Table 5
Possibilities for translation from imperative ATL to OM

<table>
<thead>
<tr>
<th></th>
<th>ATL_I</th>
<th>→</th>
<th>OM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
<td>Imperative</td>
<td>✓</td>
<td>Imperative</td>
</tr>
<tr>
<td>Directionality</td>
<td>Unidirectional</td>
<td>✓</td>
<td>Unidirectional</td>
</tr>
<tr>
<td>Cardinality</td>
<td>M:N</td>
<td>✓</td>
<td>M:N</td>
</tr>
<tr>
<td>Traceability</td>
<td>User-specified</td>
<td>✓</td>
<td>User-specified</td>
</tr>
</tbody>
</table>

### Table 6
Possibilities for translation from OM to imperative ATL

<table>
<thead>
<tr>
<th></th>
<th>OM</th>
<th>→</th>
<th>ATL_I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
<td>Imperative</td>
<td>✓</td>
<td>Imperative</td>
</tr>
<tr>
<td>Directionality</td>
<td>Unidirectional</td>
<td>✓</td>
<td>Unidirectional</td>
</tr>
<tr>
<td>Cardinality</td>
<td>M:N</td>
<td>✓</td>
<td>M:N</td>
</tr>
<tr>
<td>Traceability</td>
<td>Automatic</td>
<td>✓</td>
<td>User-specified</td>
</tr>
<tr>
<td></td>
<td>User-specified</td>
<td>✓</td>
<td>User-specified</td>
</tr>
</tbody>
</table>

### Table 7
Possibilities for translation from Core to imperative ATL

<table>
<thead>
<tr>
<th></th>
<th>Core</th>
<th>→</th>
<th>ATL_I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
<td>Declarative</td>
<td>✓</td>
<td>Imperative</td>
</tr>
<tr>
<td>Directionality</td>
<td>Multidirectional</td>
<td>✓</td>
<td>Unidirectional</td>
</tr>
<tr>
<td>Cardinality</td>
<td>M:N</td>
<td>✓</td>
<td>M:N</td>
</tr>
<tr>
<td>Traceability</td>
<td>User-specified</td>
<td>✓</td>
<td>User-specified</td>
</tr>
</tbody>
</table>

### Table 8
Possibilities for translation from Relations to imperative ATL

<table>
<thead>
<tr>
<th></th>
<th>Relations</th>
<th>→</th>
<th>ATL_I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
<td>Declarative</td>
<td>✓</td>
<td>Imperative</td>
</tr>
<tr>
<td>Directionality</td>
<td>Multidirectional</td>
<td>✓</td>
<td>Unidirectional</td>
</tr>
<tr>
<td>Cardinality</td>
<td>M:N</td>
<td>✓</td>
<td>M:N</td>
</tr>
<tr>
<td>Traceability</td>
<td>Automatic</td>
<td>✓</td>
<td>User-specified</td>
</tr>
</tbody>
</table>

#### 6.2.1.3. ATL_I, Relations, and Core
We do not consider translations from imperative ATL to Relations and Core, because this corresponds to the imperative to declarative case. Tables 7 and 8 respectively compare these languages for the translations Core to ATL_I and Relations to ATL_I.

These scenarios do not expose obstacles. All the advanced features of Core and Relations such as multidirectionality, automatic traceability, special transformation scenarios, etc. can be translated into imperative code. Similarly to the scenario considered in the previous section, the execution algorithm, rule scheduling, rule organization, and navigation language are not problematic here. This situation is not surprising, because it corresponds to the classical scenario of implementing a declarative language using an imperative language.

#### 6.2.2. ATL and Tefkat
We consider separately the cases of translations between declarative ATL and Tefkat, and imperative ATL and Tefkat. The drawing of conclusions about translating hybrid ATL programs is based on the two other cases. It is not presented here.

#### 6.2.2.1. ATL_I and Tefkat
For the translation from declarative ATL to Tefkat, we build Table 9.

The table indicates the applicability of all the heuristic rules. However, we need further analysis regarding the other language features.
Table 9
Possibilities for translation from declarative ATL to Tefkat

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>ATL → Tefkat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declarative</td>
<td>✓/ ✓</td>
</tr>
<tr>
<td>Directionality</td>
<td>Unidirectional ✓/ ✓</td>
</tr>
<tr>
<td>Cardinality</td>
<td>M:N ✓/ ✓</td>
</tr>
<tr>
<td>Traceability</td>
<td>Automatic ✓/ ?</td>
</tr>
</tbody>
</table>

Table 10
Possibilities for translation from Tefkat to declarative ATL

<table>
<thead>
<tr>
<th>Tefkat → ATL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
</tr>
<tr>
<td>Directionality</td>
</tr>
<tr>
<td>Cardinality</td>
</tr>
<tr>
<td>Traceability</td>
</tr>
</tbody>
</table>

- **Translating ATL rules to Tefkat rules.** ATL matched rules and Tefkat rules have similar structures. Both are declarative and do not include rule references and explicit flow of control. Target patterns can be directly translated. Source patterns, however, pose problems due to the differences in the query languages.

- **Query language.** ATL uses OCL expressions as guards and OCL helpers for navigation over the source model. Tefkat does not use OCL expressions and may use recursive patterns. The main problem here is the translation of OCL guards and helpers into Tefkat patterns. We cannot guarantee that this is possible in all the cases. Therefore, the difference in the querying languages is a major obstacle to achieving interoperability.

- **Rule organization.** ATL and Tefkat support similar mechanisms of rule inheritance. In any case, we can avoid translating the inheritance by flattening the transformation rules and eliminating inheritance.

- **Reflection.** ATL uses OCL to support reflection. It is similar to the reflective capabilities in Tefkat inherited from ECore. We do not expect problems here.

The discussion revealed that although all the heuristics indicated high chances for interoperability, this may not be true in the case of complex OCL expressions in the form of ATL rules. Differences in query languages are important and must be taken into account when interoperability is discussed.

Table 10 shows the details of the translation in the opposite direction: from Tefkat to declarative ATL.

The potential problem is handling the user specified traceability in Tefkat that may not always be compatible with the internal ATL mechanism. The ATL traceability records are tuples of source elements, rule names, and the created target elements. If the user defines another structure in the Tefkat tracking classes, then the translation may be impossible.

Below we discuss two relevant language features when translating from Tefkat to declarative ATL.

- **Query language.** The navigation used in Tefkat patterns is simple and may be translated in OCL expressions. However, Tefkat also uses recursive patterns: a feature not supported in ATL. The closest construct in ATL is the recursive helper. We do not have direct experience in translating Tefkat recursive patterns into ATL helpers. This translation is a possible obstacle for their interoperability.

- **Reflection.** Tefkat provides the three forms of reflection described earlier. The first two are available in ATL by using OCL expressions. Type and property expressions are not available in ATL.

In summary, we identified three obstacles for translating from Tefkat to declarative ATL: tracking classes not compatible with ATL traceability structures, recursive Tefkat patterns, and reflections based on type and property expressions in Tefkat.

6.2.2.2. **ATL I and Tefkat.** The analysis of the translation between imperative ATL and Tefkat is simpler than in the case of declarative ATL. Tables 11 and 12 illustrate the possibilities for both directions. In the case of translation from imperative ATL to Tefkat, we go from an imperative to declarative language. The remarks about the translation from OM to declarative ATL are also valid for this case. Generally, we rule out the possibility of translation.
Table 11
Possibilities for translation from imperative ATL to Tefkat

<table>
<thead>
<tr>
<th></th>
<th>ATL</th>
<th>→</th>
<th>Tefkat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
<td>Imperative</td>
<td>√</td>
<td>Declarative</td>
</tr>
<tr>
<td>Directionality</td>
<td>Unidirectional</td>
<td>√</td>
<td>Unidirectional</td>
</tr>
<tr>
<td>Cardinality</td>
<td>M:N</td>
<td>√</td>
<td>M:N</td>
</tr>
<tr>
<td>Traceability</td>
<td>User specified</td>
<td>√</td>
<td>User specified</td>
</tr>
</tbody>
</table>

Table 12
Possibilities for translation from Tefkat to imperative ATL

<table>
<thead>
<tr>
<th></th>
<th>Tefkat</th>
<th>→</th>
<th>ATL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
<td>Declarative</td>
<td>√</td>
<td>Imperative</td>
</tr>
<tr>
<td>Directionality</td>
<td>Unidirectional</td>
<td>√</td>
<td>Unidirectional</td>
</tr>
<tr>
<td>Cardinality</td>
<td>M:N</td>
<td>√</td>
<td>M:N</td>
</tr>
<tr>
<td>Traceability</td>
<td>User specified</td>
<td>√</td>
<td>User specified</td>
</tr>
</tbody>
</table>

Table 13
Possibilities for translation from declarative ATL to GT

<table>
<thead>
<tr>
<th></th>
<th>ATL</th>
<th>→</th>
<th>GT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
<td>Declarative</td>
<td>√</td>
<td>Hybrid</td>
</tr>
<tr>
<td>Directionality</td>
<td>Unidirectional</td>
<td>√</td>
<td>Unidirectional</td>
</tr>
<tr>
<td>Cardinality</td>
<td>M:N</td>
<td>√</td>
<td>M:N</td>
</tr>
<tr>
<td>Traceability</td>
<td>Automatic</td>
<td>√</td>
<td>User specified</td>
</tr>
</tbody>
</table>

The translation from Tefkat to imperative ATL does not expose the obstacles we found in the case of translating Tefkat to declarative ATL. For instance, both recursive Tefkat patterns and reflections based on type and property expressions can be translated to imperative code.

6.2.3. ATL and graph transformation languages

Instead of comparing ATL with every one of the three graph transformation languages (VIATRA2, GReAT, and AGG), we consider the common features of graph transformation approaches and reason about them. We make the following assumptions:

– A graph transformation language is hybrid. It provides declarative rules and imperative description of the control flow;
– A graph transformation language is unidirectional with M:N cardinality;
– A graph transformation language requires user specified traceability.

We refer to a graph transformation language as GT.

6.2.3.1. ATL_D and GT. We have the Tables 13 and 14 for both directions of translation between declarative ATL and GT below.

For the translation from declarative ATL to GT, the table indicates no major problems. However, the following incompatibilities may be obstacles:

– **Expressing the ATL execution algorithm in GT.** Implementation of the ATL execution algorithm with the imperative control flow language of GT may be a problem. However, the execution algorithm is relatively simple and the constructs available, for example, in VIATRA2 and GReAT should be enough. Another problem is the clear separation between read-only source model and write-only target model in ATL. In GT, the rules operate on a rewrite manner.

– **Query language.** The incompatibilities in querying languages pose more serious problems, as we already discussed in the section about ATL and Tefkat. It may be impossible to translate some OCL expressions to the equivalent graph patterns. In addition, in GReAT C++ expressions are used to initialize attribute values. This requires translation from OCL to C++;


Table 14 shows a problem in translation from GT to declarative ATL. This problem is caused by differences in their programming paradigms. Also, the user specified traceability mechanisms may be incompatible with the ATL automatic mechanism. Another problem is the translation of recursive and negative graph patterns to ATL patterns and OCL guards.

6.2.3.2. ATL and GT. Translating imperative ATL to GT (Table 15) again has the problem of translating OCL expressions. Furthermore, it should be possible to translate the ATL imperative constructs to the imperative constructs of GT. Finally, the called rules in ATL must be translated to the declarative graph rewriting rules. In the general case, this is not always possible.

The opposite direction of translation from GT to imperative ATL seems less problematic (Table 16). However, all the potential obstacles regarding the translation of graph patterns and flow of control constructs are valid here. For example, in case of VIATRA2, we have to translate the ASM constructs to the ATL flow of control structures.

6.3. Summary on interoperability between languages

We covered some translation scenarios between ATL and at least one of each type of the other languages. The results are summarized in Table 17.

The table shows that the translation from declarative ATL to Operational Mappings is the least problematic, since we did not identify any obvious obstacles. We may expect the same situation in translating imperative ATL to Operational Mappings, since these two languages are even more similar. It should be again noted that the discussion does not guarantee the 100% achievability of interoperability. Implementing the actual translation is the ultimate test.

On the basis of heuristics, we managed to identify many potential obstacles. In the course of the discussion, however, it became clear that these heuristics are not enough. Even in the cases when all the heuristic rules hold, there may be some problems.

An important obstacle is the incompatibility between query languages. Languages based on graph patterns are significantly different from languages based on OCL.

The second important obstacle is rooted in the possible incompatibilities in the imperative constructs of two languages. Although the widely used constructs like conditionals and iterations are easy to handle, there are special

<table>
<thead>
<tr>
<th>Table 14</th>
<th>Possibilities for translation from GT to declarative ATL</th>
</tr>
</thead>
<tbody>
<tr>
<td>GT</td>
<td>( \rightarrow )</td>
</tr>
<tr>
<td>Paradigm</td>
<td>Hybrid</td>
</tr>
<tr>
<td>Directionality</td>
<td>Unidirectional</td>
</tr>
<tr>
<td>Cardinality</td>
<td>M:N</td>
</tr>
<tr>
<td>Traceability</td>
<td>User specified</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 15</th>
<th>Possibilities for translation from imperative ATL to GT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATL_I</td>
<td>( \rightarrow )</td>
</tr>
<tr>
<td>Paradigm</td>
<td>Imperative</td>
</tr>
<tr>
<td>Directionality</td>
<td>Unidirectional</td>
</tr>
<tr>
<td>Cardinality</td>
<td>M:N</td>
</tr>
<tr>
<td>Traceability</td>
<td>User specified</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 16</th>
<th>Possibilities for translation from GT to imperative ATL</th>
</tr>
</thead>
<tbody>
<tr>
<td>GT</td>
<td>( \rightarrow )</td>
</tr>
<tr>
<td>Paradigm</td>
<td>Hybrid</td>
</tr>
<tr>
<td>Directionality</td>
<td>Unidirectional</td>
</tr>
<tr>
<td>Cardinality</td>
<td>M:N</td>
</tr>
<tr>
<td>Traceability</td>
<td>User specified</td>
</tr>
</tbody>
</table>
Table 17
Summary of the translation scenarios

<table>
<thead>
<tr>
<th>Source Language</th>
<th>Target Language</th>
<th>Identified obstacles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ATL\textsubscript{D}</td>
<td>OM</td>
<td>– Paradigm difference</td>
</tr>
<tr>
<td>2. OM</td>
<td>ATL\textsubscript{D}</td>
<td>– Translating OCL navigation and guards expressions to Tefkat patterns</td>
</tr>
</tbody>
</table>
| 3. ATL\textsubscript{D} | Tefkat | – Translation of Tefkat recursive patterns  
– Tefkat reflection  
– Track records translation |
| 4. Tefkat | ATL\textsubscript{D} | – Translation of Tefkat recursive patterns  
– Tefkat reflection  
– Track records translation |
| 5. ATL\textsubscript{I} | Tefkat | – Paradigm difference |
| 6. Tefkat | ATL\textsubscript{I} | – Translation of Tefkat recursive patterns  
– Tefkat reflection |
| 7. ATL\textsubscript{D} | GT | – Translation of OCL expressions to graph patterns |
| 8. GT | ATL\textsubscript{D} | – Paradigm difference |
| 9. ATL\textsubscript{I} | GT | – Translation of OCL expressions to graph patterns  
– Translating ATL imperative constructs to the constructs in GT |
| 10. GT | ATL\textsubscript{I} | – Translating GT imperative constructs to the constructs in ATL  
– Translating graph patterns to ATL queries based on OCL |

Finally, the third major obstacle that we encountered is related to reflection. Fortunately, this is a feature that is not available in the majority of the languages. Furthermore, it seems that it is useful in a limited set of scenarios.

The three obstacles mentioned above should be explicitly included as a part of the heuristics and guidelines that software engineers must take into account when designing a translator from one transformation language to another.

We did not study all the possible translations. For example, it is interesting to study the interoperability between graph transformation languages and QVT. Tefkat and graph transformation languages share many common features. A study on how the semantics of the earlier version of Tefkat may be based on graph transformations is presented in [26].

7. Conclusions

In this paper, we explored the problem of interoperability between model-to-model transformation languages. We outlined several possibilities and focused on the approach based on language translations. We proposed an initial set of heuristic rules to reason about the feasibility and potential obstacles for a given translation before trying to implement it. In the course of the discussion we found that, although this initial set of heuristics provides valuable insight, it is not sufficient. Other considerations should be taken into account, based on the identified classes of obstacles.

Although model-to-model transformation languages have a common and well-defined problem domain, the analysis showed that achieving a high degree of interoperability based on language translation in the general case is difficult. This is due to the variability and incompatibilities that the languages may expose. A major obstacle is the difference in the programming paradigms: declarative, hybrid, and imperative. Translating an imperative program to an equivalent declarative one is in some cases impossible. This is a general observation valid for any programming language. Apart from that, there are other issues specific to the domain of model transformations. An important obstacle is the difference in the query languages. Transformation languages based on the OMG standards usually
rely on OCL. Languages based on the graph transformations paradigm employ graph patterns. Bridging these two approaches is an open issue. For the moment, we can say that it is not clear how OCL-based queries are translated to graph patterns and vice versa. This remains an open issue for future research.

We managed to identify possibilities for interoperability at the conceptual level between languages that share common features. For example, we did not find major obstacles for translating from ATL to QVT Operational Mappings and from Operational Mappings to imperative ATL. This means that once such a translation is implemented, we may claim the QVT compliance of the ATL development tools.

Most of the explored translations exposed some obstacles. This does not mean that interoperability is not achievable at all. It is possible to perform translation for a limited set of source programs that satisfy certain constraints.

The presence of these obstacles motivates us to explore further the second major interoperability option: finding a lower level language that is a target for compiling model transformation languages (see Fig. 1(b)). Obvious benefits are the portability, reusability of functionality implemented in different languages, and of course, interoperability. Before taking such an approach, a careful analysis of the requirements for such a common intermediate language should be performed. Our work on the ATL virtual machine could be used as a starting point for this purpose.

The feasible translations need validation by implementing them, ultimately. This is a resource consuming task that is further complicated by the possible large number of language pairs. This is the reason why, in the paper, we limited ourselves to reasoning on the base of heuristics instead of trying to implement the actual translations. The required next step is an in-depth analysis of the translations indicated as feasible by our framework.

Acknowledgements

This work is partially supported by the IST European MODELPLEX project (MODELing solution for comPLEX software systems, FP6-IP 34081), and the OpenEmbeDD project. We would like to thank the members of the ATLAS Group and the anonymous reviewers for their useful feedback on the ideas presented in the paper.

References