Preface

Model transformation is an essential operation in Model-Driven Engineering. It may be used to generate lower-level code from higher-level models, but its application scope has now gone beyond Model-Driven Development scenarios. The 1st International Workshop on Model Transformation with ATL aimed at showing new applications for model transformation, as well as at improving the technology. The papers published in these proceedings and the presentations given during the workshop push back the boundaries of Model-Driven Engineering, and more particularly of model transformation with ATL. Some of these works present original applications of model transformation to solve modeling problems like measuring or checking models, testing model transformations, but also to solve problems in other technologies (e.g., providing interoperability between non-modeling tools). Other works look at how ATL may be improved with new or improved features like: traceability, transformation chaining, QVT compatibility, and direct support for UML profiles in order to support even more advanced scenarios. The workshop started with a guest talk on the definition of the meaning of models using model transformation. This talk provided an enriching introduction, which concluded with a challenge-oriented view of the workshop program. Finally, I thank all who made this workshop possible: the program and organization committees, the reviewers, our guest speaker, and last but not least the authors.

July 2009

Frédéric Jouault
Organization

Mt.ATL 2009 was organized by the AtlanMod INRIA & EMN joint team, Nantes, France, and colocated with RMLL 2009 (10th Libre Software Meeting).

Program Chair
Frédéric Jouault AtlanMod (INRIA & Ecole des Mines de Nantes)

Local Organization Committee
Jean Bézivin AtlanMod (INRIA & Ecole des Mines de Nantes)
Hugo Brunelière AtlanMod (INRIA & Ecole des Mines de Nantes)
Benoît Combiernale AtlanMod (INRIA & Ecole des Mines de Nantes)
Guillaume Doux AtlanMod (INRIA & Ecole des Mines de Nantes)
Kelly Garcés Ascola & AtlanMod (INRIA & Ecole des Mines de Nantes)
Jean-Sebastien Sottet AtlanMod (INRIA & Ecole des Mines de Nantes)
Massimo Tisi Politecnico di Milano

Program Committee
Patrick Albert ILOG, an IBM Company
Ronan Barrett Ericsson
María Cecilia Bastarrica Universidad de Chile
Raphaël Chenouard LINA, CNRS, Université de Nantes
Florent de Lamotte Université de Bretagne Sud
Jean-Marie Favre University of Grenoble
Louis Feraud IRIT
Mathias Fritzscbe SAP Research CEC Belfast
Dragan Gasevic Athabasca University
Martin Gogolla University of Bremen
Guillaume Hillairet Université de La Rochelle
Zhenjiang Hu National Institute of Informatics
Frédéric Jouault AtlanMod (INRIA & Ecole des Mines de Nantes)
Ivan Kurtev University of Twente
Stéphane Lacrampe Obeo
Ralf Laemmel The University of Koblenz-Landau
Denivaldo Lopes Federal University of Maranh
Frédéric Madiot MIA-SOFTWARE
Ed Merks Macro Modeling
Milan Milanovic School of Business Administration, University of Belgrade
Richard Paige  University of York
Marc Pantel  FeRFA/IRIT/INPT/ENSEEIHT/University of Toulouse
Alfonso Pierantonio  University of L'Aquila
Nicolas Rouquette  Jet Propulsion Laboratory
Jesús Sánchez Cuadrado  University of Murcia
Roberto Tairas  University of Alabama at Birmingham
Massimo Tisi  Politecnico di Milano
Antonio Vallecillo  University of Malaga
Bert Vanhooff  DistriNet, K.U.Leuven
Juan Manuel Vara  University Rey Juan Carlos
Dennis Wagelaar  Vrije Universiteit Brussel

External Reviewers

K. Garcés
F. Thomas
A. Vignaga
# Table of Contents

**Guest Talk** ....................................................... 1
Assigning Meanings to Models ........................................ 1
  *Antonio Vallecillo*

**ATL in Use** ....................................................... 2
Using ATL to define advanced and flexible constraint model transformations .................................................. 2
  *Raphaël Chenouard, Laurent Granvilliers, and Ricardo Soto*

An Approach for Constructing a Domain Definition Metamodel with ATL ....................................................... 18
  *Vanea Chiprianov, Yvon Kermarrec, and Patrick D. Alff*

**Composite Transformations** ........................................ 34
Orchestrating ATL Model Transformations .................................. 34
  *Jose E. Rivera, Daniel Ruiz-González, Fernando López-Romero, and Antonio Vallecillo*

Typing ATL Models in Global Model Management ...................... 47
  *Andrés Vignaga, and María Cecilia Bastarrica*

**Advanced Model Transformation Techniques** ....................... 63
White-Box Coverage Criteria for Model Transformations ................ 63
  *Jacqueline McQuillan, and James Power*

Advanced Traceability for ATL ........................................... 78
  *Andres Yie, and Dennis Wagelaar*

Leveraging Model Transformations by means of Annotation Models .......... 88
  *Juan Manuel Vara, Veronica Andrea Bollati, Belén Vela, and Esperanza Marcos*

**ATL and Other Approaches** ........................................ 103
Composing Models with Five Different Tools: a Comparative Study .......... 103
  *Juan Pedro Silva, Miguel De Miguel, Javier Fernández Briones, and Alejandro Alonso*

Achieving QVTO & ATL Interoperability .................................. 119
  *Alfons Laarman*
Short Papers ................................................................. 134

On Using UML Profiles in ATL Transformations ..................... 134
  Manuel Wimmer, and Martina Seidl

Checking syntactic constraints on models using ATL model transformations 140
  Skander Turki, Eric Senn, Dominique Blouin, Saadia Dhouib,
  Jean-Philippe Dignet, and Johan Laurent

Mutation Analysis for Model Transformations in ATL ............... 145
  Piero Fraternali, and Massimo Tisi

Towards Traceability support in ATL with Obeo Traceability ......... 150
  Freddy Allilairre

UMLQualityAnalysis: UML models measurements with ATL .......... 154
  Mathieu Vénisse

Author Index ............................................................... 160
Assigning Meanings to Models

Antonio Vallecillo
GISUM/Atenea Research Group. Universidad de Málaga (Spain).
av@lcc.uma.es

Abstract. Why do we model? Apart from to generating code, models can have (and should have) many different usages in the realm of Software Engineering including, e.g., understanding and reasoning about the system under study, simulating it, or analyzing its properties before the system is built. For these tasks we need to be able to make questions about the model, and therefore count on languages for expressing both the models and the questions, at the right level of abstraction, and using the appropriate notations. This talk discusses the need to count on different models to describe a system, using different languages, and how semantics can be assigned to them using model transformations. Such semantics define the "meanings" of models, making them amenable to interpretation and analysis. These analyses can range from behavioral simulation and formal reasoning (correctness, validation, model checking) to more agile ones, such as the graphical visualization of models for the detection of design anomalies, for instance.
Using ATL to define advanced and flexible constraint model transformations

Raphaël Chenouard¹, Laurent Granvilliers¹, and Ricardo Soto¹,²

¹ LINA, CNRS, Université de Nantes, France
² Escuela de Ingeniería Informática
Pontificia Universidad Católica de Valparaíso, Chile
{raphael.chenouard,laurent.granvilliers,ricardo.soto}@univ-nantes.fr

Abstract. Transforming constraint models is an important task in recent constraint programming systems. User-understandable models are defined during the modeling phase but rewriting or tuning them is mandatory to get solving-efficient models. We propose a new architecture allowing to define bridges between any (modeling or solver) languages and to implement model optimizations. This architecture follows a model-driven approach where the constraint modeling process is seen as a set of model transformations. Among others, an interesting feature is the definition of transformations as concept-oriented rules, i.e. based on types of model elements where the types are organized into a hierarchy called a metamodel.

1 Introduction

Constraint programming (CP) systems must combine a modeling language and a solving engine. The modeling language is used to represent problems with variables, constraints, or statements. The solving engine computes assignments of variables satisfying the constraints by exploring and pruning the space of potential solutions. This paper considers the constraint modeling process as constraint model transformations between arbitrary modeling or solver languages. It follows several important consequences on the architecture of systems and user practices.

Constraint programming languages are rich, combining common constraint domains, e.g. integer constraints or linear real constraints, with global constraints like alldifferent, and even statements like if-then-else or forall. Moreover the spectrum of syntaxes is large, ranging from computer programming languages like Java or Prolog to high-level languages intended to be more human-comprehensible. This may be contrasted with the existence of a standard language in the field of mathematical programming, which improves model sharing, writing and understanding. The quest of a standard CP language is a recent thread, dating back to the talk of Puget [15]. Another important concern is to employ the best solving technology for a given model. As a consequence, a new kind of architecture emerged. The key idea is to map models written with a high-level CP language to many solvers. For instance within the G12 project,
MiniZinc [13] is intended to be a standard modeling language, and Cadmium [3] is able to map MiniZinc models to a set of solvers. Essence [5] is another CP platform offering an high level modeling language refining Essence specifications to Essence’ models using Conjure [6]. Then hand-written translators can generate models for several different solvers. The role of a mapping tool is to bridge modeling and solver languages and to optimize models for improving the solving process. Cadmium is based on Constraint Handling Rules [8] and is the closest CP platform from our model-driven approach.

In our approach, we suppose that any CP language can be chosen at the modeling phase. In fact, finding a standard language is hard and existing languages have their own features. It then becomes necessary to define mappings between any (pure modeling or solver) languages. This is just the first goal of the new architecture for constraint model transformations defined in the sequel. It follows many advantages:

- Any user may choose its favourite modeling language and the known best solving technology for a given problem provided that the transformation between languages is implemented.
- It may be easy to create a collection of benchmarks for a given language from different source languages. This feature may speed up prototyping of one solver, avoiding hand rewriting of problems into the solver language.
- A given problem may be handled using different solving technologies. Users may not have to play with solver languages.

To this end, we define a generic and flexible pivot model (i.e. an intermediate model) to which any language is mapped. Considering a new language in this framework only requires a parser and a generally simple transformation to the pivot model.

The second goal is to define refactoring operations and optimizations of constraint models using declarative rules. Implementing them over pivot models guarantees the independence from external languages. In other words every operation is implemented once, by means of a so-called concept-oriented rule. In our model engineering approach the elements of models are specified within metamodels, which can be seen as a hierarchy of concepts or types. The rules are able to filter models according to these types, which may be more powerful than syntax-oriented rules.

The third goal is to apply the best transformations for given solving technologies. For instance, a matrix with a few non null elements could be transformed into a sparse matrix when using a linear algebra package. The selection of transformation steps is implemented as a sequential procedure, applying transformations until at least pivot models fit the structure requirements of the target language.

This architecture has been fully implemented using a model-driven engineering (MDE) approach [14]. MDE tools enable us to separate the grammar concerns from modeling concepts using dedicated tools and languages like TCS [11] and ATL [12, 10]. The main advantage is that we can reason about concepts and
their relations through a metamodel. Transformations are specified by defining matchings between concepts at the metamodel level of abstraction. Thus, grammar concerns are relegated into the foreground, while concepts processing becomes the major task.

With respect to previous works, e.g. [4], the new architecture gives more freedom in constraint modeling. s-COMMA is not always the source modeling language and refactoring steps can be chosen. Thus, users can play with any modeling language, until it is mapped to our platform. Dealing with a solver does not require to manipulate its language. Moreover, handling a new language or a new transformation in the system requires a few work. The main limitation of our approach is that only the modeling fragments of languages can be processed i.e., the declarative part. It is not possible to partially execute a computer program that builds the constraint store.

This paper is organized as follows. Section 2 presents an overview of our general transformation framework. Next section introduces the metamodels of two CP languages illustrated on a well-known problem. The pivot metamodel and the transformation rules are introduced in Section 4. Section 5 presents the whole model-driven process including the possibility of selecting relevant mappings. The related work and a conclusion follow.

2 The Model-Driven Transformation Framework

Figure 1 depicts the architecture of our model-driven transformation framework, which is classically divided in two layers M1 and M2 [14]. M1 holds the models representing constraint problems and M2 defines the semantic of M1 through metamodels. Metamodels describe the concepts appearing in models, e.g. constraint, variable, or domain, and the relations among these concepts, e.g. inheritance, composition, or association. In this framework, transformation rules are defined to perform a complete translation in three main steps: translation from source model A to the pivot model, refactoring/optimization on the pivot model, and translation from the pivot model to target model B. Models A and B may
be defined through any CP languages. The pivot model may be refined several times in order to adapt it to the desired target model (see Section 4).

A main feature resulting from a model-driven engineering approach is that transformation rules operate on the metamodel concepts. For instance, unrolling a `forall` loop is implemented once over the `forall` concept, which is independent from the many syntaxes of `forall` in CP languages. In fact, no grammar specification is required for the pivot model. Syntax specifications of CP languages must be defined separately using specific tools achieving text-to-model or model-to-text mappings like TCS [11], which implement both tasks.

### 3 A Motivating Example

In this section, we consider two CP languages, and we motivate the needs and the means for implementing transformations between them.

ECL/PS [17] is chosen as a leading constraint logic programming system. s-COMMA [16] is an object-oriented constraint language developed in our team. Their metamodels are partially depicted in Figure 2 and 3 using UML class diagram notation. The roots of these hierarchies are equivalent, such that the model concept represents the complete constraint problem to be processed.

In s-COMMA, a model is composed of a collection of model elements. A model element is either an enumeration, or a class, or a constant. Each class is composed of a set of class features which can be specialized in variables, constant or constraint zones. Variable with a type defined as a class is an object. Constraint zones are used to group constraints and other statements such as conditionals and loops. The concepts of global constraints and optimization objective are not...
shown here, but can be also defined. The concept of expressions are not detailed in this paper since it is based on classical operator expressions using boolean, set and arithmetic operators.

In the ECL\textsuperscript{iPSe} metamodel, we propose to define a model as a collection of predicates holding predicate elements and variables. Predicate elements are variable features or statements. Variables features is either a constant value assignment, a domain definition, an array or a set definition related to a variable. In fact, we consider that variables are implicitly declared through their features.

Considering the well-known problem of the social golfers, Figure 4 and 5 show two versions of the same problem using s-COMMA and ECL\textsuperscript{iPSe} languages. This problem considers a group of $n = g \times s$ golfers that wish to play golf each week, arranged into $g$ groups of $s$ golfers, the problem is to find a playing schedule for $w$ weeks such that no two golfers play together more than once.

The s-COMMA model is divided in a data file and a model file. The data file contains the golfer names encoded as an Enum concept at line 1 and the problem dimensions defined by means of constants (size of groups, number of weeks, and groups per week). The model file represents the generic social golfers problem using the Model concept. The problem structure is captured by the three classes SocialGolfers, Group, and Week, which are conformed to the Class concept. The Group class owns the players attribute corresponding to a set of golfers playing together, each golfer being identified by a name given in the enumeration from the data file. In this class, the constraint zone groupSize (lines 30 to 32) restricts the size of the golfers group. The Week class has an array of Group objects and the constraint zone playOncePerWeek ensures that each golfer takes part of a unique group per week. Finally, the SocialGolfers class has an array of Week objects and the constraint zone differentGroups states that each golfer never plays two times with the same golfer throughout the considered weeks.
// Data file

enum Name := {a, b, c, d, e, f, g, h, i};

int s := 3; // size of groups
int w := 4; // number of weeks
int g := 3; // groups per week

// Model file

main class SocialGolfers {
    Week weeks [w];
    constraint differentGroups {
        for all (w1 in 1..w) {
            for all (w2 in w1+1..w) {
                card(weeks[w1].groups[g1].players intersect weeks[w2].groups[g2].players) < 1;
            }
        }
    }

    class Week {
        Group groups[g];
        constraint playOncePerWeek {
            for all (g1 in 1..g) {
                for all (g2 in g1+1..g) {
                    card(groups[g1].players intersect groups[g2].players) = 0;
                }
            }
        }
    }

    class Group {
        Name set players;
        constraint groupSize {
            card(players) = s;
        }
    }

    label sets(L);
}

Fig. 4. The social golfers problem expressed in s-COMMA.

Fig. 5. The social golfers problem expressed in ECLiPSe.
Figure 5 depicts the ECL'PS' model resulting from an automatic transformation of the previous s-COMMA model. The problem is now encoded as a single predicate whose body is a sequence of atoms. The sequence is made of the problem dimensions, the list of constrained variables \( L \), and three statements resulting from the transformation of the three s-COMMA classes. It turns out that parts of both models are similar. This is due to the sharing of concepts in the underlying metamodels, for instance constants, forall statements, or constraints. However, the syntaxes are different and specific processing may be required. For instance, the forall statement of ECL'PS' needs the param keyword to declare parameters defined outside of the current scope, e.g. the number of groups \( G \).

The treatment of objects is more subtle since they must not participate to ECL'PS' models. Many mapping strategies may be devised, for instance mapping objects to predicates [16]. Another mapping strategy is used here, which consists in removing the object-based problem structure. Flattening the problem requires visiting the many classes through their inheritance and composition relations. A few problems to be handled are described as follows. Important changes on the attributes may be noticed. For example, the weeks array of Week objects defined at line 9 in Figure 4 is refactored and transformed to the weeks, groups, players flat list stated at line 5 in Figure 5. It may be possible to insert new loops in order to traverse arrays of objects and to post the whole set of constraints. For instance, the last block of for loops in the ECL'PS' model (lines 27 to 39) has been built from the playOncePerWeek constraint zone of the s-COMMA model, but there is two additional for loops (lines 21 and 22) since the Week instances are contained in the weeks array. Another issue is related to lists that cannot be accessed in the same way than arrays in s-COMMA. Thus, local variables \( V_i \) and the well-known nth Prolog built-in function are introduced in the ECL'PS' model.

4 Pivot metamodel and refactoring rules

The pivot model of a constraint problem is an intermediate model to be transformed by rules. The rules may be chained to implement complex transformations. In the following, the pivot and some structural refactoring and optimization rules are presented.

4.1 Pivot metamodel

Our pivot model has been designed to support as much as possible the constructs present in CP languages, for instance variables of many types, data structures such as arrays, record, classes, first-order constraints, common global constraints, and control statements. We believe that it is better and simpler to establish a general CP metamodel, while it is more complex to find a standard CP concrete syntax.

Figure 6 depicts the metamodel associated to pivot models. A pivot model is composed of a collection of elements, divided in three main concepts: types,
features and the concrete concept of predicate. The inheritance tree of types is the same as in the s-COMMA metamodel (see Figure 2). The inheritance tree for model features is also quite similar, except for the concept of record which is an untyped collection of features.

4.2 Pivot model refactoring

We define several refactoring steps on pivot models in order to reduce the possible gap between source and target model. These steps are implemented in several model transformations, most of them being independent from the others. The idea is to refine and optimize models in order to fit the target languages supported concepts.

Model transformations are implemented in the declarative transformation rule language ATL [12]. This rule language is based on a typed description of models to be processed, namely their metamodel. In this way, rules are able to clearly state how concepts from source metamodels are mapped to concepts from the target ones. For the sake of simplicity, only a few of the more representative rules of transformations are shown. ATL helpers are not detailed, but they only consist of OCL navigation.

Composition flattening This refactoring step replaces object variables by duplicating elements defined in their class definition. Names of duplicated variables are prefixed using their container name in order to avoid naming ambiguities. This refactoring step processes object variables and their occurrences, while other entities are copied without modification. In fact, two ATL transformations are defined to ease each refactoring step. The first one removes classes and object variables by replacing them by the concept of record (see Figure 7).
It can be highlighted that there is no ATL rule where the source pattern matches elements being instances of CSPClass. Thus, they are implicitly removed from models (obviously no rule creates class instances). The second transformation removes records to get flattened variables (see Figure 8).

Fig. 7. An extract of ATL rules used to remove the concept of class in pivot models.

In Figure 7, the first rule (lines 1 to 8) is used to copy the root concept of model. Most of other concepts are duplicated with similar rules like the the second one (lines 9 to 22). The helper mustBeDuplicated is defined for each CSPModelFeature and it returns true when: (1) the considered element is an object variable (its type is a class) or (2) it is a feature of a class. Using the last rule, object variables are replaced by records. The helper isObject returns true only if the type of variables is a class. In this rule, features of variable classes are browsed using OCL navigation (collect statement over s.type.features). The rule duplicate is applied on each feature. This rule is lazy and abstract. It is specialized for each CSPModelFeature concrete sub-concepts and it creates as many features as it is called.
The second transformation processes records by replacing them by their set of elements. This is easily done by collecting their elements from their container as shown on Figure 8 at lines 7 to 11. The helper `getAllElements` returns the set of `CSPModelFeature` within a record or a hierarchy of records.

```plaintext
rule CSPModel { from s : PivotCSP!CSPModel to t : PivotCSP!CSPModel { name <- s.name,
    elements <- s.elements->union(s.elements->select(r|
    r.oclsTypeOf(PivotCSP!CSPRecord)
    )->.collect(r|
    r.getAllElements
    )->.flatten())
}}

rule RecordArray { from s : PivotCSP!CSPRecord (not s.array.oclsIsUndefined()) and
    s.elements->select(e|
    e.oclsIsKindOf(PivotCSP!CSPStatement)
    )->.size()>0
} to t : PivotCSP!CSPForAll { index <- i,
    constraints <- s.elements->reject(e|
      e.oclsIsKindOf(PivotCSP!CSTypedElement)
    ),
    i : PivotCSP!CSPIndexVariable { name <- s.name,
        domain <- d
    },
    d : PivotCSP!CSPIntervalDomain{
        lower <- l,
        upper <- thisModule.duplicateExpr(s.array.n)
    },
    l : PivotCSP!CSPIntVal{
        value <- 1
    }
}
```

**Fig. 8.** Main ATL rules used to remove the concept of record in pivot models.

However, some other complex rules must be defined to process arrays of records, (formerly arrays of object variables). Indeed, contained statements have to be encapsulated in a for loop to take into account the constraints for all objects in the array. This task is performed by the rule `RecordArray` which create a new for loop over the record statements (lines 25 to 27). A new for loop requires also a new index variables with its domain (lines 28 to 38).

Using the concrete syntax of s-COMMA, Figure 9 shows the result of this refactoring step. The name of the variable at line 1 corresponds to the concatenate-
nation of all object variable names. The two for loops (lines 2 and 3) were created from the arrays of objects using their name for index variables.

```
1    int set weeks.grou**%s_players[w*g] in [1, 9],
2    for all weeks in [1*w] {
3        for all groups in [1,g] {
4            card(weeks.grou**%s_players[w**w+g*groups]) = g,
5        ...
6    }
7 }
```

**Fig. 9.** Extract of the social golfers pivot model after composition removal and enumeration removal transformations.

**Enumeration removal** During this refactoring step, enumeration variables are replaced by integer variables with a domain defined as an interval from one to the number of elements within the enumeration. Line 1 in Figure 9 shows the result of this transformation on the enumeration called *Name* in the social golfers model: the variable has an integer domain from 1 to 9 replacing the set of nine values \{a, b, c, d, e, f, g, h, i\}. In the same way, occurrences of CSPEnumLiteral are replaced by their position in the sequence of elements of the enumeration type.

**Other implemented refactoring steps** Some other generic refactoring steps have been implemented in ATL to handle some structural needs. They are not detailed since their complexity is similar to the previous examples and to detail all of them is not the scope of this paper.

- If statements can be replaced by one constraint based on one or two boolean implications. For instance, *if a then b else c* becomes \((a \rightarrow b) \land (\neg a \rightarrow c)\).
- Loop structures can be unrolled, i.e. the loop is replaced by the whole set of constraints it implicitly contains. Within expressions, the iterator variable used by the loop structure is replaced by an integer corresponding to the current number of loop turns.
- Expressions can be simplified if they are constants. Boolean and integer expressions are replaced by their evaluation. Real expressions are not processed, because of real number rounding errors. More subtle simplifications can be performed on boolean expressions such as \(a \lor \neg a\) that is always true. Only atomic boolean elements are processed by this last step.
- Matrices are not allowed in all CP language, thus they can be replaced by one dimension arrays. Their occurrences in expressions must also be adapted: the index of the array is computed as follows: \(m[i,j]\) becomes \(m[j + (i \times ncols)]\), where ncols is the number of columns of the matrix m.
- The ECLiPSe language does not allow some sort of expressions. For instance, arrays of int sets cannot be accessed like other arrays with \([\ ]\). Thus, an
specific transformation processes expressions and introduces local variables if needed, as shown on Figure 5 with \( V_i \) variables and \( n \)th predicate calls.

5 Handling CP languages and transformation chains

In this section, we describe the whole transformation chain from a given CP language to another language.

5.1 Parsing CP languages

The front-end of our system parses a source CP language file to get a model representation (on which transformation rules act) matching the concepts of the CP language (injection phase). The back-end generates the code in the target CP language (extraction phase) from the model representation. Interfacing CP languages and metamodels is implemented by means of the TCS tool [11]. This tool allows one to smoothly associate grammars and metamodels. It is responsible for generating parsers of CP languages and also code generators.

Figure 10 depicts an extract of the TCS file for s-COMMA. In a TCS file every concrete concept must have a corresponding template to be matched. For instance, the SSMAClass template implements the grammar pattern for class declarations using at the same time features of this concept defined in the metamodel of s-COMMA. At parsing time on the s-COMMA social golfers example (see Figure 4, the “class” token is matched for the week class statement. Then Week is processed as the name attribute (a string in the metamodel) of a new class instance. Then the “{” token is recognized and the class features (the array of groups and the constraint) are processed by implicit matchings to their corresponding templates using the features reference. Finally the “}” token terminates the pattern description. In the SSMAClass template (lines 4 to 8), several TCS keywords are used. Here is a description of the most important keywords use in Figure 10:

- context defines a local symbol table.
- addToContext adds instances to the current symbol table.
- refersTo accesses to the symbol tables according to the given parameter (here the name) to check the existence of an already declared element.

5.2 Model checking rules

The presented metamodels (see section 2) and the previous subsection show how to get CP language models. However, many irrelevant or erroneous models can be obtained without any additional checking [2]. For instance, variables may be defined with empty domains or expressions may be ill made (e.g. several equalities in an equality constraint).
Several ATL transformations are used to check source models. We transform a source CP model to a model conform to the metamodel Problem defined in the ATL zoo\(^3\). A Problem model corresponds to a set of Problem elements. This concept is only composed of three features:

- **severity** is an attribute with an enumerated type which possible values are: error, warning and critic.
- **location** is a string used to store the location of the problem in the source file.
- **description** is a string used to define a relevant message to describe the problem.

Multiple ATL rules have been implemented to check models. Here is an extract of the list of properties to check:

- Some type checking on expressions. Operands must have a consistent type with the operator. For instance, an equality operator may operate on arithmetic expressions.
- The consistency of variable domains: they must be based on constant expressions and interval domains must have a lower bound smaller than the upper bound.
- No composition or inheritance loops in s-COMMA.

### 5.3 Chaining model transformations

After the injection step or before the extraction step, models have to be transformed with respect to our pivot metamodel. All the refactoring steps presented in Section 4.2 are clearly not necessary in a transformation chain. Indeed, it clearly depends on the modeling structures of the source and target CP languages. The idea is to use most of constructs supported by the target language to have a target model close, in terms of constructs, to our source model. For

\(^3\) [http://www.eclipse.org/m2m/atl/atlTransformations/#KM32Problem](http://www.eclipse.org/m2m/atl/atlTransformations/#KM32Problem)
instance, when translating a s-COMMA model to ECL’PS®, we should transform the objects. So, we choose the composition flattening step. We also need the enumeration removal and other refactoring steps such as the use of local variables and nth predicates. Optionally, we may select the expression simplification steps.

The whole transformation chain is based on three kind of tasks: (1) injection/extraction steps, (2) transformation steps from/to the pivot model, (3) relevant refactoring steps. Transformation chains are currently performed using Ant scripts⁴. These scripts are hand-written, but they can be automatically generated using the am3 tool [1] and the concept of megamodel [7] to get a graphical interfaces to manage terminal models, metamodels and complex transformation chains. However, Automating the building of transformation chains is not possible with current tools. It would require to deeply analyze models and transformations to build relevant transformation chains.

6 Experiments

The benchmarking study was performed on a 2.66Ghz computer with 2GB RAM running Ubuntu. The ATL regular VM is used for all model-to-model transformations, whereas TCS achieve the text-to-model and model-to-text tasks. Five CP problems were used to validate our approach as shown in Table 1. The second column represents the number of lines of the s-COMMA source files. The next columns correspond to the time of atomic steps (in seconds): model injection (Inject), transformations from s-COMMA to Pivot (s-to-P), refactoring composition structures (Comp), refactoring enumeration structures (Enum), transformations from Pivot to ECL’PS® (P-to-E), and target file extraction (Extract). The next column details the total time of complete transformation chains, and the last column corresponds to the number of lines of the generated ECL’PS® files.

<table>
<thead>
<tr>
<th>Problems</th>
<th>Lines</th>
<th>Inject</th>
<th>s-to-P</th>
<th>Comp</th>
<th>Enum</th>
<th>P-to-E</th>
<th>Extract</th>
<th>Total</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>SocialGolfers</td>
<td>42</td>
<td>0.107</td>
<td>0.169</td>
<td>0.340</td>
<td>0.080</td>
<td>0.025</td>
<td>0.050</td>
<td>0.771</td>
<td>38</td>
</tr>
<tr>
<td>Engine</td>
<td>112</td>
<td>0.106</td>
<td>0.186</td>
<td>0.641</td>
<td>0.146</td>
<td>0.031</td>
<td>0.056</td>
<td>1.166</td>
<td>78</td>
</tr>
<tr>
<td>Send</td>
<td>16</td>
<td>0.129</td>
<td>0.160</td>
<td>0.273</td>
<td>-</td>
<td>0.021</td>
<td>0.068</td>
<td>0.651</td>
<td>21</td>
</tr>
<tr>
<td>StableMarriage</td>
<td>46</td>
<td>0.128</td>
<td>0.202</td>
<td>0.469</td>
<td>0.085</td>
<td>0.027</td>
<td>0.040</td>
<td>0.951</td>
<td>26</td>
</tr>
<tr>
<td>10-Queens</td>
<td>14</td>
<td>0.132</td>
<td>0.147</td>
<td>0.252</td>
<td>-</td>
<td>0.017</td>
<td>0.016</td>
<td>0.564</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 1. Times for complete transformation chains of several classical problems.

The transformation chain is efficient for these small problems. The text file injection and extraction are fast. The parsing phase is more expensive than the extraction, since it requires the management of symbol tables. The extraction phase settle for reading the ECL’PS® model. It can also be noticed that model transformations to and from the pivot are quite efficient, more especially the

⁴ http://wiki.eclipse.org/index.php/AM3_Ant_Tasks
transformation to ECLiPSe model. It can be explained by the refactoring phases on the pivot model which simplify and reduce the data to process. We see that the composition flattening step is the more expensive. In particular, the Engine problem exhibits the slowest running time, since it corresponds to the design of an engine with more object compositions.

<table>
<thead>
<tr>
<th>Problems</th>
<th>Inject (s)</th>
<th>s-to-P (s)</th>
<th>Comp (s)</th>
<th>Forall (s)</th>
<th>P-to-E (s)</th>
<th>Extract (s)</th>
<th>Total (s)</th>
<th>Lines (--)</th>
<th>Total/Lines (--)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-Queens</td>
<td>0.132</td>
<td>0.147</td>
<td>0.252</td>
<td>0.503</td>
<td>0.071</td>
<td>0.019</td>
<td>1.124</td>
<td>80</td>
<td>≈0.014</td>
</tr>
<tr>
<td>10-Queens</td>
<td>0.132</td>
<td>0.147</td>
<td>0.252</td>
<td>1.576</td>
<td>0.286</td>
<td>0.060</td>
<td>2.447</td>
<td>305</td>
<td>≈0.008</td>
</tr>
<tr>
<td>15-Queens</td>
<td>0.132</td>
<td>0.147</td>
<td>0.252</td>
<td>3.404</td>
<td>0.659</td>
<td>0.110</td>
<td>4.704</td>
<td>680</td>
<td>≈0.007</td>
</tr>
<tr>
<td>20-Queens</td>
<td>0.132</td>
<td>0.147</td>
<td>0.252</td>
<td>6.274</td>
<td>1.224</td>
<td>0.178</td>
<td>8.207</td>
<td>1205</td>
<td>≈0.006</td>
</tr>
<tr>
<td>50-Queens</td>
<td>0.132</td>
<td>0.147</td>
<td>0.252</td>
<td>32.815</td>
<td>13.712</td>
<td>1.108</td>
<td>48.166</td>
<td>7505</td>
<td>≈0.006</td>
</tr>
<tr>
<td>75-Queens</td>
<td>0.132</td>
<td>0.147</td>
<td>0.252</td>
<td>80.504</td>
<td>54.286</td>
<td>2.456</td>
<td>137.777</td>
<td>16880</td>
<td>≈0.008</td>
</tr>
<tr>
<td>100-Queens</td>
<td>0.132</td>
<td>0.147</td>
<td>0.252</td>
<td>175.487</td>
<td>126.607</td>
<td>4.404</td>
<td>307.029</td>
<td>30005</td>
<td>≈0.010</td>
</tr>
</tbody>
</table>

Table 2. Time of complete transformation chains of the N-Queens problem.

Table 2 presents seven different sizes of the N-Queens problem where the loop unrolling step has been applied. This experiment allows us to check the scalability of our approach according to model sizes. It can be analyzed through the ratio given in the last column which aims at quantifying the efficiency of a transformation chain considering the execution time per generated lines.

As shown on this table, the ratio first decreases, but after 50-Queens it slowly grows up. In fact, the first four row ratios are impacted by the steps before the loop unrolling process, but for the last three rows they become negligible comparing to the whole execution time. It may be noticed that for big problems (after 50-Queens) the ratio smoothly increases. We can thus conclude that our approach is applicable even for huge models, although translations times are not the major concerns in CP.

7 Conclusion and Future Work

In this paper, we propose a new framework for constraint model transformations. This framework is supported by a set of MDE tools that allow an easy design of translators to be used in the whole transformation chain. This chain is composed by three main steps: from the source to the pivot model, refining of the pivot model and from the pivot model to the target. The hard transformation work (refactoring/optimization) is always performed by the pivot which provide reusable and flexible transformations. The transformations from/to pivot become simple, thus facilitating the integration of new language transformations. In this paper, only two languages are presented, but translation processes with Gecode and Realpaver [9] are already implemented.

In a near future, we intend to increase the number of CP languages our approach supports. We also want to define more pivot refactoring transformations to optimize and restructure models. Another major outline for future work is to
improve the management of complex CP models transformation chains. Models can be qualified to determine their level of structure and to automatically choose the required refactoring steps according to the target language.

References

An Approach for Constructing a Domain Definition Metamodel with ATL

Vanea Chiprianov1,3, Yvon Kermarrec1,3 and Patrick D. Alff2

1 Institut Telecom, Telecom Bretagne, UMR CNRS 3192 Lab-STICC
Technopôle Brest Iroise, CS 83818 29238 Brest Cedex 3, France
{vanea.chiprianov},{yvon.kermarrec}@telecom-bretagne.eu

2 BT-North America, 2160 E. Grand Ave, El Segundo, CA 90245, United States

3 Université européenne de Bretagne, France

Abstract. Present day Telecommunications competitive market requires a rapid definition process of new services. To ensure this, we propose to replace the current paper-based process with a computer-aided one. Central to this later process is an information model that captures domain specific knowledge. We approach its construction by defining model querying and model transformation rules in ATL over existing network abstraction layers. We also report on the way we used ATL to define these rules and the benefits of doing so, and pinpoint issues that may be addressed in future ATL releases.

1 Introduction

Present day Telecommunications customer-centric market is characterized by a high demand rate for new services and fierce competition. To remain competitive, service providers need to speed up their service definition process by shortening the concept-to-market time and designing the product right-the-first-time. Currently, this is a paper-based process, relying mainly on trays of documents being exchanged between service designers and programmers. We propose to replace this process with a computer-aided one, which will iteratively capture domain specific knowledge in a Domain Definition Metamodel (DDMM) (the central entity in Fig. 1; in this figure we represent with filled ellipses what we have already done; in parentheses we indicate the toolkit we used). A DDMM provides a sharable, capitalizable, stable and organized structure of information.

Starting from the DDMM, we can define one or several Domain Specific Languages (DSLs) (Sect. 2) which increase the performance of service designers. The DDMM can support the service designers in their collaborative work when defining a new service. It can also be used for verifying properties on models that were defined using the aforementioned DSL. Therefore, as highlighted by [1] also, the DDMM is central to our approach. It is essential that it is wide enough to provide the majority of the concepts service designers need, but also that it is formal and close enough to existing Network Abstraction Layers (NALs) to enable mapping of service specific concepts with network specific components.
Consequently, we decided to construct the DDMM by simplifying existing NALs (Sect. 4) and it constitutes the main focus of this document. The approach of extracting only the relevant aspects of an information model for the generation of DSLs has already been used, for example by [2]. However, their method consists in tagging the information model, whereas we construct a new model by applying transformation rules.

An NAL captures a lot of information, among which there is a big part of the service domain, but in a much more detailed manner than necessary for a designer (an NAL can have tens of thousands of entities, while the service domain has hundreds of concepts). This introduces an additional complexity. The solution is to present the users only the information that is pertinent for them. By eliminating most of the entities that are unknown to service designers and shrinking the inheritance hierarchies, we elaborate such a model. Some NALs are expressed as large class models in UML, so we use model querying (Sect. 3) and transformation techniques and write the model querying and transformation rules using ATL [3] (Sect. 4.2). Because we define more than queries on the input model, changing relations, we go beyond model querying, into model transformation.


2 A Simple Graphical Telecommunications Specific Modeling Language (SGTSML)

If we consider the DDMM corresponding to the abstract syntax of a language (for more details of language definition using model-based tools see [4]), we can define a modeling language for service definition and generate tools for it (e.g.; graphical or textual editor). Moreover, if needed to describe disparate aspects of the service (e.g.; structural, behavioral), the DDMM can be augmented with the information needed to define several DSLs. Sharing the DDMM between several DSLs ensures a consistent view. A similar approach has already been proposed by [2] for policy specification.

We approached the definition of the DDMM with an iterative method in mind, so we started with a simple prototype. This prototype is aimed at defining a simple virtual private network (Fig. 2). The prototype consists of a Network, which may contain several inner networks and several Nodes. The nodes are either Computers, Internet or Routers; they are connected by links which constitute outlinks for the source nodes, and inlinks for the target nodes. The routers can be either customer edge routers (CE) or provider edge routers (PE). Each PE and CE has an Interface, which contains a virtual routing and forwarding (VRF) table containing the VrfRouteTargets and information about the neighboring PEs (BgpIpv4AddressFamilyNeighbors). PEs use the Border Gateway Protocol (BgpRoutingProtocol) to communicate with each other. We also enriched the DDMM with validation rules [5], thus enabling domain level validation. As tool for defining the DDMM we chose TOPCASED [6], a strongly model oriented system engineering toolkit for critical and embedded applications.

For the concrete syntax (see filled ellipses on the right top of Fig. 1) we considered that a graphical syntax would be much easier to use by service designers, as it provides a synthetic, high-level view of the system being considered. Therefore, we defined one using TOPCASED, which has a feature that allows automatic generation of graphical editors for DSLs based on their Metamodel (MM). Using TOPCASED, we also generated the graphical editor for SGTSML.

To describe the semantics of our SGTSML we decided to use the semantics of an existing general purpose object-oriented programming language, Smalltalk. Consequently, we defined template-based code generation rules towards Smalltalk, using OpenArchitectureWare [7]. More details about the definition of SGTSML can be found in [5].

3 Model Querying and Transformation

UML class models can quickly become very large, comprising thousands of classes. Viewing the entire model imposes a high cognitive charge on designers. They usually need to concentrate only on the classes related to a precise functionality (i.e.; a model slice).

Model slicing, as introduced in [8], is a model querying technique, rooted in the classical definition of program slicing, but extends it to UML class models.
Program slicing, as defined by [9], applies a slicing criteria on a program to compute a slice (i.e.; a subset of the source code). Model slices are as well defined using a slicing criterion that is specified with predicates over the model’s features. Consequently, model querying and model slicing in particular constitute a good starting point for our approach. However, because we will change elements in the output model (e.g.; for hierarchy shrinkage we will change the parent class in a generalization relation), we need mechanisms more powerful than just querying, we need model transformations.

As we mentioned in Sect. 1, we construct the DDMM by eliminating classes and shrinking inheritance hierarchies from NALs expressed as UML class models. We think that the most significant reductions will be due to hierarchy shrinkage. Therefore, we are particularly interested in hierarchy shrinkage methods. A more focused technique of program slicing is the class hierarchy slicing. An algorithm for slicing class hierarchies in C++ programs is described in [10]. This algorithm eliminates from an C++ class hierarchy the data and function members, classes and inheritance relations that are unnecessary for ensuring that the semantics of a program P that uses the hierarchy is maintained. However, this type of algorithm is context-sensitive, as it needs the program P that uses the hierarchy. The DDMM that we build is intrinsically context-free, as it has no knowledge and should not depend on the future models that will be defined using it. Therefore, such an approach is not suitable for us.
4 Enlarging the Domain Definition Metamodel

As we argued in Sect. 1, the DDMM is central for our approach. We started by defining a simple prototype, presented in Sect. 2. In order to enlarge the DDMM, we considered using domain analysis methods such as Family-oriented Abstractions Specification and Translation [11] or Organization Domain Modeling version 2 [12]. However, these methods require a lot of time. Moreover, the models defined using the language constructed around the DDMM should be easy to map towards existing models of network components (i.e.; NALs). Consequently, we decided to start from an NAL, specified as a large UML class model (Fig. 3), and define model querying (Sect. 3) rules such that the output model will correspond to the needs of service designers.

![Fig. 3. Model Transformation](image)

In Fig. 3, we exemplify the NAL through an UML class model called LargeHierarchy. LargeHierarchy conforms to its MM, which is UML. The UML MM can be written in several MM definition languages: MOF\(^4\), Ecore\(^5\), etc. We indicate here Ecore because we use Eclipse Modeling Tools\(^6\) as toolkit and, consequently, the UML MM written in Ecore\(^7\). The output MM is also UML, and we call the output model SmallHierarchy, which is an example of a DDMM. Because the transformation has the same metamodel (i.e.; UML) for input and output, it is an endogenous transformation [13]. The transformation rules are written in the module HierarchyReduction, in ATL.

---

5 [http://download.eclipse.org/modeling/emf/emf/javadoc/2.5.0/org/eclipse/emf/ecore/package-summary.html#details](http://download.eclipse.org/modeling/emf/emf/javadoc/2.5.0/org/eclipse/emf/ecore/package-summary.html#details)
The example we chose to illustrate the NAL, *LargeHierarchy*, is presented in Fig. 4. On one hand, an NAL’s components are entities with attributes but no methods. The most frequent types of relation between these components are *association* and *generalization*. On the other hand, the service designers describe a service as a chain of calls to entities named *capabilities*, much as calls to functions. Therefore, a preliminary operation of mapping the capabilities (some hundreds) on the entities of the NAL (some tens of thousands) is done manually. This results in some of the NAL’s classes having methods. They are represented in *LargeHierarchy* by the classes B, C, E, F, G, H. These classes will be part of the DDMM. We call this type of NAL classes, that after slicing appear in the DDMM, *generators*.

![Fig. 4. Example of NAL: Large Hierarchy](image-url)
The model transformation rules are presented, written in natural language, in Sect. 4.1 and written in ATL, in Sect. 4.2. The output model resulted from applying the model transformation rules on LargeHierarchy is presented in Fig. 5. We call it SmallHierarchy and it constitutes an example of DDMM. One can observe that all classes from LargeHierarchy that have at least one method (i.e.; B, C, E, F, G, H) exist in SmallHierarchy too. The initial relations between them (e.g.; the association relation from F and G and the generalization relation between F and H) also appear in the output model. The most interesting relation in SmallHierarchy is the association from classes B and C, resulted from the shrinkage of the initial hierarchy between A and B.

![Fig. 5. Example of DDMM: Small Hierarchy](image)

### 4.1 Model Transformation Rules

Our main idea for querying an NAL is to start from a set of initial generators (i.e.; the classes that have at least one method) and select as generators other classes that are related to the initial generators in a way that is relevant for service designers. The types of relation between classes from NAL, that we consider important for service designers, are association and generalization.

The rule to select the set of initial generators, in natural language:

1. Select all classes from NAL that have at least one method.

The rules for selecting associations, in natural language:

1. Select all direct associations from NAL that relate two generator classes.
2. We define the notion of least derived generator (ldSAG) as the generator which, in a hierarchy that contain generators in a generalization relation, is the highest in the hierarchy. Move association relations down in the hierarchy (i.e.; towards the more derived classes) to the least derived generator.
The rules for selecting **generalizations**, in natural language:

1. Select all direct generalizations from NAL that relate two generator classes.
2. Select all generators implementing an abstract generator.

### 4.2 Model Transformation Rules in ATL

In this section we present the rules for model transformation, written in ATL.

The rule to select the set of initial generators:

```plaintext
module HierarchyReductionUML; /* Module Template
create SmallHierarchyUML : UML from LargeHierarchyUML :
UML;

rule Package {
  from
  ps : UML! Package
to
  pt : UML! Package(
    name <- ps.name)
}

rule Class {
  from
  cs : UML! Class (cs.ownedOperation->notEmpty())
to
  ct : UML! Class (name <- cs.name,
                  package <- cs.package,
                  ownedOperation <- operationLst
                  ),
  operationLst : distinct UML! Operation foreach
    (oper in cs.ownedOperation.asSequence()){
      name <- oper.name)
}
```

The rules related to **association** are *DirectAssociation* and *moveAssocDown-Hierarchy*:

```plaintext
rule DirectAssociation {
  from
  as : UML! Association(
    thisModule.allSags->includes(as.memberEnd.
      asSequence()->first().type) and
    thisModule.allSags->includes(as.memberEnd.
      asSequence()->last().type))
```
The rule DirectAssociation uses the attribute allSags:

```plaintext
---attributes
helper def : allSags : Set(UML!Class) =
let allClasses : Set(UML!Class) = UML!Class.
allInstances() in
allClasses->select(i|i.ownedOperation->notEmpty());

rule moveAssocDownHierarchy{
---moves an association
---does NOT promote an element to an SAG
from
ps : UML!Property(
---the source element should have its participant in
  a hierarchy
  if (ps.ldSagToMoveAssocDownHierarchyTo = '')
  ---the downHierarchy should have at least one SAG
  ---(to simplify the problem, I consider here only
    the ldSAGs)
  then false
else
  if (ps.doesDownHierarchyContainSeveralLdSags(ps.
    ldSagToMoveAssocDownHierarchyTo))
  ---the hierarchy should have only one ldSAG
  ---(it can have a SAG on only one branch)
  then false
else true
endif
using{
targetLdSag : UML!Class = ps.
  ldSagToMoveAssocDownHierarchyTo;}
}
```
— in the target model, we construct an association between one initial class and,
— instead of the other class, from the hierarchy,
the ldSAG from its downHierarchy

name <- ps.association.name,
package <- ps.association.package,
memberEnd <- memberLst
)

memberLst : distinct UML! Property

foreach (asMember in ps.association.memberEnd.
asSequence()){

name <- asMember.name,
type <- if asMember = ps
then asMember.type
else targetLdSag
endif
}

The rule moveAssocDownHierarchy exemplifies the fact that we go beyond model slicing. Not only that we select elements and relations from the input model, but we also change attributes on some of these relations (e.g.; the type of the memberEnd of the association is changed to the targetLdSag). This rule uses the helpers ldSagToMoveAssocDownHierarchyTo, doesDownHierarchyContainSeveralLdSags:

helper context UML! Property def :

  ldSagToMoveAssocDownHierarchyTo : UML! Class =
  let otherMemberEnd : UML! Property =
    if (self.association.memberEnd -> asSequence() -> first() = self)
    then self.association.memberEnd -> asSequence() -> last()
    else self.association.memberEnd -> asSequence() -> first()
    endif
  in
  thisModule.ldSags -> iterate (ldSag; goodLdSag : UML! Class = '' |
  if (ldSag.upperHierarchy -> includes (otherMemberEnd.
type))
  then ldSag
  else goodLdSag
  endif);

helper context UML! Property def :

doesDownHierarchyContainSeveralLdSags (firstLdSag : UML ! Class) : Boolean =
let otherMemberEnd : UML! Property =
if (self.association.memberEnd->asSequence()->first() = self)
then self.association.memberEnd->asSequence()->last()
else self.association.memberEnd->asSequence()->first()
endif
in
thisModule.ldSags->iterate(ldSag; has : Boolean = false |
if (ldSag <> firstLdSag)
then
if (ldSag.upperHierarchy->includes(otherMemberEnd.type))
then true
else has
endif
else
has
endif);

The helper doesDownHierarchyContainSeveralLdSags uses the attributes ldSags and upperHierarchy, the latter using at its turn the helpers ancestors and parents to navigate through the hierarchy:

helper def : ldSags : Set(UML! Class) =
let sags : Set(UML! Class) = thisModule.allSags
in sags->select(sag | sag.upperHierarchy->iterate(
i; notExists : Boolean = true |
if (sags->includes(i))
then notExists = false
else notExists
endif));
—sag.upperHierarchy->excludesAll(sags) ;

helper context UML! Class def : upperHierarchy : Set(UML! Class) =
self.ancestors();

helper context UML! Class def : ancestors() : Set(UML! Class) =
let pars : Set(UML! Class) = self.parents() in
pars->union(
pars->iterate(parent; ancest : Set(UML! Class) = Set{}
| ancest->union(parent.ancestors())))};
helper context UML! Class def : parents() : Set(UML! Class) =
  let allGens : Set(UML! Class) = self.generalization in
  allGens->iterate(gen; par : Set(UML! Class) = Set{} | par->union(Set{gen.general}));

The rules related to generalization are DirectGeneralization and markAsSag:

rule DirectGeneralization { from
gs : UML! Generalization(
gs.general.ownedOperation->notEmpty() and gs.
specific.ownedOperation->notEmpty())
to
gt : UML! Generalization(
general<-gs.general,
specific<-gs.specific)
}

rule markAsSag{
—promotes an element to an SAG
—updates the allSAG and ldSAG lists
—creates an association to the new SAG
—creates generalizations to the new SAG (if necessary)
from
ps : UML! Property(
  if (ps.ldSagToMoveAssocDownHierarchyTo = ' ')
  then false
  else
    if (ps.doesDownHierarchyContainSeveralLdSags(ps.ldSagToMoveAssocDownHierarchyTo))
      then true
    else false
  endif
) endif)
using{
  otherMemberEnd : UML! Property =
    if (ps.association.memberEnd->asSequence()->first() = ps)
      then ps.association.memberEnd->asSequence()->last()
    else ps.association.memberEnd->asSequence()->first()
  endif;
futureLdSag : UML! Class = otherMemberEnd.type;
auxLdSags : Set (UML! Class) = thisModule.ldSags;

to
c\tct : UML! Class(
\t\tname <- futureLdSag.name,
\t\tpackage <- ps.association.package
\t),
\nat : UML! Association(
\t\tname <- ps.association.name,
\t\tpackage <- ps.association.package,
\t\tmemberEnd <- memberLst
\t),
\tmemberLst : distinct UML! Property
\tforeach (asMember in ps.association.memberEnd.
\t\tasSequence()){
\t\tname <- asMember.name,
\t\ttype <- asMember.type)
\tdo{
\t\tthisModule.allSags <- thisModule.allSags->including(
\t\tfutureLdSag);
\t\tthisModule.ldSags <- thisModule.ldSags->including(
\t\tfutureLdSag);
\t\tfor (ldSag in thisModule.ldSags){
\t\tif (ldSag.upperHierarchy->includes(futureLdSag)){
\t\t\tauxLdSags<-thisModule.ldSags.excluding(ldSag);
\t\t\tthisModule.createGeneralization(ldSag,
\t\t\tfutureLdSag);
\t\t}
\t\t}
\t\tthisModule.ldSags<-auxLdSags;
\t}\n
The rule markAsSag uses the rule createGeneralization:

rule createGeneralization (de : UML! Class, a : UML! Class){
to
gt : UML! Generalization(
\t\tgeneral<-a,
\t\tspecific<-de)
4.3 Preliminary performance results

In order to have an idea of the performance of the transformation, we did some preliminary tests. We used as machine a Dell Latitude E4300, with an Intel Core2 Duo CPU P9300 @ 2.26GHz 1.58GHz, 3.45Go RAM, with Microsoft XP SP3. As input model we used the model presented in Fig. 4, which we duplicated several times to obtain a bigger model. Table 1 shows on each line a model with increasing dimensions. The 'Factor' column represents the number of times the initial model has been duplicated. The column 'File' represents the dimension of the input model file, in bytes. The columns 'Classes', 'Associations' and 'Generalizations' represent the number of classes, associations and generalizations respectively contained by each model. The column 'Execution time' represents the execution time, in seconds, of the transformation rules applied on each model respectively. To measure the execution time we used the Eclipse facilities (Run→Run Configurations, Advanced tab, and select 'Run mode only: print execution times to console: 1) transformation only, and 2) total (including model loading and saving)'; we mention that there was only one time printed at the console). The result of under 3 minutes for a model containing approximately 20,000 entities encourages us to think that, when applied on industrial-scale NALs, the transformation will have satisfactory execution times.

<table>
<thead>
<tr>
<th>Ctr</th>
<th>Factor</th>
<th>File (B)</th>
<th>Classes</th>
<th>Associations</th>
<th>Generalizations</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>6,105</td>
<td>14</td>
<td>2</td>
<td>8</td>
<td>0.093</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>42,400</td>
<td>104</td>
<td>16</td>
<td>64</td>
<td>0.312</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>306,348</td>
<td>832</td>
<td>128</td>
<td>256</td>
<td>6.5</td>
</tr>
<tr>
<td>4</td>
<td>1,024</td>
<td>2,120,162</td>
<td>14,336</td>
<td>2,048</td>
<td>8,192</td>
<td>161.531</td>
</tr>
</tbody>
</table>

Table 1. Performance results

5 Lessons Learned

High level of abstraction. We have found that using ATL to describe our model querying algorithm offers a high level of abstraction, especially when compared to hierarchy slicing algorithms like the one presented in [10], due to its declarative constructions (i.e.; the matching of model elements).

Expressive code. The code written to implement the algorithm is much more compact and expressive than if written in a general purpose programming language, like C++; this is a direct consequence of ATL being a DSL for model transformation.

Code modularization and change management. Rule definition provides a strong mechanism for code modularization (i.e.; a rule encodes by itself all the functionality) and change management (e.g.; adding new behavior to the algorithm is as simple as writing new rules).
Performance. The preliminary results we obtained encourage us to think of ATL as applicable to industrial-size models.

Tool support. ATL comes with a virtual machine, an editor with syntax highlighting and code completion for metamodel elements, a debugger. Although sufficiently mature to support development, these tools have missing features that would increase their efficiency (e.g.; adding a breakpoint has to be done from the outline view, there is no code completion for rules, attributes, helpers defined in the same module, no code completion for data types). Also, there are minor bugs (e.g.; the operation `excludesAll` on a collection does not work in the ATL version\(^8\) we used - we had to find a workaround - see helper `ldSags`).

Functional programming style. The functional programming style (e.g.; used to specify the conditions for matching) may be difficult for many programmers to use. Moreover, this programming style produces complex and long expressions, hard to read and understand. Having the documentation of an element appear as a tool tip when hovering over it may be highly useful.

Factorization limits. When comparing the rules `moveAssocDownHierarchy` and `markAsSag`, one observes that the from parts are very similar. We have actually tried to write only one rule, but did not succeed. However, having different rules contributes to the modularity and readability of the code, as each addresses different functionality.

6 Conclusion and Future Work

In this work we were interested in defining a Domain Definition Metamodel (DDMM) for Telecommunications service definition by model querying and transforming large Network Abstraction Layers (NALs) expressed as UML models. We defined the querying and transformation rules in ATL, finding this approach well suited. In the future, we intend to measure the performance of our model transformation rules on industrial-scale NALs (tens of thousands of classes). We also plan to evaluate the DDMM against service designers and use their input to further enlarge and refine it.

References


\(^8\) org.eclipse.m2m.atl.engine.vm 2.0.0
Orchestrating ATL Model Transformations


GISUM/Atenea Research Group. Universidad de Málaga (Spain).
{rivera,daniruiz,fernando,jmbautista,av}@lcc.uma.es

Abstract. The design and development of any complex applications using a Model-driven Engineering approach involve not only the use of many models, but also the use of many model transformations between them. These transformations should be chained together, using Model Transformation Orchestration notations and tools. This paper introduces Wires*, a graphical executable language for the orchestration of ATL transformations, which provides appropriate mechanisms to enable the modular and compositional specification and execution of complex model transformations chains.

1 Introduction

Model-Driven Engineering (MDE) advocates the use of models as the key artifacts in all phases of development, from system specification and analysis, to design and implementation. Each model usually addresses one concern, independently from the rest of the issues involved in the construction of the system. Thus, the basic functionality of the system can be separated from its final implementation; the business logic can be separated from the underlying platform technology, etc. The implementation of a system is then obtained by applying a set of transformations to the different models defined for it. These model transformations are usually interconnected, i.e., the output models of some transformations are the input models of others.

Model Transformation Orchestration (MTO) aims at supporting the construction of complex model transformations from other transformations already defined. MTO is especially relevant in the context of Global Model Management [1, 2], in which megamodels are kinds of registries for resources available from a given model-driven platform, recording all accessible entities like models, metamodels, transformations, tools, and the various relations between them. In this setting, we expect that model transformations become available off-the-shelf and that can be reused to build new tools, processes and, of course, new transformations.

Eclipse provides an ideal infrastructure for building tools to support the use of models. While there is a large selection of tools available for working with individual models and with individual transformations, there is less support for working with collections of models and with collections of model transformations. When we want to chain model transformations, we normally end up using a set of Ant tasks. Although this solution works, it does not provide a high level
specification of chains of transformations, i.e., it remains at quite a low level. Furthermore, the analysis capabilities of the Ant task specifications are limited.

In this paper we present Wires*, a Domain Specific Language that enables the high-level orchestration of model transformations. It provides a visual notation for defining chains of model transformation in a modular and compositional manner, and it is supported by a graphical framework and an execution engine that loads the appropriate models and execute the transformations along the pre-defined path.

The structure of this document is as follows. After this introduction, Section 2 presents the Wires* language. Then, Section 3 describes the execution engine that interprets Wires* models and executes the model transformations. Finally, Section 4 compares our work with other related proposals, and Section 5 draws some conclusions and outlines some future research activities.

2 The Wires* Language

Wires* is a graphical executable language for orchestrating ATL transformations. Fig. 2 shows the metamodel of the language. Basically, Wires* assumes a data-flow process, in which a set of input models (conforming to their corresponding metamodels) are processed by a chain of ATL transformations until a set of output models is produced. The chain is composed of transformations, which act as processing nodes. Parameters represent the consumed and produced data by transformations. Transformations are wired together by directed connectors (called Dataflows or, simply, Wires) that indicate how the outputs of the transformations are linked to the inputs of the next ones. Fig. 1 shows a simple example, where an input model m1 is transformed by two transformations in a row to produce an output model m3.

![Fig. 1. A simple chain of 2 model transformations.](image-url)

Although our notation may resemble at first simple UML 2 activities (transformations are represented by activity nodes, models are represented by object nodes, and wires by object flows), there is a significant difference in the semantics. Basically, Wires* is a data-flow oriented language, whilst UML 2 activities can be either data- or process-flow oriented, or both.

The following paragraphs describe the main concepts of the language with more detail.

Models. In our approach we consider that models and transformations are course-grained building blocks of the MDE process. Models are typed by the
Fig. 2. The Wires* Metamodel
initial metamodel (ModelType) they conform to [3]. Models without incoming Dataflows represent the input models of the process (i.e., the models that are initially loaded), while models with incoming Dataflows represent output models (i.e., models that are saved). Primitive data types (BasicDataType) are also supported.

Transformations. We contemplate the three different kinds of units defined by ATL: modules, libraries and queries. ATL modules and queries are specified by instances of AtomicModelTransformation and Query metaclasses, respectively. They are separated from their corresponding specifications (AtomicModelTransformationType and QueryType, respectively), on which we specify the data file where they are stored (the path attribute), their input and output formal parameters, as well as the auxiliary Libraries they make use of. AtomicModelTransformations may have multiple input and multiple output models. Queries have one single output, which is a primitive type value; one common use of a Query is the generation of a textual output (encoded in a string value), which can be stored in a text file (using BasicData instances).

In addition to Atomic transformations, which represent basic ATL transformations, we also allow the definition of Composite transformations, which represent chains of transformations which can be later used as a single (atomic) transformation, i.e., as building blocks to specify further transformations. Fig. 3 shows an example of the specification of a composite model transformation, CT1, made of t1 followed by t2.

![Fig. 3. The specification of a Composite transformation](image)

We have also identified two special kinds of atomic transformations: Identity and Generic. Identity transformations are model transformations that do not alter the data which flow through them (they are very useful for defining conditional compositions of transformations, as we shall later see). Generic transformation are those used to load and execute ATL transformations produced by high-order transformations. They have a special input parameter (typeParam) that should conform to the ATL metamodel.

Fig. 4 depicts an example of the specification of a Generic transformation. It shows a high-order transformation HOT that produces a transformation outTransf, which is passed as input to the generic transformation gt. Then, gt builds the ATL code corresponding to the model outTransf, compiles it and executes it.
on the input model $m_1$ to produce model $m_3$. The special input parameter type-$\text{Param}$ of a generic transformation is represented by another pin, of the same color as the transformation, to differentiate it from the normal input parameters of the transformation (which are colored as models).

**Fig. 4.** The specification of a *Generic* transformation

**Conditional compositions.** Of course, not all chains of model transformations follow linear paths. A *DecisionNode* is an element whose input parameters are the outputs of *Queries* (i.e., primitive data types), and that contains an OCL expression that will be evaluated to decide which one of its two branches ($\text{trueBranch}$ or $\text{falseBranch}$) is enabled.

**Fig. 5.** A conditional composition

Fig. 5 shows an example of a conditional composition of transformations, using *Queries* and *DecisionNodes*. Assuming that we have two different refactoring algorithms (*Refactor1* and *Refactor2*, implemented by two model trans-
formations), the composition applies one or the other to an input model \( m_1 \) depending on the deep inheritance tree (implemented by \( \text{Query DIT} \)) of \( m_1 \).

In Wires* every transformation can be connected to a \textit{DecisionNode} using \textit{trueBranch} or a \textit{falseBranch}. If unconnected, the transformation will always be enabled. When connected to a \textit{DecisionNode}, the transformation will be enabled only if the corresponding branch is activated after evaluating the \textit{DecisionNode} expression.

![Diagram](image)

**Fig. 6.** Another conditional composition

Fig. 6 shows another example of a conditional composition of transformations, this time using an \textit{Identity} transformation. The diagram represents a situation in which a \textit{Query} \((q)\) checks whether a model that represents a component-based system is already organized in efficient clusters or not. If so, the model is left unchanged; otherwise the model is changed by applying the \textit{cluster} transformation.

**Parallel composition.** As shown in figures 5 and 6, several \textit{Dataflows} can come out of a \textit{model} \((m_1)\), meaning that the model can be used in parallel as input of the corresponding transformations. In case of conditional compositions, only one of these transformations will be active. However, branches without decision nodes are also allowed, i.e., \textit{models} can have several outgoing \textit{Dataflows}. They enable implementing parallel composition of transformations. For example, Fig. 7 shows how a model \( m_1 \) serves as input of two transformations \( t_1 \) and \( t_1' \) which are executed concurrently to produce models \( m_2 \) and \( m_3 \).

In case a model is connected as the output of two or more transformations, and they are executed in parallel, the result is undefined. Please notice that this does not happen with model \( m_1' \) in the conditional composition shown in Fig. 5, because only one of the transformations \( r_1 \) and \( r_2 \) will be executed. The same happens to model \( m_1' \) in Fig. 6, which can be produced by either the \textit{Identity}
Fig. 7. A parallel composition

or the cluster transformations, but not by both during the same execution of the chain.

Loops. In Wires*, loops are simply expressed using DecisionNodes and Dataflows to connect one of their branches to the beginning of the loop. For instance, Fig. 8 shows the Wires* specification of a process that, given an endogenous model transformation \( b \) and an initial model \( m_\text{in} \), iteratively applies \( b \) until it reaches a fixed point (i.e., its input model coincides with its output model). \text{Equals} is a Query that checks whether its two input models are equal or not, returning a Boolean value with the result.

Notice that if \( b \) is an endogenous transformation defined by a set of rules, which are used to specify the behavior of metamodel \( \text{MM1} \) (see [4, 5]), and provided that ATL supports this kind of in-place semantics, then the process depicted in Fig. 8 implements the simulation of the behavior of the system, from an initial state defined by model \( m_\text{in} \). If required, one additional model transformation could take the intermediate output models \( m_1' \) and visualize them, hence enabling the visualization of the successive states of the system during the simulation, and therefore producing a motion picture show.

Temporary vs. persistent models. Finally, models can appear in a Wires* specification not only as inputs or outputs of the whole process, but also in-between. For instance, Fig. 9 is similar to Fig. 1 but storing the intermediate result in a model \( m_2 \). In this way such an intermediate model is created and made persistent, while in the chain shown in Fig. 1 that model is temporarily created by the Wires* execution engine, and deleted after model \( m_3 \) is created.

3 The Wires* Execution Engine

Apart from a graphical notation, Wires* is supported by an execution engine that interprets Wires* models and execute them.

Basically, the way in which the engine works is as follows. Given a Wires* model, it starts by reading and loading the input models, which are those with no incoming Dataflows. For each of these models the execution engine looks for
those transformations which are enabled (i.e., transformations with every input models loaded and with their branch enabled if they are connected to a decision node) and whose input models are already loaded. Once these transformations are identified, they are loaded and executed, producing the corresponding output models. The process continues until no enabled transformation is left unexecuted.

The pseudocode shown in Fig. 10 describes the structure and basic behavior of the algorithm. To illustrate this behavior, let us describe the execution of the chain depicted in Fig. 6. Firstly the execution engine identifies the models without incoming data flows (input models), in this case model $m_1$. The algorithm determines its outgoing elements, which are the $\text{id}$ identity transformation, the query $q$ and the $\text{cluster}$ transformation. Although the execution order is not pre-determined, let us suppose a left-to-right order. Then, the algorithm checks whether $\text{id}$ is enabled, but it is not because the decision node has not activated any branch yet. Secondly, the engine tries to execute the query $q$, which is possible because its input model is loaded and it is enabled (not connected to any
1. elements = searchInputElements();
2. void execute(elements) {
3.     for each e in elements {
4.         targetElements = searchTargetElements(e);
5.         for each targetElement in targetElements {
6.             if ( enabled(targetElement) ) {
7.                 executeOneElement(targetElement);
8.                 targetOutgoings = searchTargetElements(targetElement);
9.                 execute(targetOutgoings)
10.             }
11.         }
12.     }
13. }

Fig. 10. The Wires* engine algorithm.

control branch). Hence the query is executed and the result is obtained, storing it in a temporary variable. Since the algorithm uses an in-depth traversal of the spanning tree, the engine continues by following that path, which leads to a decision node whose expression is just the Boolean output of the query. Suppose that the value is true. Then the corresponding transformation is enabled (in this case, id) and then executed because its input model is already loaded. Being an Identity transformation, it just copies its input model into its output model m1. The engine continues by exploring the last outgoing possibility of input model m1, trying to execute the cluster transformation. However, it is not possible because its incoming branch is not active. And the process ends. In case the result of the clustered query was false, the algorithm would have followed the other branch of the DecisionNode, executing the cluster transformation instead. It is important to remark that the information about the actual input and output parameters of the transformations is used to build the invocation of the corresponding ATL transformations.

The current version of the execution engine is sequential, i.e., in case of possible parallel execution of transformations it executes one after the other (but the order is not pre-determined).

The tool (including the graphical editor and the execution engine) can be downloaded from http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Wires*. Although still in alpha version, it provides all the functionality described here and is fully operative. Further extensions are described later in Section 5.

4 Related Work

This proposal was inspired by the needs of a real project for visualizing component-based systems in order to detect design anomalies. The architecture of the project is based on a set of chained ATL model transformations, which progressively extract the model of the code from the C++ files and DLLs; organize the classes
into components; cluster them to build the software architecture; measure the
components and their connectors; and finally visualize the system. This approach
provided a very clean and modular way to design, architect and develop the tool,
and is shown in Fig. 11.

There are already some approaches for chaining model transformations, some
of them graphical and very powerful such as UniTI [6], other textual such as
MCC [7], and others that use UML activity diagrams for specifying the flow of
control (e.g. [8]). In general, they allow model transformations written in differ-
cent languages to be composed to form a chained process that can be executed.
The Sensoria Development Environment (SDE) [9, 10] provides a very powerful
graphical notation and environment for the orchestration of software tools (and
in particular of model transformations) using a MDE approach. However, when
we tried to use these tools with ATL in our project we ended up using none of
them, but a set of Ant tasks [11]. Basically, we wanted a quick and very simple
way to chain several ATL transformations in a row. In this sense, ATLFlow [12]
is a graphical editor for transformation processes with ATL. It describes
the structure of a transformation flow, and it has the ability to execute both trans-
formations and generators. However, it does not provide support for conditional
branches nor for composite transformations.
The QVT language [13] can also be used to specify complex transformations. In theory it supports composition of transformations through various reuse mechanisms, such as extension or black-box reuse of transformation libraries through its \textit{access} mechanism. Combined with control constructs, structured transformation compositions can be created. The graphical QVT syntax might be used for similar constructions, although the current graphical notation is more apt for specifying relations between the concepts of a single transformation, than for composing transformations. As mentioned in [8], a UML2-oriented approach with composite activities (or even composite structures) can leverage specification of higher-order transformations. However, we found no readily available tool for supporting what we pursued in a simple way.

Other sources of inspiration for our work include those from the CBSD and Software Architecture communities with their ADLs, the Workflow community (that uses UML activity diagrams and other process-based notations such as BPMN or SPEM), and the SOA community, with orchestration languages such as BPEL, for instance. Again, they all provide notations and tools very useful within their contexts, but not directly applicable to our concrete problem.

In this sense, our approach is much more modest than those previously mentioned. It focuses just on ATL transformations, and provides a notation and an execution engine for orchestrating this kind of model transformations. On the other hand, being specific means that it realizes some concrete concepts and mechanisms which are particular of this context, e.g., naturally dealing with high-order transformations.

Here we have focused on the composition of complete transformations. Other works discuss the composition of model transformations at a lower level, i.e., at the level of their constituent rules [14–16]. In general, it is difficult to make a meaningful comparison between both approaches, and it is not clear whether they can be easily combined or not. In any case, this issue falls outside the scope of this paper. For a more detailed evaluation of the composition possibilities offered by rule-based transformation languages we refer the reader to [14].

5 Conclusions and Future Work

This paper has presented \textit{Wires*}, a graphical executable language for the orchestration of ATL transformations, which provides appropriate mechanisms to enable the modular and compositional construction of complex model transformations chains. The language is supported by an execution engine that provides not only a proof-of-concept for the approach, but also a tool that effectively realizes the orchestration of the execution of the model transformations that compose a given chain.

There are many different lines of work that we plan to explore now that we have a tool for orchestrating ATL transformations. Firstly, we want to define and use a type system that allows the static validation of the \textit{Wires*} models, for instance that the \textit{Dataflows} connect output models with valid input models. In other words, we would like to be able to check the type substitutability between
the output of a transformation and the input of another, in such a way that type safety is guaranteed. Thus, we are currently adding a metamodel subtyping algorithm [3, 17] to our execution engine.

Secondly, we also want to connect our tool with other kinds of model transformations, such as model injectors and extractors (defined using, e.g., TCS).

Finally, as future extensions we’d also like to explore the possibility of integrating and executing model transformations defined in other languages, as well as incorporating new ATL features as they are released.

Acknowledgements. This work has been supported by Spanish Research Projects PET2006-0682-00, P07-TIC-03184 and TIN2008-03107.

References

Typing ATL Models in Global Model Management

Andrés Vignaga and María Cecilia Bastarrica
MaTE, Department of Computer Science, Universidad de Chile
{avignaga,cecilia}@dcc.uchile.cl

Abstract. A typing approach for models, and in particular for model transformations, is required for preventing type errors during execution in a Global Model Management (GMM) environment. Based on our previous proposal, in this work we specifically address the typing of ATL models, which are ATL modules and ATL libraries. We discuss function types for ATL modules, and how they are affected by a specific property exhibited by ATL higher-order transformations. We also propose a new type for ATL libraries and discuss additional checks enabled by it. We organize these types in a hierarchy of types for GMM.

1 Introduction

Model transformations are widely recognized as key assets in Model Driven Engineering (MDE) and the AtlanMod Transformation Language (ATL) is a widely used language for defining model transformations. Although the development of a single transformation can be properly addressed using ATL tools, managing a large amount of transformations, along with all their associated assets, could be impractical. Proposals like Global Model Management (GMM) [6], based on the notion of megamodel [4], tackle this issue. In practical terms, a tool implementing GMM enables the execution and composition of model transformations among others. AM3 is one of those tools, which provides specific support for ATL transformations execution via the GMM4ATL extension. In the future, execution support for other model transformation languages (e.g., QVT) may be integrated into AM3 by developing other appropriate extensions. The language supported by the GMM4CT extension is not a general purpose model transformation language, but rather it is a language for defining transformations as compositions of other transformations. When any form of execution is taken into consideration, the notion of execution error is also present. A form of execution error is a type error. A definition of type error depends on a specific language, but always includes the application of a function on arguments for which it was not defined, and the attempted application of a non-function [16]. This kind of errors may be prevented by typing. A type system addressing a set of forbidden errors [7] may be used for determining if a program is well typed or not. A program which is well typed with respect a consistent type system exhibits good behavior, that is, it does not produce forbidden errors upon execution. Our main
concern is to ensure good behavior in GMM, and typing elements within GMM is therefore a means for achieving it.

Understanding model transformations as functions, transformation application and composition may cause the aforementioned kinds of type errors. Thus, an appropriate typing approach for model transformations is core to our goal. The typing approach currently defined in GMM does not ensure good behavior in some specific situations, especially when higher-order transformations (HOT) are involved. This issue was addressed in [15]. However some concerns which are required for its practical realization have not been addressed yet. For example, the potential coexistence of heterogeneous atomic transformations was not taken into account and a single language approach was assumed for that kind of transformations. Additionally, specificities of ATL transformations, such as the separation of modules from libraries, were also ignored. GMM4ATL is currently the only extension for managing atomic transformations within GMM. For this reason in this paper we address in detail the typing of ATL models: ATL modules and ATL libraries. ATL queries are another kind of ATL models but they are not supported by GMM, and they fall out of the scope of this work.

Our proposal enables a more powerful typing approach for ATL models within GMM than the one currently implemented. Its realization then contributes to an enhanced environment for using ATL transformations in practice. Moreover, our results may be used as a basis, or may be even extended, for defining equivalent typing approaches for future GMM extensions.

The remainder of this paper is structured as follows. Section 2 reviews typing approaches for model transformations and their shortcomings. The typing of ATL modules is discussed in Sect. 3. In Sect. 4 ATL libraries are addressed and an improvement to the GMM4ATL metamodel is proposed. Our conclusions and future directions are discussed in Sect. 5.

2 Background

In this section we review the basic concepts of Global Model Management which enable a proper description of the problem addressed in this work. This includes an overview of GMM, a description of its general typing approach, a detail of how ATL models are typed, and an example that illustrates its shortcomings.

2.1 An Overview of GMM

A Global Model Management approach is based on several general concepts corresponding to a generic conceptual MDE framework [5], and also on the concept of a megamodel [4] as a building block for modeling in the large [6]. A megamodel can represent different artifacts (i.e., models) involved in a real world system or process, and their relationships by specifying associated metadata. The type of an artifact, the identifier of a given artifact and its location, etc., are examples of such registered metadata.
A megamodel may be understood as a repository where representations of models and links between them are stored and organized. Such representations may be used for several different purposes. For example, a transformation model may be executed on some given source models. This usually leads to the creation of a set of target models. Such an execution, as well as those target models, are new elements that are also registered in the megamodel. Moreover, links connecting these new elements to other existing elements may be registered as well. Examples of such links include tracing information (i.e., a link connecting source and target models of a given execution) and typing information (i.e., a link connecting the created elements to their appropriate type).

Such a typing information is vital for preventing some execution errors. In its simplest form, the type of a model must be known in order to check if it can be safely passed as a parameter to a given transformation. In turn, the type of a transformation must be known in order to check if it can safely accept a given model, but also for appropriately typing the models it eventually produces. Not being aware of the type of either a model or a transformation is a sufficient condition for making unsafe any execution in which either of these elements are involved.

2.2 Typing in GMM

In GMM elements are entities or relationships between entities. Models are specific variants of entity. Every model conforms to a given reference model, and this relation is in fact the basis for the typing approach in GMM. The type of a model is the reference model it conforms to. We denote this typing relation by \( \mathcal{GMM} \). In the most common case, a terminal model conforms to a metamodel. Additionally, a metamodel conforms to a metametamodel, and a metametamodel conforms to itself. Therefore, only reference models (i.e., metamodels or metametamodels) may occur at the right side of \( \mathcal{GMM} \). Special attention deserves the case of transformations. As a transformation definition is a model, and more specifically a terminal model, its type in terms of \( \mathcal{GMM} \) is a metamodel. Such a metamodel, and more specifically a transformation metamodel, describes the abstract syntax of the language used for expressing the transformation definition, but says nothing about the domain and codomain of the transformation (i.e., as a function type does). For example, the type of any ATLModule in terms of \( \mathcal{GMM} \) is the transformation metamodel ATL. As we shall illustrate later, the \( \mathcal{GMM} \) has-type relation is not well suited for typing transformation models, and a new relation involving function types will be required.

An ATLTransformation is a relationship associated to an ATLModule, which provides, among others, information about the domain and codomain of the associated module. Such information is the type of the source and target models. For this reason it can be regarded as a function type for modules. However, since source and target models are typed in terms of \( \mathcal{GMM} \) within an ATLTransformation, function types are not recursively applied, leading to a one-level function type. Note that such a one-level function type provided by ATLTransformation exactly matches the type information contained in the header of an ATL
transformation definition. As a consequence, the ATLTransformation relationship may be regarded as a proper function type for an ATLModule, but only when non-transformation terminal models are involved as the source or target in that module.

2.3 A Concrete Example

For illustrating this issue we consider the case of the ATL transformation TracerAdder reported in [10]. This transformation receives a transformation \( t \) and returns an updated version \( t' \) of it. The update consists in the addition of target patterns and actions to every rule of \( t \). With such additions, the \( t' \) transformation still produces the same results as \( t \), but also produces a tracing model. Such a model provides tracing information about source and target elements of \( t' \). We recall that the header of an ATLModule expresses the same type information as the ATLTransformation associated to that module. The ATL header of TracerAdder as taken from its original version is as follows (the has-type relation denoted by ' : ' is \(:_{GM} \)):

\[
\text{create OUT : ATL refining IN : ATL;}
\]

TracerAdder is a HOT and is thus in the hypotheses of the case we described before. This header says “TracerAdder receives any ATL transformation and produces an ATL transformation.” In particular, there is no restriction on the sources and targets of \( \text{IN} \), which is appropriate. However, there is also no restriction on the sources and targets of \( \text{OUT} \). Even though TracerAdder is typed as a function, \( \text{IN} \) and \( \text{OUT} \) are not, and thus this header is a one-level function type. This causes a loss of information which may lead to execution errors. For illustrating this we use the sample transformation Src2Dst introduced in the presentation of TracerAdder. The header of Src2Dst is:

\[
\text{create OUT : Dst from IN : Src;}
\]

The result of applying TracerAdder to transformation Src2Dst is transformation Src2DstPlusTrace. The header of such a transformation is therefore:

\[
\text{create OUT : Dst, trace : Trace from IN : Src;}
\]

It can be seen that there exists a relationship between both headers. In particular, the header of Src2DstPlusTrace is the same as that of Src2Dst with the addition of target model trace. Such a relationship holds for any pair of source and target transformations of TracerAdder. However, it cannot be captured in the header of TracerAdder. If that relationship could be expressed, then it could be possible to statically infer the function type of the target transformation \( t' \) as soon as the function type of the source transformation \( t \) is known. With the currently available information, the type of source and target models of \( t' \) are unknown, even after it was generated. As a consequence, further executions of
t’ will be type blind, as the type of its sources is not present. The sufficient condition for the unsafe execution of such target transformation is therefore established.

The case of ATLLibrary models is to some extent similar, as such models are currently typed using :GMM as well. We claim that a different type may be considered for those models, containing richer information, which could enable stronger type checks. In further sections we discuss in detail the function type for ATLModule models proposed in [15], now taking into account some ATL specificities. We also propose a new type for ATLLibrary models.

3 Typing ATL Modules

In this section we address function types for transformation models, specifically for ATL modules. We start by introducing a function type for first order transformations, which is inspired by the similarity of transformations and functions from functional programing languages. Then we discuss the case of higher-order transformations. HOTs exhibit a specific difference with higher-order functions which motivates additional considerations for function types.

3.1 First Order Transformations

Model transformations may be understood in terms of the mathematical notion of a relation. In particular, unidirectional transformations correspond to the notion of a function [8]. Functions are typed by function types. A function type A→B is the type of functions with arguments of type A and results of type B [7]. As in [15], we adopt function types for typing model transformations in what we call transformation types. Therefore, for transformation models, we override the :GMM has-type relation with a new one :Trans. According to this new kind of has-type relation, transformation models are typed by transformation types, which specify the type of the source and the target models of the transformation. Such source and target models are typed in terms of :GMM or :Trans depending on the specific kind of models they are; transformation models are typed using :Trans and other terminal models are typed using :GMM as before.

Figure 1 shows an initial simplified version of the type hierarchy for GMM. GMMType is the base type of all types. A MetamodelType is used via :GMM for typing non-transformation terminal models and a MetametamodelType is used for typing, again via :GMM, both metamodels and metametamodels. In turn, a TransformationType is used for typing transformation models via :Trans, and specifies the type of both the source and target models. Note that if for example the source is a non-transformation terminal model, then the actual type will be an instance of MetamodelType. If it is a transformation model, then the type will be an instance of TransformationType, and so on. Since functions specify one single argument and produce one single result, we incorporated the ProductType type which corresponds to the notion of cartesian product for representing...
For illustrating the use of such types, we show in Fig. 2 the very simple case of the `Class2Relational` transformation [2]. In (a) we show all involved types: metamodels `Class` and `Relational` are the source and target types of transformation type `Class→Relational` respectively. In (b) we show a concrete application of `Class2Relational`, which is an ATL module, on model `sampleClass` for producing model `sampleRel`. Note that we used :GMM for typing these terminal models, whereas we used :Trans for typing `Class2Relational`.

Transformation models are naturally expressed in a concrete language, we therefore label `Class2Relational` as an atl module. However we found more appropriate to keep transformation types technology independent. This enables multiple transformations definitions, which correspond to the same conceptual transformation, in potentially different languages to be typed by the same type. For example, an implementation of `Class2Relational` in QVT could be typed by `Class→Relational` as well.
This scheme enables the occurrence of transformation types within transformation types, unlike the current typing approach. This could suggest that it suffices for properly handling cases such as that of TracerAdder introduced before. Next, we will discuss the case of higher-order transformations, and we show that additional considerations are required, and a compromise concerning the decision just discussed above must be made.

3.2 Higher Order Transformations

As model transformations can be regarded as functions, it is natural to think of higher-order transformations as higher-order functions. The typing approach described above is in fact capable of handling cases as that of TracerAdder. In Fig. 3 we show a similar but more advanced application of types than the one in Fig. 2. In (a) all involved types are represented. At the top we show a (transformation) type for the TracerAdder module. Both of its source and target types are transformation types. The source represents the type of an arbitrary transformation, which is parameterized at its source and target types. Such types are parameters (denoted by a dashed box) and could be any kind of GMM type. Therefore the type of such an arbitrary transformation could be read as \( \forall A:B:GMMType.A \rightarrow B \). Then the type of the result is another transformation type. First, types \( A \) and \( B \) are the same as those used in the source transformation type. In this way we capture that relation between the source and target types of TracerAdder we discussed in the previous section. Second, the target type of the target transformation is actually a product \( B \times \text{Trace} \), where \( B \) is a parametric type and \( \text{Trace} \) is a concrete metamodel. In Fig 3 we do not explicitly represent such a product, rather, we include an association stereotyped as target for each component of the product. In (b) we show the application of TracerAdder to the sample Src2Dst transformation for producing Src2DstPlusTrace. For inferring the type of the latter, type parameters need to be bound. Such a binding is expressed on the link between Src2Dst and TracerAdder. Such an instantiation produce Src2Dst’s type from the source type of TracerAdder, meaning that the
application is correct. But also, that instantiation produces from the target type of TracerAdder the right type for \texttt{Src2DstPlusTrace}.

Higher-order transformations and higher-order functions exhibit some fundamental differences which compels us to further refine our typing proposal. First, in functional languages, functions are assumed to be implemented in the same language. This is not the case with transformations in GMM. Second, a higher-order function receives a function typically for applying it, or in some cases for passing it as a parameter to another function or even for returning it as a result. At least in ATL, a higher-order transformation receives another transformation just for reading and processing its definition. Consider the following implementation in Haskell of the well-known \textit{map} higher-order function:

\begin{verbatim}
map f [] = []
map f (x:xs) = f x : map f xs
\end{verbatim}

In the second line, argument \texttt{f} is used for applying it to \texttt{x} and also for passing it as a parameter in the recursive call. In a higher-order ATL transformation, the definition of a source transformation is used for producing the result. For example, \textit{TracerAdder} reads the definition of \texttt{Src2Dst} and produces the result based on the elements present in that definition. As a consequence it is not the case that \textit{TracerAdder} accepts an arbitrary transformation. Instead, \textit{TracerAdder} accepts an arbitrary ATL transformation, because it relies on the ATL metamodel for processing the input. Note that this particular piece of type information was already provided when transformations were typed based on \texttt{:GMM}, and it was lost in the transition to the \texttt{:Trans}-based approach.

In concrete terms, a transformation type needs to indicate a transformation metamodel name. The meaning of this is that transformation models of that type must conform to the abstract syntax specified by the metamodel. We realize this
idea by annotating transformation types with the name of a transformation language. For example, the type of the Class2Relational transformation of Fig. 2, which was defined as an ATL module, should be now Class^{\text{ATL}}_{\text{Relational}}. In Fig. 4 we show an updated version of the type hierarchy for GMM. TransformationType is now abstract and we added the specific type ATLModuleType which corresponds to $\text{ATL} \rightarrow \text{Relational}$. New transformation types for future languages should be siblings of this type. Note that we already used this new type in part (a) of Fig. 3.

Higher-order transformations will depend on the abstract syntax of the languages used for defining their inputs as long as they are intended to process the definitions of such inputs. In turn, the ability of supporting transformations expressed in different languages is in the rationale of GMM. However, if every transformation language supported by GMM could compile to a common base language, as the .NET Framework does with the Common Intermediate Language (CIL) [9], then HOTs still would be in position of processing transformation definitions, but the dependency to different languages could be relaxed. Therefore transformations would be actually defined in a single language, thus language annotations in transformation types could be removed, but transformations could still be presented to end users in different concrete syntaxes. Such a base language could be the ATL virtual machine bytecode. In that case, transformation models would be replaced by ASM [3] models. Besides this, this approach poses two severe restrictions. First, every language supported by GMM should have an ASM compiler and decompiler. ATL naturally has such tools, and for example such a compiler for QVT is available in the QVT to ATL Virtual Machine Compiler transformation [2]. However, other languages may present irreconcilable incompatibilities with ASM. Second, HOTs must operate on ASM transformations, thus relying on ASM abstract syntax. This could be awkward for most transformation developers.

In the next section we address the typing of other ATL models present in GMM: ATL libraries. We propose a new type for such models and discuss some applications of it. In particular, stronger checks involving libraries and modules are enabled.

4 Typing ATL Libraries

In this section we focus on ATL libraries. First, we discuss the definition of the concept of a library within GMM and suggest an improvement to the GMM4ATL extension. Then, we propose a new type for ATL libraries. Finally, we discuss possible connections between the type of ATL modules and the proposed type for ATL libraries, which leads to stronger type checks.

4.1 ATL Libraries in GMM

ATL libraries are models defining a number of ATL helpers, which may be imported from either other ATL modules or other ATL libraries. Helpers defined within an ATL library may be called from any ATL model importing that library.
However, in GMM, ATL models, and in particular ATL libraries, are a specific kind of transformation models [11]. We do not share this view. ATL helpers, like transformation rules, do operate on source elements. Unlike rules, helpers do not produce target elements. We argue that no transformation is actually carried out by an ATL library itself, and therefore it should not be considered as a transformation model. Every Transformation relationship is associated to a transformation model through its transformationModel property. However, a library is not associated to any Transformation relationship in that way. The definitions that make an ATL library a transformation model are located in the GMM4ATL extension, which should be revised.

As a possible way to deal with this issue, we propose that if the ATLModel class is to be preserved, then its generalization association to TransformationModel should be removed. Additionally, ATLMODULE should be a subclass of TransformationModel, whereas ATLLibrary could simply be a subclass of TerminalModel. This modification is conceptually simple and should not imply a significant development effort. Although it would impact existing megamodel definitions, it would not change the way these related concepts are used in practice. Most importantly, with a modification like this, the GMM4ATL extension would better represent the nature of these GMM concepts.

4.2 A Type for ATL Libraries

ATL libraries are terminal models and as such they could be the subject of some processing through a model transformation. ATL libraries are currently typed using the \( \text{GMM} \) has-type relation. This means that the type of any library is the ATL transformation metamodel. In practical terms, a transformation accepting a library will accept any library, and besides the fact that it is indeed a library, no further checks could be performed. We argue that if defining a new type specific to ATL libraries then stronger checks would be enabled. To that end, we override \( \text{GMM} \) with a new has-type relation \( \text{Lib} \) which relates ATL libraries with a new library type.

ATL libraries define ATL helpers which operate on model elements of one or more reference model. Note that model element names are qualified by the name of the reference model that defines them. For example, a reference to the Table class of the Relational metamodel would be expressed as \textbf{Relational/Table}. A reference to a model element class could be optionally found in the helper body, but at least one such references will be found in the helper’s context. Note that libraries do not have a default module element, and thus any helper within a library must have a context. We define a type operator \( \text{Lib}(A) \) for typing ATL libraries, where \( A \) is a non-empty set of reference models. Such reference models are those that contain the classes which define the context or are used by the helpers within a library. For example, \( \text{Lib(Relational)} \) is the type of all libraries that include helpers which operate only on classes defined by the Relational metamodel. A library may define helpers which operate only on data types or ATL classes, in which case set \( A \) should include the ATL transformation metamodel. The type hierarchy in Fig. 4 includes the library
type just described. Type ATLLibraryType is associated with a non-empty set of reference model types.

This :Lib relation enables restricting the domain of libraries we deal with at a given moment, since the reference models a library operates on are known. As an example, let us consider a transformation MoveToLib, that takes a transformation and moves all its helpers, which are not defined in the context of the module or data types, to a separate library. The type of such a transformation could be \( \forall A,B : \text{GMMType}, (A^{\text{ATL}} B) \rightarrow (A^{\text{ATL}} B) \times \text{Lib}(A) \). The result of MoveToLib is a product. Its first component is the input transformation with the helpers removed, and the second and last component is a library containing those helpers. Such helpers are expected to operate on the source type of the modified transformation. The MoveToLib transformation does not need to be necessarily an ATL transformation. However, since its source transformation is assumed to define helpers, both the source transformation and the first component of the result must be ATL transformations.

MoveToLib transformation may be applied to simple transformations such as Class2Relational, where parameter \( A \) should be instantiated with Class and parameter \( B \) with Relational. However, parameter \( A \) may be any kind of type. This introduces an interesting case for type substitution for \( \text{Lib}(A) \), since \( A \) here is not any type but rather a set of reference model types. Therefore we need to define a specific substitution strategy so that any type can fit within a set of reference model types. When \( A \) is to be substituted with a reference model such as Class, then the context of the library type is a set containing such a reference model. In turn, when \( A \) is to be substituted with a product, then the context of the library type is a set containing the recursive substitution of each component of the product. The case of substituting a transformation type is straightforward. The type actually substituted is the metamodel of the language of the transformation, which is a reference model. For example, if \( A \) is to be substituted with an ATL transformation type, which means that MoveToLib is applied to a HOT, then the second component of the instantiated type of MoveToLib will be \( \text{Lib}(\text{ATL}) \).

Furthermore, the type matching mechanism needs to be specialized for library types as well. Since the context of a library type is a set, the order of its elements is irrelevant for matching. However, it is not necessary that both contexts of two library types are equal to get a match. In fact, if a transformation expects a library conforming to \( \text{Lib}(A) \), then it is sufficient for a model conforming to \( \text{Lib}(A') \) for being safely passed to that transformation that \( A \subseteq A' \) holds. This is because the passed library at most will contain unused helpers (e.g., those defined in the context of some class of a reference model belonging to \( A' \setminus A \)).

4.3 Relating ATL Libraries and ATL Modules

For concluding our discussion on typing ATL models, we further explore relationships between the type of ATL libraries and the type of ATL modules. Even though they could be related, ATLLibrary models are separate entities from ATLModule models. A transformation involving an ATL library (either
as a source or a target model) must have a dedicated argument or result for that element. Therefore, the most simple relationship between the type of an ATLModule and the type of an ATLLibrary is that several instances of the latter may occur in an instance of the former. An example of this is the case of MoveToLib transformation discussed before. Type Lib(A) occurs in the target type of the transformation type of MoveToLib.

An ATL module may import several ATL libraries, and this relationship may be exploited for performing additional type checks. If a module m and a library l are both received by a transformation, then in the case m actually imports l, we could require that the set of types which are the context of l include the source types of m. This could be used for ensuring that the helpers included in the libraries imported by a module are usable on source model elements of that module. Note that by this means we cannot ensure that the libraries are actually those imported by the module; just a match in the types is ensured.

This mechanism relies on links between models and not between types, thus it cannot be used for making such links mandatory. In fact, the specification of that must be done at the type level. Therefore, the type of a module would need to be associated to the type of a library (a set of those types, in general). Doing so, if a transformation type T involves a module type M and a library type L where M and L are associated, then a transformation t:T will only accept module m:M and library l:L if m imports l. This mechanism could be realized adding a zero-to-many association from ATLModuleType to ATLLibraryType in the hierarchy of Fig. 4.

For illustrating all the issues presented so far, in the next section we discuss an example involving a third-order transformation and library types.

5 Example

In this section we use the Measuring ATL Transformations [14] case study for illustrating all the concepts presented in the previous sections. The purpose of such case study is to define an ATL model transformation for computing metrics on another ATL transformation. What is to be measured is the complete ATL definition of a transformation, that is, the module and all imported libraries. The result of the measuring process is a measure model. The solution is as follows. Since the complete definition of an ATL transformation involves at least one module but an arbitrary number of libraries, and an ATL definition requires its arguments to be explicitly enumerated, it is not possible to define a single measuring transformation accepting an arbitrary number of parameters. Instead, a family of measuring transformations are defined. One transformation would measure only a module, another transformation would measure a module and one library, yet another transformation would measure a module and two libraries, and so on. Since all transformations in the family have everything in common, except for the header and some other fixed parts, instead of manually developing every member of the family, a generator working on a template may be defined. Such template embodies one single implementation of the metrics,
Fig. 5. Types involved in the measuring transformation

where a proper modification transforms it to a specific family member. Figure 5 shows the types involved in such a solution.

At the top the type of the transformation that generates the concrete measuring transformation is shown. Argument \texttt{tempIn} corresponds to the template, which is a measuring transformation which operates on an arbitrary module only, and produces a measure model about that module. Argument \texttt{tdIN} corresponds to a transformation descriptor which basically provides the number of libraries which needs to be supported. Therefore, \texttt{OUT} is a measuring transformation like \texttt{tempIn}, which operates on an arbitrary module and the number of libraries indicated by \texttt{tdIN}.

This means that the type of \texttt{OUT} is actually a dependent type which depends on the number of the involved libraries. Here we have a very particular case, since \texttt{MeasurerGenerator} is a transformation that produces ATL transformations with variable headers. The usual way to type functions involving an arbitrary number of input or output values is through collections. This way, the type of \texttt{OUT} could be understood as: “from an ATL module and a collection of ATL libraries, to measure.” However, first ATL does not support collection parameters, and second libraries need not to be all of the same exact type. Our approach is to consider a dependent type \texttt{Lib(C')_n} which for a concrete value of \texttt{n} is expanded to \texttt{Lib(C'_1) \times \ldots \times Lib(C'_n)}, which enables potentially different library types. Although uncommon, this kind of cases deserves further investigation.

In Fig. 6 we show the typed entities involved in an concrete application of the solution described above for measuring the \texttt{KM32Measure} transformation [2], which measures KM3 metamodels. Such an application involves two separate steps: the generation of the measurer transformation, and its execution on the \texttt{KM32Measure} transformation. The first step involves the \texttt{MeasurerGenerator} transformation, which executes on the \texttt{MeasuringTemplate} ATL module and the \texttt{KM32MDesc} terminal model. Such a model describes \texttt{KM32Measure}, and in particular, expresses that the transformation imports four ATL libraries. Such an execution binds \texttt{n} to 4 and produces the \texttt{Measurer4} ATL module.
Fig. 6. Generation of a measurer transformation and its application to the KM32Measure transformation

The second step involves the execution of Measurer4 on the complete definition of KM32Measure: the KM32Measure ATL module, and the four ATL libraries FLAME4KM3, MOOD4KM3, QMOOD4KM3 and EMOOSE4KM3. All four libraries compute different metrics on KM3 metamodels and therefore they are typed by Lib(KM3). Binding A' and B' to KM3 and Measure respectively (which are source and target types of KM32Measure), and C_i to KM3 (which is the context of all four libraries), the execution of Measurer4 safely proceeds for producing the Measures4KM32M terminal model. Since all four libraries are imported by KM32Measure, the inclusion of the instantiation of A' in the instantiation of each of C_i can be checked. This is for making sure that the libraries operate on elements within the source metamodel of KM32Measure, which is in fact the case.

6 Conclusions and Further Work

In this work we have discussed in detail the typing of ATL models, which are ATL modules and ATL libraries, in the context of GMM. Our contribution builds on our more general approach to typing GMM elements presented in [15]. We incorporated the notion of specific transformation types, such as ATLModuleType, which was motivated by how HOTs treat their involved transformations in a multi-language context. We showed that ATL library models are not transfor-
mation models. We also incorporated the new type for libraries \texttt{ATLLibraryType}, which enable stronger checks when a library model is involved in a transformation. Our proposal improves our more general typing approach for GMM as ATL models were addressed in more detail. This may be used to enhance GMM-based environments, such as AM3 [1], as some type errors may be prevented by appropriately typing transformations and by checking their applications, either when they are stand-alone or when they occur within the definition and execution of composite transformations. Our GMM type hierarchy may be used as a basis for typing approaches in future GMM extensions as well.

The typing of ATL models has been addressed in the definition of the ATL language when the headers of ATL transformations were defined. This definition influenced the typing approach currently supported by GMM. To the best of our knowledge, a detailed discussion on the typing of ATL models including HOTs and libraries has not been presented before. The formalization of the MOF layered architecture using Constructive Type Theory in [12] focuses on typing terminal models in general and MOF-based metamodels specifically. In particular, model transformations as MDE-based assets are not addressed, and function types for their typing are only suggested. Model typing is also addressed in [13], where model transformations are typed as well. There, model transformations are in-place procedures, rather than functions, defined in a proprietary research-oriented model transformation language. In particular, transformations are not treated as models, and HOTs are not supported.

In what follows, we plan to integrate our results to the AM3 tool for realizing the benefits of our proposal in a practical environment. Other future work include providing a similar treatment to the typing of other specific models managed by GMM, such as weaving models and text-to-model/model-to-text transformations. ATL models may be related among them, and in this paper we discussed one such static relationship from a typing perspective. Another possible relationship is “superimposition,” which enables a module to incorporate or override either rules or helpers from another module. Such a relationship is processed at module load-time and is therefore dynamic. Currently AM3 does not support superimposition and we plan to address this issue in the future. However, from a typing point of view, it suffices that two modules have the same type for one being safely superimposed by the other. Since the language for composite transformations supported by the GMM4CT extension involves the execution of subtransformations, the multi-language aspect is not an issue like in the case of HOTs. This should to be further taken into account when typing such composite transformations by appropriately using the \texttt{TransformationType} type shown in Fig. 4. Our approach would benefit from metamodel subtyping as supported in [13] as it enables model substitutability in transformation applications. This is related to subtyping transformation types, which in turn enables the potential application of different transformations to the same set of models.

References

White-Box Coverage Criteria for Model Transformations

Jacqueline A. McQuillan and James F. Power

Department of Computer Science,
National University of Ireland, Maynooth,
Co. Kildare, Ireland
{jpower@cs.nuim.ie}
http://www.cs.nuim.ie/research/pop/

Abstract. Model transformations are core to MDE, and one of the key aspects for all model transformations is that they are validated. In this paper we develop an approach to testing model transformations based on white-box coverage measures of the transformations. To demonstrate the use of this approach we apply it to some examples from the ATL metamodel zoo.

Keywords ATL, model transformations, software testing, coverage criteria, metamodels.

1 Introduction

Modelling is concerned with the construction and maintenance of models, but also the transformation from one model to another in the context of Model Driven Engineering (MDE). Model transformations are core to MDE and, similar to conventional software, it is vital to validate the transformations and ensure they are correct. One possible validation method is to systematically test the transformation. While much research has been conducted into transformation languages and tool support for conducting transformations, relatively little attention has focused on testing the model transformations.

In this paper we address the problem of testing model transformations by examining white-box coverage measures for model transformations. We report on some initial experiments of applying this approach to some transformations from the ATL transformation zoo.

The remainder of this paper is organised as follows. Section 2 outlines some of the background information related to testing model transformations. In Section 3 we consider what kind of white-box coverage measures can be derived from ATL transformations. In Sections 4 and 5 we illustrate the use of our test criteria using two model transformation. Finally, Section 6 concludes the paper and discusses future work.
Fig. 1. Coverage measures for model transformations might be derived from metamodel coverage measures, or from research on grammar coverage, or by analogy with code coverage.

2 Background and Related Work

Research relevant to testing model transformations, particularly from a coverage perspective, may be divided into three main strands, as shown in Figure 1 and discussed in the following subsections.

2.1 Coverage of the input metamodel

The most obvious source for coverage data is gained from considering the degree to which the model transformations cover the input metamodel. In standard testing terminology this corresponds to testing using different inputs, most typically characterised as black box testing since it does not presuppose access to the transformation source code.

A range of coverage criteria have been suggested for the various UML diagrams [1]. Since a metamodel can be described using a UML class diagram, coverage criteria for class diagrams provide a basis for developing similar criteria for metamodels. For example, Andrews et al. define a number of coverage measures for class diagrams [2]. A parallel stream of research investigates the generation of test instances of models and metamodels. For example, Gogolla et al. [3] describe an approach to automatically generating model instances (snapshots) from UML class diagrams. Another approach is that of Ehrig et al. [4] which involves the automatic creation of an instance-generating graph grammar for the given metamodel. However, this work does not consider how to evaluate the adequacy of these test cases during the testing process.

The most important work related to coverage analyses of model transformations is that of Fleurey et al. [5–7]. Starting with the model coverage measures...
of Andrews et al., they extend these to the source metamodel for model transformations. They also use these coverage criteria to generate test cases for model transformation testing. Since they focus on generating inputs for the transformation, this approach may be characterised as black-box testing.

An important contribution of the work by Fleurey et al. is the identification of the effective metamodel. This is the section of the source metamodel, often a proper subset, that is relevant to the transformation. Clearly it is important that coverage data be calculated in terms of this effective metamodel, rather than the whole input metamodel, so as not to underestimate the level of coverage.

2.2 Grammar coverage

The use of measures based on input models relies on generating a test suite that adequately covers the input domain of a transformation, as defined by the input metamodel, or a relevant subset. However, there is little work on directly considering the coverage of the transformations themselves. One related area is that of grammar testing, since the process of transforming an input language using a grammar (or a generated parser) is analogous to a model transformation. Various coverage criteria have been proposed, the most simple being rule coverage, which requires that each rule in the grammar be used during testing, although there are many more complex variations [8, 9].

A model transformation consists of more than just rules to match the input, and so any consideration of coverage should also deal with model generation and any internal operations. To date there has been relatively little work on linking coverage of the “front end”, as defined by a grammar or the input metamodel, with coverage of the “back end” as defined by transformation internals and generation code. Hennessy and Power show that applying test suite reduction using only grammar coverage as a criteria yielded poor results for the internals of a C++ parser [10], and thus would suggest that coverage of transformation internals should be considered further.

2.3 Code coverage

One of the standard ways of determining the adequacy of a test suite is to determine the degree to which the test suite exercises the system under test using a coverage analysis [11]. For conventional programming languages the degree of coverage of elements such as statements, decisions, paths, functions etc. can be calculated for a test suite, with the goal of achieving 100% coverage of the chosen element. Given the widespread use and acceptance of such measures in the programming domain, it is natural to consider their use for modelling and model transformation.

Since metamodels can be implemented in program code, often automatically, applying coverage measures to this generated code provides one means of measuring metamodel coverage. This approach was taken in our earlier work on the measurement metamodel where line and branch coverage were used to evaluate coverage for an implementation generated by the Octopus tool [12]. While this
approach has the advantage of simplicity, it is rather indirect, and depends to some degree on decisions taken by the code generator.

3 White-Box Coverage Measures for Model Transformations

In order to calculate the coverage of the ATL rules during the transformation it is necessary to profile the operation of each ATL rule as the transformation takes place. Fortunately the design of the ATL system provides two useful features that facilitate this. First, the compiled ATL rules are actually executed on top of a special-purpose virtual machine [13]. Second, it is possible to run the ATL system in debug mode which prints out the step-by-step execution of instructions on this virtual machine.

The ATL virtual machine is similar in concept to the Java Virtual Machine (JVM) which greatly eases comprehension. It has instructions to access and create model elements, to manipulate data on the stack, and control instructions for selection, iteration and method calls.

3.1 Implementation

Thus, to measure coverage of ATL transformations we implemented a program that works in two phases:

1. First, we process the file of compiled ATL instructions (conveniently represented in XML format) to extract information about the operations, instructions and branch locations and targets.
2. Second, we run the transformation and process the resulting log file to record the actual coverage data for that transformation.

3.2 Relevant structures in the compiled ATL file

We can partition the code generated by the ATL compiler into three main categories:

1. Scaffolding code, such as internal routines to initiate the matching process and help with resolving references. This includes generated functions such as main, _matcher_, _exec_, _resolve_ and resolveTemp.
2. Code corresponding to the rules, which is broken into two separate functions in the assembled code. For any user-defined rule R, the ATL compiler will generate a function _matchR to handle the filtering of model elements, and a function _applyR to handle the instantiation of the target elements, assuming a successful match.
3. Code corresponding to helpers, which can be further subdivided into code for attribute helper initialisation, and code for operation helpers. Attribute helper initialisers will always be invoked by the internal ATL routines but, since they may contain conditional expressions, there is still possibility that they will not be fully covered for a test case. Operation helpers on the other hand must be invoked by the user’s code, and thus there is a possibility that they may not be used at all for a given test case.

Since scaffolding code functions are largely transparent to the user, we do not consider them further in this study.

3.3 Possible coverage measures

Based on the structure of the compiled ATL file, we can immediately identify three kinds of coverage measures:

Rule Coverage is analogous to rule coverage in a grammar: it is simply the percentage of rules that were executed at least once during a transformation. Since each rule is represented as an operation on the ATL virtual machine, implementing rule coverage involves tracking and recording the calls to the operation corresponding to each rule.

Instruction Coverage is analogous to code coverage in a high-level language, with the additional benefit that formatting and layout do not effect the totals. The instruction coverage for a set of transformations is the percentage of instructions that were executed at least once during the transformation. The debug trace for the ATL virtual machine lists each instruction as it is executed, so it is relatively straightforward to measure this coverage.

Decision Coverage measures, for each decision in a program, whether the true and false paths were taken. In ATL transformations, branches are represented by if instructions, and whether they evaluated to true or false can be determined from the trace file.

While instruction coverage corresponds to the most common kind of code coverage, it is not obvious that such a low-level measure is useful in the context of ATL transformation. First, it has the disadvantage of being linked directly to the ATL compiler and low-level VM implementation. Second, it is a measure more suited to an “imperative” style of programming, and can be difficult to relate back to ATL source code, which often contains large nested expressions.

In the remainder of this paper we focus on decision coverage, since this is relatively easy to relate to the ATL source code, and is not so directly tied to the underlying implementation. Since ATL transformation rules are implemented as functions in the compiled code, decision coverage will effectively subsume rule coverage which becomes a special case.
Fig. 2. Four simple possible input models, in increasing order of size, for the *Families2Persons* transformation given in Figure 3. These examples are based on the example distributed with the *Families2Persons* transformation.

4 A Simple Example: The Families2Persons transformation

As an example, consider the *Families2Persons* ATL transformation shown in Figure 3 at the end of this paper, taken directly from the ATL examples [14].

As well as the scaffolding code, the ATL compiler will generate the following methods:

**Code for attribute helpers:** The generated function `_initfamilyName` initialises the attribute helper `familyName` (lines 3-15)

**Code for function helpers:** `isFemale` is generated corresponding to the function helper of the same name (lines 16-25)

**Code for Member2Male:** Two functions called `_matchMember2Male` and `_applyMember2Male` are generated and correspond to lines 28 and 30-32 respectively of the rule `Member2Male`

**Code for Member2Female:** Again, two generated functions `_matchMember2Female` and `_applyMember2Female` corresponding to lines 36 and 38-19 respectively of the rule `Member2Female`

4.1 Deriving coverage data

Even with this simple example, it quickly becomes apparent that it is not trivial to define undisputed coverage measures. The logical place to seek a definition of the *effective metamodel* is in the two transformation rules, and this would appear to indicate that the source model element `Families!Member` should be covered. Thus a rather naive attempt might be the minimal family shown in Figure 2(a).
Table 1. The number of calls of each of the functions in the Families2Persons transformation for each of the four possible inputs in Figure 2. This provides a crude measure of the level of cover, but fails to adequately distinguish between the test cases.

<table>
<thead>
<tr>
<th>Family:</th>
<th>Fig 2(a)</th>
<th>Fig 2(b)</th>
<th>Fig 2(c)</th>
<th>Fig 2(d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>initfamilyName</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>isFemale</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>_matchMember2Male</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>_applyMember2Male</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>_matchMember2Female</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>_applyMember2Female</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2. The decision coverage percentage for each of the functions in the Families2Persons transformation for each of the four possible inputs in Figure 2. These data allow for a greater level of distinction between test inputs than the measures in Table 1.

<table>
<thead>
<tr>
<th>Decision Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Family:</td>
</tr>
<tr>
<td>initfamilyName</td>
</tr>
<tr>
<td>isFemale</td>
</tr>
<tr>
<td>_matchMember2Male</td>
</tr>
<tr>
<td>_applyMember2Male</td>
</tr>
<tr>
<td>_matchMember2Female</td>
</tr>
<tr>
<td>_applyMember2Female</td>
</tr>
</tbody>
</table>

Of course, it is clear that there is a rule each for female and male family members, so logically a better input model would contain at least one instance of each, such as shown in Figure 2(b). A quick analysis shows that all six generated functions are executed at least once for this test case: the number of calls to each function are shown in the data columns of Table 1. However, this simple statement masks some potential complexity. The filter that decides which rule is selected is defined in the helper operation isFemale, so, technically, constructing the effective metamodel correctly is contingent on being able to interpret this helper fully.

An analysis of the decision coverage data for the six functions is shown in Table 2. From this it can be seen that the test case of Figure 2(b) still covers only 75% of the decisions of the isFemale function, and even less of the initialiser for familyName. Adding a daughter to the family, as shown in Figure 2(c) completes the coverage for isFemale. Similarly adding a son familyName, as shown in Figure 2(d) completes the coverage for familyName. Comparing this data with that shown in Table 1 clearly shows that decision coverage is delivering a more complete picture than just counting the number of calls for each function.
4.2 Discussion

Even this simple example shows some of the difficulties involved in identifying the effective metamodel for a transformation. Since the code for `isFemale` could be substituted into the rule definitions, it should clearly be considered relevant to determining the effective metamodel. However, the defined attribute `familyName` is also relevant to the discussion, since it is defined entirely over the input metamodel, and thus a test suite would need to ensure that all of the possible permutations are exercised. Indeed, in this example there is very little of the code that is not relevant to fully defining the effective metamodel.

Even considering decision coverage, as above, does not quite give a full picture of metamodel coverage. For example, the coverage data for `isFemale` is the amalgamated coverage for each call to the function. In theory, a fuller picture would be given by considering the context of the call, so that we could distinguish between `isFemale` as used by `Member2Male` and as used by `Member2Female`. Only the example shown in Figure 2(d) exercises all these options, but this is not shown in the context-independent data of Table 2. There is an obvious potential for combinatorial explosion here, and further research would be required to see if adding context was justified in terms of the improvement in test suite analysis.

5 A larger study

In this section we apply the coverage measures to a larger example, the UML2 to Measure transformation from the ATL Transformations zoo [15]. This transformation calculates a set of metrics for UML class diagrams. As such, it has a well-known source metamodel, so plenty of test cases are available. It also has a computationally-intensive back-end that calculates 51 metrics over the input UML class diagram. It thus represents an extreme example of a transformation where the exact effective metamodel is not easily deducible.

The UML2 to Measure is composed of four modules: one main UML22Measure module, and three modules that calculate metrics called MOOD4UML2, EMOOSE4UML2 and QMOOD4UML2. In what follows, we abbreviate these as U2M, MOOD, EMOOSE and QMOOD respectively.

5.1 A test suite of UML class diagrams

The test cases used in our study were taken from the Eclipse UML2 Tools project [16]. This project includes, among other examples, 19 UML class diagrams from chapter 7 of the UML Superstructure Specification. The 19 class diagrams are described briefly in Table 3, mainly to provide reference to the original source. While there was no coverage data or analysis provided with these models, they were selected as they presumably covered a wide range of features in class diagrams.

The relevant coverage measures were calculated for each of the 19 test cases individually, and then calculated for the test suite as a whole.
5.2 Decision coverage results

Since there are 75 if instructions, this makes a total of 150 possible decisions, and over half of these (88) in the U2M module. The results from the coverage analysis are summarised in Table 4 on a per-model basis. This table has one row for each of the UML models described previously in Table 3. The data in each row represent the fraction of decisions covered for each module in the ATL transformation, summed over all functions in that module.

For example, the value “28/88” in the first column of the first data row of Table 4 measures the decision coverage for the U2M ATL module when run with class diagram #19 as input. There were 44 if instructions in the module, so the total possible decision coverage would have been 88. In this case, only 28 of the possible true/false decisions were taken.

As can be seen from the data in Table 4, the test cases are relatively similar in terms of their module-by-module coverage. This possibly reflects the nature of the transformation itself: since all metrics are calculated for each test case, the level of coverage is quite similar for each. It is notable that decision coverage

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Description used in UML2 Tools project</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.19</td>
<td>Graphic notation indicating exactly one association end owned by the association</td>
</tr>
<tr>
<td>7.20</td>
<td>Combining line path graphics</td>
</tr>
<tr>
<td>7.21</td>
<td>Binary and ternary associations</td>
</tr>
<tr>
<td>7.22</td>
<td>Association ends with various adornments</td>
</tr>
<tr>
<td>7.23</td>
<td>Examples of navigable ends</td>
</tr>
<tr>
<td>7.24</td>
<td>Example of attribute notation for navigable end owned by an end class</td>
</tr>
<tr>
<td>7.25</td>
<td>Derived supersets (union)</td>
</tr>
<tr>
<td>7.26</td>
<td>Composite aggregation is depicted as a black diamond</td>
</tr>
<tr>
<td>7.27a</td>
<td>An AssociationClass is depicted by an association symbol (a line) and a class symbol (a box)</td>
</tr>
<tr>
<td>7.27b</td>
<td>Association Class</td>
</tr>
<tr>
<td>7.28</td>
<td>Class notation - details suppressed, analysis-level details, implementation-level details</td>
</tr>
<tr>
<td>7.30</td>
<td>Examples of attributes</td>
</tr>
<tr>
<td>7.32</td>
<td>Comment notation</td>
</tr>
<tr>
<td>7.33</td>
<td>Constraint attached to an attribute</td>
</tr>
<tr>
<td>7.34</td>
<td>{xor} constraint</td>
</tr>
<tr>
<td>7.39</td>
<td>Example of element import</td>
</tr>
<tr>
<td>7.40</td>
<td>Example of element import with aliasing</td>
</tr>
<tr>
<td>7.48</td>
<td>Multiple ways of dividing subtypes (generalization sets) and constraint examples</td>
</tr>
<tr>
<td>7.54</td>
<td>Instance specifications representing two objects connected by a link</td>
</tr>
</tbody>
</table>

Table 3. A summary of the “Chapter 7” class diagrams from the UML2 Tools project (release 0.8.1 (2008/09/23) [16]. In future tables we refer to these models by number only; this table can be used to refer back to the models in the UML2 distribution.
for any individual test case rarely exceeds 50% in a module, and is particularly low for the EMOOSE module.

The final row of Table 4 shows the cumulative decision coverage when all 19 UML models were transformed, and thus represents the total coverage for all models considered as a test suite. This is important to consider since, even though the individual totals are similar, the decisions being covered may not be the same in each case. For example, all of the test cases cover either 10 or 12 of the 24 decisions in the MOOD module, but the cumulative total of 24 shows that these must be different decisions in at least some of the cases.

The overall coverage of 107/150, or 71% seems quite a reasonable level of coverage for a suite that was not designed with such coverage in mind. Nonetheless, it would clearly be advisable to augment the suite to achieve full coverage.

### 5.3 Detailed analysis of decision evaluation

As a further example of the type of information available from decision coverage analysis, Table 5 analyses the 75 decisions in the four modules for the test cases. This table has one row for each test case, and shows the number of if statements that were never evaluated, that evaluated just to false or true respectively, or...
<table>
<thead>
<tr>
<th>UML Model</th>
<th>No. of if statements</th>
<th>Neither</th>
<th>False</th>
<th>True</th>
<th>Both</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>26</td>
<td>24</td>
<td>18</td>
<td>7</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>26</td>
<td>24</td>
<td>18</td>
<td>7</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>27</td>
<td>24</td>
<td>17</td>
<td>7</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>23</td>
<td>25</td>
<td>15</td>
<td>12</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>26</td>
<td>24</td>
<td>18</td>
<td>7</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>26</td>
<td>24</td>
<td>18</td>
<td>7</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>29</td>
<td>17</td>
<td>19</td>
<td>10</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>33</td>
<td>16</td>
<td>21</td>
<td>5</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>27a</td>
<td>33</td>
<td>16</td>
<td>21</td>
<td>5</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>27b</td>
<td>25</td>
<td>23</td>
<td>21</td>
<td>6</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>21</td>
<td>30</td>
<td>10</td>
<td>14</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>14</td>
<td>33</td>
<td>8</td>
<td>20</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>35</td>
<td>13</td>
<td>24</td>
<td>3</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>30</td>
<td>21</td>
<td>20</td>
<td>4</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>32</td>
<td>15</td>
<td>24</td>
<td>4</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>34</td>
<td>11</td>
<td>24</td>
<td>6</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>28</td>
<td>17</td>
<td>22</td>
<td>8</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>28</td>
<td>16</td>
<td>22</td>
<td>9</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>27</td>
<td>21</td>
<td>23</td>
<td>4</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>Cum. Total</td>
<td>12</td>
<td>13</td>
<td>6</td>
<td>44</td>
<td>75</td>
<td></td>
</tr>
</tbody>
</table>

Table 5. A breakdown of the overall cumulative decision coverage data for all 19 UML models. This table splits the decision instructions into four categories based on the degree to which they were covered during the transformations.

that evaluated to both false and true. This gives a deeper insight as to the overall cause of low coverage, since this could be due either to decisions not being covered at all, or not being evaluated to all possibilities.

For example, the first data row of Table 5 shows the coverage details for class diagram #19. From this we can see that 26 of the decisions in the ATL transformation were never executed, 24 were executed and only ever evaluated to FALSE, 18 only ever evaluated to TRUE, and just 7 were fully tested, being evaluated to both FALSE and TRUE during the transformation. For comparison with Table 4, we can calculate the total decision coverage for this test case as $24 + 18 + (7 * 2) = 56$, as shown in the final column of the first row of Table 4.

For individual test cases, relatively few of the IF statements evaluate to both true and false: only three test cases cause more than 10 of the 75 IF statements to be fully evaluated. Happily, the final row of Table 5 shows that in total 44 of the 75 IF statements are fully evaluated, and efforts to augment the test suite need only concentrate on the remaining 31 statements.
5.4 Comparison with instruction coverage

The ATL transformation contains 151 functions in total (excluding scaffolding code), and these contain a total of 2988 ATL byte-code instructions. The total cumulative instruction coverage for all 19 test cases is 2626 instructions, or 88% of the total. Thus the instruction coverage runs well ahead of the decision coverage figure of 71%.

Just 20 of the 151 functions were never called during any of the 19 transformation runs, and these account for a total of 201 instructions. Since 362 instructions were not covered in total, this means that the remainder, 161/362, or 44% of the uncovered instructions are directly attributable to decisions not taken. Indeed, since the non-coverage of these instructions might have resulted in functions not being called, the figure of 44% is actually a lower-bound on the influence of decision coverage. This demonstrates that improving decision coverage can have a significant impact on improving the coverage of the transformation as a whole.

In the previous subsection we noted that the deficiencies in decision coverage resulted from the incomplete coverage of just 44 if statements. In fact, we can make a further attempt to estimate the ease of localising the lack of coverage. Of the 131 functions that were called at least once during the 19 transformation runs, 104 of these have 100% decision coverage. In fact, of these, 73 contain no decision instructions, and so the function call covers all decisions by default. This means that locating the decisions not taken is localised to just 27 of the 151 functions in the ATL transformation. This suggests that tracking down incomplete decision coverage is at least feasible, even in a relatively large transformation.

6 Conclusion and Future Work

In this paper we have noted the dual nature of a model transformation: part input-recognition, like a grammar, and part generation, like program code. Thus it is possible to extend the notion of rule coverage from grammars to model transformations, and use instruction and decision coverage to evaluate the remaining elements. We have developed tool support to measure decision coverage for the transformation language ATL. Finally, we have shown how these criteria were used in the process of testing a specific model transformation.

The work presented in this paper takes place in the overall context of developing a framework for calculating metrics from various kinds of models. Our approach is based on designing a single metamodel, called the measurement metamodel that describes the quantifiable elements used in software metrics [17, 12]. We are in the process of developing a set of model transformations from other artifacts, such as UML class diagrams and Java programs, into this measurement metamodel. Hence, in order to ensure the correctness of the resulting metrics it is important that the model transformations faithfully represent the source models in each case.

This paper is a first step in identifying suitable white-box coverage measures. In future work we plan to validate the utility of these coverage measures primarily
by examining their correlation with coverage of the effective metamodel. We also intend to compare the fault-detection effectiveness of the coverage criteria presented in this paper with other test adequacy criteria in the literature such as that in [5, 2]. Using this information we plan to establish a full set of test adequacy criteria for testing model transformations and use these criteria for automating the generation of test cases for model transformation testing.

References


module Families2Persons;
create OUT : Persons from IN : Families;

helper context Families!Member def: familyName : String =
  if not self.familyFather.oclIsUndefined() then
    self.familyFather.lastName
  else
    if not self.familyMother.oclIsUndefined() then
      self.familyMother.lastName
    else
      if not self.familySon.oclIsUndefined() then
        self.familySon.lastName
      else
        self.familyDaughter.lastName
      endif
    endif
  endif;

helper context Families!Member def: isFemale() : Boolean =
  if not self.familyMother.oclIsUndefined() then
    true
  else
    if not self.familyDaughter.oclIsUndefined() then
      true
    else
      false
    endif
  endif;

rule Member2Male {
  from
  s : Families!Member (not s.isFemale())
  to
  t : Persons!Male (fullName <- s.firstName + ' ' + s.familyName)
}

rule Member2Female {
  from
  s : Families!Member (s.isFemale())
  to
  t : Persons!Female (fullName <- s.firstName + ' ' + s.familyName)
}
Advanced Traceability for ATL

Andrés Yie$^{1,2}$*, Dennis Wagelaar$^{2}$**

$^1$ Grupo de Construcción de Software, Universidad de los Andes, Colombia
$^2$ System and Software Engineering Lab (SSEL), Vrije Universiteit Brussel, Belgium

{*ayiegarz, dennis.wagelaar}@vub.ac.be

Abstract. Tracing information is an essential part of the ATLAS Transformation Language (ATL). It is used to support interaction between transformation rules, where each rule can use the output of other rules by means of the implicit tracing mechanism. However, tracing information is often useful outside the scope of the transformation execution as well. Currently, ATL offers limited access to the implicit tracing information via the \texttt{resolveTemp()} method. In addition, the tracing information is always discarded after the transformation execution. We propose a method that allows richer runtime access to the tracing information, as well as a method for efficiently storing the tracing information in a separate model.

1 Introduction

The tracing information used in the execution of an ATL transformation consists of the relationships between every source element and its corresponding target elements. These relationships are represented as tracing links, which are used by the ATL virtual machine (ATL-VM) to resolve the interactions and implicit dependencies between the different rules involved in the executed transformation.

Each time one transformation rule is matched, a new tracing link is created between the matched source element and all its corresponding target elements. Subsequently, when a transformation rule implicitly requires the target elements produced for a different rule, the ATL-VM automatically resolves the dependency using the tracing links. Furthermore, when a rule produces several target elements and a different rule needs a specific one of them, the ATL user needs to explicitly specify the name of the output variable associated with the required element using the \texttt{resolveTemp()} API method. This method is the standard way to access the tracing information and is described in more detail in Section 2.

Often, the tracing information is also required outside the scope of the execution as tracing models. These models represent the relationships between source elements and its corresponding target elements. There are several uses for these models [1], for instance, to analyze the impact of a change in the source models,

* This research was supported by the Flemish Interuniversity Council (VLIR) funded CARAMELOS project (http://ssel.vub.ac.be/caramelos/)

** This work is part of the VariBru project, which is funded by the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB).
to propagate the changes, to verify if the requirements are fulfilled in the code or as input in complex transformation chains. [2].

However, ATL offers limited access to the target elements via the `resolveTemp()` API method, using the name of the output variable. Advanced access, such as selecting the output elements by their type, is not possible in the actual ATL-VM. Furthermore, the ATL-VM discards the tracing information after the execution of a transformation rule. This situation forces the ATL users to extend their transformation rules for building and storing the required tracing model. The extension of the transformation rules to generate a tracing model can be done automatically, by using a High-Order Transformation (HOT). In [3], a HOT is presented to automatically append the additional output elements to the original rules to generate the required tracing model. Nevertheless, this solution creates an overhead problem because for every time that the original transformation rules are changed it is necessary to re-apply the HOT. This means, to inject the modified transformation rule into an ATL model, execute the HOT, and extract the ATL code from the generated model. Moreover, the modification of the HOT for adding customized tracing information, or to use a different tracing metamodel, is a complex task.

We propose two methods to offer access to tracing information to the ATL user. The first one is a small extension of the ATL-VM providing richer access to the tracing information during the transformation execution. This richer access can be used with an endpoint rule\(^1\) to generate a tracing model. The second method is based on ATL bytecode adaptation that allows to automatically serialize the existent internal tracing model together with the target model. This method has a minimal impact in the performance of the ATL-VM and offers a user-friendly tracing functionality to the transformations developer.

\section{The implicit tracing mechanism}

ATL’s implicit tracing mechanism is based on an internal facility that encodes relationships between the source element and its corresponding target elements (so-called `transient links`) using the native type `ASMTransientLink`. Every time that one rule is matched, one `ASMTransientLink` is created, the name of the matched rule is assigned and the source and target elements are added to it. Finally, the new link is added to a collection of transient links. This collection is encoded in the internal type `ASMTransientLinkSet` and stored in the module field `thisModule.links`. These two internal types offer basic facilities to manage transient links and its source and target elements. However, the `ASMTransientLink` and the `ASMTransientLinkSet` operations cannot be accessed by the ATL end user. Figure 1 presents a UML diagram with the `ASMTransientLink` and the `ASMTransientLinkSet` native types.

For instance, in the transformation module `Class2Relational` presented in Listing 1, the transformation rule `Class2Table` (line 3) will transform `Class` elements into `Table` elements, and the rule `SingleValuedAttribute2Column` (line 11)

\(^1\) A called rule automatically executed at the end of the transformation.
will transform \textit{Attribute} elements into \textit{Column} elements. Every time one of these rules is matched, a new transient link is created and the source and target elements are assigned to it. Subsequently, the rule \textit{Class2Table} will need to assign all the \textit{Column} elements generated from every \textit{Attribute} of the \textit{Class} into the generated \textit{Table}. Therefore, an implicit dependency exists between both rules. The ATL-VM resolves this dependency using the tracing links by translating the assignments of the source element into assignments with the target elements. In the presented case, the ATM-VM finds all the \textit{ASMTransientLinks} in the \textit{ASMTransientLinkSet} collection that have an \textit{Attribute} as source element and collect the target elements for them.

When an ATL transformation uses the \textit{resolveTemp()} method, the ATL-VM finds the required transient links, but returns only the target elements associated with the name of the variable received as parameter. For instance, in Listing 1, the \textit{resolveTemp()} method is used (line 24) to explicitly assign the owner (table) of a column. Hence, the ATL-VM finds the transient link that has the owner (class) of the attribute (\textit{a.owner}) and assigns the output element with the variable name \textit{‘table’}. This output variable belongs to the rule \textit{Class2Table} (line 7).
3 Runtime access to the tracing information

As was mentioned before, the ATL users occasionally require richer access to the tracing information, for instance, to find target elements by type, or to generate a tracing model. One possibility to create a tracing model, is to access the implicit tracing mechanism at the end of the transformation and recreate the tracing links in a new model. Therefore, we extend the ATL native types implementation to offer an improved access to the implicit tracing mechanism.

We extended the `ASMTransientLinkSet` and `ASMTransientLink` types with a couple of methods that allow to iterate over the `ASMTransientLinkSet` collection and enrich the access to the source and target elements of the `ASMTransientLinks`. These methods are:

- `ASMTransientLinkSet.getLinks()`: this method returns a collection of every transient link in the transformation execution.
– **ASMTransientLink.getSourceElementsMap()**: this method returns a map of the source elements of the transient link. The key of the map is the *name* of the source variable and the *value* is the source element.

– **ASMTransientLink.getTargetElementsMap()**: this method returns a map of the target elements of the transient link. The key of the map is the *name* of the target variable and the *value* is the target element.

With these added methods, the ATL user can access the implicit tracing information using the ATL (hidden) helper attribute `thisModule.links`. For instance, this richer access allows to the developer to get the target elements by their type or to produce a tracing model. It is critical to only access this information as a final step in the transformation, when all the elements are created and assigned after all the rules have been matched and all (traced) model elements are created. In practice, this means that the tracing information should not be accessed in the *from* part of matched rules or in helper attributes without context. To be safe, this access can be done in an endpoint rule, as was presented in Listing 2. The purpose of the rules in Listing 2 is to generate a tracing model as a final step of a transformation execution.

```
1 endpoint rule getTraceModel() {
2   to
3     trace : TRACE!TraceModel {
4       name <- thisModule.toString(),
5       traces <- thisModule.links.getLinks()->collect(e |
6         thisModule.getTraceLink(e))->flatten()
7   }
8 }
9
10 rule getTraceLink(inSource : OclAny) {
11   to
12     trace : TRACE!TraceLink {
13       name <- inSource.getRule().toString(),
14       sources <- inSource.getSourceElementsMap().getKeys()->collect(e |
15         thisModule.getElement(e, inSource.getSourceElementsMap().get(e))),
16       targets <- inSource.getTargetElementsMap().getKeys()->collect(e |
17         thisModule.getElement(e, inSource.getTargetElementsMap().get(e)))
18   }
19   do {
20     trace;
21   }
22 }
23
24 rule getElement(name : String, element : OclAny) {
25   to
26     outelement : TRACE!TracedElement {
27       name <- name
28     }
29   do {
30     outelement.refSetValue('ref', element);
31   }
32 }
```

Listing 2. Tracing transformation rules

We use three rules to generate a tracing model: (lines 1-8) the endpoint rule that creates the root of the tracing model and iterates over the transient link
collection, calling the \textit{getTraceLink} rule for every element in the \textit{links} collection.

(lines 10-22) called rule that creates a link with the name of the rule and its source and target elements. (lines 24-32) is the rule that creates \textit{Traced Elements} for the source and target elements. Although, the presented rules are purely imperative, they are easier to adapt than the HOT presented in [3].

This method offers two main advantages with respect to the HOT method presented before: 1) It is possible to add a \textit{superimposed} module [4] to almost any existent transformation rule\(^2\) to generate a tracing model, keeping the original rules and the tracing-specific rules nicely separated in two different modules. 2) The rules used to generate this model are simpler and easy to change when a customized tracing metamodel is used. The drawback of this method is its poor performance. The cause behind this is the necessity to create a copy of the internal tracing model as a final step of the transformation. We tested this method in the Regular ATL-VM with a copy transformation for a model with nearly 9000 elements and it took in average \textbf{12.4} seconds, this is \textbf{136\%} more than the normal execution of the same transformation that took in average \textbf{5.3} seconds\(^3\).

\section{Automatic storing of the tracing information}

The performance drawback of the previous method is caused by the necessity to iterate and recreate the tracing model. A better strategy, is directly export the implicit tracing model, minimizing the impact in the execution performance of the transformation rules.

The second method that we present has a low impact in the transformation performance, because automatically stores the implicit tracing information in a tracing model after the execution of the transformation rule. Additionally, it is the most end-user friendly, because no extra tracing rules are required. The ATL user only needs to select the generation of the tracing model in the advanced launch configuration\(^4\) and give a name and path for output tracing model\(^5\). The launch configuration dialogs are presented in Figure 2. Once the transformation is executed the tracing model is stored in the selected path.

This strategy is based on bytecode adaptation similar to the way that \textit{superimposition} is implemented in ATL, and is based on three steps, which are explained in the following subsections.

\(^2\) If the existing transformation rule has an endpoint rule a small adaptation is required.

\(^3\) The experiment code can be downloaded from \url{http://ssel.vub.ac.be/svn-pub/ATLTrace/TracingBenchmark}

\(^4\) Provided that the option \textit{Allow Inter-model references} in the advanced launch configuration is activated

\(^5\) Our adaptation automatically selects the tracing metamodel
4.1 Tracing metamodel and model loading

When the transformation starts, an additional metamodel and an output model are loaded. The metamodel is the Tracing metamodel and the model is the output tracing model that conforms to the metamodel. The metamodel is internally named \_TRACE and the model \_trace.

The \_TRACE metamodel is presented in Figure 3, and has a TransientLinkSet meta-class and a TransientLink meta-class that replace the native types. Additionally, this metamodel has a TransientElement meta-class that represents a source or target element in the transformation, and has the name of the rule variable and a reference value to an EObject. This reference points to the actual elements in the source and target models. Furthermore, using the EMF code generation facilities, we implemented the native TransientLinkSet and TransientLink methods in an EMF plug-in. This EMF plug-in allows the visualization of models conform to the \_TRACE metamodel and the transparent call to the methods of the EMF version of the TransientLinkSet and TransientLink inside the ATL-VM.

4.2 Bytecode adaptation

The second step is the bytecode adaptation, which occurs when the ASM file is loaded. The adaptation replaces all the references to the native implementations of TransientLinkSet and TransientLink by our proposed EMF versions. These EMF versions are conform to the \_TRACE metamodel.

In the standard ASM code, the instances of TransientLinkSet and TransientLink are created by three contiguous instructions: 1) a push of the type (TransientLinkSet or TransientLink), a push of \#native and 3) the new instruction. When the ATM-VM executes these instructions a new native instance is created. Our adaptation is done by replacing those push \#native instructions by push \_TRACE instructions. Therefore, when the ATL-VM executes the adapted instructions, instances of our EMF metaclasses TransientLinkSet or
Tracing Metamodel

TransientLink are created instead of the native ones. The standard and adapted instructions are presented in the Figure 4.

In the transformation execution, the ATL implicit tracing mechanism transparently uses the TransientLinkSet and TransientLink EMF versions instead of the native ones.

4.3 Tracing model serialization

When the transformation is finished, the tracing model is stored together with the other target models. An example of a generated tracing model is presented in Figure 5.

The main advantages of this method are that it does not require changes to the original transformations, it is end-user friendly and it has low performance overhead. We tested this method with a copy transformation for a model with nearly 9000 elements and it took approximately 6.0 seconds. This 12% more than the normal execution. Although the _TRACE metamodel and the output _trace model cannot be customized by the end user, we offer the same rich access to the implicit tracing mechanism as the method presented in Section 3. Since
the EMF version is a drop-in replacement of the extended native tracing model presented in Section 3, the transformation rules presented in Listing 2 can be transparently used, or modified to use a customized tracing metamodel.

5 Conclusions and future work

In this paper, we presented a method that enriches the access to the tracing information during transformation execution. This richer access can be used to find target elements by type or to generate an explicit tracing model by iterating over all the transient links. Additionally, we presented a method that automatically stores the implicit tracing model after the execution in a user-friendly way. This method is based on ATL bytecode adaptation.

These two additional methods enrich the ATL-VM with possibilities to the end-users to easily access the tracing information, and to avoid changing their original transformations to generate a tracing model\(^5\). The first method offers the possibility to easily generate a custom tracing model by using a customized tracing metamodel. Even so, the use of this method represents a reduction in the performance of the transformations. The second method offers the most end-user friendly option to automatically generate the tracing model with almost no repercussions in the performance of the transformations.

In order to have a complete traceability solution for ATL it is necessary to consider several possible improvements. First, in some situations when a chain of transformation is used, is useful to calculate transitive closures over a set of tracing models. This means, to offer dedicated operations to navigate through the different tracing steps involved in a transformation chain. Second, the actual implementation of ATL do not trace the elements created by a called rule. It is necessary to define a representation for these elements and relate them with

\(^5\) The code of both extensions can be downloaded from the TracingModel branch of the ATL CVS (http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.m2m/org.eclipse.m2m.atl/plugins/?root=Modeling_Project&pathrev=TracingModel)
the matched rules that trigger the called rule. Finally, in some cases the rule’s name and variable’s name is not the most suitable meta-data for the tracing links and a more customized information is required. This could be possible if we add trace annotations to the transformation rules that can be added to the trace model automatically.

References

2. Vanhooff, B., Baelen, S.V., Joosen, W., Berbers, Y.: Traceability as input for model transformations. (Third ECMDA traceability workshop 2007)
Leveraging Model Transformations by means of
Annotation Models

Juan M. Vara, Verónica A. Bollati, Belén Vela, Esperanza Marcos
Research Group Kybele
Rey Juan Carlos University
Madrid (Spain)
{juanmanuel.vara, veronica.bollati, belen.vela, esperanza.marcos}@urjc.es

Abstract. Model transformations are the key to automate any software
development proposal based on model-driven engineering. However, it might
happen that a unique transformation does not suit for every possible scenario.
This could be the case when the gap between source and target metamodels is
too large or the target metamodel is too complex. In such situations, it may
happen that the transformation never generates some constructions, unless its
execution is driven to do so. In other words, to obtain the most accurate models
we need to introduce some design decisions that guide the transformation. A
way to do so is to model our design decisions as annotations over the source
model – in a model-driven engineering context, everything should be a model.
Then, we can use such annotation model as an additional input for the model
transformation. This work shows how we have applied that technique to
improve our proposal for model-driven development of XML Schemas. The
solution is based on the use of weaving models as annotation models.

Key words: Model-Driven Software Development, XML Schema, Annotation
Models, Weaving Models.

1 Introduction

Since the World Wide Web Consortium (W3C) proposed the eXtensible Markup
Language (XML), it has become the current de facto standard for information
interchange between different organizations.

Initially, the way to define the structure of XML document was by declaring a
Document Type Definition (DTD). DTDs were very efficient at the beginning.
However, as the use of XML documents increased, the weaknesses of DTDs arose.
They present syntactic and semantic failings, especially when the structure of the
conforming XML documents is complex. For instance, they are not well-formed
XML documents, thus developers have to learn how to use two different syntax.
Besides, their mechanisms for defining arity are rather poor. To overcome these
drawbacks, the W3C proposed a new standard for defining the structure of XML

88
documents: the XML Schema Language [23]. It is an alternative to the use of DTDs based on XML that provides a series of advantages with respect to DTDs.

The main improvement of XML Schemas regarding DTDs was providing with a vastly improved data typing system. XML Schemas also support namespaces, which allow different parts of a particular XML document to conform to different XML Schemas [2]. All this given, the XML Schema has been commonly adopted as the de-facto standard for XML document modeling.

In the line of the new trend in software development, in [4] we applied the principles of the Model-Driven Engineering (MDE) approach [20] to the development of XML Schemas. MDE proposes the use of models in each step of the development process. Such models represent the Information System (IS) at different abstraction levels. Besides, the transformation rules between these models have to be defined.

Our proposal starts from a Platform Independent Model (PIM) represented by a UML class diagram. Next, a model to model transformation (M2M) generates a Platform Specific Model (PSM) that represents the XML schema model. Finally, a model to text transformation (M2T) generates the XML document that implements the XML Schema model.

However, when we addressed the task of developing the tooling support for the proposal we faced a common problem on MDE: we need some design decisions to drive the PIM to PSM mapping. Nevertheless, according to the principles of MDE, a development process must provide for the highest degree of automation. In fact, once the PIM has been defined, the rest of the process should be completely automatic. The simplest solution in this case is to use a default value for these design decisions when coding the model transformation.

But defining a one-size-fits-all model transformation in such contexts is not enough. It may occur that some constructions are never generated on the target model. This approach could be improved by using a parameterizable transformation. Non-uniform mappings [10] and generic transformations [21] were the first works in this direction. Note that all the artefacts handled on a MDE process should be models. So, the parameters we need to drive the execution of the transformation have to take the shape of a model.

In this work we use a weaving model [1] as a container for those parameters or design decisions. Before executing the model transformation, we define a weaving model that annotates the source model. Then, both the source and the weaving model are the inputs to generate the target model. This way, different target models can be obtained from a particular source model, depending on which weaving/annotation model is used.

The results lend strong support to the idea that current MDE tools, like model transformations and weaving models are powerful enough to fulfill the requirements of XML Schema development.

The rest of this paper is organized as follows. Section 2 briefly introduces two MDE concepts: weaving models and annotation models. Section 3 presents the proposal. To that end it describes the model-driven development process for XML Schemas, the involved metamodels and the design decisions allowed when moving from the PIM to the PSM. Section 4 focuses on the implementation of the proposal by means of a case study. Section 5 summarizes related works. Finally, section 6 sums up the main conclusions as well as the future work.
2 Preliminaries

Before focusing on the development of the model transformation addressed in this work, we introduce some previous concepts on which our work has been based: Weaving Models and Annotation Models.

2.1 Weaving Models

Model transformation is essentially intended to define executable operations. Hence it is not always adapted to define and to capture various kinds of relationships between models elements. However, we often need to establish and handle these correspondences between the elements of different domains, each one defined by means of a model. The correspondences may be informal, incomplete, and preliminary. In many cases they may not be used directly to drive an executable operation. Model weaving is the process of representing, computing, and using these initial correspondences. This way, a set of correspondences between different model elements is represented as a weaving model [1].

A Weaving Model is thus a special kind of model used to establish and handle the links between models elements. This model stores the links (i.e., the relationships) between the elements of the (from now on) woven models. We illustrate this idea in Fig. 1: Mw is a weaving model that captures the relationships between Ma and Mb (the woven models), denoted by the triple [Mw, Ma, Mb]. Then, each element of Mw links a set of elements of Ma with a set of elements of Mb. For instance, the r2 element of Mw defines a relationship between a2 and a3 from Ma, and b1 from Mb.

![Fig. 1. Model Weaving overview](image-url)

To create and handle the weaving models used in this work we used the ATLAS Model Weaver (AMW). The model weaver workbench provides a set of standard facilities for management of weaving models and metamodels [9]. Moreover, it supports an extension mechanism based on a Core Weaving Metamodel [8]. The Core Weaving metamodel contains a set of abstract classes to represent information about links between model elements. These classes are extended to specify new domain-specific weaving metamodels.
2.2 Annotation Models

MDA must support incremental and iterative development. This means that mappings between models must be repeatable. So, if a mapping requires input in addition to the source models, this information must be persistent. However, it must not be integrated into the source model, because it would mean polluting the source with information from outer domains, which is not desirable. These additional mapping inputs take the form of annotations [15].

Models are annotated or decorated to insert information that is not defined in the source metamodel. Annotation data usually is not conceptually relevant to be part of the metamodel. For example, annotations are often meta-information used for pre-processing, testing, logging, versioning, or parameterization [8].

The idea behind the use of model annotations for model transformation is the following: a model transformation specifies a set of rules that encodes the relationships between the elements from the input and output metamodels. Thus, it is defined at metamodel level, i.e., it maps elements from the input and output metamodels. It can be used to generate an output model from any model conforming to the input metamodel. That is to say that the model transformation program works for any model defined according to the input metamodel. However, in some situations this approach could be too generic and some additional considerations have to be made each time the transformation is executed. These considerations can take the form of annotations and we can collect them in an annotation model.

For instance, given a PIM and a PSM metamodel, a model transformation between them, and one terminal model conforming to the PIM metamodel, different PSM will be generated for each annotation model used to execute the transformation. This is the approach we follow in this work. Its application is showed in the following sections.

3 Automatic XML Schema Development in MIDAS framework

This work is framed in MIDAS [13], a model-driven methodology for IS development. Specifically, our proposal focuses on the content aspect of MIDAS that corresponds with the traditional concept of Database (DB). Fig. 2(a) summarizes the development process. At PIM level we use a conceptual data model represented by an UML class diagram. At PSM level, we use two different models depending on the technology selected to implement the DB: the Object Relational (OR) model and the XML model. In [4] and [5] we introduced the proposed MDE development process for XML and OR technology, respectively.

In order to support the MIDAS framework we are building a MDE environment for IS development called M2DAT (MIDAS MDA Tool). The work in this paper is integrated in the M2DAT-DB (MIDAS MDA Tool – Database) module, which provides the tooling for the content aspect of MIDAS.

All the technical solutions used to develop M2DAT share a common basis: they are part of the Eclipse Modelling Project (EMP, http://www.eclipse.org/modeling/). The EMP facilitates the deployment of any model-driven engineering process by providing a unified set of modeling frameworks, tooling, and standards
implementations. All of these facilities are built upon a common modelling framework: the Eclipse Modelling Framework (EMF) [16]. Using EMF we have developed the model editors for each metamodel considered in MIDAS.

For depicting the class diagrams used as conceptual data model at PIM level we use UML2, the implementation of the UML 2.0 standard of EMF: To develop the PIM to PSM model transformation we use the ATLAS Transformation Language (ATL) [11]. Currently, ATL is considered the de-facto standard for M2M transformations. It offers an Integrated Development Environment (IDE) completely integrated in Eclipse. Besides, it is framed in the AMMA (ATLAS Model Management Architecture) platform that includes other facilities in the MDE context, such as the KM3 metamodeling language or the ATLAS Model Weaver (AMW) tool.

We have evaluated several proposals for code generation, such as MOFScript, JET and XPand. Finally we are using the MOFScript [17] language. It is a prototype implementation based on concepts submitted to the OMG MOF M2T transformations RFP process [18]. Since it was the first submission to the OMG RFP, it is probably the most contrasted and the most commonly used, despite the fact that recently XPand and other template-based approaches are gaining ground. Besides, the training period of MOFScript is quite short. After coding some M2M transformations, moving to M2T transformations is quite easy.

As shown in Fig. 2(a), this work focuses on the transition from the conceptual data model to the XML model. The first step towards the completion of this transition was to define the mapping rules from PIM to PSM using graph grammars. Afterwards, we coded these rules using ATL and finally, we coded the M2T transformation that returns the XML Schema. For more details see [4].

However, by the time we were coding the ATL module, we realized that some information needed to generate the target model was not included in the source model. For each execution of the transformation some extra information was needed. In some sense, this extra information can be shown as a way of parameterize the transformation. In a first iteration we opted for using a set of default values for these extra data. Nevertheless, it turned out that working this way, the transformation was not able to produce some constructs on the target model, whichever the source model used was. For instance, all the attributes of a particular XML element had to be grouped using the same compositor, whether it was sequence, choice or all. We will show a detailed example in the following sections.

The first option to overcome this drawback was to extend the source metamodel to support the modeling of this extra information. However, it is not fair to pollute the metamodel with concepts not relevant for the domain that it represents. Back to the mentioned example, the decision on how a set of PIM attributes should be mapped to an XML Schema model is a platform specific matter. It should not be considered when defining the PIM and it should not have any influence on the way we define the PIM.

Therefore, we needed a different way to collect this extra information that was related to the source model but not included in it. Since this information or parameters had to be available for the ATL program and considering that we were in a MDE context, the best option was to use another model (and thus to define a new metamodel): an annotation model.
Finally, instead of defining a completely new metamodel to create our annotation models, we use a weaving model to annotate the input model. To that end, we opted for using the annotation metamodel defined as an extension to the core weaving metamodel in [8].

All this given, the resulting PIM to PSM mapping is summarized in Fig. 2(b). For every execution of the ATL transformation - in other words, for each source model (Conceptual Data Model) - we define a weaving model (Annotation Model) that contains a set of annotations. They represent the extra information needed to execute the transformation (we may refer to them as the parameters of the transformation). Thus, the target model is generated from the source model and the weaving model. This process allows obtaining different XML Schema models from a particular conceptual data model just by modifying the weaving model.

3.1 Metamodels

As Fig 2(b) shows, we use three different metamodels to map a conceptual data model to an XML Schema one: the UML2 metamodel, the XML schema metamodel and the Annotation metamodel. Since the UML2 metamodel is well known, in the following we briefly introduce the other two.

It is worth mentioning that our first step towards a model-driven approach for XML Schemas development was the definition of a UML profile for XML Schema modeling [22]. However, when we addressed the task of implementing the PIM to PSM model transformation, we decided to shift from UML profiles to Domain Specific Languages (DSL) [14]. This decision was mainly based on technical matters. As a matter of fact, technology is playing a key role in the distinction between UML based and non-UML based tools. The facilities provided in the context of the EMP and other DSL frameworks, like the Generic Modelling Environment (GME) or the DSL Tools, have shifted the focus from UML profiles to MOF-based DSLs. Therefore, regarding existing technology for (meta-) modeling and model transformations, it seemed more convenient to express the new concepts related with XML schema modeling using a new DSL. To that end we have developed a MOF-based metamodel for XML Schema modeling.
XML Schema Metamodel. Supporting all the constructions defined by the standard resulted in a very complex metamodel. For the sake of space, Fig. 3 shows only some parts of it. But the way they are connected helps to understand the complete metamodel that you can find at [http://www.kybele.etsii.urjc.es/MtATL/](http://www.kybele.etsii.urjc.es/MtATL/).

![Fig. 3. Partial view of the XML Schema Metamodel](image)

As Fig. 3 shows, we included a pair of modifications regarding the standard. On the one hand, we have added some hierarchies. On the other hand, some classes include an election property. The type of this property will be the root class of one of the added hierarchies. This way, when we set the value of the election property, we are identifying which, among the different child classes, will be the instantiated class. These modifications help on easing the management of the metamodel.

Let’s show an example to better understand how these modifications work: the election property of the ElementGlobal says that its type will be an AbstractTypeLocal type. That is, it will be a ComplexTypeLocal XML element or a SimpleTypeLocal XML element. At the same time, the ComplexTypeLocal class owns an election property of AbstractContent type. This one has three children: SimpleContent, ComplexContent and Other. If we choose the latter, we can decide whether we will use a GroupRef, Sequence, Choice or All compositor. All together, the result is that the elements of a XML element whose type is ComplexTypeLocal, could be grouped using a Sequence, a Choice or an All compositor.

Finally, using different colours simplifies the task of identifying which hierarchy is used for defining the type of the election property in each specific case.
Annotation Metamodel. An annotation model includes a single-valued reference to the AnnotatedModel plus a set of annotation objects. Each annotation contains a single-valued reference to the model element plus a list of properties. The properties have an identification key and the corresponding value. The AnnotatedModelElement class acts as the proxy for the linked/annotated elements. That is, each record is merely a set of key-value pairs. The bottom of Fig. 4 shows the annotation metamodel used along with the core weaving metamodel [8] (top).

Fig. 4. Annotation Metamodel

3.2 PIM to PSM Transformation: design decisions

In the following we summarize the design decisions that can be taken to map a conceptual data model (PIM) to a XML Schema model (PSM). As we have already mentioned, in [4] we presented an initial implementation of such transformation. Here we modify some of the rules comprised in that initial version to allow the introduction of design decisions. Next, we focus only on those mapping rules. They are mainly related with the mapping of the properties of a class and the properties of a composition relationship.

- Class Properties: the mapping rule said that every class will be mapped to an ElementGlobal, which represents an element of the XML schema, plus a ComplexTypeLocal to define its type. The properties of that class are mapped to a sub-element (ElementLocal) of the ComplexTypeLocal. The designer can set the compositor used to group those ElementLocals: all, choice or sequence (Fig. 3). The semantics associated with each type of compositor is the following:
  - all: specifies that the child elements can appear in any order. Each child element can occur 0 or 1 time.
  - choice: allows only one of the elements contained in the declaration to be present within the containing element.
sequence: specifies that the child elements must appear in a sequence.
Each child element can occur from 0 to any number of times

Default behavior: the default compositor is sequence. The designer may modify
this behavior by adding an annotation to the UML class. That is, by adding an
annotation object in the weaving/annotation model. Such annotation will contain a
property object in the form \{key = Attribute, value = Choice\} or \{key = Attribute,
value = All\}.

Properties of a composition relationship: composition relationships are mapped
by including a sub-element within the ComplexTypeLocal element that maps the
“WHOLE” class of the composition. This sub-element will be also a
complexTypeLocal. It will include a set of XML sub-elements. They will map the
“PART” class of the composition. The designer may choose the compositor used to
group those sub-elements: all or sequence.

Default behavior: by default, the sequence compositor will be used. The designer
may modify this behavior by adding an annotation to the UML association. That is,
by adding an annotation object in the weaving/annotation model. Such annotation
will contain a property object in the form \{key = Association, value = All\}.

4 Case Study

In this section we use part of a case study to show the use of annotation models for
model-driven development of XML Schemas. The case study is an XML DB model
to store information about bibliographical references. We will start by defining the
UML class diagram (section 4.1) and we will show how the annotation model (section
4.2) drives the execution of the transformation to generate the desired XML schema
model.

Note that, once the conceptual data model is defined, the rest of the process is
automatic. In fact, the weaving model is optional. The ATL rules have been codified
to show a default behaviour if there is no annotation.

4.1 Conceptual Data Model

As shown in Fig 5, there are different types of bibliographical references: articles,
books, chapters, translations and thesis.

Each reference has a title, a reference type, a publication date and it may has been
written by more than one author and published by several publishers. In turn, a
publisher may publish several references and an author may appear in more than one
reference. Both, authors and publishers have a first name and a surname. The books
are composed of several chapters. Each chapter belongs to one book and it may have
been translated several times. Finally, each publication is composed of several
articles.

The figure is a screenshot of the conceptual model represented by a class diagram
using the Eclipse UML2 class diagrammer.
4.2 Annotation Model

Fig. 6 shows the weaving model used to annotate the previous class diagram. We added an annotation to the Publisher class. Such annotation contains a property (key = Attribute, value = Choice) that indicates that a choice element has to be used to map the properties of the UML class. Working this way, the designer may add an annotation to each class of the source model. The annotation sets the compositor (sequence, choice or all) used to map the properties of the class. If there is no annotation the default compositor is used (sequence).

4.3 Using annotations to parameterize the transformation

In this section, we show the ATL code for processing the annotations. To that end, we focus on the mapping of the Publisher class and its properties. This processing is
encoded in a set of rules for each type of compositor: sequence, choice and all, plus a set of auxiliary functions (helpers).

Fig. 7 shows the corresponding matching rules. For space reasons, here we show only those for using a sequence or a choice object, though the matching rule for using an all object is similar.

The guard of each rule checks which the decision of the designer was by calling the `mapTo()` helper.

```plaintext
rule Class2ElementGlobalSeq{
from c : UML!Class ((c.mapTo() = 'Sequences')and c.GetGeneralization().oclIsUndefined())
to xml : schemaXML!ElementGlobal |
  id <- c.name,
  name <- c.name + '<<ElementGlobal>>',
  Gener <- thisModule.package,
  eleccion <- cmpTyp,
  cmpTyp : schemaXML!ComplexTypeLocal |
  id <- c.name + '<<Type>',
  eleccion <- Other,
Other: schemaXML!Other |
  eleccion <- Seq,
Seq: schemaXML!Choice |
  eleccion <- Seq,
Seq: schemaXML!Sequences{
  }
}
rule Class2ElementGlobalChoice{
from c : UML!Class ((c.mapTo() = 'Choice')and c.GetGeneralization().oclIsUndefined())
to xml : schemaXML!ElementGlobal |
  id <- c.name,
  name <- c.name + '<<ElementGlobal>>',
  Gener <- thisModule.package,
  eleccion <- cmpTyp,
  cmpTyp : schemaXML!ComplexTypeLocal |
  id <- c.name + '<<Type>',
  eleccion <- Other,
Other: schemaXML!Other |
  eleccion <- Seq,
Seq: schemaXML!Choice {}
}
```

Fig. 7. Partial view of matching rules for mapping UML classes

As shown at the bottom of Fig. 8, the `mapTo()` helper returns the value of the designer decision by calling the `getLink()` and `getAnnotationValue()` helpers.

```plaintext
helper context UML!Class def: mapTo() : String =
  if self.getLink().oclIsUndefined() then
    "Sequences"
  else
    if self.getLink().getAnnotationValue("Attribute") = "Sequences" then
      "Sequences"
    else
      if self.getLink().getAnnotationValue("Attribute") = "Choice" then
        "Choice"
      else
        "All"
      endif
    endif
  endif;
```

Fig. 8. Helper `mapTo()`

The `getLink()` helper (Fig. 9) navigates the annotation model to return the annotation object referencing the particular property. In our case study, the annotation references the `Publisher` class. By calling the `getAnnotationValue()` helper (Fig. 10) over the
annotation object, the value of its Attribute property is returned. In this case, its value is Choice. So, the ElementLocal objects that will map the properties of the Publisher class will be grouped using a choice compositor.

![Fig. 9. Helper getLink()](image)

![Fig. 10. Helper getAnnotationValue()](image)

Finally, Fig. 11(a) shows the result of executing the parameterized transformation. The source models were the conceptual data model shown in Fig. 5 and the annotation model of Fig. 6.

![Fig. 11. Partial view of the XML schema model obtained: (a) using the annotation model. (b) default behavior](image)

The code for mapping the properties of a composition relationship is very similar. As well, we encoded a set of helpers and matching rules for each type of compositor: sequence and all.

Fig. 12 shows the matching rule. Again, for space reasons we show just the one for using a sequence object.

![Fig. 12. Partial view of matching rule for mapping UML composition relationships using a sequence object.](image)
5 Related works

Regarding previous works on this topic, there are two main lines to consider. On the one hand, at the end of 2000, several works focused on the use of UML to model XML Schemas. More specifically, they used UML class diagrams. Besides, they proposed to generate the XML Schema directly from the UML model [6, 7]. Working this way, the semantic gap between the abstraction levels considered is just too big. Moving from the conceptual data model to the source code is not recommendable. You will find that there are a lot of constructions that could not be obtained in the resulting code. For instance, all classes will be mapped using the same compositor. In real situations, where very complex models are used, this drawback is even more harmful. The generated XML Schema will not satisfy the needs of the designer. A language closer to the deployment platform is needed, i.e. something akin to a DSL for XML schema modelling.

A variation to this approach can be found at [19], where the mapping rules to obtain a UML model from an XML schema are defined. This proposal shows the same problem and it also lacks of any technical support.

Finally, there exists some more recent proposal focused on UML for XML Schema modelling. In [12] a comparison between them can be found. As a conclusion, we can say that none of them offer technical support.

Our proposal comprises a DSL for XML schema modelling, the mapping rules for moving from a conceptual data model to a XML Schema model, the code generation facilities to obtain the source code of the modelled Schema and the tooling to integrate these artefacts. In addition, the process can be customized by introducing some design decisions on the mapping. Moreover, in front of previous works, the one presented here is framed in a MDA framework. This fact results in additional advantages. For instance, right now we are developing the support to move from the XML technical space to the OR technical space.

6 Conclusion

In [4] we completed and automated our proposal for XML schema model-driven development. To that end, we defined a new metamodel for XML Schemas modelling, and we coded the M2M and M2T transformations needed.

This work has focused on the improvement one of those tasks: the transformation from conceptual data model (PIM) to XML Schema model (PSM). When validating the initial M2M implementation, we realised that we need to include certain design decisions in order to consider all the possible options when generating the XML Schema model. This article shows how we solved this problem using weaving models as annotation models. By using annotation models we can parameterize a M2M transformation without losing its generic nature. Furthermore, we are able to persist the design decisions that guided the development process through the use of models as the container for those design decisions.
The paper shows that the solution may be considered as quite simple. This is mainly due to the simplicity, the genericity and power of the AMW tool and its good coupling with the ATL model transformation solution.

The approach contributes to improve the accuracy and the quality of the models used at different stages of development as well as the subsequent code generated from them. These activities are especially important in proposals aligned with MDE because it proposes the models to be used as a mechanism to carry out the whole software development process.

At the present time we are working in two main directions. On the one hand, we are working to control entries that are mutually contradictory or inconsistent by adding OCL constraints at the metamodel.

On the other hand, we are working to support reverse engineering from the XML documents. We are defining the syntax of our XML Schema metamodel with TCS (Textual Concrete Syntax). Thus, one could not only extract an XML Schema from a model, but also inject an XML Schema to an XML Schema model.

Finally, we are working to apply the technique used here in the rest of the M2M transformations of M2DAT.

Acknowledgments This research has been carried out in the framework of the projects: MODEL-CAOS (TIN2008-03582/TIN), AGREEMENT-TECHNOLOGY (CSD2007-0022) both project financed by the Spanish Ministry of Education and Science and the IDONEO project (PAC08-0160-6141) financed by ‘‘Consejería de Ciencia y Tecnología de la Junta de Comunidades de Castilla-La Mancha’’.

References

Composing Models with Six Different Tools: a Comparative Study

Juan Pedro Silva, Miguel de Miguel, Javier F. Briones, Alejandro Alonso

DIT-UPM, ETSI de Telecomunicación, UPM, Avda. Complutense s/n, 28040 Madrid, Spain
{psilva, mmiguel, jbriones, aalonso}@dit.upm.es

Abstract. Model composition is the cornerstone for many settled and emerging software (and not only) development approaches, such as Model Driven Engineering, Aspect-Oriented Modeling and Software Design, Multi-Dimensional Separation of Concerns, etc. Although it could be performed manually by a modeler with a determined expertise, this would be an error-prone activity even for moderate-size models. Model composition tools, therefore, are of fundamental importance for supporting model-driven development. In this paper we compare six different model composition tools: ATL, AMW, XWeave, Kompose, Theme/UML, and Epsilon’s EML, evaluating each one’s merits and shortcomings after using them in a real-world example.

1 Introduction

According to the Wikipedia [1], a Model Composition Tool is a tool used to compose (i.e. to merge according to a given composition semantics) several source models. In the words of [2], “model composition refers to the process of combining or integrating two or more abstract representations of a system of interest”. This goes hand in hand with the concept of Separation of Concerns (SoC) [3], a concept basic (not only) to software engineering. Subsequently, model composition is a fundamental piece in many existent approaches: Model Driven Engineering (MDE), Aspect-Oriented Software Design (AOSD), Multi-Dimensional Separation of Concerns (MDSoC), and more to come.

For [4], in the particular case of aspect-oriented modeling (AOM), a design is presented in terms of multiple user-defined views (aspects) and model composition is often carried out to obtain a model that provides an integrated view of the design. This conception of the design of the system can be extended to other areas outside of AOM with the same validity. In architectural viewpoints, for instance, different stakeholders (i.e. security engineers, software architects, developers) may need or prefer to work on a particular view of the system, according to the concerns they are focusing on. Those views may contain redundant overlapping information, but each view incorporates new concepts to the whole system model. Model composition comes into play when combining two or more of these partial models into one complete model to be used for simulation, test, or code generation [5].
To date, there are many published articles on the subject of model composition tools ([6], [7], [8], to name a few), but, to the best of the authors’ knowledge, this is the first comparative study made of available tools.

The paper is structured as follows. In this section an introduction to the universe of model composition is provided. Section 2 deals with the evaluation framework, introducing the selected criterion, and presenting the simple use case that will be used in this article for illustration purposes. A brief description of the different tools is provided in section 3. Afterwards, a description of the implementation of the presented use case with each one of the tools is included in section 4. The results obtained from the evaluation are presented in section 5, and the paper concludes with section 6 providing a summary and discussion.

2 Evaluation Framework

2.1.1 Basic Vocabulary

There are some concepts used in this article which the reader should be familiar with before he continues reading. This section introduces such concepts.

Model composition appears in numerous development methodologies, each one possibly having a different vocabulary. The terminology employed in this paper, aspect-oriented terminology predominantly, will be introduced in this section. The definitions presented here are taken from [9].

Concern and Aspect. A concern is any piece of interest or focus in a program, a particular set of features or behaviours that appear in a system. An aspect is a part of a program that affects its core concerns (often referred to as “cross-cuts”, meaning that it repeatedly appears in more than one concern).

Base and Aspect Models. On multi-dimensional separation of concerns, you normally have a different model for each concern, all of with are composed together to generate the resulting system model. If it is a symmetric approach, there is no distinction between models. However, in aspect-oriented software design, as well as in other asymmetric approaches, the core system concerns are modelled in a model called “base model”, while crosscutting concerns are modelled in one or more “aspect models”. The later terminology is used in this article to indicate that one model (the aspect model) contains the modifications to be performed on the original model (the base model).

JoinPoints and Pointcut. Modifications in the base model take place in a number of places (or points). Each tool supports a determined set of points, which could be fix or specified by the developer, depending on the tool. These points are called join points. A pointcut is a set of join points

Name matching refers to the use of the name of the model elements to establish a correspondence between them. If a model element in the aspect model has a corresponding element in the base model (corresponding means that both name and type are equal) the elements are merged.

Guards are conditions imposed for elements to match. For example, checking that two elements of the same type are equally named is a guard for name matching.
2.2 Evaluation Criteria

In order to obtain a useful comparison of the different tools, a criterion has to be stated. The different concepts to be considered, in no particular order, are:

- **Graphical Composition.** If the tool provides means for graphically specifying the composition.
- **Input Flexibility.** Evaluates the requirements a particular tool places on the input models (correspondence to a particular metamodel or modeling methodology, to the same metamodel for all models, or limitations on the number of input models).
- **Necessary Knowledge to use.** The different modeling techniques or programming/transformation languages the user needs to know to use the tool adequately.
- **Composition Rules Flexibility.** Represents the possibility of incorporating user-defined rules, and the flexibility of customizing the rules that come with the tool.
- **Stability and Robustness of the Tool.**
- **Available Documentation.** To use and to extend the tool.
- **Situation and Evolution of the Tool.** If the tool has a published evolution plan and milestones. If there is an active community around it.

2.3 A Simple Use Case to Visualize the use of the Tools

The use case under which the tools where evaluated, described in [10], considers the composition of access control considerations and security policies into a functional model, all three corresponding to different metamodels. Such composition was prototyped and tested with an online banking example.

Although the tools have been tried and evaluated with a real life example, for the sake of brevity it will not be presented in this paper. On the other hand, a very simple but representative use case will be defined here to illustrate and visualize the use of the different tools.

![Simple Use Case Metamodel](image-url)
Figure 2 – Base, Aspect, and Desired Result Models of the Simple Use Case

The illustrating example to be presented should demonstrate the use of the composition tool to add, modify, and delete model elements. The selected use case for this article is not more than a static structural modification of a model. The metamodel for the inputs (in this use case, base and aspect correspond to the same metamodel, as some tools don’t allow the use of different metamodels) is presented in Figure 1. The input base model, illustrated in Figure 2, is composed by an interface with a unique operation: *OperationToDelete*. The desired modifications are:

- Add two operations which name is “FirstOperation” and “SecondOperation”, prepending the name of the interface.
- Delete all the previously existent operations in the interface (*OperationToDelete*).
- Add a new component that also implements “AInterface”.

The aspect model that shows the additions is also presented in Figure 2, as well as with the desired result.

3 Evaluated Set of Tools

Table 1 enumerates the technical details of the tools under evaluation. The following sub-sections provide a brief introduction to each of the evaluated tools.

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>Platform Dependencies</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATL</td>
<td>2.0.1</td>
<td>Eclipse</td>
<td>ATL Editor, Handlers for EMF and UML2.</td>
</tr>
<tr>
<td>AMW</td>
<td>1.0.0</td>
<td>Eclipse, ATL, AM3</td>
<td>Wizard, weaving editor, extensions, uses ATL editor.</td>
</tr>
<tr>
<td>Kompose</td>
<td>0.0.3.1</td>
<td>Eclipse, Kermeta</td>
<td>Contextual menus, uses Kermeta editors.</td>
</tr>
<tr>
<td>XWeave</td>
<td>4.3.1</td>
<td>Eclipse, oAW</td>
<td>Pointcut language, uses oAW editors.</td>
</tr>
<tr>
<td>Theme/UML</td>
<td>1.0.4</td>
<td>Eclipse</td>
<td>Profile, contextual menus.</td>
</tr>
<tr>
<td>EML</td>
<td>0.8.6</td>
<td>Eclipse, Epsilon</td>
<td>EML, EOL, ECL editors, ModeLink editor.</td>
</tr>
</tbody>
</table>
3.1 ATL [11]

The Atlas Transformation Language (ATL) is a general purpose transformation language, hybrid of declarative and imperative programming, created by the ATLAS INRIA & LINA research group as an answer to the OMG MOF/QVT RFP. It is a model transformation language specified as both a metamodel and a textual concrete syntax. In the field of Model-Driven Engineering (MDE), ATL provides developers with means to specify how to produce a number of target models from a set of source models. [12]

3.2 AMW [13]

The Atlas Model Weaver (AMW) is the Atlas tool specifically focused on model composition. The AMW tool allows the specification of different kinds of links between model (or metamodel) elements, which are then saved in a weaving model, which conforms to an extensible abstract weaving metamodel provided by the tool.

3.3 Kompose [14]

Kompose is an open-source generic model composition tool that runs on top of Kermeta [15], an object and aspect oriented metamodeling language which allows describing both the structure and the behaviour of models.

In the words of [16], Kompose is a generic framework to support model composition, implementing a generic structural composition operator that can be specialized to any particular modeling language described by a metamodel. The core composition mechanism (implemented in Kermeta as a separate metamodel) can be specialized for each specific domain.

3.4 XWeave

XWeave is a model weaver running on top of openArchitectureWare (oAW) [17] that supports weaving of models in an asymmetric aspect-oriented fashion. XWeave takes a base model and a aspect models as input and weaves the content of the aspect model into the base model.

XWeave pointcuts can be specified by name matching, and by explicit pointcut expressions, using the specific, and yet simple, oAW expression language [18].

3.5 Theme/UML Profile and Eclipse Plugin [19]

“Theme” [20] is one of the most established and documented aspect-oriented approaches. The Theme Approach is an aspect-oriented methodology that encompasses the requirements analysis, design and mapping to implementation phases of the development lifecycle [19]. Theme/UML is the modelling language used to represent the concerns and the relationships among the different concerns.

Model-Driven Theme/UML is a toolset for model-driven engineering of Theme/UML designs and automatic transformations to composed models and source
code. It is somehow awkward that, although Theme/UML has been around for many years\(^1\), and the existence of supporting tools is reported ([21], [22]), such tools have not been available to the open public until very recently\(^2\).

### 3.6 Epsilon Merging Language (EML) [23]

The Epsilon Merging Language (EML), part of the Epsilon platform, is a rule-based language for merging models of diverse metamodels, after first identifying their correspondences [24]. For identifying such correspondences, Epsilon provides the Epsilon Comparison Language (ECL). These two are task-specific derivations of the Epsilon Object Language (EOL), an imperative model-oriented procedural language based on OCL. Epsilon provides a family of metamodel-agnostic languages for creating, querying and modifying EMF (and other types of) models in various ways.

### 4 Implementing the Use Case with the Different Tools

#### 4.1.1 Implementation with ATL

ATL imposes no restrictions on the input models: they need not correspond to the same metamodel, neither have to be “clean” (meaning they have no references to primitive types libraries, or elements in other metamodels, or profile applications). The only consideration to have is to include in the ATL transformation all referenced metamodels or profiles as inputs. For an introduction to ATL, please refer to [12].

An ATL transformation produces n output models from n input models. In this case, two input models (base and aspect) generate one output model. Figure 2 shows the models used for this implementation, and an excerpt of the resulting ATL transformation is shown in Listing 1.

There is no standard way in ATL to indicate that a particular element (class, operation, attribute, etc.) should be deleted. Not copying such element into the output model (which is the equivalent to including the semantics of this modification in the transformation code) is enough.

#### 4.1.2 Implementation with AMW

AMW’s implementation demands a little more work. For AMW, you have to:

- Design an extension of the abstract weaving metamodel [25] provided by AMW, or use a pre-existent extension.
- Implement the semantics of such extension in ATL. These semantics are generic and can be reused among all weaving models of this kind.
- Create a weaving model instance of this extension, taking the base and aspect models as inputs.

---

\(^1\) According to [20], the first ideas of the approach were introduced by Siobhán Clarke in 1997, although the name “Theme” wasn’t used until 2001.

\(^2\) The reported date of publishing of the Theme/UML Eclipse plug-in on the site is 06/02/09.
Composing Models with Six Different Tools: a Comparative Study

Listing 1 – Excerpt of the ATL Transformation

```plaintext
Listing 2 – ATL Transformation for AMW

Figure 3 – Weaving Model in AMW
```

The input models are the ones presented in Figure 2. The weaving metamodel extension designed for this use case has two new weaving associations: AddAndRenameOperations (adds operations to a class, and renames them...
correspondingly) and DeleteOperation (deletes an operation), shown in the weaving model instance in Figure 3. The implementation of the semantics of these new weaving actions are introduced in the ATL transformation presented in Listing 2.

An ATL transformation needs to be executed in order to compose the models. This transformation takes the AMW weaving model and the base and aspect models as inputs. There are two possible ways in which to address such transformation: to match the weaving-operations occurrences, or to use them as guards in other rules. The implementation presented here uses this second approach: the rule matches all operations in base and aspect models, but for each one it checks if it “isToBeDeleted”, or if it “isToAddAndRename”.

4.1.3 Implementation with Kompose

Kompose, in its current state, imposes some restrictions: base and aspect models have to correspond to the same metamodel and only name matching is supported. This last restriction has two important consequences, remarked in section 5.1.3.

For composing models with Kompose, the different artifacts to be developed are:
- A Kompose specialization for the particular input metamodel (Kompose includes two specializations, one for Ecore models, and other for a database schema model).
- A ‘.kompt’ file with the weaving instructions in Kermeta language. The resulting ‘.kompt’ file for the implementation with Kompose is included in Listing 3.
- If the metamodel references other metamodels/libraries, these have to be specifically imported in the Kompose specialization.

For the generation of the Kompose specialization a wizard is provided, and then you have to modify the generated Kermeta file to add in it the metamodel objects to complete the task. This approach does not scale well for large metamodels.

Kompose offers pre and post processing instructions blocks that allows, among other things, renaming model elements for name matching, or deleting elements. Such feature allows the use of a generic aspect model, encouraging reuse.

4.1.4 Implementation with XWeave

XWeave allows name matching, so the same input models presented in Figure 2 could have been used. However, a new aspect model that makes use of oAW expression language was created, in order to illustrate its use.

The new aspect model is presented in Figure 4. This aspect model makes use of two operators of the pointcut language: “%” functions, and “?” functions. These functions must be defined in external “extend” files within the classpath. Percent-sign functions return a collection of objects that represent the resulting pointcut. Dollar-sign functions, in turn, return a string, and are used for generating elements’ names.

---

3 The “*” wildcard was tried out, but it resulted in duplicated elements, probably caused by some bug in the tool.
There is currently no means in XWeave to delete model elements. To do this, the oAW platform provides a companion tool called XVar. The only documentation currently available on XVar is a video [26]. The use of the XVar tool differs from that of XWeave, and will not be commented here.

The produced artifacts for XWeave, all presented in Figure 4, are:

- A new aspect model that makes use of the "%" and "$" functions (marked A).
- An extension file, implementing those functions (B).
- An OAW workflow indicating the input (base and aspect) and output models (C).
**4.1.5 Implementation with Theme/UML**

The Theme/UML tool requires, quite logically, that its inputs follow the Theme/UML methodology. This implies having to modify the models. Due to the characteristics of the particular composition used in this example, the most suitable approach is to include both base and aspect models as two “base themes” in the same Theme/UML model. Theme/UML supports name matching or matching-indicating associations between elements of the same type. Name matching was the used approach in this occasion.

Theme/UML provides two composition mechanisms: merge and override. A dependency stereotyped “override” (marked as “A” in Figure 5) between the two themes indicates the composition mechanism to execute. This stereotype provides a “delete” tag value (marked as “B” in Figure 5) that allows stating which elements are to be deleted. The final implementation then consists of a UML model with two base “theme” packages, one each (base and aspect) models in Figure 2 respectively. Theme/UML provides no means, however, of renaming the operations.

Running the composition is simply done by selecting the appropriate option in the Theme/UML contextual menu. The resulting Theme/UML model is presented in Figure 5.

**4.1.6 Implementation with EML**

The implementation with EML resembles the one in ATL or AMW because both platforms are, in many ways, similar (both approach things similarly, both languages are OCL-based, similar rule’s structure and syntax, etc.). In EML, the implementation takes two steps. First, you create an ECL file stating the correspondences between the elements (similar to AMW’s weaving model, but textual). You then create an EML file to specify the semantics of the composition in merging and transformation rules.

There is one other possible strategy for model composition in Epsilon. Epsilon provides, among other tools, one called “ModeLink”, an editor (very similar to AMW’s weaving editor) for creating associations between different models. Just as in AMW, a metamodel specifying the associations’ types to be used in ModeLink has to be created. ModeLink, in contrast with AMW’s weaving editor, only supports two or three panels (AMW’s imposes no limit, although is not clear how usable it would be with more than a few panels). The model resulting from the use of ModeLink (see Figure 6) could be fed to an EML transformation holding the semantics. This strategy is almost identical to that of AMW. Regretfully, at present there is no possibility of generating an ECL matching from ModeLink, so the editor cannot be used with EML.

**5 Results of the Evaluation**

Table 2 and Table 3 give a summary of the results of applying the evaluation criteria to the selected tools. Each tool has its own pros and cons. However, there are some common characteristics, for example that documentation is not abundant in any of them. The following subsections describe the findings for each tool independently.
Table 2 – Evaluation Summary, First Part

<table>
<thead>
<tr>
<th>Tool</th>
<th>Graphical Composition</th>
<th>N° of Input models</th>
<th>Mandates Input Format</th>
<th>Allows ≠ MM</th>
<th>User Defined</th>
<th>Customizable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATL</td>
<td>NO</td>
<td>No limits</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>AMW</td>
<td>YES</td>
<td>No limits</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Kompose</td>
<td>NO</td>
<td>2</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>XWeave</td>
<td>YES</td>
<td>2</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Theme/UML</td>
<td>YES</td>
<td>No limits</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Epsilon EML</td>
<td>Could be</td>
<td>No limits</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>

Table 3 - Evaluation Summary, Second Part

<table>
<thead>
<tr>
<th>Tool</th>
<th>Required Modelling Methodology</th>
<th>Requires Programming Language</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATL</td>
<td>NO</td>
<td>YES (ATL)</td>
<td>Multiple</td>
</tr>
<tr>
<td>AMW</td>
<td>Weaving Models</td>
<td>YES (ATL)</td>
<td>Userguide, Use Cases.</td>
</tr>
<tr>
<td>Kompose</td>
<td>Name Matching</td>
<td>YES (KERMETA)</td>
<td>Video, Examples.</td>
</tr>
<tr>
<td>XWeave</td>
<td>EMF-Like + Poincts</td>
<td>POSSIBLY (XTEND)</td>
<td>Userguide, Video.</td>
</tr>
<tr>
<td>Theme/UML</td>
<td>Theme/UML</td>
<td>NO</td>
<td>Book, Video, Guide.</td>
</tr>
<tr>
<td>Epsilon EML</td>
<td>NO</td>
<td>YES (EOL, ECL, EML)</td>
<td>Book, Videos.</td>
</tr>
</tbody>
</table>
5.1.1 ATL

ATL is a generic transformation language with no special support for model weaving. Accordingly, it does not provide graphical means for model composition.

On the other side, it is one of the most flexible tools, not imposing any requirements on the input models: not on the modeling methodology, neither on the use of the same metamodel, nor on the number of input models. Furthermore, this flexibility also exists among the weaving rules, as the user can specify almost any composition rule in ATL. Besides, it allows for the definition of rules’ libraries, which can be extended and redefined, facilitating its adaptation to the specific case.

With respect to available documentation, ATL is the best documented tool subject to evaluation in this article. Wiki pages, user guide, developer guide, specification of the ATL Virtual Machine, starter’s guide, an active newsgroup compose this documentation. ATL also has published milestones as part of the M2M project plan. However, sometimes such documentation is not up-to-date, and users need to wander through different documentation sources to find the necessary information.

On the other hand, it is not fair to compare its documentation with that of the rest of the tools (perhaps only with Epsilon’s), being ATL a generic transformation framework, and not specifically a model composition tool. When compared to the documentation of similar products, such as oAW or Kermeta, ATL documentation is not well organized, being dispersed over many different places.

The only required knowledge to use ATL as a composition tool is the ATL language itself and, out from the authors’ experience, the use of ATL resulted in being relatively fast and simple, probably thanks to its documentation. It probes itself to be a stable tool. Leveraging its metamodel to provide graphical editors for defining transformations would definitely be a great enhancement for ATL.

5.1.2 AMW

Being AMW the AMMA [27] tool targeted specifically on model composition, it leverages most of ATL strengths, while addressing some of its shortcomings. For instance, AMW provides a graphical editor for modeling compositions, while maintaining the flexibility of not limiting the number of input models, or the correspondence to different metamodels. Still, some of the available features packed with AMW (automatic element matching, for instance), are limited to two input models, and the metamodels have to be clean (no dependencies on other metamodels).

Likewise, the flexibility of being able to define its own merging rules is maintained, although ATL is still necessary (for defining the semantics of the rules and running the weaving). Using AMW also implies understanding and extending the abstract weaving metamodel, and learning how to use the weaving editor, although the last is quite simple. On the whole, using AMW for the first time is a little more complicated than using ATL, but it simplifies things in the long run.

As previously mentioned, AMW provides some useful features that come with the tool (element matching through different matching algorithms, for instance). The advanced features also include the ability to automatically generate weaving rules between models. Nevertheless, using this advanced feature demands some further expertise from the user.
AMW’s basic use is well documented and has a number of available use cases and examples. However, deeper documentation on some advanced features and extending the tool would be desirable. Future plans focus on improving integration with Eclipse.

From the point of view of the authors, AMW is a well balanced, stable tool that provides support for many advanced model composition operations. Using it requires some ATL expertise, as well as being confident dealing with metamodels. It is not the simplest of the evaluated tools, but is among the most powerful ones.

5.1.3 Kompose

Kompose is the Kermeta framework solution for model composition. It is code-based, not providing any graphical means for specifying the composition directives of the models, and limiting its inputs to two models from the same metamodel.

Kompose only supports name matching, but allows changing element names from within the pre and post directives. As a result, for each model element you wish to modify, one matching element must be included in the aspect model. Aside from simple models, this doesn’t scale well and is impractical for recurring modifications.

There is no available documentation on Kompose, but an exemplary video. There is, however, a relatively active mailing list in which questions are readily answered. Documentation has been promised for the short term, but no date has been provided.

This can complicate its use, primarily in generating specializations for the metamodels, an activity for which there are only two examples.

Kompose appears to be an interesting tool which is taking its first steps (version 0.0.4 just released at the time of submitting this paper). In the authors’ opinion, it promises to be an interesting tool in the future, and it would improve a lot if extended with a user interface and a pointcut language. These are planned features, but no deadlines have been imposed for them. Kermeta’s main focus is on creating models that include behaviour and, although Kompose claims to be a generic model-composition tool, its features and evolution direction may be driven by its supporting infrastructure’s objectives.

5.1.4 XWeave

Perhaps the easiest tool to use among the ones under evaluation, XWeave is an aspect-oriented model composition tool with a clear asymmetrical distinction between base and aspect models. By means of a very simple, but still useful pointcut language, XWeave allows for aspect reuse and facilitates model elements modification.

Although limited to two input models from the same metamodel, the greatest limitation imposed by XWeave is having to use a different companion tool for removing elements or features from the models.

Using XWeave for model composition requires knowing how to use the oAW’s workflow for driving the composition. In the case the aspect models make use of the pointcut language, there is also need to create an extension file in Xtend language.

Available documentation is scarce, limited to an exemplary video, and a brief user guide packed with the tool. However, this documentation is enough to get going, and
any further doubts can be posted on the oAW forum. In the case of XVar, oAW’s tool for removing elements or features, the only documentation is an exemplary video.

The impression caused by the use of XWeave is that of a simple, very specific tool, focused on aspect-weaving, but with an interesting approach. The pointcut language is very basic but at the same time productive and powerful enough for most cases.

Nevertheless, the results presented some problems, leaving the impression that the tool still needs a little tweaking. Maintenance of the tool is somehow stalled, as the developers are currently implicated in some other projects, and activity around the tool is sparse within the forum. The authors’ feeling towards this tool is that it is a proof of concept with potential, but it remains to be evolved.

5.1.5 Theme/UML

The impression obtained from using the Theme/UML tool is that it is a first implementation with some bugs to correct. The Theme approach itself is quite stable and complete, but in the case of the tool, even this simple use case couldn’t be completed in its actual state. The published profile has two properties with no specified type, one of which is the “delete” tag value of the “override” stereotype, which prevents from using it. “Merge” compositions, on the contrary, work correctly.

The remaining limitations of the tool are similar to those of the rest of the tools under evaluation. In this case, there is no limitation on the number of input “themes” to compose, but it mandates the use of the Theme approach (it is only oriented to this approach in reality). The input metamodel must, therefore, always be UML plus the Theme/UML profile.

The Theme methodology, the only required knowledge to use this tool, is broadly documented, and a video, user and developer’s guides are provided for the tool. No evolution plans have been reported.

5.1.6 Epsilon/EML

Conclusions on Epsilon/EML are very similar to those of ATL and AMW, as both tools are very similar indeed. Just like ATL/AMW, Epsilon and EML don’t impose any limitations on the number of input models or metamodels, and provides for rule extension and redefinition. EML’s syntax is specifically focused on model merging. This facilitates the creation of composition rules. The provided editors have more functionality than those of ATL (for a comparison with a very similar tool). Consequently, it could be said that Epsilon provides greater “user-friendliness”.

Conversely, the ModelLink tool is not leveraged to be used as an input for model composition (ATL provides operations for fetching the elements referred by the weaving links; there is no equivalent in Epsilon EOL, EML, or ECL). Automatically deriving ECL matches from ModelLink would be a great addition to Epsilon.

Available documentation is adequate, providing some examples, and screen casts on the use of the tools. More importantly, there is a book [24] that describes the different languages provided by Epsilon, although it is more focused on the description of the languages than how to use them in conjunction for some objective. A cookbook, or at least some recipes or cheat sheets would be nice to have.
Composing models in Epsilon implies learning how to use EOL, ECL, and EML. Those are very similar languages, so the learning curve shouldn’t be steep. The use of the ModeLink editor is intuitive, and the included screen casts clarifies most doubts.

Regarding Epsilon’s evolution plans, the languages are considered stable in their current state. The platform has plans on improving its debugging facilities, supporting additional modeling technologies and improving the languages’ scalability and performance. There are also plans for integrating ECL with EMF Compare [28].

All in all, EML offers a good balance between flexibility, usability, and potential.

6 Summary and Discussion

This article describes the strengths and shortcomings of six different model composition tools under study. It is, to the knowledge of the authors, the first comparative study published on the specific subject of model composition tools.

As can be inferred from Table 2 and Table 3, some of the evaluated tools have limitations, for instance, on the number of input models, or input metamodels to be used, while others allow greater flexibility in those aspects. However, most times such flexibility comes hand in hand with greater complexity.

Although different in many aspects, the evaluated tools share a common concern: none of them is extensively documented. This prevents greater adoption from the general audience. This can be one of the causes of the not abundant activity (in newsgroups, forums, etc.) around them, another common characteristic.

Another sensible deficiency is the debugging capabilities of the tools. In most occasions, composition’s developers will find themselves doing “print to screen” debugging. Moreover, error messages are generally cryptic, not helping much to understand the problem. There is a lot of work to do in this area.

By the same token, most tools and languages provide few or no graphical means for performing the composition and for working on the models. Having to write code for managing the models somehow hinders the adoption of the model-driven methodologies. There is room for improvement in this aspect.

In general, leaving aside the great differences in the maturity level of the evaluated tools, the overall sensation is that model composition tools are prototypes of published research. They don’t give the feeling of being ready for industrial use. In the final analysis, the sum of this remarks lead to the conclusion that model composition, even though it has been around for some time, is still in the cutting edge, not being an stable, widespread and established activity within the industry so far.

Acknowledgements

The authors would like to thank all the people in the different newsgroups who answered uncountable questions, especially Ed Merks, Marcos Didonet Del Fabro, Éamonn Linehan, Jacques Klein, Mickael Clavreul, Markus Voelter, Didier Vojtisek, Dimitris Kolovos, and so many others. They would also like to thank the reviewers for their comments that help improve the manuscript.
References

17. openArchitectureWare (oAW), http://www.openarchitectureware.org/
22. AOSD.08 Demos, D5: Model-Driven Theme/UML. Brussels (2008)
Achieving QVTO & ATL Interoperability
An Experience Report on the Realization of a QVTO to ATL Compiler

Alfons Laarman
a.w.laarman@cs.utwente.nl

Software Engineering Group, University of Twente, The Netherlands

Abstract. With the emergence of a number of model transformation languages the need for interoperability among them increases. The degree at which this interoperability can be achieved between two given languages depends heavily on their paradigms (declarative vs imperative). Previous studies have indicated that the QVT and ATL languages are compatible. In this paper we study the possibility to compile QVT Operational to the ATL virtual machine. We describe our experience of developing such a compiler. The resulting compiled QVT transformations can run on top of existing ATL tools. Thereby we achieve not only QVT/ATL interoperability but also QVT conformance for the ATL tools as defined in the QVT specification.

1 Introduction

Model Driven Engineering (MDE) is an emerging approach for software development gaining more and more attention by the industry and the academia. MDE emphasizes the need for thorough modeling of software systems before their implementation. Implementations can be derived from their models by applying model transformations, possibly in a fully automated way.

Since the adaptation of MDE, several model transformation languages have been defined. As their tools grow more mainstream, the need for interoperability among them increases. Each of these languages has its own application domain and therefore software developers should be able to compare and select the languages for their particular problem. In some cases it might be desirable to combine several transformations in different languages with each other. The degree at which this interoperability can be achieved depends heavily on the language paradigm (declarative vs imperative) [10].

QVT is a de facto standard specification for model transformations with multiple possibilities for implementation. Current tools employ various approaches like compilation to Java (SmartQVT [15]), interpretation (ProceduralQVT [14]), etc. However, the use of these implementations is limited because of their youth [12]. Thus, for practical reasons a user might want to use other languages with better tool support, for example ATL, which is based on a virtual machine architecture. It may be beneficial to compile QVT programs to the ATL virtual
machine. Potential benefits of this form of interoperability are: reusing QVT programs on top of the ATL virtual machine, reusing ATL tools, claiming QVT compliance for the ATL tools and comparing ATL and QVT programs.

In this work, we report our experience of implementing a QVT to ATL compiler. We describe here a design for the compiler, which has been implemented as a proof of concept. This solution will provide significant QVT/ATL interoperability and will prove ATL tools to be QVT Operational conformant.

In the next section, we will state the problem with the current situation in MDE. The following sections will provide background for QVT, ATL and associated tools and languages. In Section 4, the approach, we explain the development method and the technology used to implement the compiler followed by the description of the design and implementation. To test the compiler some example transformations were used. These examples are presented in the discussion section together with the problems we encountered during implementation and with future work. In the last section, we draw conclusions.

2 Problem Statement

With the emergence of a number of model transformation languages the need for interoperability among them increases. Each of these languages has an own application domain and therefore software developers should be able to compare and select the languages for their particular problem. In some cases it might be desirable to combine several transformations in different languages with each other. The degree at which this interoperability can be achieved depends heavily on the language paradigm (declarative vs imperative) [10].

Interoperability between QVT and ATL is desirable for additional reasons: QVT is a de facto standard for model transformations and ATL is not only designed to support QVT transformation scenarios but goes beyond QVT context. ATL does this by supporting scenarios where source and target models are artifacts created with various technologies such as databases, XML documents, etc [12]. Furthermore ATL provides tool support, whereas direct QVT implementations are still in a phase of infancy [12].

Both ATL and QVT specify several languages distributed across multiple layers. Based on language features Czarnecki and Helsen describe similarities between ATL and QVT [5]. Jouault and Kurtev believe ATL and QVT to be interoperable with each other [10]. They made a detailed comparison between the languages using following language properties:

- Relative abstraction level
- Transformation scenarios (model transformation, synchronization and conformance checking)
- Paradigm (declarative, imperative or both)
- Directionality (multidirectional or unidirectional)
- Cardinality (M-to-N or M-to-1)
- Traceability (automatic or user-specified)
There are three different QVT languages, which all differ from the ATL language according to these criteria. Finally they conclude that QVT Operational (one of the three languages specified by [4]) has the highest potential to be interoperable with ATL.

Thus a transformation from QVT Operational (QVTO) to ATL virtual machine can be implemented at relatively low cost. Concrete implementation details for such a transformation are however not provided in [10]. Therefore the hypotheses of [10] still require investigation. We provided this proof by realizing a compiler implementation.

3 Background

3.1 QVT Architecture

The Meta Object Facility Query/View/Transformation (MOF QVT) specification [4] is the solution for model transformations in the OMG modeling framework. It is designed to be a standard and does not provide a reference implementation. In the language dimension of the QVT specification we find three different languages: QVTO, QVT Relations (QVTR) and QVT Core. Next to the language dimension a conformance dimension is defined. A tool designer can use this dimension to give a degree of QVT conformance to his transformation language implementation.

QVTO is a completely imperative language, which only supports model transformation scenarios in an unidirectional M-to-N fashion. Mapping operations (see Listing 1) perform the central task of producing output model elements from input model elements. They can be defined on model elements making in an object-oriented language.

```
1 transformation Uml2Rdb(in srcModel:UML, out destModel:RDBMS);
2 main() {
3   srcModel.objects[Package]->map package2schema();
4 }
5 mapping Package::package2schema() : Schema { .... }

Listing 1. Example QVTO code
```

Traceability is automatic in QVTO [10]. Several language constructs are provided to request trace information in the transformation design. But the execution semantics also rely on the traceability information. For example, if an operation has been executed before with the same input parameters, the second execution will not create any new target model elements, instead it will return those already created. The abstractness of the language is almost at the same level as ATL but probably a bit lower [10].

3.2 ATL Architecture

AtlanMod Transformation Language (ATL) is a modeling platform (language and tools) developed by AtlanMod (INRIA-EMN) [11,14]. The architecture of
ATL mimics the Java Virtual Machine (VM) architecture. First the ATL program is parsed into a model representation, then this is compiled into the assembly format which can be executed by the ATLVM (see Fig. 1).

![Fig. 1. ATL compilation process (taken from [14])](image)

ATLVM executes an assembly language much like Java byte code and handles models on the basic level of querying and creating model elements [13]. Its paradigm is imperative. Assemblies can contain (virtual) operations which can be defined on model elements (their context), which makes the VM object oriented. The data types handled by the ATLVM consist of the basic primitive types, composite types like in OCL [6] and model element types. The ATLVM is stack-based and understands three kinds of instructions; Operand stack (push, pop, load, swap and store), Control (if, goto, iterate and call) and Model handling (create, fetch and get).

The ATLVM is the basis of the current ATL implementation and is also the target language of the compiler described in this paper. Listing 2 shows an example ATLVM operation named “container” on the element Object from the QVT metamodel. When called, this operation will load the contextual parameter on the stack (line 2) which will subsequently be used as contextual parameter of the operation call “refImmediateComposite()” (line 3).

```
1 context QVT! Object def : container () {
2    load self;
3    call 'J.refImmediateComposite () : J';
4 }
```

Listing 2. Example ATLVM code

### 3.3 DSL Support from ATL

The AtlanMod Model Management Architecture (AMMA), which ATL is part of, also contains some tools for the creation of domain specific languages (DSLs). Language creation can be done by expressing a syntax (abstract and concrete) and semantics. AMMA solves these subtasks using MDE [7]:

122
the abstract syntax is captured in a metamodel,
– the concrete syntax is represented as a transformation which maps the concrete syntax elements to the abstract syntax,
– and the semantics for DSL\textsubscript{A} can be expressed as a transformation mapping DSL\textsubscript{A} constructs to those of an existing DSL\textsubscript{B}.

For each of these, AMMA provides a dedicated language:
– KM3 provides a formalized way to specify metamodels with only the most basic concepts [8],
– TCS can specify the textual concrete syntax of a language in terms of the KM3 metamodel [9],
– and lastly ATLVM Code Generator (ACG) is a transformation language which maps the KM3 metamodel elements onto the ATLVM.

The ATL language is also written in KM3, TCS and ACG [7]. The process of executing an ATL definition involves two steps: (1) parsing of the definition’s syntax using TCS and (2) running the ACG definition on the resulting model (which conforms to the ATL metamodel as expressed in KM3).

Contrary to what Fig. 1 suggests, the ACG definition is not executed directly. Fig. 2 shows how ACG definitions are compiled to (ATLVM) assemblies. For this reason the performance of compiling ATL transformation definitions is more than satisfactory. Even on the largest existing ATL transformations [3], the compilation is done almost instantly when the definition is edited and saved.

4 Approach

By implementing a QVTO to ATL compiler, we can achieve interoperability and establish ATL conformance to QVT. Such a compiler can be built using any
compiler design framework/technology of preference. In the previous section, we showed how the AMMA tools can be used to define DSLs that can be executed in ATL environments (read: on the ATLVM). Since these tools have proven to provide a satisfactory performance for ATL and integrate needly into the MDE environment, they are the primary candidate for implementing the compiler.

With AMMA the implementation comes down to defining a metamodel and grammar for the syntax and expressing the semantics in ACG. It is a straightforward task to define the QVTO syntax in KM3 and TCS. In fact, we even skipped implementing the syntax and used the parser and metamodel from SmartQVT [15], which conforms to the QVT specification.

Thus, we only had to implement the semantics in ACG and use that as the QVT transformation compiler (see Fig. 3, QVTO2ATLVM\(^1\)). The transformation definition on the left is the model of a parsed QVTO transformation. The resulting assembly (transformation definition on the right) could then directly be executed on the ATLVM just like ATL transformations. Moreover, compatibility with ATL transformations can be achieved through means of superimposition [16]. In effect, this approach is not subordinate to a QVTO to ATL solution (with subsequent step to ATLVM). It is less complicated, however, since the ATLVM is at a lower abstraction level [10].

![Fig. 3. The QVTO to ATLVM compiler in ACG](image)

The compiler semantics can be derived from the QVT specification. Like any specification this one contains some points which are ambiguous or incomplete. Whenever we encountered impreciseness in the specification we tried to resolve it by discussing it with the communities of other (open source) implementations (especially [14]). The results of this process are presented in Section 6.3. We aimed to implement the full QVTO specification. To verify whether we achieved this goal, a representative set of QVTO transformations should run on the compiler. To this end the demo transformations from [15] are used.

\(^1\) For simplicity, again we omit the fact that the ACG transformation is transformed to ATLVM first.
In this section, we will detail the QVTO to ATL compiler. Since ACG is a very specific syntax [1], we will refrain from listing code samples in ACG. To illustrate the way in which we defined the QVT semantics in ATL we will use pseudo code which abstracts from the stack-based virtual machine implementation by using variables. Like any good compiler implementation, the compiler will provide the semantics for each construct of QVTO (or each type in the abstract syntax tree (AST)). So the pseudo code will define templates for QVTO constructs.

Listing 3 shows such a template for the construct MappingOperation. It will be executed for each MappingOperation \( m \) found in the AST of the QVTO program that is being compiled. When executed, a virtual machine operation will be emitted (line 2). It will have a context, a name and parameters, which are all derived from \( m \) using OCL queries (in bold). For understanding these queries it is highly recommended to read about the QVTO abstract syntax in [4].

In pseudo code, we sometimes summarize large computations with natural language in italics. If these operations are not trivial, they will be detailed in a subsequent listing. Square brackets are used inside the pseudo code to indicate code generation from queried model elements. So line 3 will call the template for whatever construct is found in the when property of \( m \). The benefit of the pseudo code is that it links compile and runtime using variables and thereby it can represent the compiler internals compactly. For example, on line 3, the result of the square brackets is emitted code (compile time) and the result from executing this code (runtime) is subsequently assigned to the variable “executable”. In the accompanied text to the code listings, QVT constructs are written in italics.

```
1 MappingOperation m {
2   operation context m.context name m.name params m.ownedParameter {
3     executable = [ m.when ]
4     if executable then
5     result = [ m.bodySection ]
6     else
7     strict = [ m.calledStrict ]
8     if strict
9     Raise exception
10    endif
11   return result
12 }
13 }
```

Listing 3. Example pseudo code

The following subsections will provide pseudo code templates for all QVTO constructs except those that have relatively simple semantics and can easily be handled with low-level ATLVM instructions.

### 5.1 OperationalTransformation

An OperationalTransformation represents a model transformation. It has a name and a set of model parameters (in, out and inout) as its interface. This concept can directly be represented using the ATLVM assembly language. The ATLVM has built-in library support and can combine several of these transformations,
like the QVT semantics prescribe. Since the ATLVM supports M-to-N transformations all the model parameters can be handled (in, out and inout). IntermediateProperties and IntermediateModels are features of transformations that can be used as structure for temporary data storage at transformation execution. They can easily be mapped to (assembly-language) global variables of a corresponding type in the ATLVM.

5.2 MappingOperation

A MappingOperation is responsible for mapping input model elements to output model elements. Its semantics are sensitive to trace information. Therefore to store and retrieve trace information we created several methods that the compiler generates for each compiled transformation. The interface used to query this information is shown in Table 1. The Trace object that is returned by the “findTrace” functions has members for the called mapping, the inputs, the outputs and the returned result.

Table 1. Interface for trace information

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace::findTrace(inputs, [reversed])</td>
<td>returns the set of outputs (if any) that are mapped to the inputs or vice versa (if reversed=true)</td>
</tr>
<tr>
<td>Trace::findTrace(mapping, inputs, [reversed])</td>
<td>returns the set of outputs (if any) that are mapped by mapping to the inputs or vice versa (if reversed=true)</td>
</tr>
<tr>
<td>putTrace(mapping, inputs, outputs)</td>
<td>stores a trace of inputs to outputs in mapping</td>
</tr>
</tbody>
</table>

Depending on the trace information a MappingOperation can be combined in several ways with other mapping operations (inheritance, disjunction and merging). It can also have a precondition (when clause) and/or post condition (where clause). A mapping operation should have at least one input (a context or in parameter\(^2\)) and one out parameter (result). In addition, it may contain several other in, out or inout parameters.

Since ATLVM and QVTO share the imperative object-oriented paradigm, mapping operations can be directly mapped on assembly operations. The return value of assembly operations can be used to handle result. The values of other out and inout parameters will have to be passed back to the calling operation by global assembly variables (this will be discussed in more detail later). In Listing 4, we show the general structure of an assembly operation that implements the semantics of a MappingOperation m.

\(^2\) There is an ambiguity in the specification which we will discuss in Section 6.3
operation context m.context name m.name params m.ownedParameter {
    Check parameter type conformance
    Check and call disjunctions
    executable = [ m.when ]
    if executable then
        trace = findTrace (m, m.ownedParameter.select(type=in))
        if trace
            return trace.result
        end if
        Retrieve inherited values for out parameters
        [ m.initSection ]
        Instantiate uninitialized model elements for all out parameters
        putTrace (m, m.ownedParameter.select(type=out))
        Call inherited operations and assign to result
        result = [ m.bodySection ]
        [ m.endSection ]
        Call merged operations and assign to result
        executable = [ m.where ]
        if !executable
            Raise exception
        else
            strict = [ m.calledStrict ]
            if strict
                Raise exception
            endif
        endif
        return result
    endif
}

Listing 4. Definition of the semantics of a mapping operation

Disjunction is a way of combining several operations with distinctive preconditions. The semantics of executing disjunctive operations (line 3) are defined in Listing 5.

for each disjunction in m.disjunction do
    executable = [ disjunction.when ]
    if executable then
        result = call disjunction
        return result
    endif
endfor

Listing 5. Definition of disjunction semantics

Semantics for operation inheritance require the inheriting operation to pass the values of its out parameters to the operation it inherits from. This is done on line 9 and 14. The value passing can be implemented using global variables or by adding extra (hidden) parameters to the operation. The following pseudo code does not discriminate against either method and refers to the passed values as a property inheritedValue (Listings 6 and 7). Merging semantics can be defined analogously.

if m.inheritedValue
    m.result = m.inheritedValue

Listing 6. Retrieve inherited values for out parameters

for each inherited in m.inherited
    inherited.inheritedValue = m.result
    m.result = result
endfor

Listing 7. Call inherited operations and assign to result
For the sake of simplicity only the result parameter is considered as out parameter. This can be easily extended to all output parameters by using a composite type. Also we did not take into account multiple result values. This would require value exchanges between operation callee and caller (also for inheritance, merging and disjunction) and can be implemented using global variables or composite values in the return value.

5.3 ImperativeCallExp

An ImperativeCallExp calls operations. Multiple result values and values of out/inout parameters can be passed as described in the above paragraph. They should however be processed after an operation call completes. Such a processing would involve decomposing the composite type or global variables and assigning the values to the appropriate variables in the operation call.

5.4 HelperOperation, ConstructorOperation and EntryOperation.

The semantics of these specialized operations do not include any other concepts than the mapping operations, therefore their definition can all be derived from the definition of the mapping operation. Except for an additional constraint of the helper operation, which states that no model element can be created in the output models. This constraint can be implemented using an OCL query on the model of the QVTO program at compile time.

Operation overriding has been defined for all kinds of operations. We did not feel the need to implement the semantics, because the ATLVM already overrides operations with the same signature.

5.5 ResolveExp

A ResolveExp has several executing semantics that have in common a source element for which trace links are searched and resulting (zero or more) target element(s). So the following properties of resolve are optional.

- an inMapping, specifying through which operation the trace link should have been created.
- a one flag, indicating that only one result should be returned
- an inverse flag, indicating that the resolve is from target to source model
- a deferred flag, indicating that the resolving process should be delayed until after the transformation execution.

Listing 8 shows the definition which supports all these semantics for a resolve r.

```
1 ResolveExp r {
2   if r.deferred then
3     Store resolve parameters and assigned property
4     return null
5   endif
6   if r.inMapping then
7     result = findTrace(r.inMapping, r.source, r.inverse)
```

128
Listing 8. Definition of resolve semantics

Line 3 handles deferred resolve semantics. Of course the result of this deferred resolve should end up somewhere in (one of) the target model at the end of the transformation execution. Therefore the QVTO specification defines the deferred assignment as the variable or model property assignment directly underneath the resolve operation\(^3\).

The target (left-hand-side) of the deferred assignment needs to be stored for the delayed resolve execution. In the words of the specification: “the execution engine stores the following information for the future variable: the source object, the function representing the filtering expression and the property or the variable reference to be assigned.” After the EntryOperation of the transformation is finished we can reuse the semantics definition in lines 6 to 13 on this stored information to execute the deferred resolve.

Listing 8 abstracts from the way the trace information is stored, like we already did in the mapping operation definition. Later we give the definition of the interface that we used so far and give possible solutions for storing of the trace information.

5.6 TryExp, RaiseExp, BreakExp and ContinueExp

Try blocks and raise expressions are the basis of an exception handling mechanism. Like the operation return (ReturnExp), these can be implemented using normal control instructions (if and goto) and extra checks around operation calls. An simplified example of exception handling without exception types is found in Listing 9.

3 In Section 6.3 we show that the case of variable assignment actually causes an ambiguity in the specification
6 Discussion

The definitions, described in the preceding section, have actually been implemented. In the next subsection, we evaluate the implementation. A list of successfully executed example transformations, illustrates to what extent the goals have been met. Thereafter the limitations of the current compiler are discussed. Finally the ambiguities in the QVT specification are explained and the possibilities for future work are examined.

6.1 Evaluation

The following example transformations were successfully executed on the ATL environment using the QVTO to ATL compiler. These examples use all of the expressive features of the QVTO language. Therefore, we can conclude that the goal of implementing a significant portion of the QVTO specification has been achieved.

– UML22RDBMS\textsuperscript{4} is a classic MDT example that uses a large part of the QVTO constructs, including resolve and iteration expressions. It features configuration parameters like intermediate models and aliases. Intermediate properties and models are also used.
– UML2Ecore\textsuperscript{4} features inheritance of mapping operations, complex OCL expressions, assertions, mapping calls with strict semantics and more.
– Ecore2EMOF\textsuperscript{4} features many advanced QVTO constructs.
– An industrial case from Obeo features at least the same complexity as Ecore2EMOF.

Especially the last three examples show that the QVTO to ATL compiler can be used for industrial applications. Furthermore, Kurtev and Jouault [10] expected that the solution presented here could be achieved at “relatively low cost”. Our experience confirms that this is indeed the case, since this effort was realized in the short time of one month by one person. Comparing this to the considerate amount of time that is normally spent on implementing solutions as complete as this, we consider their hypothesis validated.

\textsuperscript{4} From the SmartQVT project [15].
6.2 Limitations

While we did not encounter any QVTO construct which could not be expressed in ATL, the compiler still has some limitations. Regarding the modeling architecture, ATL takes a different view of models and transformations than QVT. QVT specifies that transformation combination is a matter of the transformation specification, while ATL solves this in the engine. For example in QVT we have to specify the model types with complete reference to the model, while an ATL transformation only specifies a label which is binded by the engine. The same could be said about transformation extension (QVT) and superimposition (ATL) [16]. To implement this in ATL we had to remove this information from the transformation definition and place it in the engines configuration.

These limitations had some effect on the examples shown in the previous subsection. The definitions had to be adapted to match the ATL architecture on the described points. However, this solution does not limit the functionality of the QVT to ATL compiler.

6.3 The QVT specification

Making an implementation of a specification allows to expose its peculiarities. This work will result in some feedback on the QVT specification. We were not able to find the following problems elsewhere (the numbers refer to sections in [4]):

8.2 It is not specified what would be the result of writing to an in parameter within an ImperativeOperation and model element creation inside a Helper-Operation.

8.2 There is no distinction between statements and expressions in QVT. A break is represented by an BreakExp, which indicates it is an expression in the metamodel. Semantically it behaves like a statement, this seems to be acknowledged by the specification by the following sentence: “A break expression ... is used in the body of imperative loop expressions (while and for expressions). A break expression cannot be directly owned by a non imperative expression; like the side-effect free OCL iterate expression.” This does however not make BreakExp semantically a statement in all cases, while can be used inside OCL expressions. This makes compiler development more difficult, because inside expressions we cannot have access to the stack contents.

8.2.1.15 “A mapping operation is an operation implementing a mapping between one or more source model elements into one or more target model elements”. While example A.2.4 defines a mapping without any input parameter.

8.2.1.20 “Unless isVirtual is true this invocation is virtual”. Logically this should be “false”.

8.2.1.22 Deferred assignments can have both a variable and a property as left-hand-side (target). In the case of a variable, the only meaningful semantics would be to find property assignments that use the variable. Otherwise the
deferred resolve will not end up in the target model. This however would make the language partly declarative even though it is defined as “completely imperative” ([4] chapter 8). The same problem arises when a deferred assignment is used inside an expression.

8.2.2.8 *SwitchExp* is specified to be sensitive to a *ContinueExp* and *BreakExp*, but the semantics is not given.

8.2.2.8 The alternative notation for switch statements is not elaborated on.

### 6.4 Future Work

A QVTR to ATLVM compiler has also been implemented. It is included in the Eclipse Model to Model Project [14]. This compiler is not documented well enough yet to be included in this experience report.

Although not all the constructs from the QVTR specification are semantically supported, enough are present to interpret the SimpleUML to RDBMS reference example. One of the concepts still missing is collection template support. It has been delayed mainly because we do not have a practical example of its use yet. The *enforce* mode is on active development. The experience reveals the implementation to be feasible.

### 7 Conclusion

In this paper, we studied in detail the interoperability between QVT and ATL. We focused on QVT Operational. Because of the layered architecture of ATL that uses a virtual machine to execute transformations, we were able to define this imperative language on top of it. We have designed, implemented and described a compiler, based on model transformation techniques (ACG). The result is that we can now specify QVTO programs and run them in the ATL environment.

The examples show that the current compiler status can cope with industrial transformation demand. Our results show that ATL is QVT conformant as defined in the QVT specification [4]. At the same time the benefits of running multiple languages on top of one transformation execution engine can be enjoyed. The ATL engine provides a model encoding technologies independent view on models and this can now be used in QVT transformations. Furthermore QVT and ATL transformations could extent each other using predefined coupling mechanism like black boxes.

This status should allow a transformation designer to: interchangeably use the languages, fairly compare transformations in different languages and reuse the ATL tools. Furthermore, ATL can be considered QVT compliant, the compiler implementation is an extra proof of this. In our experience, the MDE paradigm, which the AMMA tool set provides, to parse and compile (actually transform) is very efficient for (transformation) language definitions. It was certainly able to compile a large extent of the QVT languages. Initial results from a QVTR to ATL compiler also confirm this [14].

132
8 Acknowledgements

First of all, I thank the people at Obeo for giving me an interesting assignment and allowing me to work with them. Former colleague Quentin Glineur helped by discussing the details of his work (Quentin worked on a QVT Relations counterpart). I am grateful for the many hours he could find in his busy life to help and for the kind patience he showed when explaining his (more difficult) work.

Ivan Kurtev offered his help for writing this paper. Due to him this effort could be realized and I am grateful for his advice and guidance. Frédéric Jouault gave useful comments while the QVTO compiler was being implemented.

References

14. The eclipse m2m project website, http://www.eclipse.org/m2m.
On Using UML Profiles in ATL Transformations

Manuel Wimmer and Martina Seidl

Business Informatics Group,
Vienna University of Technology, Austria
{wimmer|seidl}@big.tuwien.ac.at

Abstract. For defining modeling languages, metamodels and UML profiles are the proposed options. While metamodels are supported by several dedicated model transformation approaches, currently no transformation language exists which support UML profiles as first class language definitions. Instead, the usage of UML profiles in transformations is implicit by using calls to external UML APIs.

In this paper, we first discuss the state-of-the-art of using UML profiles in ATL. Subsequently, three approaches for supporting profiles as first class language definitions within ATL transformations are introduced and discussed. In particular, these approaches aim at using stereotypes and tagged values within declarative rules without using external API calls. The benefits are: first, the enhanced static checking of ATL transformations, second, the more explicit representation of transformations logic enhances the application of higher-order transformations, and third, enhanced tool support such as code completion may be provided.

1 Introduction

Context. For defining languages in the field of model engineering, metamodels and UML profiles are the proposed options. While metamodels, mostly based on the Meta Object Facility (MOF) [3], allow the definition of languages from scratch, UML profiles are used to extend UML [4] with new concepts.

Problem. Metamodels are supported by current model transformation languages as first class language definitions, however, UML profiles are not. Nevertheless, UML profiles may be used in transformations with a little work-around. UML profiles are considered as additional input models and by calling external UML APIs, profiles are applied. Although, this is a technical possibility, the development of such transformations code is challenging (cf. Section 4).

Solution. As a first step of using UML profile definitions as first class language definitions in model transformations, we discuss in this paper the state-of-the-art of using UML profiles in ATL [1], identify some shortcomings, and propose three ways of using UML profiles in ATL transformations more systematically by employing a running example. In particular, the explicit use of profiles as language definitions provides various benefits. First, errors concerning the misuse of profiles may be detected at design time. Second, transformations may be easier enhanced by higher-order transformations. Third, tooling issues may be improved, e.g., code completion for recommending stereotypes that may be assigned on a UML element gets achievable.
2 Motivating Example

Two main scenarios exist where UML models annotated with profile information have to be transformed. The first one is the vertical transformation scenario by following the model-driven architecture proposed by the OMG. Platform independent models are created which are subsequently refined with platform specific information by applying profiles consisting of stereotypes and tagged values for specific platforms. From these platform specific models, code is generated where the profile information is one of the main driver. The second scenario is the horizontal transformation scenario, where modeling languages have to be bridged to UML. For example, in the context of the ModelCVS project [2], one industry partner was using the CASE tool AllFusion Gen\(^1\) (AFG) from ComputerAssociate which supports a language for designing data-intensive applications and provides sophisticated code generation facilities. Due to modernization of the IT department and the search for an exit strategy (if tool support is no longer guaranteed), the need arises to extract models from the legacy tool and import them into UML tools while at the same time the code generation of AllFusion Gen should in the future be usable for UML models as well. Therefore, models have to be exchanged between AFG and UML tools without loss of information which requires the extension of UML by an AFG profile.

Running example. As a running example, we are using an excerpt of a tool integration case study conducted in the ModelCVS project. The goal was to bridge the structural modeling part of AFG and UML, i.e., the AFG Data Model with the UML Class Diagram. Since AllFusion Gen’s data model is based on the ER model, it supports ER modeling concepts like \textit{EntityTypes}, \textit{Attributes}, and \textit{Relationships}. Furthermore, two concrete subtypes of the abstract \textit{EntityType} concept can be distinguished, namely \textit{AnalysisEntityType} and \textit{DesignEntityType}. AllFusion Gen is typically used for modeling data intensive applications which make excessive use of database technologies. Therefore, the data model allows the definition of platform specific information typically usable for generating optimized database code, e.g., \textit{EntityTypes} have special occurrence configurations. It is obvious that the corresponding UML model type for AllFusion Gen’s data model is the class diagram.

3 State-of-the-Art in ATL

After having introduced the modeling languages to be integrated, we now proceed with bridging AFG with UML. Due to space limitations, we only consider an excerpt which is used in the subsequent ATL listings. Details for the \textit{EntityType,2,Class} mapping are illustrated in Fig. 1. The abstract metaclass \textit{EntityType} of AFG is mapped to metaclass \textit{Class} in UML. In addition, the attribute \textit{noInstances} on the LHS is mapped to the attribute \textit{isAbstract} on the RHS. Several platform specific attributes of AFG remain unmapped which have to be represented as tagged values in the AFG profile.

---

AFG Profile. The profile excerpt resulting from the EntityType_2_Class mapping is shown in Fig. 2. An abstract stereotype EntityType is defined for the abstract metaclass EntityType. Furthermore, unmapped attributes are represented as tagged values. Two concrete sub stereotypes are defined in addition which have tagged values attached for their unmapped attributes.

Transformation Architecture. Currently the only way to make use of UML profiles in model transformations is that the profiles are additional input models for the model transformation. Fig. 3 illustrates this by showing the runtime configuration for transforming AFG models into UML with ATL. The user has to define the AFG model as first input model, and the UML profile as second input model.

ATL transformation code. Listing 1.1 illustrates an excerpt of the transformation from AFG to UML. It consists of an abstract transformation rule, which results from the EntityType_2_Class mapping. The current ATL version does not allow to define do blocks for super rules, thus, feature to tagged value mappings, e.g., for avgOccurrence, must be defined in concrete sub rules. In fact, two concrete sub rules are necessary for our running example, one for transforming AnalysisEntityType (cf. second rule in Listing 1.1) and one for DesignEntityTypes (not shown due to space limitations). These
sub rules have to implement stereotype applications and feature to tagged value assignments for super stereotype tagged values (e.g., \textit{avgOccurrence}) as well as for leaf stereotype tagged values (e.g., \textit{phase}).

\begin{verbatim}
module AFG2UML;
create OUT:UML from IN:AFG, IN2:PRO;

helper def: ste : PRO!Stereotype = PRO!Stereotype.allInstances()
  -> select(e|e.name = 'AnalysisEntityType').first();

abstract rule ET_2_Class {
  from s : AFG!EntityType
to t : UML!Class (isAbstract <- s.noInstances )
}
rule AnalysisET_2_Class extends ET_2_Class{
  from s : AFG!AnalysisEntityType
to t : UML!Class
do {
  t.applyStereotype(ste);
  /* assign tagged values from super stereotypes */
  if (not s.avgOccurrence.occIsUndefined()){
    t.setTaggedValue(ste,'avgOccurrence',s.avgOccurrence);
  }
  /* assign tagged values from leaf stereotype */
  ...
}

\end{verbatim}

4 Profile-Aware Transformation Language

In this section we first discuss some shortcomings of ATL concerning the usage of UML profiles based on the afore presented listing and present three approaches for tackling these shortcomings.

\textbf{Shortcomings.} When taking a closer look on the afore shown ATL listing, the following shortcomings may be identified. 1) Feature to tagged value assignments have to be done in \texttt{do} blocks. Because \texttt{do} blocks cannot be used for super rules, these assignments have to be done for each sub rule again and again. 2) The application of stereotypes is only implicit by calling external UML APIs. Therefore, it cannot be check if a certain stereotype is applicable for the UML element and because of stereotypes are only encoded as a String values, it is not ensured that the stereotype actually exists in the profile. 3) The same problems as for stereotype assignments exist for tagged value assignments. 4) Some low-level details of external UML APIs have to be considered in the transformations. For example, the assignment of a null value to a tagged value results in a runtime exception.

\textbf{Profile-aware Transformation Language.} We now propose and discuss three approaches how UML profiles may be used as first class language definitions.

\textit{(1) Merge.} This approach is a lightweight approach, meaning that no ATL language modification is necessary. Instead of using the UML metamodel and
the UML profile as separated definitions, they are merged into one metamodel. For this, stereotypes become metaclasses, tagged values become features, and extension relationships become inheritance relationships. In Listing 1.2, the ATL code is shown which is capable of transforming AFG models into UML models which conform to the merged metamodel. Please note, that the transformation code is more concise compared to Listing 1.1. Advantages of this approach are: the merged metamodel is automatically created and it is not necessary to extend the ATL language with new syntax elements. However, there are also some drawbacks. In addition to the merged metamodel, model adapters are needed to transform UML models conforming to the merged metamodel into standard UML models using profiles. Furthermore, in cases where more than one stereotype is applicable on the same element, this approach is not sufficient.

Listing 1.2. AFG to UML (merged UML metamodel)

```
module AFG2UML;
create OUT: UML from IN:AFG;
abstract rule ET<ET {
from s : DSML! EntityType
  isAbstract <- s.noInstances,
  avgOccurrence <- s.avgOccurrence,
  ...}
to t : UML! EntityType ( phase < s.phase )
}
rule AnalysisET<AnalysisET extends ET<ET{
from s : DSML! AnalysisEntityType
  phase <- s.phase }
to t : UML! AnalysisEntityType( phase <- s.phase )
}
```

(2) Preprocessor. This approach uses a slightly modified ATL syntax for describing UML profile aware transformations and a preprocessor which creates standard ATL transformations for execution purposes. The modified ATL syntax is used in Listing 1.3. Please note that we have introduced a using keyword in the header definition for referencing the used profile and an apply keyword which is used in the to part for applying stereotypes on target elements. For this modified ATL syntax, we are now able to provide dedicated code completion and static validation to ensure the proper usage of UML profiles. Furthermore, we allow to use feature to tagged value assignments in the to parts of the transformations, thus the inheritance feature between declarative rules can be fully exploited. Advantages of this approach are that only a modified syntax of ATL has to be provided as well as a transformation to standard ATL. Disadvantages are that debugging is only supported for the generated standard ATL transformations and a new development line is created that requires a parallel development with the standard ATL development line.

(3) Extending ATL. Finally, the additional syntax elements for using UML profiles can be directly integrated in ATL. This requires to extend not only the syntax, but also the ATL compiler which is much more implementation work compared to the previous approaches. However, the benefit is that neither a preprocessing of metamodels and models (first approach) nor of ATL code
(second approach) is necessary. In addition, the complete tool support of ATL may be used such as the debugger.

Listing 1.3. AFG to UML (extended ATL syntax)

```plaintext
module AFG2UML;
create OUT:UML using AFG.Profile from IN:AFG;
abstract rule ET2ET {
  from s : DSML!EntityType
  to t : UML!Class apply EntityType {
    isAbstract <- s.noInstances,
    avgOccurrence <- s.avgOccurrence,
    ...
  }
}
rule AnalysisET2AnalysisET extends ET2ET {
  from s : DSML!AnalysisEntityType
  to t : UML!Class apply AnalysisEntityType {
    phase <- s.phase
  }
}
```

5 Conclusions and Future Work

In this paper, we discussed the state-of-the-art of using UML profiles within ATL transformations, identified some shortcomings, and proposed three approaches how to use UML profiles as first class language definitions. In future work, we plan to integrate UML profiles by extending the ATL (approach three). In particular, we want to evaluate the consequences on writing transformations which make heavy use of UML profiles.

References

Checking syntactic constraints on models using ATL model transformations

Skander Turki, Eric Senn, Dominique Blouin, Saadia Dhoubi, Jean-philippe Diguet, Johan Laurent.

Université de Bretagne Sud
Lab-STICC / CNRS UMR 3192
skanderturki@gmail.com,eric.senn,Dominique.blouin, dhoubi, jean-philippe.diguet, johann.laurent}@univ-ubs.fr

Abstract. This paper presents a use case on using ATL model transformations to check syntactic correctness constraints on a UML/MARTE model. This model transformation is a part of an interoperability model transformation from UML/MARTE to AADL in which UML/MARTE is used as a UML profile for AADL. As input, the constraints checking model transformation takes a UML/MARTE model and produces as output a diagnosis model that conforms to a diagnosis meta-model (VERIF) which is also presented in this paper.

Keywords: Meta-modeling, Model transformation, Constraints checking, MDE, ATL, UML, MARTE, AADL.

1 Global context

AADL (Architecture and Analysis Design Language) [3] is a standard language defined by the Society of Automotive Engineers (SAE) for the specification and analysis of real-time and embedded systems [2]. AADL is based on architecture description constructs such as components, ports and connections. It also provides constructs for the specification of properties. AADL defines precise informal semantics for all constructs and formal execution semantics for threads. AADL allows the high-level analysis of the specification, the verification of functional and non functional properties of the system, and even code generation for the targeted hardware platform [4, 5, 6]. AADL is at the basis of the European ITEA2 SPICES project [1] (Support for Predictable Integration of mission Critical Embedded Systems) for which a suite of tools is being developed. CAT (Consumption Analysis Toolbox), among which this work is done, is the SPICES sub-project that provides a method to estimate power consumption at an early stage of the design process [7]. Power models have been built for many different hardware components [8] (Digital Signal Processors, General Purpose Processors, Memories, Busses, FPGA, etc) as well as for software components [9] (Operating Systems Services: Inter-Process Communications, Ethernet layers, etc). CAT is integrated with OSATE (an AADL based design tool).
In this work, our goal is to allow real-time and embedded systems designers that use UML with the MARTE profile (UML/MARTE) [10] to use any AADL-based analysis tool such as CAT by transforming their models into an AADL model (MarteToAADL model transformation) with the ATL [12] model transformation tool. Such a model transformation will also allow MARTE users to stand upon precise semantics that are those of AADL. In addition to this the OMG MARTE consortium is working with the AADL standardization committee to align MARTE with the AADL semantics which will encourage integration of both languages and underlying tools in a same design process. The MarteToAADL transformation tool will also allow AADL users to use well-established CASE tools. This integration with MARTE will also accelerate the dissemination of AADL among the real-time and embedded community.

One of the difficulties with the MarteToAADL model transformation is to bridge the syntactic gaps between these different languages. In fact, MARTE inherits very large design capabilities from UML. This may lead to illegal constructions (with respect to the corresponding AADL constructs syntax). In this paper, we don't present the MarteToAADL model transformation itself. We only focus on a solution to bridge syntactic gaps between UML/MARTE and AADL. The adopted solution consists in a constraints checking model step that is executed before the core model transformation. We could have added these constraints directly into the MARTE profile, but we decided not to add any extension to MARTE as this would break the standard aspect of MARTE. Besides, in this case, why not build an AADL profile from scratch? And the answer to this is that we are targeting the MARTE profile which is supposed to be the standard language for real-time and embedded systems. The solution was then to check these constraints in the beginning of the model transformation. This constraints-checking step is itself defined as an ATL model transformation.

2 The VERIF Meta-model

For his diagnostics model transformation, Bézivin in [13] uses a quite simple diagnostics meta-model that has one “problem” class with “severity”, “location” and “description” properties. In our case, we also need a complete report on the constraints checking step with the report date and the decision on the possible continuation of the transformation chain. We need to explain to the designer the origins of the errors and give him advices to correct his design in order to have a correct Marte2AADL transformation. So we defined the VERIF meta-model shown in figure 1. We also defined in VERIF operations that can be called from ATL to execute Java code on the model.
For example, we defined the `getHigherSeverityNotification()` operation on the `VfVerificationReport` metaClass. In the Marte2AADL transformation, this operation will be called to go through all the notifications registered in the report and to test if the Marte2AADL transformation can continue (if the result is > 2 it cannot continue. The code of this operation is as follows:

```java
public int getHigherSeverityNotification() {
    int res=0;
    for (VfNotification notif : getMyNotifications() ) {
        if ( notif instanceof VfCriticalErrorMessage ) res = Math.max(res, 4) ;
        else if ( notif instanceof VfErrorMessage ) res = Math.max(res, 3) ;
        else if ( notif instanceof VfWarningMessage ) res = Math.max(res, 2) ;
        else if ( notif instanceof VfDesignCriticMessage ) res = Math.max(res, 1) ;
    }
    return (res);
}
```

Like explained by Voelter [14] the DSL expresses the “what”, the model processor adds the “how”, in our case the DSL (Domain Specific Language) is the VERIF “language”. So the model conforming to this meta-model will contain a complete report. Then the model processor, which will be the Marte2AADL transformation, will decide “how” to deal with each element of the report.
3 Checked Constraints

We give in the following the overall structure of the ATL module with an example of a checked constraint. In the ATL code, a rule will call all lazy rules that will check constraints and create appropriate notifications. The validation will be decided by executing blocking constraints only (see the validation mapping in the following code).

```
rule myVerificationReport {
from s : UML2!Package (s.oclIsTypeOf(UML2!Model))
to p: VERIF!VfVerificationReport{
  name <- 'Report_ON_Model_' + s.name,
  sourceModel <- s.name,
  validation <- if s.cstAllClOneCategStereo() and ... 
    then true else false endif,
  myNotifications <- s.getIllCategStereo()->collect(e | 
    thisModule.notifyAboutClCategStereo(e)))
  ->append(...) -- other notifications
}
```

3.1 cstAllClOneCategStereo (Urgent)

In AADL 1.0, a component can either be one of the ten AADL component categories. Applying two different category stereotypes to a component is then prohibited.

```
helper context UML2!Package def :
cstAllClOneCategStereo() : Boolean=
  if (self.packagedElement->select(c | 
    c.oclIsTypeOf(UML2!Class))->forAll(s | 
    s.hasNoMoreOneCategStereo()) and -- call for helper defined next 
    self.packagedElement->select(p | 
    p.oclIsTypeOf(UML2!Package))->forAll(t | t.
    cstAllClOneCategStereo()))-- recursive call for contained packages 

helper context UML2!Class def : hasNoMoreOneCategStereo() 
  : Boolean 
  if (self.getAppliedStereotypes() 
    ->select(c | 
      (c.toString() = 'IN!HwBus') or 
      (c.toString() = 'IN!HwDevice')) or 
      (c.toString() = 'IN!HwProcessor') or 
      (c.toString() = 'IN!HwMemory') or 
      (c.toString() = 'IN!MemoryPartition') or 
      (c.toString() = 'IN!SwSchedulableResource'))
    ->size()) < 2 then true else false endif;
```
5 Conclusions and perspectives

In this paper, we presented a part of a MARTE to AADL model transformation that is used to solve the syntactic gaps between the two languages by constraints checking.

We also demonstrated how we can use a model transformation to check constraints on models. We also detailed the presentation of the ecore VERIF meta-model. We presented the structure of the principal ATL rule that will call all other constraint checking rules. We only presented one constraint for reasons of space limitation. For some syntactic gaps, we chose not to create a constraint. For implementation reasons, some “bad designs” can just be ignored like allocations between abstract classes.

References

Mutation Analysis for Model Transformations in ATL

Piero Fraternali, Massimo Tisi
Politecnico di Milano, Dipartimento di Elettronica e Informazione
Milano - Italy
{massimo.tisi, piero.fraternali}@polimi.it

Abstract. For Model Driven Engineering to gain acceptance among developers, it is imperative that the coding of transformations be supported by methodologies and tools comparable to those available for traditional programs. In particular, testing is an area where tools are much needed, both for the generation of test suites and for their validation. This paper tackles one of the enabling tools for test suites.

1 Introduction

In this paper, we describe a mutation analysis framework for model transformations, written in ATL. Mutation analysis consists in the systematic creation of so-called mutants, i.e., faulty versions of a supposedly correct program, and in checking the efficiency of a test dataset to detect the injected faults. The main interest of mutation analysis is to provide an estimate of the quality of a test set based on the rate of faulty programs it detects. This estimate is supposed to reflect quite closely the “fault revealing power” of the test set.

To be effective, mutation analysis must create mutant programs that correspond to realistic faults. The faults are injected into the correct program by means of a set of mutation operators. The problem of identifying a set of realistic mutation operators for model transformations, independently from a particular transformation language, has already been studied in [2]. The authors distinguish four error classes, corresponding to the four main operations performed by model transformations, i.e., navigation, filtering, output creation, input modification. For each one of the error classes, a set of mutation operators is defined to represent the most common mistakes in transformation development. For instance one of the simplest mutation operators is Collection Filtering Change with Deletion (CFCD) that represents the mistake of forgetting a needed filter from the left hand side of a transformation rule. The other eleven mutation operators are quite self-explaining as well, like Relation to another class change, Relation sequence modification with deletion, Classes association creation addition.

In previous work, these operators were only defined in natural language and applied manually during the experimentation part [2]. In the next sections we propose a formalization of mutation operators as Higher Order Transformations (HOTs) and provide an implementation of the mutation framework in the AmmA
platform [1]. An HOT is precisely defined (e.g. in [4]) as "a model transformation such that its input and/or output models are themselves transformation models". As we show, the use of HOTs allows the developer to specify mutation operators in a very concise and abstract way. These simple mutation operators are first transformed into other HOTs in a preprocessing step, and then the generated HOTs are used for the actual production of the mutants.

2 ATL transformations as mutation operators

The higher-order transformational approach supports the definition of mutation operators for model transformations in a natural way. For instance, the CFCD mutation operator can be specified in ATL as follows:

```java
rule CFCD {
  from
    m : ATL!InPattern (
      not m.filter.oclIsUndefined())
  to
    m1 : ATL!InPattern (
      elements <- m.elements,
      rule <- m.rule,
      location <- m.location,
      commentsAfter <- m.commentsAfter,
      commentsBefore <- m.commentsBefore)
}
```

This transformation defines a mapping that is applied whenever an input transformation contains an ATL rule whose left hand side has a filter. The ATL rule is then copied, leaving behind the filter, thus injecting an error of the CFCD class into the ATL program.

In mutation analysis, a mutation operator is usually applied to a single point of the correct program, in compliance with the so-called single-point-of-failure hypothesis. However, the ATL execution engine does not apply the previous transformation in this intended way. In fact, a model transformation engine such as the ATL virtual machine detects all the matches of the left-hand side of the rule in the input model, and then applies the right-hand side to all of them at the same time. In this way, the application of the mutation transformation would generate a single mutant containing all the possible instances of CFCD errors. To keep the conciseness of the mutation specification, while adapting its execution semantics to the single-point-of-failure hypothesis, a model-driven technique is required, such that:

- is completely transparent to the user,
- does not change the standard transformation engine (for compatibility and manageability),
- has the minimal computational cost, to support the generation of a significant number of mutants, as required for the statistical significance of the experimental validation.
The solution we propose exploits a second order HOT, that is, another transformation that preprocesses the mutation operator and translates it into a corresponding version with a different execution semantics. Then the generated version is iteratively executed in the normal ATL virtual machine, to mutate the correct program in every possible way.

For instance, the second order HOT translates the previous CFCD transformation rule into the following two rules:

```java
rule CFCD {
  from
    -- apply to the nextMatch in mutatorMode
    m : ATL!InPattern (
      thisModule.isMutatorMode and
      m.location = thisModule.nextMatch
    )
  to
    -- apply the mutation
    m1 : ATL!InPattern ( -- mutation RHS
      elements <- m.elements,
      rule <- m.rule,
      location <- m.location,
      commentsAfter <- m.commentsAfter,
      commentsBefore <- m.commentsBefore
    )
}

rule toCFCD {
  from
    -- if we have a match in nonMutatorMode
    m : ATL!InPattern (
      not thisModule.isMutatorMode
      and not m.filter.oclIsUndefined() -- mutation LHS
    )
  to
    -- don’t apply the mutation
    m1 : ATL!InPattern ( 
      elements <- m.elements,
      rule <- m.rule,
      filter <- m.filter,
      location <- m.location,
      commentsAfter <- m.commentsAfter,
      commentsBefore <- m.commentsBefore
    )
  do {
    -- record the mutationPoint
    thisModule.addTrace(m.location);
  }
}
```

The rules make reference to a library of ATL helpers that control the transformation execution. The mutation transformation is executed in two different modes, and the helper isMutatorMode is queried at transformation time to detect the current running mode. The transformation is at first executed in nonMutatorMode, a phase in which the model is analyzed to collect all the matching points: only the toCFCD rule can match, which uses only the left-hand-side of the original transformation to detect the mutation points and save their locations into the Trace model, by means of the addTrace helper. The Trace model is an auxiliary model, which is always an input and output of the transformation executions, as shown in Figure 1. In the second phase, for each mutation point stored in the Trace model, the transformation is executed again in mutatorMode.
In this phase, only the CFCD rule can match, and only once, at the location returned by the helper `nextMatch`. The CFCD rule uses only the right-hand-side of the original transformation, to produce the specific fault. This phase generates a single different mutant at each execution.

By analyzing the resulting code, together with the provided library of helpers, it can easily be proved that the generated version of the HOT checks and updates the Trace model in constant time. Therefore, impact of the two-steps execution on the performance of the mutation generation engine is minimal.

3 Framework implementation

Figure 2 shows an overview of the architecture of the proposed transformation framework for mutation analysis.

The mutation operators described in [2] are formalized as a set of HOTs, that can be easily extended by the user, to define different classes of faults. From the mutation operators, the preprocessing HOT builds a set of mutation transformations, which constitute the so-called mutation engine. The mutation framework extracts the model representation of the original correct transformation, executes the mutation engine, and injects the resulting mutants back into ATL code. The mutants are then executed on the input test models, and their resulting output is compared to the correct one. Finally a comparison tool assigns a mutation score to the test set under examination.

The framework is completely implemented by ATL rules, orchestrated by a Java scaffolding. The complete code of the framework can be found in [3].
4 Conclusions

This paper studies a case in which HOTs are the central transformation in an artifact generation task: the production of mutants for supporting the testing of transformations. There are several such cases, especially all the tasks that derive an artifact from the code-generation transformation. Anyway, to our knowledge, none of these works has addressed mutation analysis. The technique discussed in this paper is also an exemplary case showing the possibility to stack HOTs, to obtain a complex transformation starting from a simpler version. Iterating this activity it would be possible to create a multi-step development process for model transformations. We believe that the use of HOTs to preprocess transformations can be useful in several cases besides mutant generation.

References

Towards Traceability support in ATL with Obeo Traceability

Freddy Allilaire
Obeo, 2 rue Robert Schuman, 44000 Rezé, France
freddy.allilaire@obeo.fr

Abstract:
ATL is widely recognized as a standard solution for model transformation. In the context of ATL industrialization, a traceability support for ATL will be added. The ongoing development product Obeo Traceability will be used to have this dedicated support. Obeo Traceability will be based on Acceleo Pro Traceability, a product to deal with traceability for Obeo tool chain products like Acceleo (a code generation tool). Acceleo Pro Traceability is a tool for ensuring the smooth integration of the MDA in professional projects. Traceability is an important feature in model-driven development (MDD). Typical software systems are comprised of a large number of artefacts that are all related. There is a need to trace and control these relations. Current approach for dealing with traceability for ATL limits dependencies to program logic by attaching traceability generation code to pre-existing ATL programs. An example of software application cartography will be shown to highlight usefulness of traceability information created and kept during model transformation execution.

1 - Introduction
ATL [1] is a model transformation language (and also toolkit) widely recognized as a standard solution for model transformation. Model transformation is a central element of model driven approaches. The present recommended approach to have a traceability in ATL is to attach traceability generation to pre-existing ATL transformations. Traceability is a relationship between elements and a key feature in MDE. This paper will present the principles of Obeo Traceability and current work of ATL and traceability with Obeo [2] approach. Obeo Traceability will be based on Acceleo Pro Traceability [3], a tool that is able to guarantee an integration of MDE approach with projects by getting a complete traceability for software production process. The paper is organized as follows. Section 2 presents some global consideration of traceability in MDE context. Section 3 shows the current recommended approach of ATL and traceability. Section 4 gives an overview of Acceleo Pro Traceability and Obeo Traceability tools. Finally the section 5 with the application cartograph will show why Obeo chose ATL as M2M solution and why it is so important to have traceability in ATL.

2 - Traceability in MDE context
The following definition to traceability could be given: “Traceability is a relationship between entities: a set of source entities, and a set of target entities. The exact meaning of the relationship depends on the context in which it is used” [4].

Traceability is very important in the MDE context. Most of typical softwares are composed of a large number of elements that are all related. These elements can be defined as any object or work product in the software life-cycle like informal text, model elements, code, tests. As the software are being developed, managed, maintained or extended there is a need to trace and control these relations. Usage scenarios for traceability include inspection of traces, analyses such as change impact analysis, coverage analysis, and orphan analysis, and support for reverse engineering.

One main difference between traditional software engineering and MDE is the use of model transformation (model to model transformation or model to code transformation). These are automatic or semi-automatic transformations using and producing traceability information.

3 – Current traceability support in ATL
As said in the introduction, model transformation languages are a key element of the MDE approach. ATL is a well known model to model transformation language and it is recognized as a standard solution for model transformation (notably in Eclipse). A language that is able to maintain relations between source and target model elements could support traceability. Links to program elements responsible for these relations may also be kept. Such a program may need to be able to generate different kinds of information. While limiting dependencies to program logic. Model transformation is used to implement this approach. To make things even more complex, a given transformation program
may actually need to support several kinds of traceability.

The current recommended approach to have a traceability in ATL [5] attaches traceability generation code to pre-existing ATL programs. With this, traceability generation code is clearly separated from transformation logic and can be attached after a program has been written. Consequently, adding support for new ranges or formats without tempering with program logic becomes possible. This kind of traceability could be easily added to ATL code and also it could be automated.

4 – From Acceleo Pro Traceability to Obeo Traceability

Dealing properly with evolutions and maintenance requires a perfect control of generated and modified code by developers. Obeo provides currently a tool called Acceleo Pro Traceability that support traceability inside Obeo products. This tool is able to guarantee an integration of MDE approach with projects by getting a complete traceability for software production process. It maintains traceability all along software development cycle. It is also useful when maintenance team gets code from other teams. It detects and corrects all incoherences and synchronization problems between application code and models. It is a tool dealing with synchronization between source code and high level models. Synchronization is asymmetric between heterogeneous code and functional models. It can detect in real time de-synchronizations thanks to Eclipse traceability markers. Coherence control is accurate to the precision of a character. It will detect manual code modifications from developers on generated parts, model or template changes and impacts, and additional files added manually.

Obeo Transformer was the current solution for dealing with model transformations in Obeo products. ATL was chosen as the new model transformation solution to replace Obeo Transformer as ATL is recognized as a standard solution for model to model transformation in Eclipse. Obeo launched the ATL Industrialization initiative to provide an industrial version of ATL and to distribute it on an international level.

A traceability support for ATL is a key feature to be integrate by industrials in complex tool chain. One possibility would have been to integrate ATL in current version of Acceleo Pro Traceability but this work around ATL traceability was the opportunity to develop a tool dedicated to traceability disconnected to Acceleo that will allow an easy integration of MDE product, that will provide all features required and a complete IDE (e.g. context menu to do round-trip operation from code generated to a source model). So, the choice was made to create a tool called Obeo Traceability. This product will keep all features of Acceleo Pro Traceability, but it will be more generic to ease integration of MDE products that want to benefit a traceability support. Each new component of Obeo Traceability will have to implement a dedicated connector that will describe elements to be traced and synchronized. Obeo Traceability will store and manage traceability elements and will provide a simple way to return required information. It will ease to have traceability between different tools and will allow traceability in a chain of MDE operations (reverse-engineering operation, model to model operation, code generation operation, etc.). The architecture of Obeo Traceability will be describe in later works. The integration of ATL in Obeo Traceability will be based on built-in support for traceability. ATL should keep links between elements generated by different transformation rules together during transformation execution. This traceability information does not need to be kept after executing a transformation in the current version of ATL. That is why it is called internal traceability. External traceability corresponds to the current approach presented in section 3. The links internally used to provide traceability information will be serialized. This results in the whole set of links between corresponding source and target elements. Each link is composed of the name of the rule, target elements, and source elements. This gathered information will be catched and integrated in Obeo Traceability toolkit to benefit existing tooling and all other advantages of this tool presented in previously with Acceleo Pro Traceability.

5 - Application cartography

After quickly presenting of ongoing Obeo Traceability and future ATL Traceability support, in this section an example of usefulness for traceability in model to model transformation is presented. This will be illustrated with an application cartography example. For this, an example presented in [6] will be reused. It is a cartography of software application based on heterogeneous technologies Cobol and C.

Figure 1 shows the big picture of the application cartography.
The chosen architecture conforms to typical approaches in reverse-engineering tooling with a very important place for metamodels in each step in this process. All this reverse process will not be re-explain here as we only want to highlight and insist on usefulness of model to model transformation and its traceability.

The first transformation used is between C, Cobol metamodels and Cartography metamodel. In this transformation all useful information are collected to gather a cartography model to later create the graphical cartography. The second transformation is between cartography metamodel and graphical cartography. This step could be done with a transformation because graphical visualization is rendered in a dedicated editor driven by metamodels.

Figure 2 presents a sample model created automatically for a simple application and conforms to cartography metamodel. Using transformation gives possibility to easily customize rendering and to only keep useful information. Of course this also could not be done without traceability during all this process. The traceability here allows to smoothly integrate model to model transformation among all other products. For example, the chain developer could easily detect the links between a C statement and its graphical representation in the resulting cartography.
6 - Conclusion

This paper presented current works on Obeo Traceability. This new tool will allow to have a traceability support in ATL. In the process of ATL industrialization, traceability support is needed as it is a key feature in MDE. Obeo Traceability will be based on Acceleo Pro Traceability, a tool that is able to guarantee an integration of MDE approach with projects by getting a complete traceability for software production process. ATL was chosen as model to model transformation solution by Obeo because it is recognized as a standard solution in Eclipse. The development of Obeo Traceability will ease integration of ATL to have good traceability support and also to integrate other MDE tools in complex tool chain operations. MDE products will more easily inter-operate between them thanks to this traceability information. Once integrated, ATL Traceability will allow us to benefit all advantages of traceability and as shown in the use case to easily create application cartography.

Acknowledgements

I would like to thank my Obeo colleagues for their support in this work.

References

[1] ATL web site: http://www.eclipse.org/m2m/atl/
Abstract. Formal software design (usually described with UML) is now a must have for Object Oriented application development. Analysts and designers have to structure and optimize models that describe the application. Labor standards are developed to assist in the achievement of reliable, structured, scalable and maintainable software. In this paper, we describe UMLQualityAnalysis: an open source application that allows performing measurements on UML models and checks whether they comply with specific design standards. Results are automatically presented in a report available in several formats. In this paper we focused on the model driven part and explain how measurements are taken…

1 Introduction

The Unified Modeling Language (UML) [1] is the most common language to models application. The advantage of UML is that it is independent of any implementation language, which is why UMLQualityAnalysis is based on it. To exploit the UML data models, we chose the Models to Models (M2M) [2] technology and more specifically three languages:

− The ATLAS Transformation Language (ATL) [3] for M2M transformation ;
− The Kernel Meta Meta Model language (KM3) [4] to write metamodel ;
− The Textual Concrete Syntax language (TCS) [5] to make Domain Specific Language (DSL).

First, we present an overview of the application with a description of the different transformation steps. The second section explains why we choose to transform an UML model into an internal representation. The third section describes the different measurements that are done. Measurements are divided in two separates concerns: generic measurement and standards design control rules. The program is an open source application, so we propose a way to configure the generics control rules to be consistent with private conception designs. This solution is explained in the fourth section. Finally we conclude in a fifth section.
2 Overview

The following diagram describes the general process and the sequence of the sub processes.

As we explain in the introduction, the general process is divided in four sub processes that will be detailed in the following paragraphs.

2.1 UML to internal UML concepts

Input UML models can be directly created from a modeler UML conforming to the metamodel UML 2. To test the application we use the software Topcased [6].

First the UML model is transformed into several internal representations: components, use cases, activities, etc. These transformations guarantee the following processes. In fact, the UML metamodel can evolve and if we want to upgrade the application to the new UML representation, just a few transformations must be build to date. The internal representations separate the original metamodel in several models depending on the measurements that will be made thereafter: for example,
some measurements on components have not to worry about the measurements on use cases.

2.2 Generic measures

![Diagram showing the process of generics measurements](image)

From the internal components model, several object-oriented metrics are implemented in UMLQualityAnalysis. These metrics are distinguished into three categories: basics metrics, Chidamber & Kemerer metrics [7] and Fernando Brito e Abreu metrics [8]. We use a table to present the metrics, the first column is the acronym of the metric, the second describes it and the third shows in which contexts the metric is calculated.

Remark. Three contexts are available: Model (M), Package (P) and Class (C). For example, if a measure get the total number of attributes in the contexts C and P we obtain three values. Firstly, the total number of attributes for a specified class, secondly the total number of attributes for the package that contains the class and finally the average number of attributes per classes in the package.

**Basics metrics.** These metrics are the simplest; it is on these that the more complicated metrics are based. They enumerate the basic UML element. The next table presents them.

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>TNA</td>
<td>Get the total number of packages</td>
<td>M</td>
</tr>
<tr>
<td>TNC</td>
<td>Get the total number of classes</td>
<td>P/C</td>
</tr>
<tr>
<td>TNAI</td>
<td>Get the total number of attributes</td>
<td>P/C</td>
</tr>
<tr>
<td>TNAI</td>
<td>Get the total number of attributes inherited</td>
<td>P/C</td>
</tr>
<tr>
<td>TNM</td>
<td>Get the total number of methods</td>
<td>P/C</td>
</tr>
</tbody>
</table>

![Table 1. Basics metrics](image)
**Mathieu Vénisse**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>TNMI</td>
<td>Get the total number of methods inherited</td>
<td>P/C</td>
</tr>
<tr>
<td>TNAs</td>
<td>Get the total number of associations</td>
<td>P/C</td>
</tr>
<tr>
<td>TNAsI</td>
<td>Get the total number of associations inherited</td>
<td>P/C</td>
</tr>
</tbody>
</table>

**Remark.** In the ATL transformation, these metrics are defined in a library which is used in the next transformations.

**Chidamber & Kemerer metrics.**

**Table 2. Chidamber & Kemerer metrics**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMC</td>
<td>Get the weighted methods per class</td>
<td>C</td>
</tr>
<tr>
<td>DIT</td>
<td>Get the Depth inheritance tree</td>
<td>P/C</td>
</tr>
<tr>
<td>NOC</td>
<td>Get the number of children</td>
<td>P/C</td>
</tr>
<tr>
<td>CBO</td>
<td>Get the couplage between objects</td>
<td>P/C</td>
</tr>
</tbody>
</table>

**Remark.** These metrics are mainly designed to measure the complexity in the design classes and the difficulty to maintain them. The WMC can predict the time and effort needed to develop a class and gives an indication on its reusability. The DIT estimates the complexity of a class and its reusability. If a class has a lot of inherited methods, the DIT will be important and the class will be not easy to understand. The NOC metric shows the influence of the class on the system meaning that this class should be tested more precisely. The CBO estimate the modularity, the reusability and the maintenance effort, the larger the number of methods that can be invoked from a class, the greater the complexity of the class.

**Brito e Abreu metrics.**

**Table 3. Brito e Abreu metrics**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIF</td>
<td>Get the attribute inheritance factor</td>
<td>P/C</td>
</tr>
<tr>
<td>MIF</td>
<td>Get the methods inheritance factor</td>
<td>P/C</td>
</tr>
<tr>
<td>AHF</td>
<td>Get the attribute hiding factor</td>
<td>P/C</td>
</tr>
<tr>
<td>MHF</td>
<td>Get the methods hiding factor</td>
<td>P/C</td>
</tr>
</tbody>
</table>

**Remarks.** These metrics evaluate the system as a whole; results are expressed in a percentage for a better interpretation. The AIF and the MIF estimate the abstraction of a class. The AHF and the MHF estimate the encapsulation.
2.3 Control design rules

UMLQualityAnalysis can check if a UML model complies with internal design rules of a project. An ATL transformation takes two models in entry: the components model and a model conform to a Configuration metamodel. Two DSL are developed: one to make configuration model and another to extract report model. The configuration model allows filtering UML elements by stereotype, name or visibility.

**Example.** If we want to check the classes stereotyped “A” that are linked together, firstly we have to make a configuration model with this filter and apply the ATL transformation with the corresponding rule.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>2 rules: check if specific classes are or are not linked</td>
</tr>
<tr>
<td>Operation</td>
<td>2 rules: check if specific operations are or are not in specific classes</td>
</tr>
<tr>
<td>Interface</td>
<td>3 rules: check if specific classes implements (or not) a specific interface or get all the classes that implement a specific interface.</td>
</tr>
<tr>
<td>Package</td>
<td>1 rule: find specific classes in specific package</td>
</tr>
</tbody>
</table>

With a configuration model and this generics rules, the possibilities are endless.

3 Conclusion and future work

The measurement results are automatically transformed into a simple Table metamodel and successively in the XML metamodel. These models are finally parsed by a Java program.

In this paper we present an application based on the M2M technology to make measurements on UML models. Generics measurements and control software design
are implemented. With the control design rules, possibilities are endless due to the Configuration metamodel. TCS simply permits to create intuitively configuration model.

As future work we will use the parsed results to create reports with an open source API as Ireport [9]. The choice of the API is not yet final at this level of progress. Reports will be automatically generated in several formats like pdf, html, etc. and results will be shown with adapted diagrams.

Reference

2. Eclipse foundation. Model To Model project (M2M). http://www.eclipse.org/m2m/.
## Author Index

<table>
<thead>
<tr>
<th>Author</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alff, Patrick D.</td>
<td>18</td>
</tr>
<tr>
<td>Allilaire, Freddy</td>
<td>150</td>
</tr>
<tr>
<td>Alonso, Alejandro</td>
<td>103</td>
</tr>
<tr>
<td>Bastarrica, María Cecilia</td>
<td>47</td>
</tr>
<tr>
<td>Blouin, Dominique</td>
<td>140</td>
</tr>
<tr>
<td>Bollati, Veronica Andrea</td>
<td>88</td>
</tr>
<tr>
<td>Briones, Javier Fernández</td>
<td>103</td>
</tr>
<tr>
<td>Chenouard, Raphaël</td>
<td>2</td>
</tr>
<tr>
<td>Chipriano, Vanea</td>
<td>18</td>
</tr>
<tr>
<td>De Miguel, Miguel</td>
<td>103</td>
</tr>
<tr>
<td>Dhoubi, Saadia</td>
<td>140</td>
</tr>
<tr>
<td>Diguet, Jean-Philippe</td>
<td>140</td>
</tr>
<tr>
<td>Fraternali, Piero</td>
<td>145</td>
</tr>
<tr>
<td>Granvilliers, Laurent</td>
<td>2</td>
</tr>
<tr>
<td>Kermarrec, Yvon</td>
<td>18</td>
</tr>
<tr>
<td>Laarman, Alfons</td>
<td>119</td>
</tr>
<tr>
<td>Laurent, Johan</td>
<td>140</td>
</tr>
<tr>
<td>López-Romero, Fernando</td>
<td>34</td>
</tr>
<tr>
<td>Marcos, Esperanza</td>
<td>88</td>
</tr>
<tr>
<td>McQuillan, Jacqueline</td>
<td>63</td>
</tr>
<tr>
<td>Power, James</td>
<td>63</td>
</tr>
<tr>
<td>Rivera, Jose E.</td>
<td>34</td>
</tr>
<tr>
<td>Ruiz-González, Daniel</td>
<td>34</td>
</tr>
<tr>
<td>Seidl, Martina</td>
<td>134</td>
</tr>
<tr>
<td>Senn, Eric</td>
<td>140</td>
</tr>
<tr>
<td>Silva, Juan Pedro</td>
<td>103</td>
</tr>
<tr>
<td>Soto, Ricardo</td>
<td>2</td>
</tr>
<tr>
<td>Tisi, Massimo</td>
<td>145</td>
</tr>
<tr>
<td>Turki, Skander</td>
<td>140</td>
</tr>
<tr>
<td>Vallecillo, Antonio</td>
<td>1, 34</td>
</tr>
<tr>
<td>Vara, Juan Manuel</td>
<td>88</td>
</tr>
<tr>
<td>Vela, Belén</td>
<td>88</td>
</tr>
<tr>
<td>Vignaga, Andrés</td>
<td>47</td>
</tr>
<tr>
<td>Vénisse, Mathieu</td>
<td>154</td>
</tr>
<tr>
<td>Wagelaar, Dennis</td>
<td>78</td>
</tr>
<tr>
<td>Wimmer, Manuel</td>
<td>134</td>
</tr>
<tr>
<td>Yie, Andres</td>
<td>78</td>
</tr>
</tbody>
</table>