ATC: A Low-Level Model Transformation Language

Antonio Estévez[1], Javier Padrón[1], E. Victor Sánchez[1] and José Luis Roda[2]

[1] Open Canarias, S. L., C/. Elías Ramos González, 4 - Oficina 304,
38001 Santa Cruz de Tenerife, Spain

[2] Dpto. Est., I.O. y Computación, ETSII, Grupo Taro, Universidad de La Laguna,
Camino San Francisco de Paula S/N, Campus Anchieta,
38271 La Laguna, Spain

Abstract. Model Transformations constitute a key component in the evolution of Model Driven Software Development (MDSD). MDSD tools base their full potential on transformation specifications between models. Several languages and tools are already in production, and OMG's MDA is currently undergoing a standardization process of these specifications. In this paper, we present Atomic Transformation Code (ATC), an imperative low-level model transformation language which decouples user transformation languages from the underlying transformation engine. Therefore work invested on this engine is protected against variations on the high-level transformation languages supported. This approach can ease the adoption of QVT and other language initiatives. Also it provides MDA modeling tools with a valuable benefit by supporting the seamless integration of a variety of transformation languages simultaneously.

1 Introduction

In the past few years, Model Driven Software Development, MDSD has successfully positioned as one of the most promising strategies in the future of software engineering [3]. OMG's MDA [2, 7] is a MDSD approach to software development based on models and aimed to provide automation through the various phases in the software development process lifecycle. It follows several standards, which include MOF [9] for describing metamodels, UML [10] for systems modeling and XMI [11], which can describe UML models in the XML format and promote tools' interoperability.

Model transformations in particular allow MDA to achieve automatic model evolution and code generation, and to generally increase productivity in software systems development. Languages to express these transformations have been devised and both commercial and open-source supporting tools capable of performing these activities have been released and are already used in production.

The MOF 2.0 Query, Views and Transformations specification, QVT [8] is particularly relevant to MDA, as it is an attempt to standardize the activities related to

Estévez A., Padrón J., Victor Sánchez E. and Luis Roda J. (2006).
ATC: A Low-Level Model Transformation Language.
In *Proceedings of the 2nd International Workshop on Model-Driven Enterprise Information Systems*, pages 64-74
DOI: 10.5220/0002472000640074

automated transformations between models. This specification will define automatic ways to apply queries, obtain views and execute transformations on models.

Experience gained during the development and handling of our transformation tool's first version, the Business Object Adaptor, BOA [12], has helped us detect several problems that we have solved in our new environment, which is now based on metamodeling and elaborating transformations as opposed to the former procedure of performing transformations in a single, monolythic big step based on XSLT templates.

The long-term goal we pursue is to develop a fully compliant MDA IDE, so the issue of transformation standards affects us deeply. The topic we'll discuss here is the ATC model transformation language. It was born with the aim of shielding our implementation work against changes in the specifications of those supported transformation languages, which inevitably change over time. Now it serves as the low level ground upon which higher-level languages, including the ones that are to become standards, will be integrated in our framework.

This paper is structured as follows: Section 2 briefly outlines the main characteristics of the latest QVT-Merge group standard proposal. Section 3 details ATC and its main components. Section 4 shows an ATC transformation syntax example. Section 5 further discusses ATC issues. In Section 6 related and future work are presented. We end with the conclusions in section 7.

2 QVT-Merge Submission v2.0

In 2002, OMG opened the QVT standardization process with the publication of MOF 2.0 QVT-RFP. As many as 8 proposals were submitted by different organizations and companies during the subsequent two years [6]. In time these organizations gradually converged into one single group, known as *QVT-Merge*. When the third review of this group's proposal (version 2.0) [13] was published, the latest to date, the group was already backed by almost every previous submitter in an effort to speed up the standardization process. At the time of this writing, this third review of the submission of the QVT-Merge group has become the only proposed candidate for the future QVT standard and it is expected that the final QVT standard will quite resemble, or at least be derived from this proposal, so here we show some of its main features.

2.1 QVT-Merge Layers and Languages

The QVT-Merge proposal covers a thorough specification of three transformation languages. The *Relations* and *Core* languages are declarative, and the *Operational Mappings* is imperative.

The specification contains the abstract and concrete syntaxes and the semantics of the three languages. Their detailed metamodels are shown in UML class diagram notation, and concrete syntaxes are described in EBNF. Hybrid collaboration between the imperative and declarative languages is also specified, along with supporting mechanisms for black-box invocations (see figure 1).

The proposal describes steps to convert *Relations* instances into equivalent *Core* syntaxes, which are lower-level. The idea behind this seems to be that compatible tools need only interpret transformation definitions written in *Core*, as those specified in *Relations* could be supported by compiling them to an analogous *Core* syntax.

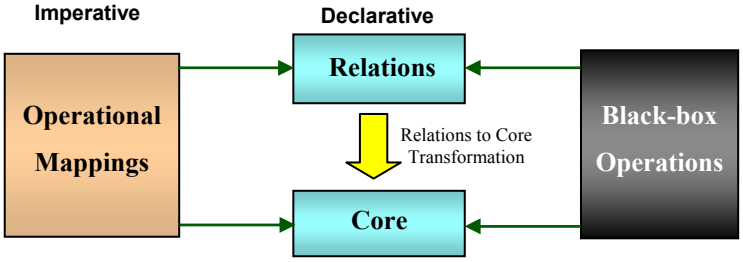

Fig. 1. Description of the QVT-Merge Languages.

3 Introducing the ATC Language

Usually there is no freedom in the election of transformation languages inside tools, since it's hard if not impossible to use any other than the one they come with by default. This also means that while MDA consolidates a QVT standard, tools that opt to give it support are expected to face a rather high amount of refactoring in some cases.

Our primary goal is to provide a tool based as much as possible on industry standards. This is why the pending QVT standard has become a serious problem for us, as the election of supported transformation languages at release time is unclear.

In the meantime we've focused on coding parts that are not affected by the final set of supported languages selected. For instance, the implementation of the lowest-level model transformation mechanisms, and on the establishment of a management infrastructure around models and metamodels. This was the first part of the work and what has emerged from it is ATC and its related transformation engine.

ATC (Atomic Transformation Code) is a general purpose model transformation language designed to operate at the lowest possible level of abstraction, so it is a somewhat harsh, verbose language. Above ATC, those higher-level transformation languages subject to receive support in our environment, will be accommodated through compilation. They are channelled and integrated into our framework to reach the ATC underlying transformation engine, named Virtual Transformation Engine (VTE) by being parsed and compiled into a set of equivalent ATC instances. This is why ATC must be Turing complete in the lax sense.

The engine VTE was created with the sole purpose of understanding and executing ATC instances, which carry information about how to transform models, and it is already being applied successfully in several projects.

There's currently an implementation of ATC and VTE in Java over Eclipse and the Eclipse Modeling Framework (EMF) [4] platforms. Its architecture is depicted in figure 2. Metamodels in this environment are described in terms of the EMF meta-metamodel, named *Ecore*, and equivalent to MOF.

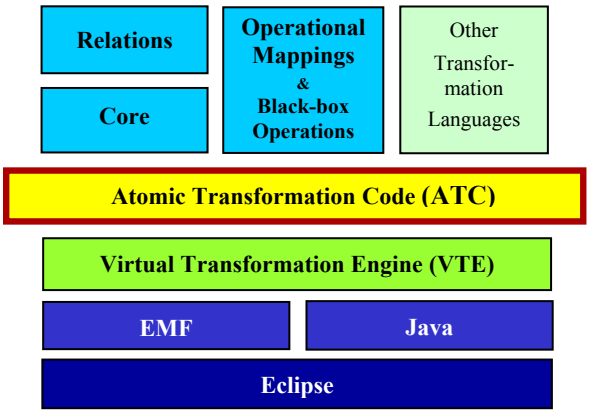

Fig. 2. Architecture layering of a particular transformation environment involving ATC.

3.1 Elements of a Transformation Instance

An ATC transformation consists mainly of:

- an arbitrary number of parameters which represent the participant models,
- identifiers for their related metamodels,
- a set of functional operations that store a hierarchy of semantic objects,
- ATC atoms, the semantic objects in question, each with its own runtime state

Model parameters for a transformation can be read-only, modifiable or created from scratch. Any sort of configuration is supported, such as a single existing model to be modified, or a new one to be created from scratch, one-to-one or many-to-many configurations, and anything that goes in-between.

Execution starts with the transformation's *main* operation call. Like every other ATC functional operation, *main* contains a body filled with atoms arranged sequentially. Some atom types are designed to hold others inside, so finally we get a hierarchy of objects representing the whole ATC transformation information. Therefore it is possible to base the persistence of an ATC transformation as a model serialization, for instance, in an XMI format file, just like EMF does. Interoperability of ATC instances with other tools will be granted as long as XMI compatibility is guaranteed.

Atoms check their state when executed to properly carry out their duties. For instance, state information often tells the engine which model fragments are to be processed. Each atom represents a kind of indivisible byte code with a minimum degree of abstraction, which is why we give it the atomicity condition.

3.2 Language Description

Currently the ATC metamodel contains over one hundred elements among enumerations, data types and classes. As it is impossible to discuss them all here, we'll summarize a classification of the ATC element types.

ATC Expressions. A class identified as an atom type represents a particular atomic transformation semantic unit. Each atom type inherits from a base class named *Expression*, which contains an abstract method with the following signature:

AtcExpression act(TransformationContext tc)

in the reference VTE implementation, which encloses the particular semantic information that makes atom types distinct from each other. It usually comprises no more than twenty lines of code. The *tc* parameter keeps track of contextual information, which includes the local variable registry, the calling stack, as well as parameters and additional runtime information the engine needs to execute the transformation accordingly.

Execution Flow. A list of expressions related with the execution flow include, but is not limited to: *AssignVars*, *Block*, *ExceptionThrow*, *FlowOpReturn, ForEach*, *GetObjectsOfType*, *If*, *InvokeOp*, *InvokeTransformation*, *While*. For instance, a *Block* atom simply encloses a list of atoms to be executed sequentially. *ForEach* takes a data collection as source, usually containing model elements, and traverses it to apply certain actions (which can in turn be an InvokeOp or a *Block* carrying further atoms). *FlowOpReturn* works similar to a return statement but its effect on execution flow is indirect.

Model Transformation. Atom types dealing with model handling and modification include: *CloneModelObject, CreateDataType, CreateModelObject, CreateModel, GetStructuralFeature, SetStructuralFeature*, and those that deal with the contents of lists or other collection types, which are relevant for attributes with multiplicity > 1.

Query and Pattern Matching. Atom types for queries provide us with means to organize data on models and reach particular model types: *GetAllModelElements, GetObjectsOfType*, *GetModelExtent*. The last one delivers the root elements that form a model fragment. Its state information includes the local variable identifier that refers to the piece of model to be queried. Pattern matching is performed explicitly in ATC. It can be achieved if we have means to apply reflectivity over model elements. ATC comes with *IsOfType*, which can be programmed to either perform exact type match or to detect subclasses of a particular model type.

ATC Specific Types. ATC comes with: *Bool*, *Float*, *Int*, *String*. Types in ATC also subclass *Expression*, so their instances are also atoms, and as such, can become the return value of an atom's *act* execution. *String* atoms come enhanced to support common string operations. Among them we find upperization of selective parts, length delivery and substring support. *Null* is a special ATC type that does nothing on its own but can help us detect null assignments in local variables.

Arithmetic and Logical Expressions. Many atom types are built around the ATC specific types. Most of them give support to arithmetics: *Add, Subtract, Multiply, Divide* and *Modulus*, and logics: *And, Or, XOr, Negate, Equals*. These atoms deal transparently with both the ATC types and the primitive types of the native language in which the engine is programmed. Thanks to the *Expression* nature of ATC types and this transparency, these operations can be chained to provide a single final result from a group of combined operations without having to store partial results.

There are still other metamodel elements left, like those that are directly related with the transformation itself: *Transformation, ModelParameter*, *Metamodel*, ...

4 Transformation Examples

We have built a compiler for the *Operational Mappings* language, which is currently quite mature. Certain issues in its EBNF definition and several ambiguities found in its syntax and semantics specification have been sorted out. Future modifications made to the language will be incorporated in the compiler so it adjusts its output accordingly. Obsolete ATC instances will be recompiled. As long as the VTE engine remains unmodified, any other high-level language already supported gets unaffected.

In this section, two small pieces of the *Encapsulation* transformation are presented both in the QVT-Merge Operational Mappings language and its equivalent ATC instances. To make things more interesting, we'll show the ATC transformation definition version from the beginning.

4.1 Mapping Definition

Text in Operational Mappings.

```
mapping inout Property::privatizeAttribute () {
    visibility := "private";
}
```

ATC Equivalent.

```
<atc:AtcTransformation xmi:version="2.0"
   [...] xmlns:atc="http://boa.opencanarias.com/atc/0.5"
name="Encapsulation">

  <metamodels name="UML2">
    <packagesNsURI>http://mset.opencanarias.com/uml2/1.0.0/UML2
    </packagesNsURI>
  </metamodels>

  <modelParameters name="uml2Model" dirKind="inout"
    metamodelId="UML2"/>

  main="//@ownedOperations.7">

  <ownedOperations xsi:type="atc:AtcMapping"
    name="privatizeAttribute">
    <formalParameters xsi:type="atc:AtcMappingParameter"
      name="a" dirKind="inout" typeQualifNm="UML2::Property">
    </formalParameters>
    <mBody xsi:type="atc:AtcBlock">
        <atcAtoms xsi:type="atc:AtcCreateDataType"
          packageNsURI="http://mset.opencanarias.com/
             uml2/1.0.0/UML" dataTypeNm="VisibilityKind"
          sourceString="private" targetId="localVar1"/>
        <atcAtoms xsi:type="atc:AtcSetStructuralFeature"
           ObjectId="a" stFNm="visibility"
           featureVarId="localVar1"/>
    </mBody>
  </ownedOperations>
  <ownedOperations [...]
```

Explanation. The first example consists of the complete definition of a small mapping, where an enumerated type is generated and assigned to a UML2 attribute of a *Property* instance, which represents the contextual parameter for the mapping call.

To keep things clean we have omitted the Operational Mappings' trace information that stores bindings created between model objects during execution, and that can be queried later on during the same transformation. This information is embedded explicitly in the ATC transformation instances during compilation.

Metamodels are defined outside the transformation block. A metamodel is made up of a list of URIs. They refer to Ecore packages in our case. In this example only the URI of our UML 2 metamodel is present. Model parameters for the transformation follow. We can identify the *main* operation as being the operation number 7.

Finally a full list with the transformation functional operations follows. Only *privatizeAttribute* is shown here. Its type is *AtcMapping*. During compilation, the contextual parameter has lost the privileged position it held in Operational Mappings to become an ordinary parameter, the first in the list. The *visibility* assignment, which spans a line in Operational Mappings, has ended up being a *Block* containing two ATC atoms. None of them acts as a container for other atoms.

The first atom obtains a *private* instance of the *VisibilityKind* enumeration type. The second atom will assign it to the *visibility* attribute of the *Property* instance, whose name identifier is '*a*'. The '*localVar1*' variable identifier is used as a *key* in a map of Java variables in order to store its associated value. It is the link established between both atoms. The behaviour of *AtcSetStructuralFeature* is straightforward.

4.2 Mapping Call

Text in Operational Mappings.

```
{
    var attrs := c.ownedAttribute;
    attrs->map privatizeAttribute();
}
```

ATC Equivalent.

```
<atcAtoms
    xsi:type="atc:AtcGetStructuralFeature"
      objectId="c" stFNm="ownedAttribute"
      featureVarId="attrs"/>
  <atcAtoms xsi:type="atc:AtcForEach"
    collectionId="attrs" elementId="localVar12">
    <forBody xsi:type="atc:AtcBlock">
      <atcAtoms xsi:type="atc:AtcInvokeOp"
        op="//@ownedOperations.0">
          <actualParameterIds>localVar12</actualParameterIds>
      </atcAtoms>
   [...]
```

Explanation. The first atom we see here, which is equivalent to the first line in the Operational Mappings sample, defines a new variable, '*attrs*', which represents the list of properties belonging to a given class whose identifier is '*c*'. In the next line a hidden traversal syntax takes place, so that the mapping invocation is produced over every element in *attrs*. This syntax becomes explicit in ATC, as can be seen by the

'*localVar12*' temporary variable managed by the *ForEach* atom. The body for this *ForEach* is merely the *AtcInvokeOp* invocation of the operation number 0, which happens to be *privatizeAttribute*. '*localVar12*' is the actual parameter identifier.

5 ATC Considerations

Imperative Nature. Declarative descriptions are often naturally found in transformation environments. Mappings are established between domain artifacts, and so on. But at the end an algorithmic approach must be followed to reconcile all this information in order to be executed by a machine, so our assumption is that it will be possible to produce ATC explicit imperative representations of those algorithmic semantics. The hard task of evaluating declarative expressions is left to the language compiler. But even if the ATC-based engine is unable to match the execution speed of a direct language supporting engine, we'll have sacrificed performace in favor of a flexible design

Multi-Language Tools. The capability for simultaneous support of different transformation languages in the same tool is very interesting, provided it doesn't bloat its responsiveness and general performance. As there is no such language capable of solving transformation problems in all kinds of situatios with complete flexibility, power and ease of use, the layering principle allows focusing on the integration of a wide range of high level languages at the hands of the architect in the same development environment. Future adoption of new emerging languages will also be possible.

This diversity opens up the possibility for the joint collaboration of several transformation languages in the sense of Domain Specific Languages (DSL) [5]. Reuse of legacy transformation languages, which in turn can see their longevity increased, is also interesting. Entire repositories of transformation definitions can be translated into ATC versions and related metamodels created to assist them. This has to do with the added value the framework earns each time new EMF metamodels are provided and ATC transformations and transformation language compilers produced.

ATC Unfolding to Java Files. Previously ATC execution performance was penalized because of transformations being model objects to be traversed and the *act* method having to be invoked for each executed atomic unit. Other issues, such as ATC's specific types operation handling, like addition, primitive type assignment, and several ones like variable declaration, comparison or execution flow control checks, came into play to further introduce overhead and slow things down.

Recently we have started a new version for VTE. To keep efficiency at maximum, one last compilation step is now performed, this time to unfold ATC XMI instances into native plain Java code. Each functional operation is analyzed and the code associated to each of its composing atoms' types already available in the previous engine version is sequentially dumped inside a method that represents the entire original operation. Specific arrangements are made for each different processed atom type, and loose pieces of code are glued together inside the overall method. The surrounding class becomes the transformation itself. References to metamodels are automatically embedded in the code.

The outcome of this plain java class matches the structural aspect of the original transformation definition, That is, as many methods are defined in the class as functional operations are present in the original transformation instance. The transformation context is treated slightly different now. To spread useful runtime information it just becomes an explicit additional parameter in every operation (including *main*).

This new mechanism not only considerably helps in boosting performance, but now it's easier to promote blind transformation and operation invocations (black-box), once a contract has been established concerning how to access the entrance point of the transformation and to choose models as actual parameters.

Support for QVT Requirements. We won't discuss here issues about supporting QVT recommended features such as traceability, bidirectionality and incrementality in our framework. These are still open areas for investigation in our case, but we expect the ATC language to be agnostic of these features, as support can be expressed explicitly in its transformation instances (similar to the trace classes infrastructure for *Operational Mappings*) or integrated in the environment through the VTE engine or in parallel with it. This helps keep the language structure focused and compact.

6 Related and Future Work

Following the same low-level principle discussed for ATC, another transformation language, ATL [1], has recently added an imperative virtual machine to its layering architecture with a similar abstraction level but different treatment of core types. Similar examples include the QVT-Merge language pair Relations vs Core. Concerning transformation representations as model objects, the QVT-Partners [14] group has an open transformation implementation based on the composition of semantical units.

As future work, it will be interesting to see how ATC is able to deal with the Relations and Core languages. A compiler for Core will soon start development. We also expect support for other languages to be developed through other research groups.

For the moment, we don't plan to port ATC and VTE to other underlying technologies aside from EMF and Java. We expect to be able to apply the MDA paradigm, so when the ATC metamodel matures to include semantics (it is currently hardwired in the engine), versions for other underlying technologies (such as MDR, or C++) will automatically be generated. Other future challenges include exploring migration issues about the integration of ATC and VTE implementations in foreign MDA tools.

7 Conclusions

In this paper we have detailed the ATC model transformation language, its composition and set of instructions, and its underlying execution mechanisms. ATC is a low-level, imperative language designed for model transformations and thus, not quite user-friendly. We've also introduced VTE, its supporting transformation engine,

which doesn't understand any other language and to which a plain java compilation process has recently been added to enhance runtime performance.

We've discussed the role of ATC in the transformation tools as an intermediate layer that assists in the integration of the common abstract transformation languages. Integration of each language is achieved by means of a compilation module that produces semantically equivalent instances in the ATC syntax.

We consider the current QVT-Merge standard proposal a reference regarding high-level transformation languages suitable for common use by transformation engineers. Here its architecture has been depicted. A compiler for ATC specifically tailored to translate *Operational Mappings* instances is already available. Two examples show what ATC instances look like and how they compare to their equivalent high-level language original syntax. We've also presented several other transformation languages each sharing certain similarities with ATC or with the transformation engine design.

We've described how the layered arrangement brings benefits to the unification of a transformation tools' architecture, with a single central engine able to give support to many simultaneous languages. Newer and older languages can coexist in the same environment and even complement each other. Finally this layering can help tools in their adoption of the upcoming QVT standard.

Acknowledgements

This paper has been supported by the *Ministerio de Educación y Ciencia* (PTQ2004-1495) and the *Fondo Social Europeo*. We would like to thank *IZFE (Diputación de Guipúzcoa)*, the *Excmo. Cabildo Insular de Tenerife* and the *DG de Universidades e Investigación del Gobierno de Canarias*, for their overall support of this work.

References

1. ATL, The Atlas Transformation Language, http://www.sciences.univ-nantes.fr/lina/atl/
2. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture, Practice and Promise. Addison-Wesley (2003)
3. Butler Group Application Development Strategies Report, http://www.butlergroup.com/reports/ads/
4. Eclipse Modeling Framework (EMF), http://www.eclipse.org/emf
5. Fowler, M.: MF Bliki: DomainSpecificLanguage, http://www.martinfowler.com/bliki/DomainSpecificLanguage.html
6. Gardner, T., Griffin, C., Hauser, R., Koehler, J.: A Review of OMG MOF 2.0 QVT Submissions and Recommendations Towards the Final Standard. 1st International Workshop on Metamodeling for MDA, York, UK (2003)
7. Mellor, S., Scott, K., Uhl, A., Weise, D.: MDA Distilled. Principles of Model Driven Architecture. Addison Wesley, 2004
8. OMG, MOF 2.0 Query/Views/Transformations RFP, OMG Document ad/2002-04-10 (2002)
9. OMG, Meta-Object Facility (MOF), http://www.omg.org/mof

10. OMG, Unified Modeling Language (UML), http://www.uml.org
11. OMG, XML Metadata Interchange (XMI), http://www.omg.org/technology/documents/modeling_spec_catalog.htm#XMI
12. Padrón, J., Estévez, A., Roda, J.L., García, F.: An MDA-Based Framework to Achieve High Productivity in Software Development. Software Engineering and Applications, Track 436-218 (2004)
13. QVT-Merge Group, Revised Submission for MOF 2.0 Q/V/T RFP, OMG Document ad/2005-03-02
14. QVT-Partners, http://qvtp.org