Loosely Coupled Traceability for ATL

Frédéric Jouault

ATLAS Group, INRIA and LINA,
University of Nantes, France
`frederic.jouault@univ-nantes.fr`
`http://www.sciences.univ-nantes.fr/lina/atl/`

Abstract. In the context of model transformation, traceability information may be used in a wide variety of scenarios. Each of them potentially requires a different format or complexity level. Moreover, a single transformation program can be used in several contexts. Consequently, such a program may need to be able to generate different kinds of traceability information. This work aims at showing how traceability can be added to programs written in ATLAS Transformation Language while limiting dependencies to program logic. Model transformation is used to implement this approach.

1 Introduction

Model transformation is widely recognized as a central ingredient of model engineering approaches. The QVT RFP [10] issued by OMG and the resulting answer by QVT-Merge [11] as well as ATLAS Transformation Language [2] [8] are two examples of model transformation solutions among many. Although they focus on transformation specification and execution, many approaches make use of traceability. In the context of this paper, we will use the following definition for model transformation traceability. A language or engine that is able to maintain a set of relations between corresponding source and target model elements is said to support traceability. Links to program elements responsible for these relations (e.g. a rule specifying how to transform a set of source elements into a set of target elements) may also be kept.

We anticipate that requirements on traceability information will vary in at least two dimensions: *range* and *format*. The *range* of a traceability use corresponds to the proportion of elements for which links are maintained. For instance, only a subset of all rules that constitute a transformation program may generate traceability links. Although full traceability (i.e. keeping every possible link) might be theoretically feasible in most cases, it is not necessary for applications where only a part of this information is actually useful. In some cases, for scalability and performance reasons, traceability elements that will not be used cannot be created and subsequently filtered out: they must not be created at all.

The *format* of traceability information corresponds to the way links are encoded. While some applications only need simple links to model elements, others will require more complex encodings. For instance, keys, identifiers or path expressions may have to be computed from simple links.

To make things even more complex, a given transformation program may actually need to support several kinds of traceability. This includes support for incompatible *ranges* and *formats* that may not even be known at program design or development time.

We provide solutions for these problems by showing how to attach traceability generation code to pre-existing ATL programs. Two simple considerations make this relatively easy: transformation programs are considered as models and so is traceability information. This means that we can add code into a program with a transformation and that this code can create traceability the same way other target model elements are created.

With this approach, 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.

The paper is organized as follows. Section 2 presents our approach. Section 3 gives a discussion on how this approach provides loose coupling as well as range and format adaptability. Section 4 concludes the paper.

2 Presentation of our approach

Section 2.1 presents a very simple case study that will be used in the rest of the paper. As was said in introduction, our approach is based on two essential considerations. In section 2.2 we show how considering traceability information as a model enables simple coding of traceability generation. In section 2.3 we present how considering transformation programs as models allows inserting traceability generation code into pre-existing ATL programs.

2.1 Case Study

Let us consider two simple metamodels presented in Fig. 1: *Src* (source) and *Dst* (destination). *Src* contains a single class named A, which in turn contains a single *name* attribute. *Dst* is identical with the exception of the class name, which is now B.

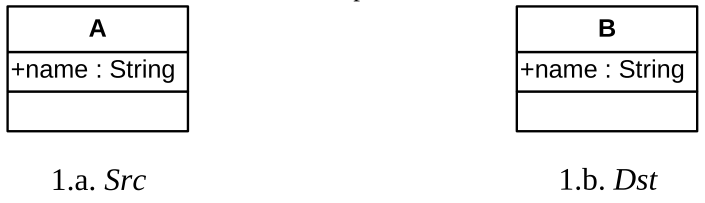

Fig. 1. The *Src* and *Dst* metamodels

We consider the transformation from *Src* to *Dst* that translates every A into a B having the same name. ATL code implementing this is given below. A transformation module, named *Src2Dst* (line 1), is declared as transforming *Src* models into *Dst* models (line 2). A single rule, *A2B* (lines 4-11), implements the translation of *A* elements (line 6) into *B* elements (lines 8-10).

```
1.      module Src2Dst;
2.      create OUT : Dst from IN : Src;
3.
4.      rule A2B {
5.              from
6.                      s : Src!A
7.              to
8.                      t : Dst!B (
9.                              name <- s.name
10.                     )
11.     }
```

2.2 Traceability as a Model

The principle "everything is a model" may be viewed as the central driving force in the present evolution of model engineering [1]. It therefore seems natural to consider traceability information as a model, more precisely as an additional target model of a transformation program.

This means that we can create traceability elements in the same way other target model elements are created. No additional language construct or execution engine modification is required, provided several target models are supported. This is the case with ATL. Additionally, once we have a traceability model, we may apply any model operation on it, including transformations.

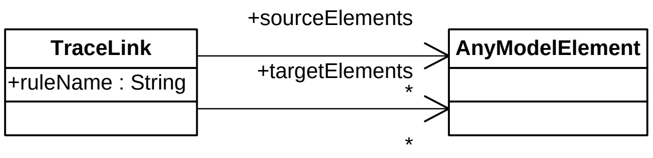

Fig. 2. The *Trace* metamodel

Since traceability is a model, it needs a metamodel. Fig. 2 presents a simple *Trace* metamodel able to play this role. It contains a *TraceLink* class owning a *ruleName* attribute to store transformation rule name and pointing to *AnyModelElement* via two multivalued references: *sourceElements* and *targetElements*. *AnyModelElement* is not a simple class. It is required here because our *Trace* metamodel must be able to link to elements of other metamodels (namely *Src* and *Dst).* They are several possible concrete implementations of this. For instance, using EMF (Eclipse Modeling Framework) [3], *AnyModelElement* can be replaced by a link to class *EObject* specified in Ecore metamodel. We will not discuss this point further in this paper. We now consider that one solution is actually in use and will proceed with the example.

Although we believe any form of traceability can actually be achieved using our approach (see discussion in section 3), we only present here a simple traceability scenario, where every named link between source and target elements is kept as a *TraceLink*.

To integrate traceability in our *Src2Dst* transformation program, we can now add a target pattern element to generate a *TraceLink* element. This results in the following program:

```
1.      module Src2DstPlusTrace;
2.      create OUT : Dst, trace : Trace from IN : Src;
3.
4.      rule A2BPlusTrace {
5.              from
6.                      s : Src!A
7.              to
8.                      t : Dst!B (
9.                              name <- s.name
10.                     ),
11.                     traceLink : Trace!TraceLink (
12.                             ruleName <- 'A2BPlusTrace',
13.                             targetElements <- Sequence {t}
14.                     )
15.             do {
16.                     traceLink.refSetValue('sourceElements',
17.                             Sequence {s});
18.             }
19.     }
```

An additional target model is specified in transformation header (line 2) to store trace links. A target pattern element (lines 11-14) has been added to rule *A2B* (renamed in *A2BPlusTrace*, line 4) to create a *TraceLink*. A single-statement imperative block has also been added to initialize *sourceElements* (lines 15-18). As a matter of fact, this property cannot be initialized using an additional binding like *targetElements* is. This is because of ATL resolve algorithm (see [8]), which is called for each binding and resolves source elements into target elements. This is not what we want in our special transformation scenario. We therefore use an alternative way of setting *sourceElements*, without resolving the value (line 16-17).

We now have traceability in our transformation program without using any specific language or engine support for it. This however required manually adding the pieces of ATL code described in previous paragraph.

2.3 Transformation Program as a Model

We have just seen how transformation programs can be relatively easily extended to support traceability generation if we consider traceability information as a model. We are now going to analyze how the fact that transformation programs are models can help too.

Since transformation programs are models, we can transform an ATL program into another ATL program by model transformation. Thus, an ATL program can be written to automatically insert the traceability creation code described in previous section. We developed such a program and called it *TracerAdder* (it effectively adds a tracer into an ATL program). ATL source code of this program is given in the appendix.

TracerAdder operates in ATL refining mode[1] and is therefore a kind of in-place transformation. Rule *Module* inserts the additional target traceability model. This is done by generating two additional (i.e. having no direct corresponding element in source model) abstract syntax elements of type *OclModel*. They represent the new target model and metamodel. Rule *OutPattern* inserts the additional target pattern element and the imperative statement to set *sourceElements* (cf. section 2.2). Here again, additional abstract syntax elements are generated to represent the tracing code. All other elements are simply copied from source to target.

TracerAdder should be used just before actual ATL compilation. Therefore, it can be considered as a precompiler. But it is neither a token stream preprocessor like in C nor a bytecode instrumentation tool operating at virtual machine level like [4] and [5] for Java. It directly operates on abstract syntax.

3 Discussion

We have presented in previous section a method to automatically extend ATL programs so that they generate traceability information. We will now discuss this solution.

3.1 Loosely Coupled Traceability

One of the main advantages of our solution is that traceability generating code (TGC) is not tightly coupled to program logic. As a matter of fact, program logic does not depend on it and TGC can even be added to a program that was not designed for it. We have actually experimented this on an old example that was lying around. TGC does not highly depend on program logic either.

Moreover, several TGC can be attached to the same program as required by a specific application. This means that a given transformation program can be automatically adapted to specific *range* or *format*, as defined in introduction.

TGC is currently embedded into transformation code, as part of *OutPattern* rule. A more complex version of *TracerAdder* could take a second model as input. This additional model would contain the TGC to inject. This TGC could then be represented in a simpler format than it currently is.

Libraries of traceability adding programs like *TracerAdder* could also be developed. This means we could have orthogonal (i.e. any member of one can be used with any member of the other) transformation programs libraries on one side and traceability adders libraries on the other side.

[1] In ATL, source models are read-only and target models are write-only; this prohibits in-place transformations. However, ATL provides a replacement mechanism: refining mode [8]. In this mode, unmatched source elements are automatically copied into target model.

3.2 Example Extensions

The example used in this paper is very simple. We have however tested the same *TracerAdder* program on a more complex example with success. There is no specific reason why it should not scale to more complex examples as long as they are declarative only. As a matter of fact, although imperative code could also be instrumented, it is more difficult to recognize source-target dependencies (i.e. what we want to represent as traceability links) in imperative code, where they are implicit, than it is in declarative code.

Traceability metamodel is not fixed and traceability models can always be further processed into any *format*. As for *range*, *TracerAdder* could be easily modified to not add TGC to specific rules, for instance, and thus reduce trace model size.

Although an EMF-specific feature was used to implement inter-model linking, other possibilities exist. For instance, model weaving [6] could be used instead and the AMW tool (available on GMT [7], along with ATL tools) could be used to browse traceability models.

In section 2.3 we said our approach operates on abstract syntax. It can of course be made to work on concrete syntax by adding a parser and a serializer respectively before and after the transformation.

3.3 Other Implementation Options

ATL, like many declarative languages, has built-in support for traceability. This is necessary to link elements generated by different transformation rules together. A target element may for instance be referred to by using its corresponding source element as a key. This is used extensively in ATL, especially by resolve algorithm. Such a form of traceability need however not persist after executing a transformation. For this reason, we call it *internal traceability*.

Conversely, *external traceability* corresponds to the ability for a language or engine to support persistence of traceability relations beyond transformation execution. When *internal traceability* is supported by a tool, a simple kind of *external traceability* can often be easily implemented. As a matter of fact, simply serializing the links internally used provides traceability information. In the case of ATL, we would obtain the whole set of links between corresponding source and target elements. Each link is labeled by the name of the rule that generated its target elements from its source elements. We do not have, however, all the benefits provided by the approach proposed in this paper such as *range* and *format* adaptability. Besides, this solution requires modifying an execution engine and is limited to this modified engine.

Logging events on target models during transformation execution is another execution-engine-level approach that could be used. However, it suffers from the same drawbacks as serialization of *internal traceability* concerning engine modifications.

Rule inheritance could also be used to attach tracing code to existing rules. It would permit more *range* and *format* adaptability than serialization of *internal traceability*. Coupling of tracing code to program logic could be relatively loose.

Reflection might also provide other means to implement traceability by letting the developer dynamically plug-in tracing code where appropriate. Coupling of tracing code to program logic could be relatively loose too.

We however believe our approach has more advantages in the general case (especially regarding coupling), at least for ATL.

4 Conclusions

We have shown how traceability generating code can be relatively easily added to ATL code. We have moreover shown that this process can be fully automated. We discussed the fact that this method of inserting code results in loosely coupled traceability and can be adapted to virtually any kind of traceability *range* or *format*.

Although we focused on traceability in this paper, we believe that this approach can actually be adapted to other kinds of ATL code instrumentation and even to non-ATL transformations (e.g. Java code instrumentation and refactoring). It indeed mostly consists in using model transformation to perform abstract syntax transformations.

The solution presented in this paper may also be viewed as a kind of aspect weaving [9]. Traceability can actually be considered as a cross-cutting concern that requires repetitive code insertion. But any part of ATL metamodel can actually be extended or transformed. On the other end, it may be harder to check such a transformation for validity.

Acknowledgements

I would like to thank Jean Bézivin, Marcos Didonet Del Fabro, Ivan Kurtev and all the members, past and present, of the AMMA team. This work has been partially supported by ModelWare, IST European project 511731.

References

1. Bézivin, J., On the unification power of models, Software and Systems Modeling, Volume 4, Issue 2, May 2005, pages 171 – 188
2. Bézivin, J., Dupé, G., Jouault, F., Pitette, G., and Rougui, J. E., First experiments with the ATL model transformation language: Transforming XSLT into XQuery, 2nd OOPSLA Workshop on Generative Techniques in the context of MDA, Anaheim, CA, USA (2003)
3. Budinsky, F., Steinberg, D., Raymond Ellersick, R., Ed Merks, E., Brodsky, S. A. and Grose, T. J., Eclipse Modeling Framework, Addison Wesley, 2003
4. Cohen, G. A., J. S. Chase, and D. L. Kaminsky, Automatic Program Transformation with JOIE, in USENIX Annual Technical Conference '98, 1998
5. Dahm, M., Byte Code Engineering with the JavaClass API, Techincal Report B-17-98, Institut für Informatik, Freie Universität Berlin, January 1999

6. Didonet Del Fabro, M., Bézivin, J., Jouault, F., Breton, E. and Gueltas G., AMW: a generic model weaver, In Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles, Jun/Jul 2005
7. Eclipse Foundation, Generative Model Transformer Project, http://www.eclipse.org/gmt/
8. Jouault, F. and Kurtev, I., Transforming Models with ATL, in proceedings of the Model Transformation in Practice Workshop, October 3rd 2005, part of the MoDELS 2005 conference
9. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G., An Overview of AspectJ, Lecture Notes in Computer Science, Volume 2072, Jan 2001, Page 327
10. OMG, MOF 2.0 Query/Views/Transformations RFP, OMG document ad/2002-04-10 (2002)
11. OMG, Revised Submission for MOF 2.0 Query/View/Transformations RFP (ad/2002-04-10), OMG Document ad/2005-07-01 (2005)

Appendix: *TracerAdder* source code

```
module TracerAdder;
create OUT : ATL refining IN : ATL;

rule Module {
	from
		s : ATL!Module
	to
		t : ATL!Module (
			outModels <- s.outModels->append(
				traceModel)
			-- [..] copy every other property
		),
		traceModel : ATL!OclModel (
			name <- 'trace',
			metamodel <- traceMetamodel
		),
		traceMetamodel : ATL!OclModel (
			name <- 'Trace'
		)
}

rule OutPattern {
	from
		s : ATL!OutPattern
	to
		t : ATL!OutPattern (
			elements <- s.elements->append(
				traceLink)
		),

		traceLink : ATL!SimpleOutPatternElement (
			varName <- 'traceLink',
			type <- traceType,
			bindings <- Sequence {ruleName,
				targetElements}
		),
		traceType : ATL!OclModelElement (
                    name <- 'TraceLink',
                    model <- thisModule.resolveTemp(
                           s."rule"."module",
                           'traceMetamodel')
             ),

             ruleName : ATL!Binding (
                    propertyName <- 'ruleName',
                    value <- nameString
             ),
             nameString : ATL!StringExp (
                    stringSymbol <- s."rule".name
             ),

             sourceSeq : ATL!SequenceExp (
                    elements <- sourceVars
             ),
             sourceVars : distinct ATL!VariableExp
                           foreach(e in
                           s."rule".inPattern.elements) (
                    referredVariable <- e
             ),
             actionBlock : ATL!ActionBlock (
             "rule" <- s."rule",
                    statements <- Sequence {stat}
             ),
             stat : ATL!ExpressionStat (
                    expression <- refSetValue
             ),
             refSetValue : ATL!OperationCallExp (
                    operationName <- 'refSetValue',
                    source <- traceLinkVar,
                    arguments <- Sequence {seString,
                           sourceSeq}
             ),
             traceLinkVar : ATL!VariableExp (
                    referredVariable <- traceLink
             ),
             seString : ATL!StringExp (
                    stringSymbol <- 'sourceElements'
             ),

             targetElements : ATL!Binding (
                    propertyName <- 'targetElements',
                    value <- targetSeq
             ),
             targetSeq : ATL!SequenceExp (
                    elements <- targetVars
             ),
             targetVars : distinct ATL!VariableExp
                           foreach(e in s.elements) (
                    referredVariable <- e
             )
}
```