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 *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 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 are 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 specify explicitly the name of the output variable associated with the required element using the *resolveTemp()* API method. This method is the standard way to access the tracing information and is described in more detail in Section 2.

The tracing information often is 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 some 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 select the output elements by their type is not possible in the actual ATL-VM. Furthermore, the ATL virtual machine discards the tracing information after the execution of a transformation rule. This situation forces the ATL users to extend their transformation rule 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 rule is changed it is necessary to inject the 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 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 virtual machine 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 and automatically generate a tracing model together with the target model, having minimal overhead.

2 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. The Figure 1 present an 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 *Tables* elements, and the rule *SingleValuedAttribute2Column* (line 11) will transform *Attributes* into *Columns*. Every time one of these rules is

[1] A called rule automatically executed at the end of the transformation

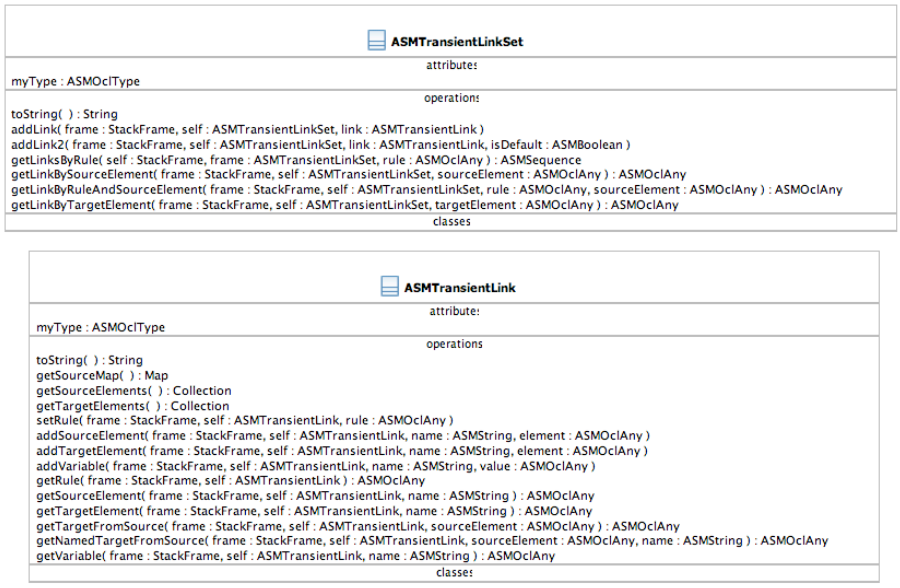

Fig. 1. *ASMTransientLinkSet* and *ASMTransientLink* native types

matched, a new transient link is created and the source and target elements are assigned to it. Subsequently, the rule *Class2Table* will need to assign all the *Columns* generated from the *Attributes* of the *Class* to the generated *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 in the *ASMTransientLinkSet* collection all the *ASMTransientLinks* that have an *Attribute* as source element and collect the target elements for them.

When an ATL transformation uses the *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 *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 (*a.owner*) and assigns the output element with the variable name *'table'*. This output variable belongs to the rule *Class2Table* (line 7).

3 Runtime access to the tracing information

As was mentioned before, occasionally the ATL users require richer access to the tracing information. For instance, to find target elements by type or to iterate over all the *transient links*. In order to do that, it is necessary to recreate the

```
1  module Class2Relational;
2  create OUT: Relational from IN: Class;
3  rule Class2Table {
4   from
5    c: Class!Class
6   to
7    table: Relational!Table (
8     name <- c.name,
9     col <- Sequence {key}->union(c.attr->reject(e|e.multiValued)),
10    key <- Set {key}
11   ),
12   key : Relational!Column (
13    name <- 'Id'
14   )
15 }
16
17 rule SingleValuedAttribute2Column {
18  from
19   a: Class!Attribute (
20    not a.multiValued)
21  to
22   c: Relational!Column (
23    name <- a.name,
24    owner <- thisModule.resolveTemp(a.owner, 'table')
25   )
26 }
27
28 rule MultiValuedAttribute2Column {
29  from
30   a: Class!Attribute (a.multiValued)
31  to
32   t: Relational!Table (
33    name <- a.owner.name + '_' + a.name,
34    col <- Sequence{id, value}),
35   id: Relational!Column (
36    name <- 'Id'),
37   value: Relational!Column (
38    name <- a.name
39   )
40 }
```

Listing 1. Class2Relational transformation module

tracing links in a different model. However, we can simplify this process giving access to the implicit tracing information to the ATL user.

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 such as the rules presented in Listing 2. The purpose of these rules 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 *getTraceLink* rule for every element in the *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 create *Traced Elements* for the source and target elements. Although, the presented rules are purely imperative, they offer a simple solution to generate a trace model from the ATL's implicit tracing mechanism.

This method offers two main advantages with respect to the HOT method presented before: 1) It is possible to add a *superimposed* module [4] to the previous rules to generate the tracing model keeping the original rule and the tracing specific rule nicely separated in two different modules. 2) The rules used to generate this model are simpler and easy to change in order to use a customized metamodel or add specific information. 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 **12.4** seconds, this is **136%** more than the normal execution of the same transformation that took in average **5.3** seconds[2].

4 Automatic storing of the tracing information

The second method that we present is the most end-user friendly because automatically stores the tracing information in a tracing model after the execution of the transformation rule. The ATL user must select the generation of the tracing model in the advanced launch configuration[3] and give a name and path for output tracing model[4]. The launch configuration dialogs are presented in Figure 2.

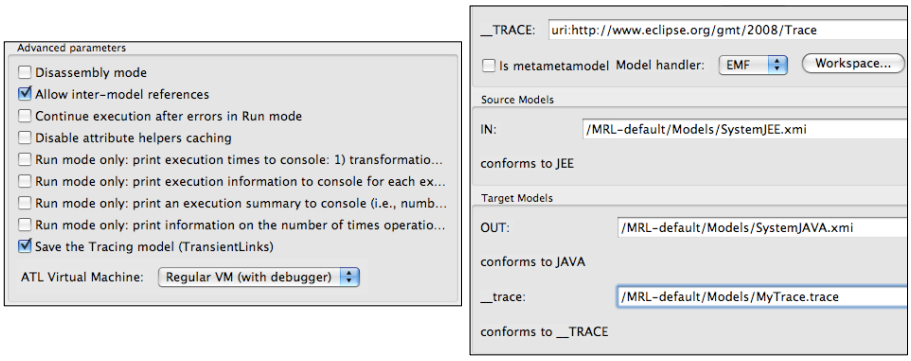

Fig. 2. ATL Launch configuration dialogs

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

[2] The experiment code can be downloaded from `http://ssel.vub.ac.be/svn-pub/ATLTrace/TracingBenchmark`

[3] Provided that the option *Allow Inter-model references* in the advanced launch configuration is activated

[4] 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 presented in Figure 3 has a *TransientLinkSet* and a *TransientLink* meta-classes 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.

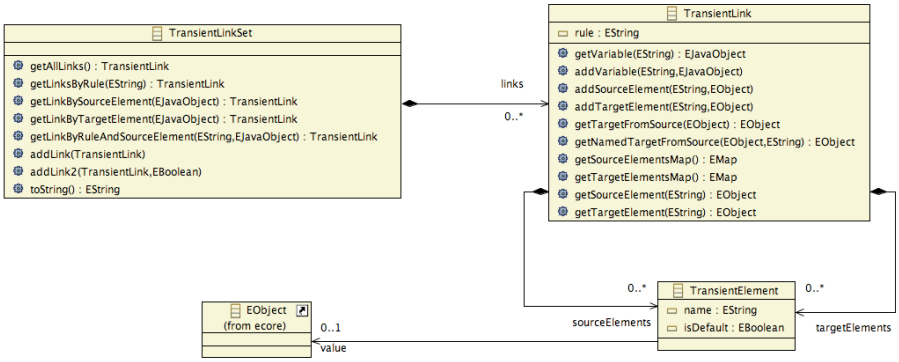

Fig. 3. Tracing Metamodel

4.2 Bytecode adaptation

The second step is the bytecode adaptation, when the ASM file is loaded. The adaptation replaces all the native implementations of *TransientLinkSet* and *TransientLink* for 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* instuctions. Therefore, when the ATL-VM executes the

adapted instructions, instances of our EMF metaclasses *TransientLinkSet* or *TransientLink* are created instead of the native ones. The standard and adapted instructions are presented in the Figure 4.

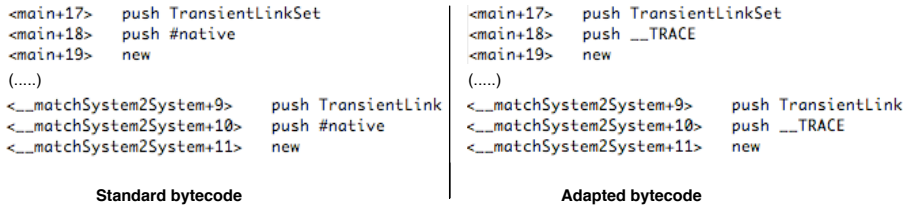

Fig. 4. ASM bytecode Adaptation

In the transformation execution the ATL implicit 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.

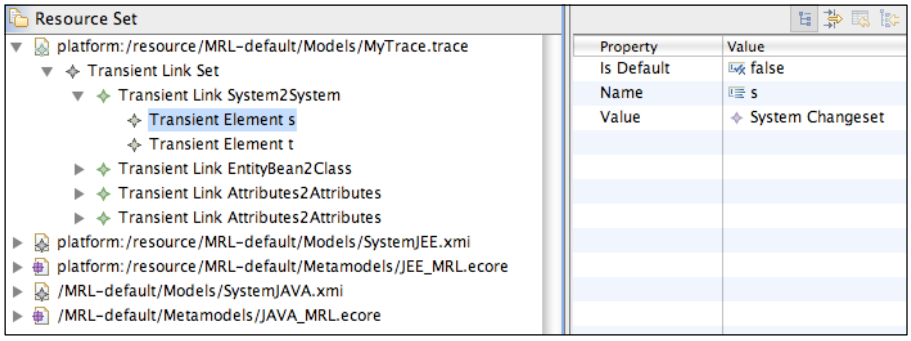

Fig. 5. Tracing Model

The main advantages of this method is that it does not requires changes to the original transformations, is end-user friendly and the performance has low 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 de end user, we offer the same richer access to the implicit tracing mechanism as the method presented in Section 3. In a similar fashion to that used by Listing 2, a customized tracing model can be generated using the enriched access provided.

5 Conclusions and future work

In this paper we present a method that enrich the access to the tracing information during transformation execution. This richer access can be used to find target elements by type or with an endpoint rule to generate a tracing model by iterating over all the *transient links* collection. Additionally, we present a method that stores a tracing model after the execution in an 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 access easily the tracing information and to avoid changing their original transformations[5]. The first method offers the possibility of easily generate a custom tracing model by using a customized 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 not repercussions in the performance of the transformations.

However, to have a complete traceability solution for ATL it is necessary to consider several missing features. First, when a chain of transformation is used, is necessary 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, to add information to each tracing link about the implicit dependencies among rules in order to analyze the impact of a change. For instance, to relate the tracing links produced by the *Class2Table* rule and *SingleValuedAttribute2Column* from Listing 1 when an implicit dependency exists. Third, the presented methods do not trace the elements created by a *called rule*. It is necessary to define a representation for this elements and relate them with 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

1. Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model traceability. IBM Systems Journal **45**(3) (2006) 515–526

[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`)

2. Vanhooff, B., Baelen, S.V., Joosen, W., Berbers, Y.: Traceability as input for model transformations. (Third ECMDA traceability workshop 2007)
3. Jouault, F.: Loosely coupled traceability for atl. In: Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany (2005)
4. Wagelaar, D.: Composition techniques for rule-based model transformation languages. In: ICMT '08: Proceedings of the 1st international conference on Theory and Practice of Model Transformations, Berlin, Heidelberg, Springer-Verlag (2008) 152–167