Compiling ATL with Continuations

Jesús Sánchez Cuadrado and Jesús Perera Aracil

Universidad de Murcia (Spain)
jesusc@um.es, jpereraaracil@gmail.com

Abstract. This paper presents a proposal to use continuations as an implementation mechanism for ATL. We introduce the notion of continuation, showing its applicability to model-to-model transformations, and develop a simple mechanism to enable continuations in model transformations. Then, the declarative part of ATL is mapped to this mechanism.

1 Introduction

Model transformations are at the heart of Model Driven Engineering (MDE), since they allow the automation of diverse kind of model manipulations. In the case of model-to-model transformations, a key element of a model transformation language is its mechanism to resolve relationships between model elements, which impact both the design of the language and the engine implementation.

In the context of ATL, rules and bindings are the constructs that allow the declarative specification of mappings and the relationships between them. At the implementation level, a two pass algorithm is enough to ensure that all available model elements are created by the corresponding rules and that bindings get resolved. Even though this algorithm has shown good performance, alternative algorithms may enable advanced features for ATL.

On the other hand, in the compiler construction field continuations have been used as an intermediate representation to enable local optimizations [1]. A continuation allows the execution state of a program to be captured, stopped, and resumed later. This characteristic could be used to devise novel strategies for resolving data dependencies in a model transformation (Section 2). So far, some intermediate languages have been proposed to implement model transformations (Section 5), but none of them proposes continuations as an implementation mechanism.

Thus, this paper presents an approach to compile ATL to a continuation-based representation. A simple, general mechanism to enable continuations in model transformations is developed, giving its metamodel and execution algorithm (Section 3). Then, the ATL execution semantics is mapped to this mechanism, explaining how ATL basic constructs can be represented with it, and outlining the current limitations of our approach (Section 4).

2 Background and motivation

This section introduces the concept of *continuation*, outlining some of its features. Then, the usefulness of using continuations as an implementation mechanism of a model transformation language is motivated.

A continuation reifies the concept of "the rest of the computation", so that the execution state of a given program can be saved into a continuation and restored later. This concept is supported in some programming languages, for instance Scheme [6] or Scala [7]. Related to continuations, the so called *continuation-passing style* (CPS) is a way of programming in which functions do not return values, but they are passed another function which represents the rest of the computation with respect to the call (i.e., a continuation). The result of the evaluation is then passed to the continuation.

As an example, the following piece of code uses CPS to show how the idea of continuations could be used in a model transformation setting. It specifies the classical UML to Java transformation, where a UML class is transformed to a Java class, and a UML attribute is transformed to a "get method". The code is written in Javascript, since it might be more familiar to the reader than other functional languages.

```
function transformClass(class) {
  jclass = new JavaClass();
  jclass.name       = class.name;

  // Returns a closure that takes an attribute
  // and adds it the newly created Java class
  return function(jFields) {
    jclass.fields = jFields;

    return function(jMethods) {
       jclass.methods = jMethods;
       return jclass;
    }

  };
}

// To simplify the explanation, these two functions
// are not written in CPS style
function transformAttributeToMethod(attr) {
  jMethod = new JavaMethod();
  jMethod.name = 'get' + attr.name;
  return jMethod;
}

function transformAttributeToField(attr) {
  jattr = new JavaAttribute();
  jattr.name = attr.name;
  return jattr;
}


uml.objects['Class'].each(function(c) {
  var continuation = transformClass(c);

  var attributes   = c.attributes.map(function(attr) {
    transformAttributeToAttribute(attr);
  });

  continuation = continuation(attributes)
```

```
  var methods      = c.attributes.map(function(attr) {
    transformAttributeToMethod(attr);
  });

  continuation(methods)
});
```

The function `transformClass` takes a UML class as a parameter, and creates a new Java class. Then, instead of trying to find the Java class members that correspond to the UML class attributes (in this case, a get method and an attribute per each UML attribute), it returns a function (that is, a closure) that stops the execution until the data is available. This closure is a *continuation*, because it contains the rest of the code that the `transformClass` function will execute when the corresponding Java fields are computed (i.e., what it has not done yet). Notice, that the continuation returns a new continuation to "wait" for the Java methods.

The `transformAttributeToMethod` and `transformAttributeToField` are in charge of creating a Java method and field for a given attribute. They are not written in CPS style to simplify the explanation.

Finally, all UML class objects are traversed. Each class is transformed into a Java class, calling `transformClass` which returns a continuation, that will be resumed when the corresponding Java attributes and methods are available. To this end, the mapping from UML attributes to Java attributes is first computed, resuming the continuation afterwards. A new continuation is got, which is resumed after mapping attributes to "get methods".

Regarding the applicability of continuations to model transformations, an important part of the implementation of a transformation language is devoted to how to resolve relationships between elements. Depending on the features of the transformation language, an algorithm has to be devised to schedule the transformation execution to resolve all data dependencies properly. Some algorithms need to perform multiple passes in order to resolve data dependencies between rules and they use some intermediate structure (typically hash maps) to index values until some pass of the transformation algorithm looks up the value. Therefore, our premise is that continuations fit well in a context where data may not be readily available, but will be produced at some time by other entities (e.g., a transformation rule). The rationale is that, as explained, a continuation captures "the rest of the computation", so a part of a model transformation could be stopped and stored into a continuation until the model element(s) that it needs is available, and then resume the continuation.

We believe that, besides implementing "standard" transformation algorithms with continuations, the use of continuations could enable the implementation of more advanced language features. In this way, we are working on applying continuations to tackle advanced model transformation features'.

- *Language interoperability* could be promoted by using continuations as a common scheduling mechanism for several languages.

– *Distributed model transformations.* The ability of continuations to be persisted could enable scheduling pieces of transformation execution between nodes.
– *Model transformations over models coming in streaming* is related to the previous point, since a piece of transformation could "sleep" in a continuation until related data is available.

In this paper, we bring continuations into ATL, which may provide opportunities to implement novel features for ATL. To this end, the standard ATL transformation algorithm is mapped to a continuation-based one. Next section describes such algorithm.

3 An execution mechanism based on continuations

This section presents a simple execution mechanism to perform model-to-model transformations based on continuations. It will be adapted to ATL in the next section. Figure 1 shows a meta-model that comprises the main concepts of the mechanism.

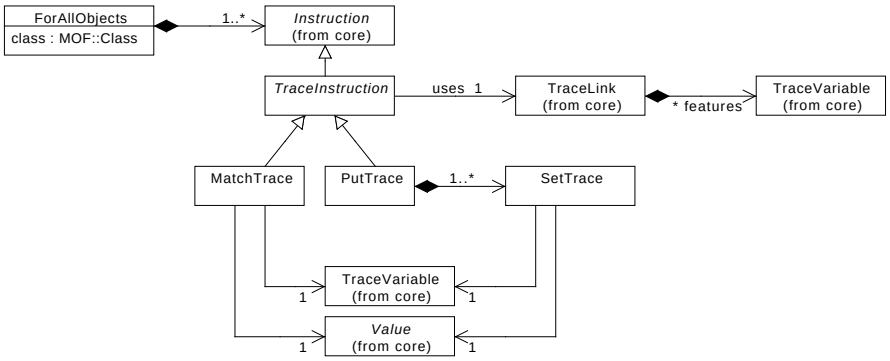

Fig. 1: Meta-model representing the constructs of the mechanism.

A transformation definition uses trace links to keep the relationships between source elements and target elements created by the rules. Trace links can be implicit or explicit. In the case of ATL they are implicit, and they are needed to resolve bindings. In the metamodel, a kind of trace link is defined by instantiating a `TraceLink` with its corresponding `TraceVariable`s.

Ideally, the execution mechanism is not tied to any execution engine. We use the abstract metaclass `Instruction` to represent an instruction or statement of some engine. Two special instructions are added, which deal with the trace model in a "continuation-based way": *PutTrace* and *MatchTrace*.

The PutTrace instruction creates a new trace link, setting its value via the `SetTrace` elements that set the value of each trace variable. The instruction is executed atomically, in the sense that the new link is issued to the trace model only when all of trace variables have been set.

The MatchTrace instruction is intended to query the trace model. The query is quite simple, since it just allows looking up a trace link that has a variable with some value (the `MatchTrace` instruction is related to some `TraceVariable` whose value must be equal to `Value`). An important difference of this instruction with respect to other approaches is that it blocks until a trace link that satisfies the query is available. In this way, if no trace links are found, the transformation engine suspends the execution of the instruction and executes other, non-dependant instructions (for instance, instructions in other rules or even other instructions of the same rule). The MatchTrace instruction is resumed later to check whether some PutTrace instruction has produced the required trace link.

The `Rule` construct represents a piece of code that iterates over all instances of a given metaclass, executing the instructions enclosed within it. It can be seen as a simplification of an ATL matched rule, but with some arbitrary code being run when it is matched. For the moment we do not focus on every possible instruction that could be executed, but we just focus on the high-level execution process.

Given a transformation that uses the constructs explained above to represent the resolution of dependencies and the iteration over model elements, the execution semantics is given by combining fix-point iteration and continuations. The following pseudo-code shows the execution algorithm for the mechanism.

```
def topLevel {
  pending = []
  for(rule in ''Transformation rules'') {
    objects = allObjectsOf(rule.sourceMetaclass)
    for(o in objects) {
       continuation = evalRule(rule, o)
       if ( continuation )
          pending.add(continuation)
    }
  }

  resolve(pending)
}

def resolve(pending) {
  newPending = []
  for(continuation in pending)
     newContinuation = tryResolveTrace(continuation)
     if ( newContinuation )
        newPending.add(newContinuation)
  }

  if ( newPending.nonEmpty ) {
     if ( pending.size == newPending.size ) {
        // Maybe I could generate an identifier for each continuation
        if ( pending == newPending ) {
           stuck()
           exit
        }
     }
     resolve(newPending)
   }
```

```
end

def stuck()
  ... depends on the language ...
end

topLevel()
```

As can be seen, the `topLevel` function iterates over all rules of the transformation, getting all instances of its source metaclass. This rule is then executed, returning a continuation object in case it stopped or `null` if it finished correctly. After all rules have been executed, some of them are stopped and queued in the `pending` list. Afterwards, all the stored continuations must be iterated and tried again in order to complete the execution of the rules, storing the new continuations produced.

If two complete iterations do not change the list of pending continuations, the transformation is stuck and it finishes. Notice that stuckness does not necessarily mean that the transformation definition is wrong, but just that some elements cannot be resolved. The `stuck()` function represents a language-dependent behaviour when stuckness arises. As will be seen, this is used in the ATL mapping to emulate part of the implicit tracing mechanism.

3.1 Example

Figure 2 shows a piece of a possible transformation execution for the transformation definition introduced in Section 2 but using the mechanism presented in this section. Please not that in this case CPS is not being used (the execution is not resumed from the Continuation object), but a continuation take the form of a "controlled goto".

The transformation would execute the `class2javaClass` rule first, creating a new `JavaClass` object and storing it in the trace. Afterwards, the rules queries the trace for the `JavaMethod` corresponding to the attribute `attr` from the UML source `class`. Since it is not yet in the trace, the query returns `null`, a continuation is created, and the control flow is returned to the transformation, so a new rule can be executed.

Then, the `attr2JavaMethod` rule is executed, creating a new `JavaMethod` and storing it. After it has finished, all rules have been executed, so the transformation has to iterate over the `pending` continuation list and execute them. The continuation previously created for the `class2javaClass` is selected, making the execution flow jump to the place where the rule had stopped. Now, the rule queries again the trace in order to find the `JavaMethod` it requires. Since it is now available, the rule can finish its execution normally.

Now, all the pending continuations have been executed, resulting in an empty `newPending` list, meaning all mappings have been executed correctly and the transformation has finished.

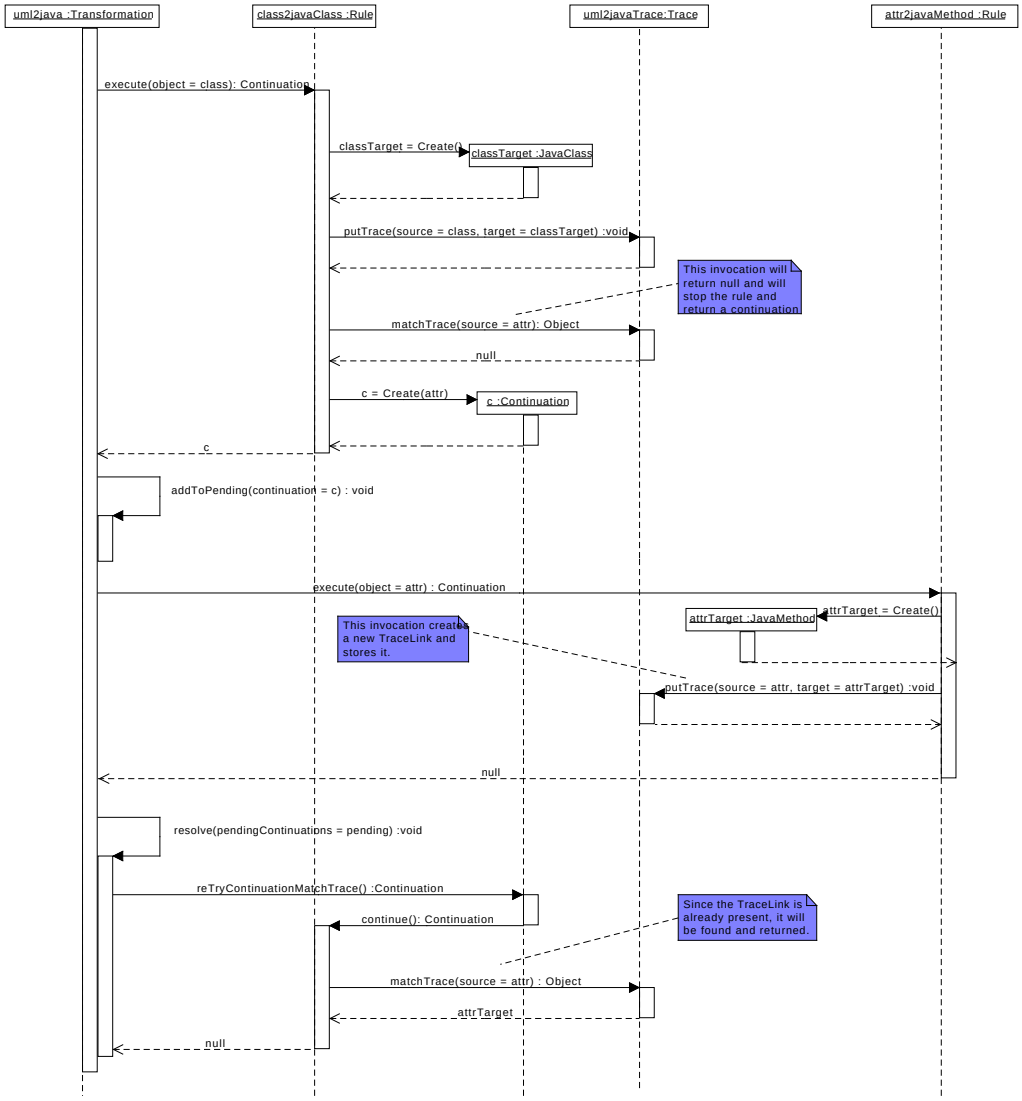

Fig. 2: Example of executing the Class2Java transformation with continuations

4 Mapping

This section describes how to map the ATL execution semantics to the mechanism proposed above. In particular, matched rules with one input pattern have been considered. Other features are later discussed, outlining some current limitations of our approach.

The ATL algorithm is described in [4] in detail, but it can be summarized as follows. It performs two passes. The first one goes through all matched rules, matching source elements against the rule's pattern and creating the corresponding target elements. For each match, the relationship between source and target elements is kept in a *trace link*. The second pass goes through all trace links, setting properties of the target elements according to rule bindings. The right part of a binding is resolved by looking up the trace links. If no trace link is found, the value of the right part is assigned whenever is compatible with the left part.

A lazy rule can be seen as a function, with the side effect of creating a target element, invoked in the evaluation of the right part of a binding. Unique lazy rules can also be seen as functions, but memoizing functions whose cache is the trace model (which is never empty).

To begin with the mapping, the shape of the trace model must be considered. In ATL rules "communicate" implicitly via bindings, so a generic type of trace link is defined to record any relationship between the source element and the target elements declared in a matched rules. It has as many elements (of type MOF::Object) as declared in the matched rule with the most source and target elements. The rationale is that a rule may have only one source element but another one may have two or more, but since the trace is generic it has to be able to trace any possible relationship in the transformation. Each execution of a matched rule will create a trace link of this kind, setting only the appropriate attributes, which will be used later to resolve bindings.

The input pattern of a matched rule is mapped to our `Rule` metaclass in order to iterate over the source model. For each target pattern, instructions to create the corresponding target elements will be added to the rule, and one `PutTrace` instruction that creates and issues to the trace model a new `TraceLink` relating the source element and the target elements. The rest of required instructions (for instance, the ones of the ATL-VM) are added.

A binding is mapped to a `MatchTrace` instruction over the generic trace model. The value obtained after evaluating the right part of the binding is the value to be looked up in the trace (i.e., the source element of the MatchTrace instruction), and the result (which will only be one target element) is assigned to the property specified in the left part of the binding. Regarding calls to *resolveTemp(obj, 'outputVar')*, they are mapped to a MatchTrace instruction as well, but getting the target element corresponding to *outputVar*.

A lazy rule is naturally mapped to a plain function, which is called each time the lazy rule is called. No PutTrace instructions are needed in this case, since lazy rules always return a different output element.

4.1 Limitations

The mapping has some limitations to get the full ATL semantics. Next, we outline those limitations and show some possible solutions.

- The ATL *implicit tracing* strategy cannot be mapped to our mechanism in an straightforward way. The reason is that in ATL when the right part of

a binding cannot be resolved, the source element is retrieved (if compatible with the left part) or just ignored. A possibility to provide this behaviour with our mechanism is to implement the `stuck()` function, to perform the required asignments before finishing the transformation.
- *Multivalued bindings.* When the right part of a binding is a collection of elements, our approach retrieves "all or nothing". Here, the limitation is more difficult to overcome, because the presence of non-transformed elements prevent the asignment of the already transformed ones. We are looking for ways to deal with these issues. A possible workaround would be to "attach" some instructions to `MatchTrace` that are executed each time an individual element of the collection is resolved.
- *Unique lazy rules* require querying the trace model in a non-blocking way, but we have not defined any mechanism for that matter yet.

5 Related work

The mechanism presented in Section 3 can be seen as part of an intermediate languages for model transformations. In this sense, some of them have been defined so far. The QVT standard proposes QVT Core as a low-level language to facilitate the implementation of QVT Relations [5]. It can be considered as an intermediate representation, but in contrast to our approach, it is just specific to QVT. Besides, we do not know of any working implementation. ATC [2] is also a low-level language for model transformations. It provides a great amount of atomic instructions, but none of them provide any means to automatically resolve data dependencies, but a specially tailored solution has to be implemented for each case.

ATL is run on top of a virtual machine, which implements a stack-based language [3]. Transformations compiled to this language have to implement its own scheduling algorithms, as is the case of ATL which is implemented with a two pass algorithm.

IDC provides transformation-specific features (e.g., instructions to match the trace model, communication between patterns and rules), and enables interoperability by sharing trace information. Additionally, IDC promotes low-level code manipulation since it is based on SSA and represented with a metamodel, so that it is easier to analyze than a stack-based language. It is part of our recent work in model transformation language implementation, and can be downloaded at http://modelum.es/projects/eclectic.

Regarding continuations, CPS style has been widely used in compiler construction as an intermediate representation for general purpose language, in order to enable local optimizations [1]. Describing a transformation definition in terms of CPS could enable analysis and optimizations typically used in functional languages to be applicable to model transformation languages.

Finally, our approach can also be seen as an alternative execution semantics for ATL. In [8] a semantics for ATL is given based on rewriting logic using Maude. A similar approach could be used to give the semantics of ATL in terms of continuations or CPS.

6 Conclusion

In this paper, we have presented a proposal to implement model transformations with continuations, as an alternative approach to resolve data dependencies. Regarding its applicability in the ATL context, we have shown how some ATL constructs could be implemented using continuations. We believe that this could enable advanced features in ATL, such as the ones commented in section 2.

As future work, we are looking into how to overcome some of the limitations of our approach. Also, we are working on using continuations to deal with transformation language interoperability, streamed models and concurrent transformation execution.

Acknowledgments

This work has been supported by Ministerio de Ciencia e Innovación (Spain), grant TIN2009-11555.

References

1. A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1 edition, February 2007.
2. A. Estévez, J. Padrón, E. V. S. Rebull, and J. L. Roda. Atc: A low-level model transformation language. In *MDEIS*, pages 64–74, 2006.
3. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008.
4. F. Jouault and I. Kurtev. Transforming models with atl. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138, Berlin, 2006. Springer Verlag.
5. OMG. Final adopted specification for MOF 2.0 Query/View/Transformation, 2005. www.omg.org/docs/ptc/05-11-01.pdf.
6. J. Rees and W. Clinger. Revised report on the algorithmic language scheme. *SIGPLAN Not.*, 21:37–79, December 1986.
7. T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In *Proceedings of the 14th International Conference on Functional Programming*, pages 317–328, 2009.
8. J. Troya and A. Vallecillo. Towards a rewriting logic semantics for atl. In *Proceedings of the Third international conference on Theory and practice of model transformations*, ICMT'10, pages 230–244, Berlin, Heidelberg, 2010. Springer-Verlag.