The ATL/EMFTVM Solution to the Train Benchmark Case for TTC2015

Dennis Wagelaar
HealthConnect
Vilvoorde, Belgium
dennis.wagelaar@healthconnect.be

This paper describes the ATL/EMFTVM solution of the TTC 2015 Train Benchmark Case. A complete solution for all tasks is provided, three of which are discussed with regard to the three provided evaluation criteria: Correctness and Completeness of Model Queries and Transformations, Applicability for Model Validation, and Performance on Large Models.

1 Introduction

This paper describes a solution of the TTC 2015 Train Benchmark Case [4] made with ATL [2] and the EMF Transformation Virtual Machine (EMFTVM) runtime engine [5]. The Train Benchmark Case consists of several model validation and model repair tasks: three main tasks and two extension tasks. All of these tasks are run again increasing model sizes in order to measure the performance of each solution for the case. A complete solution for all tasks is provided, and is available as a GitHub fork of the original assignment[1]. Section 2 of this paper describes the ATL transformation tool and its features that are relevant to the case. Section 3 describes the solution to the case, and section 4 concludes this paper with an evaluation.

2 ATL/EMFTVM

ATL is a rule-based, hybrid model transformation language that allows declarative as well as imperative transformation styles. For this TTC solution, we use the new EMF Transformation Virtual Machine (EMFTVM). EMFTVM includes a number of language enhancements, as well as performance enhancements. For this TTC case, specific performance enhancements are relevant.

2.1 JIT compiler

EMFTVM includes a Just-In-Time (JIT) compiler that translates its bytecode to Java bytecode. EMFTVM bytecode instructions are organised in *code blocks*, which are executable lists of instructions. When a code block is executed more often than a predefined threshold, the JIT compiler triggers, and will generate a Java bytecode equivalent for the EMFTVM code block.

2.2 Lazy evaluation

EMFTVM includes an implementation of the OCL 2.2 standard library [3], and employs lazy evaluation for the collection operations (e.g. `select`, `collect`, `flatten`, `isEmpty`, etc.). That operations invoked

[1] `https://github.com/dwagelaar/trainbenchmark-ttc`

on collections are only (partially) executed when you evaluate the collection. For example, the `lazytest` query in Listing 1 invokes `collect` on a Sequence of all numbers from 0 to 100, which replaces each value in the Sequence by its squared value, but eventually only returns the last value of the Sequence. `collect` returns a lazy Sequence, which is just waiting to be evaluated. Only when `last` is invoked, the `square` operation is invoked on the last element of the input Sequence. As a result, `square` is only invoked once.

```
1 query lazytest = Sequence{0..100}->collect(x | x.square())->last();
2 helper context Integer def : square() : Integer =
3   (self * self).debug('square');
```

Listing 1: Lazy collections in ATL

2.3 Caching of model elements

Model transformations usually look up model elements by their type or meta-class. In the Eclipse Modeling Framework (EMF) [1], this means iterating over the entire model and filtering on element type. Often, an element look up by type is made repeatedly on the same model. In the case of this benchmark, the same query/transformation is run multiple times on the same model. For this reason, EMFTVM keeps a cache of model elements by type for each model. This cache is automatically kept up to date when adding/removing model elements through EMFTVM. The cache is built up lazily, which means that a full iteration over the model must have taken place before the cache is activated for that element type. This prevents a build up of caches that are never used.

3 Solution Description

The Train Benchmark Case involves first querying a model for constraint violations, and then repairing some of those constraint violations that are randomly selected by the benchmark framework. This means that the matching phase and the transformation phase, which are normally integrated in ATL, are now separated by the benchmark framework. The framework first launches the matching phase, and collects the found matches. After that, it randomly selects a number of matches, and feeds them into the transformation phase.

ATL provides a **query** construct that allows one to query the model using OCL and return the resulting values. The selected matches are fed back into the ATL VM through a helper attribute, specified in the framework repair transformation module shown in Listing 2. The benchmark framework copies the returned lazy collection into a regular `java.util.ArrayList`, which ensures that the performance measurements are valid.

The Repair transformation module contains a helper attribute `matches`, which is used to inject the matches selected by the benchmark framework. Furthermore, it contains a lazy rule `Repair`, which does nothing in this framework transformation. The `Repair` rule is invoked by every element in `matches` by the `Main` endpoint rule. The `Main` endpoint rule is automatically invoked. Normally, ATL transformations use matched rules that are automatically triggered for all matching elements in the input model(s). However, this benchmark requires the elements to transform to be set explicitly. Hence the need for this framework transformation module. All specific repair transformation modules are *superimposed* [6] onto the framework transformation module, and redefine the `Repair` rule. This means that for each task we only need to define an ATL query and a `Repair` rule. Because of space constraints, two out of five tasks will be discussed in this paper.

```
module Repair;
create OUT: RAILWAY refining IN: RAILWAY;
helper def : matches : Collection(OclAny) = Sequence{};
lazy rule Repair {
  from s: OclAny
}
endpoint rule Main() {
  do {
    for (s in thisModule.matches) {
      thisModule.Repair(s);
    }
  }
}
```

Listing 2: Framework repair transformation module in ATL

3.1 Task 1: PosLength

Listing 3 shows the ATL query for Poslength. It simply collects all Segment instances with a length of zero or smaller. Listing 4 shows the ATL repair transformation module for Poslength. It imports the framework Repair transformation module from Listing 2, and redefines the `Repair` rule. As no new elements need to be created, an imperative **do** block is used to make the required modification directly on the source element. The `<:=` assignment operator is used instead of the `<-` binding operator, such that the implicit source-to-target tracing is skipped.

```
query PosLength = RAILWAY!Segment.allInstances()->select(s | s.length <= 0);
```

Listing 3: PosLength query in ATL

```
module PosLengthRepair;
create OUT: RAILWAY refining IN: RAILWAY;
uses Repair;
lazy rule Repair {
  from s: RAILWAY!Segment
  do   { s.length <:= -s.length + 1; }
}
```

Listing 4: PosLength repair transformation module in ATL

3.2 Task 2: SwitchSensor

Listing 5 shows the ATL query for SwitchSensor. It collects all Switch instances for which the sensor is not set. Listing 6 shows the ATL repair transformation module for SwitchSensor. This time, the `Repair` rule also contains a **to** section that creates a new Sensor instance `se`. In the **do** section, this Sensor is assigned to the sensor reference of the input Switch element.

3.3 Extension Task 1: RouteSensor

Listing 7 shows the ATL query for RouteSensor. The query collects Tuples of each match, where a match is defined by Route `r`, SwitchPosition `p`, Switch `sw`, and Sensor `s`. A Tuple is created for each SwitchPosition connected to a Sensor that is not connected to the Route, for each Route that has Sensors connected to it. Listing 8 shows the ATL repair transformation module for RouteSensor. The `Repair` rule takes the Tuple match as input element, and adds the Sensor in the match to the Route's definedBy sensors.

```
query SwitchSensor = RAILWAY!Switch.allInstances()->select(s | s.sensor.oclIsUndefined());
```

Listing 5: SwitchSensor query in ATL

```
module SwitchSensorRepair;
create OUT: RAILWAY refining IN: RAILWAY;
uses Repair;
lazy rule Repair {
  from s: RAILWAY!Switch
  to   se: RAILWAY!Sensor
  do   { s.sensor <:= se; }
}
```

Listing 6: SwitchSensor repair transformation module in ATL

4 Evaluation and Conclusion

The solutions for the Train Benchmark Case are evaluated on three criteria: (1) *Correctness and Completeness of Model Queries and Transformations*, (2) *Applicability for Model Validation*, and (3) *Performance on Large Models*. We will now discuss how the ATL solution aims to meet these criteria.

4.1 Correctness and Completeness

The benchmark framework provides a set of expected query/transformation results, against which the output of the ATL solution can be compared. The `ATLTest` JUnit test case verifies that the output of the ATL solution matches the reference solution. The test results of each build are kept in the cloud-based Travis continuous integration platform[2]. This independent platform provides an objective proof that the ATL solution unit tests are passing.

4.2 Applicability

In order for a solution to be applicable for model validation, it must be concise and maintainable. Even though ATL is not primarily intended for interactive querying and transformation, it was easy to fit the ATL implementation into the benchmark framework. Simple queries are trivially expressed in OCL, using a functional programming style (PosLength, SwitchSensor). Complex queries that return tuples as matches (SwitchSet, RouteSensor, SemaphoreNeighbor) require a navigation strategy to be implemented. All repair phase transformations are all simple, single rule transformation modules that are *superimposed* onto a single framework Repair transformation module (see Listing 2). Query matches are

[2] https://travis-ci.org/dwagelaar/trainbenchmark-ttc

```
query RouteSensor = RAILWAY!Route.allInstances()
  ->select(r | r.definedBy->notEmpty())
  ->collect(r |
    r.follows->select(p |
      not p.switch.oclIsUndefined() and
      not p.switch.sensor.oclIsUndefined() and
      r.definedBy->excludes(p.switch.sensor)
    )->collect(p |
      Tuple{r = r, p = p, sw = p.switch, s = p.switch.sensor}
    )
  )->flatten();
```

Listing 7: RouteSensor query in ATL

```
module RouteSensorRepair;
create OUT: RAILWAY refining IN: RAILWAY;
uses Repair;
lazy rule Repair {
  from s : TupleType(r : RAILWAY!Route, p : RAILWAY!SwitchPosition, sw : RAILWAY!Switch,
                     s : RAILWAY!Sensor)
  do   { s.r.definedBy <:= s.r.definedBy->including(s.s); }
}
```

Listing 8: RouteSensor repair transformation module in ATL

provided via the rule **from** part, whereas the model element modification is done in a **do** block. Any new elements are specified in the **to** block.

4.3 Performance

In the ATL language, performance is achieved by using helper attributes instead of operations where possible, as helper attribute values are cached; accessing a helper attribute more than once on the same object will not trigger evaluation again, but just returns the cached value. EMFTVM also applies certain performance optimisations: complex code blocks are JIT-compiled to Java bytecode, which in turn may be JIT-compiled to native code by the JVM. Collections and boolean expressions are evaluated lazily, preventing unnecessary navigation. Finally, model elements are cached by their type, making repeated lookup of all instances of a certain metaclass more performant.

References

[1] Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick & Timothy J. Grose (2003): *Eclipse Modeling Framework*. The Eclipse Series, Addison Wesley Professional. Available at `http://safari.awprofessional.com/0131425420`.

[2] Frédéric Jouault, Freddy Allilaire, Jean Bézivin & Ivan Kurtev (2008): *ATL: A model transformation tool*. *Science of Computer Programming* 72(1-2), pp. 31–39, doi:10.1016/j.scico.2007.08.002.

[3] Object Management Group, Inc. (2010): *OCL 2.2 Specification*. Available at `http://www.omg.org/spec/OCL/2.2/PDF`. Version 2.2, formal/2010-02-01.

[4] Gábor Szárnyas, Oszkár Semeráth, István Ráth & Dániel Varró (2015): *The TTC 2015 Train Benchmark Case for Incremental Model Validation*. In: *Proceedings of TTC 2015*. Available at `https://github.com/FTSRG/trainbenchmark-ttc/raw/master/paper/trainbenchmark-ttc.pdf`.

[5] Dennis Wagelaar, Massimo Tisi, Jordi Cabot & Frédéric Jouault (2011): *Towards a General Composition Semantics for Rule-Based Model Transformation*. In Jon Whittle, Tony Clark & Thomas Kühne, editors: *Proceedings of MoDELS 2011*, *Lecture Notes in Computer Science* 6981, Springer-Verlag, pp. 623–637, doi:10.1007/978-3-642-24485-8_46. Available at `ftp://progftp.vub.ac.be/tech_report/2011/vub-soft-tr-11-07.pdf`.

[6] Dennis Wagelaar, Ragnhild Van Der Straeten & Dirk Deridder (2009): *Module superimposition: a composition technique for rule-based model transformation languages*. *Software and Systems Modeling* 9(3), pp. 285–309, doi:10.1007/s10270-009-0134-3.