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. The Train Benchmark Case consists of several model validation and model repair tasks, all of which 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 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 the ATL/EMFTVM [2] solution of the TTC 2015 Train Benchmark Case [4]. The Train Benchmark Case consists of several model validation and model repair tasks, all of which 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].

The remainder of this paper is structured as follows: section 2 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

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) [5]. EMFTVM includes a number of language enhancements, as well as performance enhancements. For this TTC case, specific performance enhancements are relevant. Each of these enhancements is described briefly in the following subsections.

2.1 JIT compiler

EMFTVM includes a Just-In-Time (JIT) compiler that translates EMFTVM bytecode to Java bytecode. EMFTVM bytecode instructions are organised in *code blocks* (see Fig. 1). Code blocks are executable lists of instructions, and have a number of local variables and a local stack space. Code blocks are used to represent operation bodies, field initialisers, rule guards, and rule bodies. Code blocks may also have nested code blocks, which effectively represent *closures*[2]. EMFTVM records how often each code block is executed, as well as some execution metadata, such as which methods were dispatched in each virtual

[1]https://github.com/dwagelaar/trainbenchmark-ttc
[2]http://en.wikipedia.org/wiki/Closure_(computer_programming)

Submitted to:
TTC 2015

method call. 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. The JIT compiler provides the best performance improvement for large and complex code blocks.

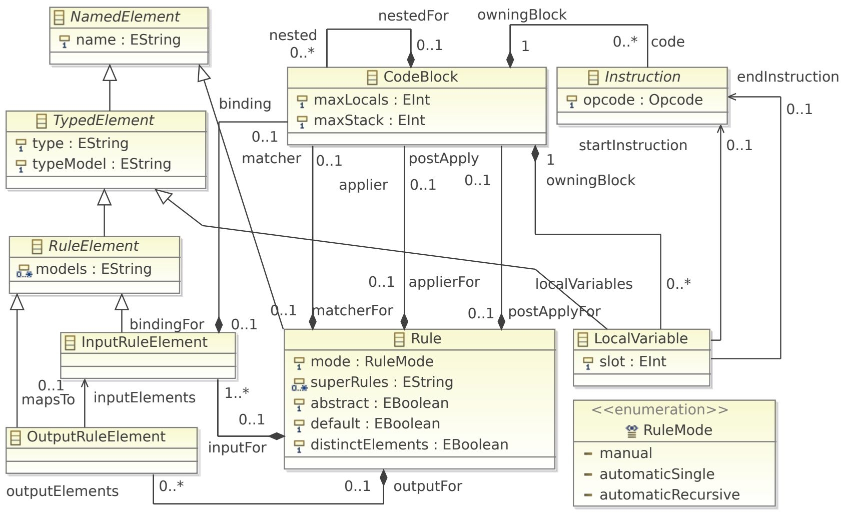

Figure 1: Structure of EMFTVM rules and code blocks

2.2 Lazy evaluation

EMFTVM includes an implementation of the OCL 2.2 standard library [3], and employs lazy evaluation for the collection operations[3] (e.g. `select`, `collect`, `flatten`, `isEmpty`, etc.). That means you can invoke operations on the collections, but those operations will not be executed until you actually evaluate the collection. Also, collection operations will only be evaluated partially, depending on how much of the collection you evaluate. To illustrate how this works, look at the example code in Listing 1. The `lazytest` query invokes ”collect” on a Sequence of all numbers from 0 to 100, and replaces each value in the Sequence by its squared value. Finally, we’re only interested in the last value of the changed Sequence. `collect` returns a lazy Sequence, which is just waiting to be evaluated. Only when ”last” is invoked on the lazy Sequence will the Sequence invoke the ”expensive” operation on the last element of the input Sequence. As a result, `square` is only invoked once.

```
query lazytest = Sequence{0..100}->collect(x | x.square())->last();

helper context Integer def : square() : Integer =
  (self * self).debug('square');
```

Listing 1: Lazy collections in ATL

In addition, short-circuit evaluation is applied to boolean expressions (i.e. **and**, **or**, and **not**). While this may not be a desirable semantics for OCL in general, it is advantageous for using OCL as a navigation language: only the relevant parts of the model are navigated. Lazy evaluation provides the best performance improvement when only consuming a small part of a string of collection operations (e.g.

[3]`https://wiki.eclipse.org/ATL/EMFTVM#Lazy_collections`

`list->reject(x | x.attr.oclIsUndefined())->collect(x | x.attr)->first()`). Short-circuit evaluation prevents having to use (nested) `if-then-else-endif` blocks everywhere.

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 (especially when doing recursive, in-place transformation[4]). 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, looking for a specific element type, 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. The resulting values are returned by the ATL VM. 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. Note that the ATL query returns a *lazy* collection, which is just waiting to be evaluated. The benchmark framework compensates for this by copying all values of the returned lazy collection into a regular `java.util.ArrayList`, which triggers evaluation. This 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. The Java code in the benchmark plug-in for ATL is made up of a base class `ATLBenchmarkCase` that provides the generic logic for:

1. instantiating a query VM, a transformation VM, and loading the metamodels (`init`);
2. loading the models (`read`);
3. performing the query phase of the benchmark (`check`);
4. performing the transformation phase of the benchmark (`modify`).

[4] `https://code.google.com/a/eclipselabs.org/p/simplegt/`

```
module Repair;
create OUT: RAILWAY refining IN: RAILWAY;

--- Helper attribute that holds the matches to transform.
--- Injected from outside the transformation.
helper def : matches : Collection(OclAny) = Sequence{};

--- Base implementation of the Repair rule that does nothing.
lazy rule Repair {
  from
    s: OclAny
}

--- Applies the Repair rule to all matches.
endpoint rule Main() {
  do {
    for (s in thisModule.matches) {
      thisModule.Repair(s);
    }
  }
}
```

Listing 2: Framework repair transformation module in ATL

Each specific task subclasses the `ATLBenchmarkCase` class, but only has to override the `init` method. The overridden `init` does all of the superclass `init`, but also loads the ATL transformation bytecode into the query VM and the transformation VM.

3.1 Task 1: PosLength

The PosLength task consists of a query that checks for Segments with a length less than or equal to zero, and a repair transformation that updates the length attribute of the segment in the match to `length+1`.

Listing 3 shows the ATL query for Poslength. It simply collects all Segment instances with a length of zero or smaller.

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

Listing 3: PosLength query in ATL

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, and no implicit tracing of source elements to target elements is required, 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.

3.2 Task 2: SwitchSensor

The SwitchSensor task consists of a query that checks for Switches that are not connected to a Sensor, and a repair transformation that creates and connects a new Sensor.

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

```
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

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

Listing 5: SwitchSensor query in ATL

to the sensor reference of the input Switch element.

```
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

3.3 Task 3: SwitchSet

The SwitchSet task consists of a query that checks for Routes that are not connected to a Sensor, which have a semaphore that show the GO signal. Additionally, the route follows a switch position (p) that is connected to a switch (sw), but the switch position (p.position) defines a different position from the current position of the switch (sw.currentPosition). Furthermore, a repair transformation is provided, which sets the currentPosition attribute of the switch to the position of the switchPosition.

Listing 7 shows the ATL query for SwitchSet. The query collects Tuples of each match, where a match is defined by Route `r`, Semaphore `s`, SwitchPosition `p`, and Switch `sw`. A Tuple is created for each wrong SwitchPosition that was found for each Route with a “GO” signal. As one can see, Tuples allow for returning matches with multiple elements to the benchmark framework.

Listing 8 shows the ATL repair transformation module for SwitchSet. The `Repair` rule takes the Tuple match as input element this time, and assigns the SwitchPosition’s position to the Switch’s currentPosition.

```
query SwitchSet = RAILWAY!Route.allInstances()
  ->select(r |
    not r.entry.oclIsUndefined() and r.entry.signal = #GO
  )->collect(r | r.follows
    ->select(p |
      not p.switch.oclIsUndefined() and p.switch.currentPosition <> p.position
    )->collect(p |
      Tuple{r = r, s = r.entry, p = p, sw = p.switch}
    )
  )->flatten();
```

Listing 7: SwitchSet query in ATL

```
module SwitchSetRepair;
create OUT: RAILWAY refining IN: RAILWAY;

uses Repair;

lazy rule Repair {
  from
    s : TupleType(
      r : RAILWAY!Route,
      s : RAILWAY!Semaphore,
      p : RAILWAY!SwitchPosition,
      sw : RAILWAY!Switch)
  do {
    s.sw.currentPosition <:= s.p.position;
  }
}
```

Listing 8: SwitchSet repair transformation module in ATL

3.4 Extension Task 1: RouteSensor

The RouteSensor task consists of a query that checks for Sensors that are connected to a Switch, but the Sensor and the Switch are not connected to the same Route. The repair transformation inserts the missing definedBy Sensors for the Route.

Listing 9 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.

```
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 9: RouteSensor query in ATL

Listing 10 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.

```
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 10: RouteSensor repair transformation module in ATL

3.5 Extension Task 2: SemaphoreNeighbor

The SemaphoreNeighbor task consists of a query that checks for Routes `r1` that have an exit Semaphore, and a Sensor `s1` connected to another Sensor `s2` – defining another Route `r3` – by two TrackElements `te1` and `te2`, for which there is no other Route `r2` that connects the same Semaphore and the other Sensor `s2`. Furthermore, a repair transformation is provided, which removes the exit Semaphore from Route `r1`.

Listing 11 shows the ATL query for SemaphoreNeighbor. The query uses a SimpleGT [5] transformation module to collect matches. SimpleGT is a minimal graph transformation language with embedded OCL support, and supports local search plans for pattern matching. Both SimpleGT and ATL compile to EMFTVM bytecode, and can hence be executed together as a single transformation. Experimentation has shown the EMFTVM local search support to be more performant than encoding the search plan directly in OCL. The pure ATL/OCL query alternative solution is given for reference purposes in Appendix A.

```
query SemaphoreNeighbourQuery = thisModule.traces.getLinksByRule('Check', true).links
    ->collect(l | Tuple{
      s = l.getSourceElement('s', true).object,
      r1 = l.getSourceElement('r1', true).object,
      r2 = l.getSourceElement('r2', true).object,
      s1 = l.getSourceElement('s1', true).object,
      s2 = l.getSourceElement('s2', true).object,
      te1 = l.getSourceElement('te1', true).object,
      te2 = l.getSourceElement('te2', true).object});

uses SemaphoreNeighbourCheck;
```

Listing 11: SemaphoreNeighborQuery in ATL

SimpleGT matches are recorded in the implicit trace model by EMFTVM. The ATL query reads the recorded traces using reflection, and creates OCL Tuples for each matching trace. Fig. 2 shows a class diagram of the EMFTVM Trace metamodel, including the built-in operations. The root object of the trace model is a TraceLinkSet, which can be accessed from ATL using the built-in `thisModule.traces` helper attribute.

Listing 12 shows the SimpleGT check transformation for SemaphoreNeighbor. It uses a “single” rule (i.e. one-shot, non-recursive matching) to match the required pattern. Whenever a navigation is possible from one element to the next, the SimpleGT compiler produces a local search expression to

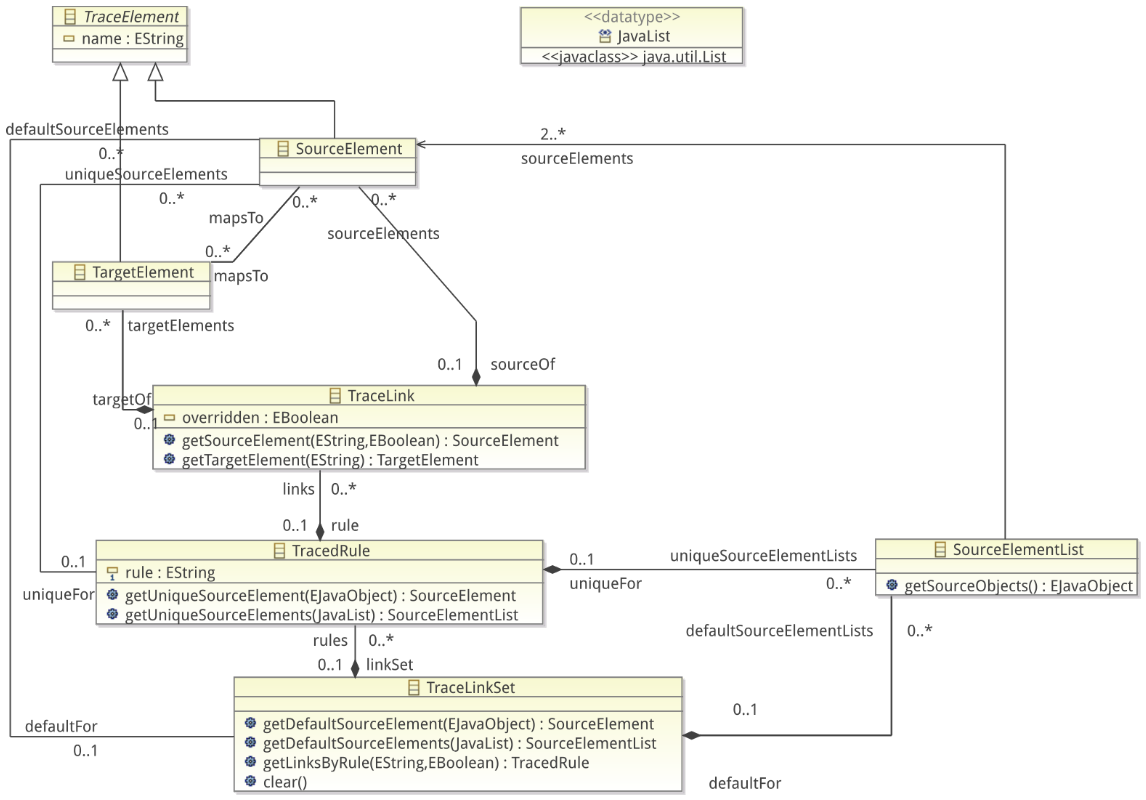
TraceElement
name : EString
<<datatype>>
JavaList
<<javaclass>> java.util.List
SourceElement
TargetElement
TraceLink
overridden : EBoolean
getSourceElement(EString,EBoolean) : SourceElement
getTargetElement(EString) : TargetElement
TracedRule
rule : EString
getUniqueSourceElement(EJavaObject) : SourceElement
getUniqueSourceElements(JavaList) : SourceElementList
SourceElementList
getSourceObjects() : EJavaObject
TraceLinkSet
getDefaultSourceElement(EJavaObject) : SourceElement
getDefaultSourceElements(JavaList) : SourceElementList
getLinksByRule(EString,EBoolean) : TracedRule
clear()
defaultSourceElements
uniqueSourceElements
sourceElements
mapsTo
targetElements
targetOf
sourceOf
links
rule
rules
linkSet
uniqueFor
uniqueSourceElementLists
defaultSourceElementLists
defaultFor

Figure 2: The EMFTVM Trace metamodel

find the next element. EMFTVM will evaluate this local search expression whenever available instead of iterating over the entire model. There is one situation where direct navigation is not possible: finding Route `r2` from Sensor `s2`. To get around this problem, the `route` helper attribute is defined on the Sensor metaclass. This helper attribute retrieves the container object for the Sensor, as `Route::definedBy` is a containment reference. The `route` helper attribute can now be used in the Check rule to navigate from Sensor `s2` to all possible Routes `r2`. The **not** part in the rule represents a *negative application condition* (NAC), which specifies a condition that may not occur in a valid match. A rule can have multiple **not** parts. SimpleGT requires you to specify a *left-hand-side* (LHS) and *right-hand-side* (RHS) for each rule, represented by the **from** and the **to** part, respectively. The LHS is always replaced by the RHS. Because we don't want the Check rule to change anything, we must repeat the LHS in the RHS to prevent it from being deleted.

Listing 13 shows the ATL repair transformation module for SemaphoreNeighbor. The `Repair` rule takes the Tuple match as input element, and adds the entry Semaphore `s` to Route `r2`.

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.

```
module SemaphoreNeighbourCheck;
metamodel RAILWAY : '/hu.bme.mit.trainbenchmark.ttc.emf.model/model/railway.ecore';
transform IN : RAILWAY;

-- Checks for matches
single rule Check {
  from
    r1 : RAILWAY!Route (definedBy =~ s1, exit =~ s),
    s : RAILWAY!Semaphore,
    s1 : RAILWAY!Sensor (elements =~ te1),
    te1 : RAILWAY!TrackElement (connectsTo =~ te2),
    te2 : RAILWAY!TrackElement (sensor =~ s2),
    s2 : RAILWAY!Sensor (route =~ r2),
    r2 : RAILWAY!Route
  not
    r2 : RAILWAY!Route (entry =~ r1.exit)
  to
    r1 : RAILWAY!Route (definedBy =~ s1, exit =~ s),
    s : RAILWAY!Semaphore,
    s1 : RAILWAY!Sensor (elements =~ te1),
    te1 : RAILWAY!TrackElement (connectsTo =~ te2),
    te2 : RAILWAY!TrackElement (sensor =~ s2),
    s2 : RAILWAY!Sensor (route =~ r2),
    r2 : RAILWAY!Route
}

-- Returns the Routes mapped by their Sensor.
context RAILWAY!Sensor def : route : RAILWAY!Route =
  let route : RAILWAY!Route = self.refImmediateComposite() in
  if route.oclIsKindOf(RAILWAY!Route) then
    route
  else
    OclUndefined
  endif;
```

Listing 12: SemaphoreNeighborCheck transformation module in SimpleGT

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[5]. This independent platform provides an objective proof that the ATL solution unit tests are passing. Furthermore, all git commits for the ATL solution are publicly available on GitHub, and it can be verified that no modifications are made to the benchmark framework and/or the expected result set.

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. While this is not as declarative as first-class patterns, it is more concise than imperative pro-

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

```
module SemaphoreNeighbourRepair;
create OUT: RAILWAY refining IN: RAILWAY;

uses Repair;

--- Applies the repair transformation.
lazy rule Repair {
  from
    s : TupleType(
      s : RAILWAY!Semaphore,
      r1 : RAILWAY!Route,
      r2 : RAILWAY!Route,
      s1 : RAILWAY!Sensor,
      s2 : RAILWAY!Sensor,
      te1 : RAILWAY!TrackElement,
      te2 : RAILWAY!TrackElement)
  do {
    s.r2.entry <:= s.s;
  }
}
```

Listing 13: SemaphoreNeighbor repair transformation module in ATL

gramming. Also, ATL provides helper attributes and operations to divide the complexity into modular blocks. For very complex patterns, EMFTVM provides a built in local search engine. Currently, this engine is only used by the SimpleGT graph transformation language, which also compiles to EMFTVM bytecode. Because of this, a hybrid ATL/SimpleGT solution can be used to perform the check phase of the SemaphoreNeighbor task, which is more readable overall.

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 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.

Most example and production ATL transformation modules are much longer than the ones used in the benchmark case solution[6], with industrial cases going up to 7000 lines of ATL for a single transformation scenario[7]. Even such long transformation modules have proven to be sufficiently maintainable, especially when compared to implementations in Java.

4.3 Performance

The benchmark framework supports running the solution against increasing model sizes, starting at 2^0 and going up by 2^{n+1}. Within the memory constraints of the SHARE image[8] of 1 GB RAM, we managed to go up to model size 512 with `-Xmx512m` as JVM arguments. While all benchmark tasks can be completed on SHARE for this model size within the time constraint of 5 minutes, this does not work all of the time. The performance of a SHARE VM is not consistent, and we sometimes saw simple tasks time out (e.g. RouteSensor, SwitchSensor), while the most complex task (SemaphoreNeighbor) finishes. On local hardware (AMD 1055T[9]), with 8 GB RAM and using `-Xmx4G` as JVM arguments (`java-1.7.0-openjdk-1.7.0.79-2.5.5.0.fc20.x86_64`), we managed to go up to model size 4096 for all tasks except SemaphoreNeighbor. The SemaphoreNeighbor task timed out above size 2048.

[6]https://www.eclipse.org/atl/atlTransformations/
[7]http://www.slideshare.net/DennisWagelaar/wagelaar-sda2014
[8]http://is.ieis.tue.nl/staff/pvgorp/share/?trgPage=LookupImage&vdiNameSearch=TTC15_ATL
[9]http://www.cpubenchmark.net/cpu.php?cpu=AMD+Phenom+II+X6+1055T

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.

Furthermore, the virtual machine (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 and allowing short-circuit evaluation of expressions. Finally, model elements are cached by their type, making repeated lookup of all instances of a certain metaclass more performant.

Fig. 3 shows the batch read and check scenario performance for the ATL/EMFTVM solution, and Fig. 4 shows the repetitive recheck and repair scenario performance. Whereas most tasks have similar performance in ATL/EMFTVM, the SemaphoreNeighbor task is an outlier. SemaphoreNeighbor proved much heavier to solve for ATL/EMFTVM than the others. SemaphoreNeighbor requires local search in order to be performant. The effect of caching model elements by type is minimal in this case, as this cache will only provide the first element of the entire pattern. The lack of the element type cache advantage is especially apparent in the recheck and revalidate scenario. SwitchSensor, on the other hand, has significantly better performance for recheck and repair than the other tasks. This is due to – apart from element type caching for all Switch instances – an efficient implementation in EMFTVM of the `oclIsUndefined()` operation, which maps directly to the `ISNULL` instruction in EMFTVM. Combined with JIT compilation, this yields very efficient Java bytecode.

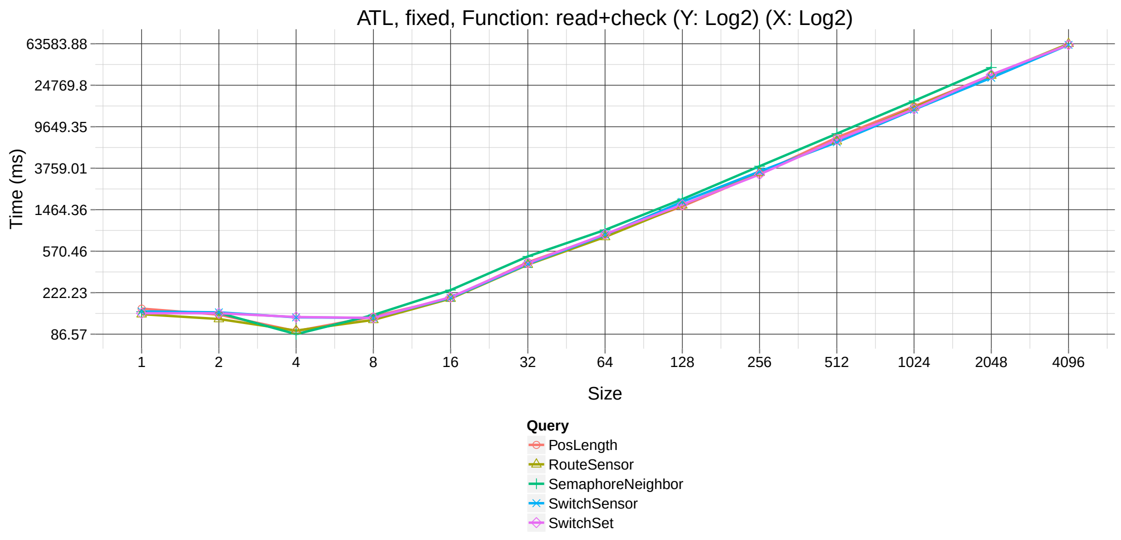

Figure 3: ATL batch validation performance for the fixed scenario

A Alternative Solution for SemaphoreNeighbor

Listing 14 shows an alternative ATL query for SemaphoreNeighbor, using only the ATL language. The query collects the match Tuples for each Route `r1` with an exit Semaphore, for which there is a Route `r2` that is different from Route `r1`, and of which the entry differs from the `r1` exit, connected to Sensor `s2`, where `s2` is connected to Sensor `s1` or Route `r1` by TrackElements `te1` and `te2`. The resulting nested

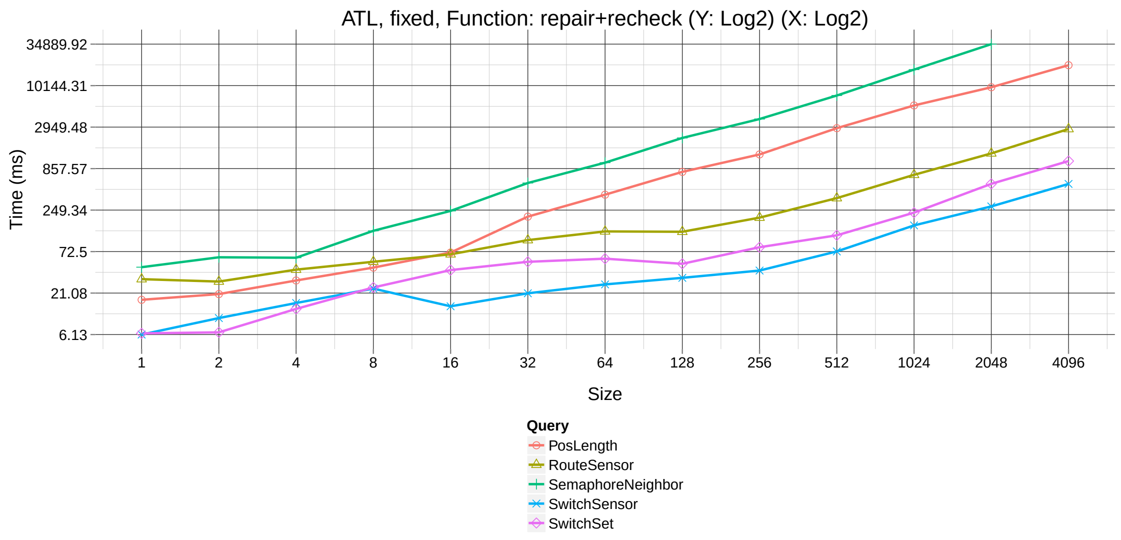

Figure 4: ATL revalidation performance for the fixed scenario

Collections are flattened into a single Collection of Tuples.

```
query SemaphoreNeighbour = RAILWAY!Route.allInstances()
  ->reject(r1 | r1.exit.oclIsUndefined())
  ->collect(r1 |
    r1.definedBy->collect(s1 |
      s1.elements->collect(te1 |
        te1.connectsTo->reject(te2 |
          let s2 : RAILWAY!Sensor = te2.sensor in
          s2.oclIsUndefined() or not (
          let r2 : RAILWAY!Route = s2.refImmediateComposite() in
          r2.oclIsKindOf(RAILWAY!Route) and r2 <> r1 and r2.entry <> r1.exit)
        )->collect(te2 |
          let s2 : RAILWAY!Sensor = te2.sensor in
          Tuple{s = r1.exit, r1 = r1, r2 = s2.refImmediateComposite(),
            s1 = s1, s2 = s2, te1 = te1, te2 = te2}
        )
      )
    )
  )->flatten();
```

Listing 14: SemaphoreNeighbor query in ATL

Fig. 5 shows the batch read and check scenario performance for the ATL/EMFTVM solution, and Fig. 6 shows the repetitive recheck and repair scenario performance. When comparing these figures to Fig. 3 and Fig. 4, one sees that the performance of this alternative query is consistently worse than the SimpleGT solution. Apparently, the overhead of the OCL collection operations – many `collects` followed by `flatten` – makes a big difference. Apart from that, the local search algorithm implemented here in OCL closely resembles what EMFTVM's built in local search engine does. To see the exact difference between the ATL and SimpleGT solution, once has to compare the EMFTVM bytecode of both solutions ("SemaphoreNeighbour.emftvm" for ATL, and "SemaphoreNeighbourCheck.emftvm" for SimpleGT). The Eclipse tooling for ATL/EMFTVM[10] includes an EMF-based editor for the bytecode.

[10]`http://marketplace.eclipse.org/content/atlemftvm`

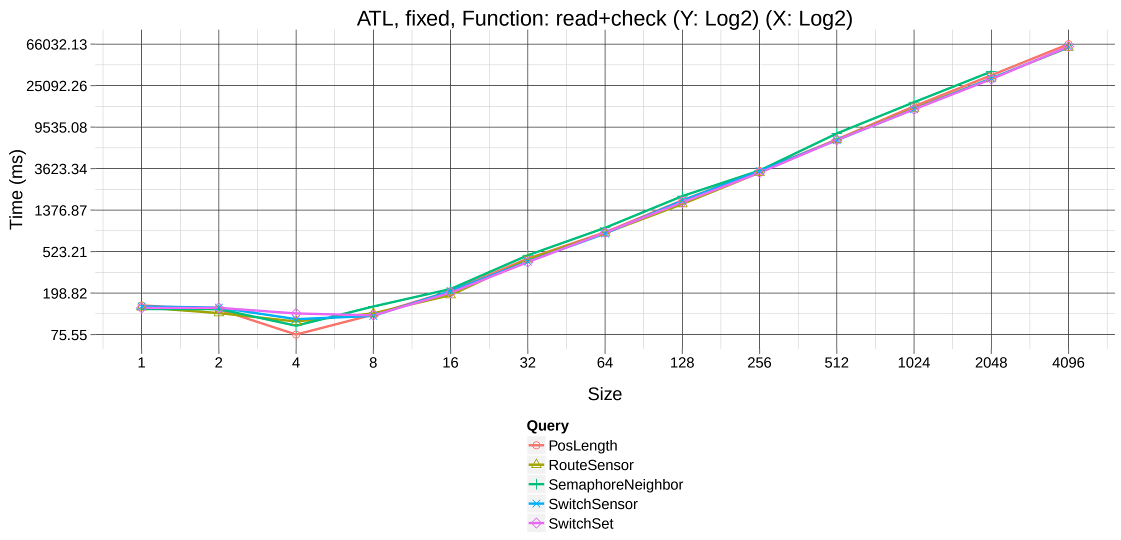

Figure 5: ATL alternative solution batch validation performance for the fixed scenario

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.

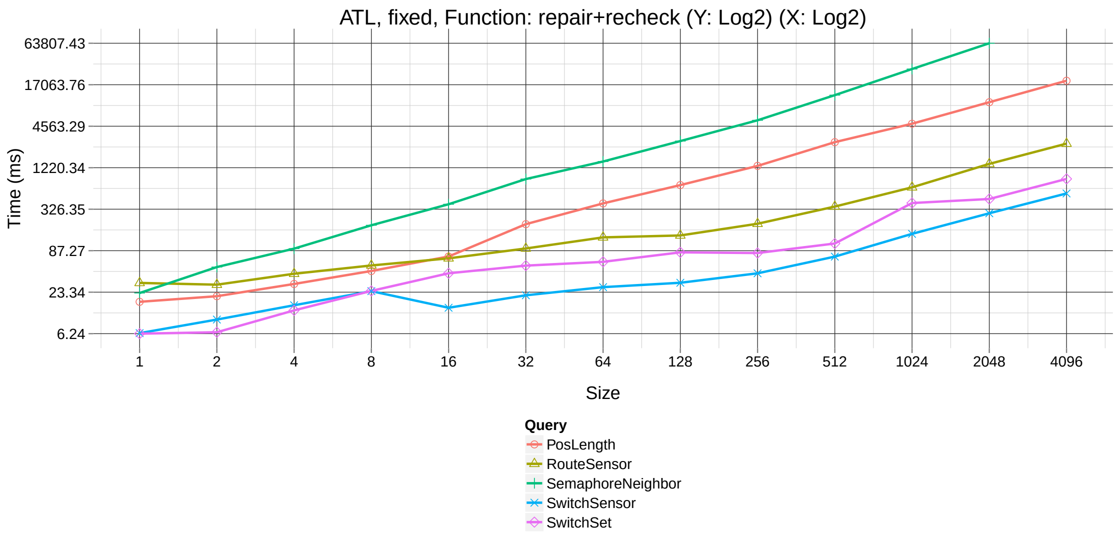

Figure 6: ATL alternative solution revalidation performance for the fixed scenario