The Epsilon Solution to the KMEHR to FHIR Case

Antonio Garcia-Dominguez[1]

[1]University of York, York, UK

Abstract

In recent years, there has been a push towards standardisation of interchange formats for health records: this case tackles the problem of mapping Patient Summarized Records between the Belgian KMEHR and international HL7 FHIR standards. The case proposed two evaluation dimensions: time/space efficiency, and understandability. This paper presents a port of the original ATL rules to the Epsilon Transformation Language (ETL), with a focus towards helping the user understand the transformation via transient visualisations using the Picto tool bundled with recent versions of Epsilon. While ETL is not as fast as ATL, the ETL script is 1,096 lines long, compared to the 1,319 lines of ATL. At the same time, during the development of this solution, a few improvements were identified and made to Epsilon itself.

Keywords

ATL, ETL, healthcare information systems, model transformation, model visualisation

1. Introduction

As described in the original case, allowing different healthcare systems to efficiently share information with each other is increasingly important to deliver rapid and effective care (e.g. when needing medical treatment in a different country). This has motivated the creation of a number of specifications to exchange the information that the system has about a patient. The Belgian KMEHR (Kindly Marked-Up Electronic Health Care Record)[1] is composed of an XML Schema describing a message grammar, a recognized set of medical transactions that use that grammar, and a set of reference tables with values to be used in those messages. Health Level 7 (HL7) is a set of international standards for exchanging health records, which includes the FHIR (Fast Healthcare Interoperability Resources) standards framework.

The original case description focused on transforming one specific type of healthcare record, known by HL7 as an International Patient Summary (IPS)[2]. An IPS is supposed to follow a patient across borders and make it possible to quickly provide the information needed for unplanned, cross-border care. The case description included a reference implementation with over 1,300 lines of ATL code, and specified two evaluation dimensions for alternative solutions: 1) their efficiency in time and space (in system memory), and 2) the understandability of the transformations.

This paper presents a solution based on the Eclipse Epsilon family of model management languages, available from Github[3]. The transformation itself is a close port of the original ATL rules to the Epsilon Transformation Language (ETL) [1], but it is complemented by the use of the Epsilon Picto [2] tool to produce interactive visualisations of its transformation trace. While ETL does not outperform the reference solution, it does take up fewer lines of code (1,096 lines, compared to the 1,319 of ATL), and its interactive visualisations can help understand the transformation and correct issues. The solution passes all the correctness checks provided by the case authors, and automated tests check that the produced FHIR models are line-by-line equivalent to the expected ones (except for changes due to unique IDs which change between executions).

The rest of the paper is structured as follows. Section 2 explains the ETL model-to-model transformation. Section 3 describes how the internal ETL transformation traces are serialised into a transformation trace model and then visualised with Picto. Section 4 presents the performance results of the solution and compares them against those of the reference ATL solution. Finally, Section 5 provides general conclusions for this paper, and points out several areas in which this work can be refined.

2. Model transformation

The original case description did not provide a detailed specification of the transformation involved, and the author was unfamiliar with KMEHR and FHIR. For that reason, the approach taken was to translate the ATL rules to ETL, given their similar approaches.

Due to time limitations, the ETL script is contained in a single `.etl` file that is 1,096 lines long, with a **pre**

TTC'23: 15th Transformation Tool Contest, Part of the Software Technologies: Applications and Foundations (STAF) federated conferences, Eds. A. Boronat, A. García-Domínguez, and G. Hinkel, 20 July 2023, Leicester, UK.

a.garcia-dominguez@aston.ac.uk (A. Garcia-Dominguez)

0000-0002-4744-9150 (A. Garcia-Dominguez)

[1]https://www.ehealth.fgov.be/standards/kmehr/en

[2]https://build.fhir.org/ig/HL7/fhir-ips/

[3]https://github.com/agarciadom/ttc2023-kmehr2fhir

Listing 1: Excerpt of the Posology rule in ETL

```
rule Posology
  transform s: KMEHR!PosologyType
  to t: FHIR!MedicationStatement, msid: FHIR!Id,
      /* ... */
{
  var i = s.eContainer();
  var tx = i.eContainer();
  var f = tx.eContainer();
  // ...
}
```

Listing 2: Excerpt of the Posology rule in ATL

```
rule Posology {
  from
    f : KMEHR!FolderType,
    tx : KMEHR!TransactionType,
    i : KMEHR!ItemType,
    s : KMEHR!PosologyType (
      i.posology = s and
      tx.item->includes(i) and
      f.transaction->includes(tx) and
      i.isMedication
    )
  to
    t : FHIR!MedicationStatement mapsTo s (
      // ...
    ),
    // ...
}
```

(pre-execution) block being used to initialise the various predefined mappings from special values in KMEHR to FHIR, and instantiate the XMLTypeFactory needed to deal with certain data types (i.e. dates).

Rules were mostly mapped 1-to-1 from the ATL sources, but some adaptations were neded due to subtle differences between ATL and ETL, which will be listed below.

2.1. Assignment and equivalent mapping

The ATL `<-` operator is similar to the ETL `::=` operator, in that it will assign to the l-value the results of transforming the r-value. However, `::=` cannot be used with objects that are not to be transformed (e.g. strings), and therefore users must be careful to use `=` or `::=` depending on the situation.

ETL does not have an equivalent to the ATL `mapsTo` keyword: in an ETL rule, every object in the `to` clause is considered equivalent to the source object. This implies that any calls to equivalent() or uses of the `::=` operator must be careful to filter the r-value so only objects compatible with the l-value are left. An alternative approach would have been to limit the objects in the ETL `to` list to those mentioned by the `mapsTo` keyword in ATL, and create the other objects from inside the ETL rule body. This approach was discarded as it would have made the transformation traces less thorough.

2.2. Number of source objects per rule

ETL only allows one source object per rule, unlike ATL which allows for multiple source objects. While ATL required using a local search compiler to avoid multiple scans of the model to obtain pattern matches, ETL rules were simply written to apply to the innermost object, and use x.eContainer() to access its containers as needed. Comparing the Posology rules in ETL (Listing 1) and ATL (Listing 2), the ETL approach is simpler to specify and compute.

An alternative approach could have been to use the Epsilon Pattern Language (EPL) to perform graph pattern matching as in the original ATL transformation, and pass the results as an additional input model to the ETL transformation. As mentioned above, this particular transformation was simple enough that this was not required.

2.3. Helper operations and lazy rules

ATL helper operations were translated to Epsilon Object Language operations, in generally a very straightforward way (with some simplifications, as the Epsilon languages do not have the `.` and `->` distinction that ATL has for accessing members, which is awkward to use for most new users). In addition, ETL allows for manually specifying whether the operation results should be cached for future calls (to save time or produce consistent results, like uuid() which generates random Universally Unique IDentifiers or UUIDs for a given FHIR object), or not (to save memory): this level of control is not apparent in ATL.

Some ATL lazy rules were translated to ETL lazy rules (e.g. the rule hierarchy dedicated to CompositionSections), whereas others were turned into EOL operations as it wasn't deemed useful to pollute transformation traces with them (e.g. the creation of FhirStrings). Note that some basic FHIR types did use lazy rules, precisely for populating the transformation trace, e.g. FhirDate and FhirDateTime. It is important to note that ETL suffers a major performance hit as soon as any lazy rule is used, as mentioned in its official documentation[4]: due to

[4] https://eclipse.dev/epsilon/doc/etl/

Listing 3: Rule inheritance in ATL with multiple non-abstract rules

```
rule SumEHRTransaction {
  from
    f : KMEHR!FolderType,
    s : KMEHR!TransactionType (
      f.transaction->includes(s) and
      s.cd->exists(cd | cd.value = 'sumehr')
    )
  to /* ... */}

rule SumEHRTransactionWithAuthor
  extends SumEHRTransaction {
  from
    f : KMEHR!FolderType,
    s : KMEHR!TransactionType (
      not s.txAuthor.oclIsUndefined()
    )
  to /* ... */}

rule SumEHRTransactionWithCustodian
  extends SumEHRTransaction {
  from
    f : KMEHR!FolderType,
    s : KMEHR!TransactionType,
    i : KMEHR!ItemType (
    s.item->includes(i) and
      i.cd->exists(cd | cd.value = 'gmdmanager')
    )
  to /* ... */}
```

Listing 4: Adapting multiple non-abstract rule inheritance of ATL to ETL

```
rule SumEHRTransaction
  transform s: KMEHR!TransactionType
  to t: FHIR!Composition,
      cid: FHIR!Id,
      patRef: FHIR!Reference,
      cStatus: FHIR!CompositionStatus,
      dateTime: FHIR!fhir::DateTime
{
  guard: s.cd.exists(cd| cd.value = 'sumehr')
  /* .... */
  if (s.txAuthor().isDefined()) { /* ... */}
  if (s.custodianItem().isDefined()) { /* ... */}
}
```

Listing 5: Mapping ATL enumeration literals

```
helper def : genderMap : Map(
  KMEHR!CDSEXvalues,
  FHIR!AdministrativeGenderEnum) =
  Map {
    (#changed, #other),
    (#female, #female),
    (#male, #male),
    (#unknown, #unknown),
    (#undefined, #other)
  };
```

time limitations, it was not possible to re-engineer the transformation to completely avoid them (e.g. by adding the appropriate guards).

2.4. Rule inheritance

It was noted that rule inheritance was somewhat different in ETL and ATL, when both the base and extending rule are non-abstract. Initial inspection of results suggests that ATL will execute the base and extending rule (if both applicable) on the same objects, whereas ETL will end up producing a different set of objects.

For instance, as shown in Listing 3 the ATL script had a base non-abstract SumEHRTransaction rule, extended by SumEHRTransactionWithAuthor and SumEHRTransactionWithCustodian. In the ETL script (as shown in Listing 4), these three rules were combined into one, where the body contained additional **if** blocks implementing the behaviour of the subclasses.

2.5. Enumeration literals

ATL has a convenient syntax for specifying enumeration literals: where the type is known from the context, it is possible to simply use #changed to refer to that literal (as shown in Listing 5. The languages in Epsilon 2.4.0 lacked this capability, requiring mentioning at least the name of the enumeration that contains the literal (e.g. CDSEXvalues#changed) as shown in Listing 6. This was awkward when writing complex expressions with one-off use of enumeration literals in ETL, as often the modeller has to stop writing and navigate the metamodels to find out the relevant enumeration to mention. It also introduced unnecessary repetition when specifying mappings between enumeration literals in the source and target metamodels.

The author extended the Epsilon grammars to allow for a more flexible syntax of the form `[Model!][Type]#literal`, where `[]` indicate optional elements. These literals are still interpreted in a context-free manner: if the reference is ambiguous, a warning will be raised. This is the reason why it was

Listing 6: Mapping ETL enumeration literals in Epsilon 2.4.0 (limited to fully-specified literals)

```
1 var genderMap = Map {
2   KMEHR!CDSEXvalues#changed = FHIR!
        AdministrativeGenderEnum#other,
3   KMEHR!CDSEXvalues#female = FHIR!
        AdministrativeGenderEnum#female,
4   KMEHR!CDSEXvalues#male = FHIR!
        AdministrativeGenderEnum#male,
5   KMEHR!CDSEXvalues#unknown = FHIR!
        AdministrativeGenderEnum#unknown,
6   KMEHR!CDSEXvalues#undefined = FHIR!
        AdministrativeGenderEnum#other
7 };
```

Listing 7: Mapping ETL enumeration literals in Epsilon 2.5.0 (supporting partial literals)

```
1 var genderMap = Map {
2   KMEHR!#changed = FHIR!
        AdministrativeGenderEnum#other,
3   KMEHR!#female = FHIR!#female,
4   KMEHR!#male = FHIR!#male,
5   KMEHR!CDSEXvalues#unknown = FHIR!
        AdministrativeGenderEnum#unknown,
6   KMEHR!CDSEXvalues#undefined = FHIR!
        AdministrativeGenderEnum#other
7 };
```

only necessary to use the model names on line 3, since both the KMEHR and FHIR metamodels had #male. The other lines in this block could not be modified as the literal names are in multiple types of both the KMEHR and FHIR metamodels.

2.6. Java wrapper and test oracle

In order to integrate with the benchmark framework and simplify its use, the transformation has been wrapped into a Java class called Transformation, similarly to the reference solution. The class provides a run() method for launching the transformation, as well as a method to specify if a trace model is desired (c.f. Section 3). The Driver class from the reference solution (which handles the environment variables from the Python `run.py` script) was largely adopted as-is, only changing the ATL-specific parts to use the new Transformation class.

In addition, the correctness of the transformation was checked with JUnit 5 tests that performed line-based differencing between the expected models and the obtained models using the `java-diff-utils` library[5], while discarding changes due to UUIDs. EMF Compare was initially considered for comparing the expected and obtained models, but was discarded as it produced too many spurious differences in its standard configuration.

3. Generation and visualisation of transformation traces

Having ported the transformation to ETL, the next goal was to help users understand the relationships between the original KMEHR model and the generated FHIR model. It was decided to use Epsilon Picto [2] to develop an interactive visualisation of the traces of the model transformation, which relate source and target objects, and rules.

3.1. Trace metamodel

Whereas ATL can automatically generate a trace model if a "trace" model is added to its launch configurations (based on a pre-existing trace metamodel), ETL only keeps an internal data structure (the TransformationTrace) and leaves its serialisation into a model up to the user (e.g. via a post-execution **post** block)[6]. It was necessary to design and implement a trace metamodel, as well as the code needed to populate trace models from the internal TransformationTraces.

The metamodel for ETL traces is shown in Figure 1, and has been designed specifically for this problem. The root of the model is a Trace from a source model to a target model (both represented by their Unique Resource Identifiers or URIs). These contain a forest of SourceObjects, a forest of TargetObjects, and a list of TransformationRules. SourceObjects and TargetObjects are both ModelObjects, for which we keep their package URI, their EClass name, and their URI fragment within their model. For TransformationRules, their name, location, and whether they are lazy or not is kept. Finally, we keep track of all the Transformations of every SourceObject, and the Transformation that produced a given TargetObject.

The use of forests to reproduce the containment trees of the source and target models is to allow the trace model to be entirely standalone for visualisation. This was mostly due to practical reasons: the KMEHR and FHIR model parsers had been provided as Maven artifacts rather than Eclipse plugins, and so it was not possible within the allotted time to browse KMEHR and FHIR

[5] https://github.com/java-diff-utils/java-diff-utils/

[6] Incidentally, the ETL TransformationTrace was internally revamped while developing this solution, replacing its original data structure (a flat list) with a faster, lookup-based data structure: https://github.com/eclipse/epsilon/pull/44

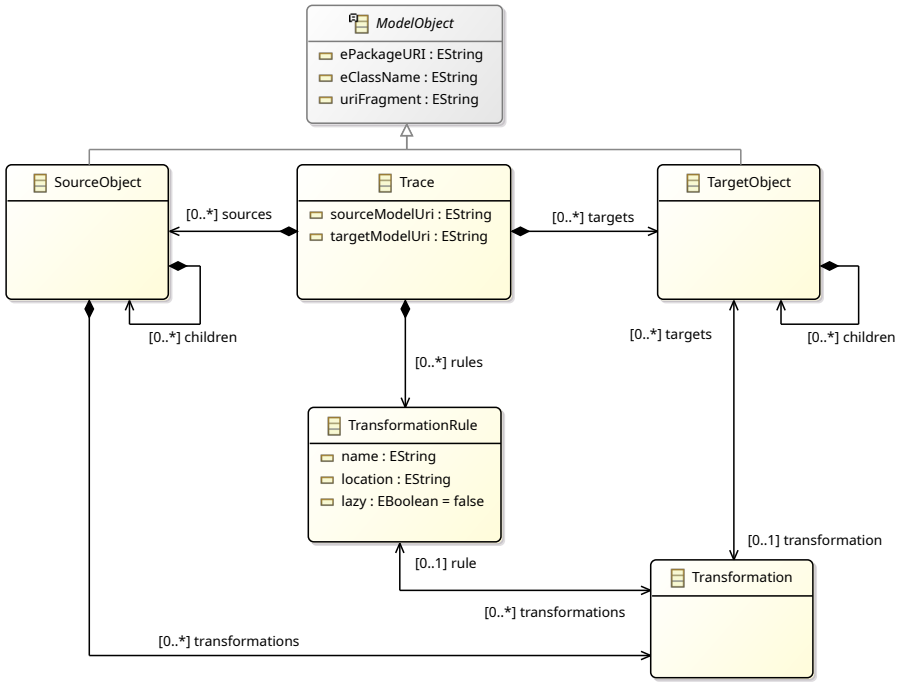


Figure 1: Trace metamodel for ETL

models using standard EMF tree-based model viewers. Given more time, it would have been possible to produce EMF tree-based model viewers for KMEHR and FHIR models, and only keep the relationships between source objects, target objects, and rules (similarly to TraceLinks in ATL's predefined trace metamodel).

3.2. Trace generation

Rather than using an **post** block in the ETL script to populate a trace model, it was decided to write this code as part of the Java Transformation class, to make it easier to integrate into future versions of Epsilon (after generalising its trace model to handling multiple source/target models).

The algorithm largely consists of these major steps:

1. Create the forest of SourceObjects, starting from the direct contents of the source model. If the source object has been involved in a transformation, ensure the appropriate TransformationRule exists and is populated from the ETL rule, create and populate its Transformation object, and temporarily associate its target objects to the Transformation object (e.g. via a throwaway IdentityHashMap).
2. Create the forest of TargetObjects, starting from the direct contents of the target model. If the TargetObject has been involved in a transformation (which we should know from the associations in the prior step), have it refer to the transformation.
3. To manage the size of the source and target object forests, these are now pruned: an object is only kept if itself or one of its descendants was involved in a transformation. This removes many of the objects that are unrelated to an ETL rule (e.g. FhirStrings), and helps reduce the amount of information shown to the user.
4. The TransformationRules in the trace are sorted by name, for easier navigation.

3.3. Trace visualisation

Once the trace models were available, Epsilon Picto was used to visualise them in an interactive manner. Picto is implemented as an Eclipse plugin which provides a user interface divided into two parts: a left part with a tree of selectable views, and a right part with an embedded web browser to show the visualisation generated from the selected view.

The view tree is generated by executing an EGX (EGL Co-Ordination Language) script with a number of rules, specified by the user by adding a file with the same name as the model plus a `.picto` suffix. An example for one

Listing 8: Sample `.picto` file

```
<?nsuri picto?>
<picto format="egx" transformation="platform:/resource/etl/src/main/egx/txtrace.egx">
</picto>
```

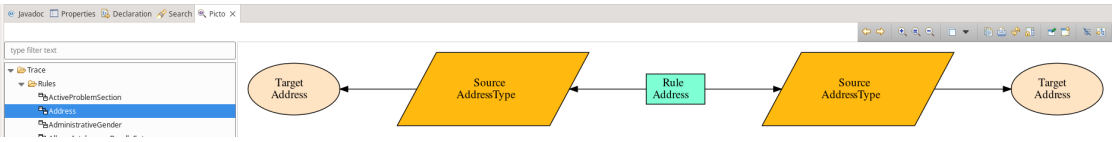


Figure 2: Screenshot of Picto visualising a TransformationRule

Listing 9: Sample EGX rule for visualising TargetObjects

```
rule SourceObject2Graphviz
  transform sob: SourceObject
{
  template : "source2graphviz.egl"
  parameters : Map {
    "path" = sob.path(),
    "icon" = sob.transformations.isEmpty()
      ? "document" : "tree_left",
    "format" = "graphviz-circo",
    "layers" = Sequence {
      Map { "id"="lazy", "title"="Show lazy rules",
      "active"=true }
    },
    "focus" = sob,
    "source" = sob,
    "trace" = trace::Trace.all.first
  }
}
```

of the traces in this solution is shown in Listing 8.

An example EGX rule for Picto is shown in Listing 9. It indicates that every SourceObject should be listed as an item in the view tree under a certain path, with an icon dependent on whether it participated in a transformation or not. If it were selected, Picto would use the EGL `source2graphviz.egl` to generate a file which would be rendered into a visualisation by using the `circo` algorithm in the Graphviz[7] tool. It also specifies an (optional) set of layers which can be toggled by the user: in this case, there is a layer for controlling whether lazy rules are shown or not (these layers need to be implemented in the invoked EGL scripts). Finally, there are some additional keys (`focus`, `source`, and `trace`) which are made available as local variables to the EGL script.

Similar rules have been defined for TargetObject and TransformationRule. Figure 2 shows what the Picto visualisation for a given rule looks like. The Rule is shown as an aquamarine rectangle, SourceObjects are orange parallelograms, and TargetObjects are light orange ("bisque") ellipses. Figure 3 and Figure 4 are visualisations of the source and target objects shown in Figure 2. It is important to note that these visualisations are interactive: users can click on a rule, source object or target object in the graphical visualisation, and it will automatically select and visualise the relevant view.

With this configuration of Picto, it is possible for users to see at a glance which objects participated in a transformation (from the icons), and then drill down into which objects they produced / were produced from, and which rules were involved. Picto is designed with the idea that a visualisation quickly becomes overwhelming when the entirety of the model is displayed at once, and instead focuses on the immediate neighbours of the selected element (as in Figures 2 to 4). As an example of its usefulness, while trying out this visualisation, the author noticed some "orphan" objects were being left out of the DocumentRoot of the target model (as shown in Figure 5), found the rules involved in their creation, and fixed the issues producing these orphan objects.

This interactivity is achieved by having the EGL scripts produce HTML links that trigger the `showView` JavaScript function contributed by Picto into the embedded web browser, which takes the tree path of the view to be selected. There was some brief experimentation with the `showElement` JavaScript function which allows for selecting a model element in an appropriate editor (e.g. to directly jump to the KMEHR/FHIR model element in question), but this did not work as expected due to the fact that the KMEHR/FHIR Maven artifacts were not contributing any tree-based model editors to Eclipse (as mentioned at the end of Section 3.1).

[7] https://graphviz.org/

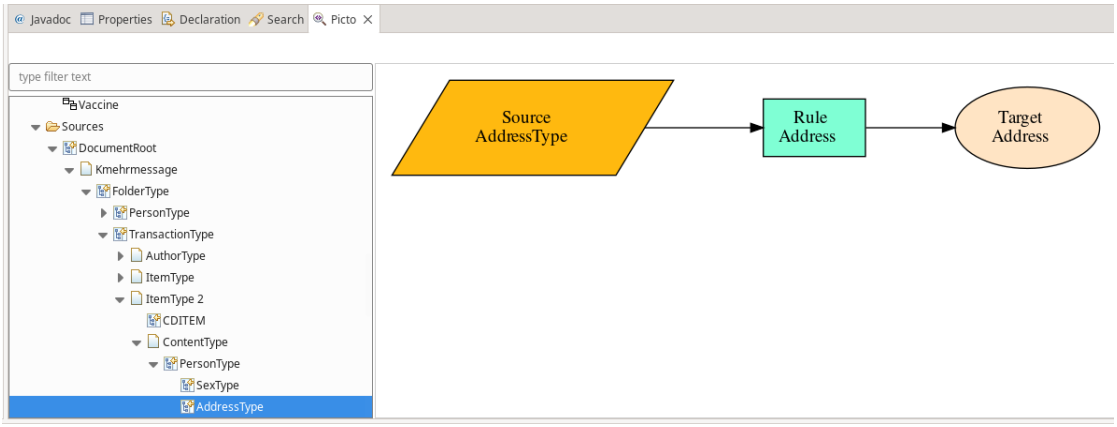


Figure 3: Screenshot of Picto visualising a SourceObject

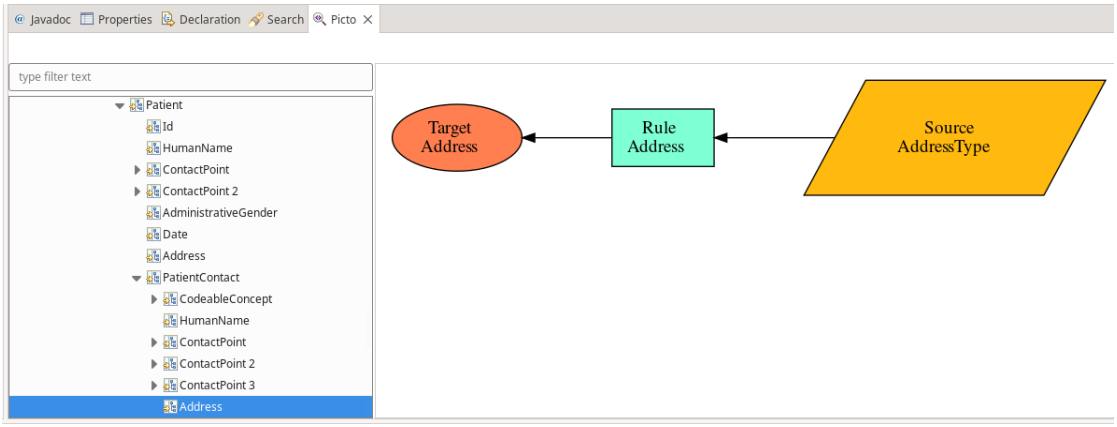


Figure 4: Screenshot of Picto visualising a TargetObject

4. Benchmark results

Having developed the transformations and visualised the results, the last part of the solution was to compare its time and space (memory) usage and compare it against the ATL reference solution. For performance comparisons, the Python script included with the reference solution was used, except for producing plots com-

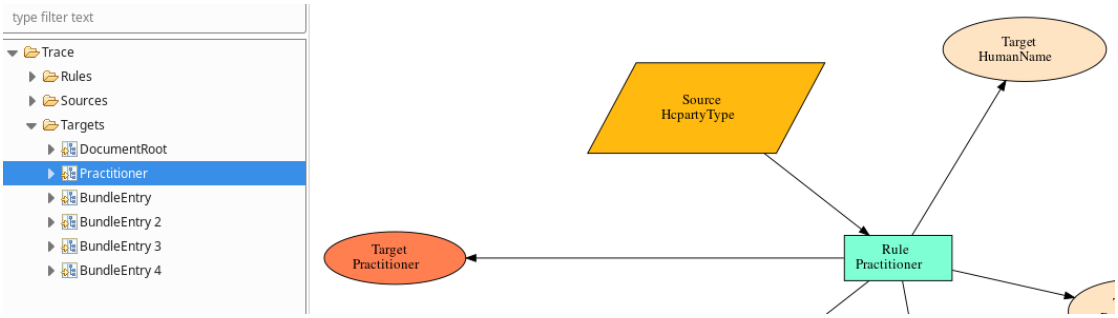


Figure 5: Screenshot of Picto visualising an orphan Practitioner

paring the results. Since the R-based reporting scripts did not seem to have been updated for the KMEHR to FHIR case study, a new set of R scripts (using Packrat for reproducible R package management) were created in the `src/main/viz` directory of the ETL solution. Each combination of tool and model size (1, 10, 100 and 1000) was run 3 times, and execution times and memory usages were averaged.

The results are shown in Figures 6 and 7. ETL did not outperform ATL in any scenario: while it took less time and memory in its Initialization and Load phases, it was approximately 10 times slower in the two smallest model sizes, 6 times slower with model size 100, and over 2 times slower with the largest model size. Looking at the graph, there may be some indications that the performance gap may be covered as models become larger, but this would require additional experimentation. As mentioned above, ETL had to switch to a slower execution strategy due to the use of lazy rules, as ported over from the ATL reference solution. On the other hand, both tools used similar amounts of memory, with ETL using slightly more (around 20% more for the largest model size) but still only reaching about 142.50MiB. Memory does not seem to be a bottleneck in this scenario.

After the contest, the author used the Epsilon profiler[8] (with a fix for ETL, currently being reviewed[9]) to measure the time used by the various rules. In a run of size 10, out of 2129ms measured by the profiler (which did not include model loading or saving times), 1332ms were dedicated to the transformation rules (with the difference being used to initialise the transformation). The main rule (DocumentRoot) took an aggregate 1212ms (including all its equivalent calls and triggered lazy rules), mostly taken up by the 1210ms of aggregate time for the Folder rule. Most of this time (889ms) was dedicated to computing the BundleEntry objects, especially the ConditionBundleEntry (346ms). There were a few other rules that took over 100ms (such as PatientContact with 107ms, or SumEHRTransaction with 137ms), but generally the cost was in the BundleEntry nodes. These results are largely within our expectations, given that most of the input model is dedicated to the medical history of the patient rather than their personal information or the high-level transactions that took place. As such, rather than pointing to a specific hotspot within the ETL script, they confirm that Epsilon has generally more overheads in the presence of lazy rules than ATL, and that ideally the transformation should be redesigned to avoid them.

[8] https://eclipse.dev/epsilon/doc/articles/profiling/
[9] https://github.com/eclipse/epsilon/pull/52

5. Conclusions and future work

In this paper, a port of the reference implementation to the Epsilon Transformation Language was presented and evaluated. ETL was extended with code that produced trace model similar in spirit to those produced by ATL, with the difference that the source and target model containment forests are represented in the trace model, after some pruning to remove subtrees that do not involve any transformations. The trace model was used with Picto to provide interactive visualisations of the transformation trace, which was helpful in detecting and correcting some of the last remaining defects in this transformation.

ETL did not outperform ATL in this scenario: it is believed that a major contributing factor may be the use of lazy rules, as ported from the original ATL script. It would be interesting to re-engineer the transformation to avoid lazy rules, and compare again the performance of ETL against ATL.

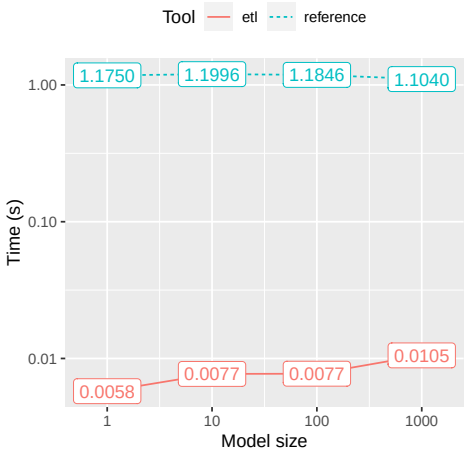
One line of work would be to take the generation of trace models in this solution and merge it into the Epsilon core, after some generalisation of the trace metamodel. An appropriate Picto visualisation could be bundled together with this feature, inspired by this solution.

Finally, there were some improvements done to Epsilon during the development of this solution: the internal ETL TransformationTrace data structure was optimised, and a pull request for adding ATL-style partial enumerations is currently under review in the Eclipse Epsilon project. There may be other opportunities to take advantage of more advanced data structures within Epsilon and improve performance (e.g. Guava caches, multimaps, and tables).

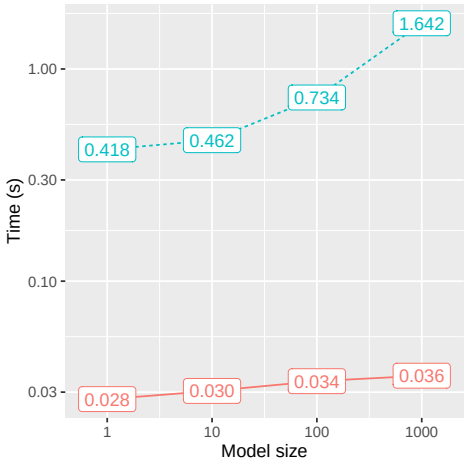
References

[1] D. S. Kolovos, R. Paige, F. Polack, The Epsilon Transformation Language, in: Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008, Proceedings, 2008, pp. 46–60. URL: http://dx.doi.org/10.1007/978-3-540-69927-9_4. doi:10.1007/978-3-540-69927-9_4.

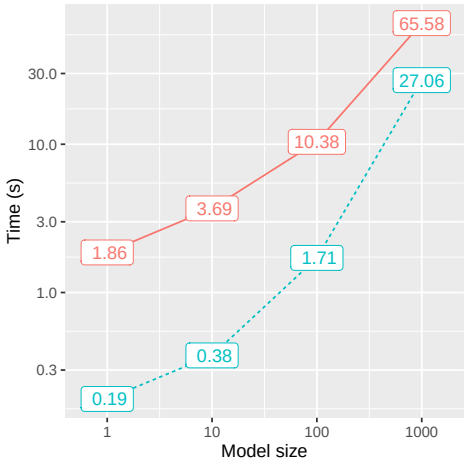
[2] D. Kolovos, A. de la Vega, J. Cooper, Efficient generation of graphical model views via lazy model-to-text transformation, in: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 12–23. URL: https://doi.org/10.1145/3365438.3410943. doi:10.1145/3365438.3410943.



(a) Initialization phase

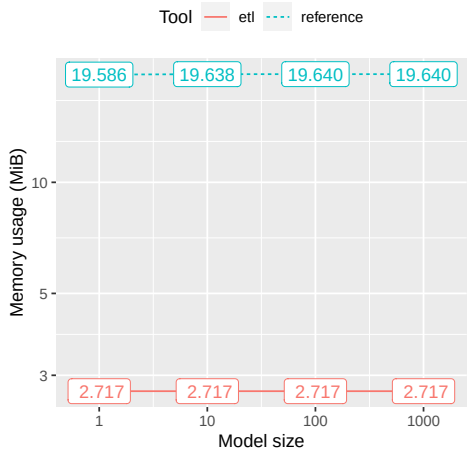


(b) Load phase

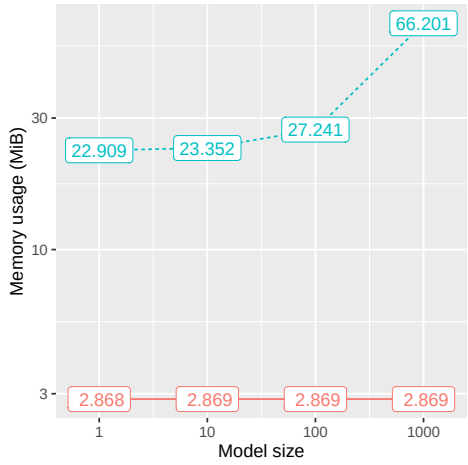


(c) Run phase

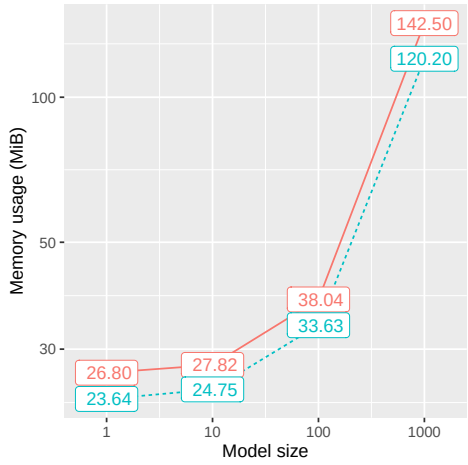
Figure 6: Average execution times in seconds, by phase



(a) Initialization phase



(b) Load phase



(c) Run phase

Figure 7: Average memory usage in MiB, by phase