Incremental ATL Solution to the TTC 2023 KMEHR to FHIR Case

Frédéric Jouault[1,2], Théo Le Calvar[3], and Matthew Coyle[3]

[1]University of Angers, LERIA, 49000 Angers, France
[2]ESEO-TECH / ERIS, 49100 Angers, France
[3]IMT Atlantique, LS2N (UMR CNRS 6004)

July 7, 2023

Abstract

This paper presents the ATOL solution to the TTC 2023 KMEHR to FHIR case study. The ATOL compiler is an alternative ATL compiler that enables incremental execution of ATL transformations. In this paper, we explain how we used ATOL to make the original KMEHR to FHIR ATL transformation incremental.

1 Introduction

With incremental model transformation engines, such as ATOL [5], NMF [2] or YAMTL [1], it is possible, after an initial application of the transformation, to detect changes on the source model and propagate these changes to affected parts of the target model without re-executing the transformation on the whole source model. This contrasts with traditional model transformation engines that recompute the whole target model from scratch after each change on the source model. Incremental model transformation engines are particularly useful when the source model is large, the transformation is complex or the source model is frequently modified.

This case study [6] involves translating between two medical data formats: from the Belgium KMEHR format, to the international FHIR format. The reference transformation is written in modern ATL with advanced features leveraging all features of the EMFTVM engine, such as multiple rule inheritance, `mapsTo` or the improved matching plan.

The transformation consists in a relatively large transformation and several helpers.

This paper is organized as follows. In section 2 we quickly present the ATOL compiler. In section 3 we present the process we developed to produce an ATL transformation compatible with ATOL. In section 4 we present the results of our approach. In section 5 we detail differences between the reference ATL transformation and our ATOL-compatible ATL transformation. Finally, in section 6 we conclude.

2 ATOL Overview

ATOL [5] is an ATL compiler that produces Java code, which in turn uses the Active Operations Framework to compute expressions incrementally. ATOL has previously been showcased on TTC cases such as the TTC 2018 Social Media Case [3] or the TTC 2021 Incremental Workflow Case [4].

Traditional ATL engines execute the whole transformation at once to produce a target model from a source model. This is referred to as batch execution. With ATOL, the transformation is first applied to a source model to produce a target model, like with a batch transformation. But, unlike with standard ATL engines, ATOL keeps a propagation graph in memory. Using this graph, changes applied on the source model can be propagated to the target without recomputing the whole transformation. This allows for faster updates to the target model at the cost of an increased memory footprint.

ATOL supports a subset of standard ATL. For instance, ATOL natively supports only unique lazy rules, helpers, functional ATL, and parts of OCL operations. It also differs from standard ATL on specific points. For instance, ATOL requires target tuple navigation for called lazy rules (see Listing 1), it also supports implicit `collect` on property navigation on collections.

```
rule SumEHRTransaction {
    from
        s : KMEHR!TransactionType
    to
        t : FHIR!Composition mapsTo s (
            ...
            section <- Sequence{
                thisModule.MedicationSection(s).t,
                thisModule.AllergyIntoleranceSection(s).t,
                thisModule.ActiveProblemSection(s).t,
                thisModule.ImmunizationSection(s).t,
                thisModule.HistorySection(s).t
            }
        ),
        ...
}
```

Listing 1: Part of a rule with target tuple navigation

3 Solution Overview

The proposed ATOL solution is similar to the reference ATL solution. However some changes were made to make it compatible with ATOL. Figure 1 illustrates the compilation pipeline of our ATOL solution.

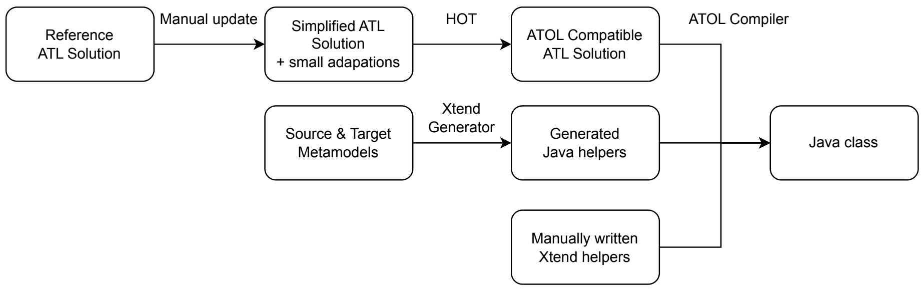

Figure 1: Compilation pipeline or the ATOL solution

The reference ATL transformation uses many advanced feature of EMFTVM and was not compatible with ATOL. For ATOL to be able to process it, we updated the reference transformation to use simpler ATL constructs supported by ATOL. These changes are done in two steps.

The first step is a manual rewriting of parts of the transformation to make it compatible with ATOL (either by simplifying the transformation or adapting it to ATOL-specific syntaxes). These rewritings are either too small and local to be worth automating (such as adding typing hints were ATOL fails to properly compile) or require understanding of the transformation semantics (such as rewriting rules with multiple inputs to rules with single inputs). Some helpers that cannot be compiled with ATOL are also rewritten with native Xtend code, such as the ATL helpers that used the `#native` syntax or the `lazy` rules `FhirString`, `FhirBoolean`, `FhirPositiveInt` and `FhirDecimal`.

The second step is performed by a Higher Order Transformation (HOT) written in ATL, and applied using a standard ATL engine. This HOT transforms standard matched rules to unique lazy rules without guards, and produces a `RESOLVE` helper that is used instead of rule matching in the transformation. Listing 2 shows part of the generated helper.

After these two steps, the produced ATL transformation is compatible with ATOL, and can be compiled to a Java class. The transformation is applied by calling the now (as a result of the HOT) *unique lazy* rule `DocumentRoot`, which transforms the root of the source model.

4 Results

The original ATL reference transformation is a batch transformation, thus the case does not provide data to test changes on the source model. To evaluate the correctness of our solution, we compared outputs of our solution with outputs of the reference solution for the three given source models. When doing so, we observed that our solution produces identical target models when serialized in the FHIR format (ignoring attributes that rely on `uuid`, which are randomly generated).

```
helper context OclAny def: RESOLVE : OclAny =
        if if self.oclIsKindOf(KMEHR!DocumentRoot) then
                let cp : KMEHR!DocumentRoot = self.oclAsType(KMEHR!DocumentRoot) in
                true
        else
                false
        endif then
                thisModule.DocumentRoot(self.oclAsType(KMEHR!DocumentRoot)).t
        else
                if if self.oclIsKindOf(KMEHR!FolderType) then
                        let cp : KMEHR!FolderType = self.oclAsType(KMEHR!FolderType) in
                        not cp -- @type kmehr!FolderType
                        .patient -- @type kmehr!PersonType
                        .oclIsUndefined() -- @type Boolean
                        -- @type Boolean

                else
                        false
                endif then
                        thisModule.Folder(self.oclAsType(KMEHR!FolderType)).t
                else
    ...
```

Listing 2: Part of the generated RESOLVE helper

Overall, the structure of the transformation is close to the reference one. However, compatibility with ATOL forces us to rewrite advanced ATL constructs. This can reduce transformation readability and maintainability.

Figures 2, 3 and 4 show runtime performance of our proposed solution. We can see that both ATOL and reference solutions have similar performances for load and initialization. For the actual application of the transformation, ATOL is a bit slower with models of a smaller size but scales better than the reference solution.

However, in Figure 5 we see that ATOL is consuming a lot more memory than the reference solution. ATOL uses more memory because, on top of the trace, it stores the propagation graph, which is needed to compute and propagate updates when the source model changes. One should note that the current version of our solution has not been optimized for memory and thus represent a worst case scenario.

Basic incremental updates on the source model (e.g., modifying properties of source elements) should work without issues. However, we know that null values in the source model will most likely cause crashes because the original transformation has not been strengthened against all possible null values.

The current implementation of ATOL also suffers of a known bug related to the rule matching system we use. As described in section 3, we use a HOT to replace all *resolvings* with a call to a *resolve helper* that calls the correct rule. However, after the initial transformation is applied, source elements can mutate, and they could now be matched by other rules. When this happen, the old rule application should be removed/disabled, and the result of the new rule application added. However, at the moment, ATOL cannot easily deactivate the bindings of the old rule. Thus, changes are still propagated through the old

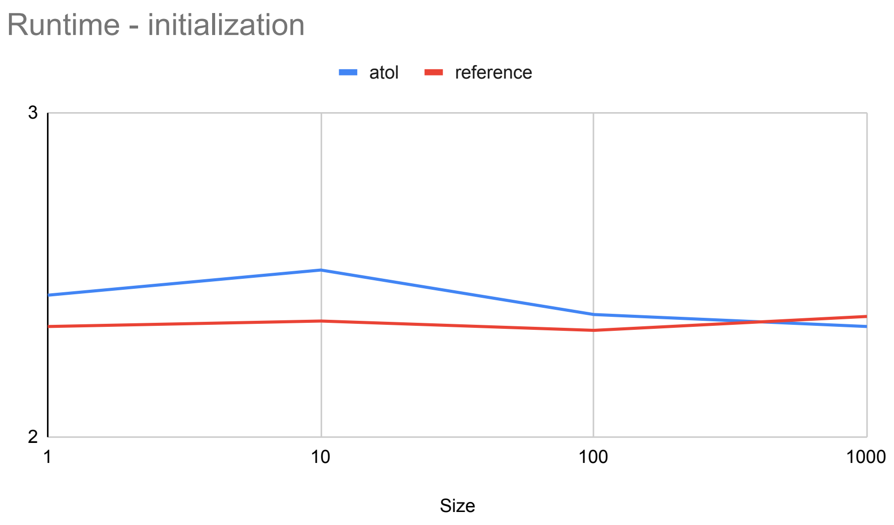

Figure 2: Initialization time in seconds for ATOL & reference solution

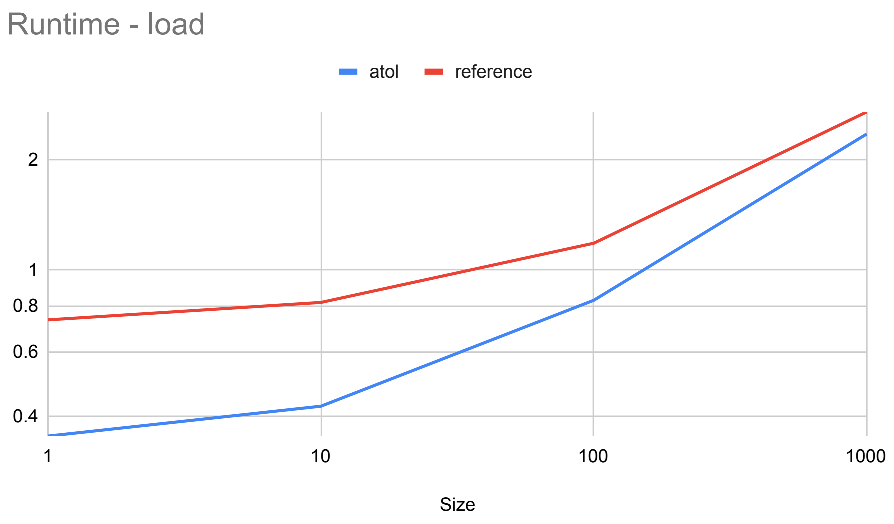

Figure 3: Loading time in seconds for ATOL & reference solution

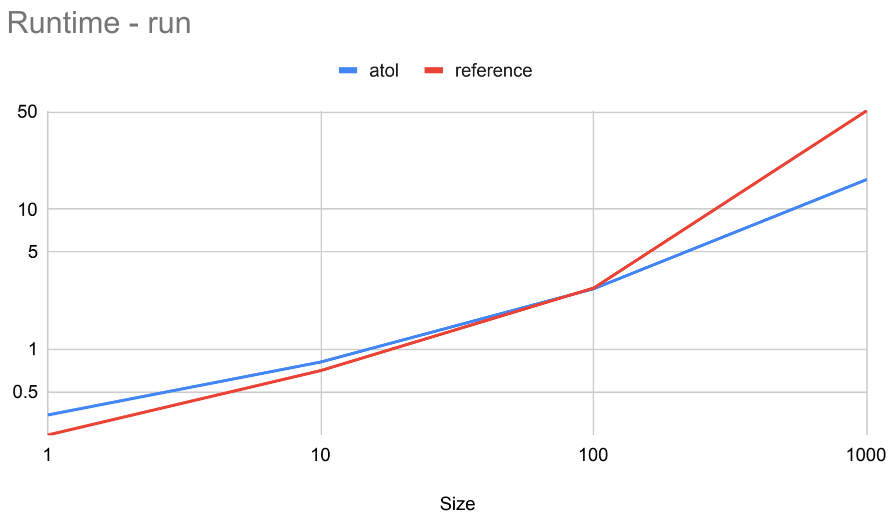

Figure 4: Run time in seconds for ATOL & reference solution

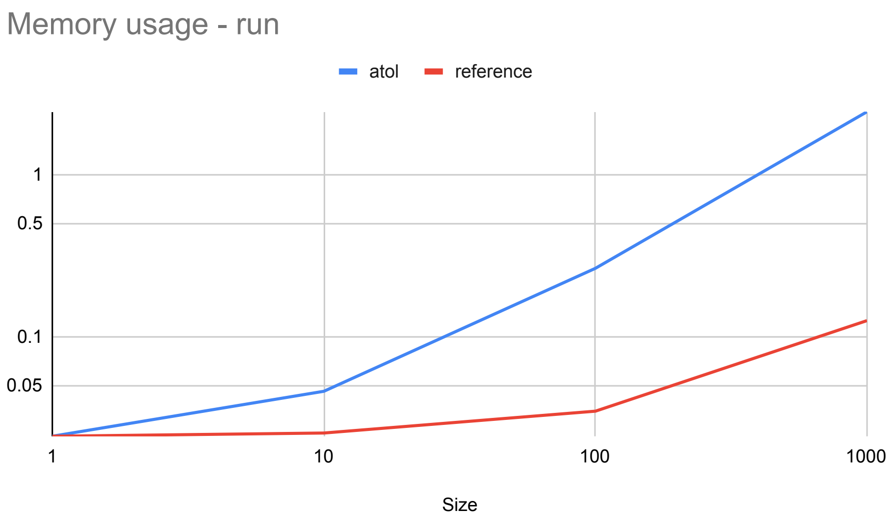

Figure 5: Memory usage in gigabits for ATOL & reference solution

rule bindings, which can cause crashes because of inconsistent properties. We identified this issue with our TTC 2023 incremental Class to Relational case, and are working on a fix.

5 Discussions

In section 3 we presented an overview of the changes we applied to the reference ATL transformation. In this section we discuss, with more details, the kind of changes we made to the reference solution, and why, as well as the improvements we made to ATOL and the tooling around it.

5.1 Differences with the reference solution

We made two kinds of changes, some that simplified the transformation without breaking compatibility with EMFTVM, and ATOL-specific changes.

5.1.1 Simplification

Helper inlining: ATOL currently only supports attribute helpers in the context of source metamodel types. To circumvent this limitation, several helpers on Strings were inlined (e.g. `normalize` or `toGender`). The impact of this change is limited, because most of these helpers were rarely used.

Manual type hints in bindings: sometimes the Java code produced by ATOL fails to compile because the Java compiler fails to unify the types. In these situations, we added `.oclAsType(<type>)` operation calls to explicitly type the expression, and fix these errors.

Rewriting of matched rules: as mentioned in section 3, ATOL only supports unique lazy rules. In order to compile the transformation, we applied a HOT that replaces all matched rules by unique lazy ones, adds a `RESOLVE` helper that calls the correct rule for its source, and adds calls to that helper when resolving is needed. This HOT is still a work in progress, and needs more work before it can be released. That is why it is not present with the ATOL solution. Instead, we provided both its source, and its target ATL files.

Rewriting rules with multiple inputs to single input: ATOL supports calling lazy rules with multiple source elements but the HOT that transforms matched rules to unique lazy ones does not. In the KMEHRToFHIR transformation, rules with multiples inputs can easily be rewritten to rules with a single input element, as other elements can be recovered using navigation. In older versions of ATL, it was a good practice to avoid rules with multiple inputs when non-necessary, to avoid the Cartesian product matching performance cost. With recent improvements to the matching algorithm of EMFTVM (since ATL 4.8.0), this old guideline is not that relevant anymore.

Rewriting some lazy rules to unique lazy rules: unique lazy rules are the only kind supported by ATOL, because in incremental contexts it is important to keep caches and not recreates target elements. Most lazy rules in the reference transformation were replaced by unique lazy rules without troubles. For the few ones that required the non-unique behavior (namely the `FhirString`, `FhirBoolean`, `FhirPositiveInt` and `FhirDecimal`) we implemented them as native Xtend helpers. For `Coding` and its subrules we added a naive support for non-unique lazy rules to ATOL. Like the previous change, this is also a good practice to prefer unique lazy rules instead of lazy rules when possible.

Rewriting a call to super rule into several call to subrules: in the rule `Folder`, the original transformation computed the union of many elements which are transformed by matched abstract rules. Because of typing issues with ATOL we instead applied the subrules for each elements before merging them in a single collection.

Rewriting of multiple inheritance in an additional rule: ATOL does not support multiple rule inheritance. Multiple rule inheritance is only used once in the transformation, for the `SumEHRTransactionWithAuthorAndCustodian` rule. In our solution, we simply duplicated code from both inherited rules into the subrule.

5.1.2 ATOL-specific Changes

Up to now, previous changes were compatible with other ATL engines. However, ATOL provides features not supported by other engines, and also requires some specific modifications.

Change to helper type declaration: enumeration types are handled as Strings in ATOL, therefore all instances of enumeration literals types were replaced by Strings.

Navigation into lazy rule target tuple: calling a lazy rule with ATOL returns a target tuple, and not just the first element. Therefore, with ATOL, it is mandatory to suffix all calls to lazy rules with the name of the element to be accessed in the target tuple (e.g. `thisModule.CompositionBundleEntry(s.transaction).be`). This is a breaking change because other engines do not support this.

Replacement of the join helper with a native operation: the iterate operation is not supported by ATOL. In this situation, the `join` helper that used the iterate operation was replaced by a custom join operation written in Xtend.

5.2 Improvements to ATOL

The KMEHR to FHIR transformation uses many aspects of ATL, several of which were not handled by ATOL. In order to compile the transformation we added support for:

- Maps and mutable sequences;
- custom `join` operation;
- initial support for non-unique lazy rules
- automated conversion between enumeration literals and Strings in our helper generator.

6 Conclusion

In this paper, we presented our solution to the KMEHR to FHIR TTC 2023 case. This solution is based on the reference ATL solution but is compiled with ATOL, which makes it incremental. Because ATOL supports only a subset of ATL, changes to the reference transformation (both manual and automated) had to be done.

We showed that runtime performance of the produced code is similar to the reference solution with slightly better scaling. We also showed that the generated target models are identical to those generated by the reference solution.

Basic incrementality should be working. We identified several propagation bugs due to the way the transformation is written, or because of known limitations in ATOL.

Finally, this case allowed us to improve our HOT and ATOL compiler.

References

[1] Boronat, A. Expressive and efficient model transformation with an internal DSL of xtend. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (oct 2018), ACM.

[2] Hinkel, G. NMF: A multi-platform modeling framework. In *Theory and Practice of Model Transformation*. Springer International Publishing, 2018, pp. 184–194.

[3] Hinkel, G., Garcia-Dominguez, A., Schöne, R., Boronat, A., Tisi, M., Calvar, T. L., Jouault, F., Marton, J., Nyíri, T., Antal, J. B., Elekes, M., and Szárnyas, G. A cross-technology benchmark for incremental graph queries. *Software and Systems Modeling 21*, 2 (dec 2021), 755–804.

[4] Jouault, F., and Le Calvar, T. (Ab)using incremental ATL on the TTC 2021 incremental laboratory workflow benchmark. In *TTC 2020/2021 - Joint Proceedings of the 13th and 14th Tool Transformation Contests. The TTC pandemic proceedings with CEUR-WS co-located with Software Technologies: Applications and Foundations (STAF 2021), Virtual Event, Bergen, Norway, July 17, 2020 and June 25, 2021* (2021).

[5] Le Calvar, T., Jouault, F., Chhel, C., and Clavreul, M. Efficient ATL incremental transformations. *J. Object Technol. 18*, 3 (2019), 2:1–17.

[6] Wagelaar, D. The TTC 2023 KMEHR to FHIR Case. In *TTC 2023* (2023).