Incremental ATL Solution to the TTC 2023 KMEHR to FHIR Case

Frédéric Jouault, **Théo Le Calvar**, and Matthew Coyle

Introduction

- Context
  - Medical data transformation
- Problem
  - The reference transformation has no support for incrementality
- Approach
  - Make it run with an incremental ATL engine: ATOL
- Results
  - Faster & (partially) incremental execution
  - This required some changes
    - Some manual changes to the original transformation
    - Compiler / pipeline improvements

ATOL Features

- Supports fine-grained online incremental execution of ATL transformations
- Compiles ATL to Java code
  - That makes use of the Active Operations Framework for incrementality
- Efficient initial & incremental computations
  - As demonstrated on
    - the Viatra CPS benchmark
    - the TTC 2018 social network case
- Provides a language extension mechanism
  - Example application
    - ATLc: coupling constraint solvers with transformations

Experimental Non-official Changes wrt. Classical ATL

- Implicit collect (*non breaking*)
- Navigation into lazy rule target tuples (*breaking*)
  - To make it possible to access other target elements
  - Alternative
    - Writing one rule per target element
      - Which requires adding rule calls
      - Whereas a simple variable access would otherwise be enough

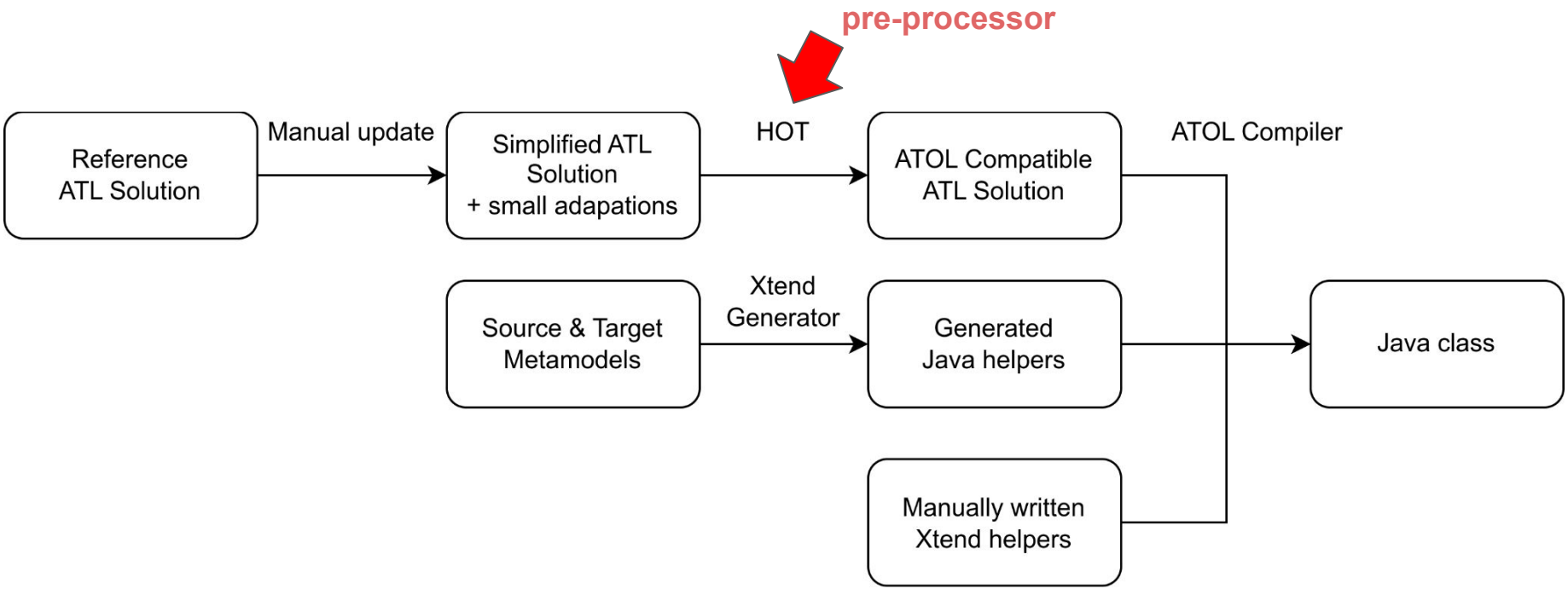
```
unique lazy rule A2B {
    from  a : MMA!A
    to    b : MMB!B,
          c : MMB!C
}
```

	Getting the first target element	**Getting the second target element**
Classical ATL	`thisModule.A2B(s)`	**Impossible**
ATOL	`thisModule.A2B(s).a`	`thisModule.A2B(s).b`

Current ATOL Compiler Limitations wrt. this Case

- No support for
  - Rule guards/filters
  - Multiple rule inheritance
  - Standard rules (only lazy ones are supported)
  - Rule-local variables (“using” block)
  - Enumeration literals
  - iterate expressions
  - #native code call
  - Lazy rule call without target tuple navigation
- Restrictions
  - Some navigations require disambiguation for the generated Java code
  - EMF does not generate change events for derived properties
    - They will not be incrementally updated
    - They could be rewritten as OCL helpers
      - which would automatically make them incremental
- Some of these issues can be handled by pre-processing

ATOL Solution Overview



Changes Performed by the Pre-processing HOT

- Standard rules into unique lazy rules
- RESOLVE helper generation
  - To dispatch elements to the appropriate rules
- RESOLVE helper call insertions
  - Requires typing information

This pre-processing approach can be extended to automatically overcome more compiler limitations.

Reference Transformation Simplifications

- These changes do not break compatibility with other ATL engines
- Improvements (arguably)
  - Changing some lazy rules into unique lazy rules
- Necessary because of current compiler limitations
  - Rewrote some calls to super-rules into calls to sub-rules
  - Removing multiple rule inheritance
    - In this case the cost is relatively low (duplicating two target pattern elements)
  - Refactored some expressions (inlining, helper extraction)
    - e.g., for rule-local variables
- Necessary because of current preprocessing HOT limitations
  - Rewriting source patterns with multiple elements into patterns with a single one
    - Because it is possible in this case & ATOL has no optimized local search plan

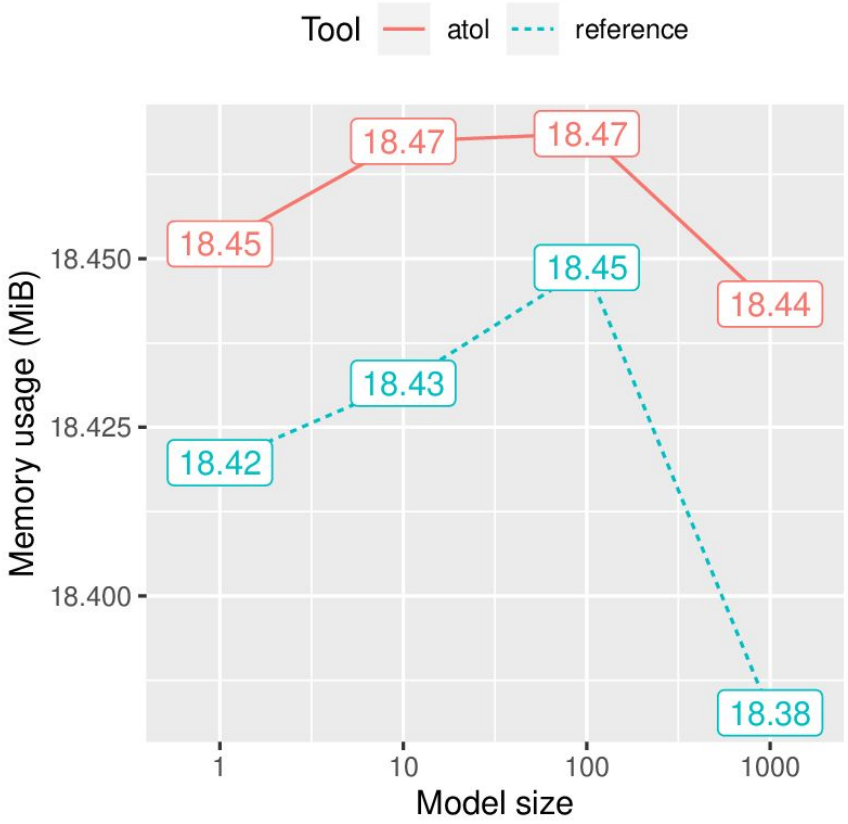
Reference Transformation Adaptations

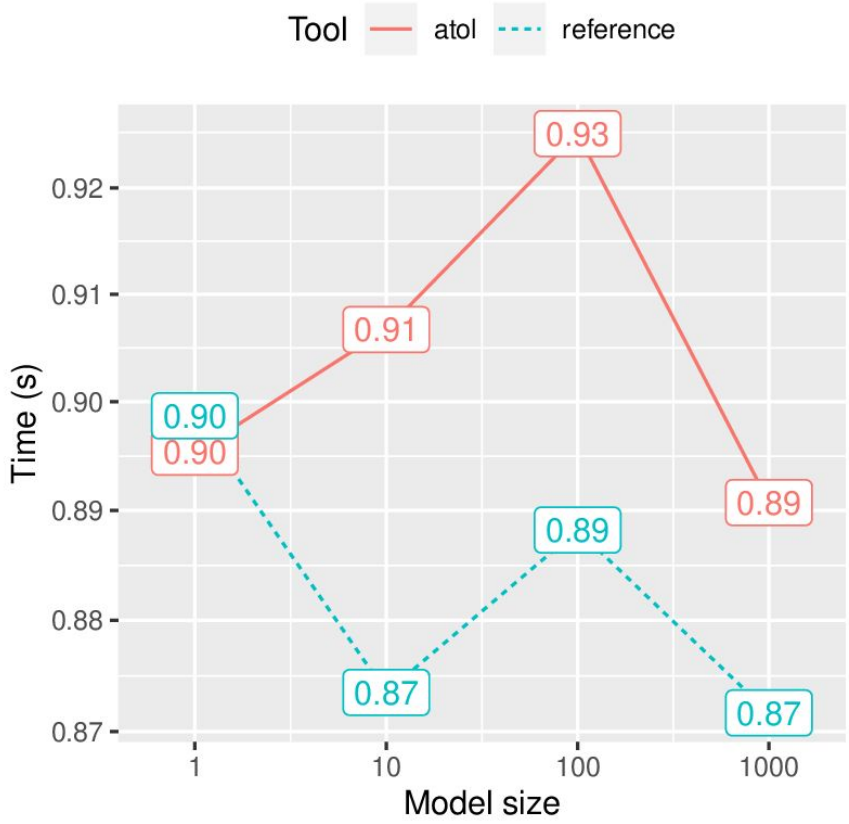
- These changes break compatibility with other ATL engines
- Because of current ATOL limitations
  - Added disambiguation suffixes to property names
  - Used strings instead of enum literals
  - Native `join` operation to avoid unsupported `iterate` expression
  - #native calls rewrote into xtend helpers
  - Added target tuple navigation to lazy rule calls
- Robustness improvement
  - Added `->reject(e | e.oclIsUndefined())` on some singletons

Robustness Improvements for Incrementality

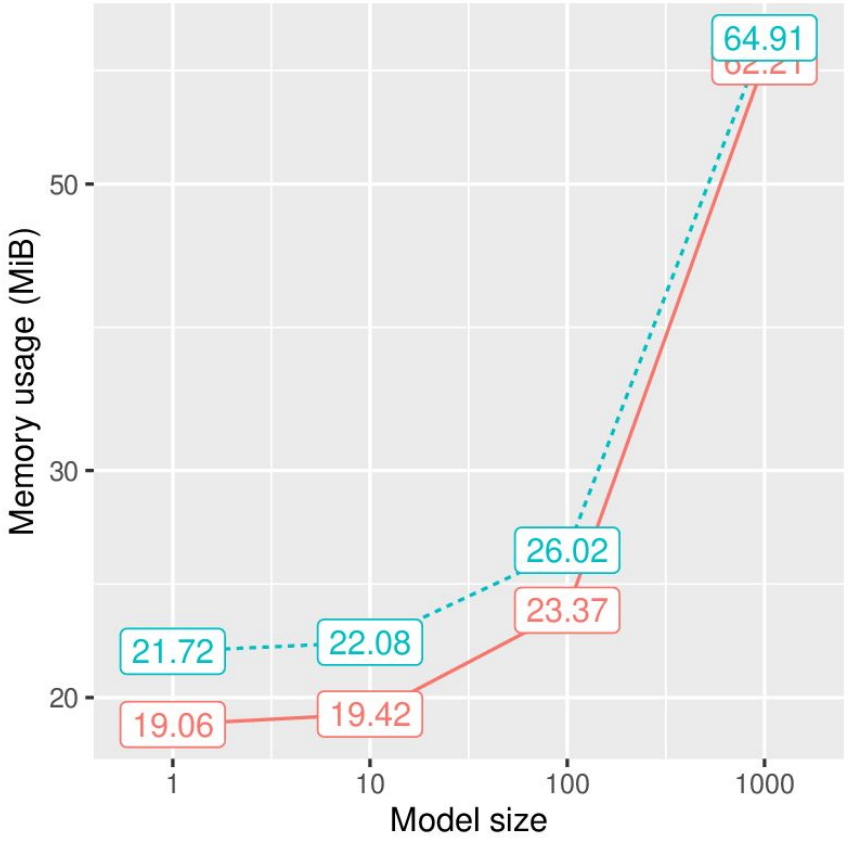
- Original transformation assumed source model was correct,
- Source model can be in an incorrect state when a set of change is applied,
- ATOL apply change atomically so the transformation has to deal with these incorrect states,
- In order to have a working incremental transformation the code needs to be hardened so that it can cope with incorrect values
  - Mostly dereferencing unset relations
- Possible action:
  - Introducing filters everywhere to remove null values

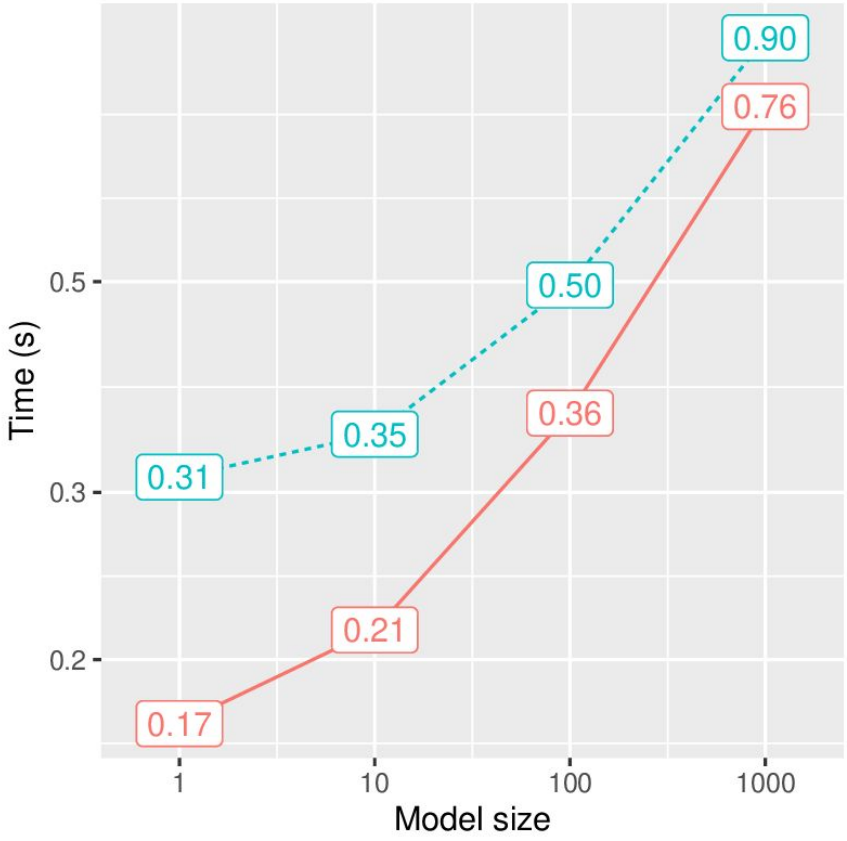
Memory & Runtime - Initialization



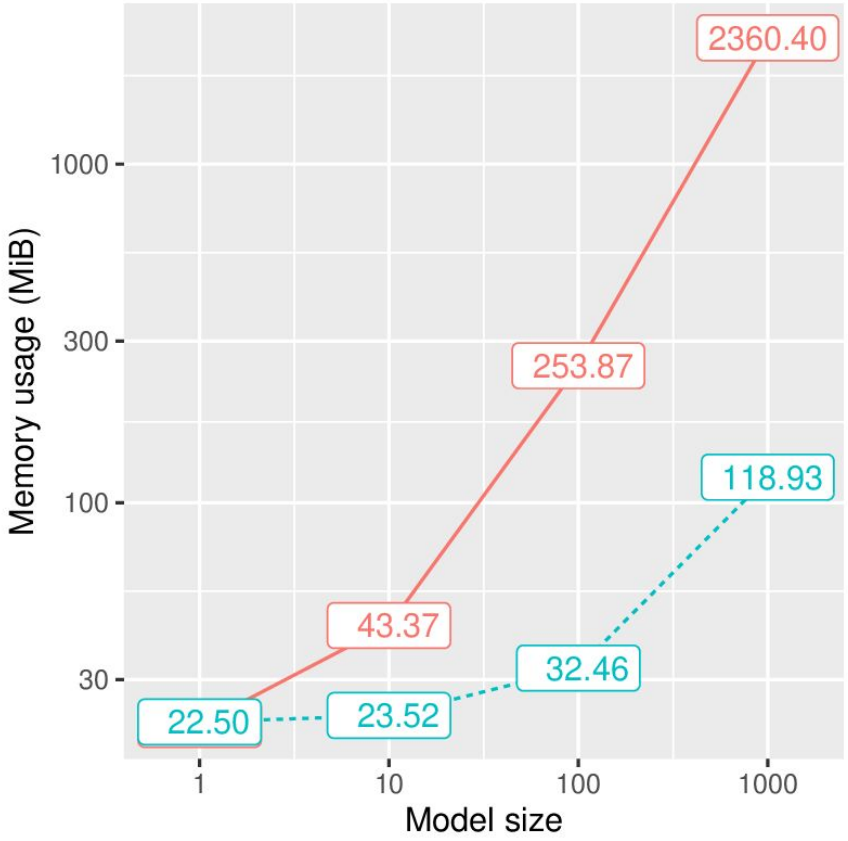


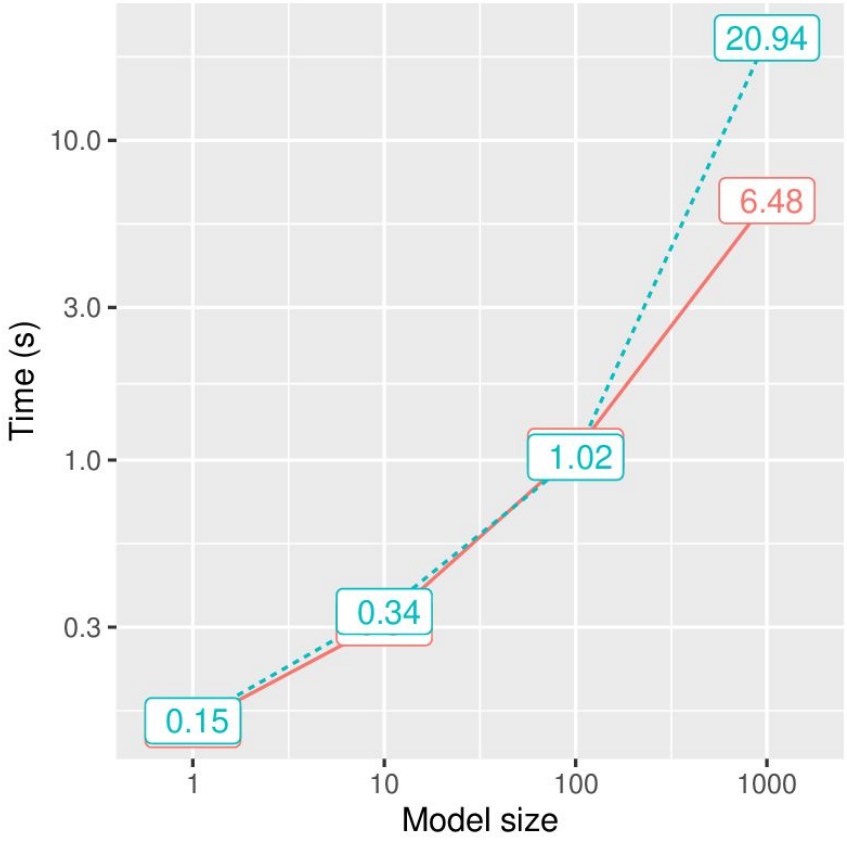
Memory & Runtime - Load





Memory & Runtime - Run





Memory Usage Considerations

- ATOL uses significantly more memory than ATL
  - Because it needs to keep the propagation graph
- We have not tried to optimize memory usage
- More caching is probably possible, with following strategies
  - Separating intermediate computations into distinct attribute helpers
    - Which will add a cache for each attribute hepler
  - Improving preprocessing
    - Performing this separation into distinct helpers automatically
  - Improving the ATOL compiler
    - Inserting caches without requiring distinct helpers

Conclusion

- ATOL is able to provide efficient incremental execution for ATL transformations
- This case helped us more clearly identify
  - Some ATOL limitations
  - Ways to overcome them

Thanks for your attention!

Outline (see notes doc for more details)

- Introduction
  - Context
  - Problem: original transformation has no support for either incrementality (or bidirectionality)
  - Approach: make it run with an incremental ATL engine
  - Results: required some changes, but faster & (partially) incremental
- ATOL Overview
- Solution Overview
- Results
  - required changes: some to simplify the problem, some because of ATOL incompatibilities
  - faster, but more memory-hungry (which is typically an incrementality trade-off)
  - limitations
    - no bidir because of data type translations
    - Kinda broken incrementality