# (Ab)using incremental ATL on the TTC 2021 incremental laboratory workflow benchmark

Frédéric Jouault[1], Théo Le Calvar[2]

[1]*ERIS, ESEO-TECH, Angers, France*
[2]*IMT Atlantique, LS2N (UMR CNRS 6004), Nantes, France*

**Abstract**
The TTC 2021 Incremental Laboratory Workflow benchmark presents a challenging incremental transformation problem. This paper presents our ATOL-based partial solution, and discusses the reasons why the kind of incrementality supported by ATOL does not match the kind of incrementality required to solve the problem.

**Keywords**
ATL, Incremental Model transformation, Incremental Recompilation

## 1. Introduction

Incrementality is a mechanism by which a computation result can be updated after input changes, without having to perform all the computations required to produce the initial result again. This results in a significant performance advantage, because much fewer computations typically need to be performed after each input change than were initially necessary. Some model transformations can be executed incrementally, depending on the language they are specified in, and the execution engine that is used. For instance, some ATL transformations can be quite efficiently executed incrementally with the ATOL [1] engine, but other approaches exist for ATL [2], or other languages [3, 4, 5]

However, not all incremental problems have the same properties, and it is not clear which incremental transformation engine can be used for each incremental problem. The TTC 2021 Incremental Laboratory Workflow benchmark [6] was specified in order to provide means to compare incremental engines on a very specific incremental problem. This paper presents a partial solution to this problem written in ATL, and executed with ATOL. Because the required kind of incrementality does not match what ATOL provides, the whole problem could not be solved without abusing ATOL.

The remainder of this paper is organized as follows. Section 2 reminds the reader about what kind of incrementality can be achieved with ATOL. An overview of the problem to solve is given in Section 3. Section 4 presents our ATOL-based solution to this problem. Experimental results are briefly presented in Section 5, and finally Section 6 concludes.

## 2. Incremental ATL with ATOL

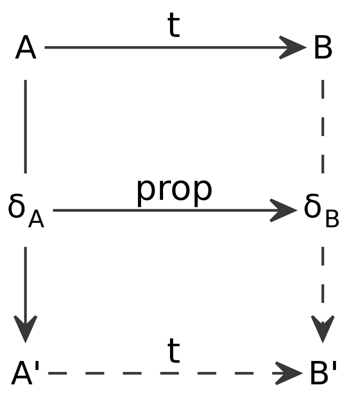
The declarative nature of ATL makes it easily amenable to different execution modes. The original execution mode is called *batch* mode, and simply creates a (set of) target model(s) from a (set of) source model(s)[1]. Incrementality is a different execution mode supported by more recent ATL execution engines, which are able to propagate source model changes into corresponding target model changes. ATOL [1] is an incremental ATL execution engine, which leverages composable *active operations* [7] in order to achieve model transformation incrementality.

The definition of incrementality supported by ATOL is that change propagation should be equivalent to full re-execution. In this paper, we will call this *commutative incrementality*. This is illustrated by the commutative diagram represented in Figure 1, in which:

- $t$ is a model transformation, which can be viewed as a function.
- $A$, $A'$, $B$, and $B'$ are models. $B$ is the result of applying $t$ to $A$, which can be written: $B = t(A)$. $B'$ is the result of applying $t$ to $A'$, which can be written: $B' = t(A')$. $B$ and $B'$ are target models of $t$, whereas $A$ and $A'$ are source models of $t$.
- $\delta_A$ and $\delta_B$ are changes, viewed as functions, respectively performed on $A$, and $B$, turning them into $A'$, and $B'$. This can be written: $A' = \delta_A(A)$, which is a source change wrt. $t$, and $B' = \delta_B(B)$, which is a target change wrt. $t$.

[1]Although ATL is capable of handling multiple source and target models, the remainder of the paper will generally refer to a single source, and a single target model to simplify explanations.

*TTC'21: Transformation Tool Contest, Part of the Software Technologies: Applications and Foundations (STAF) federated conferences, Eds. A. Boronat, A. García-Domínguez, and G. Hinkel, 25 June 2021, ~~Bergen, Norway~~ (online).*
frederic.jouault@eseo.fr (F. Jouault); theo.le-calvar@imt-atlantique.fr (T. Le Calvar)
0000-0002-2395-9623 (F. Jouault); 0000-0003-2273-2053 (T. Le Calvar)



$$\begin{tikzcd}
A \arrow[r, "t"] \arrow[d, no head] & B \arrow[d, dashed, no head] \\
\delta_A \arrow[r, "prop"] \arrow[d] & \delta_B \arrow[d, dashed] \\
A' \arrow[r, dashed, "t"] & B'
\end{tikzcd}$$

**Figure 1:** Commutative incrementality diagram for transformation $t$ creating $B$ from $A$

- $prop$ is a change propagation higher-order function, which can translate changes on $A$ into changes on $B$. $\delta_B$ can thus be obtained from $\delta_A$ by applying $prop$, which can be written: $\delta_B = prop(\delta_A)$

This diagram commutes because $B' = t(\delta_A(A)) = \delta_B(t(A))$. This can be rewritten without referring to $\delta_B$ in the following way: $B' = prop(\delta_A)(t(A))$. Transformation re-execution corresponds to $B' = t(\delta_A(A))$, whereas change propagation corresponds to: $B' = prop(\delta_A)(t(A))$.

ATOL automatically derives $prop$ from $t$, making it relatively easy for transformation developers to execute classical ATL transformations incrementally. Although the same $B'$ can be theoretically obtained from $A$, $\delta_A$, and $t$ by following either path, each one has specific practical properties:

- **Change propagation is generally faster.** Transformation re-execution requires about the same amount of computation for most simple source changes[2]. Change propagation is typically much faster for simple changes, because it does not need to process the whole source model again.
- **Change propagation performs in-place updates.** Very often, only the actual structure of a model matters, and models duplicated by re-executing a whole transformation are equivalent to in-place updated ones for all purposes. However, in some situations, object identity matters, and it makes a difference whether a model is updated in-place or not. This is the case, for instance, when target model elements are linked to non-model elements, such as Graphical User Interface (GUI) elements. Updating the existing model will not break its link with the GUI elements, whereas recreating it will result in a new model that does not have any link with the GUI elements.

[2]A precise definition of what a *simple* change is is beyond the scope of this paper. The intuition is that a simple change modifies a number of model elements which is small compared to the total number of elements in the model.

Now that the definition of commutative incrementality on which ATOL is based has been given, the role of *active operations* can be explained briefly. An *active operation* is a basic transformation operation, such as an OCL `collect` (called `map` in other languages), or `select` (called `filter` in other languages), equipped with a propagation algorithm respecting commutative incrementality. Model transformations can be specified by composing these operations, and change propagation is obtained by composing their change propagation algorithms.

## 3. Problem overview

The full specification of the TTC 2021 Incremental Laboratory Workflow benchmark is presented in [6]. This section gives a brief overview of the problem definition. Solving this problem basically requires developing a compiler for laboratory workflow jobs given to robotic liquid handlers. These robots can perform many tasks required to, for instance, analyze biological samples. However, they must be given very detailed descriptions of what low-level tasks, such as transferring tube contents, incubating samples, or washing containers, to perform on which samples. An abstract syntax for these low-level job descriptions is provided as part of the benchmark in the form of a metamodel called *JobCollection*.

Because this metamodel operates at a very low level of abstraction, a second, higher-level metamodel is introduced: *LaboratoryWorkflow*. At this level, it is only necessary to specify a collection of tasks to be applied to every specified sample. The job of the compiler to develop is first to create a model conforming to *JobCollection* for every model conforming to *LaboratoryWorkflow* it is given. This requires assigning samples to tube runners that may only contain a maximum of 16 samples, and then replicating jobs as necessary to transfer all samples onto microplates that may only contain a maximum of 96 samples. Moreover, all transfer operations must be performed on microplate columns that may only contain a maximum of 8 samples.

Then, the compiler must be able to support addition of new samples at any time, and to incorporate feedback from the robot in the form of information about partially failed jobs. When new samples are added after older samples have already been partially processed, they must be processed up to the same point by adding new jobs. When some samples fail to be properly processed by a job, they must be marked as failed so that all future jobs can skip them. However, already executed jobs must not be changed. Compilers that support applying these modifications incrementally are expected to be more efficient than those that must recompile the full process.

## 4. Solution presentation

The previous section has briefly presented the problem to solve, and this section continues by presenting our ATOL-based solution. The solution presented here was partially inspired by the NMF [4] solution provided in the `solutions/Reference` folder of the case repository[3].

### 4.1. Problem incrementality vs. ATOL incrementality

Firstly, change propagation must be able to proceed after either source modifications (adding samples), or target modifications (marking jobs as partially failed). This requires some level of bidirectional propagation, which *active operations* can achieve, but for which ATL is ill-equipped. Its OCL-based syntax does not support specifying enough information for reverse change propagation to work in all cases. For instance, an OCL `collect` operation can only take one lambda expression as argument to process elements in the forward direction, whereas bidirectionality additionally requires a lambda expression to process elements in the reverse direction. ATOL can, however, be used for this purpose because it enables users to specify some helper functions in xtend[4], rather than in ATL, while still keeping most of the transformation as declarative ATL code.

Secondly, the solution must be able to perform some changes while taking into account the fact that part of the process has already been executed by the robot. This means that change propagation should not result in the same model as what would be obtained by fully re-executing the transformation. The kind of incrementality required here is therefore noncommutative, whereas ATOL was designed to make sure it performs commutative incrementality.

### 4.2. Transformations

The overall process we implemented is represented on Figure 2. The source model (called M1) conforming to the *LaboratoryWorkflow* metamodel is first processed by the *LaboratoryChunking.atl* transformation, which distributes samples into tube runner-sized chunks, as well as in microplate column-sized chunks. The latter are then distributed into microplate-sized chunks (i.e., $96/8 = 12$ rows). This results in model M2 conforming to the *LaboratoryWorkflow'* metamodel, which extends the *LaboratoryWorkflow* metamodel to include information about chunks. This extension is basically the same as in the NMF solution, and is implemented by leveraging the capability of ATOL to consider metaclasses that are not specified in Ecore. It is defined in the LaboratoryWorkflow.xtend source file. This first transformation is relatively simple, but required implementing a new chunking active operation. Because appending samples is the only required change, we have not implemented propagation of other changes, such as removal, or insertion. Moreover, because this transformation targets a superset of its source metamodel, it was implemented using ATL's *refining* mode, which supports in-place model modification. This means that only the new elements must be created, and the already existing elements can be kept as is. This is relatively efficient when compared to a regular non-refining transformation, which would have to copy all the existing elements.

In a second step, model M2 is then transformed by Lab2Job.atl, which performs the actual translation to the *JobCollection* metamodel. Most of this transformation is relatively simple standard ATL code. The tricky part is related to noncommutative incrementality, and is detailed in the next section.

### 4.3. Abusing ATOL incrementality

Because the problem calls for change propagation that is not equivalent to full transformation re-execution (i.e., noncommutative incrementality), solving it with ATOL is not an easy task. We have not yet been able to fully solve the problem with ATOL. Nonetheless, we have managed to abuse it into performing noncommutative change propagations. To achieve this, the main mechanism is change filtering, which takes place for reverse propagation of tip state to sample state. This is performed using an *ad hoc* noncommutative active operation. It is implemented in method `mapState` of Lab2Job.xtend.

However, even with this hack, our solution is not able to perform the required incremental propagations correctly in all cases. The remaining problems with our solution could possibly be solved by similarly abusing ATOL, but we decided not to do so until we better understand what having noncommutative active operations entails. Here is the list of remaining problems in our solution that we have identified:

- **Problem 1.** Failed samples reappear in failed jobs, which is a change filtering issue. This may not be an actual problem in practice because these jobs will not be re-executed anyway.

[3]https://github.com/tecan/ttc21incrementalLabWorkflows/
[4]https://www.eclipse.org/xtend/

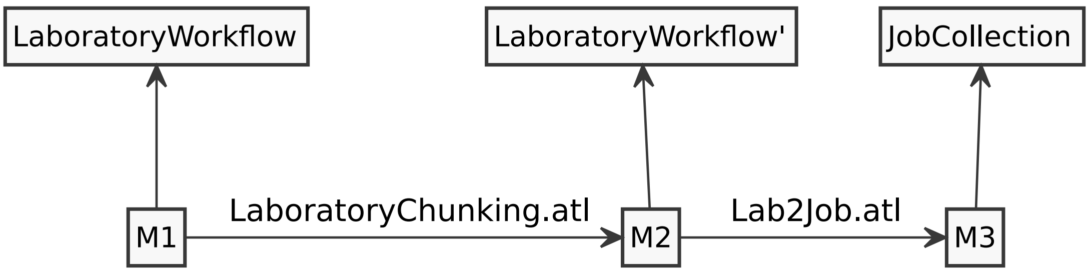


**Figure 2:** Overview of the transformation process of our ATOL-based solution

- **Problem 2.** Newly added samples require additional jobs to perform jobs already completed on existing samples. This is again a case of noncommutative incrementality, which we have not attempted to solve.

### 4.4. Non-ATL supporting code

In addition to ATL code, and the previously described xtend helpers, some xtend code was also required for the following aspects (identified with comments in the LaboratoryChunking.xtend and Lab2Job.xtend files):

- **Accessing tuple properties.** This is necessary on account of a current ATOL compiler limitation, but this code could be automatically generated.
- **Enumeration literal comparison helpers.** This could be implemented in ATL, although string comparison would have to be used because ATOL translates enumeration literals to and from strings.
- **Number zero-padding.** This is necessary to generate tube runner and microplate names that conform to what is used in the benchmark's change specification files (although [6] does not specify padding). This could be implemented in ATL, but would be cumbersome because of limited OCL support for string operations.

Finally, the transformation driver, and the change application code are also written in xtend.

## 5. Experimentations

Our ATOL solution to the TTC 2021 Incremental Laboratory Workflow benchmark is available on GitHub[5]. It reuses driver code from previous TTC contests.

Performance seems relatively good, as can be seen on Figures 3, 4, and 5, although we have not spent much time on optimization, which could still be performed. However, since our solution does not completely solve the problem, comparing its performance to other solutions may not be that meaningful.

[5]https://github.com/ESEO-Tech/ttc21incrementalLabWorkflows

## 6. Conclusion

This paper has presented our partial ATOL-based solution to the TTC 2021 Incremental Laboratory Workflow benchmark. Although this benchmark requires some kind of incrementality, it is not exactly the same kind of incrementality supported by ATOL, unless a better way to encode the problem as model transformation has eluded our investigations. We have shown that ATOL can be abused to at least partially solve the problem, but we have no theory with which to reason about our solution. It may therefore be the case that it could fail in unexpected ways beyond the already identified ones.

This benchmark is definitely not the one to showcase ATOL on, but it presents an intriguing problem. A new expanded theory of incrementality would be necessary to understand it precisely wrt. the active operation approach. However, it is not clear whether such a new theory would be general enough to support other relevant noncommutative incrementality problems, or if each new problem would require extending the theory.

## References

[1] T. Le Calvar, F. Jouault, C. Chhel, M. Clavreul, Efficient ATL incremental transformations, J. Object Technol. 18 (2019) 2:1–17. doi:10.5381/jot.2019.18.3.a2.

[2] S. Martínez, M. Tisi, R. Douence, Reactive model transformation with ATL, Science of Computer Programming 136 (2017) 1–16. doi:10.1016/j.scico.2016.08.006.

[3] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, Road to a reactive and incremental model transformation platform: three gen-

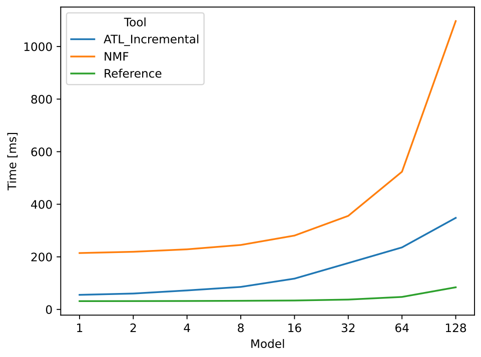
erations of the VIATRA framework, Software & Systems Modeling 15 (2016) 609–629. doi:10.1007/s10270-016-0530-4.

[4] G. Hinkel, NMF: A multi-platform modeling framework, in: Theory and Practice of Model Transformation, Springer International Publishing, 2018, pp. 184–194. doi:10.1007/978-3-319-93317-7_10.

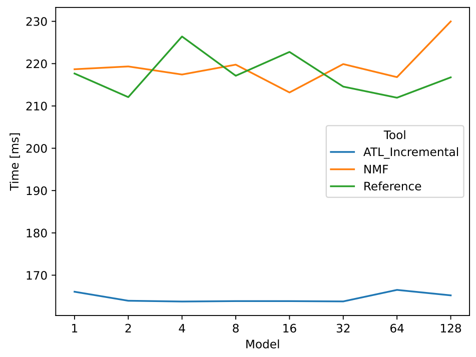
[5] A. Boronat, Incremental execution of rule-based model transformation, International Journal on Software Tools for Technology Transfer 23 (2020) 289–311. doi:10.1007/s10009-020-00583-y.

[6] G. Hinkel, Incremental recompilation of laboratory workflows, in: Proceedings of 14th Transformation Tool Contest (TTC 2021), 2021. To appear.

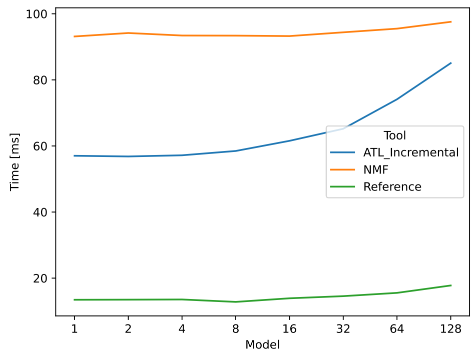
[7] O. Beaudoux, A. Blouin, O. Barais, J. Jézéquel, Active Operations on Collections, in: Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I, volume 6394 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 91–105. doi:10.1007/978-3-642-16145-2_7.



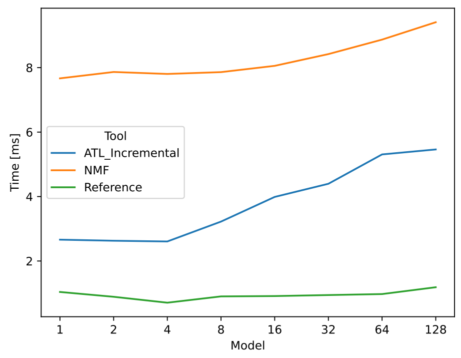
(a) Initial



(b) Initialization

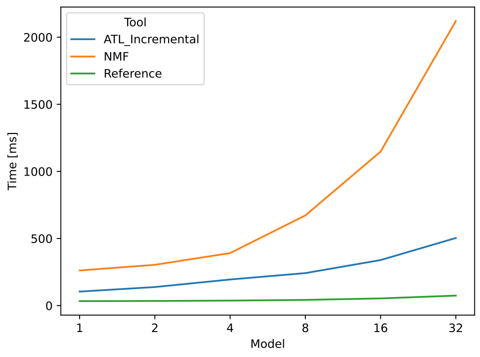


(c) Load

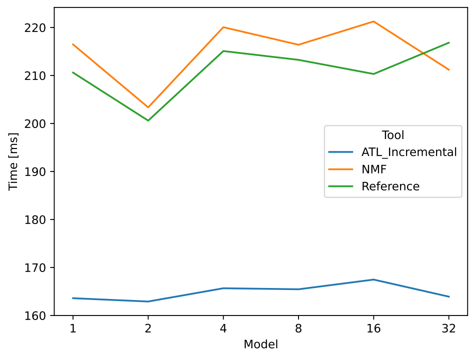


(d) Update

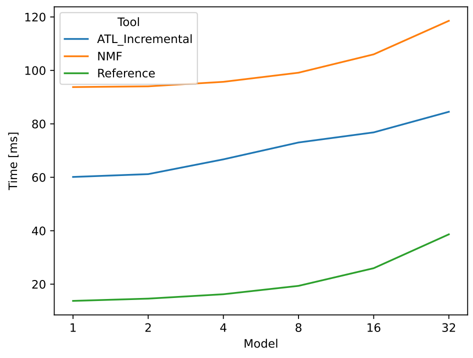
**Figure 3:** Benchmark results for *new_samples* scenario



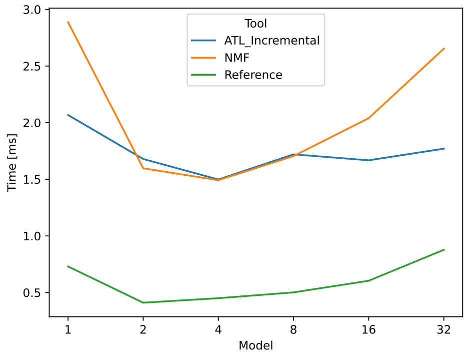
(a) Initial



(b) Initialization

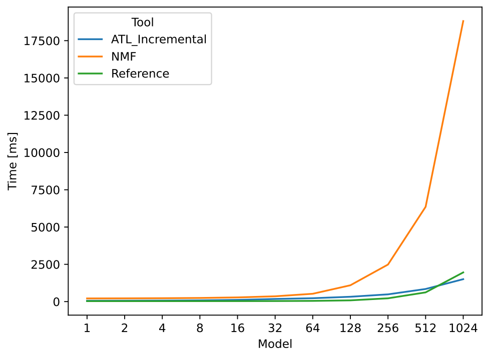


(c) Load

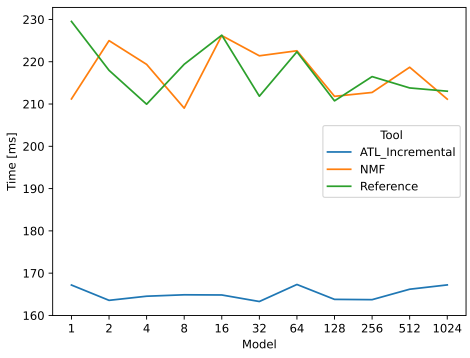


(d) Update

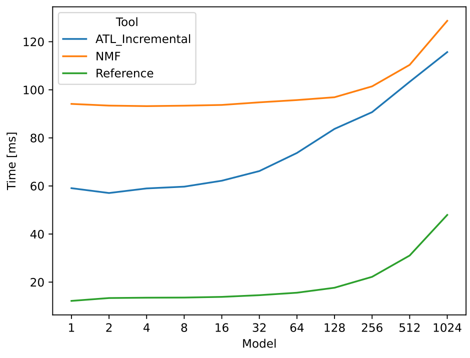
**Figure 4:** Benchmark results for *scale_assay* scenario



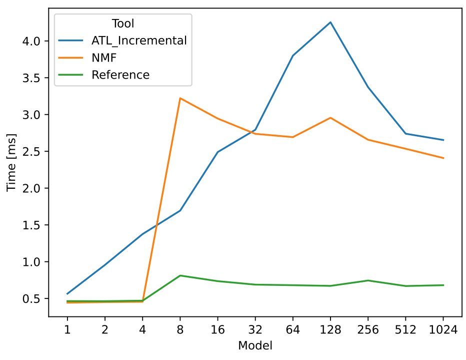
(a) Initial



(b) Initialization



(c) Load



(d) Update

**Figure 5:** Benchmark results for *scale_samples* scenario