Efficient OCL-based Incremental Transformations

Frédéric Jouault and Olivier Beaudoux
Groupe ESEO, Angers, France
firstname.lastname@eseo.fr

Introduction

- At OCL 2015 in Ottawa, we presented how active operations can incrementally evaluate OCL expressions
  - An atomic change in the queried model is propagated into changes to the values of the expressions that use the changed part
- At OCL 2016 in Saint-Malo, we show that this can be done efficiently, and in a scalable way
  - On a model transformation benchmark

CPS to Deployment Benchmark

- Originally a Viatra demonstrator
  - Overview
  - Domains
    - CyberPhysicalSystems & Deployment metamodels
  - Comes with some transformation correctness JUnit tests
- Extended into a benchmark
  - Source model generator
  - Performance tests, also JUnit-based
  - Multiple implementations (e.g., batch, IncQuery, Viatra)

Active Operations Framework (AOF)

- Active operations enable incremental execution of OCL-like expressions
  - Bidirectionality is also possible with some limitations, but the CPS to Deployment benchmark does not evaluate this
- Every mutable value is represented as an observable box
  - Either a *root* box corresponding to a model element property value
  - Or a *derived* box computed from other boxes using a sequence of active operations (e.g., select, collect)
  - Box types: Set, OrderedSet, Bag, Sequence, One, or Option
- Every active operation provides initial computation plus fine-grained propagation algorithms

AOF Benchmark Implementation

- Source model traversal strategy
  - This transformation can be implemented with explicit rule call (no pattern matching)
  - It corresponds to what one would write in ATL using
    - one regular matched rule for the root element
    - unique lazy rules in all other cases
  - This is more efficient than implicit rule call, so we chose this
- Syntax: we used xtend
  - Which enables much more concise syntax than Java (even Java 8 with lambdas)
  - By leveraging extension methods

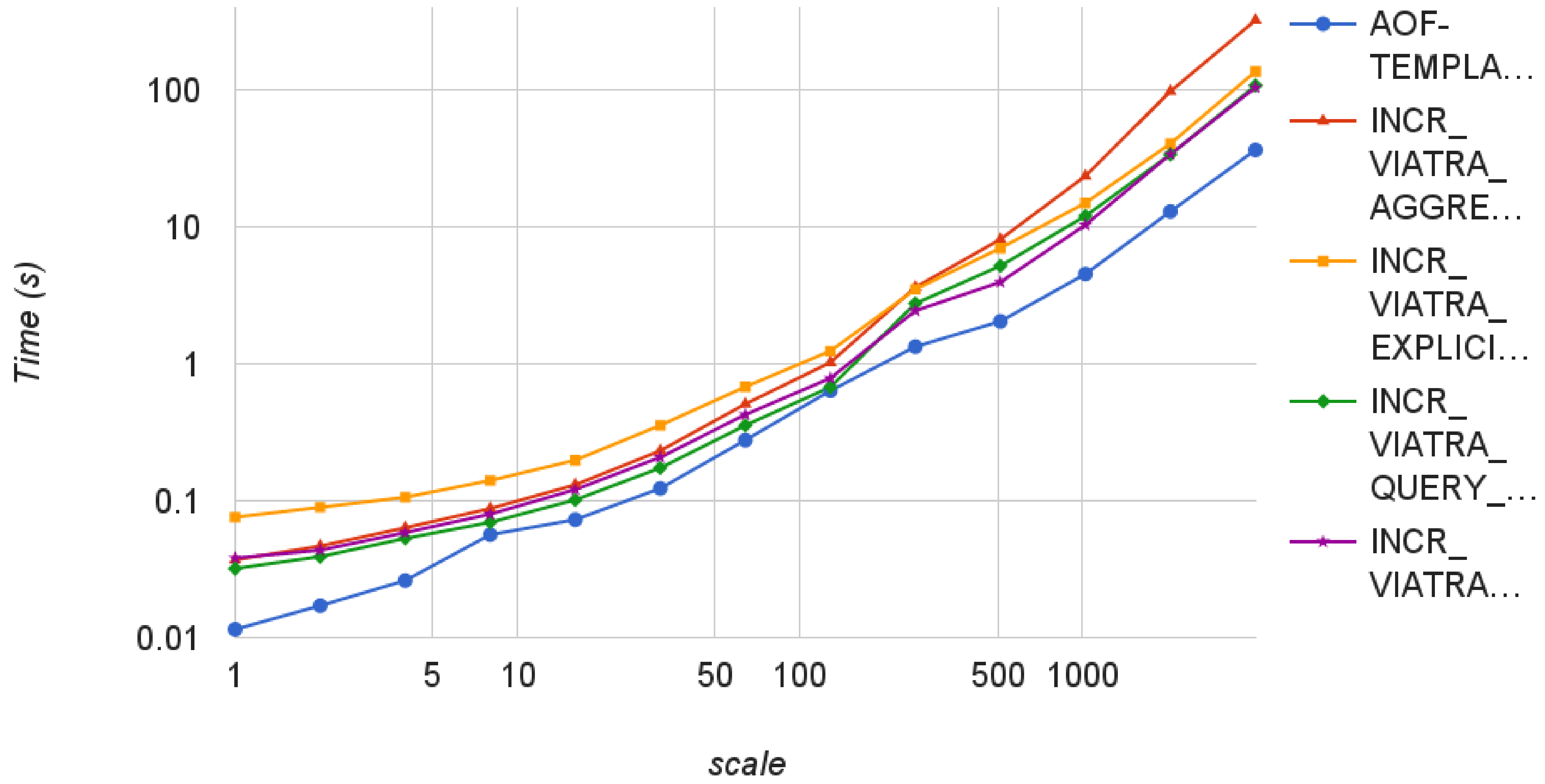
Syntax Example

- Embedded DSL in Xtend

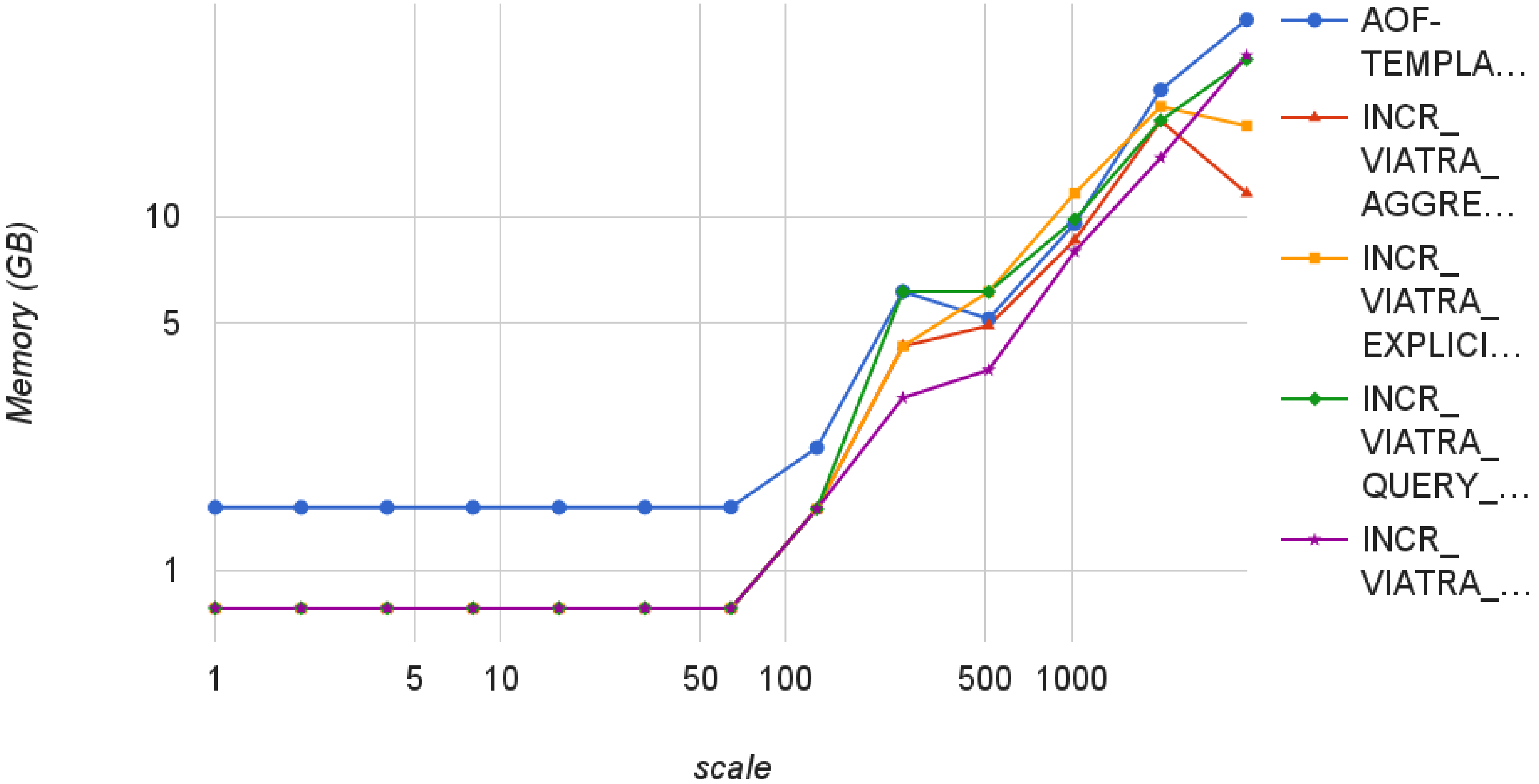
```
target._ip <=> source._nodeIp


target._applications <=>
        source._applications.collectTo(
                applicationInstance2DeploymentApplication
        )
```

- All _<property-name> are actually extension methods
- <=> is the spaceship operator overloaded to behave as a binding on boxes

Execution Time: Publish-Subscribe



Memory Usage: Publish-Subscribe



Required Optimizations 1/3

- AOF optimizations
  - Conversion of expensive sanity checks into asserts
    - They would not have been triggered any way because of the way we use AOF when writing a transformation
    - These asserts can be enabled for debugging
  - Local optimizations on some propagation algorithms, notably in asSet() and asOrderedSet()
- These issues had not been detected earlier because of the lack of a performance benchmark
- Early development focused mostly on correctness

Required Optimizations 2/3

- Removal of duplicate boxes creation
  - Duplicate boxes = created from the same property box by the same sequence of operations, with the same arguments
- Solution: caching the creation of boxes wrt. source box, operation, and its arguments
- However, some operation arguments are lambdas, which are not comparable in Java by default
  - Therefore, we had to write the transformation with explicit calls to caches
  - This will no longer be an issue once we have a fully functional OCL front-end
    - The interpreter or compiler can take care of specifying appropriate hashCode() and equals()
- This issue had also not been detected earlier because of the lack of a performance benchmark
- We implemented a duplicate box analyzer, which can point to such issues by analyzing the expressions structure (easier to use than a Java profiler)

Required Optimizations 3/3.1/2

- Specific operation: groupBy

```
let keys : Set(OclAny) =
    myCollection.collect(e | getKey(e))->asSet() in
keys->collect(key |
    Tuple {key = key, elements =
        myCollection->select(e | getKey(e) = key)})
```

Naive execution is quadratic.

We implemented an optimized active groupBy operation

```
myCollection.groupBy(getKey)
```

Required Optimizations 3/3.2/2

- Specific operation: selectBy

```
myCollection.select(e | selector(e) = someMutableValue)
```

This requires the creation of a large number of mutable booleans to represent the result of the equality check

We rewrote it into:

```
myCollection.selectBy(selector, someMutableValue)
```

By implementing an optimized active selectBy operation

Ideally, these kinds of rewrites should be automated

More Optimization Perspectives

- Optimizing parts of AOF not used in this specific benchmark
  - We need more benchmarks
- Reducing memory usage by not storing intermediate data that can be recomputed
  - A time-memory trade-off
- Giving all cache control to the user
  - Would not be more complex to use, but would save bookkeeping and memory
- Relaxing order preservation on Sets and Bags
- Parallel execution?
  - AOF has some similarities with Java 8 streams

Advantages of Active Operations over Viatra

- Collection ordering preservation
  - RETE-based Viatra does not preserve ordering
  - AOF always preserves it (even for Sets and Bags)
  - Execution, and result order is therefore deterministic
- Broader OCL compatibility
  - AOF is based on OCL operations
  - Only part of OCL can be translated into Viatra patterns
- AOF has some bidirectional propagation capabilities, while Viatra does not
  - But this is not necessary here

Drawbacks of Active Operations Compared to Viatra

- Require the transformation to be expressed in an OCL-like manner
  - Remark: this was also an advantage, but unfortunately, this may be seen as an inconvenient by some
- No pattern matching
  - As available in Viatra for instance
- No global optimization

Conclusion

- Active operations are as scalable as state-of-the-art Viatra
- This is only measurable thanks to the Viatra benchmark
  - Actually only **achievable** thanks to that benchmark
- But benchmark creation is expensive
- There is a real need for a benchmarking framework
  - For incremental (and/or bidirectional) engines
  - Adaptable to different transformations
    - To reuse parts of the framework (e.g., model generators)
  - From Ed Willink's paper at BigMDE 2016 we infer that his work on QVT would also benefit from this
  - This is also a need of the BX (bidirectional transformation) community
    - With additional requirements (e.g., not EMF specific, Haskell-compatible)

Thanks for your attention!

Questions ?