Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Towards Optimisation of Model Queries: A Parallel Execution Approach

Sina Madani[a] Dimitris Kolovos[a] Richard F. Paige[ab]

a. Department of Computer Science, University of York, UK

b. Department of Computing and Software, McMaster University, Canada

Abstract The growing size of software models poses significant scalability challenges. Amongst these challenges is the execution time of queries and transformations. In many cases, model management programs are (or can be) expressed as chains and combinations of core fundamental operations. Most of these operations are pure functions, making them amenable to parallelisation, lazy evaluation and short-circuiting. In this paper we show how all three of these optimisations can be combined in the context of Epsilon: an OCL-inspired family of model management languages. We compare our solutions with both interpreted and compiled OCL as well as hand-written Java code. Our experiments show a significant improvement in the performance of queries, especially on large models.

Keywords Epsilon; Scalability; OCL, Query performance.

1 Introduction

Modern software systems are often required to process an ever-increasing volume of complex data with more stringent throughput and latency requirements. Although model-driven engineering helps to curtail the complexity of systems, the performance of many popular modelling tools leaves much to be desired. This is particularly problematic since larger projects are arguably likely to benefit the most from a model-driven approach. Scalability is a notable challenge with model-driven engineering [1], and a multi-faceted one too [2].

Model management workflows often involve a variety of tasks such as validation, comparison, model-to-model and model-to-text transformations. Despite their differences, these tasks typically use a common set of queries and transformations on collections of model elements. With very large models (in the order of millions of elements and gigabytes in size), these operations can incur a significant performance cost, making overall tasks slow and less productive. Furthermore, even the most complex queries on collections of data (model elements) can be expressed using a relatively small set of simpler operations. The Object Constraint Language (OCL) is one of the most well-known and frequently used languages for querying and validating models. As a functional and declarative language, OCL offers a useful set of operations on collections, including operations involving predicate logic.

In this paper, we demonstrate how declarative collection operations can be implemented in a data-parallel manner, with up to linear improvements in execution times with the number of cores. Given the diminishing generational improvements in single-thread performance due to physical and technical constraints, combined with the increasing number of cores in virtually all computing devices, the case for parallel processing of large data sets on modern computers is clear. We also show how lazy evaluation and short-circuiting can be combined with parallelism to provide further performance benefits. Unfortunately however, the OCL specification in its current form [3] is "far from perfect" [9], making desirable performance optimisations difficult without substantial effort from tool developers.

This work builds on [6], which introduces parallel variants of several collection querying operations with revised performance evaluation experiments and the addition of lazy execution, with an emphasis on the benefits relative to the current specification and implementations of OCL.

The remainder of the paper is organised as follows. Section 2 briefly outlines the limitations of OCL and motivations for this work. Section 3 demonstrates parallel execution algorithms for first-order operations. Section 4 proposes more advanced optimisations which combine parallel execution and lazy evaluation. Section 5 provides a brief overview of our testing methodology and performance metrics. Section 6 reviews pertinent work on optimising queries and iteration operations. Section 7 concludes the paper and suggests extensions for future developments.

2 Motivation and OCL limitations

OCL is a commonly used query and expression language in model management programs. Although it was designed to enrich metamodels with invariants (validation constraints) which can only be expressed programmatically, it is often used outside of pure model validation for consistency. Being an OMG standard, the language has a mature specification and history with convenient syntax and semantics for users and developers of modelling tools. Amongst its most desirable properties is the lack of side-effects and mutability. OCL is a functional language which enables reasoning and analysis. Functional languages are also inherently parallelisable, as we shall demonstrate shortly. However one controversial aspect of OCL is its exception-handling semantics. In object-oriented languages such as Java, a significant part of the language constructs and specification are dedicated to dealing with unexpected execution paths with various causes. However unless such exceptions are explicitly handled in the program, the default behaviour is to keep propagating the exception until eventually the program halts. By contrast, OCL deals with exceptions in a functional manner by having exceptions be *values* of expressions, just as *null* is a value for a reference in object-oriented languages. That is, if an exception occurs for whatever reason during execution of an expression, the result is "*Invalid*".

As noted by Willink [9], the specification unfortunately precludes some optimisations which would enhance the efficiency of collection-based operations. Most notably, this includes the inability to perform short-circuiting because all errors in OCL must be caught and propagated as *Invalid*. This means that if for example the last element in a collection is null and the predicate of a first-order operation dereferences the element, even though the last element may never be reached under short-circuiting evaluation the specification requires that all elements be evaluated / checked for validity. Furthermore, the specification forbids lazy evaluations of chained operations, and mutability. However as Willink claims, the underlying implementation need not be purposely wasteful so long as the observable end result (from the user's perspective) is consistent with the specification.

To demonstrate these shortcomings, let us consider a simple query. Suppose we have a transportation model of a city and we want to know whether there are any cars of a given brand registered after a particular year. One way to write this in OCL is shown in Listing 1.

```
1  Car.allInstances()
2      ->select(c | c.year > 2017)
3      ->collect(c | c.brand)
4      ->exists(b | b = 'BMW')
```

Listing 1 – Inefficient chained query

In a typical OCL execution engine, the following sequence of evaluations would occur: All instances of *Car* elements are retrieved from the model (line 1). The predicate of *select* is applied to each and every element, returning a new collection containing the subset of elements satisfying the criteria (line 2). The transformation function passed to *collect* is then applied for every element in the subset return by *select*, and the results are added to a new collection (line 3). Finally, every element of this new collection is iterated through with the predicate of *exists* being applied to it (line 4). If any elements satisfy this predicate, the result flag is set to true. However if execution of the predicate on any element fails, the result is set to *Invalid*. The result is returned once all elements have been evaluated.

This is a very inefficient algorithm for expressing this query, even though from the user's perspective it is intuitive to write. A much more optimal representation is shown in Listing 2, which not only requires a single iteration but also avoids creating intermediate collections as well as partial short-circuiting of the expression, since OCL permits short-circuiting of Boolean values and expressions.

```
Car.allInstances()->exists(c | c.year > 2017 and c.brand = 'BMW')
```

Listing 2 – Optimised query

Even though Listing 2 is significantly more efficient, it can be further optimised. Firstly, it is not necessary to retrieve all instances of the *Car* type into memory before evaluating the *exists*, but rather iterating through them as and when required. Secondly, if the OCL specification was less strict about propagation of *Invalid* then *exists* could stop executing once a match (i.e. an element which satisfies the predicate criteria) has been found. Finally, the execution of *exists* need not happen sequentially, as there are no dependencies between iterations and elements used in the operation.

In this paper, we demonstrate how it is possible not only to execute declarative operations on collection types in parallel, but also how to combine parallelism with laziness and short-circuiting. Furthermore, we show how it is possible to make the query in Listing 1 have the same order of complexity as the more optimised version in Listing 2 as a result of laziness and short-circuiting, whilst still taking advantage of multiple processor cores. However in order to do this, we must first free ourselves from the constraints imposed by the OCL specification. We therefore use the OCL-inspired

Epsilon Object Language (EOL) for our implementation. Given the similarities between OCL and EOL, we use EOL and OCL interchangeably in the rest of the paper. We use OCL syntax due to its familiarity (which is also valid in EOL).

3 Parallel Execution in Epsilon

Epsilon is an open-source Eclipse project[1] and family of languages which allows users to perform model management tasks on a wide variety of modelling formats and technologies. All task-specific languages build on top of a common interpreted OCL-like imperative language – the Epsilon Object Language (EOL) [4]. The language can be thought of as a more Java-like OCL, since it supports imperative constructs such as while loops, if-else, switch statements and mutable, dynamically typed variables. Users can define their own operations, including extending the functionality of existing types, and also work with Java types and invoke methods, since Epsilon is implemented in Java and uses reflection to execute expressions. Advanced features include the ability to cache the result of invoking user-defined operations to avoid re-computation, and extended properties which allow individual objects / model elements to have arbitrary additional values associated with them.

3.1 Concurrent execution engine

In our previous work [5], we identified a number of challenges in supporting multi-threaded execution in Epsilon. Unsurprisingly, the main difficulties come from shared mutable state due to e.g. global variables and advanced features such as cached operation and extended properties. To avoid creating a subset of Epsilon with limited support for imperative features, we tackled such issues directly whilst maintaining the behavioural semantics of the language (akin to the sequential implementation where possible). As a representative example, we describe our solution to variable declarations.

Variables are stored in a *FrameStack* and are disposed when they are no longer in scope. However when executing an expression from a different thread, declared variables should only be accessible by the executing thread. To avoid synchronization, we use thread-local frame stacks. A *ThreadLocal*[2] is a data structure used in serial thread confinement where every thread has its own copy of an object. Each thread-local framestack has a reference to the parent framestack; which is used by the main thread. This is so that if a global variable is referenced, then we can still access its value. However if the global variable reference is mutated, the results are non-deterministic. Thus users can write imperative code if desired so long as they do not mutate global state. Although mutability is not supported in OCL, variables are still declared in the body of operations and invariants, even if they are implicit (e.g. as iterator parameters).

Another thread-local data structure is the execution trace, which is mainly used for traceability purposes in the event of an exception. For instance, if the user tries to invoke a non-existent variable or operation, tries to navigate a null property or explicitly throws an error message, a Java-like stack trace is presented with the cause and exact line and column numbers where the error occurred along with the preceding calls. When such errors occur under multi-threaded execution, all threads stop execution and control returns to the main thread which then reports the stack trace.

[1] www.eclipse.org/epsilon/ [2] docs.oracle.com/javase/8/docs/api/java/lang/ThreadLocal.html

For executing jobs in parallel, we use an extension of *ThreadPoolExecutor*[3] with, by default, the number of threads being equal to the number of logical processors available to the JVM. The *ThreadPoolExecutor* has a queue which allows to submit jobs in the (form of code blocks, i.e. *Runnable* or *Callable*) and automatically maps jobs to threads. Since we rely on ThreadLocal data structures, we use a fixed size pool with infinite life span for the threads so that they are not disposed or recreated. We also use a custom ThreadFactory to name our threads for ease of identification and to define an exception handler for uncaught exceptions. The executor is re-used throughout the program for simplicity and efficiency, and is generally suitable for CPU-bound data processing. There is no (soft) limit to the number of jobs which can be enqueued for processing.

3.2 Parallel first-order operations

OCL offers declarative operations which provide a convenient way to express simple queries and transformations in a functional style using lambda expressions requiring little more than a predicate as a parameter which is then applied to each element. The operations are ideal candidates for concurrent execution without synchronization because they satisfy key properties: they do not mutate global state and apply the same, potentially expensive expression(s) over a (potentially) large source of data.

In our previous work [6] we demonstrated parallel execution algorithms for almost all declarative operations in OCL / Epsilon, including *select*, *reject*, *any*, *exists*, *forAll one*, *none*, *collect*, *mapBy* and *sortBy*. In all cases, we were able to achieve a level of parallelism directly proportional to the number of elements by applying the function expression (in most cases, a predicate) over each element independently. This data-parallel approach is implemented by creating a job for each model element and submitting it to a custom thread pool executor. A job is a function which evaluates the user-defined lambda expression over a given element. If we take the *collect* operation (which transforms each element to a different type, e.g. by deriving a property from it) as a representative example, the parallel implementation can be described by Listing *parallelCollect*.

```
1  <IN, OUT> Collection<OUT> collect(Collection<IN> source,
2        Function<IN, OUT> mapper) throws EolRuntimeException {
3
4     var resultsCol = new ArrayList<OUT>(source.size());
5     var jobFutures = new ArrayList<Future<Object>>(source.size());
6     EolExecutorService executor = context.getExecutor();
7     for (IN element : source) {
8        jobFutures.add(executor.submit(() -> mapper.apply(element)));
9     }
10    for (Future<OUT> futureRes : jobFutures) {
11       resultsCol.add(futureRes.get());
12    }
13    return resultsCol;
14 }
```

Listing 3 – Simplified *parallelCollect* execution algorithm

Note that we submit the jobs in encounter order (lines 7–9). Upon submission, a *Future*[4] (also known as a *Promise*) is immediately returned, which is a wrapper of the

[3] docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html
[4] docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html

job's result. Once all jobs have been submitted, we then loop through all of the Futures (lines 10–12) and attempt to retrieve their values. Since we must eventually wait for all results, it does not matter when we block – whenever a result is available, it will be published to the Future and be immediately retrievable. Moreover, by gathering the results in order of submission and adding them to the resulting collection (line 11), we can guarantee ordering which is equivalent to the sequential implementation, even though the computations completed in a non-deterministic order.

Non-short-circuiting operations such as *collect* and *select* are significantly simpler to implement than short-circuiting ones in parallel. To illustrate, let us take the example of *any*, which will return any item in the source collection which matches the given predicate. A sequential implementation is relatively straightforward, as shown by Listing 4.

```
<T> T any(Collection<T> source, Predicate<T> predicate) {
   for (T element : source)
      if (predicate.test(element))
         return element;
   return null;
}
```

Listing 4 – Simplified sequential *any* execution algorithm

In a multi-threaded execution environment, we do not have the luxury of simply returning an element once it has been found without manually stopping other threads to prevent unnecessary computation. The fundamental challenge is that after all jobs have been submitted, we do not know which will finish first and so it is the responsibility of each job to notify the main thread when the result has been found. This also means we have to block the main thread waiting for a condition to be signalled and then retrieve the result object. Further complicating matters are exceptions which may occur during execution of any jobs, which also requires stopping all computations and reporting the cause.

Our solution uses an *ExecutionStatus* object as the centrepiece for coordinating short-circuited concurrent tasks. The idea is that we start a separate thread to wait for all jobs to complete by attempting to retrieve values from the Futures, just as we saw with *parallelCollect*. The main thread then waits on the *ExecutionStatus*, waiting for it to be notified. The notification can come from either a job in which a result has been found, or from the waiting thread once all jobs have completed. If a job signals completion, it does so by setting the result object property on the *ExecutionStatus*. When the main thread is awoken and a result is present, it then loops through all submitted jobs (Futures) and attempts to cancel them. The result is then returned. A similar strategy is followed in the event of an exception, though instead of setting the result property on the *ExecutionStatus*, we set the exception instead which can then be retrieved and reported by the main thread.

A simplified implementation of *parallelAny* is shown in Listing 5. However unlike with non-short-circuiting operations, we cannot guarantee a consistent result, although the OCL 2.4 specification does not require *any* to be deterministic. Whereas the sequential implementation would always return the first element which matches the predicate criteria, the parallel implementation may return any.

```
<T> T any(Collection<T> source, Predicate<T> predicate) {
   EolExecutorService executor = context.getExecutor();
   var jobs = new ArrayList<Future<Optional<T>>>(source.size());
```

```
 4     for (T element : source) {
 5        Runnable job = () -> { try {
 6           if (predicate.test(element)) {
 7              executor.getExecutionStatus().completeWithResult(element);
 8           }
 9        } catch (Exception ex) {
10           executor.getExecutionStatus().completeExceptionally(ex);
11        }}
12        jobs.add(executor.submit(job));
13     }
14     return executor.completeShortCircuit(jobs);
15  }
```

Listing 5 – Simplified *parallelAny* execution algorithm

A basic implementation of short-circuiting asynchronous jobs is shown in Listing 6. The main idea is to have one thread wait for the jobs to complete (and stop waiting once a terminal condition, such as a result or exception, is met) whilst the main thread blocks, waiting for the terminal condition (i.e. for all jobs to finish, or an early result, or an exception). An early result (short-circuiting) is signalled from one of the jobs, e.g. in line 7 of Lisitng 5.

```
<T> T completeShortCircuit(Collection<Future<T>> jobs) throws Exception {
   if (jobs.isEmpty())  return null;
   ConcurrentExecutionStatus status = getExecutionStatus();

   Thread compWait = new Thread(() -> {
      try {
         for (Future<T> future : jobs) {
            if (status.isInProgress())
               future.get();
            else return;
         }
         status.completeSuccessfully();
      }
      catch (ExecutionException ex) {
         status.completeExceptionally(ex);
      }
      catch (CancellationException | InterruptedException ice) {
         // This means we finished early (short-circuit)
      }
      assert !status.isInProgress();
   });
   compWait.start();

   boolean success = status.waitForCompletion();
   compWait.interrupt();

   if (!success) {
      shutdownNow();
      throw status.getException();
   }
   else for (Future<T> future : jobs) {
      future.cancel(true);
   }
```

```

    return status.getResult();
}
```

Listing 6 – Simplified short-circuiting algorithm in *EolExecutorService*

3.2.1 Delegation

The remaining predicate first-order operations can be implemented by delegation. For example *exists* delegates to *any*, *forAll* can be expressed as a negated *exists* with a negated predicate (proof by contradiction), *reject* as *select* with a negated predicate. The *sortBy* operation is an extension of *collect* with a decorator pattern applied for sorting properties, where comparison is normalised to binary operations between integers. This latter part can be trivially parallelised.

3.3 Nested Parallelism

When we initially designed the concurrent engine for Epsilon, we used ThreadLocals under the assumption that a given thread would only execute a single job at a time. Although this seems like a reasonable approach which simplifies programming and provides good performance, it does come with the limitation that there can be no nested parallelism – that is, parallel jobs being spawned inside other parallel jobs. This is because a ThreadLocal's data is, by definition, bound to a particular thread. When there is a single execution context and a single thread pool (say with n threads), this presents no issues. However suppose that a parallel job is spawned from within another parallel job. This then requires n^2 number of ThreadLocals (and of course, threads), each with appropriate scoping. Note that the same threads cannot be re-used because then each ThreadLocal would contain data (stack trace, variables etc.) from different scopes of execution, leading to non-deterministic behaviour. Performance would also be heavily reduced as there would be too many threads created, which can slow down the system or at worse result in the JVM crashing. Memory usage would also increase exponentially. Although it is possible to partially circumvent this by using "job-local" as opposed to thread-local data structures, the book-keeping overhead, complexity and loss of performance and stability means on balance it makes much more sense to disallow nested parallelism at any level.

It is for this reason, as well as the lack of guarantees about the user's code being side-effect-free, that we cannot simply replace sequential first-order operations with their parallel variants. Instead, we have three variants for each operation. Consider *select* as an example:

- *parallelSelect* – uses the parallel implementation
- *sequentialSelect* – uses the sequential implementation
- *select* – delegates to *parallelSelect* if it is not being invoked from a parallel operation and to *sequentialSelect* otherwise.

Therefore the default automatically chooses the appropriate one. However the user can also explicitly specify which implementation they want if they desire. Explicit nested parallelism will throw an exception. Detecting nested parallelism can be done in a number of ways (for example, checking the name of the current Thread) but we found the most reliable approach is to have a flag in the execution context when a parallel task is started.

4 Optimised Operations

We have shown how commonly used operations on collections can be parallelised due to their inherently functional nature. Less obvious is how such operations can be further optimised not by increasing throughput / evaluations per second, but avoiding unnecessary evaluations in the first place. More advanced optimisations require knowledge of the program's intent and the context in which the operations are applied, rather than studying each operation in isolation. Inevitably this requires static analysis which although is common – even extremely advanced – in development environments such as IntelliJ IDEA and Visual Studio for mainstream programming languages, cannot be directly reused in more specialised cases such as Epsilon's model management languages. The fact that such task-specific languages are bespoke and interpreted means they require purpose-built static analysis frameworks.

4.1 *count*

Take for example a query of the form *collection–>select(predicate)–>size()*. Clearly this cannot be short-circuited or evaluated lazily with any benefit, however it can be optimised. Notice that immediately after *size()* returns, the collection which has been built is thrown away. The evaluation of select essentially spent much time and memory building a collection consisting of a subset of the original elements, only to discover how many elements satisfy a given criteria. Instead, we propose a *count* operation which does not require building an intermediate collection. Even though this requires a mutable counter which is incremented when a matching element is found, a lock-free parallel implementation is possible by using an *AtomicInteger*[5], which can perform compare-and-swap (CAS) operations atomically using direct memory access. Since no caching occurs, there is no need for synchronization and updates are immediately visible to all threads. Listing 7 shows a parallel implementation for this operation.

```
<T> Integer count(Collection<T> source, Predicate<T> predicate) {
   EolExecutorService executor = context.getExecutor();
   AtomicInteger result = new AtomicInteger(0);
   for (T element : source) {
      executor.execute(() -> {
         if (predicate.test(element)) {
            result.incrementAndGet();
         }
      });
   }
   executor.awaitCompletion();
   return result.get();
}
```

Listing 7 – Simplified algorithm for *parallelCount* operation

4.2 *nMatch*

In some cases non-short-circuiting operations such as *select* can be expressed as short-circuiting ones by looking just one or two expressions ahead. To demonstrate this, we devised an *nMatch* operation which, in addition to a predicate as the filtering criteria

[5] docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html

also takes an integer parameter n, and returns a Boolean. There are three possible semantics for this operation where the result is true (and false otherwise): if at least, at most or exactly n elements satisfy the predicate. This operation can therefore be used to rewrite expressions of the form *collection–>select(predicate)–>size() = n* as *collection–>nMatch(predicate, n)*.

Similarly, for checking whether at least n elements meet the criteria, *collection–>select(predicate)–>size() >= n* can be re-written as *collection–>atLeastNMatch(predicate, n)*. These semantics enable the operation to be used as a delegate implementation for *any* with $n >= 0$ and *forAll* with *n = source–>size()*. To see why this is short-circuiting, consider the logic in Listing 8:

```
1  boolean shouldShortCircuit(int sourceSize, int targetMatches,
2        int currentMatches, int currentIndex, MatchMode mode) {
3     if (
4        (mode != MAXIMUM && sourceSize < targetMatches) ||
5        (currentMatches > targetMatches) ||
6        (mode == MINIMUM && currentMatches == targetMatches) ||
7        (sourceSize - currentIndex) < (targetMatches - currentMatches)
8     ) return true;
9     else return false;
10 }
11 boolean determineResult(int currentMatches,
12       int targetMatches, MatchMode mode) {
13    switch (mode) {
14       case EXACT: return currentMatches == targetMatches;
15       case MINIMUM: return currentMatches >= targetMatches;
16       case MAXIMUM: return currentMatches <= targetMatches;
17       default: return false;
18    }
19 }
```

Listing 8 – Short-circuiting logic for *nMatch* operation

The first condition (line 4) is checked before beginning execution since if the number of elements is less than the desired number of matches, the result is automatically false. The second condition (line 5) checks whether we have exceeded the required number of matches which would result in the operation returning false if the mode is to match exactly or at most n elements. The third condition (line 6) is to short-circuit in the event that the minimum number of matches has been met if that is the desired semantics. The last condition (line 7) is a comparison between the number of remaining elements and the number of remaining matches. This is to ensure that subsequent evaluations can still change the result. For example, if n=10 and so far we have matched 5 elements but there are only 4 elements remaining, then even if the predicate is satisfied for all remaining elements the result would still be false, since $(5+4=9) < 10$.

The *nMatch* operation is slightly easier to parallelise than *any* because we do not need a container for the resulting element. However we do need two mutable counters: one for the number of elements evaluated and another for the number of elements for which the predicate is satisfied. Again we can use *AtomicInteger* to achieve this, as shown in Listing 9.

```
<T> T nMatch(Collection<T> source, Predicate<T> predicate,
      int n, MatchMode mode) throws Exception {
```

```

    EolExecutorService executor = context.getExecutor();
    AtomicInteger currentMatches = new AtomicInteger(0);
    AtomicInteger evaluated = new AtomicInteger(0);
    int ssize = source.size();
    var jobs = new ArrayList<Future<Void>(ssize);
    for (T element : source) {
       jobs.add(executor.submit(() -> {
          int c = predicate.test(element) ?
             currentMatches.incrementAndGet() :
             currentMatches.get();
          int e = evaluated.incrementAndGet();
          if (shouldShortCircuit(ssize, n, c, e)) {
             executor.getExecutionStatus().completeSuccessfully();
          }
          return null;
       }));
    }
    executor.completeShortCircuit(jobs);
    return determineResult(currentMatches.get(), n);
}
```

Listing 9 – Simplified *parallelNMatch* execution algorithm

4.3 Parallel and Lazy evaluation with Streams

It is possible to effectively eliminate unnecessary expression evaluations without static analysis when queries and transformations are chained. Take the query in Listing 1 as an example. Although the query in Listing 2 is more efficient and concise, it is easy to imagine that in more complex cases such a refactoring can become less readable and unnatural to write. Furthermore, chained queries may be easier to debug by, for example, adding print statements to the end of each expression[6].

Fortunately there is an elegant solution to optimising such queries by treating the data source as a *stream* – an abstract pipeline of computation. Instead of evaluating the operations eagerly at every stage in the pipeline before moving on to the next, a Stream would instead fuse the operations and compose a single query, so that each element is evaluated across the entire pipeline before moving on to the next element. In the Listing 1 example, the first element would go through select, if it matches the predicate it then gets passed to collect, and finally to exists, where if the predicate is satisfied, then the computation terminates and no further evaluations occur.

Even more fortunate is that such a processing model was introduced as a major feature of Java 8. Conveniently, since Epsilon is written in Java and uses reflection, it is possible to leverage such capabilities directly from Epsilon code. Although it is possible to obtain a Stream from a Collection without additional work, the main challenge with supporting Streams in EOL is transforming user-defined lambda expressions into compatible Java functional interfaces to be passed as parameters to stream operations. This essentially requires the engine to discover the required interface parameter(s) for the invoked operation and implement the interface(s) at runtime with the user-supplied EOL expression. This is achieved through the use of a Proxy[7].

[6] In Epsilon, calling *.println()* on any object returns the object for convenience
[7] docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html

One notable challenge with this approach is that the functional interfaces in Java do not allow throwing checked exceptions, so to maintain an easily traceable error reporting stack trace, we extended these interfaces to allow for throwing checked exceptions. Of course there is no way to force the Streams API to invoke the method which throws a checked exception, so the default implementation calls the checked variant and wraps the checked exception as an unchecked one and throws it, where the root cause can then be later retrieved.

```
Car.allInstances()->parallelStream()
   ->filter(c | c.year > 2017)
   ->map(c | c.brand)
   ->anyMatch(b | b = 'BMW');
```

Listing 10 – Lazy and parallel chained query

The query in Listing 1 can be expressed using Streams as shown in Listing 10, and is computationally equivalent to the query in Listing 2 (ignoring the overhead of invoking streams from Epsilon). Furthermore, it is trivial to parallelise by invoking the *parallel()* method on the Stream, which then uses the common Fork-Join thread pool, using as many threads as there are cores available to the JVM and a work-stealing scheduling algorithm. Since we have already addressed concurrency issues which may arise, there are no further issues faced by executing in parallel streams compared to our bespoke parallel first-order operations. It is possible to altogether replace collections with streams and only materialise them where required, and to map the stream operations to their OCL-named equivalents, however for compatibility purposes we do not do this in Epsilon. However an OCL implementation could use Streams internally and map the existing operations to those provided by the Stream API.

Perhaps the most notable benefit of supporting Streams for direct use in Epsilon programs is that they need not necessarily operate on a finite collection. All streams are derived from a *Spliterator*[8], which is essentially an iterator that can be recursively divided into other iterators and reports various characteristics such as immutability, ordering, distinctness and whether the number of elements is known. Since Epsilon by design is not tied to any modelling technology and its languages are decoupled from the underlying model implementations, one could develop a model driver which returns a lazy collection when invoking *allInstances()* or *getAllOfKind()*. A notable example is the JDBC driver for Epsilon [8].

The reason for maintaining a bespoke implementation of query operations as opposed to using streams under the hood is to maintain compatibility with user's expectations. This includes eager evaluation and also guarantees ordering of results in the case of *parallelSelect* and *parallelCollect*. A custom implementation is also more flexible and extensible, and allows us to support operations which are not available in the Streams API (such as *nMatch*).

5 Evaluation

In this section, we evaluate our parallel solutions for both correctness and performance. Resources for our experiments can be found on GitHub[9].

[8] docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html [9] github.com/epsilonlabs/parallel-erl

5.1 Correctness

Epsilon has a thorough test suite consisting of thousands of automated tests. We extended these with a thorough series of tests for each first-order operation, including equivalence testing between parallel and sequential variants. To perform the bulk of our tests in a declarative manner, we used EUnit [11] – a JUnit-style testing framework for Epsilon. This allows us to declare each test as an operation and perform assertions on the results obtained from applying a given first-order operation on some test data. We try to test all reasonable boundaries where it makes sense, and achieved 100% code coverage for all of the operations (both sequential and parallel variants). We also perform more advanced tests for the parallel variants such as testing the scoping of global variables and operations, nested operations and ensuring nested parallelism can be detected as well as exception handling semantics. We also test for equivalence between parallel streams and our bespoke parallel first-order operations, and also test the Stream (and by extension, the conversion from Epsilon lambdas to Java functional interfaces) functionality independently with EUnit tests.

5.2 Performance

Due to the large combination of factors which constitute a benchmark (model size, query / script, implementation strategy, number of threads etc.) we mostly try to demonstrate the efficiency of various implementations of the same query. The query is quite complex and uses a mixture of short-circuiting and non-short-circuiting operations, however our benchmark is focused on evaluating the performance of the outer-most operation, which is the *select* operation. We chose this because it is one of the most widely-used operations, and to make the comparison with OCL fair since it is not short-circuiting. We compare many implementation which all produce the same output. These include parallel and sequential variants of EOL's *select*, the Stream-equivalent *filter* (both sequential and parallel variants), compiled and interpreted OCL. Since our query is of the form *select(...)–>size()*, we also benchmark the *count* and *parallelCount* variants. To assess the overhead of using Java Streams from Epsilon, we also wrote the benchmark query in Java using streams (both parallel and sequential). It should be noted that the Stream version in both Java and EOL uses the more optimal *count* terminal operation as opposed to *collect(Collectors.toList()).size()*. For our OCL implementation, we define the query as an operation and invoke it via the Eclipse OCL Pivot API.

The context for our query is the Internet Movie Database (IMDb) metamodel which basically consists of two top-level model element types: Movies and Actors, where each Movie has a reference to its Actors and similarly each Actor has a reference to all the Movies they have starred in. We used models ranging in size from 100 000 elements to over 3.53 million elements. The query attempts to find the number of actors whose co-actors have featured in at least three of the same movies.

For our experiments, we used the latest interim version of Epsilon, Eclipse Modelling Framework 2.15 and Eclipse OCL 6.7.0. We ran each experiment five times in separate JVM invocations, and took the mean average time in milliseconds. As expected, there were no outliers in any of the runs. We exclude the time taken for parsing the model, which on average took approximately 40 seconds for the largest model and 26 seconds for 2.5 million elements. For our parallel variants, we used as many threads as logical cores in the system. The test environment for our experiments was as follows:

AMD Ryzen Threadripper 1950X @ 3.6 GHz (in "Creator Mode") 16-core / 32-thread CPU, 32(4x8) GB DDR4-3003 MHz RAM, Fedora 29 OS (Linux kernel 4.20), OpenJDK 11.0.2, JVM options: "*-XX:MaxGCPauseMillis=730 -XX:+UseNUMA -XX:MaxRAMPercentage=90 -Xms768m*"

Table 1 – Benchmark results for query with 2.5 million model elements

Implementation	**Exec. Time (ms)**	**Speedup**
Interpreted OCL	4 296 853	–
Compiled OCL	4 309 659	0.997
Sequential EOL (select)	2 556 609	1.681
Parallel EOL (parallelSelect)	203 094	21.16
Sequential EOL (count)	2 577 140	1.667
Parallel EOL (parallelCount)	205 732	20.89
Sequential EOL (filter)	3 349 400	1.283
Parallel EOL (parallelFilter)	293 619	14.63
Java (filter)	133 350	32.22
Java (parallelFilter)	17 677	243.1

Table 2 – Thread scalability for parallel EOL *parallelSelect* query with 2.5 million model elements

Threads	**Exec. Time (ms)**	**Speedup**	**Efficiency**
1	2 507 820	–	1.00
2	1 305 142	1.921	0.96
4	679 137	3.693	0.92
8	389 400	6.440	0.81
16	223 433	11.22	0.7
32	203 094	12.35	0.39

Table 3 – Model scalability for parallel EOL with 32 threads *parallelSelect* compared to interpreted OCL

Model Elements	**Exec. Time (ms)**	**Speedup**
100K	9 589	16.71
500K	40 964	18.22
1M	84 285	18.87
2M	165 650	20.19
3.53M	286 408	30.38

Table 1 compares the relative performance of EOL, OCL and Java Streams for a model with 2.5 million elements (approximately 221 MB in size)[10]. There are a number of surprising results on display. Firstly, Epsilon outperforms OCL by a significant margin even without multi-threading. This makes the gains from parallelism look more spectacular than the reality, since if we compare the speedup of parallel EOL to sequential EOL, we would observe a performance increase of 12.6x as opposed to 20.9x when comparing to interpreted OCL. Secondly, it is unexpected that compiled OCL performs slightly worse than interpreted. However we should note that where interpreted OCL uses a query defined in a "CompleteOCL" document,

[10] Speedup is relative to Interpreted OCL

Table 4 – Model scalability for parallel EOL with 4 threads *parallelSelect* compared to sequential EOL

Model Elements	**Exec. Time (ms)**	**Speedup**
100K	26 663	4.459
500K	135 721	3.566
1M	279 093	4.358
2M	575 720	4.320
3.53M	1 000 528	3.754

the compiled version uses a genmodel where the query is embedded in the metamodel as “OCLinEcore”. Nevertheless, we expected compiled OCL to be significantly faster than Epsilon, but this is not the case and requires investigation that is beyond the scope of this paper. We also see that in this instance, there are no performance benefits from using the *count* operation, at least with this configuration. Furthermore, the overhead of using Java Streams in EOL sees the *count* operation outperforming its Stream equivalent by 1.3x.

Unsurprisingly, a hand-written version of the query using Java Streams in native Java outperforms all other implementations, though by an unexpectedly huge margin. In particular, we see that parallel stream is over 243x faster than interpreted OCL, though it is only 7.54 times faster than sequential stream. However it is worth noting that parallel stream in EOL is 11.45x faster than sequential stream. We see that the overhead of using Streams in EOL relative to Java is quite large: over 25x in the sequential case though only 16.1x in parallel. Perhaps the overhead of interpreting Epsilon AST scales well in parallel whereas in pure Java this overhead is not present, leaving less potential performance gains on the table.

Table 2 demonstrates how well our parallel variant of the *select* operation scales with more threads. We see a consistent drop-off of 11% in efficiency (that is, the speedup divided by number of threads) when moving from 4 to 8 and 8 to 16 threads. The large decrease from 16 to 32 threads can be explained by the lack of physical cores, indicating that the workload is indeed CPU-intensive. Our experience with using Hyper-Threaded Intel processors is that more significant gains are achievable with simultaneous multi-threading, however it is most likely a symptom of memory access / bandwidth bottleneck. It’s also interesting to note that the overhead for using the parallel operation and concurrent execution engine with only a single thread is very low, given the parallel implementation provides 98% of the performance of the sequential variant.

Table 3 shows the performance delta between our parallel *select* query in EOL and its equivalent in interpreted OCL across a range of model sizes from 100 000 elements to over 3.5 million. We see that there is quite a drastic difference in speedup between the largest and smallest model and, as expected, the performance improvements are amplified with larger models. However even in the smallest case we observe a 16.7x performance improvement. Even if we account for Epsilon’s inherent speed advantage compared to OCL in this particular benchmark, the gains are still significant.

Although we did not measure memory usage programmatically due to inconsistencies caused by garbage collection, we did observe that compiled OCL consumed significantly more memory than Epsilon. Perhaps this is also related to lack of improvements over interpreted OCL. When benchmarking compiled OCL with the largest model in our sample, we consistently encountered *OutOfMemoryError* due to insufficient heap space, despite the VM having access to over 28 GB memory. This is especially alarming considering that the model is "only" 329 MB in its serialized form.

Table 4 compares the scalability of parallel EOL with 4 threads to sequential EOL for the *select* query. Unexpectedly, we observe better-than-linear performance improvement three of the models. It is difficult to interpret any fundamental reason for this based on the implementation algorithm and the query, however the remarkable consistency in speedup for the 1 million and 2 million element models shows this is not an anomaly, at least within our experiments. The complex architecture of our CPU could partially explain such gains due to its cache structure and the fact that it's essentially four quad-core modules in a single chip, however in previous experiments we have also observed better-than-linear speedups when using 4 threads with a different CPU. The results in Table 2 and Table 4 seem to imply that four threads provides the peak balance between speedup and efficiency.

Overall, the results suggest that our parallel implementation of the *select* operation scales with the number of cores and that the performance gains are amplified by increasing model size. We also see that Epsilon significantly outperforms OCL even without parallelisation. This can probably be explained by the lack of short-circuiting operations in OCL, since we make use of these in our query. When combining the efficiency from short-circuited evaluation and parallelisation, the benefits become clear as shown by the last row in Table 3: a 30-fold reduction in execution time. This is especially relevant when the absolute execution times are in the order of hours, not seconds. In this example, a query which took over 2 hours and 25 minutes with OCL is reduced to 4 minutes and 46 seconds. This is without even considering the gains from a non-interpreted implementation, which can potentially further reduce the execution time from minutes to seconds. The large gap in performance between interpreted OCL and an equivalent hand-written Java query perhaps gives an indication of the potential scope for improvements.

6 Related work

Optimisation of model management programs – particularly model queries and collection operations – is an active research topic which has received increasing attention in recent years. Generally, solutions can be categorised as either being incremental, lazy, parallel or a combination. Although incrementality has traditionally received the most attention in the modelling community (especially in model-to-model transformations), more researchers are turning to lazy evaluation since, as with incrementality, avoids unnecessary evaluations without the complex book-keeping and memory overhead of partial execution. There is also an increasing trend in translating queries written in modelling languages such as OCL into the native language of the underlying data source. However parallel execution is still relatively novel and the least explored optimisation solution in the modelling community, despite the mature support and libraries for concurrent and parallel programming in the languages used to implement most modelling tools.

6.1 Parallel streams

The Java standard library provides streams [10], which are an abstract processing pipeline over a fixed or infinite data stream, as discussed in Section 4.2. Streams can also execute in parallel, though the output is unordered in that case. Parallel streams internally use a divide-and-conquer approach, delegating Java's fork-join processing framework. The key to making this possible is the ability to split the data source, and perhaps more fundamentally, assuming that none of the operations have side-effects or rely on mutable global state.

Furthermore, the iterator-based nature of streams means that the entire operation chain can be evaluated on individual elements, enabling lazy evaluation [12]. That is, instead of requiring the intermediate results of e.g. a filter (*select* in OCL) operation be pooled into a collection only to be filtered again, the operations themselves can be fused to provide short-circuiting behaviour.

6.2 LINQ and Task Parallel Library

The .NET platform offers a similar declarative style of data processing to Java Streams with its Language-INtegrated Query (LINQ) [13]. This provides an SQL-like querying syntax built directly into a language such as C#, allowing developers to express queries in a unified and object-oriented manner irrespective of the data model without resorting to languages which are specific to the modelling technology (e.g. XQuery for XML or SQL for relational databases). There is also a parallel execution engine for LINQ (PLINQ), which can be enabled in a similar manner to parallel Streams in Java. Parallelism is centred around the *ParallelEnumerable* class. .NET also offers the Task Parallel Library [14], which can be used to execute *for* loops in a data-parallel manner. Since all first-order collection operations used a *for* loop, this library could be used to implement parallel variants provided that there are no side-effects (OCL would be a good candidate, for example).

6.3 Lazy OCL

On the topic of lazy evaluation of expressions on collections, Tisi et al. (2015) propose an iterator-based approach in [16]. The basic premise of their work is to treat OCL collections in a similar manner to Java Streams, such that operations on collections are evaluated only when required by a subsequent computation. This is achieved by returning an iterator which evaluates the desired expression on each element when it is iterated over; which may be in a chain of other operations. The authors also apply this lazy approach to the *allInstances()* operation, which retrieves all elements of the target type. Since this is often the source collection on which further operations are invoked, it enables a lazy evaluation strategy to be applied in the entire chain of computation rather than only the intermediate operations.

Willink (2017) [9] proposes a novel way to implement OCL collections which overcomes the limitations associated with an intuitive, one-to-one implementation of OCL collections using Java types. He shows that although OCL collections require inefficient properties such as immutability, eager evaluation and lack of short-circuiting, it is possible to get around these by using a custom data structure consisting of a HashMap and ArrayList to represent all four collection types with improvement in execution time an memory consumption in some cases. He also notes that other optimisations, such as element selection, can be performed in constant time thanks to the use of a HashMap as opposed to linear time which is the norm for the usual traversal algorithm.

6.4 Formal parallelism

In [17], Vajk et al. (2011) take a more formal approach to parallel execution of OCL expressions. Using the well-established Communicating Sequential Processes (CSP) model of concurrency, the authors provide a mapping between OCL expressions and CSP processes, focusing on binary expressions for task parallelism and iterators for data parallelism. The CSP is then compiled to C# code. However they do not provide a complete library of parallel operations, instead relying on the most general first-order operation – *iterate* – to implement and evaluate their approach.

6.5 Suboptimal Code Detection

Wei and Kolovos (2014) [7] present a model-based static analysis framework for Epsilon. They build on this framework with a pattern matcher that is able to detect computationally expensive EOL expressions which can be refactored to be more efficient, with recommendations to the user. Examples of suboptimal code include unnecessary calls to *allInstances()* when reverse navigation is possible, chained *select* (where the expressions can be combined with logical AND), *select* on Sets (which can be short-circuited with *any*), and replacing *select(...)–>size() > 0* with *exists*. Such capabilities can be used to also detect uses of *select(...)–>size()* where our *count(...)* operation could be used, or *select(...)–>size()* followed by an integer expression with *nMatch*.

6.6 Efficient database queries

The work of Kolovos et al (2013) [8] on supporting SQL databases as a modelling technology in Epsilon demonstrates the importance of optimisations not just in the implementation of operations in isolation, but also in how data is retrieved from the source. The JDBC driver for Epsilon transforms queries on collections of model element types (including *allInstances()*) into SQL queries which are lazily evaluated. Each operation internally builds an SQL query and returns a lazy collection, allowing for further operations to be composed before executing the query. Unlike the Java Streams approach however, the returned results can be materialised by invoking a specific method. Daniel et al (2018) [15] present the Mogwaï tool, which transforms OCL query expressions into Gremlin; a generic NoSQL database querying language. This shifts the burden of executing the query to the underlying database technology, which can inevitably perform more optimisations since the engine and query expression are specified at the appropriate level rather than being limited by the semantics of the OCL specification, for example. Ultimately, these works show that optimising around the data source is vital for efficient queries, both in temporal and spatial costs.

6.7 Incrementality

Jouault and Beaudoux (2015) [18] devised a bidirectional incremental implementation for OCL operations which avoids unnecessary re-computations of OCL expressions. The premise is that given a source collection and a result (possibly another collection), it is possible to propagate changes in either direction in a manner which only applies expressions of intermediate operations on the changed elements. This is achieved using "*boxes*" (containers with references to immutable values) and listeners. However this complicates the use of operations since there are many variants with various semantics for change propagation.

7 Conclusions and future work

In this paper, we have demonstrated several optimisations which can be made to the evaluation of OCL collection operations. We have shown that if the specification is revised, it is possible to combine lazy evaluation, short-circuiting and parallel execution to improve performance as well as to avoid penalising users for suboptimal expressions of queries. We have shown that non-short-circuiting operations can be executed in parallel whilst maintaining ordering, and that short-circuiting operations can be executed in parallel without full evaluation whilst still maintaining exception handling semantics. The performance gains in terms of execution times from parallel execution and short-circuiting alone are drastic: we observed a 30-fold speedup on a 16-core machine when comparing our solution to interpreted OCL for a large model.

Future work could build on these optimisations by leveraging advanced static analysis capabilities to replace suboptimal expressions. Our implementation requires that users explicitly invoke lazy and/or parallel variants of first-order operations. With sufficiently advanced static analysis, it would be possible to automatically detect cases where parallelism and laziness can be applied and perform the substitution transparently. Finally, to reduce memory consumption and unnecessary loading of models in memory, iterator-based model drivers are needed. Implementations should consider interfacing with models by using Spliterators which can be used by the Stream API to efficiently chain parallel and lazy operations.

References

[1] Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Scalability: The Holy Grail of Model Driven Engineering. In: Proceedings of the First International Workshop on Challenges in Model Driven Software Engineering, Toulouse, pp. 10-14 (2008)

[2] Kolovos, D.S., Rose, L.M., Matragkas, N., Paige, R.F., Guerra, E., Cuadrado, J.S., De Lara, J., Ràth, I., Varrò, D., Tisi, M., Cabot, J.: A research roadmap towards achieving scalability in model driven engineering. In: Proceedings of the Workshop on Scalability in Model Driven Engineering, Budapest. Article No. 2 (2013)

[3] Object Constraint Language 2.4 Specification, https://www.omg.org/spec/OCL/2.4/

[4] Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Object Language (EOL). In: Proceedings of the Second European Conference on Model Driven Architecture – Foundations and Applications, Bilbao. pp. 128–142 (2006)

[5] Madani, S., Kolovos, D.S., Paige, R.F.: Parallel Model Validation with Epsilon. In: Proceedings of the 14th European Conference on Modelling Foundations and Applications, Toulouse. pp. 115–131. Springer, Cham (2018)

[6] Madani, S., Kolovos, D.S., Paige, R.F.: Parallel Execution of First-Order Operations. In: Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, Copenhagen. pp. 132–145. ACM, (2018)

[7] Wei, R., Kolovos, D.S.: Automated Analysis, Validation and Suboptimal Code Detection in Model Management Programs. In: Proceedings of the 2nd BigMDE Workshop, York. pp. 48–57. Springer, Cham (2014)

[8] Kolovos, D.S., Wei, R., Barmpis, K.: Towards Scalable Querying of Large-Scale Models. In: Proc. 2nd Extreme Modeling Workshop, ACM/IEEE 16th International Conference on Model Driven Engineering Languages & Systems, Miami. ACM, (2013)

[9] Willink, E.D.: Deterministic Lazy Mutable OCL Collections. In: STAF 2017 Collocated Workshops, Marburg. pp. 340–355. Springer, (2017)

[10] Java Streams API, https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

[11] García-Domínguez, A., Kolovos, D.S., Rose, L.M., Paige, R.F., Medina-Bulo, I.: EUnit: a unit testing framework for model management tasks. In: Proceedings of the 14th international conference on Model driven engineering languages and systems, Wellington. pp. 395–409. Springer Berlin, Heidelberg (2011)

[12] Subramaniam, V.: Lets Get Lazy: Explore the Real Power of Streams, Devoxx United States (2017). Available at: https://youtu.be/F73kB4XZQ4I

[13] Language-Integrated Query (LINQ), https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/introduction-to-linq-queries

[14] .NET Task Parallel Library, https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/data-parallelism-task-parallel-library

[15] Daniel, G., Sunyè, G., Cabot, J.: Scalable Queries and Model Transformations with the Mogwaï Tool. In: Proceedings of the 11th International Conference on Theory and Practice of Model Transformation, Toulouse. pp. 175–183. Springer, Cham (2018).

[16] Tisi, M., Douence, R., Wagelaar, D.: Lazy evaluation for OCL. In: Proceedings of the 15th International Workshop on OCL and Textual Modeling, Ottawa. pp. 46–61. Springer (2015)

[17] Vajk, T., Dávid, Z., Asztalos, M., Mezei, G., Levendovszky, T.: Runtime model validation with parallel object constraint language. In: Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation, Wellington. Article No. 7 (2011)

[18] Jouault, F., Beaudoux, O.: On the Use of Active Operations for Incremental Bidirectional Evaluation of OCL. In: Proceedings of the 15th International Workshop on OCL and Textual Modeling, Ottawa. pp. 35–45. Springer (2015)

About the authors

Sina Madani is a PhD student in the Department of Computer Science at the University of York, specialising in parallel and distributed execution of model management programs. His research on performance optimisations in model management languages has been integrated into the open-source Eclipse Epsilon project, which he regularly contributes to. He can be contacted at *sm1748@york.ac.uk*.

Dimitris Kolovos is a Professor of Software Engineering in the Department of Computer Science at the University of York, where he researches and teaches automated and model-based software engineering. He is also an Eclipse Foundation committer, leading the development of the open-source Epsilon model-based software engineering platform, and an associate editor of the Software and Systems Modelling journal. He has co-authored more than 150 peer-reviewed papers and his research has been supported by the European Commission, UK's Engineering and Physical Sciences Research Council (EPSRC), InnovateUK and by companies such as Rolls-Royce and IBM.
http://dimitris.io/contact

Richard F. Paige is a Professor of Software Engineering at McMaster University, Canada, and holds the Chair in Enterprise Systems at the University of York, UK. He is an expert in Model-Driven Engineering, focusing on industrial applications and technology transfer. He is on the editorial boards for the Springer journals Empirical Software Engineering and Software and Systems Modeling, and is the special sections/themes editor for the Journal of Object Technology. He can be contacted at paigeri@mcmaster.ca.

Acknowledgements The work in this paper was supported by the European Commission via the CROSSMINER H2020 project (grant #732223).