
Proceedings Paper:
Madani, Sina, Kolovos, Dimitrios S. orcid.org/0000-0002-1724-6563 and Paige, Richard F. orcid.org/0000-0002-1978-9852 (2018) Parallel model validation with epsilon. In: Modelling Foundations and Applications - 14th European Conference, ECMFA 2018, Held as Part of STAF 2018, Proceedings. 14th European Conference on Modelling Foundations and Applications, ECMFA 2018 Held as Part of STAF 2018, 26-28 Jun 2018 Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) . Springer-Verlag , FRA , pp. 115-131.

https://doi.org/10.1007/978-3-319-92997-2_8

Parallel Model Validation with Epsilon

Sina Madani, Dimitrios S. Kolovos, Richard F. Paige

Department of Computer Science, University of York, UK
{sm1748,dimitris.kolovos,richard.paige}@york.ac.uk

Abstract. Traditional model management programs, such as transformations, often perform poorly when dealing with very large models. Although many such programs are inherently parallelisable, the execution engines of popular model management languages were not designed for concurrency. We propose a scalable data and rule-parallel solution for an established and feature-rich model validation language (EVL). We highlight the challenges encountered with retro-fitting concurrency support and our solutions to these challenges. We evaluate the correctness of our implementation through rigorous automated tests. Our results show up to linear performance improvements with more threads and larger models, with significantly faster execution compared to interpreted OCL.

1 Introduction

Many MDE tools face performance difficulties when dealing with very large models. Scalability has been a long-standing issue with MDE [1]. Moreover, it is a multi-faceted problem, with challenges in model persistence, collaboration, domain-specific languages, queries and transformations [2]. Most popular MDE tools were not designed to handle models with millions or even tens of thousands of elements. Furthermore, their execution engines typically perform unnecessary computations and do not exploit capabilities of modern hardware. Improving the efficiency of existing model management programs would help to address their poor performance over large models.

This paper aims to improve the execution time of model validation constraints by devising a novel parallel execution approach which can scale not only with the constraints, but also with the model elements. Specifically, we apply a novel rule and data-parallel approach to the task of model validation. We implement our solution by modifying the execution engine of the Epsilon Validation Language – a hybrid (i.e. declarative and imperative) model validation language. In doing so, we uncover a multitude of practical challenges with concurrency and provide solutions to such challenges. We hope that by choosing a complex model validation language and supporting all of its features without any additional constructs or change in syntax or semantics, other model validation solutions with equal or more limited expressive power can also be adapted in a similar manner. Our implementation is open-source, available from Epsilon Labs repository [3].

The remainder of this paper is as follows. Section 2 reviews the most relevant work related to our research. Section 3 introduces the Epsilon Validation

Language. Section 4 discusses challenges encountered with parallel model validation. Section 5 presents an overview of our parallel implementation. Section 6 evaluates the correctness and performance of our implementation, including a comparison to interpreted and compiled OCL. Section 7 concludes the paper and outlines further development opportunities.

2 Related Work

Our work is a direct continuation of a previous effort to parallelise the Epsilon Validation Language in a Masters' project [4]. The implementation and choice of metamodel for performance evaluation was inspired by this work. Although some promising speedups were achieved and the main challenges with parallelisation were identified, there remained some outstanding issues with the implementation and evaluation, which we hope to have addressed after substantial refactoring.

2.1 Background

Generally there are three approaches to improving model management program performance: *incrementality*, *laziness* and *parallelism*. The MDE community has put substantial effort into studying incrementality as this can lead to significant performance improvements. This arises by caching results for model elements which have already been evaluated so that after a small change, the program is only executed over the changed elements. Furthermore, a *reactive* execution engine can be obtained by combining incrementality with laziness, so that the program is not only re-executed on a subset of the model, but also only when the results of the program are actually used. Most research in this area focuses on optimising model transformation engines such as ATL [5], as the ATL language and its engine's architecture make such semantics more straightforward to implement. Consequently, there are incremental [6], lazy, [7], parallel [8], distributed [9] and reactive [10] extensions of ATL.

Although incrementality and laziness are valuable optimisations, they do not improve execution times when computations are mandatory, for example when a program is executed on a model for the first time, or a large proportion of the model is changed, and when the program results are always used. By contrast, parallelism exploits modern hardware to perform computations at a higher rate, rather than reducing the overall number of computations.

2.2 Model Validation optimisations

The most well-known and commonly used language for model querying and validation is the Object Constraint Language (OCL) [11], which is a functional language free from side-effects and imperative features. Most optimisations of model validation algorithms are built on OCL.

Cabot and Teniente (2006) [12] designed an incremental model validation algorithm which ensures the smallest / least work expression can be provided to

validate a given constraint in response to a CRUD (Create/Read/Update/Delete) event / change in the model. It automatically generates the most efficient expression for incremental validation for a given event. Tisi et al. (2015) [13] proposed an iterator-based lazy production and consumption of collection elements. Iteration operations return a reference to the collection and iterator body, which produces elements when required by the parent expression. Laziness in this context is useful when a small part of a large collection is required, as the iteration overhead can actually be worse than eager evaluation in some cases.

Vajk et al. (2011) [14] devised a parallelisation approach for OCL based on Communicating Sequential Processes (CSP). The authors' solution exploits OCL's lack of side effects by executing each expression in parallel and then combining the results in binary operations and aggregate operations on collections. They demonstrate equivalent behaviour between the parallel and sequential OCL CSP representations analytically. Their implementations use CSP as an intermediate representation which is then transformed into C# code. Users must manually specify which expressions should be parallelised. The authors' evaluation was brief, with relatively small models and simple test cases. Despite the absence of any non-parallel code in their benchmark scenarios, their implementation was 1.75 and 2.8 times faster with 2 and 4 cores respectively.

3 Epsilon Validation Language

Epsilon [15] is an Eclipse project that provides languages for model management tasks such as validation, transformation, comparison and pattern matching. All task-specific languages build upon a feature-rich model-oriented language – the Epsilon Object Language (EOL) [16] – a dynamically typed, interpreted language which supports imperative programming. EOL also supports native types, effectively allowing for execution of arbitrary Java code. A key feature of Epsilon is that it works across a range of modelling technologies by abstracting model operations through a connectivity layer, so a script written to work on an EMF model can also be used on an XML document or a spreadsheet without changes.

The Epsilon Validation Language (EVL) [17] extends EOL to enable users to express their validation constraints in a more structured and declarative manner. Since EOL supports all features of OCL with similar syntax and built-in operations, EVL is can be used in the same manner as OCL. Users define constraints within the context of a model element type, where each constraint has a check expression (or statement block for more complex constraints) returning a Boolean. In addition, constraints may also have a *guard*, which is semantically identical to prepending the constraint check expression with a Boolean expression followed by the *implies* operator. Guard blocks can also be declared in a context. Constraints may have dependencies on other constraints using the *satisfies* operation, which returns the result of calling the specified constraint(s) for the current element. Constraints declared as *lazy* will only be executed when invoked by a *satisfies* operation. A context declared as lazy is equivalent to having all of its constraints being lazy. EVL also allows users to defined fixes for

constraints, which can modify the model with arbitrary imperative code. Like all Epsilon rule-based languages, EVL has *pre* and *post* blocks, which allow for arbitrary code to be run before and after the main program, respectively.

3.1 Example Program

Suppose that we have a model of a Java program, and we want to ensure that every class overrides the *equals* method according to the contract[1]. Listing 1 shows how this could be implemented in EVL. Note that by declaring the *hasHashCode* constraint as lazy, we only check it once per *ClassDeclaration* as a pre-condition for the *hasEquals*. If *hasHashCode* fails for the current element under consideration (referred to as *self*), then we avoid executing the *hasEquals* check expression for the current class. This means a class may fail to satisfy either *hasHashCode* or *hasEquals*, but not both. Therefore our results will never contain the same class more than once. Also note that by declaring the *getMethods* operation as cached, we avoid re-evaluating it for a given model element, so that if a class satisfies *hasHashCode*, we do not need to find all of its methods again in line 9.

Listing 1. EVL program over Java metamodel

```
1  @cached
2  operation AbstractTypeDeclaration getMethods() : Collection {
3     return self.bodyDeclarations.select(bd|bd.isKindOf(MethodDeclaration));
4  }
5
6  context ClassDeclaration {
7     constraint hasEquals {
8        guard : self.satisfies("hasHashCode")
9        check : self.getMethods().exists(method |
10          method.name == "equals" and
11          method.parameters.size() == 1 and
12          method.parameters.first().type.type.name == "Object" and
13          method.modifier.isDefined() and
14          method.modifier.visibility == VisibilityKind#public and
15          method.returnType.type.isTypeOf(PrimitiveTypeBoolean))
16    }
17    @lazy
18    constraint hasHashCode {
19       check : self.getMethods().exists(method |
20          method.name == "hashCode" and
21          method.parameters.isEmpty() and
22          method.modifier.isDefined() and
23          method.modifier.visibility == VisibilityKind#public and
24          method.returnType.type.isTypeOf(PrimitiveTypeInt))
25    }
26 }
```

[1] Equal objects must have the same hash code, but unequal objects do not necessarily have different hash codes.

3.2 Execution Semantics

The execution algorithm for sequential EVL is given in Listing 2.

Listing 2. Simplified sequential EVL algorithm

```
1  preBlock.execute();
2  for (Context context : contexts) {
3     for (Object element : context.getAllOfKind()) {
4        if (!context.isLazy() && context.guard(element)) {
5           for (Constraint constraint : context.getConstraints()) {
6              if (!constraint.isLazy() && constraint.guard(element)) {
7                 if (!constraint.check(element)) {
8                    unsatisfiedConstraints.add(constraint, element);
9                 }
10             }
11          }
12       }
13    }
14 }
15 postBlock.execute();
```

For each context (line 2), we loop through all elements of that type and subtypes (line 3). Provided that the guard blocks of each context and constraint are satisfied and they are not marked as lazily evaluated (lines 4 and 6 respectively), we simply execute the check block (line 7) of each constraint within the declared context (line 5) for the current element. We add each failure to the set of unsatisfied constraints (line 8). Not shown in Listing 2 is the constraint trace, which keeps track of results to avoid re-evaluating constraint and element pairs in case of a satisfies operation (i.e. dependencies between constraints). The semantics of how this is used will be discussed in the next section. Also note that the *pre* and *post* blocks (lines 1 and 15, respectively) are not of interest as they may contain arbitrary imperative code. We have also excluded fixes for simplicity.

4 Challenges with Parallelisation

Our observation from Listing 2 is that each iteration of the three loops need not be performed sequentially, since there is no dependency between them (except for occasional constraint dependencies, discussed below). Fixes in EVL are performed optionally after validation and initiated by the user, so the model is only queried, never written to[2]. In theory, this makes the task of executing read-only operations (check blocks) within a loop inherently parallelisable. However in practice, this is complicated by a number of factors, to which we now turn.

4.1 Accessing Data Structures

A key challenge with retro-fitting concurrency into an existing program is handling of access to data structures. When multiple threads have shared access

[2] Parallel execution of fixes is beyond the scope of this paper.

to the same mutable data, the non-deterministic nature of parallel execution can lead to inconsistent states (Readers-Writers problem). There are generally three solutions to this problem: Not sharing such data between threads, making the data immutable; or using synchronization whenever accessing the data [18]. Unfortunately in most cases, the easiest option of the three (synchronization) is adopted. This has a major impact on performance not only because a single thread can execute synchronized regions at a time, but also the overhead introduced by synchronization mechanisms. This is especially problematic for data structures which are subject to frequent writes.

Even though model validation is in principle a read-only task, intermediate data structures such as the set of unsatisfied constraints need to be written to concurrently. Furthermore, caches (such as those used to store model elements) can present problems if they are written to during execution. In Epsilon, caching of model elements is performed lazily, i.e. when all elements of a specified type are requested for the first time.

4.2 Control Flow Traceability

It is important to be able to report on errors encountered during execution. EVL scripts are interpreted, so errors such as accessing an invalid model property are reported at runtime. Epsilon therefore records the execution stack trace so that in the event of an error, the location of the fault can be identified and reported to the user. When executing concurrently however, each thread could be executing different parts of the script or the same parts with different data. When an exception occurs, a co-ordination mechanism is needed to stop all threads from executing, and for the cause of the exception to be correctly reported. Furthermore, since exceptions are usually propagated to the program's top level, the reporting needs to be able to capture the stack trace of the thread which encountered the issue and make it available to the main thread, as parallel execution should be terminated at this point.

4.3 Handling Properties and Variables Scope

EVL is a structured extension of EOL, which supports almost every feature of a general-purpose scripting language. Amongst these are user-defined operations which may be defined in the context of types such as model elements or even built-in types. More fundamentally however is the ability to define variables in different scopes. Epsilon therefore has an internal frame stack which is used heavily throughout the code base. With multiple threads executing concurrently, the scoping of variables needs to be respected in an equivalent manner to sequential execution. So, for example, whenever a variable is declared in an executable block, once that block has finished execution, the variable should be discarded and inaccessible from all threads. Similarly, if a variable is declared globally in the *pre* section, it should be visible at all times to all threads.

Furthermore, EOL also allows individual objects (e.g. model elements) to have extended properties associated with them. These properties should be accessible from multiple threads.

4.4 Lazy Constraints and Dependencies

A classic impediment of parallelism is dependencies. In EVL, this can occur through *satisfies* operation calls. This is typically used in the guard block of a constraint to prevent it from executing if another constraint (or set of constraints) is not satisfied for the same model element. With multiple threads of execution, the target(s) of a *satisfies* operation may be executing concurrently with the caller. This means that there may be a duplication of effort, with the same constraint being executed at least twice, or the caller may need to wait for the result. In the latter case, not only does performance become single-threaded but there is a co-ordination overhead of notifying the caller when the result is made available. Further complicating matters are *lazy constraints*, which are only executed when invoked by a *satisfies* operation.

4.5 Testing for Correctness

Finally, we would like to emphasize the non-deterministic nature of concurrent programs. With single-threaded execution, the behaviour of the program is predictable, so a test suite which passes once will always pass for the same program with identical inputs. However with multiple threads, those same tests may become "flaky"; failing only on some occasions (depending on thread scheduling). In the best case, inconsistent output would result in a failure on at least one occasion, thus exposing a potential issue. Much more dangerous is correct behaviour under test conditions but spurious runtime exceptions resulting from a malformed internal state. Furthermore, debugging concurrent programs is also difficult, since the same tools and techniques used to detect issues with sequential programs may be inadequate or misleading when used for concurrent programs.

5 Parallel Solution

In this section we give a high-level overview of our solution to the problems identified in the previous section. Firstly we begin with an outline of the parallel execution approach.

5.1 Architecture

Our parallel solution abstracts the execution process using an extension of the *ExecutorService* [19] interface. This allows us to, in theory, substitute any parallel execution infrastructure without depending explicitly on threads. Our implementation uses a custom *ThreadPoolExecutor* [20] with a fixed pool of threads. Parallel execution begins when the EVL script has been parsed and the models under validation are fully loaded into memory, and ends once all constraints have been checked.

5.2 Parallelisation strategies

To achieve maximum parallelism, it is important to choose the appropriate level of granularity. For instance, parallelising only constraints themselves would have no performance benefits if there is just a single constraint in the script. A similar argument can be made for contexts. Since the issue of scalability is rooted in the size of models, parallelism should ideally be performed at the element level. Parallelisation is performed by wrapping the desired jobs into a function and passing it to the *ExecutorService*. We have experimented with the following parallel implementation strategies:

Element-Based In this strategy, we parallelise the second for loop (line 4 onwards) in Listing 2. Each context and constraint is executed in a single thread, but a separate job is created for each element. This is ideal if the model is large and the number of constraints and contexts is small.

Stage-Based This strategy is unlike the previous two in that it splits the execution into three distinct phases, where the input to each phase is the output from the previous phase. In the first phase, we loop through all elements in all contexts (as in lines 2 and 3), and submit a job for each context and element pair. The job simply checks whether the context should be executed (as in line 4) and if so, it adds the context and element pair to a thread-local batch data structure. Once this is complete, the results from all threads are merged and passed on to the next phase. In the second phase, we loop through the results from the first phase and all of the constraints for each context in the results (as in line 6) and check whether the constraint should be executed (as in line 6) and if so, it adds the element and constraint pair to another thread-local batch data structure. As before, once the jobs have completed the thread-local results are merged and passed to the final phase. In the third phase, we submit a job for each constraint and element pair (as in lines 7–9) and await the results.

This strategy clearly separates the three stages of the algorithm, with the main advantage being that we can achieve maximum parallelism at all of the for loops. Furthermore, it may be helpful for garbage collection since the data is more clearly scoped as a result of the staged filtering process. On other other hand, the overhead introduced by additional intermediate results structures could be detrimental to performance and/or memory consumption.

Constraint-Based This strategy differs from the element-based solution in that the parallelisation is performed at the third for loop (line 6 onwards). This means we create a job for each constraint and model element pair (i.e. it is both data and rule parallel), but the context guards are executed by the main thread. This is ideal if there are many constraints and there are no guarded contexts.

5.3 Thread-Safe Data Structures

It is useful to classify access to internal data structures during execution of the script into one of three categories: read only, write only, and read and write. The first category is inherently thread-safe since it is immutable. Such structures include the script itself, the model, the constraints and constraint contexts. The second category will never be queried during execution. An effective solution for this is to create a per-thread data structure and then merge all of the thread-local data structures once execution has completed. Since no thread will ever attempt to read from the structure, we do not need to merge or synchronize access during execution. The set of unsatisfied constraints (i.e. the results data structure) belongs in this category. The third category is unsurprisingly the most complex to deal with. Structures which fall into this category include the operation contributor registry (a cache for storing operations available on a given object), constraint trace (a cache of executed constraint-element pairs and their results), execution controller (which keeps track of the stack trace and allows for debugging of statements and expressions) and the frame stack.

Our solution is to use a thread-local structure (serial thread confinement) with base delegation. We will use the frame stack as an example to illustrate this. The idea is that each thread has its own frame stack (which is only accessible from that thread) so that whenever the *getFrameStack()* method is called, we return the frame stack associated with the calling thread. Each thread-local frame stack also has a reference to the main thread's frame stack. Whenever a variable lookup is performed, we first check the thread-local stack and if it is not present, we then look in the base. Once parallel execution has finished (i.e. all constraint and elements pairs have been checked), we merge the thread-local results back into the base frame stack (i.e. that of the main thread). In the constraint-based strategy, the main thread also needs to write to the frame stack during execution, so we make the base structure thread-safe by using an appropriate collection. In all cases, this is either a *ConcurrentLinkedDeque* [21] (e.g. for frame stack); a lock-free double-ended queue structure where writes are based on atomic compare-and-swap operations or *ConcurrentHashMap* [22] (where there is no synchronization for reads and locking for writes is of high granularity). We found that this approach eliminated many concurrency issues and is sufficient for supporting EVL's imperative features without introducing a major performance bottleneck due to excessive synchronization.

5.4 Exception Handling

By using a thread-local execution controller, each thread is able to keep track of its own execution trace so that when an error occurs, the cause can be identified in a similar manner to sequential execution. However we found that propagation and signalling that an exception has occurred to be more involved. Our solution is to use an *ExecutionStatus* object which encapsulates the state of success and/or failure within our *ExecutorService*. The idea is that when all jobs have been submitted to the executor, we start a termination thread which blocks

until the *ExecutorService* has finished executing all jobs. Meanwhile, the main thread locks onto the *ExecutionStatus*, waiting for a signal. So at this point, both the main and termination threads are idle. If the *ExecutorService* completes all jobs successfully, the termination thread signals a condition which notifies the main thread, at which point the termination thread ends and normal execution is resumed. If an exception occurs, the main thread is also signalled and the exception can be propagated as usual. The exception signalling occurs by calling the *setException* method in our *ExecutionStatus* object. Before this method gets called, we first capture the exception message, since the thread-local stack trace will disappear once parallel execution ends.

5.5 Dependencies

The original EVL algorithm added every constraint and element pair it checked to the constraint trace. This was wasteful since in most cases there are no dependencies between constraints. This is also the only structure for which we use a synchronized collection rather than thread-local base delegation, which introduces considerable overhead after checking each constraint and element pair. We changed this behaviour (in both sequential and parallel EVL) to avoid unnecessary writes to the constraint trace whilst also limiting the number of times each constraint-element combination is checked to at most 2 times. This is achieved by keeping track of the set of constraints depended on. When a *satisfies* operation is invoked, we first check whether the constraint is in this set. If so, we then proceed to check the trace for the specific constraint and element. If the result is not present, we perform the check and add it to the trace. If the constraint was not in the set of constraints depended on, we add it and also add the result to the trace. In practice, we optimise the checking of the constraints depended on every time a constraint is executed using a flag which indicates whether the constraint is a dependency. This flag is set to true on the constraint when it is first invoked by a *satisfies* operation. If this flag is true, we know to check the trace for a result, otherwise we proceed as usual.

As an example, suppose that constraint A depends on constraint B. If A runs before B, then B is checked during A. However B is then added to the trace and constraints depended on, so when B runs, it will not be re-checked. If B runs before A, then unfortunately B is checked again when A runs, but won't be checked afterwards because it will be in the trace. Future invocations (i.e. with different elements) will then know to check the trace first because constraint B will be in the set of constraints depended on.

6 Evaluation

As our solution is built on an already established model validation engine and does not change the syntax, semantics or supported features of the existing platform, our evaluation criteria will focus exclusively on *correctness* and *performance*. We consider correctness to be a hard requirement, since concurrent

programs are notoriously difficult to reason about due to the non-deterministic nature of execution.

6.1 Test Models and Scripts

Our main test script runs over models conforming to the Java 5 language metamodel provided by MoDisco [23]; a model-driven reverse engineering project designed to migrate legacy code artefacts into models. We chose this metamodel because it is substantially complex (749 elements) yet relatively easy to comprehend given the familiarity of the domain to Java programmers. Moreover, we are able to obtain models from real code artefacts automatically using MoDisco's Java Discovery feature as opposed to synthetically generating large models. Of course, since there are plenty of open-source Java projects, it is easy to obtain models of various sizes. For convenience and reproducibility, we used the models from [24], which vary from approximately 100,000 to over 4.35 million elements.

For our validation constraints, we took inspiration from the Findbugs[3] project, which lists a large number of "code smells" in Java code. Some of these require sophisticated static analysis, so in order to minimise errors with our validation constraints we implemented a subset of the simpler bug locators. Our EVL program consists of 31 constraints across 16 contexts (model element types), written in a declarative style. To ensure that our parallel implementations scale as intended, we also created three other scripts. One of these contains a single constraint for each of the 16 contexts, another contains 9 constraints within a single context, and one which consists of a single constraint within a single context. The rationale is to test throughput and identify any potential weaknesses in the scalability of our solutions.

6.2 Correctness

It would be challenging to formally prove correctness of our parallel solution using static analysis techniques due to the size and complexity of the codebase. Instead, we opt for a thorough series of automated dynamic JUnit tests. In this section, we give a brief overview of our testing methodology.

Epsilon already has a large suite of unit tests, especially for EOL, which we build upon. Our test suite for EVL ensures that all language features are exercised thoroughly. This is achieved by having a test script, a minimal plain XML model and by assertion of an expected number of unsatisfied constraints for each context and constraint in the script. The features tested include pre and post blocks, lazy constraints, constraint dependencies, contextless constraints, constraint pre-conditions, imperative code, user-defined, cached and imported operations, constraint messages, fixes as well as ensuring correct scoping of variables. We also have another script which accesses a non-existent property of the model to ensure correct propagation and reporting of exceptions in the user's code.

[3] `http://findbugs.sourceforge.net/bugDescriptions.html`

Our second test suite consists of equivalence tests with the sequential implementation. The infrastructure for this is rather complex, since we need to ensure that we are comparing the same model and script combinations whilst varying the modules (execution strategies); which themselves have different configurations as well. We automate this check by calculating an ID for each model, metamodel and script combination. We refer to a combination of model, metamodel, script and module as a *scenario*. Our first test is whether the scenario can actually execute without exceptions. Once this is established, we then check that for the oracle scenario (which uses the original sequential implementation), the results and internal data structures are "equal". These include the unsatisfied constraints, frame stacks, the constraint trace, constraints depended on, operation contributors and stack trace manager. We use the scripts described in the previous subsection as well as other complex scripts and models which were developed independently from the project. For this suite alone, we have 15 models, 5 metamodels and 9 scripts.

Since some changes were made to the Epsilon core code base, we also need to ensure that the original sequential EVL engine produces correct results when executed over a real model and script, as opposed to ones solely designed for testing the engine internals. To do this, we took our *java_findbugs* script and rewrote it in OCL. Since we execute the same model with EVL and used the same constraint names, we can could compare the references (i.e. memory address) of the *EObjects* (model elements) in our test suite to ensure that the set of results from EVL and OCL are identical.

All three of the above test suites are parameterised with an EVL engine implementation, so we could repeat the tests for all our solutions. Since we have three parallelised implementations, each of which accepts a number of threads to use as a parameter, we opted to use 1, 2, the number of logical cores on the system and many (8191) threads for each implementation. To detect concurrency issues, we also parametrised each suite with a number of cycles, which would repeat the tests a specified number of times. Given the very large number of combinations of modules and parameters, scripts and models as well as the tests themselves, each run results in thousands of unique tests.

After running the tests and real experiments on both small and large models tens of thousands of times on a HPC cluster, we can be confident in the correctness of our implementation from a practical sense as we did not encounter any exceptions, crashes or incorrect results.

6.3 Performance

Our platform for performance evaluation has the following characteristics: Windows 10 Enterprise (v1607), Intel Core i7-4790K @ 4.00 GHz (4 cores / 8 threads) CPU, 16 GB DDR3-1600 MHz RAM, Samsung 850 EVO 500 GB SSD, Java HotSpot 64-bit Server VM (build 9.0.4+11). We disabled Turbo Boost for a

fairer comparison when varying the number of threads. The arguments to the JVM were as follows [4]:

```
-XX:InitialRAMFraction=16 -XX:MaxRAMFraction=1
-XX:+AggressiveOpts -XX:+UseParallelOldGC
```

Table 1: Selected benchmark results

MODULE	SCRIPT	ELEMENTS	TIME	SPEEDUP	MEMORY
Sequential	findbugs_all	4M	9 227 448	—	2 758
Stage–Based (4)	findbugs_all	4M	3 009 693	3.066	4 342
Compiled OCL	findbugs_all	4M	22 024	418.972	475
Sequential	findbugs_all	1M	948 871	—	3366
Stage–Based (4)	findbugs_all	1M	330 027	2.875	4 967
Compiled OCL	findbugs_all	1M	7 078	134.059	559
Sequential	findbugs_all	200K	47 181	—	5 016
Stage–Based (4)	findbugs_all	200K	16 636	2.836	5 043
Compiled OCL	findbugs_all	200K	2 593	18.196	12
Sequential	findbugs_all	2M	3 815 590	1.119	3 782
Stage–Based (1)	findbugs_all	2M	4 265 051	—	4 822
Stage–Based (2)	findbugs_all	2M	2 162 179	1.973	5 050
Stage–Based (4)	findbugs_all	2M	1 251 400	3.408	4 981
Stage–Based (8)	findbugs_all	2M	874 808	4.875	5 031
Sequential	findbugs_all	3M	5 892 807	—	4139
Stage–Based (1)	findbugs_all	3M	6 626 802	0.889	4 428
Elements–Based (1)	findbugs_all	3M	6 693 093	0.88	4 196
Interpreted OCL	findbugs_all	3M	5 845 796	1.008	3 442
Compiled OCL	findbugs_all	3M	18 757	314.166	478
Sequential	findbugs_1Constraint	4.3577M	11 024	—	1 269
Stage–Based (4)	findbugs_1Constraint	4.3577M	8 248	1.337	2 149
Elements–Based (4)	findbugs_1Constraint	4.3577M	8 180	1.348	2 320
Interpreted OCL	findbugs_1Constraint	4.3577M	12 745	0.865	584
Sequential	findbugs_1Context	2.5M	30 193	—	1 236
Elements–Based (8)	findbugs_1Context	2.5M	6 038	5.000	3 490
Constraints–Based (8)	findbugs_1Context	2.5M	7 139	4.229	3 700
Stage–Based (8)	findbugs_1Context	2.5M	7 126	4.237	3 693
Elements–Based (4)	findbugs_1Context	2.5M	7 527	4.011	3 274
Constraints–Based (4)	findbugs_1Context	2.5M	8 379	3.603	3 368
Stage–Based (4)	findbugs_1Context	2.5M	8 997	3.356	3 582
Interpreted OCL	findbugs_1Context	2.5M	25 701	1.175	697

[4] We use the ParallelOld garbage collector since we're interested in throughput, as the G1 collector (the default in Java 9) is geared towards latency.

We measure the parsing and loading time of a model independently from the execution time of a module, using *System.nanoTime()*, and report our results in milliseconds. We calculate memory consumption (in MB) by summing the peak usage of all memory pools after measuring execution time. We run each experiment three times and use a script to automatically calculate the mean, speedup and efficiency of our results. We should also note that each experiment is run in a separate JVM invocation to minimise interference and warm-up effects.

6.4 Analysis of Results

Table 1 shows our main result, which we split into five sections (separated by alternated shading) for convenient analysis. All speedups are relative to the sequential EVL implementation unless otherwise indicated. The number of threads are in parenthesis where applicable.

Our parallel implementations perform similarly for the *findbugs* script, with around 3x speedup using four threads. This decreases slightly with model size, but for models with hundreds of thousands of elements, the performance improvements are still significant. The imperfect efficiency of our parallel solution can be explained mostly by the overhead of creating and submitting jobs to the executor. As demonstrated by our second group of results, when running a parallel version with a single thread as the baseline we see an almost perfect speedup using two threads. The original sequential implementation is over 10% faster than the single-threaded stage-based implementation, which can be interpreted as an estimate of the parallelisation overhead. The third group of our results shows broadly similar results across the parallel implementations, with single-threaded efficiency of 89% relative to the sequential implementation for three million elements. The fourth group of results act as a test of throughput, since the script contains only a single constraint. Here we see that despite the large model size, the absolute execution time is too small for parallelism to provide significant benefits, with efficiency dropping to just 33%. The last group of results shows that for multiple constraints within a single context, the element-based implementation is clearly superior. With four threads, we observe 100%, 90% and 84% efficiency for the element, stage and constraint-based implementations respectively. However it is the Hyper-threaded performance which shows marked differences. Although the constraint-based implementation is faster than the stage-based one with four threads, this difference disappears once we add the remaining logical processors. With eight threads, they both achieve 4.23x speedup, whereas the element-based implementation is five times faster.

We observe similar performance between interpreted OCL and sequential EVL for our main *findbugs* script, though there is a 15% difference for both the single constraint and single context variants, with OCL being faster in the former case and slower in the latter. However we should note that the implementations of EVL and Eclipse OCL are fundamentally different. Compiled OCL is in a league of its own in terms of performance, though its advantage greatly diminishes with smaller models. The benefits of using EVL over compiled OCL include a richer feature set and the ability to work with any modelling technology, amongst

others [17]. Furthermore, the performance of parallel EVL will improve over time as the number of cores in developer workstations increases. Currently, compiled Eclipse OCL requires the user to embed their constraints in the metamodel and manually regenerate the code for any changes to the constraints or metamodel.

Overall, these results indicate what one would expect from a parallel algorithm in that with smaller problems, the overhead of co-ordination outweighs the gains but with a bigger problem, the parallel version is able to "catch up" and overtake the sequential algorithm's performance [26]. Typical speedups for parallel model management programs in the literature with four threads range between 2.5x – 3x (in the case of LinTra [25], 1.19x with four threads and 3.24x with sixteen). However these are in the context of model-to-model transformation; a task which is arguably more complex.

7 Conclusions and Future Work

In this paper, we have presented a novel parallel model validation solution which is scalable with both the number of constraints and number of model elements. Along the way, we have identified and provided solutions to concurrency issues arising from uncommon features in model validation such as constraint dependencies and imperative programming constructs. We have also tested our solution by not only exercising all features of the language, but also through equivalence testing with the non-concurrent version as well as with OCL; a popular model querying and validation language with a well-defined specification. In terms of performance, we observed roughly linear improvements in execution times as we increased the number of threads, though naturally our parallel solution does impose an overhead which reduces the efficiency in cases where the absolute execution times are relatively small. However we realise that smaller models and scripts benefit significantly less, if at all, from parallelisation.

To further improve performance, we intend to combine our parallel solution with an incremental one [27], which will undoubtedly present new challenges. To improve scalability, we are also considering a distributed solution; though the lack of shared memory and communication costs in distributed systems would require some modifications to our proposed parallel solution. If our solution is modified to accommodate partial and lazy loading of models from non-volatile memory, we could also avoid the upfront cost (both in time and memory) of parsing the model and possibly even make the process multi-threaded.

Going beyond model validation, we plan to refine our solution by applying it to other rule-based model management tasks in Epsilon, such as pattern matching (EPL) [28], model comparison (ECL) [29] and model-to-text transformation (EGL) [30]. In principle, our systematic analysis and devised approach should also be applicable to other model management languages outside of Epsilon.

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. Parallel EVL implementation, `https://github.com/epsilonlabs/parallel-erl`
4. Smith, M.: Parallel Model Validation. Masters' thesis, University of York (2015).
5. Joualt, F., Allilaire, F., Bèzivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming, vol. 72 (1–2), pp. 31–39. Elsevier (2008)
6. Tisi, M., Jouault, F.: Towards Incremental Execution of ATL Transformations. In: Proceedings of the Third international conference on Theory and practice of model transformations, pp. 123-137 (2010).
7. Tisi, M., Martínez, S., Jouault, F., Cabot, J.: Lazy Execution of Model-to-Model Transformations. In: Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems, Wellington. pp. 32–46 (2011)
8. Tisi, M., Martínez, S., Choura, H.: Parallel Execution of ATL Transformation Rules. In: Proceedings of the 16th International Conference on Model Driven Engineering Languages and Systems, pp. 656–672 (2013)
9. Benelallam, A., Gómez, A., Tisi, M., Cabot, J.: Distributed model-to-model transformation with ATL on MapReduce. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, pp. 37-48 (2015)
10. Martínez, S., Tisi, M., Douence, R.: Reactive model transformation with ATL. Science of Computer Programming, vol. 136 (C), pp. 1–16 (2017)
11. Object Constraint Language (OCL) specification, `http://www.omg.org/spec/OCL/About-OCL/`
12. Cabot, J., Teniente, E.: Incremental Evaluation of OCL Constraints. In: Proceedings of the 18th International Conference on Advanced Information Systems Engineering, Luxembourg. pp. 81–95 (2006)
13. Tisi, M., Douence, R., Wagelaar, D.: Lazy evaluation for OCL. In: Proceedings of the 15th International Workshop on OCL and Textual Modeling co-located with 18th International Conference on Model Driven Engineering Languages and Systems, Ottawa. pp. 46-61 (2015)
14. 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)
15. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.C.: The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In: Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems, Potsdam. pp. 162–171 (2009)
16. 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)
17. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On the evolution of OCL for capturing structural constraints in modelling languages. Rigorous Methods for Software Construction and Analysis, pp. 204–218. Springer Berlin, Heidelberg (2009)

18. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice. Addison–Wesley (2005)
19. java.util.concurrent.ExecutorService, https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ExecutorService.html
20. java.util.concurrent.ThreadPoolExecutor, https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ThreadPoolExecutor.html
21. java.util.concurrent.ConcurrentLinkedDeque, https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ConcurrentLinkedDeque.html
22. java.util.concurrent.ConcurrentHashMap, https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ConcurrentHashMap.html
23. Brunelière, H., Cabot, J., Dupé, G., Madiot, F.: MoDisco: A model driven reverse engineering framework. Information and Software Technology, vol. 56 (8), pp. 1012–1032. Elsevier, (2014)
24. Eclipse platform EMF models, http://atenea.lcc.uma.es/index.php/Main_Page/Resources/LinTra#Input_Models_3
25. Burgeño, L., Troya, J., Wimmer, M., Vallecillo, A.: Parallel In-place Model Transformations with LinTra. In: Proceedings of the 3rd Workshop on Scalable Model Driven Engineering, L'Aquila. pp. 52–62 (2015)
26. Goetz, B.: From Concurrent to Parallel. QCon 2017, London. https://www.infoq.com/presentations/tecniques-parallelism-java
27. Incremental EVL, https://github.com/epsilonlabs/incremental-evl
28. Kolovos, D.S., Paige, R.F.: The Epsilon Pattern Language. In: Proceedings of the 9th International Workshop on Modelling in Software Engineering, Buenos Aires. pp. 54–60 (2017)
29. Kolovos, D.S.: Establishing Correspondences between Models with the Epsilon Comparison Language. In: Proceedings of the 5th European Conference on Model Driven Architecture – Foundations and Applications, Enschede. pp. 146–157 (2009)
30. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: The Epsilon Generation Language. In: Proceedings of the 4th European Conference on Model Driven Architecture – Foundations and Applications, Berlin. pp. 1–16 (2008)