Software and Systems Modeling (2021) 20:1689–1712
https://doi.org/10.1007/s10270-021-00878-x

REGULAR PAPER

Distributed model validation with Epsilon

Sina Madani[1] · Dimitris Kolovos[1] · Richard F. Paige[1]

Received: 8 March 2020 / Revised: 12 January 2021 / Accepted: 18 February 2021 / Published online: 25 March 2021

Abstract
Scalable performance is a major challenge with current model management tools. As the size and complexity of models and model management programs increases and the cost of computing falls, one solution for improving performance of model management programs is to perform computations on multiple computers. In this paper, we demonstrate a low-overhead data-parallel approach for distributed model validation in the context of an OCL-like language. Our approach minimises communication costs by exploiting the deterministic structure of programs and can take advantage of multiple cores on each (heterogeneous) machine with highly configurable computational granularity. Our performance evaluation shows that the implementation is extremely low overhead, achieving a speed up of 24.5× with 26 computers over the sequential case, and 122× when utilising all six cores on each computer.

Keywords Model-driven engineering · Model validation · Distributed computing · Model management · Parallelism

1 Introduction

Model-driven engineering is an established approach for managing complexity in large projects that involve many stakeholders and heterogeneous platforms and implementation technologies, by elevating machine processable models into first-class artefacts of the development process. Precise and fine-grained modelling and automated model management (model validation and transformation, code generation) are used extensively in domains such as automotive and aerospace where software defects are very expensive to remedy or can have dear consequences.

In such domains, large (e.g. MATLAB Simulink) models are commonplace and typically the result of detailed modelling of complex component-based systems to support model-based verification activities and, eventually, full code generation. Large models (of the order of gigabytes) can also emerge by reverse engineering legacy code bases for comprehension and re-engineering purposes [1]. Growing model sizes pose numerous and wide-ranging challenges [2] for modelling tool user interfaces, model persistence, collaboration and versioning mechanisms, as well as for tools and languages for queries and transformations. Popular dedicated tools for such tasks like Eclipse OCL suffer from poor scalability with large models, both in memory and execution time as the semantics and most implementations were not designed with performance in mind. Whilst there are numerous works which attempt to optimise Object Constraint Language (OCL) expressions (e.g. [3]), alternatives to OCL allow for a broader range of optimisations to be applied [4], due to some limitations imposed by the OCL specification [5]. That said, OCL (and similar) languages can also benefit from several optimisations (as described by in [6]).

Many optimisations have been proposed to improve scalability in automated model management (e.g. transformation, validation, code generation) languages. Such optimisations include lazy evaluation (i.e. avoiding unnecessary computation), incrementality (avoiding unnecessary re-computation through caching) and finally, exploiting the parallelism in modern hardware architectures.

The case for parallel computing is now clearer than ever. With single-core computers effectively obsolete, the increasing number of cores on modern CPUs in recent years, the relatively stagnant growth in clock speeds and instructions-per-cycle (IPC) improvements, and the demon-

Communicated by Antonio Vallecillo.

✉ Sina Madani
sina.madani@york.ac.uk

Dimitris Kolovos
dimitris.kolovos@york.ac.uk

Richard F. Paige
richard.paige@york.ac.uk

1 University of York, York, United Kingdom

strable performance benefits from multithreading, an obvious solution for model management programs is to take advantage of the available resources. However, such resources need not be limited to the capabilities of a single machine. With the ever-increasing number of computers and declining cost of computing resources, cloud computing is more accessible, enabling large and complex systems to scale with the available resources. Furthermore, the improvements in infrastructure and networks allows for distributed computing to be more efficient than ever before.

To exploit the capabilities of modern hardware (and cloud computing), software must be (re-)written from its monolithic form to granular blocks which can be executed independently. It is difficult to split models into independent parts due to the variety in (meta)models, which can be highly interconnected in nature. Nevertheless, the computations which are performed on such models are often (or can be) expressed independently, making them amenable to parallelisation. Unfortunately, most model management tools do not take advantage of such opportunities.

Clasen et al. [7] motivate the need for supporting distributed model transformations in the Cloud. The research challenges consist of two main parts: storage/distribution of models (decentralized model persistence) and distributed computation/execution of programs. Our research aims to address the latter in the context of model validation. Most research in the area of performance optimisations in model-driven engineering is primarily focused on model-to-model transformations. However, the same scalability issues apply to other model management tasks such as validation. The main contributions of this paper are:

1. A distributed execution architecture for the Epsilon Validation Language [8]; a complex, feature-rich and mature OCL-inspired model validation language. Our approach exploits the deterministic decomposition of programs and takes advantage of both distributed and local parallelism.
2. A bespoke prototype implementation of the proposed approach using the Java Message Service API. By virtue of being Java-based, our implementation can make use of heterogeneous computing resources, with different operating systems and hardware architectures being used in the same application.

Our performance evaluation shows that the implementation is extremely low overhead, achieving a speedup of 24.5× with 26 computers over the sequential case, and 122× when utilising all six cores on each computer. With 88 computers (a total of 536 cores), our distributed solution was 340× faster than the baseline sequential program.

The rest of the paper is organised as follows. Section 2 reviews pertinent works in parallel execution and performance optimisations in model validation. Section 3 gives a high-level conceptual overview of our distribution strategy. Section 4 provides background by introducing Epsilon and its validation language. Section 5 outlines our distributed execution approach in detail. Section 6 describes how this approach is realized concretely using JMS. Section 7 benchmarks the performance of our solution in comparison to the status quo of model validation. Section 8 concludes the paper and advocates extensions for further improvements.

2 Background and related work

Software models are often used in complex projects involving many stakeholders as central artefacts in the development process. These models conform to a domain-specific *metamodel* which captures the most pertinent concepts and relationships between components in the system's architecture. For example, a metamodel can be used to represent the abstract syntax of a programming language and, through reverse-engineering techniques, to parse programs as models conforming to the metamodel (grammar). For instance, one could build a model of a project derived from a version control system repository in order to perform further tasks at a higher level of abstraction. One of the earliest stages in model management workflows is validating the model to ensure it meets certain domain-specific requirements, which cannot be expressed by the metamodel alone due to their complexity.

In many cases, models are not small and modular but large and monolithic. This is due to the graph-like nature of models and dependencies between components within the system being modelled, as well as poor support for modularity in commonly used persistence formats. Although it is possible to construct more fine-grained models—for example, one model per file in the case of reverse-engineered source code—validation constraints may require a view of the whole system for consistency checks and to produce meaningful, complete diagnostics. This can arise when validation needs to be performed across different levels of abstraction—for example, if the presence of a file in the model must coincide with the correctness of a particular line of code. Therefore even if the components of a system are inherently modular, the correctness of the system or project as a whole cannot be determined solely by examining the correctness of its components if the system is more complex than the sum of its parts.

Model validation can be viewed as either a query or transformation. One could view a model validation program as a series of filtering (or *select*) operations which attempt to find model elements for which invariants of interest are not satisfied. Another perspective is that the invariants are akin to transformation rules which classify each applicable model element as being either satisfied or unsatisfied. Bèzivin and

Jouault (2006) [9] demonstrate how a model transformation language can be used to validate models and produce diagnostics.

The most well-known and commonly used language for model querying and validation is the Object Constraint Language (OCL) [10], which is a functional language free from imperative features. Most optimisations of model validation algorithms are built on OCL. Due to its desirable properties, it is also used as a query language in some modelling technologies and transformation languages such as QVT and ATL.

Although scalability and performance of model management programs is an increasingly active research area, most works primarily focus on model-to-model transformations. In this section, we provide an overview of relevant work.

2.1 Parallel graph transformation

Significant overlap exists between the graph transformation and model-driven engineering communities; particularly in regards to model transformation optimisations (also known as *graph rewriting*). As noted by Imre and Mezei [11], there are two phases of executing a transformation: the first is searching for applicable model elements to transform, and the second is performing the transformation. We can think of the first phase as being read-only, since the model is only queried, not modified. Therefore, this first phase can be parallelised. The second phase is much more complex, since the model may be modified and the order of modifications may impact the final state.

Krause et al. [12] apply the Bulk Synchronous Parallel (BSP) method to execute model transformation programs in a distributed manner. Notably, their devised algorithm performs both the pattern matching and rule execution (i.e. modifying the model) in parallel, with the assumption that there are no conflicting rules. The BSP model of distributed computation allows for communication between worker nodes, which is useful in model transformations for merging partial matches. Their approach scales both horizontally (i.e. with more workers) and vertically (i.e. using more CPU cores per worker). With the largest model in their evaluation, they were able to achieve 11× speedup moving from 2 to 12 workers.

As previously mentioned, most research in the area of optimising model management programs to improve scalability focus exclusively on model-to-model (or graph) transformations. As such, their solutions are highly elaborate and thus more constrained due to the complex nature of model transformations. Our work focuses specifically on the model querying and validation process, which does not require inter-node communication or modification of the model. Hence, we are able to devise a simpler solution for parallel and distributed execution.

2.2 Distributed model transformation

Benelallam et al. (2015) [13] demonstrate a distributed processing approach for the ATL model transformation language based on the popular *MapReduce* paradigm. Their approach executes the entire transformation on a subset of the model, exploiting the "nice" properties of ATL—namely locality, non-recursive rule application, forbidden target navigation and single assignment on target properties—which make such a strategy possible. The "local match apply" or *Map* phase applies the transformation on a given subset of the model, whilst the "global resolve" or *Reduce* phase brings together the partial models and updates properties of unresolved bindings. However, each worker requires a full copy of the model and the splits are randomly assigned. Their implementation was up to 5.9 times faster than sequential ATL with 8 nodes.

The authors expanded upon this work by attempting to improve data locality and efficiency of partitioning [14]. They note how computing a full dependency graph and framing the task as a linear programming optimisation problem can outweigh the benefits, so a faster heuristic is needed. Instead, they rely on static analysis to compute "footprints" of transformations (i.e. model element accesses for each rule) and approximate the dependency graph on-the-fly. Their approach splits the model into as many parts as there are workers, and attempts to maximise the dependency overlap in each worker's partition whilst also balancing the number of elements assigned to each worker. However, their solution requires the modelling framework to support partial loading of models and access to model elements by ID.

Burgeño et al. [15] present a model transformation approach based on the Linda co-ordination language. The idea is that a lower-level implementation is used which allows for multiple distributed tuple spaces that can hold (serializable) Java objects. Multiple threads can access the tuple space(s) and query it using SQL-like syntax. Although this approach works in both distributed and local environments, the scalability is far from linear, achieving an average of 2.57 times speedup with 16 cores compared to sequential ATL.

2.3 Distributed pattern matching

As previously mentioned, much of the computational expense in some model management programs stem from finding applicable model elements to execute rules upon. There are various approaches for executing this stage in parallel, and is of particular interest to the graph transformation community.

Szárnyas et al. [16] present *IncQuery-D*, a distributed and incremental graph-based pattern matching tool based on Rete network algorithm. Their architecture provides a framework for distributed model queries which can be performed incrementally, thanks to a distributed model indexer and a model access adapter with a graph-like API which allows for optimised queries based on the underlying storage solution, as their architecture allows for different types of persistence backends. They found the overhead of constructing Rete network makes it less efficient than non-incremental engine for smaller models, but the benefits outweigh the costs for medium-size models and above. Perhaps the main benefit of such an approach is the near-instantaneous query evaluation (after caching) even for models with well over 10 million elements. This work therefore serves as evidence that distribution and incrementality are not mutually exclusive.

Mezei et al. [17] propose distributed model transformations based on parallelisation of the pattern matching process. They use a 3-layer approach, with the master coordinating execution, "primary workers" applying re-write rules and "secondary workers" computing matches for the rules using a pseudo-random function. In other words, the master handles transformation-level parallelism, primary workers handle rule-level parallelism and secondary workers perform the pattern matching process. The authors approach the task of model transformation from a graph isomorphism / pattern-matching perspective, noting the wide range between worst-case and best-case execution time for computing matching rules.

2.4 Incremental and lazy execution

Incrementality is one of the most commonly explored solutions for improving performance of model management programs. An early example is presented by Cabot and Teniente [18], who 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 change in the model (i.e. Create/Read/Update/Delete events). They conceptualise the problem of model validation and use this prove that their solution automatically generates the most efficient expression for incremental validation in response to a given CRUD event.

Laziness is also used to delay or avoid unnecessary computations. Often the most demanding operations in model queries involve the (eager) retrieval of model elements, though an entire in-memory collection is rarely required. Tisi et al. [19] 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.

2.5 Parallel model validation

Vajk et al. [20] 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.

Finally, our recent work [21] presents a data-parallel approach for the execution of a complex hybrid model validation language. Notably, our work details the challenges with concurrency in this context and provides solutions for efficiently dealing with issues such as dependencies between rules (invariants). Our work builds on top of this, enabling us to take advantage of parallelism on a larger scale.

3 Overview

Rule-based model management programs typically execute some user-defined code over a subset of model elements. The language may provide some special constructs to decoratively filter the subset of model elements to apply a given rule to. To make this concrete, consider Listing 1, which shows the structure of an OCL model validation program. The *context* represents the model element type from which elements are drawn. For each element, each invariant within the context is executed, taking as input the model element (which is assigned to the variable *self*). Arbitrary logic (within the constraints of the language of course) which returns a Boolean may be expressed within each invariant.

Listing 1 Anatomy of OCL invariants

```
context ModelElementTypeA
   inv AC1: <expression>
   inv AC2: <expression>
context ModelElementTypeB
   inv BC1: <expression>
   inv BC2: <expression>
```

To execute a program structured in this way, the engine needs to loop through all model elements for each type and execute every invariant applicable to that type for the given model element. Assuming that each context is considered in order of declaration and, crucially, that the underlying model returns all model elements of a given type in a deterministic order, then we can build a list of model elements to be evaluated. We know which invariants to execute for a given element based on its type. Since this list of model elements can be reproduced identically for a given invocation of the same program and input model(s), we can spawn multiple processes with the complete program and model(s), on separate machines, to execute a given subset of this list. That is, each worker process will be assigned a subset of model elements, and evaluate all applicable invariants based on the type of those elements. Each invariant which is unsatisfied (i.e. the expression returns false) for a given model element is sent to the master. The invariant is identified by its name, and the model element by its index in the list. Since we can refer to model elements by their index in a list, we can batch consecutive elements in the list (i.e. instead of sending each job individually, we send a list of consecutive jobs to be executed atomically) to minimise network traffic and maximise CPU utilisation on each worker.

Sections 5, 6, and 7 describe how this can be efficiently achieved in practice, maximising parallelism and fairly distributing the workload amongst independent processes. We implement our solution on top of the Epsilon Validation Language using a messaging system for distribution and load balancing of jobs.

4 Epsilon validation language

In this section, we introduce the Epsilon Validation Language (EVL), which we use as a host language for implementing our distributed approach. We chose this language due to its notably rich set of features and compatibility with a wide range of modelling technologies. However, in principle other model management languages (e.g. OCL) can benefit from the approach discussed in this paper.

4.1 Epsilon

Epsilon[1] [22] is an open-source model management platform, offering a family of task-specific languages for model transformation, querying, comparison, merging, text generation, pattern matching and validation. All languages build on top of EOL [23]: an OCL-inspired model-oriented language with both imperative and declarative constructs. EOL offers many powerful features, such as the ability to define operations on any types and call native Java methods. A distinctive feature of Epsilon is that its languages are decoupled from the modelling technologies, so programs can be run on models irrespective of their representation. This is achieved through a model connectivity layer which offers an abstraction that can be implemented by tool-specific *drivers*. Several such drivers have been developed that enable Epsilon programs to operate on EMF models, XML documents, spreadsheets and models captured with commercial tools such as Matlab Simulink, PTC Integrity Modeller and IBM Rational Rhapsody.

[1] eclipse.org/epsilon.

4.2 EVL

The Epsilon Validation Language (EVL) is a model validation language similar to OCL but with more advanced capabilities. Like OCL, invariants (*Constraint*s in EVL terminology) are expressed as Boolean expressions in the context of model element types. However, EVL allows the body of invariants (the *check* block) to be arbitrarily complex, combining imperative and declarative style statements and expressions. This is in contrast to OCL, which is not suitable for imperative style of programming. Since EOL is not strictly a functional language and allows for side-effects (such as declaration of global variables), users can specify *pre* and *post* blocks which are executed once before and after the main program, respectively. The *pre* block is typically used to set up variables which can be referred to within the constraints and *post* block, and may contain arbitrary imperative code.

Other features of EVL include the ability to declare preconditions for constraints (semantically identical to using the *implies* operator), specify custom messages for unsatisfied constraints (using information from the failed instance) and even the ability to declare solutions (*fixes*) for unsatisfied constraints. Perhaps the most interesting feature from a research perspective is that constraints may have dependencies between them and can also be lazily invoked. Constraints may invoke one or more constraints through the *satisfiesOne* and *satisfiesAll* operations. These operations can be invoked on a given model element type (usually derived from *self*, i.e. the current model element), taking as input the name(s) of the constraint(s) for which the given element should be validated against. The former checks that at least one of the specified constraints return true, whilst the latter requires all of the constraints to be satisfied.

4.3 Example program

Listing 2 shows a simple EVL program demonstrating some of the language features, which validates models confirming to the metamodel shown in Fig. 1. Execution begins from the *pre* block, which simply computes the average number of actors per Movie and stores it into a global variable, which can be accessed at any point. The *ValidActors* constraint checks that for every instance of *Movie* which has more than the average number of actors, all of the actors have valid names. This is achieved through a dependency on the *Hash-*

ValidName invariant declared in the context of *Person* type. This constraint is marked as lazy, which means it is only executed when invoked by *satisfies*, so avoiding unnecessary or duplicate invocations. The *HasValidName* constraint makes use of a helper operation (*isPlain()*) on Strings. Once all Movie instances have been checked, the execution engine then proceeds to validate all *Person* instances, which consists of only one non-lazy constraint *ValidMovieYears*. This checks that all of the movies the actor has played in were released at least 3 years after the actor was born. Finally, the *post* block is executed, which in this case simply prints some basic information about the model.

Listing 2 EVL program over IMDb metamodel

```
pre {
 var numMovies = Movie.all.size();
 var numActors = Person.all.size();
 var apm = numActors / numMovies;
}
operation String isPlain() : Boolean {
 return self.matches("[A-Za-z\\s]+");
}
context Movie {
 constraint ValidActors {
   guard : self.persons.size() > apm
   check : self.persons.forAll(p |
    p.satisfies("HasValidName")
   )
 }
}
context Person {
 @lazy
 constraint HasValidName {
   check : self.name.isPlain()
 }
 constraint ValidMovieYears {
   check : self.movies.forAll(m |
     m.year + 1 > self.birthYear
   )
 }
}
post {
 ("Actors per Movie="+apm).println();
 ("# Movies="+numMovies).println();
 ("# Actors="+numActors).println();
}
```

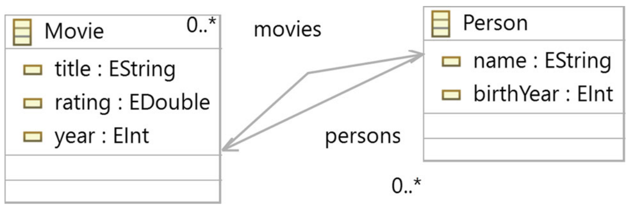


Fig. 1 Movies metamodel

5 Parallel EVL

In this section we briefly review the parallel implementation, as described in [21].

5.1 Algorithm

The architecture is relatively straightforward: a fixed pool of threads (equal to the number of logical cores in the system) is managed by an *ExecutorService*. The execution algorithm is conceptually the same as in the sequential implementation, however each ConstraintContext is executed independently for each applicable model element.[2] That is, the algorithm is data-parallel. The result of this algorithm is a Set of *UnsatisfiedConstraint* objects, where each UnsatisfiedConstraint is a diagnostic describing the invariant and the offending model element. The details of how dependencies are handled is explained in [21]. Briefly, we set a flag on a Constraint when it is invoked as the target of the dependency, so that we know whether we should check the trace (cache of executed constraint-element pairs and their results) rather than re-executing it. This avoids wastefully reading from and writing to it for each invocation. For simplicity, Algorithm 1 does not show this.

5.2 Data structures

Parallel execution of Epsilon programs is non-trivial due to the internal design, assumptions and data structures used in the respective interpreters. There are three notable core engine structures (discussed below) where thread safety is guaranteed via serial thread confinement. That is, in each of the cases, we use *ThreadLocal* instances of the structure, so that each thread has its own independent copy. Note that in visitor-based interpreter implementations this may not be necessary, though we found this solution to be the most convenient without re-architecting Epsilon.

5.2.1 Frame stack

The *FrameStack* is used to store variables during execution. Variables are mutable and may be stored in different scopes. For example in EVL, variables declared in the *pre* block are always visible. Therefore the FrameStack is split into multiple regions. Note that not all variables are explicitly declared by the user. For example when executing a constraint, the *self* variable is bound to the model element under execution by the engine and placed into the FrameStack. Although under parallel execution each thread's FrameStack is unique, any variables declared in the main thread (e.g. in the *pre*

[2] *getAllOfKind()* returns all types and subtypes described by the ConstraintContext.

Algorithm 1 Simplified parallel execution algorithm

```
results = new ConcurrentHashSet
p = Runtime.availableProcessors()
executorService = new ThreadPoolExecutor(p)
for all cc in module.getConstraintContexts() do
    for all element in cc.getAllOfKind() do
        executorService.submit(new Runnable {
            for all c in cc.getConstraints() do
                if c not lazy and c.executeGuard(element) then
                    if not c.executeCheckBlock(element) then
                        results.add(new UnsatisfiedConstraint(c, element)))
        }
executorService.awaitCompletion()
```

block) must always be visible. Therefore when resolution of a variable fails under parallel execution, the main thread's FrameStack is queried.

5.2.2 Execution trace

All module elements (ASTs)—i.e. all expressions, statements, operators etc.—are executed through an internal data structure, which may have listeners to record various information before and after execution. Most notable is the stack trace, so that if an error occurs, the exact location of the expression / statement, along with the complete trace leading up to it, is reported to the user. With multiple threads of execution, different parts of the program (or the same parts with different data) may be executed simultaneously, therefore each thread needs to keep track of its own execution trace. The same applies to profiling information. We track the cumulative execution time of each Constraint on a per-thread basis, then merge the results back to the main thread's structure.

5.2.3 Operation calls

Epsilon's languages are interpreted and make heavy use of reflection internally. EOL programs can invoke arbitrary methods defined on objects, since it is implemented in Java. This is in addition to built-in operations which extend the functionality of existing types (known as *operation contributors*), as well as user-defined operations. When invoking an operation on an object, a tuple of the target object and the operation (Java Method) being invoked is created. The operation resolution and execution process is quite complex. With multiple threads of execution, the infrastructure used to handle operation calls needs to be modified to be able to deal with multiple simultaneous invocations. Again we use serial thread confinement for this.

5.2.4 Caches

As with most complex applications, there are various caches involved in the execution engine and the modelling layer which need to be thread-safe. This is especially true for models, where a cache of the model's contents as well as of its types (internally, a *Multimap* from type name to a collection of its elements) is kept for improved performance. In the case of the Multimap, not only does the underlying map need to be thread-safe, but also the collection of elements. We use high-performance data structures such as *ConcurrentHashMap* and a custom non-blocking collection based on *ConcurrentLinkedQueue* to avoid synchronization where possible. The reason thread-safe caches are needed even though we never mutate / write to the model is because these caches are populated lazily at runtime when required.

5.2.5 Results

The output of an EVL program is a Set of *UnsatisfiedConstraint*s, where each UnsatisfiedConstraint contains the invariant, the model element which violated the invariant and the (optional) description for why it failed. During execution, this set is only ever written to, but not queried. Therefore any solution which is "write-safe" is valid. We could use persistent thread-local values and merge them once parallel execution has completed, however with many threads and UnsatisfiedConstraints this is expensive. Our solution uses a non-blocking collection to write all results to, which is then converted to a Set.

6 Distributed EVL architecture

In this section, we describe our distribution strategy for the Epsilon Validation Language (EVL) and its execution semantics independently from any underlying distribution framework or transport technology. First, we turn to the data structures, as any distributed computation approach requires a way to break down the computational problem into smaller tasks.

6.1 From parallel to distributed

As mentioned previously, the motivating factor behind distributed execution is to improve performance when the model(s) under validation are large. As with the parallel implementation (discussed in the previous section), we can decompose the validation program at the finest levels of granularity in a data parallel manner. The core building block of each computational "*atom*" is the *ConstraintContextAtom* (see Fig. 2). Essentially this is a data structure comprising of a *ConstraintContext*—the construct which contains the constraints applicable to a given model element type—and a single model element of that type. Every EVL program can then be decomposed into these atoms. For example, the program in Listing 2 would result in as many atoms as there are instances of *Movie* and *Person* in the model. Algorithm 2 shows the job creation algorithm, along with a graphic in Fig. 3.

More generally, if an EVL program includes invariants for M model element types, such as M_a, M_b, M_c etc. with N instances (model elements) of each type (e.g. N_a for M_a) then the number of atoms will be $M * N$ for all M.

We can also adapt Algorithm 2 to be both rule- and data-parallel by making use of a *ConstraintAtom* instead, where each job consists of a constraint-element pair. However,

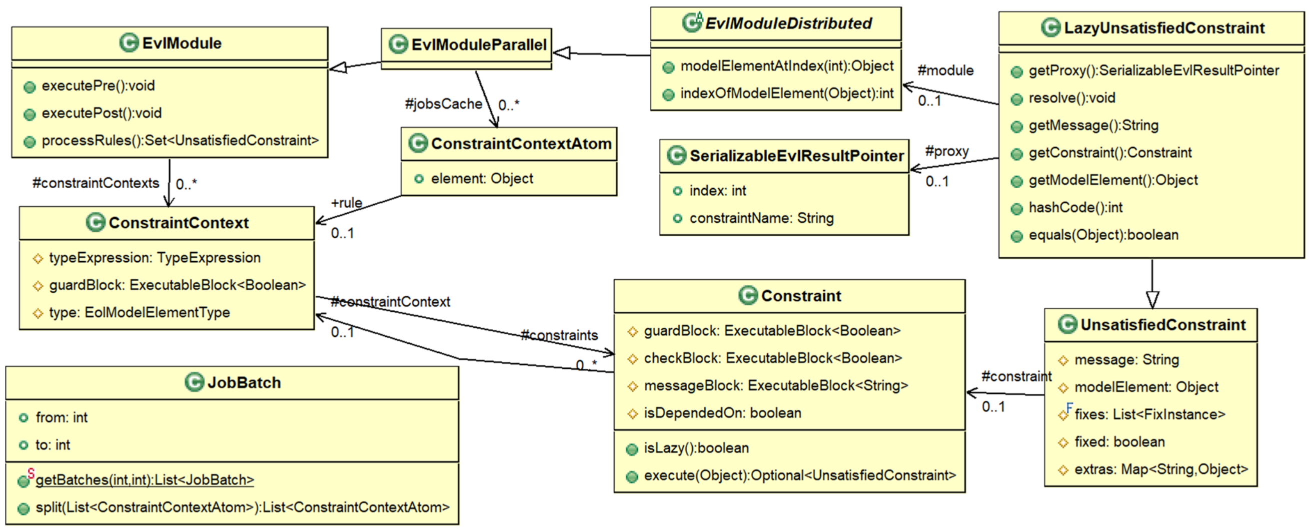
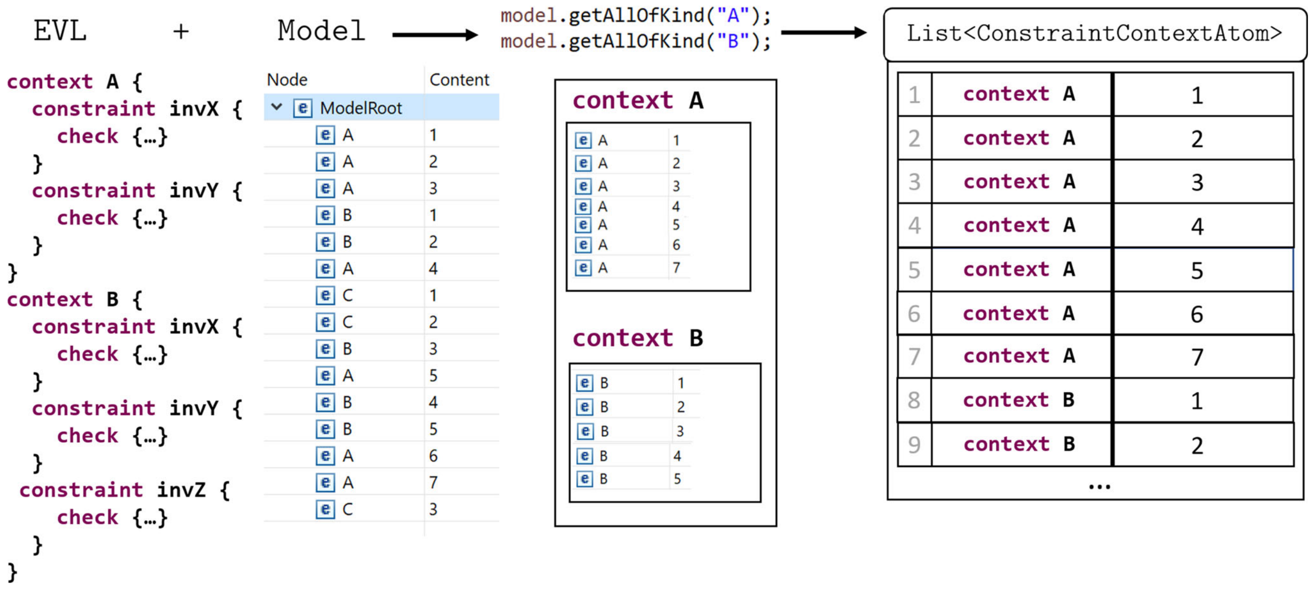

EvlModule
executePre():void
executePost():void
processRules():Set<UnsatisfiedConstraint>
EvlModuleParallel
EvlModuleDistributed
modelElementAtIndex(int):Object
indexOfModelElement(Object):int
LazyUnsatisfiedConstraint
getProxy():SerializableEvlResultPointer
resolve():void
getMessage():String
getConstraint():Constraint
getModelElement():Object
hashCode():int
equals(Object):boolean
#module
0..1
#jobsCache
0..*
ConstraintContextAtom
element: Object
SerializableEvlResultPointer
index: int
constraintName: String
#proxy
0..1
#constraintContexts
0..*
+rule
0..1
ConstraintContext
typeExpression: TypeExpression
guardBlock: ExecutableBlock<Boolean>
type: EolModelElementType
#constraintContext
0..1
#constraints
0..*
Constraint
guardBlock: ExecutableBlock<Boolean>
checkBlock: ExecutableBlock<Boolean>
messageBlock: ExecutableBlock<String>
isDependedOn: boolean
isLazy():boolean
execute(Object):Optional<UnsatisfiedConstraint>
UnsatisfiedConstraint
message: String
modelElement: Object
fixes: List<FixInstance>
fixed: boolean
extras: Map<String,Object>
#constraint
0..1
JobBatch
from: int
to: int
getBatches(int,int):List<JobBatch>
split(List<ConstraintContextAtom>):List<ConstraintContextAtom>

Fig. 2 Highly simplified class diagram of relevant EVL data types



```
context A {
  constraint invX {
    check {…}
  }
  constraint invY {
    check {…}
  }
}
context B {
  constraint invX {
    check {…}
  }
  constraint invY {
    check {…}
  }
 constraint invZ {
    check {…}
  }
}
```

Node	Content
A	1
A	2
A	3
B	1
B	2
A	4
C	1
C	2
B	3
A	5
B	4
B	5
A	6
A	7
C	3

1	context A	1
2	context A	2
3	context A	3
4	context A	4
5	context A	5
6	context A	6
7	context A	7
8	context B	1
9	context B	2

Fig. 3 Atomic decomposition illustration. The numbers identify the order of each element for a given type

Algorithm 2 Simplified EVL job creation algorithm

```
jobs = new ArrayList
for all cc in ConstraintContext do
    for all element in cc.getAllOfKind() do
        atom = new ConstraintContextAtom(cc, element)
        jobs.add(atom)
```

this level of granularity is unnecessary as it imposes extra communication overhead based on experiments. Given that problems with scalability are rooted in the number of model elements, we focus on a data-parallel approach.

Now that we have a fine-grained atomic unit of computation, the next challenge is to distribute these atoms, which requires serializability. Serializing the entire EVL program is non-trivial, since the *ConstraintContext* DOM element has references to its parent and children, and the model element could potentially have transitive references to the entire model. Rather than serializing the entire object graph, or sending the program's resources directly to workers, a more scalable approach is needed. Our solution exploits the consistent nature of the execution algorithm to refer to units of computation by a proxy.

Algorithm 2 shows how the deterministic order of jobs can be exploited. Assuming that *getAllOfKind()* (which gets all model elements that are of the type and its subtypes of the ConstraintContext as specified by its name) returns a deterministically-ordered iterable from the model, we can then refer to jobs by their indices, since the number of jobs and their order is known prior to distribution. Thus, the only information we need to distribute is a collection of *batches*, where each batch consists of a start and end index in the list (see *JobBatch* in Fig. 2). This approach allows us to distribute multiple jobs with minimal overhead, as it is far more efficient to serialize two integers than Strings or other serializable forms. This requires each worker to have pre-computed the jobs (i.e. every pair of model element type and model element applicable to that type) as shown in Algorithm 2; which not only requires the full program but also the model(s) to be available.

Next, we need to consider the results. The outcome of executing a ConstraintContext on a model element is a collection of *UnsatisfiedConstraint*s. Recall that each ConstraintContext contains one or more *Constraint*s. Executing a Constraint for a given element returns the result of the *check* block or expression, which will be a Boolean. Internally however an UnsatisfiedConstraint object is created and added to the results set if the *check* block returns false. An UnsatisfiedConstraint basically consists of the Constraint, the model element and a message describing the failure. We describe the serialization strategy in detail in a subsequent section.

6.2 Preparation

Distributed execution begins from the master[3]; which co-ordinates execution and sends jobs to workers. We assume that all workers have access to the necessary resources (i.e. the same resources available to the master): the EVL program (and any of its dependencies, such as other EOL programs), models and metamodels.[4] Whilst initially it may appear straightforward to co-ordinate distributed execution when every participating node has its own local copies of all resources, the sequence of execution events needs to be revised compared to single-node EVL to maximise efficiency. This is because preparing execution of constraints requires parsing the script, loading the models, constructing the jobs list and executing the *pre* block. Whilst this is straightforward when dealing with a single shared-memory program, with multiple processes (and in fact, multiple computers in this case), the same steps must be repeated for all participating processes (we use the term nodes, processes and workers interchangeably). Furthermore, additional steps must be taken to co-ordinate the execution between workers and the master, which are heavily implementation-dependent. The order of steps in preparation for distributed execution are as follows:

1. The master must establish communication channel with workers[5] (and vice-versa)
2. The master must send program configuration (i.e. script, models etc.) to workers
3. The master and all workers must perform the following steps:
   (a) The EVL script (program) must be parsed
   (b) The model(s) must be loaded (if appropriate, depending on EMC driver[6])
   (c) The execution engine must be initialized
   (d) The jobs list (ConstraintContext-element pairs) must be computed (see Algorithm 2)
   (e) The *pre* block must be executed[7]
4. All workers must report to the master when they are ready to begin processing jobs.

[3] This is a single-master architecture.

[4] Ensuring that workers have copies of the necessary resources is trivial and not a focus of this paper.

[5] Each worker is a separate machine.

[6] The Epsilon Model Connectivity layer (EMC) is the abstraction which allows Epsilon's languages to be independent of specific modelling technologies.

[7] The *pre* block may contain arbitrary imperative code and is executed sequentially on each worker independently.

The crucial part here is (3)—that worker processes need to essentially replicate the potentially expensive process of loading the program configuration. Since workers need to perform all the steps in (3), and the configuration is known in advance, it makes sense to perform this simultaneously on the master and workers. This means rather than the master loading the configuration, then sending it to workers, then waiting for them to load it, both the master and workers can perform these in parallel to each other to avoid effectively doubling the time spent loading the configuration. Note that jobs can be sent to a worker as soon as it and the master are ready (i.e. have loaded the configuration), so we are not bottlenecked by the slowest worker in the group.

Since all workers have access to the required resources (i.e. their own local copy), the master needs only to send pointers to these resources. Specifically, this (serializable) configuration data consists of all of the information required to instantiate the execution engine, such as: the path to the validation script, the model URIs and properties, additional variables to be passed to the script, where to log the output (results) and profiling information to and any optional flags for the execution engine, as well as internal configuration like how many threads to use when evaluating jobs in parallel. The serialization of this information into key-value pairs is relatively straightforward, however since workers may be heterogeneous in their environment (e.g. directory structures), some substitution is required for paths. Note that the distribution framework is oblivious to the hardware, given our solution is Java-based.

6.3 Job granularity

In our index-based approach, a trade-off exists between granularity[8] and throughput. This is especially important when the computational expense of each job can vary greatly. This is because invariants (Constraints) can be arbitrarily complex and navigate any properties of a given element. In this regard, smaller batches allow for greater control with regards to load-balancing, so if a worker receives a few very demanding jobs, the rest can be distributed to other workers. With larger batches, the work may be distributed unevenly, with the worst-case scenario leaving all but one of the workers idle. On the other hand, if the computational complexity (time-wise) is fairly uniform across jobs, then larger batches place less load on the underlying distribution technology and network, which means communication is less likely to become a bottleneck.

At this point it should not be surprising that the batch granularity is an important parameter which can have a massive impact on the effectiveness of the distribution (and hence performance). To easily set this parameter in a normalised way, we define a *batch factor*, which may be a percentage or absolute value. When set as a percentage (i.e. a number between 0 and 1) where 1 means one batch per job (i.e. each batch refers to a single job in the list) and 0 means all jobs are a single batch (i.e. there is one batch with start index 0 and end index the number of total jobs). The split is performed by creating sublists of indices where the granularity of each batch (i.e. the difference between the *to* and *from* indices) is $Nj * ((1 - b)/w)$ (except for the last batch, which may be smaller), where Nj is the total number of jobs, b is the batch factor and w is the number of workers. If this number (i.e. b) is greater than or equal to 1, then a fixed batch size (i.e. b) is used. By default, b is set to the number of logical cores on the master. This maximises CPU utilisation at the finest level of granularity. The rationale for this is discussed in the next subsection. Ultimately, the purpose of the batch factor is to minimise the difference in time between the first and last worker finishing their workload by maximising core utilisation on each worker.

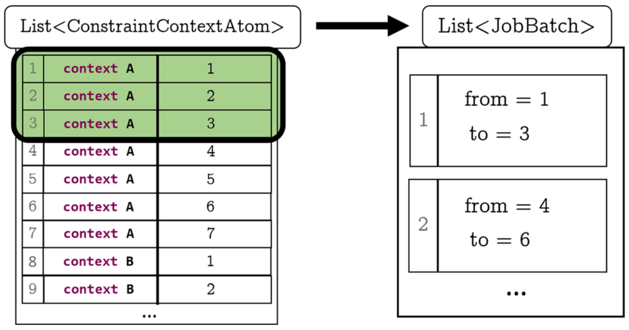


Fig. 4 Illustration of job batching with batch factor of 3

A demonstration of the mapping from the deterministic jobs list to batches is shown in Fig. 4, where the size of each batch is 3 jobs.

6.3.1 Local parallelisation of jobs

When a worker receives a job, it can execute it in parallel using the parallel execution capabilities discussed in Sect. 4 (and which are now part of Epsilon). The level of parallelism depends on the number of jobs received, which is determined by the granularity of batches. For example, if a worker receives a batch with index from 0 (inclusive) to 8 (exclusive) and has eight logical cores, it can perfectly map each ConstraintContextAtom represented by each index to a separate thread and evaluate them in parallel.[9] If however it receives a batch containing one job (e.g. index from 0 to 1), then execution will effectively be single-threaded, since the job

[8] By “granularity”, we mean the difference between end index and start index of a batch; i.e. its size.

[9] Load balancing of local jobs is performed by a thread pool executor service.

received represents only a single atomic unit of work. Note that this assumes the distribution framework waits for jobs to complete before sending them the next; i.e. that workers “pull” jobs from the work queue. If this is not the case, then that means the distribution framework does not perform any load-balancing since it has no information on which workers are “busy”, so job execution will be asynchronous and parallel.

For simplicity, we assume the distribution framework only sends one job at a time to workers and waits for the results before sending another. Hence, parallelisation within workers is only made possible by the fact that each batch represents multiple jobs (if the batch size is greater than 1). The corollary of this is that the optimal batch size is equal to the maximum number of hardware threads (logical cores) on all participating nodes, as mentioned in the previous subsection. Note that a batch size greater than the number of cores still results in efficient utilisation, as the job scheduling within nodes is handled by the thread pool executor service, as in the parallel case. However, a batch size too large means work may be less evenly distributed, since each batch represents a larger proportion of the jobs list. We show how this can be partially mitigated in Sect. 6.4.1.

If a machine receives only C jobs at a time, and has C logical cores, then some jobs may finish faster and so the cores may go underutilized. Whilst this is a possibility when the network is heavily congested, in practice our distribution algorithm ensures the minimal amount of data is sent, so unless the computer running the messaging broker is particularly slow at dispatching jobs, this shouldn’t be a bottleneck.

6.4 Job assignment

It is worth noting that the master itself is also a worker, since it has already loaded the configuration required to perform computations. In most cases, we can treat the master just as we would any other worker, except there is no need to reload the configuration.

If the number of workers is known in advance, we can perform a further optimisation, since it does not make sense to serialize jobs and send them to itself, or indeed to serialize the results of jobs executed on the master. In this case, execution of some jobs on the master works outside of the distribution framework. It is worth bearing in mind that unlike most distributed processing tasks, in our case the workload is finite, ordered and known in advance and hence, we can perform further optimisations such as assigning a certain number of jobs to the master directly. We can distinguish between master and worker jobs where the ratio is statically assigned when starting the master.

In general, the master proportion should be set according to the relative strength in performance of the master and workers, based on the CPU. The background tasks of masters and workers should also be taken into account, since this will impact performance. The network speed and latency will also need to be accounted for—with slower networks or more distant computers, a greater proportion should be assigned to the master. Assuming the master and workers are identical in their compute performance, a sensible default proportion of jobs to statically assign to the master is $1/(1 + number of workers)$. If the master is significantly more powerful than workers, the proportion of jobs assigned to the master should be increased, bearing in mind the co-ordination overhead incurred on the master and accounting for whether the broker is hosted on the master or a different machine.

To be clear, the difference between static and dynamic assignment of jobs on the master is that the former operates outside of the distribution framework, whilst the latter allows for load balancing. In both cases, no separate worker process is spawned on the master, so all resources, state, memory etc. are reused.

Regardless of how jobs are distributed amongst workers and the master, both sets of jobs need to be executed, which can be performed in any order. The execution of the master’s jobs are performed asynchronously (i.e. in parallel) to worker jobs. Since we have two sets of jobs to execute and require both to complete but be processed independently of each other, there is some co-ordination required. Here we can take advantage of Java’s asynchronous execution utilities to pipeline these two tasks and wait for both to complete.

Worker jobs (i.e. those which go through the distribution framework) are dynamically assigned. The exact worker a job ends up executing on and the granularity of jobs is implementation-specific. We assume that the distribution framework automatically performs load balancing, so that jobs are assigned efficiently, maximising aggregate CPU utilisation across all workers. Note that if the number of workers (or master proportion) is unspecified, then the master itself becomes a regular worker (i.e. it operates within the distribution framework), except that it does not reload the configuration: all state is shared. The master executes jobs locally when it receives them from the distribution framework, so jobs are dynamically load balanced between workers and the master. The implementation becomes more complex however, depending on the facilities offered by the distribution framework. For instance, it may not be necessary to send back results from the master’s “local worker” (which is basically a “job receiver”) to the master, since the master’s worker operates within the same process as the master.

6.4.1 Shuffling

Regardless of what distribution framework is used, our objective is to maximise core utilisation on workers (and the master), with ideally maximum CPU usage at almost all times on all workers (and the master) until execution is complete. We have seen how, in the absence of knowledge about the computational complexity of a given job—as this can vary depending not only on the constraints but also on the values of the individual model element. However we cannot assume that the computational cost of jobs is uniformly distributed across the job list (cf. Algorithm 2). For this reason, we shuffle the batches on the master prior to distribution. The effectiveness of this randomisation again depends on the number of batches. With more fine-grained batches, there will be more distributed jobs to shuffle. With fewer but larger batches, randomisation has less of an effect and could actually make matters worse. Shuffling the batches helps to balance the computational workload so that if expensive jobs are concentrated at particular parts of the list, they are spread out more normally. The only reason to avoid this randomisation is to obtain more consistent results or for debugging, since the order in which jobs are distributed will be deterministic.

6.5 Processing results

For simplicity and compatibility with all use cases, we collect the results of evaluating each Constraint in full on the master. This preserves compatibility with existing EVL implementations and user expectations, and also permits the option of writing to a file on the master if needed. An *UnsatisfiedConstraint* consists of a Constraint, the model element for which the constraint was unsatisfied, a message detailing the failure and fixes (which are optional and executed as a post-processing step). Workers need to send back serializable instances of this to the master. As with our index-based job distribution strategy, our solution further exploits the deterministic ordering of jobs for results, although it is slightly more complex.

Since the execution of each ConstraintContextAtom (i.e. ConstraintContext-element pair) may output multiple UnsatisfiedConstraints (because each ConstraintContext may have multiple Constraints), we need a way to uniquely identify each Constraint and the element. We create a new datatype containing the names of the Constraints along with the position of the ConstraintContextAtom in the job list, so the UnsatisfiedConstraint scan be resolved by examining both the name of the Constraint (and its associated ConstraintContext) and the corresponding model element from the ConstraintContextAtom. Furthermore, we don't need to transmit the message of the UnsatisfiedConstraint, since this can be derived on the master from the Constraint instance when required. Whilst the "message" block of an EVL constraint may contain arbitrarily complex code, in most cases the block is a simple String expression, or absent entirely (in which case a default message is generated for a Constraint-element pair). Implementation-wise, we only need to add two methods: one for finding a model element's index (position) in the job list during the serialization process (performed on workers), and one for finding an element by its index during the resolution process (performed on the master). Furthermore, resolution of the UnsatisfiedConstraint is performed lazily, since for example computing the message can potentially be expensive. In the absence of a message block, the practical performance difference between lazy and eager resolution is likely to be negligible, since looking up an item in an ArrayList by index is an O(1) operation.

Depending on the implementation technology (i.e. the framework used for realising our distributed approach), it may be necessary to return a result from the execution of each job even if there are no UnsatisfiedConstraints—for example, as an acknowledgement that the job has been processed or if the implementation performs a *map* operation which requires a result.

6.5.1 Dealing with dependencies

Although each job is unique in that it is only sent and processed once, there may still be duplicate results (unsatisfied constraints) due to dependencies. For example, if constraint C1 depends on constraint C2 and C2 is not lazy, then one worker (W1) may get a job to execute C1, and another (W2) may get one to execute C2. W1 will also execute C2, and if C2 is unsatisfied this will be returned. Since each worker's state is independent of other workers, constraints with dependencies must be evaluated on the worker that received the job (but can be cached thereafter on the worker). It is the master's responsibility to filter duplicates; which is usually trivial since a Set is used for the unsatisfied constraint instances. One issue which arises from dependencies on workers is that if a constraint has one or more dependencies, then any unsatisfied constraints encountered during execution must also be reported as part of the result of the job; unless they have already been evaluated for the given element and sent to the master. To effectively handle this, we reset the set of UnsatisfiedConstraints each time we receive a job, since workers do not need to persist such information. Once execution of a job is complete, we can then serialize the results. This works because evaluation of constraints automatically adds UnsatisfiedConstraints to this collection, rather than returning them as a result.

It should be noted that the set of UnsatisfiedConstraints is independent from the *ConstraintTrace*; which is used to keep track of which constraints have been evaluated and their results. The constraint trace is not shared between nodes, though it shared between threads within nodes. This trace is only written to and read from for constraints with dependencies. The optimisation details of this are outlined in [21]. A consequence of this is that a given constraint-element pair will not be evaluated repeatedly and thus not added to the set of UnsatisfiedConstraints more than once, and hence only sent to the master once from each worker in the worst-case scenario.

To clarify, in our implementation no communication happens between workers: if a dependency exists between constraints, the dependencies are executed separately on each worker when required.

6.6 Alternative serialization strategy

The index-based distribution strategy we have presented relies on deterministic ordering, however not all modelling technologies return elements in order. Our alternative design thus relies on a different serialization mechanism. A notable feature of Epsilon is that its languages are decoupled from modelling technologies through a model connectivity interface (EMC). This API supports various features, one of which is the ability to retrieve model elements by an identifier (ID) and vice-versa.

Our alternative serialization strategy uses IDs to refer to model elements, and Strings to refer to model element types and Constraints. This information can be trivially serialized and used to distribute both jobs and results. However this approach is much more expensive than our index-based one in every measurable way. Not only is there more network traffic and serialization overhead (due to the larger objects), but also more expensive to resolve each job / result. Whilst finding Constraints by their name is relatively inexpensive, the main expense comes from finding elements by ID (and in some cases, computing the ID). Also, this feature is not supported by all modelling technologies. In our experiments with in-memory EMF models, resolving elements by ID turned out to be quite expensive. This is where lazy resolution of unsatisfied constraints on the master becomes even more crucial.

7 Implementation

In this section, we describe our reference implementation for Distributed EVL using the Java Message Service (JMS) API. This is a custom, hand-coded solution which is more low-level than what would usually be required to develop such a distributed system. Nevertheless, it allows us to explore in greater detail the communication and also grants greater flexibility, enabling optimisations which would not be possible with off-the-shelf distributed processing frameworks such as Apache Flink. Our implementation is open-source on GitHub [24].

7.1 Java messaging service

JMS[10] is an API specification for enterprise Java applications designed to facilitate communication between processes. The API revolves around *Message*s, which are created by *Producer*s, sent over-the-wire to *Destination*s and processed by *Consumer*s. A Message has metadata, a body (which can be any Serializable object) and can any number of serializable properties. The API supports both synchronous and asynchronous messaging, along with both point-to-point and publish-and-subscribe semantics. There are two types of Destinations; *Queue*s (point-to-point / exactly-once delivery) and *Topic*s (pub-and-sub / broadcast). Communication is administered by a broker service which implements the API and provides a *ConnectionFactory* which each JVM can use to connect to the broker. For our implementation, we use Apache ActiveMQ Artemis[11]; a JMS 2.0 compliant broker.

7.2 Message-based architecture

Our implementation follows a standard master-worker architecture. Figures 5 and 6 illustrate the communication between the master and workers. Note that in Fig. 5, the communication channels are asynchronous and reactive, with the direction of arrows signifying data flows. Figure 6 illustrates the order of events, although it should be noted that the timeline of workers and the master are separate due to the asynchronous design.

The master is started in the usual way (e.g. through a UI or command-line) with all of the parameters specified. The additional arguments are the URL of the broker (communication protocol is TCP) and a session ID, which is used to uniquely identify this invocation of the program and avoid receiving irrelevant messages by appending it to the names of Destinations. The master is also told how many workers it expects so that jobs can be divided more evenly. Workers are started with only three arguments: the address of the broker, the session ID and base path for locating resources. The order in which workers and master are started does not matter. When a worker starts, it announces its presence to the *registration queue*, sending a message containing its *TemporaryQueue*; a unique queue for the worker. It then waits for a message to be sent to this queue. When the master starts, it loads the configuration and creates a listener on the *results queue* for processing the results. It then creates a listener on the *registration queue*. When it receives a message, it increments a counter for the connected workers, creates a Message containing the configuration along with the worker's ID based on the number of connected workers, attaches the master's

[10] docs.oracle.com/javaee/7/api/index.html?javax/jms/package-summary.html.

[11] activemq.apache.org/components/artemis.

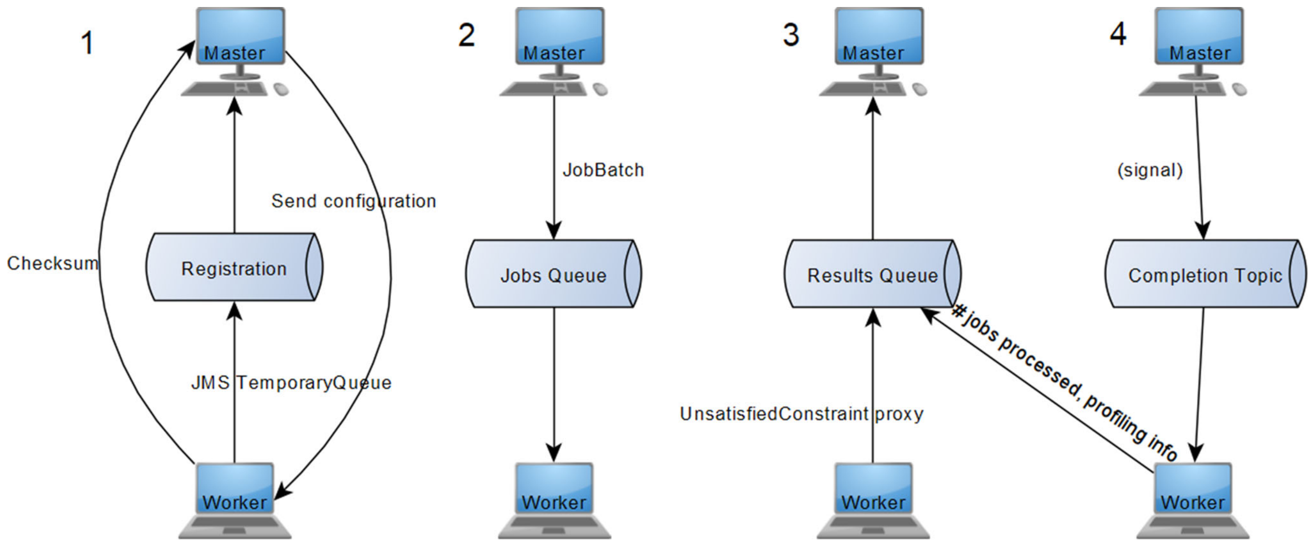
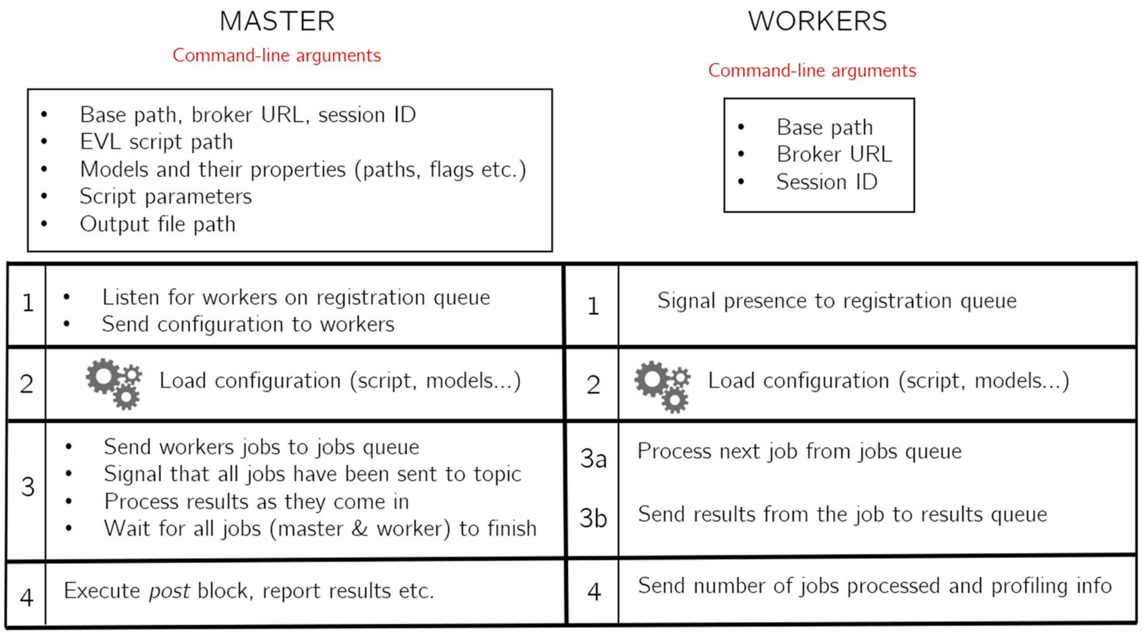


Fig. 5 Summary of communication channels in JMS implementation



MASTER

Command-line arguments

- Base path, broker URL, session ID
- EVL script path
- Models and their properties (paths, flags etc.)
- Script parameters
- Output file path

WORKERS

Command-line arguments

- Base path
- Broker URL
- Session ID

	MASTER		WORKERS
1	• Listen for workers on registration queue • Send configuration to workers	1	Signal presence to registration queue
2	Load configuration (script, models...)	2	Load configuration (script, models...)
3	• Send workers jobs to jobs queue • Signal that all jobs have been sent to topic • Process results as they come in • Wait for all jobs (master & worker) to finish	3a	Process next job from jobs queue
		3b	Send results from the job to results queue
4	Execute *post* block, report results etc.	4	Send number of jobs processed and profiling info

Fig. 6 Summary of asynchronous communication steps

TemporaryQueue and sends it to the *TemporaryQueue* of the received message. When a worker receives this message, it loads the configuration and takes its *hashCode*. It then creates a listener on the *jobs queue* and sends the *hashCode* of the configuration in a message to the master's *TemporaryQueue*. The master checks the *hashCode* is consistent with its own for the configuration and increments the number of workers which are ready. Jobs can be sent to the *jobs queue* at any point once the master's listener on the *results queue* is set up. In our implementation, we wait for at least one worker to be ready before sending jobs to the jobs queue, because otherwise messages may not be processed depending on the broker settings.[12]

If the number of workers is known in advance (or the master proportion parameter is specified), the proportion of work assigned to the master is given by the formula $1/(1 + Nw)$, where Nw is the expected number of workers. If the master proportion or number of workers is unspecified, then the master creates a special worker which bypasses the initial registration step. This worker shares all state with the master,

[12] The "Dead Letter Queue" may be unset, so messages are only sent once with no attempts to re-send them.

and so does not send anything to the results queue since any unsatisfied constraints are automatically added to the results as a side effect of execution. Consequently, jobs are dynamically load balanced between master and workers, as previously described.

Once all jobs have been sent, the master sends an empty message to the *completion topic* to signal the end of jobs. A worker terminates once its jobs queue is empty, there are no jobs in progress and the end-of-jobs message has been received. Before terminating however, a worker needs to tell the master that it has processed all of its jobs, so that the master can also terminate once all workers have completed. Once there are no more messages left in the jobs queue, each worker sends a final message to the *results queue* with a special property set to signal completion, along with the number of jobs it has processed. It also sends its profiling information for the jobs it executed (which consists of a cumulative sum of CPU time consumed by each Constraint) to the master. The master then merges all of these execution times together to produce a final report.

7.3 Exception handling

Failures in execution are inevitable. Whether they are the fault of users (e.g. in the script by navigating null properties, referencing non-existent variables etc.) or unrecoverable, low-level faults with hardware, network, out of memory errors, etc. it is reasonable to expect a distributed system to be able to deal with the former. In the event of an exception, execution should halt and all workers as well as the master should stop. The master should report the exception, along with the stack trace / traceability information, in the usual way (i.e. equivalent to the single-node version of EVL).

To facilitate this, we create a *termination topic* which all workers listen to. When a message is sent to this topic, the workers terminate; though this can only be signalled by the master. When encountering an exception on a worker, we send the job that caused the exception back to the master via the *results queue*, along with the exception. The worker continues processing other jobs as normal. When the master receives this message, it adds the job to a collection of failed jobs. Depending on the nature of the failure, it can either try to execute the job locally or, if the exception was due to user error (a problem with the script), it produces a stop message and sends it to the *termination topic* to stop all workers. The master then stops executing its own jobs and reports the received exception.

7.4 Termination criteria

Another concern is how to determine when all jobs are processed, since the master sends jobs to workers and processes results asynchronously in different threads. When each worker finishes, it sends back to the master the number of jobs it has processed, along with profiling information for the rules. The master also keeps track of how many jobs it has sent to the jobs queue. Each time a worker signals to the completion topic, the master increments the total number of jobs processed by workers, and when this value is equal to the number of jobs sent, it assumes completion. Another possible strategy is to require a response from workers for each job sent, but this incurs greater overhead both at runtime and during development, since it is on a per-job (per-message) basis. Our solution is arguably simpler and only requires a one-time response from each worker.

8 Evaluation

We aim to evaluate the performance of our solution by comparing it to the status quo. Thus, the metric we focus on is *speedup*; i.e. how much faster it is compared to the baseline sequential execution algorithm. However, to truly assess the efficiency of our approach, we also put the speedup into context by accounting for the number of processing cores.

Since our distribution strategy does not perform any static analysis on the program or model to intelligently partition the workload, each constraint is treated as a "black box". Therefore, we aim to assess how our solution copes with the worst-case scenario, where a single constraint dominates the execution time.

We used the models and validation programs in [21]. This allows for better reproducibility and comparability of our results. However, since the codebase of Epsilon has evolved and we are using different hardware, we replicate the experiments in [21] with the latest version of Epsilon, EMF 2.15 and Eclipse OCL 6.7.0. Our main focus is on the *findbugs* script, which is inspired by the open-source project for detecting code smells in Java. We chose this scenario because the MoDisco Java metamodel is very large and complex, exercising almost all features of Ecore. There are 30 invariants in the EVL program which exercise different parts of Java models as obtained by reverse engineering the Eclipse JDT project. The resources for our experiments and results are available on GitHub [25].

Another option we considered for evaluating the performance of our solution was the Train Benchmark model generator [26]. We opted for the Java/Modisco-based scenario instead as it involves a larger and more complex metamodel and a larger set of constraints (the Train Benchmark paper describes only six, contrived, constraints).

8.1 Experiment setup

We use one system as a baseline for all benchmarks, and up to 87 workers when running Distributed EVL. All systems were on the same wired (Ethernet) network in the same building. Their specifications are as follows:

- 1× AMD Ryzen Threadripper 1950X 16-core CPU @ 3.6 GHz (in "Creator Mode"), 32(4x8) GB DDR4-2933 MHz RAM, Fedora 30 OS (Linux kernel 5.1.12), HotSpot 11.0.2 JVM.
- 1× Intel Core i7-4790K quad-core CPU @ 4.00 GHz, 16(2x8) GB DDR3-1600 MHz RAM, Windows 10 v1607, OpenJ9 11.0.3 JVM.
- (Remaining) Intel Core i5-8500 hex-core CPU @ 3 GHz (4 GHz turbo), 16 GB RAM, Windows 10 v1803, HotSpot 25.181 JVM.

All workers had all models and scripts required for the experiments on their local disk drives (i.e. we did not use a shared network drive; every worker had all resources it needed locally). Before timing each experiment in distributed mode, we ensured all workers were started first. We used our base Threadripper system as the master node except otherwise stated. We used Apache ActiveMQ Artemis 2.10[13] as our broker with message persistence disabled. When recording execution time, we exclude the loading time for the configuration (i.e. the model and script parsing). We repeated each experiment several times and took the mean average time in milliseconds. We used a maximum JVM heap size of at least 80% of the total memory of each machine.

We are primarily interested in evaluating the efficiency of our distributed approach in terms of execution time. Model loading is a one-off task with a constant cost for a given model (growing linearly at worst with model size), whilst execution time of a complex program may grow exponentially. Since model loading time is constant factor in all cases, we exclude the model loading time from our speedup measurements to simplify the analysis. The loading time measurements in the following tables are measured on the master.

8.2 Parameters

Since our implementation is configurable, we show the results for what we found to be optimal settings. We briefly discuss the chosen parameters for our experiments in this subsection.

8.2.1 Master proportion

In all of our experiments, we know in advance the number of workers, so we always statically assign a proportion of jobs to the master rather than relying on the broker to assign jobs to the master. Since jobs assigned to the master do not need to be serialized or communicate with the broker, they have no overhead and so it is preferable to give bias to the master. More so in our case, since the master has sixteen cores and more memory than the workers, however it is also burdened with deserializing the results and also hosting the broker. Nevertheless, it would be optimal to assign a greater proportion of jobs to the master than workers in this case, so we set a figure of 0.08 (for sixteen workers) which is slightly above the 0.0588 figure as would be assigned by the default formula. We did not want to set this figure too high as we are interested in testing the performance gains from distributed processing and not necessarily the optimal for this specific scenario.

8.2.2 Batch size

As discussed previously, there is a tradeoff between granularity and throughput. In practice, so long as the size of each batch is greater than or equal to the number of cores, the broker should be more than capable of handling such loads, since each batch only consists of two integers, and the result format is also index-based. In our experience, the CPU usage of the broker never exceeded 2%. If setting the batch factor as a percentage, we found this is best kept close to one, especially for unevenly distributed workloads. This parameter should be proportional to the model size, as larger models will result in bigger batches otherwise. For the models in our experiment, we found a figure of between 0.9992 and 0.998 works well, keeping all machines at between 95 and 100% CPU usage when they were active, and the time between the first worker finishing and the last being one or 2 min. A higher percentage (i.e. more batches) yields more consistent performance at the expense of greater network traffic. We conducted further experiments where we set the batch size to the default value (i.e. the number of logical cores in the master), which yielded more consistent results with all workers and the master finishing within a few seconds of each other. Hence, our recommendation for the default batch factor is to set it as an absolute value equal to the highest number of logical cores of all computers in the experiment.

8.3 Results and analysis

Table 1 shows the results for *findbugs*, with speedup relative to sequential EVL. Note that the relatively small execution times, and in some cases this is less than the model loading time. We see that even when execution times are small (measured in seconds, rather than hours), our distribution strategy still provides large gains when excluding loading times. However if we factor in the overhead of model loading on workers and the master, the gains in absolute terms are more negligible, since model loading becomes the main bottleneck. In such cases local parallelisation may be more practical, which in this case is ten times faster than interpreted

13 activemq.apache.org/components/artemis.

Table 1 Results for simplified *findbugs* script with 16 workers

Model	Implementation	Load	Execute	Speedup
1m	Interpreted OCL	7686	42,803	0.474
1m	Compiled OCL	8312	8871	2.289
1m	Sequential EVL	9354	20,308	–
1m	Parallel EVL	9317	4966	4.089
1m	Distributed EVL	9058	6777	3.000
2m	Interpreted OCL	15,769	86,659	0.447
2m	Compiled OCL	16,774	16,993	2.280
2m	Sequential EVL	18,132	38,741	–
2m	Parallel EVL	18,179	8637	4.485
2m	Distributed EVL	18,371	8299	4.668
4m	Interpreted OCL	31,856	145,540	0.422
4m	Compiled OCL	32,337	29,928	2.054
4m	Sequential EVL	35,294	61,482	–
4m	Parallel EVL	34,623	14,541	4.228
4m	Distributed EVL	34,751	13,736	4.476

OCL. Despite an additional 94 processor cores, distributed EVL is only able to narrowly beat parallel EVL when there are over 2 million model elements.

8.3.1 Adding a demanding constraint

Since the execution times are small in absolute terms, we also added a very demanding constraint to the *findbugs* script, which is responsible for approximately 99% of the execution time. The invariant in question is shown in Listing 3. It is an inefficient algorithm for ensuring that all imports in all Java classes are referenced within the class. Since there are many imports, and for each instance almost every model element is looped through, there are an exponential number of computations and reference navigations performed, even though the logic is very simple.

Listing 3 Most demanding invariant

```
context ImportDeclaration {
 constraint allImportsAreUsed {
   check : NamedElement.allInstances()
     .exists(ne | ne == self.importedElement
        and ne.originalCompilationUnit
        == self.originalCompilationUnit
     )
  }
}
```

We repeated the experiments in the previous subsection (with identical parameters) with this constraint added to assess how scalability varies; especially given the imbalance in execution times between jobs. The intention is to test how well random assignment works in tandem with our batch-based approach. The results are shown in Table 2.

Table 2 Results for *findbugs* script with 16 workers

Model	**Implementation**	**Execute**	**Speedup**
1m	Sequential EVL	2,478,312	–
1m	Parallel EVL	300,494	8.247
1m	Distributed EVL	45,189	54.843
1.5m	Sequential EVL	5,110,216	–
1.5m	Parallel EVL	594,846	8.591
1.5m	Distributed EVL	83,342	61.316
2m	Sequential EVL	9,590,029	–
2m	Parallel EVL	1,158,201	8.28
2m	Distributed EVL	128,664	74.535
4m	Sequential EVL	22,823,792	–
4m	Parallel EVL	2,662,679	8.572
4m	Distributed EVL	316,049	72.216

Although parallel EVL provides a speedup between 8.2 and 8.6× on our 16-core system, distributed EVL varies considerably more, scaling better with model size. At its peak, we see an improvement of almost 75× compared to the single-threaded engine with 2 million model elements. In absolute terms, this means a configuration which took approximately 2 h and 40 min is reduced to just 2 min and 8 s. However we see that the extent of the improvement stops with 4 million model elements. This can be explained by a suboptimal batch factor. Setting this parameter lower would reduce the variance between when workers finish. We investigate the effect of setting the batch factor to a fixed optimal value in the next section.

The potential speedup for distributed EVL compared to sequential is $(6 * 15) + 4 + 16 = 110$ based purely on number of cores. However, the overhead of distribution, lower CPU clock speeds with higher utilisation and the fact that our base system is not the same as our distributed ones should be taken into account. We argue that the speedups achieved are not too distant from the true realistically achievable limit, and that with further tuning of the batch factor and master proportion for each specific scenario could provide greater improvements. However we can see that parallel EVL is bottlenecked—perhaps by memory access—in the *findbugs* scenario. Similar bottlenecks may be present in distributed EVL, since each worker also executes jobs in parallel using the same infrastructure.

8.3.2 Scalability with 87 workers

In Fig. 7 we attempt to assess the potential scalability of our solution with specific tuning parameters. We use 87 workers (a total of 536 cores) with a relatively small batch size of 32, although larger than the number of logical cores on the workers. Nevertheless, we found this configuration to be a good

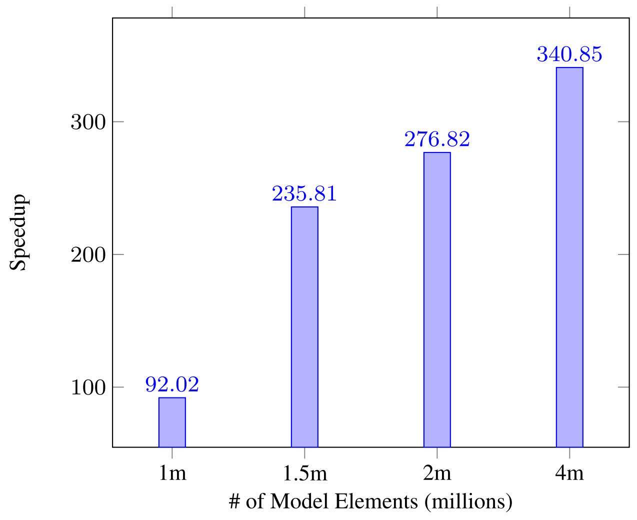


Fig. 7 Results for *findbugs* script with 87 workers (speedup relative to sequential EVL)

balance between minimising communication overhead, maximising CPU utilisation and minimising the time between the first and last worker finishing. We also found that assigning 1.5% of jobs to the master was optimal; even a proportion of 2% left the master finishing much later (in relative terms) than the workers. This is still higher than the default figure of 1.1% though the master is significantly more capable than the workers. With this many workers, the execution time of the script on the largest model is down to an average of just over 68 s, compared to the sequential case of over 6 h and 20 min. However as with the results in Table 1, with smaller models the execution time becomes comparable to the model loading time.

An interesting observation from Fig. 7 is the relatively poor performance for one million model elements. This is actually explained by the large standard deviation times. We found that, with approximately a 50% probability, the execution time is either around 38 s or 12 s (±1 s). This is because for this specific configuration, the batch factor is too large, resulting in one worker finishing significantly later than others in some cases. The fact that execution times are grouped around two results can be explained by shuffling, as it is the only source of randomness.

8.3.3 Single constraint, 2 million elements

Now we take a deep dive, focusing on one specific scenario to further assess the scalability and efficiency of our solution. We used the single most demanding constraint (discussed previously) with the 2 million elements model for this benchmark. Except in the sequential case, we ran all workers and the master with as many threads as there are cores in the system. For this experiment we used only the i5-8500 machines, with the broker on a dedicated system (i.e. not on the master) so that we can better assess the performance delta when adding more machines. The results are shown in Fig. 8. As usual, the parallelism for each participating worker in the distributed implementation was set equal to the number of cores in the system (i.e. 6). Note that the numbers indicate the number of threads in the case of parallel (P) implementation, and the number of workers in the case of distributed (D).

The results in Fig. 8 are inline with our expectations. The parallel and distributed implementations perform almost identically at around 4.7× faster than the sequential implementation. The execution time further decreases as we add more workers, albeit with increasingly diminishing returns. That said, scalability does not sharply decline, so at the top end we still see a significant improvement. With 16 total workers (including the master) each equipped with 6 cores, the maximum theoretical speedup possible is 96×, and our implementation achieves 70×. This shows that our approach has a remarkably low co-ordination overhead in terms of computational cost. We can deduce this based on the efficiency of the parallel implementation: if a single 6-core machine is “only” 4.7× faster than the sequential implementation and we are using the same parallel execution infrastructure in the distributed case, then clearly 96× speedup is unachievable. Based on the results for this experiment, the parallel implementation is 78.9% efficient with a parallelism of 6, whereas the distributed implementation is 72.9% efficient with a parallelism of 96: a mere six percentage point drop in efficiency with a 16× increase in overall parallelism. This drop-off is far lower than it would be if we were to add more cores to the parallel implementation, as evident by the more rapidly diminishing speedups under shared-memory parallelism. It is also worth noting that the parallel implementation algorithm and infrastructure, despite its elements-based level of granularity and all of the underlying complexity in the data structures and thread-locals, is relatively low overhead in terms of performance compared to sequential EVL. Based on these results, running the parallel implementation with a single thread gives an efficiency of 95.7%. This could perhaps be improved further by using more coarse-grained jobs through batching to reduce the scheduling overhead brought about by the (fixed) thread pool.

An interesting takeaway from this experiment is that when the master and workers are using identical hardware, despite the broker being located on a dedicated machine, the master still runs significantly slower than the workers. We know this by examining the output logs which can tell us when a worker or the master finishes processing its jobs and also how many jobs they processed. As the number of workers increases, the default ratio of master to worker jobs combined with the default batch factor ensures a fairly efficient distribution such that workers and the master finish at roughly the same time. However with fewer workers and long-running jobs, the master proportion becomes much more important since there are more jobs at stake, and jobs are statically assigned to the master beforehand rather than being dynamically load-balanced like worker jobs. This was particularly the case for our experiment with 1 and 2 workers, where the difference in execution time between the master and workers was up to 30% of total execution time. This is because the master also has to deal with co-ordination, sending jobs and results processing whilst simultaneously executing its own jobs. With identical hardware, a worker bias is needed, the extent of which will depend on the workload. In this case, we found that setting the master proportion to 25% for 2 workers (as opposed to 33%) reduced execution time by up to 2 min, whilst setting it to around 45% with one worker (instead of 50%) reduced the overall execution time by up to 10%.

So far we have presented two extreme cases: one where execution time is very long due to a single computationally expensive constraint and another where this constraint is removed, making overall execution time relatively small and comparable to the time taken to load the model into memory. We see that the overhead of distribution begins to pay off once execution times are measured in the order of minutes. Overall, we would advocate the use of local parallelisation when execution times are relatively small and the model takes a long time to load, and distributed parallelisation when the program is computationally expensive.

8.4 Single-thread parallelisation

In Fig. 8, we also ran the benchmark with local (shared memory) parallelism set to 1, to be able to compare the effi-

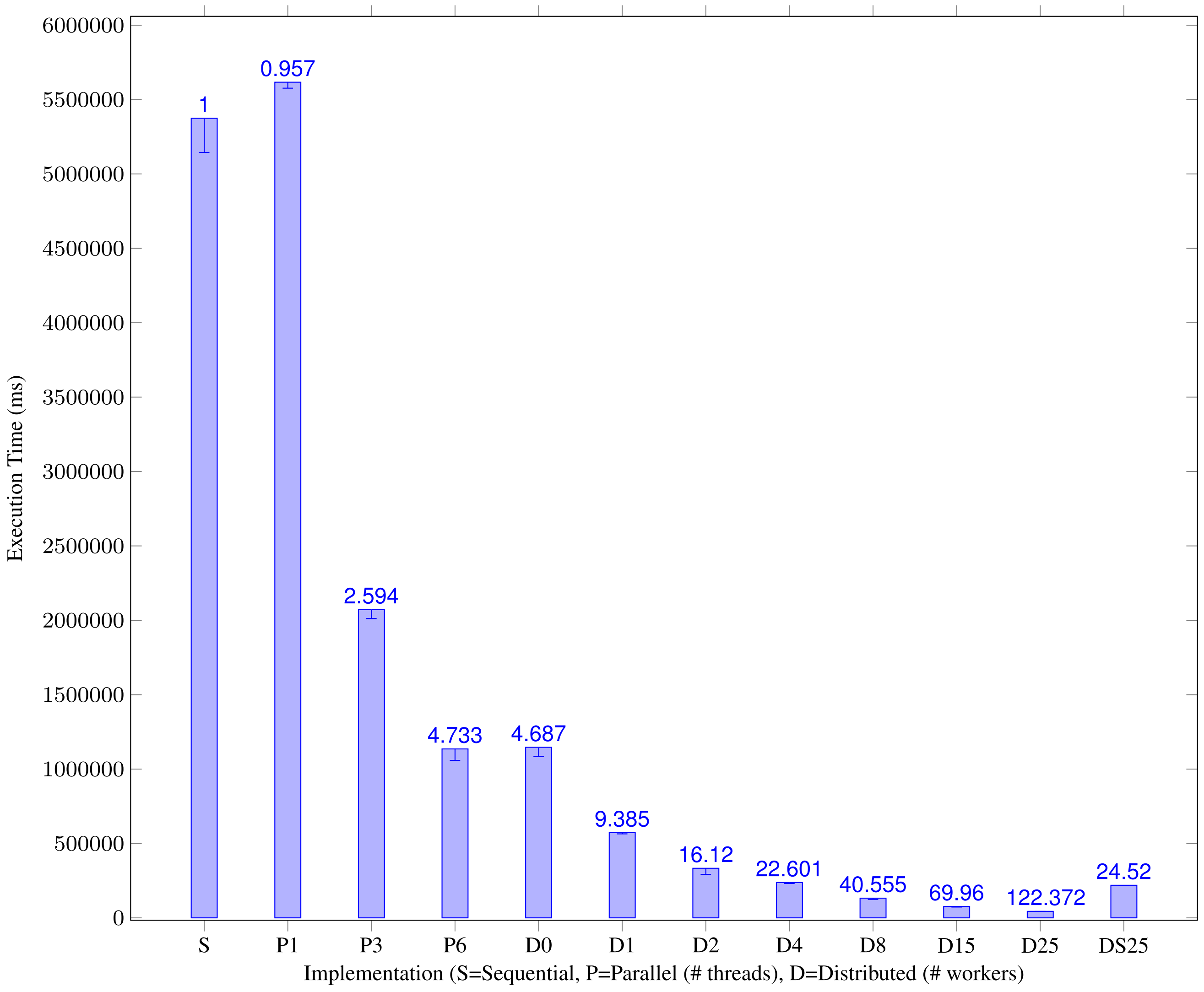


Fig. 8 Results for *1Constraint* script with 2 million model elements, batch size = 6, i5-8500 16GB RAM Windows 10. Labels show speedup. Error bars show standard deviation (one direction only)

ciency of the distributed approach without any parallelisation overhead. We tried this with 25 workers (the result for single-threaded distributed EVL is labelled "DS25") and observed a speedup of 24.5× over sequential EVL: very close to the theoretical limit of 26×. We also ran the benchmark with local parallelism set to the number of cores as usual for comparison. With over 122× speedup out of a theoretical 156×, the efficiency being 78.2% in this case) is on par with our expectations based on previous results. For context, where sequential EVL took over 1 h 30 min, with 25 workers this is reduced to just 44 s. The motivation for this experiment is that now compare single-threaded and multi-threaded distributed execution. Leveraging all six cores on all machines leads to almost exactly 5× speedup compared to the sequential case with the same number of workers. This is slightly higher than the 4.7× speedup provided by parallel EVL over sequential EVL, which requires some explanation. As with all performance-related phenomena, there are a number of contributing factors. Perhaps the major contributor is the fact that because there is only a single constraint, all cores in all computers are executing the same instruction with slightly different inputs (i.e. the model elements), thus the load balancing is near-perfect. Since each computer processes fewer jobs, there is also less total work for each JVM instance (for example, garbage collection) to do, as well as reduced memory contention and even less heat, leading to higher sustained boost clock speeds.

8.5 Local distribution vs. parallelisation

Another way to assess the efficiency of our distributed approach is to see how it compares to parallelisation when controlling for distribution overhead encountered by network communication. Rather than using multiple machines, we can instead use a single computer but spawn multiple processes. Of course, the total level of parallelism will be limited by then umber of logical cores on the machine, so we need to ensure that $(W + 1) * T = C$, where W is the number of processes (independent workers, excluding the master), T is the number of threads per process and C is the number of logical cores (hardware threads) on the computer.

We experimented with the *findbugs* script with 100k, 200k, 500k and 1m model elements on a different system, equipped with a Ryzen 7 3700X 8-core (16 thread) processor, 32 GB dual-channel DDR4-3200 MHz RAM, MSI X570 A-PRO motherboard. Given that each process would load the model and program into memory, as well as needing to host the broker, this limits the maximum model size, since for example with 8 processes we need to have 8 instances of the model loaded in memory simultaneously. We set the batch factor equal to local parallelism: with 1 worker this was set to 8, with 3 workers this was 4 and 7 workers it was 2. Master proportion was set as usual: 0.5 with 1 worker, 0.25 with 3 workers and 0.125 with 7 workers.

The results in Fig. 9 show that for larger models, shared-memory parallelism is significantly less efficient than having multiple independent processes (each with their own isolated in-memory resources) working in parallel. This indicates that the parallel infrastructure is bottlenecked, perhaps by access to a shared data structure (e.g. caches), or perhaps the combination of JVM, CPU and OS scheduling algorithm is to blame. Regardless of the reasons, it's clear that using multiple processes is computationally faster, albeit an unrealistic scenario given the memory requirements. With 1 million elements, the optimal arrangement seems to be 4 process using 4 threads each; giving 7× speedup over the sequential implementation. By contrast, the single-process parallel implementation's speedup is less than 5×. As the model size decreases, so too does the efficiency of both the parallel and distributed implementations. The difference is more drastic with multiple processes however. We can see that with 500 k elements, 4 workers is still the optimum, giving 5.5× speedup, whereas with 200 k elements, the overhead of the distributed implementation means the shared-memory (single process) parallel implementation is superior.

8.5.1 Simulink

There is a unique circumstance where distribution can provide performance benefits where shared-memory parallelisation cannot. With most common modelling technologies, there are no complex concurrency issues to handle since we are only reading from the model, and we already have a solution in place for thread-safe caches. Despite this, some EMC drivers relying on external, often proprietary tooling may not be suitable for concurrent accesses. One such case is that of the Simulink driver for Epsilon [27]. Interacting with Simulink models requires launching MATLAB from Java, as the driver uses MATLAB's Java API for model access. For reasons beyond the scope of this paper, executing an EVL program in parallel over these does not work. Rather than trying to debug the root cause, which is likely to be with MATLAB or its Java API (and thus beyond our control), we can still reduce the execution time using our distribution strategy.

The idea is that since each worker runs in its own process and on a separate computer, it does not share any resources with other instances, therefore any interference or thread-safety issues are impossible provided that there is only a single thread of execution. We have asserted that the Simulink EMC driver returns model elements in a deterministic order for separate invocations for a given model, so our batch-based approach can be used. As usual, the master splits the job list between itself and workers. The difference here is that it also instructs all workers (and itself) to use a local

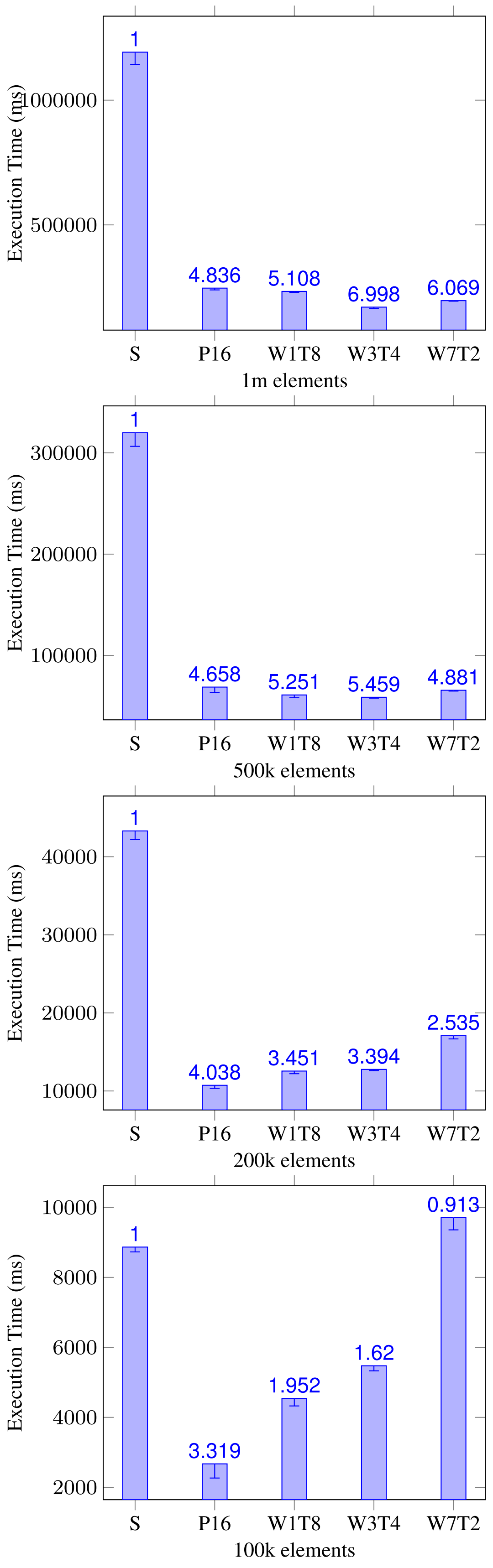


Fig. 9 Results for *findbugs* script with varying model sizes and workers. R7-3700X 32 GB RAM Windows 10 JDK 13.0.1. S=Sequential, P=Parallel, W=Workers, T=Threads per worker. Labels show speedup

Table 3 Results for single-threaded Simulink experiment, batch size = 6

Workers	Load	Execute	STDEV	Speedup
0	33 165	484 535	31 546	–
15	14 399	205 766	17 581	2.355

parallelism of 1 to avoid thread-safety issues. This way, we are able to partition the work, effectively achieving a level of parallelisation equal to the number of workers.

To test this theory, we used the artefacts used in [27] (we have included these in our open-source benchmark repository [25]). There are several models available, however all of them are of similar size and in our experiments their execution times are similar for the provided validation program. We ran the “liveValidation.evl” script (consisting of 9 constraints across 6 model element types) over the “darkd0.slx” model, which is 1.1 MB but for context, when transformed to EMF and serialized to XMI, is 133 MB.[14] Note that we made no attempts to optimize this script or alter it in any way. We ran our experiments using only the worker PCs (i5-8500, Windows 10) for a fair comparison. The MATLAB version was R2018a. For compatibility we used the JRE shipped with MATLAB, which was 1.8.0_181 (HotSpot VM). The master proportion and batch size parameters were set as the default. The results are shown in Table 3.

There are a couple of noteworthy caveats regarding the results. Firstly, despite repeating the experiment ten times, there was significant variance in execution time and model loading time of sequential EVL. Intuitively we would expect to see such variance in parallel and distributed execution, but not with sequential. With model loading in particular saw a standard deviation which was close to the average load time, whilst under distributed execution load times were remarkably consistent (around 13.5 s) with only a single outlier at 21 s. Execution time was also more volatile than we expected, with sequential EVL showing a standard deviation of over 31 s. Furthermore, when examining the execution time of the constraints, we saw a lack of consistency in the dominant rule, which always seemed to vary with sequential EVL. These puzzling results can largely be explained by the interaction between the EMC driver and the MATLAB engine. During execution we noticed that both MATLAB and the EVL Java process were using CPU time simultaneously, each running at near 100% usage for a given core. The MATLAB Java API supports asynchronous access, so perhaps the source of this volatility arises from this communication.

What is more difficult to explain is the exceptionally poor speedup achieved by 15 additional computers. Although the

[14] All of the Simulink models available to us were of approximately equal size; we chose this one at random.

maximum possible speedup is 16, we observed less than a sixth of this theoretical potential, with an efficiency (that is, speedup divided by number of workers) of just 14.7%. This is in stark contrast to our EMF-based experiments where distributed efficiency was in some cases superior to parallel. There are a couple of reasons for this. Firstly, the *pre* block of the validation script took a significant amount of time to execute, averaging around 40 s. Since this contains arbitrary imperative code and is used to set up variables used in the constraints, this must be executed for each worker independently. Secondly, model element accesses are performed lazily on-the-fly by the Simulink EMC driver. Even though we enabled caching of *getAllOfKind*, model element types which do not appear as *ConstraintContext*s in the validation script will not be loaded eagerly. Since our distribution algorithm is essentially random assignment, in practice each worker inevitably ends up accessing all necessary parts of the model. Since model access is the primary bottleneck in the Simulink driver, we see the effects of this dominate the execution time, hence the poor speedup. However, we were still able to more than halve the original execution time with no additional optimisation.

The Simulink experiment exposes the main weakness in our approach: the inability to exploit data locality due to the lack of intelligent assignment and partial loading. Although this is largely an issue at the EMC level rather than an issue with the distribution and/or parallelisation infrastructure, it nevertheless makes the case for more advanced distribution strategies which can take additional factors into account for modelling technologies where model accesses are very expensive.

8.6 Threats to Validity

The performance gains shown in our evaluation cannot necessarily be generalised for all models and scripts, neither can the parameters as they are specific to the scenarios presented. Our experiment used computers on the same network and were physically located in the same building. Furthermore, the workers were homogeneous. Performance may differ greatly depending on the hardware and network topology. However, the lessons learnt should be more generalisable. We have made our implementation's parameters configurable to improve generality, and discussed the significance of these parameters in the general case since, as we have seen, performance is highly sensitive to these parameters. In our benchmarks, we have tried to demonstrate "worst-case" scenarios, and used benchmarks which were designed independently of the work presented. Furthermore, the script and models were written independently from the experiment to avoid bias.[15] Based purely on analysis of the distribution algorithm, we have no reason to believe that the potential performance gains of our solution should not be generalisable to other models and scripts, given sufficiently large models and time-consuming validation logic. As we saw with the Simulink experiment, our solution does not generalise to all modeling technologies. Our solution relies on model access for (random) individual elements being relatively fast; certainly no worse than O(1) in time complexity. Hence, we observed large speedups with in-memory EMF models. Performance may not scale as well for models stored in e.g. databases or file systems. Thus, a degree of in-memory caching may be necessary for optimal performance. Future work should ideally perform experiments with different models and programs than described here, as well as with different modeling technologies which use databases for persistence to assess the scalability of our solution, especially for larger models which cannot be loaded entirely into memory.

9 Conclusions and future work

In this paper, we have demonstrated how the task of model validation can be expressed in a highly parallelisable manner which is amenable to distributed execution using the Epsilon Validation Language. We showed how every EVL program can be decomposed into a finite and deterministically ordered list of rule-element tuples. By exploiting this fact, we can re-create the program execution environment on multiple computers and assign a subset of this list to each computer, referring to jobs by their position in the list. The results show that our index-based distribution approach has minimal overhead, and that the execution time decreases linearly with more computers and larger models. We saw improvements of up to 340× compared to sequential validation with 87 hexa-core workers. This was made possible by our efficient asynchronous implementation, so that all participating computers begin loading the models and program independently from each other. We determined that the optimal granularity of jobs is equal to the maximum number of hardware threads across all participating computers, allowing each computer to fully utilise all cores whilst also allowing for efficient load balancing to be performed by the broker. We established the efficiency of our distributed approach by performing a further experiment with 25 workers each executing sequentially (single-threaded), resulting in a 24.5× speedup. Repeating this experiment where each worker leverages all of its cores, we saw a further 5× speedup, making it over 122× faster than the traditional single-machine, single-threaded execution engine. Overall, we found that our distributed

[15] The models are sourced independently, and the EVL scripts were written years before this work.

execution approach is actually more efficient than shared-memory single-process parallelism, due to alleviation of the von Neumann bottleneck (memory access).

The main limitation of our current approach is that it requires all workers to have a full copy of the models and the program. For very large models, this presents a significant barrier to entry since memory-constrained devices cannot participate in the execution. Such a limitation could be addressed by employing advanced static analysis to optimally distribute jobs based on the parts of the model they exercise, so that workers only need partial models. This could be combined with an efficient model indexing repository such as Hawk [28] to lazily load model elements, thus reducing memory footprint and the upfront temporal cost of model loading. Static analysis could also help to identify computationally expensive jobs which would result in more balanced distribution of jobs between master and workers. Whilst random assignment through shuffling of batches, combined with the load balancing capabilities of the distribution framework provided good results in our experiments with in-memory EMF models, this is not generalisable for all persistence backends, since randomisation minimises data locality. In modelling technologies where model accesses are extremely expensive—such as Simulink—a more intelligent assignment algorithm would be beneficial to exploit data locality and lazy loading. In any case, future work should focus on distribution of the model as well as the computation.

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

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. Cuadrado, J.S., Jouault, F., Molina, J.G., Bèzivin, J.: Optimization Patterns for OCL-Based Model Transformations. In: Models in Software Engineering: Workshops and Symposia (MODELS), Toulouse. pp. 273–284, Springer (2008)
4. Madani, S., Kolovos, D.S., Paige, R.F.: Towards optimisation of model queries: a parallel execution approach. J. Object Technol. **18**(2), 3:1–3:21 (2019)
5. Willink, E.D.: Deterministic Lazy Mutable OCL Collections. In: STAF 2017 Collocated Workshops, Marburg. pp. 340–355. Springer (2017)
6. Cuadrado, J.S.: A Verified Catalogue of OCL Optimisations. Software and Systems Modeling, vol. 19, pp. 1139–1161. Springer, Berlin (2020)
7. Clasen, C., Del Fabro, M.D., Tisi, M.: Transforming very large models in the Cloud: a research roadmap. In: Proceedings of the First International Workshop on Model-Driven Engineering on and for the Cloud, Copenhagen. pp. 3–12, Springer (2012)
8. 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)
9. Bèzivin, J., Jouault, F.: Using ATL for checking models. Electronic Notes in Theoretical Computer Science **152**(1), 69–81 (2006)
10. Object Constraint Language 2.4 Specification, https://www.omg.org/spec/OCL/2.4/
11. Imre, G., Mezei, G.: Parallel Graph Transformations on Multicore Systems. Multicore Software Engineering, Performance, and Tools. LNCS 7303, pp. 86–89, Springer (2012)
12. Krause, C., Tichy, M., Giese, H.: Implementing Graph Transformations in the Bulk Synchronous Parallel Model. In: Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering, pp. 325–339, Springer (2014)
13. Benelallam, A., Gomez-Llana, 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, Pittsburgh. pp. 37–48, ACM (2015)
14. Benelallam, A., Tisi, M., Cuadrado, J.S., de Lara, J., Cabot, J.: Efficient model partitioning for distributed model transformations. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam. pp. 226–238, ACM (2016)
15. Burgeño, L., Wimmer, M., Vallecillo, A.: A Linda-based platform for the parallel execution of out-place model transformations. Inf. Softw. Technol. **79**, 17–35 (2016)
16. Szárnyas, G., Izsó, B., Ráth, I., Harmath, D., Bergmann, G., Varró, D.: INCQUERY-D: A distributed incremental model query framework in the cloud. in: Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems, Valencia. pp. 653–669, Springer (2014)
17. Mezei, G., Levendovszky, T., Meszaros, T., Madari, I.: Towards truly parallel model transformations: A distributed pattern matching approach. IEEE EUROCON 2009, St. Petersburg, pp. 403–410, (2009)
18. 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)
19. 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)
20. 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)
21. 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)
22. 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)
23. 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)
24. Distributed EVL implementation. Available at: github.com/epsilonlabs/distributed-evl
25. Distributed EVL evaluation. Available at: github.com/epsilonlabs/parallel-erl
26. Szárnyas, G., Izsó, B., Ráth, I., et al.: The Train Benchmark: cross-technology performance evaluation of continuous model queries. Softw. Syst. Model. **17**, 1365–1393 (2018)
27. Sanchez, B.A, Zolotas, A., Hoyos, H., Kolovos, D.S., Paige, R.F: On-the-fly Translation and Execution of OCL-like Queries on Simulink Models. In: Proceedings of the 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, Munich
28. Barmpis, K., Kolovos, D.S.: Hawk: towards a scalable model indexing architecture. In: Proceedings of the Workshop on Scalability in Model Driven Engineering, Budapest. Article No. 6 (2013)

Sina Madani obtained his PhD from the Department of Computer Science at the University of York. His research focused on improving the performance of model management programs through parallelisation. He is an Eclipse Foundation committer and contributed extensively to the Epsilon model-driven software engineering platform.

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-driven software engineering. He is also an Eclipse Foundation committer, leading the development of the open-source Epsilon model-driven software engineering platform, and an 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.

Richard F. Paige is Professor of Software Engineering at McMaster University, Canada, and an Honorary Professor at the University of York, UK. He has published around 300 papers on topics related to software engineering, safety and security. He has chaired numerous software engineering conferences and workshops and is on the editorial boards of Software and Systems Modeling and the Journal of Object Technology . His research interests are in model management, model-based systems engineering, software processes, agile methods, and safety critical systems.