Parallel and Distributed Execution of Model Management Programs

Sina Madani

Ph.D

University of York

Computer Science

July 2020

Abstract

The engineering process of complex systems involves many stakeholders and development artefacts. Model-Driven Engineering (MDE) is an approach to development which aims to help curtail and better manage this complexity by raising the level of abstraction. In MDE, *models* are first-class artefacts in the development process. Such models can be used to describe artefacts of arbitrary complexity at various levels of abstraction according to the requirements of their prospective stakeholders. These models come in various sizes and formats and can be thought of more broadly as structured data. Since models are the primary artefacts in MDE, and the goal is to enhance the efficiency of the development process, powerful tools are required to work with such models at an appropriate level of abstraction. *Model management* tasks – such as querying, validation, comparison, transformation and text generation – are often performed using dedicated languages, with declarative constructs used to improve expressiveness. Despite their semantically constrained nature, the execution engines of these languages rarely capitalize on the optimization opportunities afforded to them. Therefore, working with very large models often leads to poor performance when using MDE tools compared to general-purpose programming languages, which has a detrimental effect on productivity. Given the stagnant single-threaded performance of modern CPUs along with the ubiquity of distributed computing, parallelization of these model management program is a necessity to address some of the scalability concerns surrounding MDE. This thesis demonstrates efficient parallel and distributed execution algorithms for model validation, querying and text generation and evaluates their effectiveness. By fully utilizing the CPUs on 26 hexa-core systems, we were able to improve performance of a complex model validation language by 122x compared to its existing sequential implementation. Up to 11x speedup was achieved with 16 cores for model query and model-to-text transformation tasks.

In memory of my grandfather,

Asadollah Eshghifar

1928 – 2019

Contents

List of Tables

List of Figures

List of Listings

Acknowledgements

Funding for this project was provided by the European Union's Horizon 2020 Research and Innovation Programme via the CROSSMINER project (grant #732223).

If I could thank only one person in this section, it would be my main supervisor, Professor Dimitris Kolovos. I have had the privilege of being his student since I started my MSc in 2015, and have always admired his work ethic, expertise and approachability. Despite being extremely busy and having an enormous list of responsibilities, he always finds time to reply promptly to e-mails (and even instant messages!), to meet and discuss progress and to give feedback. The feedback I have received from Dimitris throughout this period has been invaluable and extremely detailed. Dimitris is one of those supervisors who has a keen eye for detail, but also sees the bigger picture. I can still remember early on in my PhD when I sent him a 6,500-word report on a Friday afternoon and received feedback the same Sunday: not a single typo went unnoticed. This was not an exception, but the norm. I have also witnessed this with Dimitris' other students: he always gives detailed and valuable feedback, be it to a PhD student or to undergraduates. Furthermore, he made the option of this PhD available to me in the first place and choosing to pursue this has been one of the greatest decisions I have ever made. It would not have been possible were it not for Dimitris' guidance. I am also extremely grateful for placing his trust in me to work on the Eclipse Epsilon project, especially early in the PhD. The fact that he was willing to allow a student to contribute their research to an established open-source project used by serious industrial partners is quite bold and I once again emphasize my gratitude. The combination of having the opportunity to incorporate my research to a real-world open-source project as well as to attend and present at international conferences has greatly enhanced my PhD experience, and I will forever be thankful to Dimitris in facilitating this.

I would also like to thank my co-supervisor, Richard Paige who, like Dimitris, is an approachable, knowledgeable and forward-thinking professor. His guidance and expertise, particularly early in the project, helped to quickly narrow the scope of the project and make clear the contributions early on. I thank my Thesis Advisor Panel chair, Dr. Leandro Soares Indrusiak, for his excellent and impartial feedback on progress throughout the project.

I extend my gratitude to members of the Enterprise Systems research group, which has been an honor to be a member of. The enthusiasm, humor, camaraderie and expertise of its members is a rare sight. In particular, I would like to thank Dr. Konstantinos Barmpis for lending his advice during our many "discussions" in the office, and his lively spirit which (subconsciously) motivated and inspired me throughout this period and made the experience so memorable. I would also like to thank Patrick Neubauer, Betty Sanchez and Faisal Alhwikem for their feedback during a mock viva.

Last but certainly not least, I thank my parents – Nahid Eshghifar and Mahmood Madani – for their continuous support and encouragement, without which I would never have reached this point. Words cannot express my level of gratitude towards my mother's nurturing, or my father's wisdom.

I have been extremely lucky to have had such incredible people in my life during this PhD project: a perfect combination of parents, supervisors and research group. As such, the achievements of this thesis are attributable not only to myself, but in small part to them.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. All sources are acknowledged as References.

This project is purely technical in nature. There has been no personal data collected, or persons involved for experimental purposes. No ethical implications of this research or its developed artefacts have been identified.

Some of the work presented in this thesis has been published and presented at various conferences. These are listed in chronological order, with the relevant corresponding chapter. Note that the thesis chapters are considerably longer and different to the publications, and the thesis is naturally more up to date (e.g. with more and improved experiments / results etc.), however the overall contributions described in the thesis and the papers are more or less equivalent. These works have been co-authored with my supervisors, so the authors for all the following papers are as follows (in appropriate order): *Sina Madani*, *Dimitrios S. Kolovos*, *Richard F. Paige.*

Chapter 4

- *Parallel Model Validation with Epsilon*. Proceedings of the 14th European Conference on Modelling Foundations and Applications, Toulouse. pp. 115-131, Springer (2018).

Chapter 5

- *Distributed Model Validation with Epsilon*, International Journal on Software and Systems Modeling (SoSyM). Under review at time of writing.

Chapter 6

- *Parallel execution of first-order operations*. MODELS Workshops, Copenhagen. CEUR-WS Volume 2245 pp. 132–145 (2018). Presented at the OCL workshop at MODELS 2018, available online in [1].

- *Towards Optimisation of Model Queries: A Parallel Execution Approach*. The Journal of Object Technology Volume 18 Issue 2. Presented at the 15th European Conference on Modelling Foundations and Applications, Eindhoven (2019), available online in [2].

Chapter 7

- Due to a lack of existing documentation for the EGX language, Section 7.1 has been published on the Epsilon website [3].

1 Introduction

The nature of technological advancement brings with it increased complexity. In the domain of software engineering and computer science in general, managing this complexity is what allows us to build more powerful and feature-rich systems. To do so typically requires us to move to a higher level of abstraction. Modern, complex software systems would not be feasible to develop if programmers were operating at the bit (0/1) level. To improve the efficacy of modern system development, it is possible that a higher level of abstraction is required than is offered by modern programming languages.

Model-Driven Engineering (MDE) promotes (domain-specific) models to be the central artifact in systems development [4]. One of the motives in doing so is to enhance the communication between stakeholders in the development process. MDE allows each stakeholder to view the parts of the system relevant to them, at the level of abstraction desired by each stakeholder and with the appropriate notation [5]. This potentially allows stakeholders (managers, architects, salespersons, customers, developers etc.) to reason about complex systems more easily. This is particularly beneficial for large and complex projects, where higher levels of abstraction may improve comprehension, communication and development time since models are the central artefacts which are used throughout the development process. Therefore, models are not only used as design-time artefacts or high-level architectural overviews, but also directly by engineers implementing the system(s). These models can be validated, compared, evolved, queried and transformed using dedicated model management tools for these tasks, which often come in the form of task-specific programming languages designed to more easily (or elegantly) perform these specific tasks [6]. Finally, and perhaps most famously, models can be used to generate the runtime code, which may not only improve developer productivity but also correctness [7].

The projects most likely to benefit from a model-driven approach and the promises of MDE tooling are likely to be those with the most stringent requirements, both functional and non-functional. Unfortunately, the current state of model-driven engineering tools leaves much to be desired not only in aspects of user-friendliness but also performance (see Section 2.1.7). Since a key motivating factor for adopting MDE techniques and tooling is to increase productivity [7], it is important to ensure that the benefits outweigh its costs in this regard. This is especially so in the case of MDE, which is a relatively niche and novel approach to software and systems engineering than more established approaches and tools [6]. Project managers and software engineers may be less likely to learn and adopt MDE tooling in their projects if the development tools have inferior performance.

Scalability is a long-running concern in the MDE community in virtually all aspects (see Section 2.1.8). Model management tools are used in MDE projects because they greatly reduce the amount of code developers must write [8] and make development more collaborative since the tools are designed to be usable (or at least more comprehensible) by domain experts. Although these tools are intended to improve design-time productivity and clarity, they often fall short at runtime where in many cases performance is a vital concern [9]. The current state of most model management tools gives developers and project managers an unnecessary dichotomy: development time vs. CPU time. Whilst mainstream general-purpose programming languages have made significant progress in improving their abstractions, with integrated development environments (IDEs) offering mature tooling with advanced features, the value of model management tools becomes more questionable especially when the performance of general-purpose programming languages is far superior.

The abstractions offered by a model-driven approach are a strength at design-time but often a disadvantage at runtime due to the overhead of inefficient tooling. Instead of this abstraction being a burden, it is possible to take advantage of the more restricted nature of model management tools to improve performance; enabling optimisations which are much more difficult or time-consuming to implement correctly using lower-level general purpose tools. By exploiting the restricted semantics of even the most powerful, extensible and feature-rich model management languages, we can optimise aspects of the execution algorithm(s) in fundamental ways such that the runtime performance is comparable to what can be achieved within a reasonable time using a bespoke solution written in general-purpose programming languages.

This thesis aims to improve the runtime performance of tools used in model management tasks. Specifically, the aim is to investigate how parallelism can be used to reduce execution times for such programs, particularly with large models as input. The goal is to design a parallelisation framework and approach which can be applied to various model management tasks with minimal modifications. Furthermore, parallelisation should be transparent to the user and ideally be semantically identical to the sequential program, supporting all features offered by the model management tool. The expected contributions of this research are efficient parallel and distributed execution algorithms for rule-based model management tasks, and an evaluation of the performance benefits through prototype implementations. We apply our solutions to an established platform of model management tools and report on the challenges in adapting existing, non-concurrent programs. We demonstrate that the execution algorithm of a complex model validation language is inherently parallelisable with a high degree of granularity. Furthermore, we develop a novel distribution approach which is applicable for finite, deterministically ordered read-only model management tasks with the same data-parallel level of granularity as a non-distributed (shared-memory) parallel solution which scales linearly with the total number of processing cores across all computers involved in the computation. The distributed approach builds on the highly optimised shared-memory parallelisation infrastructure and comes at little to no cost due to our efficient distribution mechanism, so the (temporal) performance scalability of the multi-process approach is equivalent, if not superior in some cases, to the shared-memory (single-process) parallel execution approach.

The overall contribution of this thesis is a parallelisation and distribution framework which provides highly optimised solutions to the main challenges with implementing scalable parallel algorithms in the context of model querying and model validation, with the goal of full support for all complex language features such as the ability to write imperative code, express dependencies between rules and other complex aspects involving caching and laziness. The technical nature of this thesis makes it suitable for engineers to use as documentation for a “reference implementation” developed as part of this project. The solutions discussed in this thesis have been incorporated into the Eclipse Epsilon open source project and will continue to be supported / developed.

1.1 Project Plan

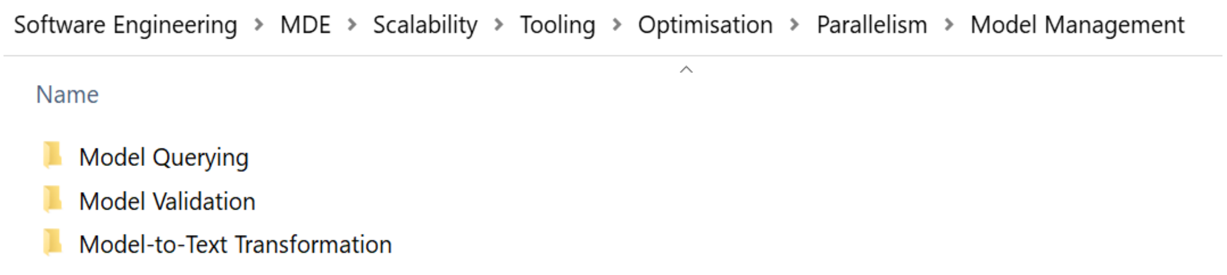
The research area of scalability in model-driven engineering is quite substantial and multi-faceted (see Section 2.1.8). Therefore, the focus of this thesis is on a small subset: improving the performance of model management programs. Section 2.2 identifies generic techniques for improving performance of computer programs, whilst Section 2.3 outlines how some of these techniques have been applied in the model-driven engineering domain. Whilst there is substantial research on minimising redundant computations through incrementality, comparatively fewer works attempt to improve the runtime performance of computationally intensive programs where incrementality and laziness cannot. Furthermore, most research is focused on model-to-model transformations rather than on other model management tasks.

This thesis aims to address a specific gap in the state-of-the-art. Whilst there are a few prototypes of parallel and distributed model transformation engines, there is a distinct lack of a generalised data-parallel approach to executing model management programs. The relatively few works which propose parallel and distributed execution algorithms focus on model-to-model transformations which, due to their complexity, do not typically scale well with more processors (see Section 2.3.1). This research attempts to discover the challenges with parallelising model management tasks and develops solutions to these challenges. We endeavour to show how the structure and declarative nature of model management languages can be exploited for parallel decomposition of their execution algorithms. We also show how this parallel decomposition can scale in a distributed environment using a novel approach.

Of course, the scope of this research is limited by time. Therefore, the model management tasks we focus on are model validation, model querying and model-to-text transformation.

This research takes a hands-on, engineering approach. We implement our solutions on top of existing model management languages and evaluate them relative to the status quo to assess the extent of performance benefits provided by parallelisation.

A more detailed description of the research objectives, hypothesis and scope is given in Section 3.2. All research can be thought of as a subset of a much larger whole, due to the complexity and breadth of knowledge. An analogy is a file system, where research is organised into various folders. For an overview of the area which this thesis resides in, consider Figure 1.1.1. For an even broader view of how this fits in with regards to the literature, refer to Figure 2.1.6.


Software Engineering › MDE › Scalability › Tooling › Optimisation › Parallelism › Model Management

Name

Model Querying
Model Validation
Model-to-Text Transformation

Figure 1.1.1 Research area of this thesis, represented as a folder structure

1.2 Thesis Contributions

Whilst there are several works which attempt to optimise model management programs in the literature, most are focused on model-to-model transformations, and relatively few works on parallel and distributed execution. Although this thesis is not the first to introduce parallelisation to model-driven engineering tasks, to the author's knowledge there are no other works which have approached the same research question in the same manner, with the same scope and with equal or superior results (as judged by quantifiable evaluation metrics). More specifically, the main contributions of this thesis are summarised in the following subsections.

1.2.1 High-granularity parallel architecture

The contributions of this thesis depend on thread-safe properties of the execution engine of the targeted model management language. In this project, we focused exclusively on the Epsilon family of model management languages, for numerous reasons discussed in detail in section 3.2.4. As noted, Epsilon's languages are complex in that they provide a multitude of features, with imperative and declarative constructs. Since they all build on a common OCL-inspired query language (EOL), most of the work on making their execution engines thread-safe is targeted towards EOL. With the help of a previous prototype [10], we identified the key data structures and provided efficient solutions for making them thread-safe. The decoupling of Epsilon's languages from modelling technologies means that we can cleanly separate concurrency issues caused by deficiencies in the execution engine of Epsilon's languages, and the implementation of specific modelling technologies.

To demonstrate our parallelisation architecture, we modified the execution engine of the Epsilon Validation Language (EVL) – a complex rule-based model validation language which allows for dependencies between rules and lazy rule execution. We made the case for data parallelism and provided solutions for concurrency issues which maintain fully compliant execution semantics with the sequential engine (i.e. no retracted features) such that our parallel engine can be used as a drop-in replacement. We tested this with a complex test suite, ensuring equivalence (of both results and engine internals) not only with sequential EVL but also with OCL; the de facto model validation language. Our performance evaluation showed that the parallel execution engine is relatively low overhead and scales linearly with more threads up to the number of physical CPU cores; albeit with a notable drop-off in efficiency beyond 6 cores depending on the script and hardware configurations. In the best case however, our solution scales linearly with logical cores, achieving up to 18x speedup with a quad-channel 16-core/32-thread system.

The core of our approach has been generalised to the Epsilon Rule Language (ERL), an internal abstraction which extends EOL for rule-based languages. Whilst the thread-safety issues for each rule-based language will inevitably vary due to the execution semantics and data structures involved, our parallelisation approach can be reused to a great extent, such that making other ERL extensions parallel "only" requires the developer to deal with concurrency issues brought about by task-specific data structures. We demonstrated the generalisability of our approach beyond model validation by applying it to EGX: a declarative co-ordination language for model-to-text transformations.

Besides the various solutions to concurrency issues regarding engine internals and data structures, we believe the solution to dealing with dependencies, especially with lazy rule execution, to be a particularly novel (and arguably generalisable) contribution.

1.2.2 Generalised parallelisation

Our parallel architecture extends beyond a single algorithm for rule-based languages. To leverage the benefits of parallelism more generally, we devised bespoke parallel declarative operations which can be used in EOL (and thus, any language based on it). These operations also benefit from data parallelism, and the execution algorithms are more varied and complex due to the more diverse semantics of the operations. Although the contribution of these parallel operations is not novel – at least not outside of MDE – implementing these operations has proven to be extremely useful in refining our architecture and for evaluation purposes. The fact that we were able to parallelise almost all of Epsilon's declarative operations with performance which scales with the number of cores helps to justify the generalisability and strength of our approach. Furthermore, since all EOL-based languages can benefit from these operations, we can also claim that our approach provides partial automatic built-in parallelism for all model management tasks supported by Epsilon, which besides model validation also include model querying, model-to-model transformation, model-to-text transformation, migration, comparison etc. We are not aware of any other tools (for example, OCL-based model-to-model transformation languages like ATL) which incorporate this kind of generalised parallelism. Our bespoke parallel *select* and *collect* operations preserve ordering, which is extremely uncommon for parallel algorithms. We tested these operations thoroughly using equivalence tests, along with testing the engine internals and short-circuiting semantics.

We recognised that although our bespoke operations are diverse, the Java Streams API provides lazy execution semantics which makes it more suitable for chaining declarative queries (which may also be parallel). We argued that, especially amongst users familiar with general-purpose programming languages, the ability to re-use the Streams API in model management languages with identical semantics is desirable. We incorporated Java Streams into EOL in a way which, as with our parallel operations, is transparent to the user. Supporting Streams with identical semantics to Java, as well as parallelism, with the ability to mix and match all features of EOL, was a non-trivial engineering task. To our knowledge, no other model management language supports parallelism of this diversity, so we can also claim that our work is the first to combine laziness and parallel execution (like Java Streams) in a model management context.

Although a relatively small contribution, we are not aware of any works which provide similar functionality to the *nMatch* operation (described in section 6.3.2), even outside of the realms of MDE. This operation transforms a filtering operation on a collection (which returns a subset of that collection based on a predicate applied to each element) followed by a size comparison into a short-circuiting operation. The ability to turn a non-short-circuiting operation into a short-circuiting one, especially in parallel is, we argue, relatively novel, especially given the extent of research in, for example, the database community on query optimisation. Note that the semantics of the *nMatch* algorithm cannot be replicated by Java Streams. This is a contribution which could be of interest to a much wider community of language engineers.

1.2.3 Efficient distributed execution

To demonstrate the scalability of our data-parallel approach beyond a single-process, shared-memory architecture, we devised two technology-independent distribution mechanisms for our parallel decomposition. In both cases, each participating process is able to maximise CPU utilisation, meaning that our approach not only scales with the number of machines but also the number of hardware threads in each computer. We demonstrated this once again in the context of EVL. Notably, our architecture is able to distinguish between the master and workers regardless of the implementation, such that the proportion of jobs performed on the master using parallel EVL can be specified by the user. Since our approach is high granularity (rule and data parallel), we assign jobs randomly, as it is assumed that no single job will dominate execution time.

In our first prototype, we devised efficient serializable representations of validation rule-element pairs, where model elements were resolved by IDs using the underlying modelling technology. Rather than serializing entire object graphs of both the EVL program (and its dependencies) and the model, our architecture can locate rules (constraints) and model elements on-the-fly from minimal information (names, IDs etc.), which applies not only to the inputs but also to the results (unsatisfied constraints). Resolution of results is performed lazily on the master.

In our refined approach, rather than relying on the modelling technology to support the notion of element IDs, and to minimise the distribution overhead resulting from the amount of data sent over the wire, we devised a much more efficient distribution format which exploits the finite and deterministically ordered nature of rule-based model management programs. Rather than sending serializable proxies, we could distribute batches of indices, where each batch consists of two numbers: a start and end index. These are then used to determine which subset of the job list to process. The results (unsatisfied constraints) could also be inferred in a similar manner. Our batch-based indexing approach eliminates the need for the modelling technology to support element IDs, making it more generalisable. Our architecture also allows for the batch size to be tuned to maximise CPU utilisations whilst minimising distribution overhead. We found that in practice, we were able to maximise CPU utilisation on all nodes by setting the batch size to the maximum number of hardware threads, with minimal overhead since each batch only consists of two integers.

We implemented our technology-agnostic approach using the Java Message Service (JMS) API. This enabled us to explore concretely the implementation challenges at a lower level, as well to perform further optimisations. In particular, the sequence of communication between the master and workers during the initial setup phase is particularly complex due to our decision to make it as asynchronous as possible to achieve maximum efficiency. Our implementation also processes jobs and results asynchronously, including support for handling of failed jobs. Our evaluation showed performance scalability in line with, if not superior to, parallel EVL: at best with 87 machines each equipped with a hex-core CPU, we were able to reduce execution time from over 6 hours and 23 minutes using sequential EVL, to under 70 seconds: a speedup of 340x. To demonstrate the generalisability of our approach, we also developed a prototype implementation in Apache Flink; a widely using distributed processing framework.

1.2.4 Automatic and user-defined parallelisation

The user can take advantage of the parallel infrastructure without any modifications to existing programs, and with fully compliant execution semantics. In other words, without monitoring their CPU usage (or thermals) the user cannot distinguish between parallel and sequential execution. Therefore, our solutions are transparent to the user. Barring a few rare exceptions, our parallel execution algorithms should work with existing programs without any changes to the programs themselves. If desired, the user can also select the level of parallelisation.

To our knowledge, this is the first work which combines a rule-based parallel execution algorithm whilst also offering general-purpose parallel execution abilities in the same program. Our architecture is able to automatically prevent invalid states, such as nested parallelism, and automatically apply parallelism where CPU utilisation can be maximised in the event of nested declarative operations. The user does not need to modify existing programs, they only need to opt to use our parallel execution engine.

For the more technically inclined user, our parallel solution is also highly flexible in where and when parallelism is applied. Besides the ability to configure the degree of parallelism (number of threads), they can also manually override the default parallelisation behaviour. In the case of rule-based languages like EVL, an annotation can be applied to determine whether the rule should be executed in parallel for a given model element. In the case of declarative operations, the user can explicitly invoke the sequential or parallel variant.

1.2.5 Parallel Model-to-Text Transformation

There are no known works which attempt to parallelise text generation from heterogenous models. One of the most frequently used and well-known model management tasks – perhaps arguably what model-driven engineering is known for outside of the community – is the ability to generate textual artefacts from models. Model-to-text (M2T) transformation is synonymous with code generation, although the generated text need not be executable code – it can be documentation, or even another model expressed using a textual form. Typical model-to-text transformation solutions are template-based. The user types static text into the source file which will be appended as-is to the output stream (although it may be processed further to format, for example, white space). There is then a special character sequence which will begin a dynamic section, at which point the user can write code to generate the output programmatically, or access model elements. This style of dynamic text generation is widely used in web architectures, for example through PHP or JSP. In the MDE community, solutions such as Acceleo and the Epsilon Generation Language (EGL) are used.

A prime opportunity for parallelisation comes from the orchestration of templates, which may be invoked with various parameters using a language such as EGX. Conceptually, this is perhaps one of the purest and simplest rule-based tasks to parallelise, as there are no dependencies between rules and each rule operates on a collection of elements. The output of the template is typically directed to a unique file, where the name is usually derived from the model element for the current rule.

1.2.6 Legacy of Tooling and Evaluation

The contributions of this thesis have resulted in practical tools integrated into the main repository of the Eclipse Epsilon open-source project, with the exception of Distributed EVL which, at the time of writing, is still a prototype (no GUI etc.). The integration of the parallel infrastructure and execution algorithms (as well as the test suite) directly into the development branch of the project they are based on shows that the contributions of this thesis are not purely theoretical, but that all practical aspects of putting them into practice have been considered, and a commitment to supporting them is made by the author. Since Epsilon is used in numerous industrial contexts [11], we hope that the outcome of this thesis has a tangible, positive impact on productivity for users, rather than being only a proof of concept.

Furthermore, we recognise that reproducing our evaluation is complex and time-consuming, so we have made not only the resources for our experiments open-source, but also included the required tooling and automation to ease the reproducibility and extensibility of our benchmarks in a portable manner, as discussed in section 3.3.4. A complete, self-contained repository of accompanying material used to evaluate the developed software artefacts of this thesis is available online [12].

1.3 Thesis Structure

The remainder of this document is structured as follows. Chapter 2 reviews the preliminaries of model-driven engineering and high-level optimisation techniques for computer programs, before exploring the intersection of the two areas (i.e. related works). Having explored the motives for this project, Chapter 3 outlines the research hypothesis, objectives and methodology for achieving them along with the evaluation criteria. We begin our contribution in Chapter 4, which demonstrates efficient solutions to the challenges posed by introducing concurrency to an existing sequential execution engine. Specifically, we show how Epsilon's model validation language can be parallelised, along with a thorough suite of tests and performance benchmarks. We then build on this infrastructure and parallel decomposition in Chapter 5 to improve scalability by implementing an efficient distribution approach which exploits the finite and deterministic ordering of the program's execution algorithm to effectively utilise multiple computers' processing capabilities. We performed further experiments, showing how our distributed approach scales even better than our parallel one in terms of efficiency. Recognising that our parallelisation approach needs to be applied on a per-language / model management task basis, and to further develop the parallelisation infrastructure, in Chapter 6 we show how parallelisation can be applied to declarative (first-order logic) query and transformation operations which are widely used in all model management languages, as well as proposing more efficient means to express them. We perform an extensive evaluation of these, comparing to OCL and handwritten Java code, on a variety of computers to better understand the performance characteristics. Chapter 7 attempts to demonstrate the generalisability of our parallelisation approach for rule-based languages by applying the solutions developed in Chapter 4 to Epsilon's model-to-text co-ordination language, along with performance benchmarks. Finally, Chapter 8 concludes the thesis and outlines potential future work.

2 Background

This chapter provides background to the research, namely preliminaries and a literature review. Sections 2.1 and 2.2 introduce basic concepts which outline the context of this thesis, whilst section 2.3 covers literature pertinent to this thesis' contributions. More specifically, section 2.1 provides an overview of Model-Driven Engineering along with the motives for improving the performance of model management programs. Section 2.2 reviews general-purpose performance optimisation methods used software engineering. Section 2.3 provides a detailed account of the overlap between the preceding sections; i.e. performance optimisation of model management programs. Meanwhile section 2.4 gives a brief overview of similar research in the context of databases. Finally, section 2.5 summarises the chapter.

2.1 Model-Driven Engineering

This section introduces Model-Driven Engineering (MDE), its concepts, applications and scalability concerns. Firstly, it is important to establish the relevant terminology, which is notoriously convoluted in the model-driven engineering field. Brambilla, Cabot & Wimmer (2012) provide a good overview of the relevant concepts and terminology [13]. The acronyms used in MDE refer to the granularity in which the approach is applied. Model-based engineering (MBE) is the broadest form, where models (used in the most liberal sense of the word) are used in the engineering process. Model-driven development (MDD) means models are used as the central artifact throughout the development process. For the remainder of this thesis, we focus on MDE in the context of software engineering – thus the most relevant acronym is MDSE (Model-Driven Software Engineering). However, for simplicity we continue to use the MDE term to avoid confusion.

2.1.1 Concepts

Model-Driven Engineering (MDE) is a software development paradigm in which models are "first-class citizens". In a model-driven approach, models drive the process and are the key artifacts of interest. As Sendall and Kozaczynski put it, "*The objective of model-driven development is to increase productivity and reduce time-to-market by enabling development at a higher level of abstraction and by using concepts closer to the problem domain at hand, rather than the ones offered by programming languages.*" [14]. Instead of focusing efforts on abstractions in the *programming*, such as the object-oriented paradigm, the emphasis is on abstracting the problem using modelling techniques; where models are treated as "first-class citizens" [15] in the solution.

A "model" is an abstraction; - a simplification of a more complex system designed so that the areas of interest of the system in question can be more easily analysed and understood [16]. One of the main benefits of using a model is that it makes the problem domain easier to understand among both technical and non-technical stakeholders [16].The concept of models and modelling is frequently used in scientific fields such as physics and economics, however the use of models to drive the software engineering process is relatively recent and the literature for this area is still quite young.

The reason models are so important in many scientific fields is that they allow one to observe the relationship between elements in a particular domain. They abstract the important “parts” of a problem domain into tangible “elements” which can be used within the model. An example of a model is a system architecture. It is a simplified representation aimed at giving an overview – it is not the system itself.

To “model” something, we require some notation. For example, if we’re drawing a system’s architecture, we need to represent this architecture in a tangible way. Commonly we may use boxes, lines and arrows with some labels on them to clarify what those boxes and arrows are representing. That is, the notation gives some guidance on how to interpret the model. The aesthetics and symbols of the language are its concrete syntax. The concrete syntax is how users of the language will learn and use it, so it should be quite clear and easy to read and write using the notation.

The types of elements that can be contained in a model as well as their features and valid relationships are typically defined by an artifact known as a “metamodel”; - a model that defines the structure of a modelling language [16]. In fact, even metamodels themselves have a model which describes them, called a “meta-metamodel”. The “meta-metamodel” is defined in itself so this is the most abstract level of model definition. In essence, there are four layers of abstraction, which can be denoted as M0, M1, M2 and M3. This can be visualised as a pyramid structure as shown in Figure 2.1.1; as we move down an abstraction layer (from M3 to M0) there is more variation. For instance, we may think of XML Schema (XSD) as being a metamodel for XML documents. XSD is itself defined in XSD [17] – so we can think of “XMLSchema.xsd” as being the M3 meta-metamodel. Each XSD document conforms to “XMLSchema.xsd”, but an XSD is just a way of modelling restrictions on XML documents. So, an XSD document is an M2 metamodel in a sense – it is an *instance* of “XMLSchema.xsd”. Then each instance of an XML document which conforms to a given schema is an M1 model – it models the domain but conforms to the rules outlined in the schema. Finally, we have the “thing” we’re modelling itself. This is the least abstract level (M0) and exists for reference. Unlike M3, M2 and M1, M0 is not a model, it *is* the “thing being modelled”. So in this hierarchical system, M0 conforms to M1, M1 conforms to M2, M2 conforms to M3, and M3 conforms to itself. It may help to think of the relationship between a model and metamodel as that of a “class” (M2) and “object” (M1) in object-oriented programming.

How do we express models and metamodels? What do models look like? We use *modelling languages*, which consist of two components: the *abstract syntax* and *concrete syntax* [16]. The former refers to the structural nature of the language: what concepts and relations can be expressed using the language. The latter is how this language is presented and used. This may be textual, graphical or a mixture of both. Concrete syntax is concerned with notation and representation, whereas abstract syntax describes the language’s capabilities. We may also include the semantics as part of the language definition; which can be defined as “*a set of rules that govern an abstract machine that is able to execute any syntactically correct linguistic utterance of the language*” [18]. In other words, the semantics are additional constraints placed on the language which exist to ensure that statements; - even those which adhere to the syntax, make logical sense. Of course, formally defining semantics is optional, however Kleppe advocates the definition of these "rules"; amongst a mapping between abstract and concrete syntax (and vice-versa) [18].

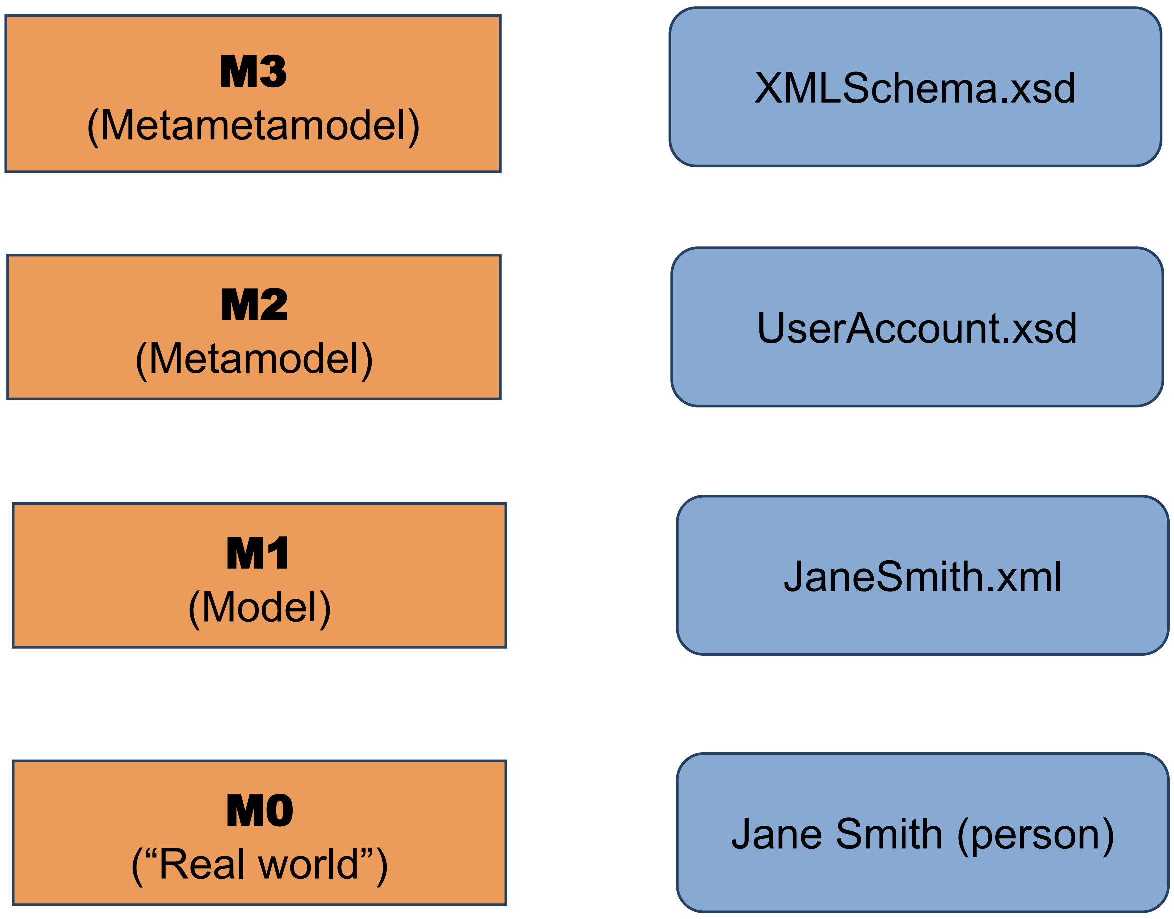


Figure 2.1.1 Example model abstraction hierarchy

Regardless of the notation – of which there may be many – the language itself has an underlying model. This is the metamodel of the language, also known as the *abstract syntax*. The abstract syntax defines the underlying concepts and structural elements of the language. For example, a UML class diagram may have many classes, interfaces etc. and these can have different relationships, such as inheritance, composition, aggregation, association, dependency etc. The concrete syntax for these is arbitrary and not important for the structure and semantics. The relationships, for example, are represented using different types of arrow heads to distinguish between whether the relationship is a dependency, aggregation, composition etc. But there is no reason why these can't be interchanged, it's just a recognizable convention. Such conventions however are not without rationale – see Moody (2009) for the science behind effective notation design [19]. By contrast, the abstract syntax is the metamodel for these symbols. The abstract syntax (metamodel) is concerned not with symbols but with concepts. For instance, the metamodel (remembering that metamodels are models themselves) for UML may have a "relationship" element and the inheritance, composition, aggregation etc. may all be "subclasses" of the "relationship" element. Furthermore, the abstract syntax may place restrictions on how to use these elements. For instance, in some cases and modelling languages, it is forbidden for a relationship element to be able to have a relation with another relationship element. In more tangible terms, this means that, for example, an "arrow" can't be connected to another "arrow". However, an "arrow" *can* connect from one class to another – that is, a relationship may exist between classes, but not relationship elements themselves. In other cases, this may be perfectly plausible – for example UML association classes. This restriction and structure is not the concern of concrete syntax; hence the need for a metamodel (abstract syntax) to define the concepts, elements and structure used in the language. Rodrigues da Silva illustrates the concepts of abstract and concrete syntax using a UML diagram, as shown in Figure 2.1.2 [16].

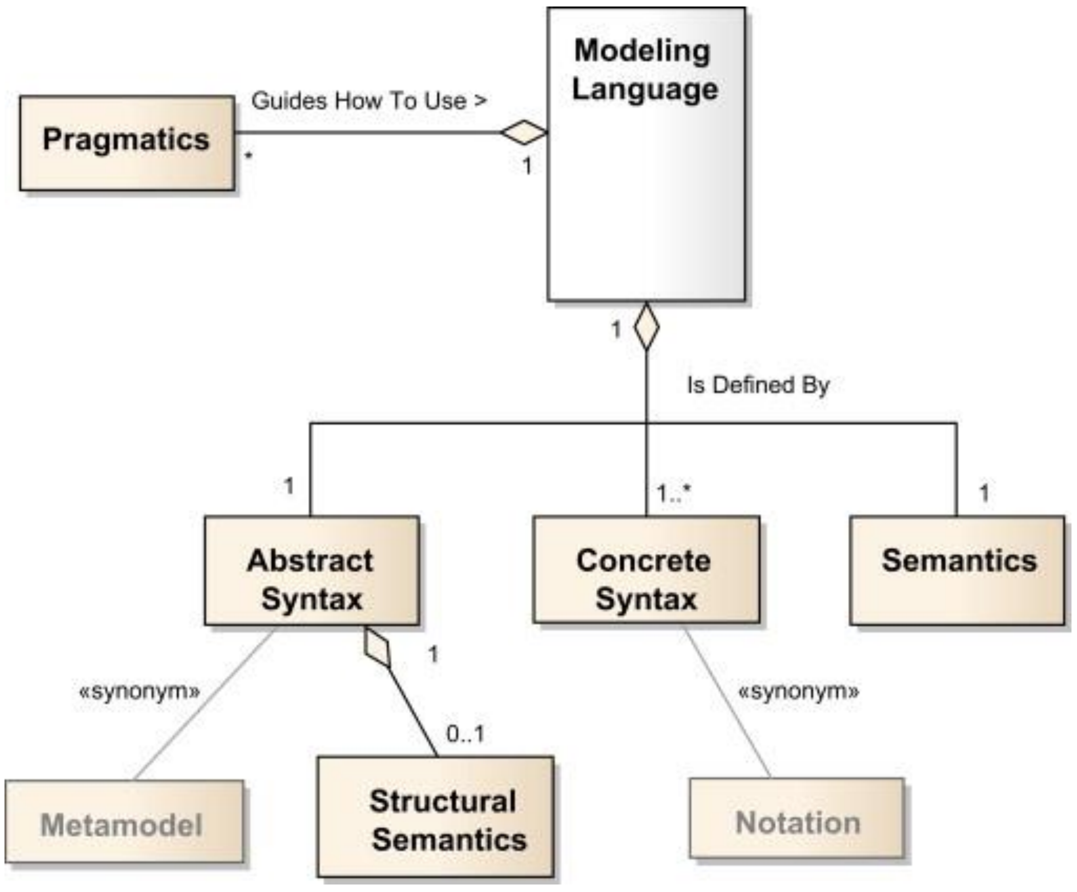


Figure 2.1.2 Modelling language definition diagram [10]

2.1.2 **Modelling Framework**

The definition of a model allows for an extremely broad range of artefacts to be models themselves. From a practical software implementation aspect however, it would be ideal if such artefacts had a common structure – a "meta-metamodel" – as well as a standardised set of APIs for interacting with them (i.e. for Create, Read, Update, Delete (CRUD) operations) and of course, a convenient persistence medium. A *modelling framework* provides such facilities which make it easier for various tools to work with models at a higher level of abstraction.

The *Eclipse Modelling Framework* (EMF) [20] is a pragmatic implementation of the Object Management Group's Meta-Object Facility (MOF) [21] specification, and the *de-facto* ecosystem for most mainstream MDE tools and research. According to its home page, "*From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model*" [20]. In other words, it provides a standardized gateway to easily access a complex model persistence format known as XML Metadata Interchange (XMI) [22]; a plaintext file format which uses XML to define the structure, relations, attributes and other metadata of model elements. Typically, a model is stored as a single XMI file, though more recent techniques allow for greater modularity by splitting models into fragments, which are then stored as multiple XMI files [23].

The main component of EMF is Ecore; an object-oriented metamodeling language. Ecore is defined in itself – i.e. it is a meta-metamodel, and provides an abstraction layer which maps its own internal structures used to represent models into their Java equivalents [24]. A high-level hierarchy diagram of Ecore's main top-level concepts is illustrated in Figure 2.1.3. Being implemented in Java (although a third-party .NET implementation is also available [25]), the type system and concepts are heavily inspired by Java. We can see that the root element is *EObject*, and there are multiple ways to define types ("*EClassifiers*"), such as *EClass* and *EEnum*. Built-in types include *EString*, *EInt* etc. which map directly to the Java data types. Types can have operations as well as attributes, similar to Java classes. This approach makes EMF more easily accessible and understandable to developers familiar with object-oriented programming, but raises the level of abstraction allowing for the focus to be on domain modelling rather than implementation details.

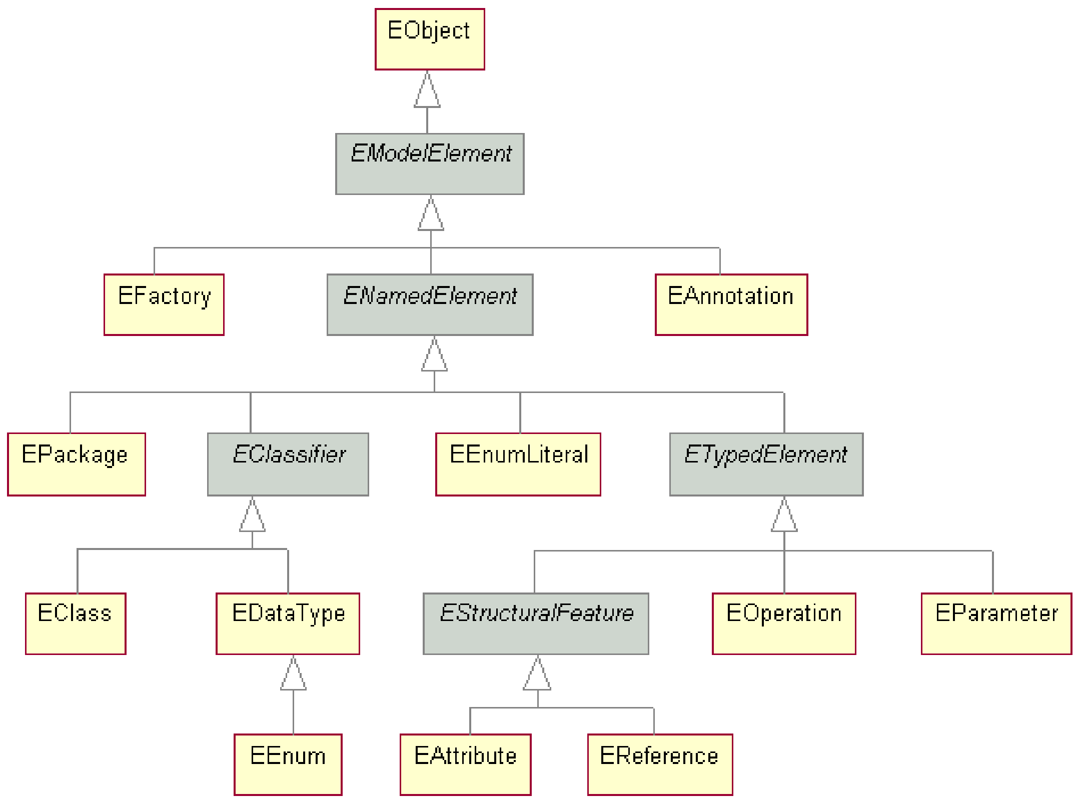


Figure 2.1.3 Simplified diagram of Ecore's components hierarchy [24]

According to the lead EMF developer, "*Ecore is the defacto reference implementation of OMG's EMOF*" [26] (Essential Meta-Object Facility) – which is the Object Management Group's standard for model-driven engineering. The EMF ecosystem and tooling makes working with models much more accessible to developers compared to, for example, manually defining and manipulating data using XML – a much-needed endeavour for increasing the adoption of MDE tools. It is not surprising then that a number of MDE tools on the Eclipse platform use Ecore as their base metamodelling API. In doing so, it allows users of these tools to seamlessly integrate them into their solution; since the model(s) all conform to a common metamodel with a well-established, well-supported API on a unified platform with well-defined concepts.

2.1.3 **Model persistence**

An important consideration in large modelling projects is scalable access to models. Although file-based solutions such as XMI are sufficient for a small number of users working with smaller models, concurrent access – especially in distributed environments – becomes an issue for collaboration. One solution to such problems would be to use databases for storing, accessing and manipulating models. The Eclipse *Connected Data Objects* (CDO) project "*is a distributed shared model framework for EMF models and meta models*." [27]. CDO is a *model repository* – that is, a persistence and access solution for multiple models. Architecturally, it provides an API for EMF-based applications, a repository server and almost any type of persistence back-end (though in practice relational databases are commonly used). According to a presentation by its author at EclipseCon 2008, the main goal of CDO is to overcome the limitations of XMI [28]. CDO provides more advanced functionality for accessing data within models, such as transactions and distributed concurrent interactions with multiple models. Although CDO enables multiple users to work with multiple models and metamodels whilst sharing a common view of the repository, Kolovos et al. (2009) note that such a solution does not address a more fundamental problem with modelling languages; namely the lack of modularity and encapsulation constructs [9]. In other words, whilst traditional model repositories like CDO provide a database façade for overcoming the limitations of file-based persistence, they do not split large models into smaller, reusable chunks.

In more recent years, NoSQL database technologies have become increasingly popular forms of persistence backends for large models. These include document databases like MongoDB and, more prominently, graph databases (given that most models are graph-like in structure, and the close relation between the modelling and graph transformation research community) such as Neo4J. The NeoEMF [29] project is now a mature and scalable model persistence solution, with early publications dating back to 2014 [30]. Databases have numerous advantages over traditional file-based formats with regards to scalability. However, in most common cases where scalability isn't an initial concern, XMI-backed models which are loaded fully into memory are still widely used in model management programs due to the simplicity and widespread tool support.

2.1.4 **Model Editors**

Another important – and perhaps the most familiar – aspect of modelling is the process of manually manipulating models and metamodels. Typically, this is performed using graphical editors, however textual languages are also an option. Metamodels in EMF can be defined using the Ecore tree editor, just as any regular model. An Ecore model conforms to the Ecore metamodel. A textual syntax for defining Ecore models (i.e. metamodels conforming to the Ecore metamodel) exists in the form of Emfatic [31], which provides an object-oriented style textual syntax for defining metamodels, similar to how a programmer may define classes. The corresponding Ecore model can then be generated from this text file. As for models themselves, the Ecore tree editor constrains element creation and property values based on the metamodel, however it is often argued that such an editor is not user-friendly or elegant since it is essentially a context-aware XML editor. A more sophisticated alternative to graphical editing of models would be to generate a dedicated editor for a given metamodel. The Eclipse Graphical Modelling Framework (GMF) provides much of the infrastructure needed for such a task, however due to its versatility and extensibility it is difficult for developers to initialize. For the most common and simplest use cases, the configuration files needed to create a functioning GMF editor can be automatically derived from a metamodel definition. This functionality is available from EuGENia [32], which uses annotated Emfatic metamodels to automatically generate all the code required for a fully functional editor which can be used as an Eclipse plug-in.

Although EuGENia provides a bridge between textual and graphical modelling (through its use of Emfatic) – allowing the structural aspects (i.e. abstract syntax) to be defined in text and the notation (concrete syntax) to be defined via annotations and further customisations via various files / editing generated code, any changes require redeployment of the generated graphical editor. An alternative is to have a reflective graphical editor which can be configured at runtime for the users' needs. Sirius is a good example of this, marketing itself as "The easiest way to get your own Modeling Tool" [33].

Despite the prevalence of graphical modelling due to its ease of use and comprehension by non-technical stakeholders, textual modelling is still commonly used due to its portability, ease of editing without dedicated tools (a text editor is usually sufficient), is often concise and unambiguous compared to some graphical forms of modelling. There are also issues with scalability in graphical editors, since displaying large and complex models is not only more computationally intensive but requires more optimised approaches, such as viewpoints. There is even a standardised specification by the Object Management Group called HUTN (Human Usable Textual Notation) [34], with an implementation of it in Epsilon [35] [36]. Note that unlike Emfatic, which is a textual syntax for defining Ecore metamodels, HUTN is itself used to define concrete instances of models rather than metamodels. Conceptually, one can think of HUTN as a "JSON for modelling". Although one can use JavaScript Object Notation for modelling, there is usually no way to enforce a schema. In that regard, HUTN is more like an XSD-backed XML document, with a less verbose syntax.

2.1.4.1 Domain-Specific Languages

The notion of Domain-Specific Languages (DSLs) and their relation to MDE is of great significance in the wider MDE community. The rationale behind DSLs is to a great extent similar to that for modelling: to make the process of defining business logic and domain concepts more accessible to non-technical stakeholders – in the case of DSLs specifically, to domain experts. Rather than using powerful general-purpose tools and languages, which are often too low-level to be expressive and concise, DSLs are designed for a specific use case and due to their specialised nature, they are better suited to expressing the essence of the domain logic. Designing a DSL is in principle similar to metamodelling: in both cases, the constructs provided by a generic "meta-metamodel" are used to define a language for modelling the domain.

Frameworks such as JetBrains' MPS [37] and Eclipse Xtext [38] provide the necessary tooling and infrastructure to easily define DSLs without having to write their own lexers, parsers, editors etc. Language engineers only need to define the grammar (i.e. the "metamodel") of the language using a relatively high-level EBNF-style language, and the rest is automatically generated. Xtext in particular has strong ties to EMF [39], allowing for grammar files to be transformed into Ecore metamodels and even vice-versa; although the latter has many limitations. The ability to treat DSL grammars as models has the benefit of being able to leverage model management tools in language engineering.

2.1.5 **Model Management**

Having explored the basic concepts of modelling and models as artefacts, we now turn to how such models can be meaningfully used in the development process. We refer to activities which programmatically process, evolve, analyse or use models in any way as *model management* tasks.

2.1.5.1 Model-to-Model Transformation

We have already seen that models can be used as an abstract description of a system – an architecture – for different stakeholders. However, the model may not be at an appropriate level of abstraction or convey parts of the system which are not of interest to a stakeholder. Even more simply, the model may be presented with labels, notation and/or terminology which the stakeholder is unfamiliar with. Supposing that we have a base "master model" which contains the full details of system, and we have different stakeholders responsible for the implementation of different aspects of the system, it would be time-consuming and error-prone to manually recreate the base model with reduced detail for each of the stakeholders' needs, or to force each stakeholder to navigate the complex model to find the parts which concern them. Instead, we can automate this process using *model-to-model transformations*, or simply *model transformations*. This is one of the most common and useful activities performed on models, and has even been referred to as "the heart and soul of model-driven engineering" [14].

A model transformation program takes as input a source model, source metamodel, target metamodel and transformation script. The transformation script specifies a mapping between the elements of the source and target metamodels. According to Sendall and Kozaczynski, model transformation "*requires a clear understanding of the abstract syntax and the semantics of both the source and target.*" [14]. Sendall and Kozaczynski identified three architectural approaches to model transformation: Direct Model Manipulation, Intermediate Representation and Transformation

Language Support [14]. Direct Model Manipulation tools, as the name suggests, allow users to directly access the internal structure of the model through some Application Programming Interfaces (APIs) which can be used in common programming languages. Intermediate Representation tools "export" the model into some intermediate textual format, such as XMI (XML Metadata Interchange – an OMG standard) [40]. Transformation Language Support offers a domain-specific language (DSL) for expressing the transformation process. Of these three, Sendall and Kozaczynski advocate a DSL for model transformation [14]. Compared to an API or an alternate representation of a model, a dedicated language specifically designed for model transformation offers much greater flexibility and ease of use. Indeed, there are a large number of various model transformation languages [41]; some of which offer graphical notation and constructs [14] which can help to improve the readability of transformations.

There are three types of model transformation languages: declarative, imperative (procedural) and hybrid (a combination of both) [41] [14]. Declarative languages tend to have a greater number of language constructs and work at a higher level of abstraction, whilst imperative languages are more low-level and require the programmer "to manually address issues such as tracing, resolving target elements with their source counterparts, and orchestrating the transformation execution." [41]. Hybrid languages offer a compromise by offering users the higher-level constructs of declarative languages without sacrificing the flexibility of the imperative style.

Model transformations typically use domain-specific languages such as ATL [42] (a hybrid model transformation language and execution engine) for performing the mappings. In declarative style, such mappings are specified as *transformation rules*; taking as input elements from the source model, the corresponding elements from the target model and then specify how to derive the values of the target model elements from the selected source model elements. In the declarative subset of ATL for example, the selection of source model elements can be a model query expression, such that the transformation rule applies only for elements meeting a certain criterion beyond simply their type. Rules may also apply to multiple target elements, so mappings be *N:M*. That is, model-to-model transformations need not be *complete*; they may use only a subset of the input model. Furthermore, rules can be depended on by other rules, or may only be activated if invoked by other rules (these are *lazy rules*). In ATL, rules can even be inherited! Another aspect of model-to-model transformations is whether they are "*in-place*" or "*out-place*". In-place transformations evolve the source model so that it conforms to the target metamodel, whilst out-place transformations create a new target model (that is, they do not modify the source model).

A classic example of model-to-model (M2M) transformations is "Class2Relational", which maps a Class (from the domain of object-oriented programming) to a relational database schema. In the following example (taken from [43]), the rule maps an object-oriented data type from the "Class" domain model to a database type from the "Relational" domain model. For simplicity in this case, the mapping is trivial because it is assumed that the type names supported in the object-oriented model are equivalent to the type names in the relational model. A minimal example of a rule in ATL demonstrating this transformation is shown in Listing 2.1.1.

```
rule DataType2Type {
    from
        dt : Class!DataType
    to
        out : Relational!Type (name <- dt.name)
}
```

Listing 2.1.1 *Class2Relational* transformation rule

The study of model-to-model transformations is closely tied to graph transformations, for example as shown by Heckel (2006) [44]. Consequently, there is an extensive research overlap between the graph transformation and model transformation community. Graph theory underpins the foundations of formal and analytical studies of model transformations. The graph transformation community refer to models as “type-attributed graphs”, and research is focused on fundamental issues such as “confluence” and “co-evolution”. For an overview of graph transformation approaches to MDE, see Taentzer et al. (2005) [45].

Model-to-model transformations are perhaps the most complex model management task, since they incorporate several core concepts and operations found in other tasks, such as pattern matching, correspondence, rules and dependencies. The act of simply querying models is itself a distinct task requiring a language for programmatically expressing the patterns to be matched. Since models and their elements are complex entities, a model querying / navigation language is more like SQL than regular expressions. One such language typically used for querying models is the Object Constraint Language (OCL) [46]. OCL is defined by the OMG as a declarative language free from side-effects and with a strong emphasis on the use of first-order operations for querying of collections. OCL is typically used in conjunction with other languages and tools for writing query expressions. For instance, ATL uses many of OCL’s features, including the data types, operators, operations and expressions. Note that OCL is not meant to be a complete programming or pattern matching language; rather, it provides a convenient syntax, data types (including collections) and built-operations on data types for side-effect-free model management activities.

With model transformations being one of the cornerstones of MDE, it is unsurprising that the Object Management Group also has a standard for this, namely QVT (Query/View/Transformation) [47]. QVT is not itself a language, but a specification for a family of related model transformation languages with different capabilities and semantics. All languages build on top of OCL as the core querying language. The two most notable are QVT-Operational; which is the imperative M2M language, and QVT-Relations; which specifies bidirectional transformations declaratively by describing consistency relations between models. All QVT languages operate on MOF-compliant metamodels, and the transformations themselves can be viewed as models. Whilst OMG-compliant implementations of QVT exist (the most notable being the Eclipse MMT project [48]), many model-to-model transformation languages (e.g. Tefkat, jQVT and most notably ATL) are “QVT-like” and trade strict adherence to the specification for better usability and/or performance.

Although most M2M languages are text-based, this is not (and need not be) always the case, especially given the complexities of some M2M languages. Graphical model transformation tools such as Henshin [49] provide state-of-the-art model transformation capabilities based on cutting-edge research in the area (for example, critical pair analysis) and various optimisations with a graphical syntax. The transformation language is backed by a metamodel. It also has code generation capabilities as well as an interpreter. The Eclipse Henshin project is a good example of a compilation of research in the model and graph transformation communities being put into practice.

2.1.5.2 Model Validation

Amongst the uses of OCL (and indeed the rationale behind the name) is to enforce constraints on models. A fundamental shortcoming of metamodels is that their structure alone cannot enforce certain desirable well-formedness constraints. For example, suppose we have a metamodel of Java and want to ensure that all classes which implement the "Comparable" also implement the "Serializable" interface. This can be achieved in OCL as shown in Listing 2.1.2.

```
context AbstractTypeDeclaration
    def: implements(type : String) : Boolean =
        self.superInterfaces->exists(si | si.type.name = type)

context ClassDeclaration
    inv comparatorImplementsSerializable:
        self->implements('Comparator') implies
        self->implements('Serializable')
```

Listing 2.1.2 Model validation example using OCL for Java metamodel

The `context` keyword sets the context for the following expressions to the specified model element type. The `inv` keyword declares the start of an invariant (aka a constraint), and finally the expression which follows returns whether the invariant is satisfied. The `self` keyword refers to the instance of the model element under evaluation, which is based on the context. That is, the invariant is evaluated for all model element types matching the specified context. OCL also allows for user-defined contextual operations, as shown by the "implements" operation.

The Eclipse OCL [50] implementation allows for constraints to be embedded within an Ecore metamodel using "OCLInEcore", or to enrich an existing metamodel using a standalone OCL document. Either way, the task of imposing such constraints is commonly referred to as *model validation*. In a typical modelling workflow, validation is often the first model management activity performed on models, since models that are malformed are unlikely to be useful in further activities. By asserting a set of assumptions on the metamodel, we can be more confident in the correctness of other model management activities.

We will discuss model validation in greater detail in section 4.1.

2.1.5.3 Model-to-Text Transformation

Model-to-text-transformation (M2T) refers to the process of transforming a model into a textual format. It is a special case of model transformation. Where model-to-model (M2M) transformation can be defined as activities "which take one or more source models as input and produce one or more target models as output, following a set of transformation rules." [14], M2T is more specific in that instead of transforming one model into another (which could take various non-textual forms), the goal is to generate text; which is often code. For this reason, the terms "M2T" and "code generation" are often used synonymously. Amongst the many uses of M2T are "to implement code and documentation generation, model serialisation (enabling model interchange), and model visualisation and exploration." [51]. By transforming a model into text, it allows engineers to work with tools which are not model-aware or support MDE tools and techniques. The typical use case for M2T is to generate executable and/or lower-level artifacts which can be used by tools that rely on textual inputs. The generated text may be anything at all, but often it represents (a subset of) the model in a textual form, such as a CSV file(s) or XML documents. Of course, one of the most common uses is transforming models into code, which is one of the most fundamental reasons for adopting MDE in some cases (see section 2.1.7).

Model-to-text transformations are typically performed using a template-based DSL [51]. With this approach, the language behaves somewhat similarly to languages like PHP combined with HTML; where there are "static" and "dynamic" regions. The static regions appear as-is, so any text in these regions is considered a string to be outputted as oppose to commands in the language. Dynamic regions are sections where there is some computational logic expressed in the language. For example, when generating HTML, a static region of a template may be as follows (Listing 2.1.3):

```
<html>
    <head> <title>Page Title</title> </head>
    <body>
        <table>
            <tr>
```

Listing 2.1.3 Static section of a template

By contrast, the following is a dynamic region because it is programmatically specifying what the output should be, as shown in Listing 2.1.3.

```
[% headings.forEach(th => println("<th>"+th+"</th>")); %]
```

Listing 2.1.4 Dynamic section of a template

As illustrated above, the main idea of a template-based M2T language is that it allows the developer to express the output based on a given model. Static regions (which are the same every time) help to improve readability, whilst dynamic regions (which vary by model instances) specify the logic for the transformation. In the above example, the "headings" variable could be a collection derived from a model. What makes M2T languages useful is the ability to take model elements and access their properties in an object-oriented manner and use these in the output. This allows for a flexible approach to the transformation. For instance, we may decide to exclude the "`table`" in the above example if the "`headings`" element is empty, as shown in Listing 2.1.5.

```
<html>
<head> <title>Page Title</title> </head>
<body>
[% if (headings.size > 0) { %]
    <table>
        <tr>
[% headings.forEach(th => println("<th>"+th+"</th>"));
}%]
```

Listing 2.1.5 Static and dynamic sections in a template

These examples are written in EGL [52]. We will return to this language in section 7.1.1.

Template-based model-to-text transformations which use a mixture of static and dynamic content may appear to be somewhat similar to modern programming languages which have convenient syntax for string literals (e.g. multiline strings), string concatenation and using expressions within string literals to access dynamic content. As such, there isn't a hard border between what is considered an M2T language and what is a standard feature in a general-purpose language. For example, one could feasibly use Python for M2T, or Xtend [53]; an Xtext-based language which adds syntactic sugar to Java, and is commonly used for code generation in Xtext projects (note that the M2T project called *Xpand* [54] has now been merged with Xtend). There are also more specialised code generation tools (commonly known as "*template engines*" which, in line with the Model-View-Controller design philosophy, separate data from presentation (or in practical terms, design in HTML from dynamic content in Java), such as Apache Velocity [55] and Freemaker [56].

Whilst there is no shortage of technologies for web development in this regard, the task of model-to-text transformation is much more generalisable. Once again, the Object Management Group has defined a standard called *MOFM2T* [57]. There are multiple implementations of this standard in the Eclipse M2T project [58] – most notable being Acceleo [59], which is a mature, OMG-compliant and popular tool for template-based model-to-text transformations over EMF models.

Given the vast choice of M2T tools available to developers, it is useful to have some way of comparing them. Rose et al. propose a feature-based model for classifying these languages [51]. The model is essentially a checklist of features which a transformation language may have. The top-level features are as follows [51]:

- **Transformation style**: This describes the control-flow of the language. An imperative style would have a main entry point and then procedurally apply the templates, whilst a declarative style applies templates to model elements less predictably.
- **Templates**: This describes how transformations are decomposed into templates. Sub-features include the ability to restrict when certain templates are applied or not, how static regions are escaped and the way in which model elements are loaded into the templates.
- **Output**: This describes where the text is output to (for example, the console, a single file or multiple files) and any post-processing before the generated text is sent to the output destination; such as formatting and "beautifying" for consistency.
- **Modularity mechanisms**: These describe the language constructs which allow the developer to re-use transformation logic. For example, these may comprise of being able to extend the language itself, inherit templates or make use of functions.
- **Traceability**: This feature describes how to navigate between source models and generated text. In other words, it's the ability to see which parts of the model (the input) contributed to which parts of the text (the output).
- **Incrementality**: This feature describes the way in which a change in the text or source model affects the execution of the transformation. For example, does the language provide constructs to preserve hand-made changes to the text after it's been generated, or are all changes overwritten? If something in the source model changes, when running the transformation, is everything re-computed and generated, or is there a "change log" which allows the language's execution to detect what's changed and only regenerate the parts of the text that are affected?
- **Directionality**: This describes whether the language supports bi-directional transformations. All model-to-text languages, as the name suggests, facilitate constructs to take as input a model and produce as output some text. However, sometimes it may be useful to reverse-engineer this process – that is, given some text generated by a transformation, can we get back the model which was used originally? This is called Text-to-Model transformation (T2M).
- **Tools**: Although not directly related to the language itself, this feature describes the additional tooling and support available which can make it easier for developers to use the language. These may include the ability to navigate between templates and generated text, or a "template extractor" utility which can take as input examples of the text to be generated and output a template which produces the examples.

In terms of what's important in the M2T process overall, Rose et al. identified four main concerns [60]:

- **Repeatability**: A change in the model shouldn't require major changes to the transformation logic. The transformations should also respect hand-made changes to the outputted text so that it isn't overwritten.
- **Traceability**: For debugging purposes, it is useful to be able to determine where each part of the source model is used in the generated text.
- **Readability**: The generated text should be human-readable and preserve things like indentation and layout so that it can be modified and understood easily.
- **Flexibility**: In order to promote re-use and avoid duplication of effort, M2T solutions should be flexible. For example, instead of being strict with what source models or elements can be used in a template, the solution should allow parameterisation of templates and transformation logic, and to make use of some functional idioms.

2.1.6 **Epsilon**

Whilst there are many different tools for each model management task, they can differ substantially in their style of use, semantics and syntax. Furthermore, most tools are designed to be used with a particular modelling technology in mind, meaning that users need to rewrite their model management programs when they want to migrate their models to a different format. It would be more productive if model management tasks could be expressed more easily using familiar syntax and semantics independently from the modelling framework.

Epsilon [61] ("*Extensible Platform for Specification of Integrated Languages for mOdel maNagement*") is an Eclipse project which aims to simplify model management by using a common core language and toolset. Epsilon supports most model management operations with a domain-specific language for each task, including pattern matching (EPL) [62], model-to-text transformation, (EGL) [63], model-to-model transformation (ETL) [64], model merging (EML) [65], model validation (EVL) [66], comparison (ECL) [67], migration (Flock) [68], textual modelling notation (HUTN) [36] and refactoring (EWL) [61]. These languages build on top of the Epsilon Object Language (EOL) [69], which claims to be "a mixture of JavaScript and OCL" [70]. It has a simple syntax with the usual imperative programming constructs as well as operations (functions). It also supports lambda expressions for querying and modifying collections. EOL supports four collection types (from unordered and non-unique to ordered and unique) and has commonly used first-order logic methods built-in. The language is quite flexible as most of the validation is performed at runtime, so it is duck-typed (has a "`var`" keyword). By building domain-specific languages atop a common, feature-rich model-oriented language with familiar syntax and semantics to many programmers, Epsilon has a relatively shallow learning curve which can enhance productivity. For instance, functionality common to multiple model management tasks can be expressed as operations in a separate EOL file. Other language features supported by EOL include extended properties (i.e. properties associated with individual model elements) and contextual operations (i.e. operations associated with particular types, including model elements).

One of the key features of Epsilon is that the model management tools and languages are decoupled from the underlying modelling technologies. This is achieved through the Epsilon Model Connectivity Layer (EMC), which provides an API for accessing models from EOL. This abstraction layer makes it possible to add new modelling APIs without having to change any model management scripts. Epsilon currently supports a diverse range of modelling formats, including Ecore, XML and comma-separated values. The architecture of Epsilon is summarized in Figure 2.1.4. For an introduction to Epsilon, its languages and development tools, see Kolovos et al. (2006) [4].

There are two main interesting aspects of Epsilon. First is its decoupling of its model management languages and the modelling technology. The architecture notably abstracts the notion of a model into a thin, standardised API called the Epsilon Model Connectivity layer (EMC). This API has the notion of types and kinds; where the former is exact type and the latter includes sub-types (if appropriate / supported by the modelling technology). It also supports CRUD (Create/Read/Update/Delete) events through property setters and getters and the ability to create or delete model elements. Finally, EMC supports the notion of IDs, allowing model elements to be retrieved by an ID or for the ID of an element to be looked up. However, it is worth bearing in mind that EMC is only an API and the features it supports may or may not be applicable or supported by a specific implementation or modelling technology. Implementations of EMC are model *drivers*, of which there are many built in to Epsilon. The most commonly used driver is EMF, which supports all features of EMC. This abstraction layer enables interoperability with a wide range of modelling technologies, including EMF (which is the *de-facto* modelling framework) as well as proprietary ones such as Simulink and MetaEdit+, amongst others. Moreover, widely used persistence formats (not just in the MDE domain) such as XML, CSV and spreadsheets such as Excel are also supported, allowing projects to more easily make use of Epsilon's model management facilities since no data migration is required.

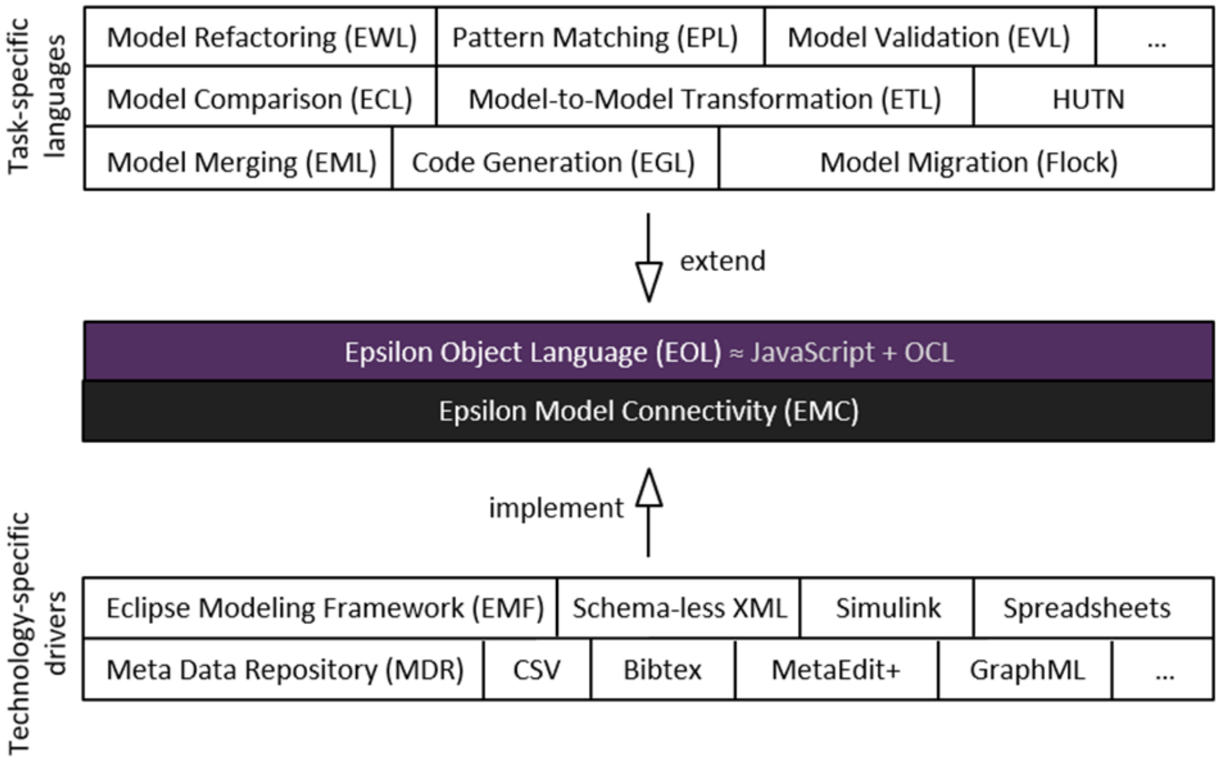


Figure 2.1.4 Overview of Epsilon's architecture [71]

The second interesting feature of Epsilon is its languages, which are implemented in Java and are interpreted. Although the Epsilon Object Language is inspired by OCL, with compatible syntax and built-in operations, it offers many more advanced features and slightly different semantics, especially regarding exception handling. For a start, EOL enables access to Java, so users can access properties and method on objects. It also allows for imperative programming, with mutable global variables and operations. There are many more features of EOL which make it far more powerful and expressive, so developers can always fall back to its imperative constructs (which include traditional while loops, if statements, break/continue etc.) and still take advantage of its convenient syntax and the model connectivity layer for uniform data querying. Given all Epsilon languages extend EOL, they are an interesting hybrid, with a declarative structure yet expressive, familiar and Turing-complete programming constructs for defining business logic.

2.1.7 **Model-Driven Engineering in practice**

(Declaration: this subsection is based on a part of the author's MSc thesis literature review).

To appreciate the broader context of this research, it may be useful to examine the experiences of using model-driven engineering technologies in industry. Specifically, we highlight some of the common motivations and benefits for adoption of MDE.

There is an increasing amount of literature which attempts to assess the real-world benefits of model-driven engineering techniques through case studies, where such approaches and tools have been used in an attempt to solve problems. Mohagheghi and Dehlen (2008) conducted a meta-analysis of 25 empirical studies on industrial applications of model-driven engineering. They sought to answer three research questions [7]:

- *Where and why is MDE applied?*
- *What is the state of maturity of MDE?*
- *What evidence do we have on the impact of MDE on productivity and software quality?*

This provides a convenient structure to the remainder of this section, where we can examine some specific case studies as well as meta-analyses of both literature and industry.

2.1.7.1 Applications of MDE in industry

In response to where MDE is applied, Mohagheghi and Dehlen found that the majority resided in telecommunications, business applications and financial organizations; with only two papers in the defence & aero sector and two in web applications. Product-lines, safety-critical and embedded systems were amongst some of the applications. A more recent (and comprehensive) series of publications by Whittle, Hutchinson and Rouncefield examine the industrial adoption of MDE technologies. In one publication in 2013, they "*surveyed 450 MDE practitioners and interviewed 22 more from 17 different companies representing 9 different industrial sectors*" to assess the successes and failures of MDE in industry [72]. The interviews and questionnaires gathered responses from a wide variety of experience levels with MDE. Amongst their findings were that many companies develop lots of domain-specific languages (DSLs), perhaps even one for each project, as this can be done relatively quickly. They found that contrary to popular belief, MDE is more widespread than many would believe, with applications in automotive, finance, printing and web applications [72]. Indeed, there are a number of case studies in applications of MDE to particular domains such as web applications (e.g. Kraus et al., 2007 [73]) and User Interfaces (e.g. Sottet et al., 2007 [74] and Vanderdonckt, 2008 [75]). Like Mohagheghi and Dehlen, Whittle et al.'s findings suggest that an MDE process is rarely pursued to develop entire systems throughout projects, but rather MDE techniques are used in combination with other methods when appropriate.

Successful use of MDE requires domain knowledge, but often there is a separation between those with technical expertise and those that know the domain well [72]. Perhaps for this reason, Whittle et al. found organizations that develop generic software (as opposed to those that target a particular domain) are less likely to see the benefits of MDE, and thus less likely to adopt it [72]. By contrast, domain-specific applications tend to involve some kind of modelling – even if it's informal or used for illustration purposes only – so those kind of projects and organizations may be more likely to adopt MDE because they can use models as development artefacts rather than throwaway sketches. Perhaps it is not surprising then to find (successful) applications of MDE in fields such as web apps, user interfaces and telecoms as these are typically designed around some domain-specific requirement. Whittle et al. also found that although MDE is often synonymous with code generation, the benefits of MDE are mostly holistic and that code generation is not the main driving factor for MDE adoption [72].

It appears that based on the evidence from industry, adoption of MDE technologies is to some extent dependent on organizational (and psychological) factors. Whittle et al. conclude that MDE is unlikely to be successful in organizations which are inflexible, autocratic and unwilling or unable to integrate new technologies into existing processes [76].

2.1.7.2 Benefits of MDE

With regards to software quality, based on the studies presented in the paper by Mohagheghi and Dehlen [7], there seems to be unanimous evidence that not only are there fewer bugs in the software (around 3 times less), but also that it is easier to detect defects earlier in the development process (by approximately 30% compared to rigorous inspections). The time to apply fixes to defects is also significantly reduced (in the order of 30-70 times faster) [7]. They also found the motivations for using MDE were basically to reduce development time and improve quality; be it through automation, standardisation or improving communication [7].

In terms of productivity, the evidence presented in the paper by Mohagheghi and Dehlen shows mixed results. Many experiments find that, at least in the short term, there is little increase in productivity, and sometimes a decrease depending on the project's complexity. This is perhaps not surprising, given that developers are likely to encounter significant learning curves when using MDE tools and techniques for the first time, whilst being intimately familiar with their existing tools and languages. The authors noted that productivity losses were mostly a product of immature tools and initial costs, and that "*modelling can be at least as complex as programming with a traditional third generation language.*" [7]. Although their study suggests a wide variance in productivity benefits (with a range between 27% loss to 800% gain), the typical scenario is a 20-30% productivity increase – a figure which may not be significant enough to offset the training and development costs in some organizations [72].

Weigert and Weil (2006) summarised the productivity sources of MDE as follows [77]:

• Design models are easier and faster to produce and test

• Labour-intensive and error-prone development tasks are automated

• Design effort is focused on applications, not on platform details

• Reuse of designs and tests between platforms or releases is enabled

• Design models can be verified through simulation and testing

• Design models are more stable, complete, and testable

• Standardized common notations avoid retraining of engineers

• The learning curve for new engineers is shortened

Whittle et al. (2011) also highlight the communication benefits from using models:

"*Organizationally our respondents suggest that the benefits of MDE can especially be seen in terms of improved communication and control within the organization, with how MDE has ensured wider, and perhaps better, organizational knowledge and communication.*" [76].

Floch et al. (2011) [78] attempted to use MDE tools to develop optimizing compilers for two different languages. In doing so, they found a significant number of benefits - especially with regards to the tooling. In particular, they emphasize the ease of defining and making changes to DSLs using tools such as Xtext, stating that "*the use of model based tools makes it easier to maintain the consistency between the parser and other components.*". They also mention other modelling tools available on the Eclipse platform, such as editors which allow easy visualisation of transformation rules, or languages like OCL (Object Constraint Language) [79] which allow for more advanced validation rules to be specified [78]. Another benefit which is in line with the rest of the literature is that the metamodel serves as documentation, since it captures the problem domain without excessive detail. This is especially important because many projects tend to be lacking in documentation, which makes it easier for new developers to understand the design decisions. However, as the title of the paper suggests, the authors find some important challenges that current MDE tools face. One of the main issues mentioned is that of scalability (as also identified by Whittle et al. [80]). With very large models, they found performance to be inadequate for industrial-grade compilers; even in the absence of demanding execution times.

2.1.7.3 Motorola case study

Although the industrial case studies of Mohagheghi, Staron and Whittle et al. suggest that the "current" state of practice in MDE is relatively immature – due to tooling, support and organisational factors – this is not to say that MDE is not worth adopting. An early example of an MDE success story comes from Motorola - a telecommunications company and mobile device manufacturer. This is an interesting case since, at the time of the study Motorola had over 15 years of experience in using MDE and reported a 200-800% increase in productivity [7]. This suggests that, given time and experience, MDE is effective in the long term compared to traditional software development techniques. Baker, Loh and Weil (2005) from Motorola present this case study in detail [81].

Interestingly, most of the benefits of MDE in the Motorola case seems to have come from code generation [77], which contradicts the findings of Whittle et al. at the industry-wide level [72]. Even with regards to the tools, Motorola claimed "*Code generators have reached a level of maturity that effectively no errors are being introduced into the resultant code.*" [77]. One should bear in mind that this was at a time when MDE was still in its infancy, at least in terms of industrial adoption and awareness. Over a decade later, one would hope that the academic literature, development tools and general awareness of MDE would be more mature, or at least no worse. Thus, it would not be surprising if the productivity benefits using modern tools and approaches may be even greater than what this case suggests. Furthermore, benefits were also apparent in platform targeting, where each platform may have different requirements in terms of performance and availability. Using MDE allows for greater experimentation with low-level implementation independently of the high-level design. In Motorola's case, this reduced the time needed to port code onto different platforms by 8 times [77]. In terms of code quality, the authors stated:

"*The models required as input for code generation are more complete and can be verified through simulation (or other techniques), resulting in significant quality improvements. The quality impact of model-driven engineering is dramatic and almost guaranteed.*" [77].

The success of MDE at Motorola did not come without challenges though. They found that although part of the difficulty in adopting MDE was due to the inflexibility of the culture, the major obstacle was a lack of a well-defined MDE process, as well as tooling [81]. The problem with the tools, at least at the time of publication, was the lack of compatibility between various tools. For instance, Baker et al. reported that migration from SDL to UML was an issue because of inadequate tooling; which was especially problematic given that the company had spent over a decade using SDL [81]. They also encountered performance issues with both the tools and the generated code; particularly with model management operations on large models [81]. This had a knock-on effect on the scalability of MDE, both due to performance and the incompatibility with legacy software [81].

Another concern with the tooling is that of integration of tools [81]. The "workflow" of the MDE process involves many stages, from designing models and expressing them using DSLs to validating instances and generating code. Whilst tools exist for each stage of the process, there was not a unified MDE environment at the time [81]. Due to the development of proprietary tools, "*This leads to a variety of testing solutions which are often duplicated, not well defined, and poorly supported.*" [81]. Indeed, Motorola actually developed their own code generation tool; which they claimed as being superior to third-party tools because it was tailored towards their particular domain and platforms; so the generated code was not only more optimized but also more complete and able to validate models better than vendor tools [81]. This is perhaps not surprising, and is in line with the studies of Whittle et al. Since much of MDE is domain-dependent, it is difficult to create a comprehensive toolset which is general enough to avoid duplication but specific enough to capture the domain(s). In the former case, one would end up with a language not too dissimilar to a third-generation programming language, and in the latter, almost every project would require its own toolkit, which increases development costs. Despite the overall success of MDE at Motorola, the challenges encountered were in line with the findings of the literature; namely that of tools. Of course, Whittle et al.'s (2013) [80] investigation came at a much later time than the Motorola case study, but similar issues seemed to have persisted over time.

2.1.7.4 Conclusion

In summary, a review of the literature suggests that current MDE tools are by no means perfect, and suffer from lack of scalability, incrementality and, in some cases, compatibility. They are sometimes too powerful (and hence, not entirely domain-specific or easy to learn) or operate at a lower level of abstraction than is desirable. Whittle et al. concluded that "*MDE tools could definitely be better. But good tools alone would not solve the problem*." [80]. It therefore appears that appropriate tools are a necessary but insufficient condition for MDE to thrive, based on the studies presented. However, if suitable tools for a given task are used correctly and with the right expertise, a model-driven architecture can provide a myriad of benefits to a variety of stakeholders; as evident by relatively early case studies such as that of Motorola [81]. That said, MDE is not a "silver bullet" with regards to reducing complexity of large projects. Where a model-driven design is successful, it is often applied to specific parts of a system to solve a specific problem with specialist tools or a domain-specific language. That said, there are some universal benefits which an MDE-based implementation can provide; namely that models serve as a documentation of the design. The abstraction techniques and automation provided by MDE can also help to tackle increasing complexity in modern software development [7].

2.1.8 **Motivation for Improving Performance**

We have seen in the previous section that one of the motivating factors for adoption of MDE is to increase productivity (often through automation). Due to the potentially large upfront cost of adopting model-driven engineering, combined with its potential advantages in improving communication between stakeholders, it can be argued that the benefits of MDE will be more apparent in larger projects with a diverse group of people involved at each stage. It is therefore unsurprising to see MDE being used in domains such as aerospace and automotive, with applications involving real-time and safety-critical systems. With large and complex projects come large and complex (meta)models. One such example cited in the MDE literature (such by [82]) is the models of the Automotive Open System Architecture (AUTOSAR) [83]. AUTOSAR is a partnership in the automotive industry to establish a standard software architecture for Electronic Control Units (ECUs). The AUTOSAR methodology involves the use of a very large and complex metamodel, with some models containing over a million elements [84]. Other areas with elements in the order of millions include civil engineering models (Building Information Modelling) and product-line architectures [84].

Another notable source of very large models (VLMs) is *model-driven reverse engineering* (MDRE). As explained by Brambilla et al. in chapter 3.3 of [13], most organisations face challenges with upgrading and modernising their legacy systems. The problem with trying to improve existing systems or replace them is that they tend to be functional and meet the business objectives but are inefficient. Rebuilding a system from the ground-up has a significant upfront cost (in both temporal and monetary terms) and may be difficult to integrate with existing systems and processes, or to construct in such a way that is correct – that is, preserving the functionality of the previous (legacy) system(s). Ideally, we would like to be able to re-use aspects of the established, functioning system in ordered to guide the development of the new system to minimise issues associated with compatibility, correctness, specification etc. Brambilla et al. state "*the first problem to be solved when dealing with evolution and/or modernization of legacy systems is to really understand what their architecture, provided functionalities, handled data, and enforced business rules and processes actually are.*". The process of discovering these is *reverse-engineering*. Model-driven reverse engineering presents an opportunity to obtain and comprehend this information at a higher level of abstraction using a three-stage process. The first step is *Model Discovery*, where existing (heterogenous) artifacts such as code, configuration and databases are parsed into models. Such models need not necessarily be high-level at this stage and may be very closely linked to their respective artifacts. The next stage – *Model Understanding* – refines these derived models through queries and transformations into a model which better represents the system. The final stage is to use these models to generate the new system – for example source code, configuration files, database structures and documentation. A notable model-driven reverse engineering tool is *MoDisco* [85], which although is a complete reverse-engineering framework, is perhaps best known for its ability to turn any Java project into a single EMF model conforming to a Java AST metamodel. Essentially MoDisco's model discovery for Java is a text-to-model transformation. The resulting model of legacy source code can be validated and transformed using MDE tools and eventually used to produce documentation, refactor code etc.

In all of the examples mentioned, very large models pose significant challenges in productivity due to the inability of most modelling tools to cope with models of such sizes. Kolovos et al. (2009) note that a typical concern in industry when deciding to adopt model-driven engineering practices is whether the tools can handle large models, incremental updates and collaborative development [9]. Scalability is a broad and widely cited concern within the MDE literature with increasing research interest, especially from well-regarded groups such as AtlanMod [86]. In fact, scalability in the model-driven engineering domain is so prominent that a Specific Targeted Research Project of the Seventh Framework Programme for research and technological development (FP7) funded the MONDO research project, which was set up to investigate and address some of these challenges [87]. Kolovos et al. (2013) state that, amongst other concerns, achieving scalability involves "*advancing the state of the art in model querying and transformations tools so that they can cope with large models (of the order millions of model elements)*" [88]. Figure 2.1.5 shows the main scalability challenges in MDE along with their relations and pre-requisites.

Of course, scalability itself is only one challenge facing model-driven engineering. The recent discussions at workshops titled "*Grand challenges in model-driven engineering*" highlighted a much broader range of long-term challenges (see Bucchiarone et al., 2020) [89], as evidenced by Figure 2.1.6. Arguably however, scalability is a much more immediate and ubiquitous practical challenge.

Sottet and Jouault (2009) present a compelling case study for which the execution time of model queries and transformation are desirable [90]. The idea is to construct a control-flow graph and program dependence graph from a given set of Java program models. The motivating factor is that documentation may be scarce or non-existent, so in order to help programmers understand a large and complex code base, it would be helpful to programmers to be able to visualise the code base at a higher level of abstraction. However, constructing such graphs from a very complex model with lots of elements is both time-consuming and memory-intensive, which is undesirable in integrated development environments (IDEs). Since the purpose is to aid program comprehension in a timelier manner, execution times of queries over the model (code base) and transformations into the appropriate control-flow and dependency graphs should be minimised.

Another case where low execution times are crucial is in real-time systems. More specifically, live processing of sensory inputs such as video or audio presents a significant challenge for current MDE technologies. For example, Dávid et al. (2014) present a case where data from a motion sensor needs to be update a model 25 times per second [91]; - even then, it is not difficult to envision cases requiring higher framerates (e.g. 60 FPS or greater) with more complex metamodels. The current modelling tools and frameworks are unsuitable for such applications because they were designed with the assumption that models do not undergo frequent changes which require immediate re-execution of various model management tasks in a short period of time. In cases such as sensory input processing, not only does the modelling framework need to be able to consistently handle many changes continuously and in a very short time period, but also to offer reactive mechanisms to immediately trigger the re-execution of various model management tasks (e.g. validation and transformation) upon those changes occurring, as well as producing the results of those programs within milliseconds.

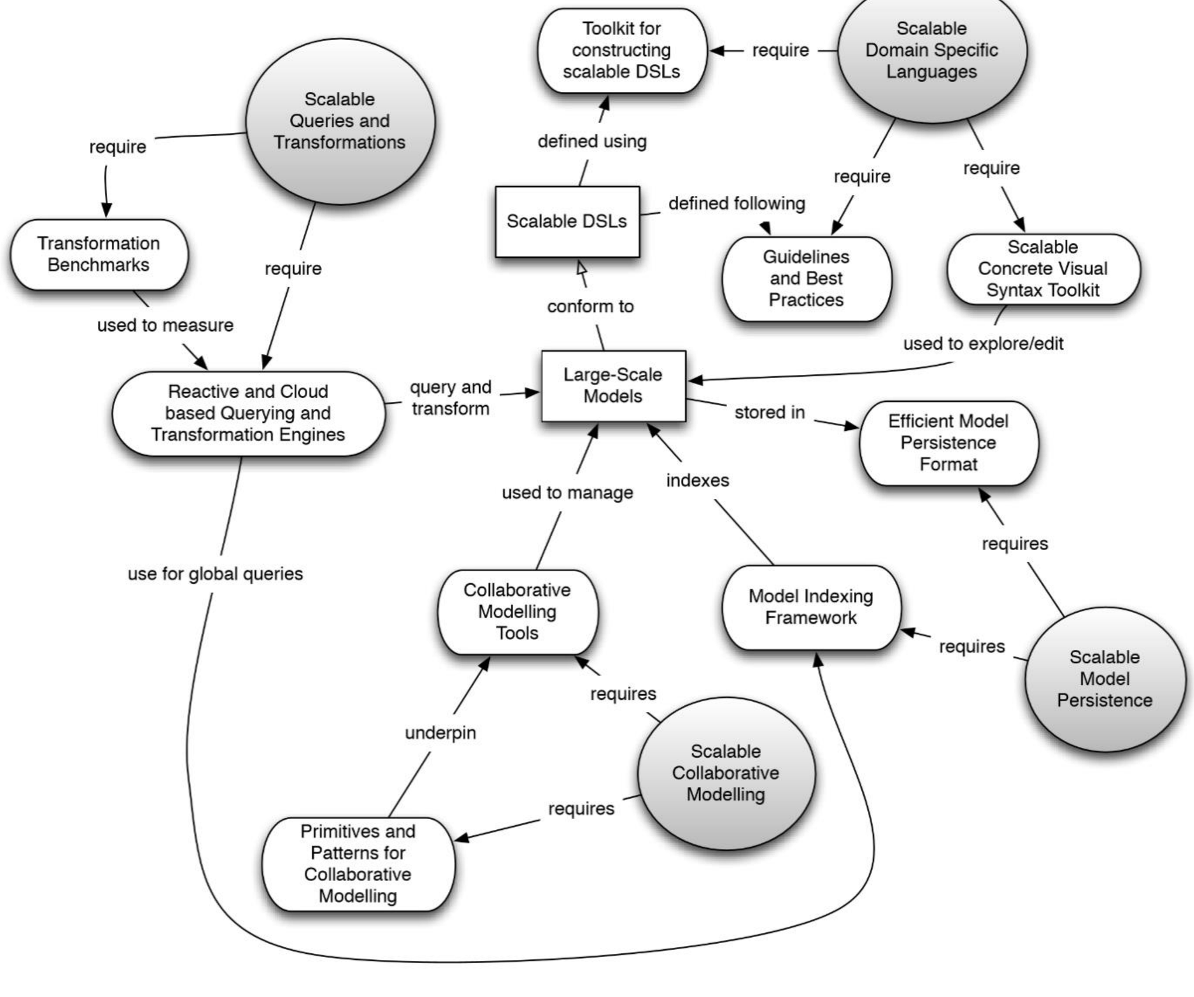

Toolkit for constructing scalable DSLs
require
Scalable Domain Specific Languages
Scalable Queries and Transformations
require
defined using
require
require
Scalable DSLs
defined following
Transformation Benchmarks
require
Guidelines and Best Practices
Scalable Concrete Visual Syntax Toolkit
conform to
used to measure
used to explore/edit
Large-Scale Models
Reactive and Cloud based Querying and Transformation Engines
query and transform
stored in
Efficient Model Persistence Format
indexes
used to manage
requires
use for global queries
Collaborative Modelling Tools
Model Indexing Framework
requires
Scalable Model Persistence
requires
underpin
Scalable Collaborative Modelling
requires
Primitives and Patterns for Collaborative Modelling

Figure 2.1.5 Overview of scalability challenges and solutions in MDE [85]

Although model-driven engineering technologies are maturing, the scalability aspects leave much to be desired. It is perhaps unsurprising that developers of MDE tools did not consider or attempt to address such challenges in early development stages. After all, building scalable tools and programs to address such use cases is not an easy task, and the future of MDE, the size of models and requirements of users were perhaps unclear when MDE was in its infancy. Either way, few would argue that scalability is a higher priority than, for example, correctness or (perceived) productivity. However, in order to make MDE more suitable for a broader range of applications, domains and requirements, the challenges of scalability need to be addressed. We have seen that scalability itself is a multi-faceted challenge (as shown in Figure 2.1.5 and [88]). This research specifically aims to address the problem of slow execution of model management programs over very large models.

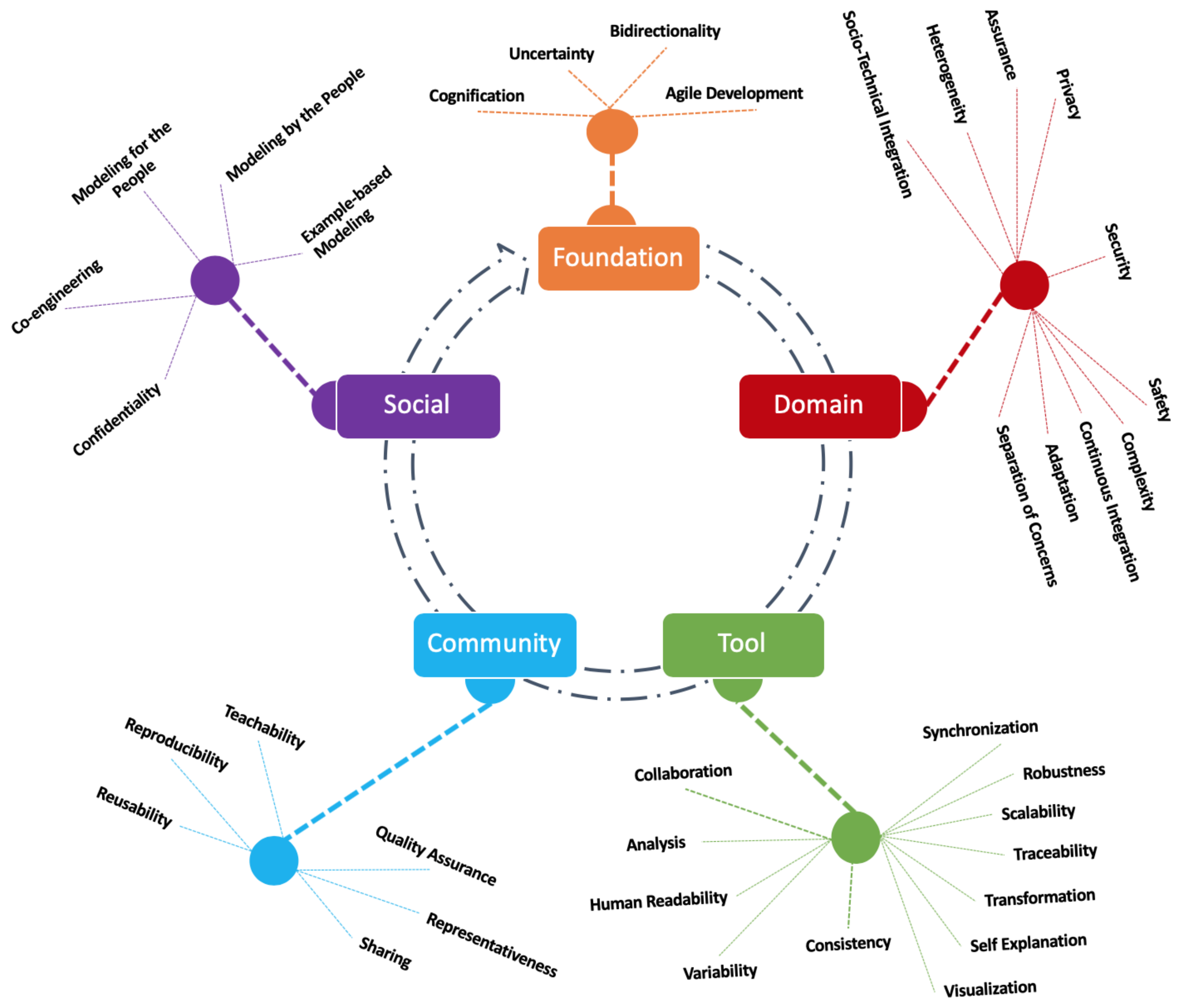


Figure 2.1.6 Taxonomy of challenges in model-driven engineering [89]

Much of the motivations for improving tooling and performance is pre-emptive research: as the saying goes, prevention is preferable to a cure. Even still, one could argue that research regarding the practical challenges of model-driven engineering, such as scalability and specifically performance of executing model management tasks, is long overdue. The oft-used analogy of "chicken-and-egg" situation comes to mind: industry practitioners may be sceptical of adopting model-based development due to concerns over tooling, support and whether the approach will still be relevant, especially given the costs involved in adopting MDE as described in the previous section. Meanwhile, in the absence of widespread adoption, it becomes more difficult to justify the significance of further research into what some may view as a somewhat niche or heavyweight approach to development. Whilst the benefits of MDE are numerous and well-known by experienced MDE practitioners and academics, the onus of demonstrating the advantages and feasibility of MDE to a wider audience lies with both industry and academia. Further collaboration and cross-disciplinary research is needed to promote MDE. By conducting research into well-known and long-standing concerns surrounding MDE, it shows a degree of confidence to sceptics and the curious regarding the relevance and enthusiasm for model-driven development.

Whilst very few experts would argue that MDE is the next "silver bullet" or that it is suitable in almost every large engineering project, regardless of its ubiquity MDE is still important to the "classic" industries that still use it. That is to say, even if the benefits and appeal of MDE do not extend beyond existing users, those industries and use cases are arguably significant enough to warrant continued investment and research into improving the MDE ecosystem. Finally, it is important not to underestimate the number of stakeholders in model-driven tooling and research. The nature of MDE lends itself well to large projects involving proprietary artefacts. When an MDE approach is used, these important artefacts become models; or at least much of their significance is expressed in models. Unfortunately for researchers, this makes it difficult to point to a large repository of projects involving big models to use as justification for their research. For the uninitiated and inexperienced MDE practitioners and academics, much of the research on self-proclaimed "important" challenges in MDE appear to be solutions seeking a problem. On the contrary, personal experiences of academics and industry professionals, especially those who have strong links between academia and industry, can attest to the relevance and significance of such research. It is an unfortunate matter in the MDE domain that evidence of large models, long-running model management tasks and other performance issues caused by inefficient tooling is extremely difficult to cite due to non-disclosure agreements and the high stakes involved in proprietary engineering projects, which have long been the main beneficiaries of MDE. Current examples of MDE users are Mathworks (particularly for Simulink), Rolls-Royce (aerospace) and DAF Trucks.

2.2 Computational Paradigms

This section explores various programming and computing paradigms which differ from the traditional software development environment; namely a single-threaded processor and application written in an imperative style and, typically, without any further optimisation procedures for scenarios requiring high scalability and performance requirements. From this section onwards, the term "non-sequential" shall be used to refer (broadly) to this definition.

The aim of this section is to make the reader aware of the various computational approaches – through both hardware and software – to improve the performance of programs in general. The applicability of these techniques to model-driven engineering are explored in section 2.3.

2.2.1 Motivations

As discussed in section 2.1.7, the property of *scalability* – or rather the lack of it – is a major shortcoming of current model management tools. Bondi identified the lack of scalability in a system or process to mean "*that the additional cost of coping with a given increase in traffic or size is excessive, or that it cannot cope at this increased level at all.*" [92]. For our purposes, "cost" refers to execution time of computations. Minimising the cost requires awareness of the tools one has at their disposal – both in terms of hardware and software.

Assume that theoretically, a piece of software is perfectly written and optimised for the hardware it runs on. Then, the runtime performance of that software is limited by the hardware's capabilities. Thus, the performance of any software is highly dependent upon the hardware that is used to execute it. Equally however, for superior hardware to be beneficial to performance, the software must be written in manner to take advantage. Traditionally, this process has been implicit and almost automatically guaranteed: for example, if processor A can execute more instructions per second than processor B (through higher clock speeds and/or superior architecture), then assuming binary compatibility of the software with both processors, the software is expected to perform better on processor A. For decades, software developers have relied on increasing clock speeds of Central Processing Units (CPUs) as well as more instructions per clock (IPC) for improving the performance of their applications. In this scenario, the software rarely, if ever, needs to be modified to benefit from hardware improvements. Even with additions to the CPU instruction set which may improve performance, a (high-level) programmer could simply wait for the underlying compilers to support such instructions and/or optimisations and recompile their source.

This reliance on "automatic" improvements in performance from hardware innovation is no longer an optimal strategy for improving performance. This is because of manufacturing and physical limitations; which have ultimately guided the design and architecture of modern processors. Namely, the advent of *multi-core processors* (the use of multiple logical CPUs on a single die) has been a promising and now ubiquitous characteristic of modern processing units. The first multi-core processor was the IBM POWER4 in 2001 [93], and ever since, multi-core processors have become so mainstream that the last single-core (desktop) CPU (Intel Celeron G470) was released in 2013 [94]. Indeed, it is becoming increasingly difficult to find single-core processors outside of extremely specialised embedded systems. Nearly all modern mobile devices have multiple processing cores.

Although smaller manufacturing processes have enabled slight increases in clock speeds, these increases are diminishing. It is rare to find CPUs with (stock) clock speeds over 5 GHz [95] [96] (at least, at the time of writing). Most high-end desktop solutions ship with stock clocks of around 4 GHz; and (stock) clock speeds of over 3 GHz have been available since the early 2000s [97]. Achieving higher clock speeds is becoming increasingly difficult due to physical limits of in semiconductor manufacturing. Furthermore, the falling demand for desktop PCs and increasing popularity of computing on more mobile devices such as laptops and tablets (which have limited battery life and space for cooling solutions) have meant that clock speeds for most computers have actually remained relatively stagnant (around 3 GHz at the time of writing); at least relative to older desktop CPUs. Even still, overclocking records of modern CPUs are only around twice as fast as stock clock speeds of top-end desktop CPUs despite using bespoke cooling solutions such as liquid helium or liquid nitrogen – for example, 7.5 GHz on a 7th-generation Intel Core i7 [98] and 8.7 GHz on an AMD

FX-8370 [99]. Since, for a given processor, increasing the clock speed has roughly a linear impact on performance, this limits the performance improvements possible from purely increasing clock speeds. Perhaps another limitation of clock speed increases are the disproportionate increase in power consumption and heat dissipation [100] [101]; which is hardly ideal in an era of increasing reliance on portable computers and efficient design.

The incremental and diminishing nature of CPU clock speeds and IPC improvements [102] mean that it is no longer optimal for developers to rely on improvements in single-threaded performance of CPUs; and this has been the case since circa 2003 [103]. Meanwhile, the number of threads which CPUs can execute simultaneously has been steadily increasing because of higher core counts and simultaneous multi-threading (SMT) technologies such as Intel's Hyper-Threading [104]; which allow a single physical core to execute multiple (usually two) threads simultaneously, albeit with shared access to many of the core's resources. Thus, the raw performance increases in modern CPUs mostly come from the ability to execute multiple threads simultaneously.

Although modern multi-core processors are more powerful than traditional single-core processors, Blake et al. note that this "*represents a significant gamble because parallel programming science has not advanced nearly as fast as our ability to build parallel hardware*" [105]. Consequently, we have at our disposal a large amount of computing potential which often goes underutilised. Since many tools that are still in use today were originally written at a time where multi-core processors were relatively new and/or uncommon (or at least built on top of older tools, frameworks or platforms), they were not designed to take advantage of multiple processors. It may be the case that software applications typically start small with prototypes, adding features as they are needed. However, designing for scalability requires an early architectural commitment. Arguably, current model-driven engineering tools did not make this commitment in the initial design; hence improving their performance by exploiting the capabilities of modern hardware is a non-trivial task.

For a more thorough and detailed analysis of the motivations for parallel programming, see "*Structured Parallel Programming: Patterns for Efficient Computation*" (McCool et al., 2012) [101].

2.2.2 **Concurrency**

In order for software to benefit from multi-core processors, it must be written with multiple threads of execution. That is, the software needs to be able to spawn multiple tasks to execute independently of each other. A *thread* can be thought of as a list of instructions grouped together; - essentially like a "lightweight process" [106] or a sub-program. Sun Microsystems defines a thread as "*A sequence of instructions executed within the context of a process.*" [107].

Although multi-threaded processors are relatively new in computing, multi-threaded programs are not. The study of *concurrency* has a long history in computer science. Concurrency is a property of a system and can be defined as "*A condition that exists when at least two threads are making progress.*" [107]. Note that this does not require a multi-threaded CPU, since concurrency is not the same as parallelism [108]. That is, a program can be said to be concurrent if multiple threads of execution are "alive" and making progress, but this need not be simultaneous. On a single-core system, multiple threads could be alternating in execution, for example. Thus, it is important to distinguish between *Concurrency* as a general property of a software system, and *Parallelism* as a special case of concurrency. Specifically, parallelism can be defined as "*A condition that arises when at least two threads are executing simultaneously.*" [107]. So, whilst concurrency requires the presence of multiple threads within a process, parallelism requires that some of these threads make progress *at the same time*. Therefore, true parallelism can only be achieved if a processor can support simultaneous execution of multiple threads.

Perhaps unsurprisingly, concurrent systems bring with them a multitude of challenges and complexity in dealing with multi-threading. Primarily, these challenges stem from the need for threads to communicate with each other and contention for shared resources between threads. For instance, a common problem with multi-threading is when multiple threads read from and write to state variables – known as the "Readers/Writers problem". The issue arises because if thread A is in the process of updating the variable and thread B tries to also update (or read) the value of the variable before thread A completes, then the perceived value of the variable between threads will be in an inconsistent state. When the scheduling of threads is non-deterministic, this can lead to race conditions as illustrated by the Reader/Writers problem. Furthermore, non-deterministic behaviour can make programs more difficult to test.

The problem of contention for shared resources is often illustrated by the "Dining Philosophers" problem [109]; in which there are five philosophers sit at a round table, and in between in is a fork (five in total). The idea is that each philosopher requires two forks to begin eating, and the philosophers cannot communicate. The challenge is then how to design a deterministic algorithm such that all the philosophers can eat fairly. Incorrect or inefficient solutions can lead to *deadlock* or *livelock*. In the context of the Dining Philosophers, a deadlock can occur when, for example, each philosopher has picked up the fork on their left and so none of them can eat because they are all infinitely waiting for the philosopher on their right to release their fork. The conditions for deadlock were illustrated by Coffman (1971) [110] as requiring *mutual exclusion* of resources, no *pre-emption* (resources must be given up voluntarily by the holder), *hold and wait* (waiting for a resource while holding another) and *circular wait* (a circular chain of waiting for resources). Like deadlock, livelock is when threads are unable to progress, though due to infinitely responding to other threads when communication is permitted [111]. In the Dining Philosophers example, if the philosophers could communicate, then a co-ordination issue may arise where each philosopher offers their fork to their adjacent philosopher; leading to none of them eating.

It is apparent that most of the complexities of concurrency arise from shared resources. Intuitively, if each thread could work independently of other threads without competing and/or co-ordinating access to shared resources, then each thread would be its own program. Goetz proposed three solutions to the problem of shared variables between multiple threads [112]:

- "*Don't share state variables across threads;*"
- "*Make the state variable immutable; or*"
- "*Use synchronization whenever accessing the state variable.*"

Thread safety indicates whether a state variable can be "safely" accessed by multiple threads without unexpected results – i.e. the behaviour is "correct". Bloch defines five levels of thread safety, whilst Goetz advocates three levels: *Immutable*, *Thread Safe* and *Not Thread Safe* [112]. Both

Bloch and Goetz emphasize immutability as preferable solutions where possible. The reason immutability guarantees thread safety is because immutable variables are, by definition, read-only and thus the order in which threads access them is immaterial. Furthermore, it is perhaps the simplest solution to many potential concurrency issues and adheres to other best design practices.

Where immutability or avoiding shared state are impractical, the only remaining solution to achieving thread safety is to co-ordinate access of state variables, such that only one thread at a time can access a given variable. This is known as *mutual exclusion*, or *synchronization*. The idea of mutual exclusion and its importance; as well as a solution was first proposed by Dijkstra (1965) [113]. A *mutex* (or *lock*) is a simple synchronisation mechanism which acts as a flag (two states – "locked" and "unlocked") for shared variables. For example, in Java, every object can act as a lock for a method or block of statements. The idea is that before entering a `synchronized` block or method, the executing thread attempts to acquire the lock on the specified object before executing the statements. When a thread holds a lock, no other threads can acquire the lock until the thread holding the lock releases it. However, excessive synchronization can lead to contention, since other threads requiring access to a shared resource must wait until the lock is released. Synchronisation using locks effectively makes regions of code which depend on shared state single-threaded; which undermines the benefits of using multiple threads to improve performance. Bloch also warns against the dangers of excessive synchronization [114]. Often, a thread may require access to shared resources if a certain condition holds. Rather than acquiring a lock and then wastefully "spin-waiting" for the condition (e.g. in a `while` loop), a more sophisticated synchronization mechanism may be used. A *monitor* is a mutex with a condition variable or expression. The idea is that after acquiring the lock, a thread can suspend execution based on a condition; freeing up the CPU to execute another thread instead. When the condition is signalled (i.e. becomes true), the waiting thread is notified and resumes execution (see [115] for an example).

A common pattern in parallel algorithm design is to decompose tasks using a "divide and conquer" approach. This is most applicable when the sequential algorithm is recursive. The approach is also known as the "*Fork/Join* model", and is centred on the idea of recursively splitting and processing tasks in parallel until a certain minimal threshold is met, in which case the computation is performed directly with no further splits [116]. In this model, the number of jobs (i.e. the parallel scalability) is determined by the number of splits. There is a balancing act between ensuring sufficient scalability to take advantage of more processing threads, and managing the overhead of splitting, creating more jobs and merging the results. For example, the parallel array sorting algorithm in Java uses 8192 as the baseline for sequential processing [117].

Some have even gone so far as to criticise the most conventional and perhaps most primitive model of concurrency; which is using threads directly. Lee (2006) [118] advocates against using threads directly, arguing that they are too low-level and allow programmers to make fatal mistakes due to their non-deterministic execution. He advocates for higher-level constructs which are more structured and easier to reason about, although he acknowledges the adoption challenges with such alternatives. The main benefit of using threads is that they can be added to sequential programming languages with relatively minor changes, hence the popularity. The fact that threads can be mapped to the kernel and hardware also means they have relatively low overhead; although as a model of computation they are quite heavyweight. Modern languages like Go and Kotlin allow for more fine-grained concurrency with lower overhead than managing threads directly within the programming language, shielding the user from how their computations are mapped to operating system threads. Recent endeavours within the OpenJDK project to introduce Fibers and Continuations in Java [119] are motivated by the need to simplify asynchronous programming [120].

The subject of concurrent programming, its challenges and solutions are subjects of many books, research papers, theses, tools and projects. This section has introduced a small subset of these concerns to make the reader aware of and appreciate the complexities associated with concurrency. Next, a brief overview of some practical solutions to simplify concurrent programming are presented. For a brief summary of various concurrency models, see the blogpost in [121].

2.2.2.1 Actor model

Most formal approaches to concurrency are in the family of “process calculi”, the most well-known being Communicating Sequential Processes (CSP) as proposed by Tony Hoare [122]. CSP is useful for modelling the behavioural aspects of a system through algebraic notation. The main idea of CSP is (synchronous) message-passing and describing interactions between sequences of events.

Another similar but arguably more powerful approach to concurrency is the Actor model, originally proposed by Carl Hewitt [123]. The Actor model is similar to CSP, except that it is asynchronous. There are two main concepts; *actors* and *messages*. Actors are computational entities that can respond to messages in the following ways:

- Send messages to other actors
- Create other actors
- Designate behaviour for the next message it receives

These actions can be performed independently of each other. The Actor model is at its core a message-passing system, with actors having identities (“mailboxes” / address). Actors can only message other actors if they have their address, which can be obtained from messages they’ve received or actors they’ve created. They can only interact with each other through (asynchronous) messages, so message ordering doesn’t matter – messages are simply sent (like packets in a network), so there is no need for (synchronous) handshaking with the receiver. Actors are reactive – they only respond to messages received, though their behaviour is deterministic based on the message contents. A well-known implementation of the Actor model is Akka [124], which underpins many of the widely-used distributed processing frameworks in the Java ecosystem. The Actor model is adapted to work in a distributed environment, although the behavioural semantics are the same irrespective of the topology. One could model complex concurrent systems through actors and messages – for example, a thread pool can be thought of as an actor system where the threads are the actors and the jobs are messages.

Since the actor model allows actors to have identity and for messages to be directed towards selected actors, more fine-grained controllable is available. For example, in a distributed system, each computer (node) could have a single “master” actor for co-ordination, and *n* children (workers), where *n* is the number of hardware threads. There could then be a global actor – the root of the actor system (i.e. the parent of all “master” actors) – which co-ordinates execution and sends jobs to nodes. Specific jobs could be mapped to a chosen node based on its environment / resources, and each node's master actor deals with delegating messages containing computationally expensive tasks to its workers – similar to how a thread pool would – and dealing with control messages such as termination signals, exceptions etc.

In one sense, the Actor model can be used to conceptually describe many other message-passing frameworks, such as those based on the Java Messaging Service (JMS). JMS is an API specification in Java Enterprise Edition designed for transactional processing [125]. The "Actors" in JMS are *MessageProducer*s and *MessageConsumer*s, which are tied to a *Session*. A Session represents a transaction – i.e. sequence of communication between actors. Producers and consumers send and receive *Message*s to mailboxes known as *Destination*s. A Destination may be a *Queue* – where it is delivered exactly once and consumed by a single MessageConsumer – or a *Topic*, where the message is broadcast to all MessageProducers subscribing to the Topic. A *Message* may contain arbitrary serializable data, but also has useful metadata for message sequencing, identification etc. Finally, each *Session* is established from a *Connection*, which represents the connection between the client and "server". This system of communication is facilitated by a *Broker*, which usually uses TCP/IP for communication. The "Actors" connect to the broker through a Connection obtained from a *ConnectionFactory*. The broker is responsible for handling the delivery and persistence (if required / supported) of Messages to their Destinations.

2.2.2.2 Structured Parallel Programming

Whilst the difficulties with concurrency and parallelism are infamous and often feared, solutions to these challenges often tend to be theoretical in nature and difficult to apply in practice. Much of the literature on concurrency involves process algebra and formalisms (see for example, CSP) which abstracts away from the programming aspects and focuses on conceptual modelling, event sequencing, interactions etc. Whilst concurrency may be better understood in these abstract worlds, the challenges and bugs present themselves in the actual programming, even if theoretically plausible solutions exist for a given parallelisation problem.

Structured Parallel Programming, as presented by Murray Cole, can be viewed as an attempt to bridge this disconnect between theory and practice. In his presentation, Cole (2004) [126] argues that "*Many* [if not most] *parallel applications don't actually involve arbitrary, dynamic interaction patterns.*". He views parallel programs as patterns of interaction, where the patterns are often predictable and thus can be templated. Although a parallel program may appear to be non-deterministic, it is sometimes "*constrained within a wider pattern*". He continues: "*The use of an unstructured parallel programming mechanism prevents the programmer from expressing information about the pattern - it remains implicit in the collected uses of the simple primitives.*". In other words, the typical low-level approach to parallelism in practice using primitive concurrency constructs means the actual intent and pattern of interactions is never expressed explicitly by the programmer. It is as if the programmer is given two-dimensional objects to build a complex three-dimensional structure. Perhaps this low level of abstraction is where concurrency bugs are introduced. To remedy this unstructured, implicitly patterned style of parallel programming, he advocates a more structured approach based on predefined patterns. He defines Structured Parallel Programming as follows:

"*The structured approach to parallelism proposes that commonly used patterns of computation and interaction should be abstracted as parameterisable library functions, control constructs or similar, so that application programmers can explicitly declare that the application follows one or more such patterns.*".

Note that unlike other concurrency solutions, this approach does not require a fundamental conceptual restructuring to adhere to a rigorous and overly constrained style. Rather than requiring conformance to a strict concurrency system such as CSP, Structured Parallel Programming permits the generalisation of parallel algorithms into library functions or language constructs. Essentially Cole argues for more high-level utilities for implementing parallel algorithms so that the programmer can select a suitable pattern and use it simply in a declarative style. To demonstrate, he presents the *eSkel* tool which contains various parallel patterns, referred to as "*Skeletons*".

2.2.2.3 SCOOP

SCOOP (Simple Concurrent Object-Oriented Programming) is a concurrency model aimed at simplifying concurrent programming in object-oriented languages. Originally proposed by Meyer (1993) for the Eiffel programming language [127], and later refined by Nienaltowski (2007) [128], SCOOP provides a higher level of abstraction for concurrency by relieving the programmer from dealing with explicit multi-threading and synchronisation. Syntactically, only a single keyword – "separate" – is added to the host language, and a source-to-source compiler can be used to interpret uses of the *separate* keyword and generate the native low-level constructs which deal with multi-threading and synchronisation. Although originally designed for and implemented in Eiffel – a contract-based object-oriented language, many of SCOOP's concepts can, in theory, be applied to any object-oriented language. This was demonstrated by Torshizi et al. (2009), who provided an implementation in Java ("JSCOOP") [129].

The *separate* keyword indicates that the specified object is handled by a different handler to the current (executing) object. In SCOOP, every object has a single *handler* (also known as a *processor*), which is a conceptual thread of control. Although a processor can be the handler for many objects, each object's handler is fixed throughout the program. Every processor has a *call stack* and *request queue*. The call stack is a LIFO (Last In First Out) structure used to execute (non-separate) feature calls. The request queue is a FIFO (First In First Out) structure which pushes non-local (separate) feature calls (i.e. those issued by another processor) to the call stack. Local (non-separate) calls are executed on the stack before items from the request queue get pushed onto it. As a mutual exclusion mechanism, processors can be *locked* or *unlocked*. The *locked* state means the processor is being controlled by another processor and is taking requests from it (i.e. its request queue is open). An *unlocked* state means the processor is not able to accept requests from another processor (i.e. its request queue is closed). Upon completion of separate requests, locked handlers will unlock themselves asynchronously when they terminate; - i.e. an unlock request is appended to the request queue of each locked processor and the processor sending requests doesn't have to wait for the locked processor(s) to unlock before proceeding. Nienaltowski notes that all concurrent systems are a collection of interacting sequential systems [128], and in SCOOP, a sequential program is a special case of a concurrent one; since by default, SCOOP has a single processor which is the handler for the top-level object in the program.

It can be demonstrated that synchronization in SCOOP guarantees “fairness” in lock scheduling and also overcomes the “dining philosophers” problem [128] [129]. This is achieved by declaring each “fork” object as *separate* and, to avoid deadlock, a pre-condition on the “eat” method requiring both “forks” to be available before proceeding. Since conditional locking is not natively supported in all object-oriented languages, additional keywords must be introduced to support pre and post conditions (e.g. “await” in JSCOOP [129]). Another attractive property of SCOOP is that since each object has a fixed handler, there is no object sharing between threads. This implies that there can never be any “race conditions, and the mutual exclusion mechanism eliminates atomicity violations (although user-induced deadlocks are still possible) [128] [129]. Furthermore, since each handler can only be locked by one other handler at a time and executions take place in a FIFO order, there can be no harmful interleaving [128].

2.2.3 **Distributed Computing**

As stated in Section 2.2.2, *parallelism* is a special case of concurrency where multiple threads execute simultaneously. However, a more fundamental distinction is that parallel processing typically (but not always) implies thread independence. A typical parallel processing architecture would involve a main (or “master”) thread dividing up a task and assigning each part to “worker” threads. The essence of parallel processing is efficient computation [130], so if threads are regularly (if at all) depending on each other, then this is arguably not true parallelism. As we discovered in the previous section, dependencies lead to synchronization; which ultimately results in sequential computation despite the presence of multiple threads. For parallel processing to be efficient and provide performance benefits, the task must be inherently parallelisable [103]. That said, it is also possible to have a parallelisable task which is typically expressed in a non-parallelisable way – for example, calculating the sum of a list of integers [103]. In this case, a parallel solution may divide up the list into *n* chunks; where *n* is equal to the number of threads which can be executed simultaneously. Then each thread would calculate the sum of each sub-list, and when two threads have terminated, another thread can be spawned to calculate the sum of their results (assuming that the sum operation is computationally more expensive than spawning threads!) and so on until only one summation (and thus, one thread) remains. Although this task is not perfectly suited to parallel processing, it can be more efficient than a sequential solution; which would involve a mutable variable that accumulates the sum whilst iterating through the list.

A “perfectly parallelisable” task would involve no interdependence between subtasks, which also implies the lack of shared state; though mutability within each subtask (or thread) would not be an issue. For instance, suppose we have a set of 1600 numbers, a 16-core/16-thread processor and our task is to return a subset containing all prime numbers. This is “perfectly parallelisable” because we can create 16 threads and assign each thread 100 numbers to check independently; where each thread adds each prime number to a thread-local mutable set. Note that none of the 16 threads require any knowledge of the overall task or the presence of other threads, and so can be treated as independent programs. Once all threads have finished, the 16 subsets can be merged into a single set by the "master" thread. The anatomy of a parallelisable task involves three steps: 1) Dividing the problem into sub-problems; 2) Solving the sub-problems in parallel; 3) Merging the results of the sub-problems. Even if a task is suitable for parallel processing, a parallel algorithm will always do more work than a sequential one due to the overhead of dividing up the problem and merging the results [103]. Consequently, Goetz advocates the *NQ > 10000* condition for deciding if the overhead of parallel processing justifies the benefits; where *N* is the number of data elements and *Q* is the computational complexity of the task (or "work per element") [103]. In other words, a sequential algorithm always has a "head start" compared to a parallel approach; so in order for parallelism to be beneficial, the problem should be sufficiently large and/or complex.

So far, we have established that a parallel algorithm should avoid shared state and dependencies amongst sub-tasks. The type of parallelism we have discussed so far involves applying the same computation over different data. In Flynn's Taxonomy, this is analogous to SIMD or SPMD (Single Instruction Multiple Data and Single Program Multiple Data, respectively), and is known as *Data Parallelism*. An alternative approach would be to apply different parts of the computation to the whole data (analogous to MISD), or even different instructions for different parts of the data (MIMD). However, we must still be wary of the dangers arising from mutability of shared state. Task (or instruction-level) parallelism is well-suited to validation / verification problems – basically any task where the data is read-only. For example, suppose we're given three numbers and our task is to compute some mathematical properties for each number (e.g. whether it's a prime, whether it's a power of two, if the sum of digits add up to 9 etc.). Since none of these operations involve modifying the inputs, the data can be safely distributed across multiple threads, and each thread will compute a different property on the whole data. This alternative way of viewing parallelism – which will be referred to as *task parallelism* – allows us to consider a broader range of programs which can be parallelised and choose the appropriate style for the task. For instance, in following the *NQ > 10000* rule, if *N* is large and *Q* is small, it would be optimal to adopt a data-parallel processing style. In contrast, if *N* is small and *Q* is large, then task parallelism may be more appropriate.

A common approach to executing parallel programs is to use multiple networked computers. Distributed computing is a special case of parallel computing and the two terms are often used interchangeably; perhaps due to individual computers being traditionally treated as single-threaded machines. Raynal (2015) clarifies the distinction between parallel and distributed computing; noting that the geographical distribution of computers is a fundamental difference; since there is also a need to handle node failures and communication over networks [130]. In other words, parallel computing is *centralised*, whereas distributed computing is characterised by uncertainty [130]. However, if we assume a "perfect" environment (no failures, reliable and high-bandwidth network…) then a distributed computation can be expressed in a centralised way; and vice-versa [130]. From a programming perspective, if a problem can be parallelised – in other words, it can be split up into independent computations, then the decision as to whether the physical resources used to perform the computation are, for example, four dual-core computers or a single eight-core machine, is a separate concern. Dealing with the additional overhead of distributed computing need not be a concern of the core application logic, and can be handled by a framework; thus allowing the programmer to focus on expressing the "business logic".

Perhaps the main difference of concern to programmers and system architects between local parallel and distributed parallelism is the lack of shared memory. In a single-computer shared memory system, it is possible for different threads of execution to have access to the same memory – both volatile and persistent resources. In a distributed system, processing needs to be either stateless (i.e. no access is required to data held in other computers) or the information needs to be replicated (copied) to nodes. Such replication may mean copying the required data to all nodes prior to parallel processing, or it may require communication between nodes. In the latter case, the crucial constraint which separates distributed and local processing is the need for *serializability*. The data which is shared needs to be self-contained, having no references to in-memory structures from its origin node. For data-parallel distributed processing to be efficient, the data needs to be relatively self-contained, small and easily decomposable into primitive forms which can be sent over-the-wire. In simple terms, this means regular objects need to be decomposed into strings and numbers, which may require a substantial re-design or a different approach than local parallelism.

In distributed computing, co-ordination of nodes – especially with regards to shared memory – poses a significant challenge. Linda is a co-ordination language developed in the mid-1980s for parallel and distributed computing, based on the idea of a shared virtual memory space using *tuples* [131]. A tuple is a list of objects (with arbitrary data types). The idea is that there is a shared *tuple space* in which the memory may be physically distributed but logically accessed as unified memory by multiple (distributed) compute nodes. Linda itself is not a programming language, but rather a concept which can be integrated into a sequential host language, such as C or Java [131]. In Linda, processes are completely decoupled – their interaction with other processes is through the tuple space only, and processes are anonymous [131]. That is, there is no "addressing information" regarding processes; no processes knows about any other processes, and interactions with the tuple space are asynchronous. There are four basic operations which can be performed on a tuple space [132]:

- ***in***: Atomically consumes (reads and removes) a tuple from the tuple space.
- ***rd***: Non-destructively reads from tuple space.
- ***out***: Produces a tuple, writing it into tuplespace (tuple may be duplicated in tuplespace).
- ***eval***: Creates new processes to evaluate tuples, writing the result into tuplespace.

The input and output operations also have non-blocking predicate forms. Addressing works by associative matching, where a subset of tuples in the tuplespace are retrieved based on the specified parameters in the provided *anti-tuple*. An anti-tuple is a pattern specification which may also use wildcards to match any tuple with the specified structure.

Linda itself doesn't provide any synchronisation mechanisms or guarantees of safety, though common solutions can be easily implemented [131]. conceptually, Linda is a very simple and flexible idea and well-suited to parallel tasks. Since there are no overheads associated with "safe concurrency", there is potential for relatively good performance (assuming an efficient implementation) compared to alternatives. Recall that for effective parallelism, sub-tasks (or processes) should not communicate with each other. If the tuples in the shared memory area are of appropriate structure, and the parallel computation is optimally expressed, then there should, in theory, be no need for synchronisation primitives. Each process could read from and write to a subset of the tuple space, and when all tasks have completed, the "master" process can consume all elements from the tuple space to produce the final result.

That said, Linda in its most basic form is far from ideal with regards to efficiency; especially when it comes to complex queries. For example, attempting to find the maximum value of some field would require the program to consume all of the tuples containing the field by a single process and searching sequentially for the value [131]. Furthermore, the consumed tuples may not be accessible by other processes during the search [131]. To address the shortcomings of associative addressing, Wells proposes *eLinda*, which allows for more powerful and flexible pattern matching criteria than the standard Linda model [131]. This is achieved by using a *Programmable Matching Engine;* which would extend querying operations (such as *in*) with additional functionality, including aggregate operations, and could perform complex queries in parallel [131].

MPI (Message-Passing Interface) is a standardised message-passing API designed for parallel computing (more specifically, distributed computing) with distributed memory systems. The specification originated in the early 1990s [133], with several portable, high-performance implementations available for C, C++ and Fortran. At its core, MPI defines the behavioural semantics of message-based communication between processes; where typically each compute node is assigned a single process. The standard is ideally targeted towards portable high-performance computing with MIMD (Multiple Instruction Multiple Data) style computations [134]. Although MPI's concepts are rich, it originates from a time where multi-core processors didn't exist, and relatively low-level, imperative-style languages such as C/C++ were the de-facto standard. Consequently, MPI has been criticised for its inability to scale beyond 32-bit data (due to backwards compatibility) and the low level of abstraction compared to more modern alternatives like Spark and Akka [135].

One of the most successful and frequently cited works in the field of distributed computing is *MapReduce* [136] – a programming model devised at Google in 2003 where the (parallel) computation is expressed in two functions: *Map* and *Reduce*. Although MapReduce was originally devised for C++, the model is conceptually derived from the functional programming paradigm (discussed in section 2.2.8). In Java terms, the signatures of the *Map* and *Reduce* functions are as follows (recall that the Map data type in Java is itself a collection of key-value pairs)

```
List<Entry<K1, V2>> map(K1 key, V1 value)

List<V2> reduce(K2 intermediate, List<V2> values)
```

The idea is that for a given key-value pair, the *map* function emits a set of intermediate keys along with a list of values associated with each key. (Note that the types of the keys and values need not be the same as the input key and value). The results of the *map* function are then passed to the *reduce* function; which takes a given key and its associated values, and performs a reduction operation to return a (potentially) smaller set of key-value pairs. A commonly used simple example used to illustrate this is the task of finding the number of occurrences of each word in a given document [136]. The input to the *map* function would be the document (with the name as key and contents as value). The function would then add each unique word to the intermediate map, and for each occurrence of the word, the integer "1" is added to the collection of values associated with each word. That is, the result of the *map* function would be a mapping from words to a collection of 1s; so the size of the collection associated with each word is equal to the number of occurrences of the word. The *reduce* function then takes each word along with the collection of values associated with it (in this case, every element in the collection is the integer 1), add up all of the values and return another map, where the key is a word and the value is a single integer which is the number of occurrences of the word in the document.

MapReduce is a parallel programming model because both the *map* and *reduce* functions can operate on their inputs independently of other inputs. In the word count example, the *map* function could be called with many different documents; with all documents being processed independently of each other. Likewise, the *reduce* function can process each word (intermediate key) independently of other words. In some cases, there may be significant repetition of intermediate keys output from *map* tasks – for example, the word "the" may occur in almost every document passed to the *map* function, and since each invocation of *map* works independently, there will be duplicate keys. For such cases, a *Combiner* function may be defined to group together intermediate values with the same key before passing them to the reduce function.

The simplicity and utility of MapReduce led to the creation of distributed processing frameworks. Amongst the most well-known is Hadoop [137], which is an implementation of MapReduce with a related family of projects for distributed computing. Notable components of Hadoop include YARN; a framework for job scheduling and resource management / co-ordination, and HDFS; a virtual distributed file system.

In high-performance computing, there is generally a trade-off between *throughput* and *latency*. The former refers to the volume of data which can be processed in a given time frame (e.g. transactions per second), whilst the latter is a measure of how long it takes to process a transaction or data (e.g. the response time to a request made on a web server). Throughput is generally important when processing large volumes of data, often present in advance, with no real-time requirements. Low latency is valued in more reactive systems where the data or processing tasks are not known in advance and there are requirements for worst-case or average response times. Consequently, there is a spectrum between *batch* and *stream* processing, which emphasize throughput and latency, respectively. Hadoop MapReduce is an example of a batch processing framework, since the data is processed in bulk rather than one at a time. Most modern distributed processing frameworks place greater emphasis on latency than throughput, as the former is often a hard requirement whereas the latter is seen more as an optimisation as it is possible to have reasonable throughput with low latency. Successors to Hadoop / MapReduce include Spark, Storm and Flink. Frameworks such as Flink offer more generalised processing capabilities, expanding the available process functions to include operations such as filtering and grouping. Such frameworks treat data as streams (at least conceptually, but often also under the hood), even when the data is finite and available in advance. The main idea is that for a given data stream, each element goes through a processing pipeline, which may consist of an arbitrary chain of transformations (i.e. *map*) and filters before finally terminating in a *sink* (a function which has no return value). Sinks are typically used to gather the data in one place (e.g. the master) or to write the results to a file. With this approach, a trade-off between latency and throughput can be made by adjusting the batch size. For instance, if latency is strongly preferred, then each data element goes through the processing pipeline one at a time. If it is less of a concern than throughput, multiple elements may sent to each function at a time.

2.2.4 Automatic Parallelisation

As with concurrency, a challenge in parallel programming is how to parallelise a given piece of code. Even though the semantics of a given computation may be easily amenable to parallelisation, the programmer is still burdened with the task of creating the appropriate number of threads, partitioning the computation into the threads and starting the computation on these threads manually. However, this need not be the case, as there are numerous ways to automatically deal with multithreading; provided that the computation is indeed “embarrassingly parallel”.

A well-known method of automatic parallelisation is to use compiler directives and/or annotations on parallelisable sections of code. An intelligent compiler can then inject multithreading constructs into the code and infer the parallel regions. OpenMP is an API for shared memory multithreading managed by the OpenMP Architecture Review Board consortium [138]. Like MPI, OpenMP was originally designed for use in C, C++ and Fortran, though an open-source alternative – OMP4J – is available for Java [139]. OMP4J is a source-to-source compiler which translates code blocks commented with “omp” into multi-threaded code. The number of threads is based on the number of logical threads in the system, or they can be adjusted by the user. OpenMP allows for both task and data parallelism [140], since each thread runs the labelled blocks of code independently from other threads. Although OpenMP is a shared memory programming model, it also provides constructs for declaring thread-local (or “private”) variables. Since shared data can lead to various concurrency issues (as discussed in section 2.2.2), OpenMP also provides synchronisation constructs, and it is the responsibility of the programmer to use these constructs in a thread-safe manner. Although OpenMP doesn’t require the programmer to explicitly deal with threading and synchronisation, it does require the programmer to manually analyse parallelisable regions of code and express access to shared memory in an efficiently parallelisable manner. To get the most benefit from OpenMP, the programmer needs to have in-depth knowledge of the code and explicitly be able to identify and label appropriately (with correct OpenMP constructs) the parallel regions. The greatest benefit of this approach is that it allows for incremental parallelisation of existing sequential programs at the instruction level [140]. On the other hand, this can be time-consuming because it requires a lot of in-depth analysis of the code base to determine which sections can be safely parallelised; and choosing the correct OpenMP directives may also pose a challenge. Furthermore, since the actual parallel code is auto-generated, it may be difficult to diagnose errors in the program since, for example, line numbers and call stacks reported by debugging tools will be for the generated code as opposed to the annotated code. This solution is therefore more complex, as there is a strong dependency on OpenMP libraries and two “versions” of the source code.

A fully automated approach is possible using SequenceL – a simple, declarative, functional language along with compilers and tools which generate parallel, thread-safe code without any intervention required from the programmer [141]. The language of SequenceL compiles to optimised parallel C++ code; which makes it interoperable with languages that can also interoperate with C++ , such as Java (through Java Native Interface) or Python [141]. The most prominent advantage of SequenceL is that unlike directive-based approaches such as OpenMP, the level of abstraction with regards to parallelism is effectively maximised; requiring no input from the programmer to infer parallelisable regions. However, the guarantee of automatic parallelism and thread safety come at the cost of flexibility. Although SequenceL’s language is very high-level, it is also very limited in its features and functionality [142]; with a clear emphasis on mathematical and scientific computing tasks [143]. Unlike OpenMP, it requires the application to be (re-)written in SequenceL language, which is unsuitable for parallelising large existing programs. This is further complicated by the purely functional nature of the language – a completely different paradigm to most sequential programs, which are typically written in imperative languages – as well the lack of complex data types [144]. Perhaps the best use of SequenceL is not to re-write entire applications, but rather to express computationally expensive numeric parts of a program using the language (to take advantage of auto-parallel code generation), and then call the function from the language the main application is written in (e.g. using JNI to invoke the generated C++ code for use with a Java application).

So far, we have focused on tool-oriented approaches to automatic parallelisation. Although tools certainly have their uses, they impose a level of dependency on an intermediate language and/or compiler. Furthermore, they are either limited in scope (as with SequenceL) or require a lot of input from the programmer (as with OpenMP). An alternative approach is leverage the native language's constructs and use them to provide a higher level of abstraction, as opposed to relying on external tools for parallel code transformation. As discussed earlier, if we have lots of data (i.e. *N* is high in the *NQ* model), then data parallelism is likely to provide substantial performance improvements. Most programming languages provide data structures for storing large collections of data elements - typically in the form of arrays, trees or hash tables. One solution to raising the abstraction of parallelising (independent) operations on large amounts of data is to use a *stream*. A stream is an abstract pipeline of computation over a source of data. In a sense, a stream can be regarded as a *transformation* of data; except that typically it does not modify the source. The principle behind a stream is that it provides an extra layer of indirection for expressing computations on the underlying data source. Rather than writing imperative code which manually iterates over the data and performing operations in a loop, a stream instead allows the programmer to declaratively express the intended series of transformations using higher-order functions (see Section 2.2.7). This allows the programmer's intended results to be decoupled from the lower-level details, so the stream's implementation can deal with optimising the actual execution of the transformation without requiring the programmer to change their code. The rationale to this approach stems from the famous fundamental theorem of software engineering:

"*All problems in computer science can be solved by another level of indirection*" [145].

In Java for example, parallelising the execution of a stream pipeline is simply a matter of declaring the stream as parallel with a method call (from the application developer's point of view) [146]. A further discussion of Java Streams is given in section 2.2.8.

Although Java Streams are implemented in a private, non-extensible way, the actual concepts and API are purely interfaces and so alternative implementations are possible. The Streams API is convenient and built into the language's standard libraries, is compatible with various data sources such as arrays, collections and files and crucially, it has rich declarative functionality with lazy and parallel execution semantics usually only found in functional languages. In the previous subsection, we identified several tools for distributed processing, with frameworks in the Apache Hadoop stack such as Spark also being declarative in nature. However, many of these tools are written with Scala in mind, and understanding the subtleties and the API of such frameworks can be cumbersome for developers, especially as they are prone to change with many competing frameworks also available.

To minimise the burden and learning curve for developers, it would be a nice idea to be able to express computations using the Streams API whilst also scaling with multiple processes. Chan et al. (2016) [147] recognised this and developed a prototype implementation of the Java Streams API with distributed semantics, designed as a drop-in replacement for parallel streams. There are significant challenges with this as one would expect. Although the Streams API can work on files and its lazy semantics allow for infinite streams or data processing that would be impossible with eagerly loaded in-memory structures, the most common sources are of course in-memory Java collections. Unfortunately the standard Collections API does not support the notion of distributed datasets, where each node has a part of the data (i.e. the data is spread across multiple computers / file systems). There is also the logistical challenge of efficiently executing short-circuiting operations (such as *findFirst()*) in a distributed setting, since the required data elements may be on another node. Compliance with the lazy semantics and also being efficient whilst supporting all operations offered by the Streams API is no simple task.

Chan et al.'s prototype "DistributedCollection" interface includes the notion of a "ComputeGroup", which is a grouping (or cluster) of "ComputeNode"s, where a node is a participating computer. Data can be partitioned across nodes, and also saved from distributed streams to the collection. Of course a crucial aspect of efficiency in distributed settings is to maximise data locality to minimise communication between nodes. To facilitate this, their DistributedStream API defines the notion of a "Partitioner", which is a function that takes as input a data element and returns the index of a node in the ComputeGroup. For convenience, it is possible to split a Stream into multiple pipelines and also join multiple Streams into a single one (lazily of course) in order to be able to perform this partitioning of data processing. Thus the API provides a mechanism by which users can help the distributed execution strategy maximise data locality. In cases where communication is expensive (and often undesirable), the API provides local variants of the stream operations (such as *forEach*). The implementation uses an MPI-based library as the transport layer.

In terms of performance, Chan et al. compared their prototype implementation to Hadoop and Spark across six different use cases. Despite the maturity of Spark and Hadoop (which are large open-source projects with many high-profile users), their DistributedStream prototype was very competitive in execution time compared to these established frameworks. Their performance was generally better than Spark with smaller datasets but this advantage diminished almost linearly as the dataset grew. This is perhaps attributable to network communication overhead since the authors admitted their implementation resulted in more network traffic. Nevertheless, their work clearly shows that the declarative data processing approach offered by the Java Streams API is adaptable to distributed settings, albeit inevitably more complex. Except for short-circuiting operations, based on this paper we can conclude that declarative data processing (at least with the functionality offered by Java Streams) can not only scale with multiple cores but also multiple computers. That said, the level of automation is reduced in a distributed setting due to the need to better exploit data locality, inevitably requiring some further input from the user. In any case, the conclusion is the same: writing data processing logic in a declarative manner (such as that offered by the Streams API) allows single-threaded applications to transition to be able to make use of compute clusters with relatively minor changes to the code. Therefore, automatic parallelisation and distribution can be achieved in many cases so long as the processing logic is expressed in a declarative manner.

2.2.5 **GPU Computing**

Where massive data parallelism is concerned, we cannot ignore the power of graphics accelerators, commonly referred to as Graphics Processing Units (GPUs). They have been around for two decades (at the time of writing), however it is only in the past decade or so that their compute potential has become increasingly leveraged by a more diverse range of applications. Unlike CPUs, which have for the most part been relatively stagnant in their compute performance in recent years, GPU compute performance has been increasingly dramatically with every new generation from both AMD [148] and NVIDIA [149] (the two major companies in the business of high-end GPU architecture design). The power of GPUs comes from their relatively simple compute units which are highly optimised for integer and floating-point operations. Recent GPU architectures have become more complex and capable of more powerful operations, however the principle behind GPUs remains the same. Whereas a modern high-end CPU may have at best core counts measured in the dozens, GPUs have thousands of "cores" with clock speeds which are becoming increasingly higher (around 2 GHz) thanks to advancements in lithography fabrication process. A high-end CPU's raw compute performance measured in floating-point operations per second (FLOPS) is in the order of GigaFLOPS, whilst GPUs are in the order of TeraFLOPS. High-end GPUs are no longer predominantly the interest of PC gamers, but also of cryptocurrency miners, data scientists, AI researchers and other scientific, numerically-intensive compute tasks such as meteorology [150]. They also come equipped with plenty of dedicated high-speed, high-bandwidth memory, making them suitable for memory-intensive workloads as well. With so much parallel compute performance and increasing interest in General Purpose computing on GPUs (GPGPU), it is worth considering how GPUs can be leveraged to improve the performance of model management programs.

One of the biggest obstacles to leveraging GPUs is the limited and relatively low-level programming model. GPU architectures are highly parallel and are designed for Single Instruction Multiple Data (SIMD) processing – a form of data parallelism at the hardware level. The *de-facto* standard is OpenCL [151], which is a C++-based framework for writing programs which work on heterogenous computing devices (e.g. FPGAs, DSPs and GPUs). However a more popular (albeit proprietary) framework is NVIDIA's CUDA [152], especially amongst researchers and scientific computing applications. The programming model for CUDA is designed specifically for NVIDIA GPU architectures, and is based on the C programming language. According to the documentation, "*CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions*." [153]. Threads are organised into *blocks*, where each block shares memory and caches. Therefore kernels (the GPU-optimized code) must be written in a way such that blocks of threads execute independently and in any order. The programming model unsurprisingly relies heavily on exploiting the hardware parallelism, and given the extremely limited features supported by CUDA C compared to modern general-purpose programming languages, CUDA code is only suitable for data-parallel algorithms where the data consists of primitives (integers or floating-point values). Also, since there ideally should be little, if any, communication between threads and also ideally no branching logic (*if* statements, for example), GPU computing is not suitable for our use case, unless a decomposition of a proportion of the program can be found which meets the criteria required for benefiting from GPU-accelerated computation.

It is worth mentioning that GPU (and more general, *heterogenous computing*) was a very active research area in the late 2000s and early 2010s. One of the reoccurring themes was research into utilizing GPUs for *MapReduce* computations. Works such as MARS [154], MapCG [155], StreamMR [156] and HadoopCL [157] demonstrate how the *MapReduce* programming model can be adapted to make use of specialized compute devices like GPUs. However most work has not kept up to date with modern GPU architectures and utilising newer features which were previously unavailable, so their designs are most likely suboptimal. Furthermore, Despite the prevalence and increasing performance of GPUs, support for utilising GPUs from a high-level language like Java, at the time of writing, remains limited. Although there have been APIs to bridge the two worlds (for example, JCUDA [158] and JOCL [159]), these are still low-level and require programmers to write low-level code in the respective frameworks – they do not add native support. Slightly more advanced projects such as Aparapi [160] and even the OpenJDK Project Sumatra [161] were comparatively ambitious, however they are no longer in active development.

That said, some interesting developments in recent years seem promising for making GPU computing more accessible to JVM-based languages without the need to write any low-level CUDA or OpenCL code. For example, Ishizaki et al. (2015) [162] present a promising approach for leveraging Java's Parallel Streams API to make use of GPU without any additional coding by the user. The premise is a JIT compiler that (automatically) optimises parallel streams for NVIDIA GPU execution without sacrificing language features (e.g. virtual method invocation, exception semantics), without any language extensions, annotations or additional API. Their approach works at the intermediate representation (IR) level, after bytecode generation. Of course, GPU usage is only possible for primitives so they look for calls to *`IntStream.forEach`* with *`parallel()`* and transforms it to regular *for* loop; which can then be optimised further. The output is twofold: NVVM IR (native GPU code) and native (host) binaries which call CUDA APIs. Despite performing some advanced optimizations which are only possible through static analysis as well as completely eliminating any need for user-specified GPU code, their approach inevitably limits the functionality available to the Java programmer since not everything can be mapped to GPU code. These restrictions apply only to the code which is to be executed on the GPU (i.e. the Consumer parameter passed to *forEach*). Only three exceptions are supported (NullPointer, ArrayIndexOutOfBounds, Arithmetic), only one-dimensional (primitive) arrays can be used and Object creation (e.g. using new) are not supported. Nevertheless, this work shows that for specialized use cases where GPU acceleration makes sense, an automatic parallelisation / translation approach is possible.

Fumero et al. (2014) [163] demonstrate an array-based functional approach to automatic parallelisation which can leverage GPUs. The idea is root in "Array Function programming", where programs are composed as function applications (transformations) over arrays of data. This is inherently a data-parallelisable way of programming and so is suitably aligned with GPU architectures which excel at SIMD computations. Unlike Java Streams API, the authors' "ArrayFunction" API is re-usable as operations are not lazy. Inputs and outputs are always arrays. The actual execution on GPU uses OpenCL, which is generated. As with other works, the generator supports a subset of the language; only primitive arrays (and tuples), one-dimensional arrays, limited method calls etc. Data is marshalled to and from device using Java Native Interface (JNI), though the authors note that sending data, executing kernel and unmarshalling back to Java are done sequentially and could be parallelised in future work. If runtime errors are found, regular Java version of the code is executed instead on the CPU. Unsurprisingly, most of the overhead comes from marshalling/unmarshalling and sending/receiving data from GPU – kernel execution takes up smallest proportion of overall time.

Vuduc et al. (2010) [164] questioned the benefits of GPU computing at a time when GPGPU was a particularly hot topic. They argue that for “irregular” computations, GPGPU can sometimes only deliver marginal performance improvements over CPU-based computation – i.e. one can achieve similar results by adding more CPU cores – and whether of moving to a more restrictive programming model is worth the effort. They also note the high cost of GPU-specific data structures and host-to-GPU copies. Furthermore, optimal tuning has strong runtime dependence and given the rapid advancements in GPU technology (especially at the time), hardware changes would require re-writing programs not only for compatibility but also for optimal performance.

Having briefly considered the prospects of GPU-accelerated parallelism, given the narrow set of functionality available, utilising GPUs is beyond the scope of this project due to the limited software support for high-level programming languages and the lack of a clear use case (i.e. a suitable algorithm or SIMD-parallelisable code) where GPU computing would be beneficial in the context of model management programs.

2.2.6 **Incrementality**

So far, we have seen how multi-threaded processing can be used to improve runtime performance by exploiting parallel hardware. However, there is a more fundamental way to improve performance orthogonal to parallel execution: avoiding unnecessary (re-)computation(s). That is, rather than performing the same computation faster, we can also improve the efficiency of the program.

The notion of *incrementality* (or *self-adjusting computation*), as defined by Hammer et al. (2014), is “a technique for efficiently recomputing a function after making a small change to its input.” [165]. The idea is that if a function is invoked repeatedly with a large input where each invocation’s inputs vary only slightly from previous invocations, then it makes sense to exploit some of the partial results calculated in previous invocations rather than re-computing over the entire input each time. Thus, there is a one-time cost with incremental execution for a given input of size *N* and potentially zero cost for the same input thereafter; as opposed to a repeated cost of *N*. As an illustrative example, suppose we have a program which calculates the moving average of a list of numbers as data comes in. Thus, for each invocation, the list will only ever be added to (no existing elements will be changed or removed).

A non-incremental implementation (in imperative Java) may be implemented as shown in Listing 2.2.1. It may be invoked with the inputs shown in Listing 2.2.2. With this approach, the entire list of numbers is re-examined every time by the “average” function, leading to unnecessary re-computations. By exploiting the knowledge of the nature of inputs, we can eliminate unnecessary computations by implementing a simple caching mechanism, as shown in Listing 2.2.3.

```java
<N extends Number> double average(List<N> numbers) {
    double total = 0;
    for (N number : numbers) {
        System.out.println("computing..."+number);
        total += number.doubleValue();
    }
    return total/numbers.size();
}
```

Listing 2.2.1 Stateless *average* function

```java
List<Integer>
    input1 = asList(100, 20, 50, 16, 3),
    input2 = asList(100, 20, 50, 16, 3, 7),
    input3 = asList(100, 20, 50, 16, 3, 7, 44);
double
    avg1 = average(input1),
    avg2 = average(input2),
    avg3 = average(input3);
```

Listing 2.2.2 Inputs to *average* function

```java
int lastSize;
double lastAverage;

<N extends Number> double averageCached(List<N> numbers) {
    int size = numbers.size();
    if (size > lastSize) {
        double total = lastSize * lastAverage;
        for (int i = lastSize; i < size; i++) {
            N number = numbers.get(i);
            System.out.println("computing..."+number);
            total += number.doubleValue();
        }
        lastSize = size;
        lastAverage = total/size;
    }
    return lastAverage;
}
```

Listing 2.2.3 Example of an incremental *average* function

In the example above, the print statement “computing…” will only be executed for each new number added to the list. Also, when invoking the function with the same list, we can guarantee that the result will be the same so we can simply return the last result (this is known as *memoization*).

Fundamentally, incrementality is a space-time trade-off: we exchange increased memory usage (in the form of caching) for reduced computation time. In the example above, the memory usage is simply a couple of numbers, which amounts to a few bytes. In more complex cases, the impact on memory usage may be exponential if large result sets are cached for each invocation. It should be noted however that as with parallel computing, the motivations behind caching are also grounded on hardware developments. In the past decade or so, memory has become abundant and exponentially cheaper, where historically memory was a scarce commodity. Further still, advancements in non-volatile memory – namely solid-state drives (SSDs) – are now several orders of magnitude faster than traditional mechanical hard drives for I/O performance. For example, the Samsung 960 PRO is an M.2 form factor (similar size to most DIMM DRAM memory modules) NVMe SSD with up to 2TB capacity with sequential read speeds up to 3.5 GB/s and writes up to 2.1 GB/s – and this is a drive which launched in 2016 [166]. Therefore, even with a ludicrous amount of caching, given a large enough page file (in case of insufficient main memory), given the speeds and capacities of modern storage technologies, many applications need not be concerned with increased memory consumption brought on by caching.

The increased memory is not the only trade-off: the code is also significantly more complex and error-prone, making it more difficult to understand, maintain and increasing the cost of adding new functionality. Although most (performance) optimisations face this increase in complexity, incrementality not only requires a refactored design, but also makes strong assumptions about the data model and required functionality.

2.2.7 **Laziness**

Another approach to improving efficiency is to delay computations until the point at which the result is used. This is known as *lazy evaluation*. The idea is that with *eager evaluation*, some intermediate computation is performed before it’s necessary and may potentially never be used; thus being a wasteful use of computational resources. For example, consider the code in Listing 2.2.4. Since “if” statements in most languages perform short-circuiting, we expect to see a number being printed approximately 50% of the time when we run the code above. However, the code is inefficient because it still calls the “expensiveComputation()” function 100% of the time despite the result only being used less frequently. The solution comes once again from adding another level of indirection [167], as shown in Listing 2.2.5. Here, we have made the call to “expensiveComputation()” lazy by wrapping the result into a supplier function so that “expensiveComputation()” is only evaluated when necessary, and only once. In fact, some languages like Scala provide lazy semantics by simply marking a variable as “lazy”, and some (such as Haskell) even provide laziness by default [167].

```java
void run() {
    boolean someCondition = Math.random() > 0.5;
    int data = expensiveComputation();

    if (someCondition && data > 16) {
        System.out.println(data);
    }
}

int expensiveComputation() {
    System.out.println("computing...");
    return (int) Math.round(Math.random()*100);
}
```

Listing 2.2.4 Example computation based on random number generator

```java
IntSupplier dataProxy = new IntSupplier() {
    int result;
    boolean notCalled = true;
    public int getAsInt() {
        if (notCalled) {
            result = expensiveComputation();
            notCalled = false;
        }
        return result;
    }
};

if (someCondition && dataProxy.getAsInt() > 16) {
    System.out.println(dataProxy.getAsInt());
}
```

Listing 2.2.5 Lazy implementation example of Listing 2.2.4

Laziness need not apply only to computations. *Lazy loading* entails loading data into memory – be it volatile or non-volatile – only at the point of consumption. Once again, this can be achieved by another level of indirection. For example, some cloud storage applications such as OneDrive for Windows 8 provide a layer of virtualisation so that the user can see all the files stored in the cloud on their local machine, but they are only downloaded and stored if and when they are required/used [168]. Similarly, the Git Virtual File System (GVFS) by Microsoft provides a virtual layer which makes files stored in a Git repository appear to be present but only downloads them upon first use [169].

2.2.8 Functional Programming

So far, we have considered some approaches to improving the performance and efficiency of our applications. In practice, implementation of such measures is far from trivial. Brooks (famously) distinguished between *inherent* and *accidental* complexity in software [170]. Inherent complexity refers to complexity which is at the essence of a problem and cannot be reduced or simplified any further. For example, searching for the maximum value of an unordered set of integers requires the algorithm to look through each and every element (provided there is no upper bound on the maximum possible value). Regardless of the choice of tools and abstractions, nothing can change this fact – it is the *essence* of the problem. In contrast, accidental complexity arises from suboptimal choice of tools and/or implementation. Arguably, much of the complexity of concurrent and parallel programming is *avoidable*, and thus *accidental* [171]. The difficulties in avoiding accidental complexity are, at least in part, attributable to the paradigm of imperative object-oriented programming. This traditional, widely-used paradigm is perhaps not optimal for concurrent programming because of its sequential nature and emphasis on state [172]. Stateful behaviour is arguably the essence of object-oriented programming. As discussed earlier, state is not problematic if it is immutable. However, for historical reasons, mainstream programming languages, algorithms and programmers' mindsets have all co-evolved to make extensive use of mutable variables.

A promising alternative paradigm to help resolve the incompatibilities between concurrency and traditional imperative programming is *functional programming*. Some go as far as to claim that in a purely functional paradigm, concurrency is a non-issue [172]. This is because functional programming emphasizes the principles of immutability and statelessness [171]. Functional programming has its roots in *lambda calculus* [171] [173] – a formal, mathematical treatment of functions. Just as objects are "first-class citizens" in the OOP paradigm, functions are the first-class citizens of the functional programming paradigm. One of the defining features of functional programming is the notion of *higher-order functions*. A higher-order function is a function which accepts another function as a parameter and/or returns a function as a result [174] [171]. The uses of this are extensive, and most modern high-level languages support higher-order functions in the forms of *lambda expressions*. A lambda expression is an anonymous higher-order function; consisting of a parameter list (arguments to the function) and a body (the implementation) [175]. A simple example of a lambda expression (in Java) is as follows:

```
Arrays.asList(1, 2, 3).forEach(i -> System.out.println(i));
```

Note that the lambda expression is simply forwarding the value received as a parameter (an integer from the list) to another function (the `println` function). This is a *function composition*, and so can be expressed more concisely:

```
Arrays.asList(1, 2, 3).forEach(System.out::println);
```

The above example is the functional equivalent to a traditional *for* loop; except that instead of requiring the programmer to define the iteration mechanism (for example, initialising a variable, checking a condition and mutating the variable), it simply requires a declaration of intent. It is not uncommon to find the terms "functional style of programming" and "declarative style of programming” being used interchangeably: just as object-oriented and procedural programming are imperative in nature, functional programming is inherently declarative.

A further tenet of functional programming is that functions should be *pure*. A pure function is a function which has no side effects [171] [176]. In other words, for a given input to the function, the results will always be the same and have no effect on any global state. Of course, technically any I/O operations are impure functions by this definition. For example, `System.out.println` is arguably impure because it does not perform any computation (and thus does not return a value) – its sole purpose is to produce a side-effect; albeit one that does not affect the program’s internal state.

The relevance of pure functions for concurrent programming is predictability. If we can guarantee that, no matter when, in what context or how many times we invoke a function, if the inputs are the same, the outputs will also be the same without affecting any other parts of the program. This makes functions self-contained, re-usable components for building applications. Furthermore, it allows for function composition (the idea of combining functions to make new functions) because pure functions are, by definition, *referentially transparent* [177]. If all functions in a concurrent program are pure, then there is no need for synchronised access to function calls: a thread can use a function without affecting global state or the operation of other threads. Therefore, pure functions help to address the problem of non-deterministic behaviour in concurrent applications.

Another cornerstone of the functional style of programming is immutability; - especially of data structures. This is also achievable in an object-oriented language such as Java by marking variables as `final`. However, this only achieves true immutability for primitive types, because the semantics of `final` only guarantees that the reference does not change; not that the object being referenced is itself unchanged. In theory, all object structures can be flattened to a set of primitives, so if all objects made their primitives `final`, then complete immutability could be achieved. Immutability is so crucial in reasoning about code – especially in the context of concurrency – that the Java architects admitted they would have made variables immutable by default if they could go back in time [178]. In more modern languages such as F#, immutability is the default, with a “mutable” keyword instead [179]. Nevertheless, it is still possible to practice immutability even when the language was not initially designed for it. For instance, in Java SE 8, an immutable date and time API was added [180]. With this API, any changes/modifications to objects always return a new instance rather than mutating the existing instance. Thus, although immutability is a fundamental property of functional programming, it is not exclusive to functional languages. A simple yet idiomatic way to practice immutability in the object-oriented paradigm is to avoid using “setters”. In fact, since Java 8, there have been several functional libraries and data structures for Java [181] [182], such as JavaSlang [183]. These libraries provide features which can simulate functionality available in more modern functional languages (such as Scala) in Java.

From a pragmatic perspective, one of the main disadvantages of immutability is that it is often wasteful, resulting in unnecessary copying of data, which is particularly expensive on modern hardware as memory is the main bottleneck. There is often a trade-off between abstraction and performance, especially when such abstractions do not map well to the hardware paradigm. In computer architectures, memory is mutable by design; it is more efficient to mutate an existing data structure than to allocate memory for a new one, copy the elements over and free the memory occupied by the old data. Nevertheless, abstractions provided by functional programming can be useful if practiced in the right places. As Subramaniam says, it's not about completely avoiding immutability or low-level constructs like "goto" [184]; these are inevitable consequences of programming for current computer architecture. Rather, one should try to exercise a "circle of immutability", relegating mutable state to lower-level and/or performance-critical sections of the code.

We examined the concept of lazy evaluation in section 2.2.7, and saw how using functions can provide lazy semantics. However, we overlooked a critical assumption: laziness requires purity (that is, no side effects); hence why hybrid languages such as Scala require explicit declaration of laziness, whereas pure functional languages like Haskell can guarantee purity and thus laziness [185]. The rationale is that if the evaluation of some function also has side-effects which may affect program execution, then delaying the evaluation to a later point than when the programmer declared it may result in incorrect behaviour. Furthermore, due to the extra layer of indirection required for laziness in mainstream, impure languages such as Java, higher-order functions in the form of lambda expressions (which are assumed to be pure in Java) provide a convenient syntax for laziness without the verbosity seen in section 2.2.7 [185] – for example:

```
IntSupplier dataProxy = () -> expensiveComputation();
```

(though admittedly without any caching). So instead of calling the "expensiveComputation()" directly, we provide a function which will call the function and return the result upon request (in this case, by calling `dataProxy.getAsInt()`). It is perhaps unsurprising then that lazy evaluations are often flaunted as a feature of the functional programming paradigm [171].

We briefly looked at the concept of *streams* in section 2.2.4, but did not examine their implementation or execution semantics. Taking the Java Streams API [146] as an example, it is evident that functional programming principles play a crucial role. Most of the intermediate operations on a stream are higher-order pure functions: they take as input a functional interface [186] (e.g. a predicate, supplier, consumer or function) and return the stream itself; allowing for a series of computations to be chained together in a relatively high-level, declarative manner. We will discuss Streams in greater detail in section 6.4.

2.2.9 **Asynchronous and Reactive Programming**

We have already encountered the concept of *asynchrony* in the Actor model and SCOOP. According to Davies, "*Code is asynchronous if it starts some long-running operation, but then doesn't wait while it's happening*." [187]. This is contrast to traditional synchronous programming; where each statement or expression is evaluated before proceeding. Intuitively, it may seem that asynchrony only applies to multi-threaded environments, however this need not be the case. A notable example is JavaScript, which is single-threaded (it provides no concurrency constructs) yet completely asynchronous [188]. This is achieved by using two structures: the *call stack* and *task queue* [188]. We encountered these concepts in SCOOP, and their semantics are similar in this context. The idea is that a JavaScript parser pushes statements/expressions onto the stack, and these are evaluated in LIFO order. Asynchronous statements/expressions are not sent to the stack, but a separate queue.

Once the stack is empty, items from the queue get pushed onto the stack in FIFO order. A common use case for asynchrony is triggering some long-running operation, such as network request, or delaying the execution of some task by a specified amount of time. With synchronous code, this would result in unnecessary waiting, making applications unresponsive and performance becomes dependent on the order of statements/expressions in the code.

When executing a piece of code asynchronously, we need a continuation mechanism. One such way is to use the concept of Futures and Promises. A *future* (also known as a *promise* [189]) is a wrapper for the result of some computation which may or may not have been completed yet [190]. The idea is that when calling a long-running operation, instead of getting back the result, the operation returns a Future; thus allowing the calling code to proceed without blocking. However, a Future does not tell us when the computation is completed and the result is available. Our options in this regard are limited: we can wait for the computation to complete and get the result, or check whether the computation is complete or has been cancelled [190] [191] [192]. This is problematic because typically, the time between when we have enough information to trigger a long-running computation (e.g. an I/O operation over a network) and the time at which we need the result is often very short. Thus, we may begin the computation and a few lines of code later request the result. Since waiting for the result is a blocking (synchronous) operation, the benefits of asynchrony are minimised when using Futures in their most basic form.

An alternative is to use a *callback*. A callback is a higher-order function which specifies the action to take upon completion of an asynchronous computation. Unlike with Futures, we no longer need to wait until the computation is complete before proceeding – instead, we specify what to do with the returned result in a function, and when the computation has completed, the function will be called with the result. A callback is generally used to execute some code after event has occurred. An "event" could be anything from an elapsed amount of time (a "timeout") or a change in the value of some variable(s) in the program. Therefore, callbacks can be integrated into Futures – for example, the Promises/A+ standard introduces the notion of "Thenable" [193]. The idea is that all Promises have a *then* function which accepts a callback. This allows us to write truly non-blocking code in an expressive manner. For example, suppose we want to automatically click a button on a web page, but the time taken for the button to appear on the page is non-deterministic (thus, we can't simply "sleep" for a fixed period). This could be handled using JavaScript Promises as shown in

```
waitForElement(By.xpath('//button'), 5000).then(btn => btn.click());
```

where `waitForElement` is:

```
function waitForElement(elementLocator, timeout, client = browser) {
    return client.wait(until.elementLocated(elementLocator)), timeout);
}
```

In this example, the `waitForElement` function does not block the calling code – it returns a promise which, when resolved, contains the element. It also allows for timeout to be set so that if the specified element cannot be found within the allotted time, the computation (in this case, searching for the element) is cancelled (i.e. the promise is not resolved). Additionally, an optional error-handling function may be supplied to a *then* call [193]. This allows the programmer to specify the action to take if the promise cannot be resolved – essentially equivalent to an exception-handling procedure.

A well-known shortcoming of callbacks is that of “callback hell”. This is a where there are many layers of nested callbacks. For example, a Promise may resolve to another Promise, which in turn returns another Promise. From a purely technical stance, this is not an issue per say – it is a different style of programming known as Continuation Passing Style (CPS). CPS is primarily a functional programming style where control flow is expressed through a series of nested functions (as opposed to statements which appear one after another), and so functions do not return to their caller [194]. The problem with this style is that when there are lots of dependencies (and hence, nested callbacks), the code becomes difficult to read and maintain due to there being many layers of indentation. Ideally, we would like to write code in a sequential synchronous style but execute it in an asynchronous manner. One solution is to use *generators*. The idea is that functions can essentially have a per-invocation state so that they can be paused and resumed. This requires a new construct – often in the form of the “*yield*” keyword as present in Python. This concept is also known as a “*Continuation*”. For languages such as JavaScript, there are ways to make CPS-style code more flat and synchronous in nature (see [195]), however even Java will support the concept of continuations in a future release at the time of writing [196].

Ultimately, the purpose of asynchronicity is to improve the performance and responsiveness of an application by eliminating unnecessary dependencies between statements/expressions in the code. If we are to maximise the asynchrony in our code, we also need a way to manage dependencies effectively in a maintainable, readable, idiomatic and concise manner. One solution is to think of our computations in terms of *stages*, such that the transition from one stage to another is unidirectional and in chronological order. That is, each stage is “triggered” by the completion of another stage in a similar manner to callbacks. In Java, the “CompletionStage” interface provides a flexible way to express our code in terms of stages [197]. There are 38 methods which can be broadly viewed in four categories and three “dimensions” [198]. The first dimension is what triggers a stage, with three options: “then” for triggering the next stage upon completion of the previous stage, “both” for triggering the next stage upon completion of the current stage and another stage, “either” for triggering the next stage upon completion of either the previous stage or another stage. The second dimension is to do with the input and output of the stage: “apply” takes a Function (where input is the result of the previous stage and output is the result of the next stage), “accept” takes a Consumer (where input is the result of the previous stage and no output), “run” takes a Runnable (no input or output). The third dimension controls the execution policy: there is a default (no name), “async” uses the stage’s default asynchronous execution policy, “async” with an additional Executor argument will execute the computation using the specified Executor (as opposed to the default asynchronous Executor). Finally, there are more general methods: “thenCompose” wraps the result of “thenApply” into another CompletionStage, “whenComplete” / “handle” take BiConsumer/BiFunction allow for exception handling (where one argument is the result, the other a Throwable, but one of them is always null) and “exceptionally”; which takes a function used for exception handling. Combining this functionality with that of Futures results in *CompletableFuture* [199], which allows for what is perhaps one of the most powerful and expressive forms of asynchronous programming through a single API (see [191] and [200] for details).

A related concept – or rather and alternative use of asynchronocity – is the notion of Reactive Programming (also synonymous with data-flow and event-driven programming) [201]. The idea essentially boils down to triggering event listeners, often in streaming applications and user interfaces (e.g. in Model/View/Controller paradigm). The main two constructs in reactive programming are *Observer*s and *Observable*s. An "Observable" (see for example [202]) represents data or an event in an application, which "Observers" (see for example [203]) can subscribe to. Whenever there are changes on an Observable, its Observers may be notified of the change and take action appropriately. Note that the concepts of asynchronicity and reactivity are unrelated to threading: a notorious example is JavaScript which is single-threaded but asynchronous [188].

2.3 Improving Scalability and Performance in MDE

Having reviewed the preliminaries, we now turn to related works. This section explores the current literature on approaches to improving performance and scalability in model-driven engineering.

Generally, the model-driven engineering literature is primarily centred around three major model management tasks: querying, model-to-model transformations (ubiquitously referred to as simply "Model Transformation") and validation. Although these are not the only model management tasks, they are the most commonly used – particularly model-to-model transformations. It is thus unsurprising that most optimisations are applied to these tasks, with model-to-model transformations being the most active field of research with regards to optimisations.

2.3.1 Parallel Model-to-Model Transformations

Parallel execution of model transformation and validation has been successfully implemented in multiple cases. Tisi et al. (2013) [84] developed a parallel version of the popular ATL transformation engine without changes to the syntax of the language. They note that generally, parallelisation of model transformations can be regarded as two independent sub-problems: *decomposition* (i.e. how to split the computation amongst multiple threads) and *synchronization* (i.e. how to co-ordinate the threads when accessing the model in shared memory). For the decomposition, they opted for implicit task-level (MISD) parallelism where each task executes a different rule over the entirety of both source and target models. Furthermore, each task can be subdivided into two further (but dependent) problems: *matching* (i.e. finding the relevant source and target elements for a given rule) and *application* (i.e. performing the actual computation which maps the source elements onto the target model for a given rule). Since each task needs the model elements to apply the rules, this separation of concerns is unlikely to provide any performance benefits; nonetheless the authors chose this for improved flexibility in synchronisation. Due to some concurrency-friendly properties of ATL, the need for synchronisation is minimised. Namely: outputs of transformations to targets are immediate when a rule is matched (so cannot be used as intermediate data), OCL expressions (i.e. model queries) cannot navigate the target model and are free from side-effects in guards and bindings, single-valued properties are "final", multi-valued properties can only be added to, and parallelisation of ATL rules and OCL expressions are completely independent/orthogonal (i.e. can have one without the other). Because of these "nice" properties, synchronisation is mostly needed in between CRUD (Create/Read/Update/Delete) operations on target model elements (as well as properties of elements) and trace links (which store mapping information for source and target elements). Interestingly, most of the necessary synchronisation requirements came from the lack of a concurrent support on the modelling framework (EMF). In terms of performance, the authors report 1.5 – 2.5x speedup on a quad-core CPU. Interestingly however, the parallel speedup was inversely proportional to the model size. This is perhaps because they took a task-parallel approach as opposed to a data-parallel one. Thus, in *NQ* terms with a fixed *Q* (and parallelising *Q* instead of *N*), each thread has to do more work when data size increases; leading to diminishing speedup. These results suggest that in terms of scalability, a data-parallel approach may be more optimal – even the authors acknowledge that inter-thread communication is reduced using a data-parallel approach (where the entire transformation is applied to parts of the model).

A series of publications by Burgueño et al. [204] [205] [206] uses the Linda co-ordination language for concurrent, parallel and distributed model transformations ("LinTra"). The initial architecture is shown in Figure 2.3.1. The idea is that model elements are written to and read from a shared tuple space by multiple threads (possibly on different machines) during the transformation. This is transparent to the user; who writes the transformation in a high-level language like ATL whilst internally, the transformation occurs in a lower-level language which deals with multi-threading and distribution based on Linda. That is, model elements could be streamed (from the tuple space), transformed and written back to the tuple space concurrently, in a potentially distributed manner. Of course, an efficient distribution of elements and computations amongst threads would need to suffice. This solution also requires additional meta-modelling and high-order transformations to be written in a lower-level language to translate transformations written in ATL to Java, for example, which is an additional overhead. The solution uses XAP Elastic Caching Edition – a Linda engine – which allows for multiple distributed tuple spaces holding serializable Java objects, internally deals with concurrent access to the tuple space and provides an SQL-like syntax for queries. For resolving dependencies, each element is given a unique ID and when there is a dependency of element *A* on *B*, the ID of *B* is stored in *A*. This allows LinTra to not explicitly store tracing information.

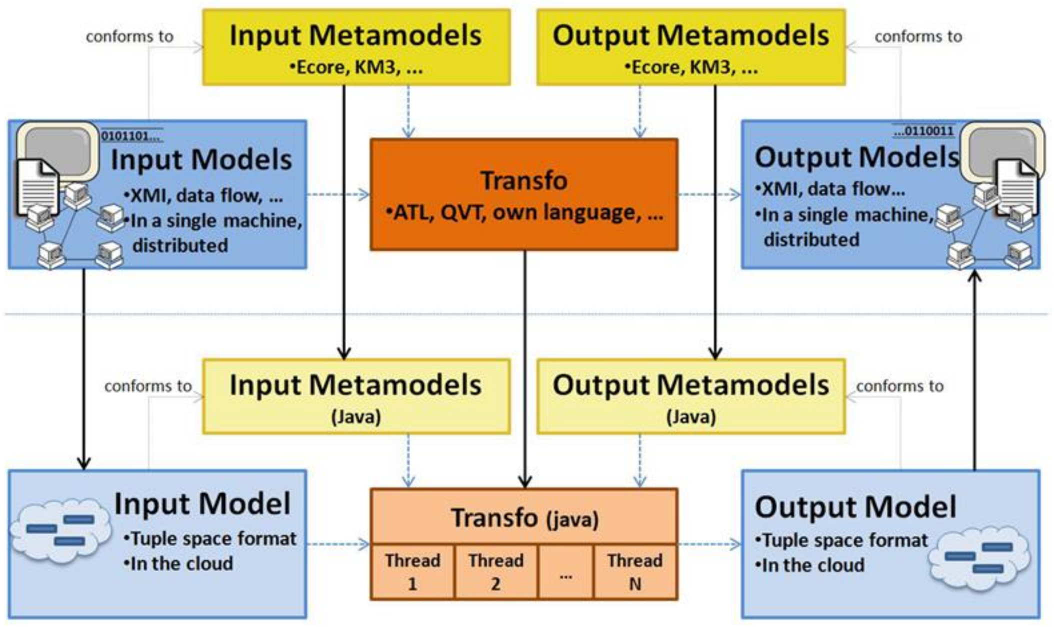


Figure 2.3.1 Proposed architecture of Linda-based transformations (Burgueño, 2013) [205]

LinTra abstracts concurrency and distribution concerns from the user – in a sense, providing automatic parallelisation. It uses the Master-Slave pattern; where the master thread co-ordinates the transformation and spawns slaves, whilst the slaves run transformation rules on parts of the source model independently. Thus, LinTra takes a data-parallel approach, which, as we shall see later, is a more efficient and scalable approach than rule parallelism. Since LinTra uses a non-recursive in-place transformation strategy (where the source model is read-only and target model is write-only and the target is derived directly from the source), the authors identify and address some potential issues which may arise from this decision. For example, handling of relationship after Create/Update/Delete events of elements and rule conflicts – or *confluence* (i.e. when multiple rules alter the same part of model – so order of execution matters). Their general resolution strategy is to

maximise flexibility, so in most cases the authors permit all possible behaviours. With regards to performance, LinTra achieved an average speedup of 2.57x compared to standard ATL (on the classic "Class2Relational" transformation), though the speedup increased with model sizes. For instance, with 4 million elements, in-place LinTra was over 955x faster than ATL. Furthermore, the authors noted that in-place transformations in LinTra are 1.81x faster than out-place. Interestingly however, the speedup of LinTra seems to scale fairly poorly with the number of cores: compared to a single core, four cores achieved 1.19x speedup, eight cores achieved 1.62 and sixteen achieved 3.24. This suggests that either further optimisations can be made and/or there are strong inherent dependencies in the transformation and/or between model elements. Nevertheless, it is clear that more cores and threads results in lower execution time.

2.3.2 Parallel Model Validation

Vajk et al. (2011) [207] designed a compiler which can generate parallel OCL ("POCL") in C#. From the observation that OCL is functional (side-effect free), they note that it is inherently parallelisable and go to the extent of formally proving this by providing equivalence mappings for constraints from sequential OCL to POCL using the well-established Communicating Sequential Processes (CSP) calculus as the intermediate language. Their solution produces imperative (both sequential and parallel) output code, with a simple compiler directive – prepending the "#" character to a keyword, operator or expression – to indicate parallel execution is desired. The authors note that although automatic parallelisation is possible, models are not static compile-time artifacts and therefore it is up to the programmer to exploit knowledge of the model structure to indicate where it is optimal to perform operations in parallel. This is perhaps a wise choice, since models can vary vastly in their structure and it may not be optimal to parallelise trivial operations. Furthermore, the burden on the programmer is minimised since they only need to type a single character next to the code they want to parallelise, without changing the semantics or compromising its expressiveness. However despite the lack of any non-parallelisable code in their performance evaluation, the speedups achieved were 1.75 and 2.8 for two and four threads respectively.

The work of Smith (2015) [10] on parallelising the Epsilon Validation Language (EVL) serve as a practical starting point for this project. Smith's work is comprehensively documented (with source code availability), and highlights many of the challenges encountered during the development process as well as the rationale behind every design decision. In an iterative development approach, the final solution consists of a rule-parallel approach with a fixed number of threads; as it was found that validation was compute-bound rather than I/O-bound due to model caching, though the author remained undecided on the optimal number of threads. Since EVL is context-based, its execution algorithm involves three main checking stages: context applies to element, constraint applies to element, constraint is satisfied. The execution of each stage is dependent upon the previous stage being fulfilled. The author's solution is to assign each constraint and model element combination to a work unit, and each idle thread processes work units from a queue. As well as modifying many of the shared data structures (such as the model cache) to be thread-safe, he also made several optimisations to minimise synchronisation (even at the expense of duplicating work) for better scalability. As expected with a parallel approach, many intermediate data structures were modified to be on a per-thread basis; the idea being that the processing happens in parallel and the intermediate results and caches are merged back into the original non-concurrent collections where single-threaded processing can resume. This is a particularly elegant approach since it limits the scope of changes which need to be made; thus resulting in less complexity and only introducing concurrency where it is required and/or beneficial. Furthermore, changes were made to the standard execution model by reducing unnecessary re-computations and also employing static analysis to efficiently prioritise the execution of constraints which have no dependencies but are depended upon by other constraints (in order to reduce idling). Overall, the results seem promising: Smith reported almost linear scaling with up to 8 threads, which exceeds some previous work in the literature. Nevertheless, despite strong initial results, there are further refinements to made. An outstanding bug remains in detecting and reporting dependency cycles / deadlock when there are more threads than model elements, and instruction-level parallelism could be introduced (especially when performing operations on collections); though this would be at the EOL, rather than EVL level (hence benefiting all Epsilon languages). The possibility of partially loading models as and when needed (see sections 2.3.4 and 2.3.5) and option for distrusted (as well as parallel) processing (e.g. by using Apache Spark) could potentially be explored.

2.3.3 Distributed approaches

Perhaps a more scalable way to improve performance of MDE programs is to use a distributed approach; taking advantage of increasingly popular and cost-effective cloud computing solutions. Clasen et al. (2012) [208] [209] propose and explore some of the open research questions and challenges in distributed, scalable model transformations. They note two important aspects in supporting very large models (VLMs) in the cloud: storage and processing. On the storage front, there is the problem of designing an efficient distributed persistence backend where access to model elements is decoupled from the underlying distribution, location and storage technology. On the processing front, the issues identified include an efficient partitioning algorithm to minimise some cost (e.g. network bandwidth, total data transfer, processing time etc.) between compute nodes, as well as handling of dependencies. There is also the type and granularity of parallelism – a distribution algorithm needs to assign the relevant model elements as well as the rules to apply to them. In an ideal, efficient solution, each node should be able to load only the required subset of the model rather than the entire model. Automating such decisions on a per-model basis will require some form of static analysis. Since the time of the authors' investigation into supporting VLMs in the Cloud, a number of relevant works have emerged (many by the authors themselves and/or affiliates) which aim to address the challenges outlined in this paper. We shall investigate some of these in the remainder of this section.

Mezei et al. (2009) [210] propose distributed model transformations based on parallelisation of the pattern matching process. They use a 3-layer approach, with the master co-ordinating 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.

Szárnyas et al. (2014) [211] present IncQuery-D, a scalable, distributed incremental pattern matching approach using the RETE algorithm. Perhaps the main novelty in this work is that it combines distribution and incrementality; which at first seems unintuitive, given that incrementality requires state. The authors use a distributed indexer, so the cache is distributed. Their solution is extensible in that storage and indexing are decoupled, allowing for different persistence backends to be used. There is also a model accesses adapter which serves multiple purposes: it provides a mechanism for uniquely identifying model elements in the entire distributed repositories, a graph-like API to the user which translates user operations to the back-end query language and a façade for propagating change notifications (in the models) to the underlying storage. Their evaluation compared a prototype implementation to state-of-the-art non-incremental distributed query engine. Their main findings were that the overhead of constructing Rete network makes it less efficient than non-incremental engine for smaller models, with the cost outweighing benefits for medium-size models. However their implementation was able to perform near-instantaneous query evaluation (after caching) even for models with well over 10 million elements.

Benelallam et al. (2015) [212] proposed an automated distribution of model transformations written in ATL using the MapReduce framework. In contrast to Parallel ATL [84], Distributed ATL ("ATL-MR") takes a data-parallel (SIMD) approach as this reduces synchronization overhead – an important factor to consider in a distributed environment where communication costs are higher than machine-local parallelism [84]. It turns out that, due to some nice properties of ATL, the semantics of model transformation and MapReduce align conveniently. The properties include data locality (only the transformation rule that created an element is allowed to modify it), immutability (properties of target model can only be assigned once), no intermediate data (as a result of non-recursive rule application) and finally the generated parts of the target model are unreadable (cannot be navigated) by rules; so the order of rule applications is immaterial. In short, rules are not as entangled in ATL, so they are more amenable to parallelisation. Each *Map* node applies the full transformation on a subset of model elements ("Local match-apply" phase). Upon completion, each Map worker sends the set of model elements it created (along with tracing information, which is used to resolve bindings to target elements) to the *Reduce* function. The "Global resolve" phase brings together the partial models and updates properties of unresolved bindings. At the beginning of this phase, all target elements are created and local bindings are resolved. Sometimes source and target elements may not be transformed in the same node during the mapping phase, so trace links are used to defer this to the reduction ("Global resolve") phase. Hence, the trace metamodel was also extended to include additional properties required for resolving bindings in the reduction phase. In terms of tool support, the Distributed ATL engine is built on top of the regular ATL virtual machine (VM) and Apache Hadoop. Each node runs its own VM but handles either the Map or Reduce phase. They set the optimal number of "splits" (model elements per worker) to the number of elements divided by number of worker nodes – ideally a one-to-one mapping – to balance the overhead of co-ordinating many splits whilst maximising available parallelism. It should be noted however that the model persistence and serialization format – XMI – is not thread-safe and must be fully loaded into memory. In terms of evaluation, they observed that at least two nodes are required to achieve the same execution time as sequential (non-distributed) ATL. Using eight nodes, they observed an average speedup of 2.5 – 3x (up to 6x maximum) across a range of model sizes. Perhaps unsurprisingly, unlike Parallel ATL, the speedup improves with model size since the authors took a data-parallel approach. There are some further optimisations to be made to further improve performance, such as parallelisation of the "Global resolve" phase to reduce I/O bottlenecks. More fundamentally however, the authors did not consider intelligent assignment of model elements to nodes, which could increase data locality, as this requires static analysis.

Recognising the need for efficient model partitioning, Benelallam et al. (2016) [213] [214] devised a solution for calculating a more optimal distribution of partial models to processing nodes. MapReduce was originally designed for relatively flat data structures with no dependencies. However, models have a graph-like underlying structure, with potentially arbitrarily complex relationships between model elements. Because of these dependencies, optimally distributing model elements across multiple nodes is non-trivial. On the one hand, we would like to maximise data locality to reduce (expensive) data access over the network, and on the other, we want to maximise parallelism by ensuring that all the available machines are being utilised to their full potential (so that they each have roughly equal number of elements to process). Most of the computational complexity of model transformations comes from pattern matching and exploring the graph structure, so data access is of critical importance for improving performance of data-parallel distributed transformations; as inefficient distribution of data can lead to severe (I/O or network-bound) bottlenecks. For declarative languages like ATL, it is possible to compute dependencies using static analysis. However, the cost of computing efficient distribution (by constructing a full dependency graph and solving linear programming optimisation problem) for models with millions of elements is itself a big task, and can outweigh the benefits (so a naïve random distribution may be better overall in terms of performance due to this overhead). Instead, “transformation footprints” are used to approximate dependencies. These footprints represent the navigation behaviour of transformation rules; which are constructed from OCL guards and bindings by recursively traversing the abstract syntax tree. The model is broken down into “splits” equal to the number of machines. Each machine has a set of elements assigned to it, where each model element (per split) is assigned to one (and only one) set. The goal of an efficient distribution algorithm is to minimize elements per machine and maximize dependency overlap in each machine’s load. The execution semantics of the authors’ proposed solution (as shown in Figure 2.3.2) does not assume the entire model is loaded into memory. Instead, model elements are streamed and assigned by a greedy algorithm to each machine; where the input to the algorithm are the footprints of the transformation and upcoming model elements in the stream. Since the order of elements in the stream can affect performance, a buffer is used to (partially) alleviate this issue. Furthermore, the partitioning algorithm distinguishes between elements which can affect the dependency graph (high-priority) and those which can’t (low-priority); though the order of arrival is still not guaranteed. Nevertheless, since the dependency graph is estimated on-the-fly, efficient partitioning depends almost entirely on quality of dependency graph approximation. To benchmark their solution, the authors compared the number of element and property accesses from the storage backend compared to random assignment. On average, their algorithm resulted in 16.7% fewer accesses, with no significant improvement for smaller models. It should also be noted that the solution assumes a thread-safe underlying framework with on-demand loading of partial models and fast lookup (caching mechanism); which are not provided by EMF. Even though this approach is far from perfect, it serves as a starting point for further improvement; especially with regards to estimation of dependencies. For instance, in certain domains, it may be possible exploit knowledge of the meta-model and the typical model topology to estimate dependencies.

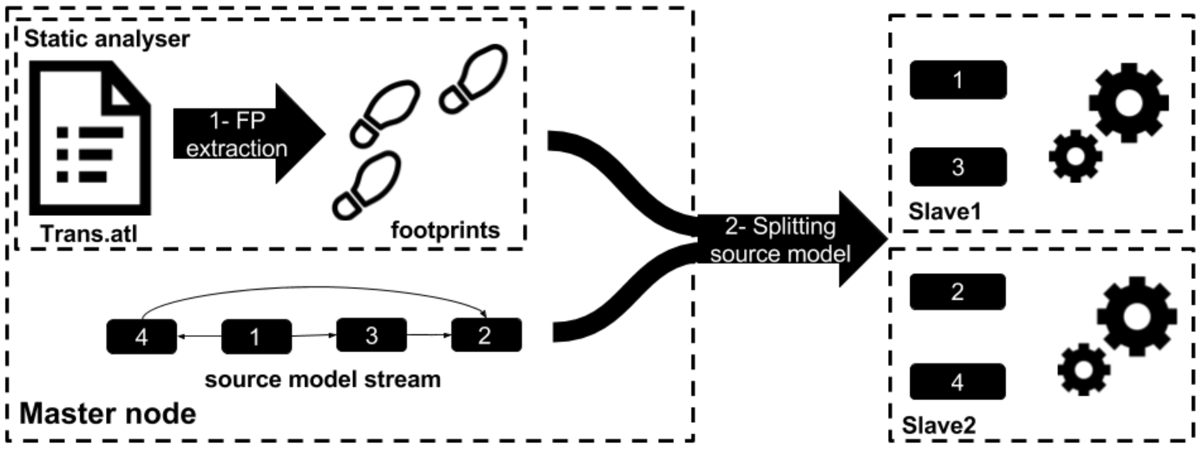


Figure 2.3.2 Execution architecture of proposed partitioning method by Benelallam [214]

On the topic of estimating dependencies, the work of Jeanneret et al. (2011) [215] may be of interest. To be clear, a "footprint" is set of model elements that are "touched" / used in a model management / querying operation. In other words, it is the minimum number of elements we need to perform the operation. Aside from efficient partitioning for data-parallel distributed model management, footprints can also be used to eliminate unnecessary loading, thus improving load times and reducing memory requirements. A *dynamic footprint* is the actual footprint obtained at runtime. Conceptually, obtaining footprints at runtime is simple, since it involves executing the code and analysing the stack trace for property accesses and operation calls (in Java for example, the runtime provides the stack trace automatically). However, the obvious flaw with this approach is that computationally, it is at least as expensive as executing the code itself; thus making the benefits redundant in many cases (especially for calculating efficient partitioning in distributed systems). Instead, we can estimate the "true" (i.e. dynamic) footprint through static analysis of the query/operation and metamodel, which gives us the *static footprint*. To guarantee correctness, the static footprint will always contain at least as many elements as the dynamic footprint. We therefore trade precision for speed. The authors' approach provides a simple and conservative estimate: they extract the metamodel elements based on the query/operation and select all their instances (in other words, for each expression, look for the types involved and select all elements of those types). For declarative languages, analysis is performed on the left-hand side of the query/operation, whilst for imperative languages, the control flow graph (all execution paths) are examined. For evaluation, the authors considered validity (which is guaranteed by construction), precision (variance from dynamic footprint) and efficiency (speedup compared to dynamic footprint). In terms of precision, the static footprints contained no irrelevant model elements and an average of 81% precision for state variables (where precision is defined as the ratio of the number of variables in the dynamic footprint to the number of variables in the static footprint). In terms of efficiency, the static footprint was several orders of magnitude faster to compute than the dynamic one; albeit with a wide variance (29x to 379x). Unlike dynamic footprints, static footprint incurs an initial, one-time cost for analysing the program. However, it should be noted that static footprints cannot be obtained for dynamically interpreted queries/operations or when using reflection.

So far we have established that in a distributed environment, a data-parallel approach is optimal. This requires splitting the input model into multiple sub-models, processing them and merging them back into a single model. Ideally, we would like to minimise (or even eliminate) the decomposition and merging steps to improve performance. This is especially relevant if model transformations are performed frequently. One way of eliminating the merging step is to use the idea of *virtual models*. According to Clasen et al. (2011), "*a virtual model is a model whose (virtual) elements are proxies to elements contained in other models*." [216]. In other words, it serves as another layer of abstraction and indirection; composing multiple models and presenting them as a single model in a transparent manner to the user. Whereas a traditional model composition (i.e. combining multiple models into one model) involves copying data from source models and synchronisation issues, virtual models don't hold any concrete data – they hold references (pointers) to the underlying source models instead. Not only do they eliminate interoperability and synchronisation issues, they also have lower memory usage and faster creation time since no data is copied. Another benefit is that contributing (concrete) model elements from base models can be composed lazily at runtime (i.e. on-demand). The granularity of virtual composition may extend to elements and even properties (from multiple base elements and properties) by using a *correspondence model* to link these elements.

Barbur et al. (2018) [217] outline the motivation, approach and preliminary results of their distributed model analytics framework using Apache Spark. Their framework is centred around document collection and indexing using a vector space model. Each dimension of the vector represents some metric of interest, such as how frequently a word appears. This approach, inspired by information retrieval and machine learning domains, is adapted to models with metamodel-driven feature extraction. They then map these to concepts in Apache Spark: the feature sets are the data and the feature comparisons are the tasks (parallelisation atoms); although a coarser-grained parallelisation approach is used in practice. They evaluated their approach by mining GitHub for Ecore metamodels, aggregating them into a single file. As expected, their approach scales with the number of cores and machines but greatly diminishes after 50-100 execution nodes (they tested with up to 500 executors). For their smaller dataset (consisting of 250 metamodels) they were able to reduce execution time by roughly 6.5x with around 25 executors. However, it should be noted that their evaluation considered the scenario where both the local and distributed had no caching. With caching enabled and a single executor, sequential execution was considerably faster than distributed without caching until the number of executors exceeded the number of cores typically available on high-end machines.

2.3.4 **Model persistence**

Besides efficient processing of very large models, another aspect of scalability in model-driven engineering is efficient storage of models [88]. Although the storage of models and execution of programs on models are orthogonal concerns, they can both benefit from distribution techniques. Fundamentally, the idea is to break down a large problem into smaller pieces and handle them using different nodes; ideally independently of each other for maximum performance. This divide and conquer approach is applicable not only to parallel computation, but also to model persistence.

The standard model persistence format is XML Metadata Interchange (XMI), where entire models (with all their elements) may be recorded in one large XML file which needs to be fully loaded into memory [88]. In order to distribute very large models in this format, an approach for splitting XMI files into multiple smaller files is needed. Garmendia et al. (2014) devised a semi-automated approach for splitting (and composing) monolithic EMF models into pieces [23]. The methodology requires the metamodel to be annotated with modularity constructs, which are inspired by Eclipse's structuring of Java source code – namely Projects, Packages and Compilation Units – where each project has multiple packages and each package has multiple compilation units (classes). A dedicated plug-in is then generated from the annotated metamodel, where users can work with large models in a more structured manner. The underlying model is stored as several XMI files, though this is transparent to the user, who just sees the structure (as defined by the annotations) and properties of the model. Although this solution provides a novel and intuitive workaround, it still uses the XMI format, which is inherently not scalable.

One of the issues with XML-based persistence of structured data is verbosity [88], since many characters are wasted on repeating boilerplate information for compliance. Alternatives XML, such as Protocol Buffers or JSON may potentially be much faster [88]. A conventional alternative to XMI is Connected Data Objects (CDO) [218]; which serves as a model repository by providing an object-relational mapping for use with (typically) relational databases. However, such centralised storage solutions may be a bottleneck for distributed systems; especially since distributed model management tasks may often require access to a subset of the model. Recognising these concerns, Gómez et al. (2015) [219] designed a transparent distributed persistence backend for EMF ("NeoEMF/HBase"). As a persistence manager which sits behind the modelling framework (EMF), its communication with the underlying database is transparent to the user; who sees a consistent API regardless of the storage technology used. Principally, the solution uses a map-based data model for persistence by flattening graph structures, as the authors claim (based on previous findings) that key-value pairs are more efficient and easily distributed, even for very large models, and also minimizes dependencies. The structure consists of one table with three column "families": *Property* for keeping object data together, *Type* for meta-level interactions (e.g. instance-of relationships) and *Containment* for describing structure in terms of containment references. By using delegate objects to track model elements' state, the solution allows for more responsive access, and by not keeping hard references, it allows for efficient garbage collection. The authors also claim transactions to be ACID (Atomic, Consistent, Isolated, Durable) at the (single) object level, making it a suitable choice of persistence for concurrent and distributed applications. Although the authors did not provide a performance evaluation, the overall architecture of their solution may serve as a good way to abstract the distributed storage aspects, allowing model management tasks to execute without explicitly accounting for the underlying persistence mechanics.

Efficient storage of models is perhaps less of a concern to application performance and users than the speed at which data can be retrieved or stored. Although in many model management programs the model is loaded in memory prior to execution, this need not (and ideally, should not) be the case as scalability is limited by memory, and poses additional overhead in parsing large models into memory structures. Other domains typically handle large amounts of data using databases; often relational (SQL) though more recently other forms such as graph, document and object databases have been gaining popularity as they are more scalable and efficient for many use cases.

Large, monolithic file-based models which are common and widely supported, e.g. XML-based formats like XMI, CSV files etc. present scalability issues, primarily in I/O and network bottlenecks when used in version control systems. This is made worse if the formats are binary files, since popular version control systems cannot usually deal with incremental changes, requiring a lot of space and for developers to manually resolve conflicts. Barmpis and Kolovos (2013) describe two solutions to these challenges: *fragmentation* and *model repositories* [220]. The former involves splitting the model into fragments and distributing only part of the model (i.e. fragments) to developers who need to work on the model, however this limits visibility of the full model, and there may be cross-references between fragments. The latter typically involves using a proprietary solution to handle model accesses and is backed by a database, with the disadvantage being the lack of extensibility and support for widely-used tooling. Barmpis and Kolovos propose an architecture which is scalable, extensible and works with established model-based tools [220]. They implement their vision in *Hawk*, a model indexer [221].

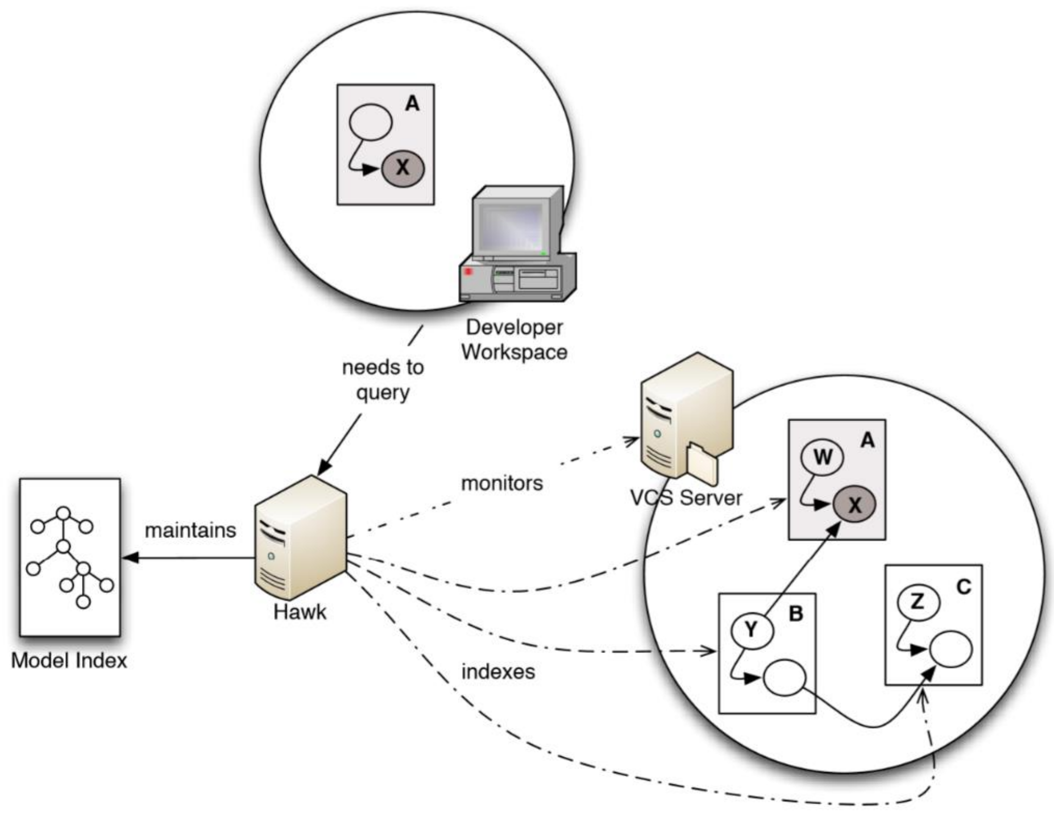


Figure 2.3.3 High-level overview of Hawk in collaborative modelling [220]

The main idea of Hawk is to provide fast uniform access to model elements on-the-fly as and when needed by clients. Instead of each developer (client) having to manually maintain its own copy of the model and synchronize its changes with the version control system, they instead use Hawk as proxy and “single source of truth” for all model management operations. One of the benefits of using Hawk is that it can be extended to work with heterogenous model sources because it adapts the contents of file-based models into an EMF resource. This resource, along with versioning information, is stored in an efficient NoSQL database, allowing for fast random access to arbitrary model elements. The Hawk instance automatically takes care of querying the version control system for changes and updating its index, as well as dealing with CRUD events on models and metamodels. Clients do not need to have a local copy of the model: they ask the Hawk server to provide them with model element instances as and when required (i.e. lazily). The way this works is that an empty EMF resource (conforming to the relevant metamodels) is provided to the client, and when the client requests a property (e.g. in a query on the model), Hawk fetches the requested element with its properties populated. However, references to other model elements are lazily resolved, to avoid potentially downloading the entire model upfront.

Besides the utility of Hawk in collaborative modelling, it can also be useful in distributed computing, since clients / workers do not need a full copy of the model. Even in non-distributed use cases, Hawk can still be valuable as it eliminates the need to fully load the contents of a model into memory and (at least to some extent) relieves clients from manually dealing with model versioning.

2.3.5 **Incremental approaches**

Incrementality is perhaps the most commonly explored solution to scalability challenges in the model-driven engineering literature. This is somewhat unsurprising given the nature of models and model management programs. Recall that there are three components in a model management program: the model, metamodel and script (program itself, i.e. transformation or query). In most cases, the model is the only variable, and typically does not undergo large changes. For example, suppose that a model represents the codebase of a software project. Every time a change is made to the code and pushed to a version control system, the model is automatically updated. For mature projects, such updates will be relatively small. If there are programs which are run on this model (for instance, validation constraints and transformations), then it doesn't make sense to re-run such programs on the entire model every time a small change is made to the code (and thus, model). Instead, the results of the programs can be cached and updated in response to the changed subset of the model. In other words, the programs are executed with a subset of the full model, containing only the changes. Of course, given the highly interconnected nature of models (which are typically graph-like structures), even a small change may have a substantial execution cost depending on the program and the sophistication of the incremental execution / caching algorithm. In many scenarios however, incrementality improves execution times by some orders of magnitude. However the benefits of incrementality are only apparent in specific workflows – typically mature projects where the programs don't change and the model undergoes small changes. In cases where the model undergoes large changes or the program source is frequently changed, incrementality provides a much smaller benefit and can even be more costly due to the bookkeeping overhead. It is also not always feasible to perform caching due to memory constraints. Incrementality also means execution is not stateless, requiring an execution trace which can become invalidated or inconsistent.

2.3.5.1 Incremental Model Validation

Cabot and Teniente (2006) [222] designed an algorithm which (they claim) ensures the most incremental (i.e. smallest / least work) expression can be provided to validate a given constraint in response to a CRUD (Create/Read/Update/Delete) event / change in the model. It automatically generates the most efficient expression for incremental validation for a given event. Their solution is conceptual, so it can be applied to any language or framework, such as EVL, though their chosen application is OCL. It is particularly applicable to EVL because OCL constraints are expressed in the context of entity types. The algorithm works as follows:

For each PSE (Potentially violating Structural Event – an event that can violate a constraint):

1) Get a set all of the possible contexts that a constraint can be expressed in, i.e. all of the types referenced in the IC (Integrity Constraint – body of a rule which validates all instances of a given type).

2) Choose the context which will produce the most efficient expression ("Best Context").

3) Redefine the IC in terms of the CT (Context Type) computed in (2).

4) Get all instances of the new CT (as computed in (3)) which are affected by this event.

5) Modify the IC to run over only the instances computed in (4).

Expressions in the body of an IC are formed as a binary tree, and each node is marked with a set of events that may violate the constraint in the IC that are produced by this node. The Best Context is then chosen from the node that produces the event.

2.3.5.2 Incremental Model-to-Model Transformation

Jouault and Tisi (2010) [223] modified ATL to support live incremental transformations. A *live* transformation applies change events directly to models, whereas an *offline* incremental transformation keeps a copy of the original models and computes the changes afresh each time a new version of the model comes in by comparing it to the older version; which is arguably less efficient. The authors' solution applies to the declarative subset of ATL (as with most other enhancements to ATL's execution) for simplicity. A notable challenge with live transformations is dependencies, since a change in an element can have effects in multiple places – including predicate filters. The authors' solution therefore tracks dependencies of OCL expressions using a map of model elements to properties for each expression. This is used to build up a trace of all the navigated attributes of model elements. More trace information is required than can be obtained from static analysis since expressions can be more complex than Boolean filters, and apply to contexts. The authors recognised this is a sub-optimal approach (as it requires evaluating expressions which may not lead to a change), though suggested adopting Cabot's efficient algorithm for Boolean OCL expressions. Aside from tracking model navigation expressions, incrementality needs to be supported at the rule-level; - that is, for a given change, only executing the relevant rules and on the individually affected elements only. The navigation trace is used to do this at the element (rather than type) level. The standard execution algorithm is modified to be event-driven (i.e. responds to create/update/delete events). Several modifications were made to the compiler – for example, adding the OCL navigation trace, event listeners and the transformation controller needs to be event-driven to call specific parts of the execution engine (not just the main method when a change occurs). The authors did not define retainment rules, so manual changes in target model overridden. They also did not provide any benchmarks for comparison to the standard execution engine. Nevertheless, the authors provide the implementation details and suggestions for future work. Of particular interest is the potential to chain transformations in parallel; where a transformation is split into a series of multiple smaller transformations.

An alternative approach to incremental model transformations is the notion of *partial evaluation*, as proposed by Razavi and Kontogiannis (2012) [224]. The idea of partial evaluation is to pre-compute some values by running the program on a subset of input data (which is statically determinable at compile-time) beforehand. That is, the "static" data (for example, a variable which is assigned the value of the sum of two integers which are known at compile-time) can be computed and replaced with the result; leaving less work for the "residual" (i.e. dynamic) computations (which are performed at runtime in the usual manner). The authors argue that for iterative model transformations, elements which change are "dynamic" and those which do not change are "static" (invariants which can be pre-computed). They developed "QvtMix" – a prototype for QVT

Operational Mappings (QVT-OM) – a hybrid transformation language from OMG. The Partial evaluator itself written in QVT-OM, and focuses on pre-computing collections and expressions. Architecturally, the design consists of an OCL Parser and Evaluator (which evaluates static expressions as they're found and caches them), a Binding Time Analyzer (which annotates expressions and variables as "static" or "dynamic", though the user can also provide annotations to guide the analysis), a Specializer ("Visitor" that traverses the AST, applies production rules to expressions and outputs the partially evaluated part of the program) and a Pretty Printer (produces the source code to be executed by QVT-OM from the partially evaluated program). The static parts are either cached in a table or inlined in the code. In their evaluation, the authors found performance benefits became apparent when the number of fixed elements is greater than 100. Their results verified that the benefits of avoiding re-computations outweigh the overhead of parsing and interpreting residual expressions and static caches. Consequently, significant performance gains (e.g. 2x) were made even for relatively modest model sizes of O(10^4) elements.

2.3.5.3 Incremental Model-to-Text Transformation

Ogunyomi (2016) [225] uses runtime analysis to identify impact of source model changes on textual artifacts. In contrast to the *model differencing* approaches (where the original and modified model are compared to obtain the changes) used in incremental model-to-model transformations, the author's solution comprises of *signatures* and *property access traces*. A signature is a string proxy computed from the output of a template execution, such that if two signatures are equal, then re-executing the template will result in identical output. Thus, signatures are used to detect changes at runtime and re-execute templates only if changes in the model also change the signature. Unlike model differencing, only one version of the model is required and a single traversal, making it a more efficient solution. A property access trace stores model navigation information in persistent memory as templates are executed. This allows the engine to determine which model elements (and their properties) affect the output of a template, so that a template is only re-executed in response to a change in the model only if the template uses the modified properties. This lead to an average performance improvement of 60% (in terms of execution time) compared to standard model-to-text transformation. An interesting challenge in the context of model-to-text transformation (especially for artifacts such as generated code) is consideration for retainment rules so that manual changes to the output artifacts are not overwritten by a template (re-)execution. The granularity (i.e. whether at file-level, element-level or attribute-level) of incrementality also impacts the effectiveness of the solution. Too high-level and it becomes overly pessimistic / less responsive to minor changes, resulting in more unnecessary re-computations. Too low-level and it becomes more complex to handle, with more memory required for trace links (keeping track of more things). Overall, the empirical results seem promising: execution times were reduced by up to 99% whilst keeping memory consumption at a fraction of the input model.

2.3.5.4 Bidirectional Model Queries

Jouault and Beaudoux (2015) [226] present an approach for achieving bi-directional incrementality by applying Active Operations to OCL. Suppose a program computes a query or transformation which involves many intermediate operations on a collection of data. Such operations could be queries and transformations like filtering, mapping, validating a predicate on elements of the collection, sorting and transforming the collection into different forms. The idea of active operations is to efficiently propagate changes in the data (model) such that only the necessary recomputations are triggered, as opposed to recomputing the query or transformation in its entirety. In other words, the goal is to perform the smallest amount of work in response to changes in the model without having to evaluate operations in full, even for the affected subsets of the model. This can be achieved by using “boxes” to wrap values, which allows for mutability and changing multiplicity (to represent additional or fewer results in response to changes). One of the main challenges with this approach (and perhaps bidirectionality in general) is reverse propagation. This problem is similar to how one-way hashing functions work (as used in cryptography, for example). When the output of a function is simpler or contains less data than its inputs, it often means there are multiple inputs which can result in the same output (as explained by the pigeon hole principle). Although the operations in OCL are pure functions, a chain of operations on a collection of elements which results in outputting of a single element requires more information for reverse propagation – so if the output (result) changes, it is not always possible to determine how the change affects the data in the source and intermediate operations. Another limitation is supporting nested operations which flatten data, such as the *closure* operation.

2.3.6 **Lazy approaches**

Laziness is especially desirable in model management programs because there is often a disparity between the user’s intentions and their expression in the given tools. Model query languages such as OCL provide convenient syntax and operations from the user’s perspective, however their general-purpose nature (at least, how they are typically implemented by tool developers) means there is often wasteful computations. Understanding which computations are redundant typically requires analysing the context of a given computation; i.e. its subsequent uses. If the result of evaluating a pure function is unused, the computation can be avoided with no discernible difference. This is the key to lazy evaluation. However laziness is not trivial to apply without static analysis in many cases. Instead, the main focus is on computations which have a significant cost, or are performed frequently. In model management programs, model accesses are relatively expensive and so much of the literature is focused on reducing reads and writes to the model. Although this emphasis is often not explicit, the nature of such programs means that eliminating redundant computations in intermediate queries and transformations leads to a reduction in overall model accesses and improvements in both execution time and memory usage.

2.3.6.1 Lazy Model-to-Model Transformation

Tisi et al. (2011) [227] designed and implemented lazy evaluation semantics for ATL. They distinguish between two orthogonal aspects of laziness for model-to-model transformations: *generation* and *navigation*. Since model-to-model transformations consume the source model and produce the target model, both the production (generation) and consumption (navigation) can be lazily evaluated. Lazy generation computes target model elements when they are requested. This works by intercepting navigation calls on the target model (from the modelling framework) and firing a generation event (e.g. call the corresponding "generateTarget" when a "get" method is called on the element in the target model). Of course, this requires the transformation engine to allow for computation of a single element. Trace links may also be used to avoid re-computations as with incremental approach. This would mean transformations are stateful, as "history" is kept in memory, which can be used to help with dependency analysis when performing computations. Lazy navigation on the other hand, is perhaps simpler to implement because it does not depend on events external to the transformation engine. Lazy navigation is concerned with evaluating queries on the source model (e.g. with OCL), so that queries on the source model are only performed when the result is used in transformation/validation (e.g. in guards) logic. In terms of performance, the authors observed a linear relationship between execution time and number of navigated target elements, whereas the eager evaluation had a constant execution time. Experiments showed that speedup is linearly proportional with model size. The book-keeping overhead of lazy evaluation can reduce performance if most of the target model is navigated, with the eager evaluation out-performing the lazy one after 48-58% of the target model is navigated. The authors note that further optimisations can be made, such as caching intermediate expression values used in target generation, and adding a lazy OCL evaluator for lazy source navigation.

2.3.6.2 Lazy Model Queries

Tisi et al. (2015) [228] proposed lazy evaluation semantics for OCL – a side-effect free functional language. The main contributions of the authors' work are lazy production and consumption of collection elements without breaking compatibility. The authors' approach for lazy collections is iterator-based: iteration operations (e.g. "select") return a reference to the collection and iterator body. The iterator body produces elements as and when required by the parent expression. They minimised modifications to syntax/semantics of strict OCL specification where possible – so no additional data types / lazy collections or keywords. The only deviation from the strict OCL specification because of lazy semantics is that it's not possible to invalidate a collection if an invalid value is added, since it's not known as with standard OCL (infinitely sized collections are possible). The OCL specification states that all collections must be finite, though with lazy evaluation, as long as only a finite part of the collection is used then queries will terminate. However, some operations may not terminate (e.g. finding the average, sum or maximum of an infinite collection). A commonly used operation on model elements is "allInstances()", which returns a collection containing all of the instances of the specified model element. This is often called and then followed by some filtering operations. Clearly, the programmers' intent is not to obtain all of the elements, yet the entire collection is computed and returned under eager evaluation semantics. The authors also modified "allInstances()" with lazy semantics, and a fair-traversal algorithm (a hybrid between breadth-first and depth-first) so that computations can terminate; since an infinite depth or breadth can prevent a depth-first and breadth-first search from ever returning a result. This “diagonal” approach relieves the user from selecting the correct search strategy. Unsurprisingly, lazy evaluation results in fewer calls and faster execution when an element which satisfies the query on a collection is near the beginning of the iteration compared to eager evaluation. However, it should be noted that laziness in this context is ideal when a small part of a large collection is required, as the iteration overhead can actually be worse than eager evaluation in some cases.

Willink (2017) [229] proposes a novel way to implement OCL collections which overcomes the limitations associated with an intuitive, one-to-one implementation of OCL collections using Java types. He shows that although OCL collections require inefficient properties such as immutability, eager evaluation and lack of short-circuiting, it is possible to get around these by using a custom data structure consisting of a HashMap and ArrayList to represent all four collection types with improvement in execution time an memory consumption in some cases. He also notes that other optimisations, such as element selection, can be performed in constant time thanks to the use of a HashMap as opposed to linear time which is the norm for the usual traversal algorithm.

2.3.7 **Model access optimisation**

So far we have seen how there are two distinct but important aspects to model management programs: the execution strategies used in queries and transformations (as defined by the user in a textual language) and the modelling technology; more specifically its persistence format. As we saw with the example of Epsilon, it is possible and even arguably desirable in many cases to have a conceptual and technical separation between the execution engine’s handling of queries / user-defined programs, and the underlying data (models) on which these programs are executed on. Such a distinction enables greater flexibility and re-use benefiting both users and tool developers. However when optimising for performance, it is important to be able to exploit knowledge of certain implementation details, which is usually difficult to access when there are abstraction layers. In the case of modelling tools, it is usually assumed that model elements are loaded into memory and accessing them has a relatively low overhead. As we have seen, for very large models this is impractical, sometimes even impossible, so researchers and tool developers are increasingly turning to databases rather than the traditional approach of parsing large files into memory. Although databases help to alleviate memory bottlenecks, a naïve model querying execution engine may still require lots of low-level accesses to specific model elements, or large portions of them. Since most databases have built-in facilities for basic querying, it would be more optimal to perform such queries using the native database facilities where possible.

The work of Kolovos et al (2013) [230] on supporting SQL databases as a modelling technology in Epsilon demonstrates the importance of optimisations not just in the implementation of operations in isolation, but also in how data is retrieved from the source. The JDBC driver for Epsilon [231] transforms queries on collections of model element types (including *allInstances()*) into SQL queries which are lazily evaluated. Each operation internally builds an SQL query and returns a lazy collection, allowing for further operations to be composed before executing the query. The returned results can be materialised by invoking a specific method.

Daniel (2018) [232] presents the Mogwaï tool and an approach for querying models without loading them into memory in an efficient manner by taking advantage of modern graph databases. The tool can convert OCL and ATL queries into database queries, using Gremlin as an intermediary language. The ability to perform efficient queries without spending time parsing and loading the model into memory is very much complementary to other performance-enhancing solutions with regards to scalability. It would also be interesting to investigate the extent to which these optimisations can be applied to such programs, especially since the queries are executed externally (i.e. on the database), allowing for optimisations to be made at a much lower level. By removing the burden of query execution from the execution engine and onto the actual persistence technology, fewer unnecessary computations occur.

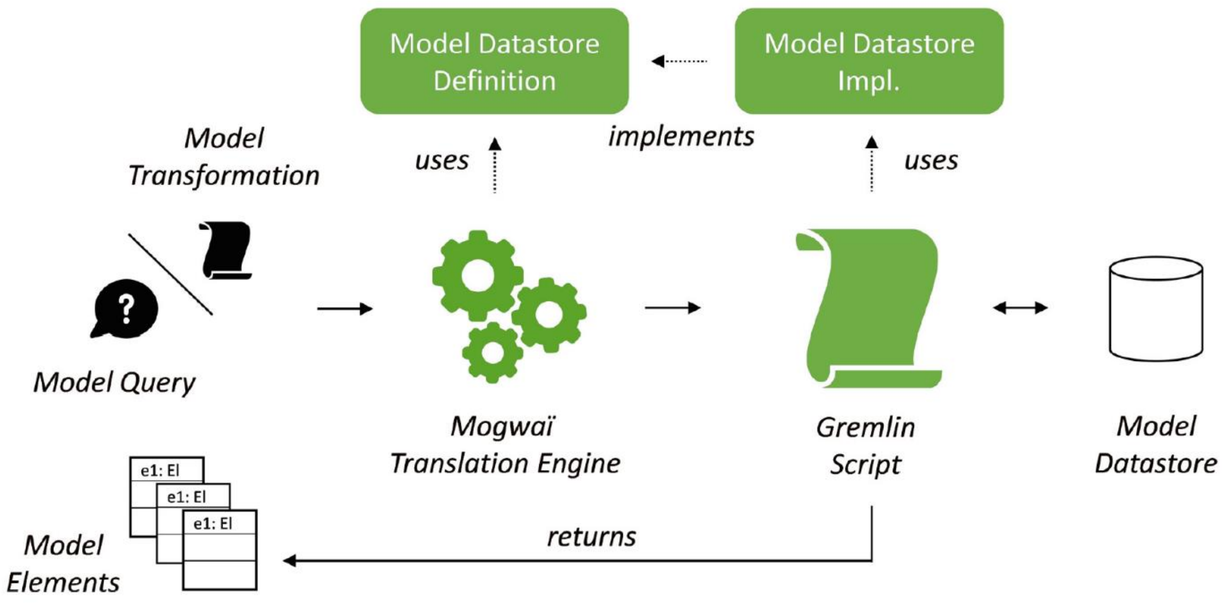


Figure 2.3.4 High-level overview of Mogwaï tool [232]

2.3.8 Reactive approaches

From a practical aspect, it would be useful and productive if the execution of model management programs could be scheduled in a more automated and intelligent manner. This can be partially achieved by combining incremental and lazy execution semantics, which require the program and potentially modelling framework to support change notifications. The notion of *reactivity* implies computations which are triggered based on changes and, ideally, only the minimal computations required based on the changes as and when required. Bergmann et al. (2012) [233] define four aspects of change with regards to models:

- Controllability – only an explicitly defined set of changes is permitted at each state of the model.

- Observability – the ability to view changes after running a model management program
- Source of information – Changes consists of pre-state, post-state and delta. We can only observe the change if we have at least two of these sources available.
- Delta representation – how (if at all) the changes (delta) are used by the model management environment.

Martínez et al. (2015) [234] demonstrated a fully reactive model transformation architecture by implementing a reactive engine for ATL ("Reactive-ATL"). The idea of *reactivity* is automating change propagation so that transformations are executed in response to changes in model elements. This means the host application doesn't need to manually trigger transformation rules. Transformations only occur on the affected elements, instead of executing the whole program again (incrementality). The authors argue that a reactive architecture allows for separation of concern, efficiency and a declarative style from host application's perspective. This is particularly helpful for complex propagation networks, as the computation logic is taken care of automatically for the general case.

Generally, a transformation must occur within a specific but potentially wide-ranging time window: after a change in the source model but before the result is used in the target model. The authors recognise that eager and lazy evaluations are two extremes on this window. *Incrementality* (change propagation) is an orthogonal dimension. Hence, the authors classify transformations into four categories. *One-shot* transformations – which are neither lazy nor incremental – execute the entire transformation over the entire source model every time with no change propagation (i.e. the user must manually trigger transformation in response to updates or requests, no caching (everything is recomputed), not lazy (computation is eager, not navigation-drive). This is one end of the spectrum, and arguably most model transformation tools can be classified as being "one-shot" in their execution semantics. On the other end of the scale is where both automatic change propagation and lazy evaluation are supported, and this is the case with Reactive-ATL. Indeed, Reactive-ATL builds on previous incremental and lazy evaluation techniques in ATL.

Since change propagation requires the transformation application to be notified of changes in the model (and requests in the target model), the modelling framework API needs to support notifications for CRUD events. The authors therefore implemented "Reactive-EMF", which provides a lazy, incremental and request/update-driven approach for the Eclipse Modelling Framework. The authors chose to add reactive functionality to EMF to preserve compatibility with existing models and tools. Their approach was to re-implement the interfaces with reactive behaviour, with the Observer pattern being the main mechanism. Amongst the highlights are support for element and property-level notifications and uniform treatment of source and target models; allowing for (reactive) transformation chaining.

An interesting feature of Reactive-ATL is that it is both lazy and incremental on the target model as well as the source model. Pre-computed elements in the target model are effectively used as cache; and these elements are "invalidated" when changes occur in the source model and only propagated if they are consumed. As with any reactive architecture, production is demand-driven, so the target model is only updated when it is navigated by the user. On the source side, OCL expressions are tracked to avoid recomputation when the source model changes by building a map of expressions to properties. In terms of efficiency, the reactive architecture tries to execute the minimum required computation in response to an update or request. To is achieve this element-level granularity (i.e.

the ability to respond – read and write – to changes at the element-level) is supported. In responding to CRUD events, the authors propose an "Invalidate/Lazy revalidate" strategy; where changes (Create/Update/Delete) invalidate the affected elements and requests (Reads) trigger the revalidation. Going even further, their solution also responds to changes in the transformation logic in a similar fashion. Each time the transformation code is changed, the reactive engine is notified and performs three steps upon notification of a change: Analysis of what's changed by using EMF Compare on a meta-model of the ATL transformation, transformation adaptation by compiling the new transformation and replacing the operations map and finally propagating the change to ensure target model consistency with the new specification.

In terms of performance, the Reactive-ATL performs at least as well (total computation time) as one-shot, lazy and incremental versions of ATL in most cases; only being slower when navigating a big part of a newly generated model with no updates. The benefit of a reactive architecture in this regard is that the user only "pays" for what they use due to laziness and incrementality; rather than a "flat fee" as with typically one-shot model transformation programs. The authors note that virtually all combinations of lazy, eager, incremental, non-incremental and one-shot transformations can also be handled by Reactive-ATL if needed; so reactivity does not preclude less advanced transformation semantics. That said, despite making no modifications to ATL's syntax, some imperative constructs and advanced features are not supported; though judging by previous work on ATL this is not surprising. A more fundamental shortcoming is that change propagation is unidirectional – that is, only source edits are supported.

Motivated by high-frequency model changes, Bergmann et al. (2015) [91] proposed an architecture and framework for streaming model transformations by *complex event processing* (CEP). The motivation came from the need to perform model transformations in response to continuous changes in the source model – for example, a motion-tracking sensor which builds a model of human limbs and their positions in space; updating the model 25 times per second. The authors recognised that traditional model transformation tools and techniques were inadequate for real-time data processing, and hence proposed more reactive approach better suited to such applications. Stream transformations are characterised by the on-going creation and consumption of model elements during transformation. That is, the whole model is not available at the beginning, and continuously generated. The model may change many times during the transformation process. The authors distinguish between *elementary* and *compound* model changes, and also between *atomic* and *complex* events. Elementary changes affect single attributes – the smallest possible change, and compound changes comprise multiple elementary changes that occur between two states of the model. Atomic events may be either elementary or compound changes, specified by type, the set of model elements and a time. Complex events are a sequence such atomic events. The aim of the authors' work is to design a language and framework for defining, detecting and responding to complex events in an efficient manner. The idea is that atomic events are continuously published to an event stream, and this stream is monitored for specified patterns of complex events. When a complex event is matched, a specified action is taken (e.g. the triggering of a specific transformation rule). Their contribution consists of an event processing DSL that allows for defining atomic events from change patterns and combining these to define complex events (using event algebra), a prototype CEP engine and reactive processing architecture. The tooling architecture consists of an incremental querying engine which continuously monitors the source model, an event stream which is continuously processed by a reactive rule engine (triggering rules where appropriate) and also monitored by the CEP for complex patterns (generating and publishing complex events to it when it finds them). Changes in the model are propagated through a Create/Update/Delete notifications API, and callbacks are registered to model elements that receive notifications. The results of queries are published to the event stream as atomic events. In terms of performance, despite being a single-threaded solution, the authors report a maximum theoretical throughput of around 24,000 complex events processed per second on a 2.9 GHz CPU.

2.3.9 **Static analysis**

So far we have discussed optimisations at the engine level. In almost all cases (with the exception of parallelisation and distribution in some circumstances) the semantics of a given program do not change as result of these optimisations. Laziness, incrementality and parallelism can be implemented in a way such that the end result of the program is unaffected and indistinguishable from a less optimal implementation. In other words, the user need not re-write their programs in order to benefit from these optimisations; although they may need to be aware of them depending on the complexity of the program and the way in which the optimisations are implemented.

Model management programs expressed using textual languages such as EOL or OCL allow users to specify their query or transformation logic in many different ways. As with general-purpose programming languages, programs are expressed by composing generic operations (functions) and applying them to the relevant subset of the data (model). Although this approach is flexible and relatively easy to work with from the users' perspective, it also places great power in the user's hands. With power comes responsibility, and this is especially the case for performance.

An effective way to improve performance is to be able to detect "code smells" or suboptimal code based on the user's intent. The aim is to either warn the user that their code is suboptimal, or even to replace it with an optimised version at runtime. Whilst it's not possible to completely prevent the user from writing suboptimal code, it is possible to detect the most simple and common cases. Such a task requires the language (or rather the execution engine) to be context-aware and ideally to detect suboptimal code at design-time rather than runtime. The concept of *static analysis* is usually applied to compiled languages and sometimes integrated into compilers (albeit in more primitive forms) to detect syntactical or logical errors. Static analysis involves analysing the source code of a program and flagging any errors or warnings prior to execution (hence the name "static" – no runtime information is required). Although advanced static analysis tools are common in general-purpose languages such as Java (and integrated into bespoke development environments such as IntelliJ IDEA), they are much less common in more specialised languages.

Wei and Kolovos (2014) [235] present a static analysis framework for the Epsilon Object Language, with a focus on suboptimal code detection. Their approach itself is a model-driven one: the abstract syntax tree (AST) of an Epsilon program is transformed into a model conforming to the Epsilon language metamodel. The tool is extensible, with the main components being an AST visitor (which can then be extended for Epsilon languages other than EOL), type and variable resolvers. One challenge with static analysis of EOL is that variables are dynamically typed and mutable, so the variable resolver establishes links between references and declarations. Perhaps a more fundamental aspect is developing an understanding of the types involved in the program. Built-in types / primitives are relatively straightforward since they are static (although users can define additional operations on these types to enrich their functionality, as with languages like Kotlin), however model element types vary depending on the metamodel, which is exogenous to the execution engine and even the model management program. Types are defined at the model connectivity layer (EMC), so the static analyser needs to interface with the metamodel in order to derive valid properties and operations for a given type. Due to the diverse nature of (meta)modelling technologies supported by Epsilon, the authors convert type information to Ecore for a simplified, uniform representation for convenience.

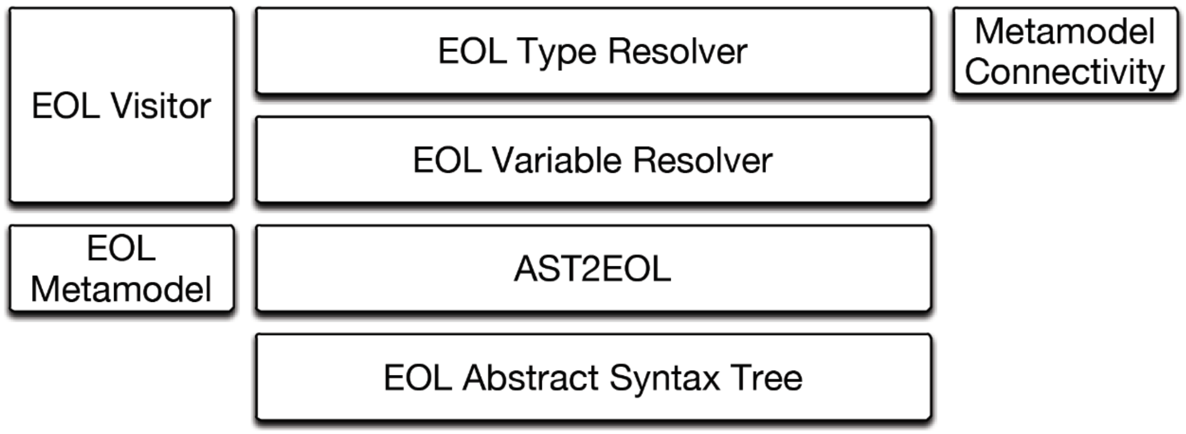


Figure 2.3.5 Wei and Kolovos' Epsilon static analysis architecture [235]

The authors build on top of this framework with a suboptimal code detector, which uses the Epsilon Pattern Language to find specific patterns of suboptimal code in EOL programs. This approach works because the framework is model-based, so all that is needed is a model of the user's EOL program (i.e. the AST) and the metamodel(s) used / navigated in the program itself (i.e. by the script). This approach is also extensible since more patterns can be defined to detect other suboptimal expressions. Most of the patterns involve the *select* operation, which filters a collection based on a given criteria (a predicate lambda expression passed to the operation as a parameter). One such example is chaining *select* operations, as shown in Listing 2.3.1.

```
Car.allInstances().select(c | c.year > 2016).select(c | c.mileage < 10000);
```

Listing 2.3.1 Example of a chained *select*

This is suboptimal because the engine must make two passes on the collection (albeit the second one being potentially smaller) since there are two distinct operations. However by concatenating the predicates using logical AND, the same result can be achieved with one pass. In other cases, the *select* operation can be replaced by a more optimal, specialised variant. For example, calling *select* and then immediately retrieving the first element is wasteful since the entire source collection is traversed, whereas with *selectOne* the first element which satisfies the predicate is returned, avoiding unnecessary further iterations. The same applies to checking whether an element satisfying the predicate exists in the collection (which can be delegated to *selectOne*), or calling *select* on unique collections (since there can only be one element, so again can be replaced with *selectOne*).

Other optimisations are possible beyond examining common query collections – for instance, unnecessary model navigation, which is often costly both in terms of time and memory consumption. This cost is especially amplified when the modelling technology / driver is suboptimal and doesn't offer lazy loading.

Static analysis is in general useful for adapting a generic toolset for specific use cases, because it enables the execution engine to assert some simplifying assumptions on a case-by-case basis. We have seen an example of how this principle can be applied for suboptimal code detection. Going further, static analysis can also be used to prevent unnecessary model loading, as demonstrated by Wei et al. (2016) [236]. The idea is that if a model management program only uses a subset of the (meta)model, prior analysis can reveal which subsets of the model need to be loaded and discard the rest; or at least attempt to where possible by using empty placeholders. This is achieved through the *effective metamodel*, derived by extracting model footprints – i.e. any model elements, properties and references navigated by the program. Obtaining the effective metamodel (the subset of the model used by the program) is relatively straightforward as the authors build on the static analysis framework mentioned previously. However the complex part is using this information to optimise the model loading process. The authors opt to use XMI; the *de-facto* model serialization format used in the Eclipse Modelling Framework. As XMI is an XML format – and thus hierarchical / tree-based in nature – the parsing algorithm needs to be modified to avoid eagerly loading all elements top-down. The default SAX parser is listener-based, so it can be overridden to perform different actions for each element encountered. In this case, the effective metamodel can be checked each time a handler is invoked to determine whether the element or property should be loaded. If not empty placeholders / null references can be used instead to keep the model consistent with the metamodel and XML document structure.

Note that partial loading is different from the notion of lazy loading. The former proactively modifies the loading process prior to program execution, so it is only useful in the classic case where the model is loaded upfront. By contrast, lazy loading does not require any analysis; model elements and properties are loaded on-the-fly (and possibly cached thereafter) only when the program navigates the model as required. Of course, lazy loading is more desirable as it does not require an intermediate analysis process or modification of the loading algorithm, however for traditional XML formats this is not possible. Unlike lazy loading, the partial loading solution can be applied to popular modelling formats and so is desirable when working with existing models without migration to more scalable formats which allow random access, such as databases or key-value stores.

Finally, it is important to realize that not all static analysis is aimed at improving performance. Whilst powerful static analysis capabilities can refactor suboptimal expressions for better performance, they can also be used to make them more readable or idiomatic. This is especially applicable for generated code, which is notoriously verbose and contains redundancies or unintuitive expressions. Recognising that readability and performance are both important factors and not necessarily mutually exclusive, Cuadrado (2019) [237] describes a detailed catalogue of optimisations for OCL. These are designed to improve code quality in terms of performance and readability. These include almost every language construct available in OCL: from removing dead if/else branches to optimising iteration operations, type comparisons, literals etc. as well as more nuanced optimisations applicable to OCL, such as the non-short-circuiting nature of conditionals, which can be optimised in a more intelligent way than a series of nested *if* statements. These kinds of optimisations were tested for correctness by comparing the original to the proposed optimised form for equivalence using runtime verification. The optimisations are applied at the abstract syntax level, using models of the program conforming to the language metamodel. Thus, the optimisation engine is itself a model-to-model transformation. Whilst the optimisations and approach are clearly target towards generated OCL code, the principle behind some of the optimisations is still generalisable to other languages, although inevitably imperative constructs (i.e. non-functional nature) may complicate this or reduce the extent of their applicability.

2.4 Database Query optimisations

Whilst the field of Model-Driven Engineering is relatively young, many of its challenges are in principle very similar to those in related fields which long predate MDE in its current form. The word "model" is notoriously overloaded not only in Computer Science but in other disciplines too. However, in the field of MDE, the term is perhaps more generic than in other disciplines. Therefore, a model may be, for example, a graph, or "just data" or both. Models can take many forms. One such form is in databases, which may be relational, object-oriented, document-based, graph-based or any other kind. Databases have been around for a long time and evolved over the years to accommodate more complex forms of data and patterns of usage. If we think of a model as a database, then much of the research and technologies developed in the context of databases becomes relevant and of interest to MDE tooling. Most, if not all model management tasks involve model querying, which is usually performed using a dedicated language such as OCL, or general-purpose programming language like Java. Perhaps the most well-known query language for databases is SQL, however suitable alternatives for non-relational (also known as "NoSQL") databases also exist, such as Gremlin [238]. Research in optimising model queries and optimising database queries is likely to have substantial overlap, since in both cases the end goal is to return a view of a subset of the data in the database (or model). The means by which this is achieved is offering filtering and mapping capabilities, often through dedicated first-order logic operations or language constructs. For example, in OCL we have the "*select*" operation, which takes as input a predicate (Boolean lambda expression) that tests whether a given model element should be included in the results. In SQL we have "*SELECT * from X WHERE*", which semantically are very similar: the goal of both the SQL and OCL operations is to return a filtered subset of some source collection where each element satisfies some arbitrary user-specified criteria.

In a short survey paper, Ioannidis (1996) [239] decomposes the process of query optimisation into several "modules", each with a specific goal. Perhaps the most common and easiest approach is *rewriting*. This technique employs static analysis to transform a query into a (hopefully) more efficient form. In the previous section, we saw some examples of this, particularly in the work of Wei and Kolovos' suboptimal code detection. The principles are similar, however as the name implies, the difference is that the execution engine should be able to not only detect suboptimal expressions but also replace (*rewrite*) them with a more efficient form whilst producing the same result, in a way that is transparent to the user. Whilst these kinds of optimisations are performed by especially well-optimized MDE tools, in databases they are often commonplace / standard features – in other words, query rewriting is only a starting point ("low hanging fruit").

Most commercial database query engines employ more advanced optimisations at the execution stage, referred to as a "*query plan*". This involves a search strategy that evaluates the available access plans at each stage of the query and selecting the best one according to a *cost model*. A query plan is typically represented as a tree, where each node is a single query operation. Many optimisations focus on the *join* operations, which combine columns from different tables in relational databases. There are different algorithms for performing joins and the ordering can greatly affect the performance, though regardless the end result will be the same. It is the query optimiser's job to select the most efficient method and order to perform the operation. As Ioannidis notes, the most important strategy is based on a dynamic programming algorithm as pioneered in IBM's System R database in the late 1970s. Once suboptimal plans have been pruned, the plan with the lowest cost – according to the cost model – is chosen. An important metric in the cost model is the estimated size of results for a given query, which are often based on histograms. Another consideration in query planning is ensuring that the actual planning stage is not too expensive: the time spent searching for an optimal execution plan is time that could be spent on executing the actual queries, so the system must ensure that the upfront investment in finding more optimal alternatives is worthwhile. Accurate cost estimation can be difficult and expensive, so more efficient heuristics are needed, hence the reliance on cardinality.

Modern relational databases such as Microsoft SQL Server [240] and Oracle Database [241] not only employ advanced query optimisers but can also leverage multiple threads to improve performance. The degree of parallelism is configurable; however the application of parallelism is at the discretion of the execution engine. The co-ordination overhead of parallelisation is taken into account by the cost model and compared to the expected benefits based on, for example, the number of elements to process, the type of query and the available degree of parallelism. In cases where the maximum degree of parallelisation is unavailable, a lower degree is attempted before abandoning parallelism. The query optimizer can automatically determine where and how to optimally apply parallelism in the workflow. Operations such as sorting and index scan (i.e. selecting the appropriate records based on a predicate) are executed with the maximum available degree of parallelism.

Chaudhuri (1998) [242] provides an overview of query optimisation techniques for SQL databases available at the time. Despite this publication being over two decades old, the paper notes that even at the time research in this field was extensive. Databases – especially relational ones were and (albeit to a lesser extent) still are a cornerstone of large enterprise applications, so it is unsurprising that query optimisations have been extensively researched, especially by tech giants driven by competition. The bulk of advanced optimisations in the literature come from finding an efficient execution plan for complex queries involving many subqueries and in particular *join* operations. Of course, there is also a great deal of research on simplifying complex queries and flattening nested ones, as well as on improved cost estimation for query plans. Perhaps more surprising is the extent of research on parallel and distributed query processing at a time when multicore processors and simultaneous multithreading did not exist. However much of the work on distributed databases was on distribution of the data itself, as noted in [242]. In some cases, the entire database may be replicated across multiple nodes (computers), with queries being split to effectively make use of multiple systems. These replicated architectures require synchronisation between nodes to maintain consistency, which itself is another area of research. In other cases the database may be split over several nodes, and a query planner may be able to exploit knowledge of what data is stored in each node to maximise data locality for a given query. The scheduling of query operations on distributed systems is another interesting subset of research in query optimisation as it adds an additional dimension to the planning stage. The communication and co-ordination overhead of this also factors in to the cost model, so execution planning with multiple nodes / processors whilst also factoring in the already advanced optimisations in single-processor sequential queries becomes significantly more complex. Finally, Chaudhuri touches on even more advanced optimisations which at the time were relatively novel. These include using runtime information to improve query planning, which requires a certain level of laziness in the query optimisation infrastructure. Another optimisation is for the execution planner to be able to take into account and reuse previously constructed ("materialized") data views, which requires being able to rephrase queries to make use of existing views, and evaluating the relative cost of doing so.

We have established that modern databases – at least relational ones – have many advanced optimisations built in to their execution engines which go well beyond rewriting inefficiently expressed queries in terms of other constructs. Whilst many elements of such research generalise beyond the realms of SQL databases, the further one ventures from the relatively narrow confines of SQL and intricacies if inner and outer joins, the more difficult it becomes to apply the same level of query optimisation to more generic operations.

In the modelling community, database-backed models are not uncommon especially when dealing with very large models which cannot be efficiently stored, queried and manipulated in memory. In these cases, some research emphasizes delegation of operation to the underlying database where possible. For example in the JDBC driver of Epsilon, *select* operations are transformed into native SQL queries [231]. The challenge here is that model management languages do not offer the same constructs as SQL, so an additional translation layer is needed. Although database query planners will try to optimise the translated (generated) SQL code, it could be improved if the optimisation process could be performed in a single pass on the original query. One solution to this misalignment between generic first-order operations and database-specific parlance is to integrate a subset of the database querying language into the model management language. Such an idea already exists for the C# general-purpose programming language, for example [243]. The idea behind LINQ (Language INtegrated Query) is that an SQL-like constructs become available for declarative querying of collections in the host language, regardless of the data source. That is, LINQ expressions can be used not only for database queries but also for querying generic in-memory collections. This makes queries more uniform and readable by those familiar with databases but less so with general-purpose programming constructs. However, LINQ expressions are not necessarily more readable than, for example, declarative OCL or Java Stream-like operations. One possible avenue of research is to translate OCL-like operations into LINQ, though for optimal results some sophisticated static analysis would likely be required.

In conclusion, although the worlds of modelling and databases overlap and share some common goals and challenges, it is often the case that MDE technologies build on top of databases and as such, any research involving well-established areas in the database community such as query optimisation are complementary in nature. Because not all models are backed by databases, model management languages are not perfectly aligned with database management languages like SQL. MDE tools and research take a more abstract and generic view of what constitutes a model, arguably more so than database-oriented tooling and research. Furthermore, model management tasks often involve more complex queries and data manipulation – especially in the area of model-to-model transformations. Model management languages therefore offer more generic and powerful capabilities which go beyond the scope of what even the most advanced SQL query optimisers can handle. With imperative constructs and complex language features thrown into the mix, query optimisation in model management languages becomes a much broader research area and a noteworthy aspect to consider in language engineering.

2.5 **Summary**

This chapter has provided a broad overview of model-driven engineering and its challenges, specifically relating to scalability. We have seen that model-driven approaches are used in domains where there is a high degree of engineering complexity and many stakeholders. Notable examples include automotive, aerospace and telecommunications. Given that model-driven engineering is well-suited to large and complex projects, thanks to its abstraction, it is unsurprising that the models and other artefacts in such domains are also large and complex.

We have also seen that scalability is one of the main barriers to adoptions of model-driven engineering. The limitations of most mainstream modelling tools and programs to deal with large models, queries and transformations inhibit further adoption in places where size and complexity is at its greatest. This is somewhat of a paradox in the state of MDE: the primary candidates for exploiting the benefits of a model-driven approach also bear the greatest cost and pain points of doing so. In order for the benefits of MDE to outweigh its costs in many projects, such costs must be addressed by researchers and tool developers. We have seen that increased productivity is one of the main motivations for adoption of model-driven engineering. Yet we also know that the performance of model management tools presents a significant cost in terms of productivity.

To tackle the challenges with poor performance arising from large models and complex model management programs, we have provided an overview of general-purpose solutions to improving software performance and efficiency. We have identified three main techniques: incrementality, laziness and parallelisation. We provided a detailed overview of these techniques, as well as variants which combine lazy and incremental execution (or *reactivity*), and various aspects and terminology relating to concurrency, parallelism, asynchronicity and distribution. We also showed that a functional style of programming is well-suited to applying these optimisations.

We provided a detailed review of research and state-of-the-art in applying these optimisations to model management programs. We have seen that this is an active research area, although the history of addressing performance (and more broadly, scalability) can be traced back to the mid-2000s, when model-driven engineering was still in its infancy. Overall, there is a great deal of interest in improving the performance of model management programs – especially model-to-model transformations. The MDE research community has employed a wide range of techniques including incrementality, laziness, parallelisation, distribution and even combinations of these to alleviate scalability challenges and make model-driven engineering more suitable for domains where performance is critical, such as real-time systems.

Finally, we acknowledged that there is a tremendous volume of research in query optimisation for relational databases, and the world of performance optimisations in model management and databases has some overlap. We argued that the MDE approach to data persistence and processing is arguably more diverse and abstract than what is commonly found in the database literature, and so the two areas of research are complementary.

In summary, we have provided all necessary background for this research: an overview of model-driven engineering, its uses and limitations; an overview of optimisations techniques; and a cross-section of how such techniques (and others) have been applied or proposed in the MDE domain. The key takeaways from this chapter are as follows:

- *Model-driven (software) engineering* promotes models to first-class artifacts in the (software) development process. Models are an abstraction of a system which can be constructed and visualised textually or graphically.
- For the purposes of this thesis, a model can be thought of as any structured data conforming to some predefined schema (i.e. metamodel).
- *Model management* refers to the process of interacting with models to perform tasks such as querying, validation, comparison, transformation, pattern matching, text generation etc.
- Scalability is a well-established concern with model-driven engineering and is multi-faceted.
- Regarding efficiency and scalability of tooling, there are (at least) two major research themes: Model Persistence and Model Management.
- Existing research on improving the execution time of model management tasks primarily focus on and target model-to-model transformations.
- Incremental and lazy execution strategies have been explored due to the more drastic performance benefits when working with very large models. There is comparatively less research on using parallelisation to improve the performance of these tools.
- The motivations for parallel and distributed execution are more apparent now than they were a decade ago, when much of the early research into scalability was being carried out.
- There exists some overlap between scalability research in the model-driven engineering, graph transformation and database communities, though much of this is complementary.

The findings of the literature review are analysed further in Section 3.1 and contextualised in the scope of this thesis. Whilst the length and breadth of Section 2.3 is indicative of research interest in this area, there remains a shortfall in the state-of-the-art with regards to performance optimisations in model management. Specifically, most of the work in this area targets model-to-model transformations. Much less attention has been given to “read-only” (i.e. those which do not modify a model) tasks. Furthermore, most optimisations focus on eliminating redundant computations through incrementality and laziness, rather than trying to speed up the computations by taking advantage of modern hardware through parallel and distributed processing.

3 Analysis and Hypothesis

This chapter will outline the research goals of the thesis in light of existing works described in the preceding chapter and attempt to justify the contributions as well as the approach. The aim is to provide a bridge between the literature review section and the remainder of the thesis. Section 3.1 highlights the gaps in the literature with regards to scalable model management programs. Section 3.2 states the main hypothesis and outlines the objectives of the research as well as an overview of the research methods used in achieving them. Section 3.3 gives details of the preliminaries common to the evaluation sections of the following chapters. Section 3.4 discusses potential limitations and generalisability of the research.

3.1 Analysis

In the previous chapter, we established that scalability is a fundamental and multi-faceted challenge with model-driven engineering. In particular, we argue that model-driven engineering is most beneficial in large and complex projects (based on the nature of MDE and the weight of evidence / use cases described in section 2.1.7), since the use of different models for various stakeholders and/or stages of the project enable communication at the appropriate level of abstraction whilst maintaining a “single source of truth”. One of the primary benefits of model-driven engineering is the ability to use common high-level artefacts throughout the project. Given a model, we can proceed to validation through task-specific languages, transformations to intermediate models for various stakeholders and finally, often, to code generation. Other tasks may include querying, comparison, merging and pattern matching. All of these tasks can be achieved using declarative languages with constructs designed specifically for one purpose, effectively eliminating the need for boilerplate code in general-purpose programming languages and allowing users and developers to express domain logic concisely. As we saw in the previous chapter, the motivation for adopting a model-driven development approach is to improve productivity. In practice, productivity depends heavily on the available tooling and, as discussed, the tooling is far from perfect.

The MDE community recognizes the importance of efficient and user-friendly tooling and research in improving the performance of model management tools has become increasingly active in the past decade. Based on a survey of the literature as demonstrated in section 2.3, there are three predominant optimisations: *incrementality*, *laziness* and *parallelism*. Each of these approaches can be applied to various model management tasks, and even a combination of all three optimisations may be possible depending on the nature of the task, its semantics and implementation (realisation). That said, although the MDE research community has made some progress in applying these techniques – especially to the widely used ATL model-to-model transformation language, considerably less attention has been given to other model management tasks. Most research in model-driven engineering is targeted towards model-to-model transformations, including research ideas proposing improvements to performance of modelling languages and their execution engines. Although model-to-model transformations are seen by many as the central concept in model-driven engineering – or at least in model management – there are other model management tasks which may be equally if not more important in model management workflows. In some cases, model-to-model transformations are absent entirely in projects, especially those where the metamodel is relatively simple and so does not require intermediate models and transformations. However the tasks of validation and model-to-text transformations, as well as model querying, are undoubtedly useful and often necessary tasks. Model validation enforces invariants on models by effectively extending the metamodel with user-specified logic, which is too complex to be expressed using object-oriented metamodelling languages such as Ecore and MOF. Model-to-text transformations are used for generating executable code and documentation from models, making them tangible artefacts which can be distributed, executed and understood using standard, widely-supported general-purpose tools rather than confining artefacts to the specialised MDE toolchain and ecosystem in which the models are defined (for example, EMF).

The current state-of-the-art in MDE performance engineering is primarily centred around incrementality and model-to-model transformations. Although progress is being made on more scalable model storage and serialization technologies than the traditional XML-based formats – which are verbose, not the most user-friendly and typically not modular, requiring full in-memory parsing – for example by using databases instead, the underlying modelling technology is essentially orthogonal from model management tasks. Although more optimised modelling technologies can reduce the overhead of performing model access (Create/Read/Update/Delete) operations, further optimisation is possible at the task level.

We argue that the literature's heavy focus on model-to-model transformations limits the scope of possible optimisations and performance gains from parallelisation. This is because model transformations are complex and need to mutate an output model, where the order of rule applications becomes important. By contrast, model management tasks which only read from models are less constrained and so can exercise greater flexibility in their execution algorithms.

We have already discussed the merits and shortcomings of incrementality, laziness and parallelism in sections 2.2 and 2.3. The rationale for incrementality is clear, especially in the case of model-to-model transformations where there are intermediate models which need to be updated by re-running transformations in full even when there are minor changes in the source model. Combining incrementality with laziness gives us *reactivity*, where changes are efficiently propagated to their destination only when the result of the change affects the final state. This kind of optimisation brings about the largest performance benefit in practice, as it does not penalise users for making small changes. The benefits can be thought of as being similar to how modern integrated development environments (IDEs) handle builds. Incremental compilation allows applications to be modified with the need to rebuild the entire project, which can have a significant impact on developer productivity.

Although incrementality and laziness improve performance significantly by avoiding unnecessary computations, they do not offer a speed advantage when computations are required. Incrementality has no performance benefits (and can degrade performance due to the trace caching overhead) when the source model or program is run for the first time, or if the tracing information is invalidated or not present. By contrast, parallel execution improves performance by making better use of available hardware. Parallelisation is arguably more challenging to implement due to concurrency issues and non-determinism. Adapting a sequential program to be parallel and thread-safe is far from trivial as there are many moving parts and shared data. Without applying a formal model of concurrency and using analytical techniques, it is difficult, if not impossible, to guarantee correctness and to prove there are no concurrency issues which could potentially alter program execution in undesirable ways. To avoid such issues entirely, one possible approach is to avoid shared-memory processing and opt for a distributed approach, where instead of multiple threads, multiple processes are used instead. However, such an approach entails high memory and communication cost, due to the inability to share common in-memory data. Moreover, parallelisation not only requires a thorough understanding of the execution engine internals and possible thread-safety issues but also a parallelisation algorithm, so that the program can be decomposed into independent units of execution (herein referred to as "*jobs*") where the order of execution is not important. In short, parallelisation is perhaps the most difficult of the three optimisations to implement and also presents great uncertainty about the correctness of the program. Furthermore, whilst the gains from incrementality and laziness are large – often making execution instantaneous – the same is not true for parallelism. Parallel algorithms almost always perform more computations than their sequential counterparts, with additional overhead for setting up jobs through decomposition, communication overhead from multiple execution threads or processes, merging of results, synchronization and any other changes to the algorithm or data structures which are necessary for correct functioning. Making matters even less appealing is the fact that performance gains are heavily dependent on the underlying hardware: more cores generally result in higher performance, however due to thermal design limits higher core utilisation results in lower overall clock speeds. Combined with the parallelisation overhead – which is not necessarily a fixed constant but may scale with the number of jobs – it is perhaps not difficult to see why parallelisation has been given a much lower priority in the literature compared to more straightforward and immediately beneficial optimisations.

Another contributing factor is that the number of cores on mainstream computers has remained relatively low, with dual and quad-core systems being the norm until recent times. Even with simultaneous multi-threading (SMT) technology, the effective possible parallelisation is double the number of cores. At the time of writing, even high-end laptops "only" have up to 8 processor cores. However with improvements in fabrication process and CPU architectures, HEDT processors such as the AMD Threadripper line and server-grade CPUs like Intel's Xeon and AMD's EPYC have seen significant advances in the number of cores, to the point where even a single-socket CPU can have 64 cores (for example, the Threadripper 3990X [244]). Meanwhile more consumer-oriented desktop processors have up to 16 cores with octa-core CPUs being much more affordable, effectively replacing quad-core CPUs at the high end in a similar way to how quad-cores replaced dual-cores a decade ago. High-performance hexacore and octa-core processors are increasingly common in laptops, whilst octa-core processors have been common in smartphones for several years.

Despite the relatively low scalability potential with individual computers, the advent of cloud computing and the relative maturity and ease with which applications can be deployed in a scalable manner through containerisation technologies like Docker and cluster management software like Kubernetes, the case for parallelisation becomes clearer in distributed computing environments. For model-driven engineering tools to be used in applications designed with scalable cloud architectures in mind, it is necessary to be able to decompose heavy monolithic single-threaded execution into more lightweight units of work. The literature has already made some progress on scalable model persistence through distributed data stores and database technologies; however the same degree of progress is distinctly lacking when it comes to execution of model management programs such as validation, comparison and text generation.

3.2 Research Hypothesis, Objectives and Scope

In this section, we outline the plan for the thesis: its goals, the means by which they will be achieved and how they will be evaluated.

3.2.1 Hypothesis

The hypothesis of this work is that model querying, validation and model-to-text transformation are amenable to both fully and semi-automated parallelisation, leading to substantial reductions in execution time in proportionality to the number of available processor cores. Furthermore, it is possible to exploit the finite and deterministic nature of these programs to efficiently distribute the work amongst multiple computers, scaling with not only the number of cores in a single machine, but all cores across multiple systems. This hypothesis applies particularly to rule-based and/or declarative constructs used in dedicated task-specific languages.

3.2.2 Objectives

The ultimate goal of this thesis is to assess whether an efficient, scalable and generalisable parallelisation and distribution approach for rule-based model management tasks is possible. This is a broad aim, so to be more specific, we decompose this into the following objectives:

1. Identify and provide solutions to the issues of retrofitting concurrency to at least one existing model management language, task or platform. In particular, this means finding common "points of failure" in data structures and algorithms of the execution engine which need to be modified to allow for correct application of parallelism.
2. Develop an efficient parallel decomposition of the execution algorithm, which ideally should be scalable with the number of model elements, since the main challenge we are trying to address is poor performance with large models.
3. Devise a systematic and thorough way for testing the developed parallel execution engine to ensure consistency in its semantics and results with the original sequential implementation.
4. Investigate potential ways in which the parallel decomposition and execution engine can be modified and/or extended to scale not only with multiple threads on a single process (i.e. shared memory parallelism), but with multiple, independent process (i.e. distributed memory parallelism). The approach should ideally scale linearly with the number of processes (assuming one process per computer), and also be able to take advantage of the resources available to each computer (i.e. the number of cores and available memory).
5. Attempt to generalise the parallel / distributed approach to other model management tasks. This could mean re-using the developed infrastructure in more interesting ways, applying it to a broader range of parallelisable aspects of model management programs. It could also mean applying the devised approach to another model management task.

6. Evaluate the scalability of the parallel and distributed implementation(s), using appropriate model management programs. The benchmarks should assess the scalability with the number of cores and threads in the parallel case and the number of computers in the distributed case. The implementation(s) should be tested using programs and models of various sizes and complexity, ideally on various different hardware. Experiments should aim to critically assess the range of performance benefits attainable through parallelisation and the factors which influence this.

3.2.3 **Scope**

To further narrow down the objectives into specific actionable items, this subsection will attempt to clarify the concrete plan for the thesis.

In the interest of time and feasibility, we target the Epsilon family of languages for several reasons as discussed below. We choose the Epsilon Validation Language (EVL) as the primary focus for our parallel and distributed algorithms. Since model validation does not modify the input model or have any ordering requirements for the execution or results, this is an ideal starting point. Thus Chapter 4 aims to fulfil objectives 1 to 3. Chapter 5 aims to fulfil objective 4 by building on the infrastructure developed in Chapter 4. To satisfy objective 5, Chapters 6 and 7 demonstrate how the parallelisation approach developed in Chapter 4 can be further developed and generalised to improve the performance of other model management tasks. Since model querying is a key aspect of all model management tasks, Chapter 6 shows how parallelisation can be used to speed up declarative querying and simple transformation operations. This further develops the parallelisation infrastructure to be able to handle a broader and more complex range of scenarios where parallelisation can be applied. Meanwhile, Chapter 7 aims to demonstrate how the approach developed in Chapter 4 can be re-used without significant modification for another rule-based model management task: co-ordinating the execution of template-based model-to-text transformations using EGX. Finally, to satisfy objective 6, we conduct numerous experiments to benchmark the performance of each of the developed implementations at the end of each technical chapter. The range of hardware and our methodology is discussed in section 3.3.

Ultimately, our goal is to demonstrate that a data-parallel decomposition of model management program execution algorithms is possible not only for rule-based languages for tasks such as model validation, but also for more general-purpose declarative query and transformation operations. We argue this is made possible by the task-specific, rule-based nature of such languages. Therefore, this thesis can be used as partial justification for the existence of task-specific model management languages as well as declarative approaches to model querying, which provide a structure that is amenable to optimisations such as parallelisation even for complex, feature-rich languages with imperative constructs such as those of Epsilon.

3.2.4 Motivation for targeting Epsilon

The motivations for choosing Epsilon as our implementation testbed are numerous. At first, one of the most compelling reasons was that the Enterprise Systems research group at the University of York leads the development of the project, so there is a high level of familiarity, technical expertise as well as influence over the codebase. This gives us the flexibility to experiment more liberally with the code and an opportunity to integrate the developed solutions into the codebase and receive feedback from users. This makes the contributions of this project practical and more directly useful, rather than purely academic and theoretical. Given the highly technical nature of the project and challenges associated with concurrency, it is important to consider the concrete realisation of proposed solutions; namely how they impact the existing codebase and consequently the behavioural semantics (i.e. compatibility with the current implementation's specification).

Epsilon is a highly extensible platform (the "E" in Epsilon stands for "Extensible"). Unconstrained by a strict specification or governing body – unlike more commercially-focused tools conforming to the Object Management Group (OMG) specifications such as OCL and QVT – Epsilon is more flexible in not only its extensions but also its current implementation. Moreover, this means Epsilon can support any modelling technology and is not tied to Meta-Object Facility (MOF)-compliant implementations such as the de-facto Eclipse Modelling Framework (EMF). The architecture of Epsilon is deliberately designed to decouple its model management languages from specific modelling technologies. The modularity of Epsilon – in particular the model connectivity layer (EMC) – enables us to experiment with a diverse range of modelling technologies, from conventional XML-based formats to spreadsheets / CSV, JDBC databases and even proprietary tools such as Simulink. This allows us to better investigate the generalisability our solutions by using models with vastly different underlying implementations. After all, scalability challenges are primarily rooted in the size of models, and so performance is highly dependent on the modelling technology's implementation – and in our case, the EMC driver for the technology in question. Although this project's focus is on parallelisation of model management tasks, performance and thread-safety issues at the modelling technology layer are inevitable, so it is important to consider multiple modelling technologies when assessing the scalability of our solutions. The rationale is that although EMF is widely used within the MDE community, it is a specialist form of modelling when compared to more widespread standards such as schema-backed XML (XSD) and spreadsheets or CSV files.

All of Epsilon's languages build on top of a common core – the Epsilon Object Language (EOL) – with all rule-based languages extending the internal Epsilon Rule Language (ERL), which makes it much more productive to develop, present and implement our solution and port it to other model management languages with ease. In other words, we can start by developing our solution for a specific model management task such as validation in EVL and then abstract / migrate the parallelisation framework and concurrency constructs / modifications to the ERL and EOL codebase, which we can then re-use when parallelising other rule-based languages. We chose to present our solution with a bottom-up approach (starting with EVL in Chapter 4) rather than a top-down approach of presenting ERL first. The rationale for this is that ERL is an abstraction, and without a concrete working example it may be more difficult to understand the challenges and solutions. We describe the generalisability of our solution in Section 8.2.

Although Epsilon's model management languages have a textual concrete syntax, this research is focused on the execution engines and is not bound to the textual aspects. Thus, the findings presented in this thesis can also be applied to languages with graphical notations, so long as they have a comparable abstract syntax.

Epsilon's model management languages are arguably the most powerful in terms of features and language constructs when compared to alternatives like OCL. EOL's heavy reliance on the reflective capabilities of Java makes Epsilon's family of languages amongst the most powerful and feature-rich, giving users a broad set of both imperative and declarative language constructs to express their domain logic. By choosing to target model management languages which are the most complex in terms of language features, we are effectively considering the "worst-case" scenario by having more challenges to encounter and overcome. Thus, we can argue that our developed solutions are generalisable in the sense that they can support languages which are at most as complex as Epsilon's. Therefore, simpler languages like OCL – which has a relatively thin standard library and is purely functional – could make use of a subset of our solutions by virtue of being more restrictive and having fewer features.

Finally, Epsilon is a well-established open-source Eclipse project, with notable industrial users as well as being used for research projects in academia [245].

In summary, the rationale for using Epsilon as our implementation vehicle is clear: it allows to us demonstrate our thesis with the least resistance, maximum impact, highest productivity and to parallelise the most languages, experiment with ideas the most etc. whilst also being the most generalisable because we're using the "worst-case" scenario of supporting both imperative and declarative (most complex languages) and any modelling technology. The project's extensibility, powerful languages and current structure makes it convenient as a starting point and also helps to set a good example of a reference implementation which other tool vendors can examine and adapt to their needs.

3.2.5 **Non-goals**

In the previous subsection we justified our choice of Epsilon for implementing our solutions, noting the high extensibility and expressiveness of its languages. However, we should clarify that our goal is not to provide parallelisation for the most generic possible case. In other words, the objective is not to provide new or alternative parallelisation constructs for general-purpose programming languages like Java. Indeed, such parallelisation constructs already exist in the language in the form of Streams. The premise of this thesis is to show that rule-based model management languages are parallelisable in a highly scalable manner and share similar fundamental characteristics which make such parallelisation desirable.

Similarly, we do not endeavour to support parallel execution for all modelling technologies supported by Epsilon or to evaluate our solutions on all EMC drivers. The fact that Epsilon's languages are decoupled from the underlying modelling technologies is an attractive property of the platform because it enables us to cleanly separate concurrency concerns at the language level independently from the data source. This thesis focuses on data parallelism, where the data comes from models. EMC allows us to treat such data in its most abstract form, which is ideal for generalising a solution which can be applied to many different modelling technologies. This is convenient for illustrative purposes, showing that our solution does not depend on a particular modelling framework. By contrast, if we were to choose Eclipse OCL as our implementation target, we would be tied heavily to the EMF API and it would be more difficult to show that our solution generalises beyond the realms of EMF to less esoteric forms of data modelling.

The fact that Epsilon's languages support imperative constructs and ability to invoke arbitrary Java code through reflection does not imply that we intend to parallelise as many parts of EOL as possible, or even to ensure deterministic and/or equivalent behaviour with the sequential implementation when using advanced features. Whilst imperative constructs provide users with greater expressive power where declarative constructs are insufficient, they naturally make parallelisation much more difficult, especially in the absence of static analysis. The fact that Epsilon has many language features enables us to investigate how far we can take parallelisation and the degree to which parallelisation can be applied whilst still maintaining compatibility and leveraging advanced language features. By targeting the Epsilon family of languages, we can investigate how limiting (or not limiting) our parallelisation solutions are, what restrictions they place on the language and what features pose the greatest challenges to support. Our goal is to try to support as many of Epsilon's features as possible and to avoid the need for a limited subset of the language to make users choose between expressive power and performance. Where possible, we endeavour to tailor our solution such that compatibility with existing programs is preserved to minimise the effort required by end users to benefit from parallelisation.

It is not a goal of this thesis to outperform compiled code, or any lower-level approach to computation (e.g. programs written in Assembly, C++ or targeting FPGAs) through parallelisation. Nor is it a goal to "compete" with orthogonal approaches like incrementality and laziness: on the contrary, we would ideally like to combine all three at some point. Although the central motivation of this project is to improve the performance of model management programs, our contribution is ultimately about demonstrating patterns and techniques through which this can be done, primarily through parallelisation. As such, we argue that the approaches presented in this thesis are applicable to more optimised implementations provided they follow a similar structure. For example, one could feasibly write a compiler for Epsilon's languages to remove the (extremely large) overhead associated with interpreted languages. This would not invalidate our approach, as the structure of programs and the execution engine would be the same. The fact that Epsilon is interpreted is purely coincidental: to our knowledge, there is nothing in our approach which limits its applicability to only interpreted languages (or indeed only to Epsilon). For reasons already discussed, Epsilon provides a highly convenient platform to build our solutions upon for demonstration purposes.

3.3 Evaluation Strategy

From the beginning of the project, the evaluation criteria guided the development of solutions. In this section, we introduce these criteria and how they are measured.

The primary motivation of this thesis is to enhance the performance of model management programs through parallelisation. Given the challenges identified with concurrency in previous sections, it is important to consider the *correctness* of the proposed solutions, as well as the performance gains. The criterion of correctness is arguably far more crucial than performance; it is difficult to identify any scenarios where a faster program which produces incorrect results (or at least produces correct results in some cases) is preferable to a program whose behaviour is consistent, deterministic and in line with the user's expectations. Therefore, the evaluation strategy will seek to verify correctness (defined as consistency with the default sequential implementation) before considering the performance.

A third criterion for evaluation could be minimising the impact of introducing concurrency and scalable parallelism into the existing code base. Although not as important as correctness or performance, this criterion is relevant from a practical aspect for software developers and architects of existing tools and projects. By considering the additional "effort", code and complexity concurrency introduces into an existing codebase, we can determine the usefulness of the proposed solutions and the impact it could have on the development of existing and future model management tools. In some sense, it enables us to consider the external validity of the solutions because a less complex, better-designed solution which requires fewer modifications to the existing code is far more likely to be adopted by existing projects with established user bases and mature tooling, support, community etc.; thus, arguably, making the contribution of this thesis more relevant to a wider group of tool developers, and in turn, users. Since most MDE tools were not initially designed to accommodate concurrency, a solution which can be retrofitted more easily can help to minimise disruption for both developers and users: rather than requiring new tooling (which breaks compatibility), users of model management programs can benefit from improved performance without migrating their scripts to a new language or tool, for example.

3.3.1 **Input Sources**

Evaluation of correctness and performance requires us to either find or write appropriate models, metamodels and model management programs. Although scripts are task-specific, finding models and metamodels is a one-time cost, as they can be used for all model management tasks. Furthermore, reusing the same pool of models can make our results more consistent across model management tasks, since the model structure and number of elements are the same. Familiarity with the metamodels also makes it easier to write scripts for evaluation purposes, as we don't need to learn a new domain every time.

A significant challenge in our evaluation has been finding suitable models. A model can only be as complex as its metamodel, and for our evaluation, it is important to use models with varying complexity. Finding very large models based on real-world data is challenging, in part due to the domain-specific nature of models. Especially in industrial contexts, large and complex models may not be publicly available as they contain intellectual property or confidential data. It would therefore be inappropriate for organisations to expose models with sensitive information in some cases. Whilst it is possible to automatically generate models from a given metamodel (as demonstrated by

Sen et al., 2009 [246]), such models may not be well-formed or realistic representations of typical real-world models.

A good source of authentic, large models with real-world applications comes from the domain of model-driven reverse engineering. In order to better understand and modernise large legacy systems, a model discovery tool can be used to reverse-engineer the code (or other low-level artifacts) into models which provide a higher level of abstraction, making the system easier to reason about and perhaps transform into a more modern language using model-to-text transformation. One such tool which provides model discovery for Java is MoDisco [85]. Essentially, MoDisco has an EMF Java metamodel of the Java programming language's abstract syntax tree; though at the time of writing only Java versions 7 and below are supported. This is unlikely to be an issue for most legacy software modernisation cases, and there are plenty of open-source code bases written in older versions of Java. Essentially, MoDisco provides (amongst other things) a text-to-model transformation for Java source files.

Amongst the benefits of using reverse-engineered Java code is that the Java language is very complex, leading to a complex metamodel. Furthermore, the size of models obtained can be easily controlled by limiting the amount of code we feed into the model discoverer. The modular nature of Java also allows us to limit the complexity of models – for instance, we may purposely choose code which is simple in its control flow and dependencies to obtain a simpler model. Furthermore, the models obtained are "natural" in the sense that they represent a derived structure of a real system, as opposed to being artificially generated for testing purposes.

Most of our test models and metamodels are obtained from [247]; which contains resources used in the evaluation of jLinTra (the Linda-based concurrent model transformation tool). We found this source to be useful because it contains three different metamodels and accompanying models of various sizes. Of particular interest are the 11 Java models; which are based on reverse-engineered code of the Eclipse Java Development Tools core; ranging from 100,000 (24 MB) to ~4.35 million elements (1.2 GB). There are also 9 models of the Internet Movie Database (IMDb) with similar range of sizes: from ~100,000 elements (7 MB) to ~3.5 million elements (345 MB). Since the IMDb metamodel is much simpler, the models are also smaller in size for a given number of elements. This stark contrast with the Java metamodel enables us to test the scalability of our solutions for two relative extremes of simple and complex metamodels. For reference, the Java metamodel contains 132 top-level objects (i.e. EClasses and EEnums), and 749 elements when including their properties. By contrast, the IMDb metamodel has only 8 top-level objects (40 elements including properties). The IMDb models in our case only have instances of *Movie*s and *Person*s, where there exists a bidirectional relationship between the two.

Finally, we have a model of the DBLP computer science bibliography database, also taken from [247]. In principle, the metamodel for this is structurally very similar to the IMDb one if we substitute "Movies" for "Records" and "Actors" for "Authors", however there are multiple types of publications (books, journals etc.) and each publication has more properties (e.g. page numbers, ISBN etc.). There is only one model for this, which is an XMI file that is 1.12 GB in size. Unlike the Java models, given the simplicity and flat nature of this model, there are more elements of a given type than in the Java models. Specifically, there are 4,162,991 "Records" and 1,462,270 "Authors". This will only be used in our benchmarks to highlight extreme scenarios.

Our experience running experiments with the largest models are that 16 GB of RAM is the minimum needed to work with an EMF model greater than 1 GB in its serialized (XMI) form using only in-memory computing, as EMF was not designed to scale efficiently to such sizes without resorting to non-volatile storage for model access. This does not mean that our solutions are not intended for larger models; rather that our evaluation does not consider more efficient model persistence mechanisms, for simplicity. Future work could focus on experiments with alternative models, which may be larger and use different persistence backends.

3.3.2 **Correctness**

A solution is “correct” if it behaves according to the program specification. Since our solutions build on top of existing model management programs, a convenient way to evaluate correctness is by comparing the program’s results (output) as well as behaviour (where plausible) to the original, non-concurrent solution. Again, since we are introducing concurrency to an established and tested codebase, we can assume that the original implementation is “correct” for our purposes. In particular, our evaluation of correctness is not of the original program or whether its specification meets user requirements, but whether the introduction of parallelism changes the output of the program when executed with identical inputs. However, as the codebase develops and evolves to accommodate concurrency, it may reveal other obscure bugs which may, under certain conditions and inputs, produce incorrect results even with the non-concurrent implementations. Furthermore, we’d like to build confidence in the correctness of our program more generally (i.e. from an end user’s perspective). To do this, we shall also compare the output of our programs with similar tools for the same inputs (or at least, as similar as possible). That is, our correctness evaluation strategy requires three implementations to be produce the same result. For example, if we’re evaluating the correctness of our parallel model-to-model transformation solution (Parallel ETL), we’d compare the results to the original, non-concurrent implementation of ETL as well as a well-known model transformation language such as ATL. If all three implementations give the same result for the same inputs, then we can be confident in our parallel solution’s correctness.

That said, our confidence can only be as strong as the complexity of the inputs to each program. A typical model management program requires at least three inputs: a source model, source metamodel and a script. If our script is too simple – i.e. it performs relatively trivial tasks on the model(s) – then it is unlikely to thoroughly test the execution engine’s capabilities. Similarly, if the source model is too simple (i.e. it has too few elements or simplistic relationships between elements), this limits the complexity of a script and could mean that certain branches of a complex script are never taken. In other words, the maximum level of confidence we can have in our correctness evaluation strategy is strongly dependent on the level of *code coverage* achieved by a given set of inputs to the program. Ideally then, we aim to provide scripts and models of sufficient complexity to test as much of the execution engine’s code as feasible.

As we are interested in maximising code coverage to build confidence in the strength of our tests, it would be wise to carry out white-box, as well as black-box testing. Only observing the results of our programs for a given set of inputs may fail to reveal some bugs with the execution engine’s internals whilst still producing the correct (i.e. expected) output. In other words, some defects may never reveal themselves for our chosen inputs. White-box testing can help to build confidence of the developer's mental model of the execution engine. By inspecting the internal state and intermediate results of a program, we can be confident that our results are not correct by chance, but by logical implication. However, there are limitations to the extent of verification which can be performed at this level of granularity – especially *during* execution – due to the non-deterministic nature of concurrent execution. Where appropriate, the intermediate results / states of the concurrent implementation will be compared with the original (non-concurrent) implementation. "*A convincing demonstration of correctness being impossible as long as the mechanism is regarded as a black box, our only hope lies in not regarding the mechanism as a black box.*" [248].

"*Testing shows the presence, not the absence of bugs*" [248] [249]. The proposed methodology for evaluating correctness does not *guarantee* correctness; it only gives us a degree of confidence – the level of which is determined by the quality and quantity of our test suite. Whilst formal verification techniques and symbolic execution can theoretically be used to make assurances about certain behaviours of our execution engines, the complexity of model management programs and their inputs make it difficult to leverage these formal methods in practice. We argue that the time investment required to be 100% confident by applying such techniques is not necessary or justified when, for example, 90-95% can be achieved through much simpler and less time-consuming means.

We can quantify our level of confidence in our test suite by the amount of code coverage. To do this, we use the Eclipse plug-in "EclEmma" [250]; a Java code coverage analysis tool which can report code coverage at the project, package, class, method and expression level. EclEmma provides its results based on the ratio of covered instructions and total instructions. Of course, we cannot directly use these percentages as a measure of the quality of our tests for a number of reasons. Primarily, there are many instructions, methods and even classes which can only be invoked explicitly by the execution engine. In other words, code that's independent of inputs to the program. For instance, this may be constructors, getters and setters, utilities etc. that may exist for compatibility, convenience or other development purposes. This may also include auto-generated code, such as that generated by Epsilon's ANTLR parser. Another problem is that exception-handling and defensive programming code – which we usually *do not* want to be executed – is considered in the coverage. Perhaps the more general problem with code coverage in our case is dependencies. Epsilon itself has a number of dependencies on other libraries, and since Epsilon itself provides a platform for a family of task-specific model management languages across a range of modelling technologies, the level of granularity we choose for our measurements is important. We therefore limit our coverage analysis to the internals of EOL and the task-specific language under evaluation; since all languages in Epsilon build on EOL. Although this provides a more suitable scope for our analysis, we still face the same problems with code coverage; - namely, the lack of weight given to each instruction, method or class. There are indeed certain parts of the code which are more crucial to determining correct behaviour than others, and so coverage of those "critical regions" should be emphasized more strongly. These regions will of course vary depending on the nature of the program and execution engine. We will therefore postpone discussion of such "critical regions" to the relevant section for each model management program we considered. Due to the limitations of code coverage analysis outlined, we will also limit our discussion of coverage results to such "critical regions", though results of other coverage metrics will be reported if they are found to be interesting or useful in analysing our testing methodology and/or results.

3.3.3 **Performance**

A good overview of performance theory is available in Chapter 2.5 of McCool et al. [101]. Most notable is Amdahl's Law, which states that "*the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.*" [101]. Basically, the overall execution time of an application is always going to be bottlenecked by the serial portion of the code, no matter how fast the parallelisable portion is. Furthermore, improving the algorithms and optimising the code leads to improved performance in both the sequential and parallel parts of the program. Where possible, we try to limit the effects of Amdahl's Law – that is, we benchmark only the parallelisable parts of the program, and disregard parts which are executed sequentially. This enables us to directly assess the efficiency of our parallel implementation and compare results from various experiments.

Evaluating the performance of our solutions is relatively straightforward compared to evaluating correctness. Primarily, we are interested in *speedup* – i.e. how much faster our implementation is compared to some oracle – typically the status quo. As with our correctness evaluation strategy, we compare the original, non-concurrent implementation to the new concurrent version where both implementations are given identical input scenarios (i.e. scripts and models). The speedup is calculated by dividing the original module's execution time by the concurrent module's execution time. Since we are interested in evaluating the relative performance of our solution to a non-concurrent implementation, we do not consider the total execution time of the program, only the part of the main execution engine's code which has been parallelised. For example, since we have not considered parallelising model parsing, we will not include the parsing time in our measurements. This allows us to minimise the effect of Amdahl's law, measuring only the parallelisable part and making it simpler to assess the efficiency of our solutions. We assess efficiency by dividing the actual speedup achieved by the potential. For example, if we are using 8 threads, the maximum theoretical speedup (ceteris paribus) is 8x. If we achieve 6x speedup, then efficiency is (6/8) = 75%.

The granularity of our measurements is at the millisecond level. This should be more than sufficient for our needs, since the execution time of our programs are in the order of seconds at the very least and hours at most. Moreover, we will focus our performance evaluations on larger models for several reasons. Firstly, the purpose of parallelisation is to reduce the execution time, which is only an issue for large models. Secondly, the gains from parallelisation will be more apparent in larger models due to the initial "setup costs" – i.e. the additional overhead of parallelisation and concurrency in general. Thirdly, we can be more confident in the significance of our results when the absolute execution time is greater (which is the case for larger models). For example, if our concurrent solution takes 2 seconds to run compared to the original's 3 seconds, this is far less significant than if the units were in minutes or hours, rather than seconds. This is because various other factors could affect the execution time by small amounts. A 1 second difference may be due to the operating system scheduling, background process activity or even a spike in CPU temperature. A 1 hour difference is much more likely to be the result of differences in the execution engine implementations than extraneous factors. Therefore, whilst the primary performance evaluation metric is speedup, in practice the difference in absolute execution time is arguably of greater importance for users. Nevertheless, measuring speedup enables us to evaluate the efficiency of our solution and identify limits to the scalability with number of cores.

Because Epsilon's implementation is based on the Java platform, the results may vary between runs. As all Java-based languages are compiled to Java Bytecode – which is then interpreted by the Java Virtual Machine (JVM) – the just-in-time (JIT) compiler and virtual machine may or may not make specific runtime optimisations depending on, for example, the number of times a particular piece of code is executed. The inner working of the JVM are complex and often unintuitive [251], though the details of such intricacies and their effects are not the concerns of typical Java programs. Tools such as the Java Microbenchmark Harness (JMH) [252] have been developed to provide more reliable benchmarking methods which can be used to simplify the process by automatically performing warmup runs and measuring various performance characteristics. However, the practical difference to our results in relative terms from various JVM optimisations are negligible, and our preliminary tests show consistent performance under identical conditions. Again, since we are dealing with execution times typically in the order of minutes or more, we need not be concerned by small differences in execution times from various runs. Furthermore, since we're only benchmarking a portion of the code as opposed to the entire program runtime (i.e. the benchmarking is performed in Java itself, not timing how long it takes to run the "java" command with specified parameters), we need not be concerned with start-up times affecting our results. Finally, we'd like to emphasize that our performance tests run in a separate JVM launch. That is, the execution time of each module is measured independently from other modules, such that the invocation of each run is performed manually (or by a bash/cmd script). This ensures that the conditions for each run are fair and representative of a typical usage scenario. An end user is unlikely to perform multiple runs of the same program with identical inputs, and even if they did, each invocation of the program would launch a new Java process.

It's important to emphasize that we are interested in the practical speedup, treating the JVM as a "black box" and only focusing our efforts on the code rather than lower-level details. Put another way, the end user is unlikely to spend weeks trying to fine-tune the garbage collector and tiered compilation thresholds to get optimal performance, as this is a false economy: the time spent tuning could be spent more productively on other activities, including just running the program with the default settings. Hence, we try to stay reasonably close to sensible defaults and not get carried away with the vast configuration parameters available in commercial JVMs. That said, it should be noted that our program is primarily focused on throughput, so any straightforward and obvious JVM options which would improve performance for this scenario may be applied where appropriate. To be clear, it is not that JVM tuning is not valuable, but that for our purposes it is mostly unnecessary since we are interested in *relative* performance of our parallel solution compared to the status quo, not improving the absolute performance. This means that the same JVM options will be applied to both the baseline and our parallel program, so any optimisations should apply to both; albeit admittedly the effects won't be identical. As noted by Berger, performance can be affected by almost anything, even the length of the username running the process [253]! Since computers are so complex and the level of programming presented in this thesis is at a relatively high level of abstraction, we cannot reasonably control for everything that could significantly affect performance, only make best efforts to ensure a consistent methodology.

Regarding JVM tuning, we have experimented with various flags which are expected to significantly affect performance. These include tuning the maximum garbage collection pause time (-*XX:MaxGCPauseMillis*), using the throughput garbage collector (*-XX:+UseParallelGC*) which was the default prior to Java 9, using the NUMA-aware flag *-XX:+UseNUMA*) which could help for the NUMA-like design of our Threadripper system, ensuring that the heap memory is eagerly allocated (-*XX:+AlwaysPreTouch*), disabling adaptive heap size (*-XX:-UseAdaptiveSizePolicy*), as well as tuning the small and large heap size (*-Xms* and *-Xmx*, although indirectly through *-XX:MaxRAMPercentage* and *-XX:MinRAMPercentage*). Intuitively, we expected that, given the sheer volume of object creation in our application as found by profiling, that even small changes to GC parameters would have a significant impact. In particular, we expected that using the throughput stop-the-world garbage collector (ParallelGC) would improve performance since the default G1GC's threads are also competing for CPU time with the application threads. Our intuition has been repeatedly challenged, where in many cases increasing the *MaxGCPauseMillis* is sufficient, such that G1 may outperform ParallelGC. Our stance on this matter is that regardless of what flags we used, the performance impact was nearly identical for both the sequential and our parallel solutions. In some cases, only the sequential implementation benefited whilst the parallel was no worse off. Either way, since we are only interested in relative performance, we consider JVM tuning to be an engineering concern for specific use cases and not the focus of the evaluation of our proposed general solutions.

When running our benchmarks, we have made every attempt to ensure the consistency of the execution environment. Specifically, we tried to minimize the number of unnecessary background processes running whilst benchmarking, and that our hardware and software configurations were identical when comparing modules executed on the same system. We also ensured that the JVM parameters were the same for all invocations: of particular note, that the maximum memory available to the JVM is equal to the memory available on the system. To further increase the confidence in our results, we ran each benchmark at least 5 times and took the average of the runs as the result, investigating and removing any outliers as and when appropriate. Furthermore, our evaluation infrastructure allows for repeating the experiment multiple times within the same JVM instance, albeit without reloading the model. This is also useful for testing. From a benchmarking aspect, repeating enables us to assess the performance once the relevant model element types has been cached, since we ran all of our experiments with caching enabled. This means the first run will always be slower since there is the additional overhead of populating caches, which is performed lazily. In most cases, we found the difference to be surprisingly small. Nevertheless, we performed two repeats per invocation so that the split between an initial and "warmed up" (i.e. when caches were populated) run would be 50/50. We believe this is more representative of typical use cases and also gives us a more conservative estimate of relative performance. As previously stated, we are trying to assess the practical benefits where possible, not best-case under ideal conditions.

One crucial factor which we have not mentioned regarding our methodology is how we account for varying clock speeds when comparing speedups. This is briefly discussed by McCool et al.; however it is not widely acknowledged, especially in the MDE literature. Modern x86-64 CPUs from Intel and AMD adjust their clock speeds dynamically based on the load, voltages and temperatures; - in many cases, going above the base quoted clock speed without requiring explicit overclocking by the user (Intel calls this "Turbo Boost"). The number of logical cores in use also determines the clock speed multiplier. For example, in a single-threaded workload, the CPU's maximum "turbo" clock is higher than if all cores are used. Whilst it is possible to disable such technology from UEFI / BIOS or other means, it is not always practical to do so and leads to a contrived scenario, since the end user is not going to purposely reduce their hardware's performance. Instead of trying to counteract the effects (for example, by loading all the cores when fewer threads are used [254]), we can simply adjust the speedup figures by the ratio of the clock speeds. As a representative example, suppose that we have a quad-core CPU without simultaneous multi-threading (SMT), a base clock of 3.2 GHz, a clock speed of 3.6 GHz under single-threaded load and 3.4 GHz under full load (i.e. all cores in use). Suppose that our speedup with four threads is 3.5x over a single-threaded implementation. The "effective speedup" is then calculated as:

$3.5 * (3.6/3.4) = 3.706$

or more generally:

$$Se = \frac{ETa - STa}{ETb - STb} * (\frac{Ca}{Cb})$$

where *Se* is the effective speedup, *ETx* and *STx* are the end time and start time of program *X* respectively, *Cx* is the average clock speed when running program *X*.

Clock speeds vary a lot more in mobile and power-constrained CPUs than traditional desktop processors. Given that mobile devices such as laptops, tablets and even smartphones are much more prevalent as computing devices amongst users and developers than full-size desktop towers, this factor has arguably become much more significant in recent years. That said, we are mostly interested in the practical benefits provided by parallelisation, so as with our argument regarding JVM tuning, we do not adjust our speedup metric based on the above formula for simplicity in cases where the difference is negligible. It is for this reason that we primarily use non-mobile devices for our benchmarks, where thermals and power have a much smaller impact on performance.

Finally, we should make it clear at this stage that we do not intend to analyse (at least in any great detail) the memory consumption of our solutions. When dealing with long-running programs which allocate hundreds of GBs of memory over the span of their execution, garbage collection happens frequently, and the facilities provided by Java to accurately measure average memory consumption are inadequate for our needs. Any measurements of memory are at best a rough indication of memory used by program at runtime to run the program and at worst, meaningless. Whilst we could measure total memory allocation, this doesn't give an idea of the memory capacity required to run the program. Measuring memory consumption at runtime using a process monitor or a JVM profiler can adversely impact performance and is not easy to automate. Furthermore, since most of the memory usage in our experiments will be from the model, it would be difficult to gauge the relative memory efficiency of our developed solutions relative to the baseline oracle (status quo).

To summarise, our performance benchmarking methodology aims to assess the practical benefits of the developed artefacts as would be experienced by an end user in a typical scenario, rather than trying to fine-tune the environment for optimal performance. As previously mentioned, almost anything can have a significant impact on performance due to the complexity of Java applications. Therefore, by running our application under realistic scenarios we gain an insight into the realistic performance benefits applicable to users rather than results which can only be obtained under optimal conditions.

3.3.3.1 Test platforms

For our test platforms, we use multiple systems to ensure that our speedups are relatively consistent across different hardware and operating systems. To avoid repeating the specifications in subsequent evaluation sections, we list the detailed specifications of the relevant computers used in performance experiments here. The PCs will be referred to by the CPU model hereafter for brevity.

"Lab" – managed by IT Services at the University of York. There are many of these PCs with identical hardware and software, which is useful for testing distributed systems.

- OS: Windows 10 (v1903)
- JVM: AdoptOpenJDK 11.0.3 (HotSpot)
- CPU: Intel Core i5-8500 @ 3.00 GHz (6 cores, boosts up to 4 GHz)
- RAM: 16(2x8) GB DDR4-2133 MHz
- Storage: 500GB Samsung 860 EVO SATA3 SSD

"Laptop" – Ultrabook device, representative of the typical power-constrained computer used by most developers and professionals. Thermals play a much larger role in thin and light laptops with high clock speed processors, so this device can give us a picture of the typical real-world speedups we can expect on a laptop once accounting for the extreme variance in clock speeds brought about by the thermal design, especially during all-core boost. That said, with Linux installed there are only two CPU governors available: "*powersave*" and "*performance*". In both cases, CPU frequency varies greatly between all-core and single-core usage. We used the *performance* governor in our benchmarks, where with a single-threaded load all (logical) cores clocked in at between 2.6 and 2.8 GHz, averaging approximately 2.7 GHz. Under all-core load, this speed dropped to between 1.2 and 1.3 GHz, occasionally reaching 1.6 GHz but generally dropping below 1.5 GHz. To calculate the effective speedup, we multiply the obtained speedup by a factor of approximately 2.1 to compensate for this discrepancy. Although the precise all-core turbo frequencies and single-threaded boost clocks varied somewhat, the clock speed under single-threaded benchmarks was generally between 1.7x and 2.3x higher. Note that this is a software limitation: we used a cooling pad whilst running our benchmarks and on mains power, with no obvious signs of thermal throttling during our benchmarks. In Windows, clock speeds are substantially higher with the highest performance setting. Unfortunately, we were unable to manually fix the clock speed for benchmarks, due to limited options offered by the UEFI and operating system. We use this system to demonstrate that when calculating speedups, it is critical to take into account dynamic CPU scaling, and hope that future researchers also take note, as it is often overlooked.

- **ASUS ZenBook UX430UA**
- OS: Ubuntu 19.10 (Linux kernel 5.3.0)
- JVM: OpenJDK 11.0.5 (HotSpot)
- CPU: Intel Core i7-8550U (4 core / 8 thread, 1.8 GHz base, up to 4 GHz boost)
- RAM: 8 (2x4) GB DDR4-1867 MHz
- Storage: 256GB SATA3 SSD (SK Hynix)

"Dev" – custom-built system running the latest (at time of writing) 3rd generation Ryzen processor. This is the PC used for development and testing in the later stages of the project. It should be noted that the architecture of this CPU is more complex than traditional Intel CPUs. AMD's modular design essentially means each processor is made up of one or more "Core Complex" ("CCX") units, where each CCX consists of four cores. In the case of the 3700X, there are two CCXs. Furthermore, not all cores are of equal performance, as noted by several articles and AMD themselves [255]. Each CCX has its preferred "fastest core" which a modern operating system kernel should be aware of to target single-threaded workloads. All of this means that achieving 8x speedup in memory-intensive applications is an unrealistic target in practice. It is even less realistic to aim for 16x speedup on the Threadripper system, as that is made of four CCXs spread across two separate dies, essentially a NUMA layout. For details on the Zen microarchitecture, see Wikichip [256].

- OS: Windows 10 (v1909)
- JVM: Oracle JDK 13.0.1 (HotSpot)
- CPU: AMD Ryzen 7 3700X (8 core / 16 thread, up to 4.35 GHz)
- GPU: NVIDIA GeForce GTX 1070 Ti 8GB
- RAM: 32 (2x16) GB Corsair Vengeance LPX DDR4-3200 MHz ("Ryzen Tuned")
- Motherboard: MSI X570 A-PRO (AGESA 1.0.0.4 update)
- (Primary) Storage: 1TB Sabrent Rocket PCIe 3.0 x4 NVMe SSD

"HPC" – This is a system running the Sun Grid Engine, used in a retired high-performance compute cluster at the University of York (YARCC) [257]. We use this system briefly in a few tests to determine performance in a NUMA architecture. Whilst the hardware is old (the CPU is from 2009), it represents a typical cloud system in cloud architectures, so it would be interesting to assess the scalability of our solutions in a multi-socket system.

- OS: CentOS 7 (Linux kernel 3.10.0-862.3.2.el7.x86_64)
- JVM: OpenJDK 1.8.0_232 (HotSpot)
- CPU: 2x Intel Xeon E5520 (4 core / 8 thread, 2.27 GHz)
- RAM: 141 GB

"Workstation" – custom-built system running the top-end first-generation Zen processor. We use this system for most of our benchmarks due to its high core count and memory capacity.

- OS: Fedora 30 (Linux kernel 5.2)
- JVM: OpenJDK 11.0.5 (HotSpot)
- CPU: AMD Threadripper 1950X (16 core / 32 thread, up to 3.7 GHz)
- GPU: AMD Radeon RX 550 4GB
- RAM: 32(4x8) GB DDR4-2933 MHz (quad-channel memory configuration)
- (Primary) Storage: 500GB Samsung 960 EVO NVMe SSD

Overall, our test platforms have been selected to demonstrate the diversity of x86-64 platforms which, in the real world, are far more complex in their design than traditional processors. Classic literature and theory on parallelism and speedup do not (and couldn't have possibly, at the time) considered the myriad of variables which affect multi-threaded performance. Now we have "fast" and "slow" cores, multiple cores where some cores are on a separate die but on the same chip, simultaneous multi-threading and most intrusive of all, widely varying clock speeds under different workloads, based on advanced algorithms accounting for thermals and voltage. There exists much confusion, even amongst hardware experts, on the inner workings of how CPU clock speeds vary with sophisticated firmware features such as AMD's Precision Boost 2, Precision Boost Overdrive, Core Performance Boost, Extended Frequency Range, AutoOC and other marketing terms. For some clarification on the terminology and the performance impact of some of these features, see the Gamers Nexus article and accompanying video on the subject [258]. Of course, this is purely on the CPU side of things. Matters are more complicated when considering memory channels, memory clock speed, CAS latency, "fabric clock" and other memory timing parameters.

3.3.4 Automation and Reproducibility

Whilst the key contribution of this thesis is ultimately the approach and developed artefacts, an oft-overlooked aspect of empirical projects such as this one is the approach to evaluation. Although in principle the objectives and criteria are straightforward, the means by which they are realised concretely and compared is much more time-consuming. An important aspect of scientific research is reproducibility, and it is the duty of responsible researchers to ensure that their experiments can be reproduced by other researchers / scientists / engineers with minimal friction. Going further, it is also arguably important that the evaluation methodology is extensible so that the developed artefacts and approach can be validated independently using different inputs and/or parameters. Even with the relatively small sample of models and model management programs (scripts) used in the evaluation of this work, there are thousands of combinations of what could be considered "useful" or "meaningful" benchmark candidates due to the state explosion of parameters involved. Even with a single evaluation script, there are different model sizes to consider, different number of threads, different computers, various JVM parameters, various parameters to the model management program under test and of course, different implementations (e.g. sequential vs. parallel vs. a competing / alternative tool). Our task is to appropriately select a meaningful and sensibly sized sample of benchmarks which are suitable for evaluating the research objectives. More concretely, we must choose a representative collection of scenarios for benchmarking and, in the end, justify why our chosen samples help to accurately assess the extent to which performance is improved as a result of the kind of parallelism proposed in this thesis. Testing for performance requires a deliberate calculated selection of inputs which can be argued to be sufficient in assessing the extent of improvements relative to the baseline (status quo). In this sense, testing for performance involves similar trade-offs and thinking to testing for correctness: we cannot test with every possible input as the combinations are infinite, rather we must select an appropriate slice of scenarios, of sufficient complexity which reflect potential real-world use cases by being similar in the nature of the features they exercise, to show that the benefits are generalisable – and of course, ensuring that the semantics of the new implementation conform to expectations (in our case, correctness is defined as equivalence to the status quo).

The various possible combinations of module (engine implementation) configurations, models, scripts etc. for a given model management program are too large to create by hand. Our infrastructure comes in two parts. The first is a runtime configuration API, used both for automated tests and performance benchmarks. The second is a ~600 line Python script for generating benchmark scenarios as *sh* (for Unix) and *cmd* (for Windows) files which invoke the program through the API in the first part with a specific combination of parameters.

Firstly, we developed an extensible API for grouping together specific combinations of models, scripts, script parameters, module implementations and other configuration info, such as where to output results, the profiling information and so on, as well as a parameter to repeat the experiment a certain number of times. We refer to this as a *scenario* or "run configuration". Each configuration class is immutable, though it comes with an accompanying builder API for easy creation of scenarios in a modern, concise and declarative manner. On top of this, there is a command-line interface (based on Apache Commons CLI) which is closely tied to the builder. These run configurations and builders are extensible inline with the semantics of Epsilon, so for example the EVL run configuration extends the EOL one, just as EVL extends from EOL. There is also a base class which provides the generic infrastructure so that we can re-use it for evaluating other tools with a uniform API and to avoid duplication of concerns such as profiling.

The Python script acts as a generator for benchmark scenarios in our harness at a higher level. At the simplest level, it can be thought of as many nested *for* loops over the various combinations of the scripts, module implementations, model sizes etc. in our harness. It also contains a command-line API which allows to specify various JVM parameters and also where the files (scripts, models, metamodels, binaries / JAR files) are located, where to generate output to, where the run configurations should write their output to etc. as well as other (fixed) arguments to the run configurations – for example, the IP address of the broker used in distributed EVL. The Python program also contains utilities for grouping together scenarios and invoking them from a single batch file. This is useful for reproducibility as only a single command needs to be invoked to reproduce an entire suite of benchmarks. Furthermore the script is also adaptive to the environment with regards to the number of hardware threads: it can detect the number of cores reported by the operating system and use this to generate scenarios (applicable only to the parallel implementations of course) with varying number of threads up to the maximum in pre-defined increments.

The convenient, portable and cross-platform generation of benchmarks is only one side of the evaluation infrastructure. It is also prudent to provide an easy means to analyse the output of these benchmarks. Since there are many scenarios, and each scenario has an associated output file, and each scenario may be run multiple times, it is unproductive to manually analyse the profiling information. Our Python script uses regular expressions to automatically extract profiling information of interest from the output logs and compute useful metrics such as the average (grouping together multiple runs of the same scenario), standard deviation and, most crucially, speedup and even efficiency (speedup divided by parallelism). In the analysis part of the Python program, we take advantage of a consistent naming scheme used in our run configurations and module implementations to be able to automatically detect which scenario serves as the baseline for any other scenario to calculate relative performance (speedup). The analysis produces a CSV file containing all of the desired metrics for all output files in a given output folder (as generated by the run configurations, which were invoked by the batch (cmd/sh) files generated by the Python program). We can then open this single CSV file in a spreadsheet editor such as Microsoft Excel and perform further post-processing if needed, such as creating charts or graphs. The Python program can also generate LaTeX tables from the results metrics, which is useful for copy-pasting results into academic publications without manually creating the table structure and inputting the results.

All these utilities combined should hopefully make the research and tooling presented here easier to evaluate and encourage other researchers to perform their own experiments. They are of course open-source – details of their availability can be found in Appendix I.

3.4 Threats to Validity

In this section, we discuss potential concerns with the research approach. We have touched on many of these points already and will recap the generalisability of the thesis in Section 8.2. Since the remaining chapters follow the same approach and evaluation process, we summarise the threats to validity here for ease of reference.

We described our evaluation strategy in Section 3.3. Although we have done our best to minimise potential biases and control for factors affecting performance, there is a chance that our chosen models, programs, JVM parameters, operating systems and hardware are not representative of the "typical" case, and thus may give an overly optimistic (or pessimistic) picture of performance gains from parallelisation. This is of course a potential threat to any empirical evaluation; hence, we tried to minimise this by using a range of hardware, and benchmarking on both Linux and Windows. Regarding JVM parameters, we did not attempt to use any "exotic" parameters which would favour parallel performance, as previously discussed. Regarding our choice of models and scripts, we tried to use external sources where possible as our inspiration for writing the model management programs, and we never used any models or metamodels that we created ourselves. Furthermore, we did not even use a model generation approach, since we wanted to ensure we have non-synthetic benchmarks where possible. Unfortunately, there is significant difficulty in finding open-source complete examples of model management programs and model combinations which are computationally expensive. Since related works in optimisation focus on model-to-model transformation programs, the difficulty is further compounded since our focus is on model validation, querying and model-to-text transformations. Where applicable, we discuss the implications of our choices and justify them in the relevant evaluation sections.

Due to the lack of formal verification or other rigorous analytical methods, we cannot *guarantee* the correctness of our parallel implementations. By correctness, we mean equivalence in outcome to the sequential implementation for a given input. However, this lack of guarantee is true of any non-trivial piece of software which has not been generated or subject to formal verification methods. We have tried to maximise confidence in our approach by developing an extensive equivalence test suite as described in Sections 4.5.1 and 6.7.1, for example.

The solutions developed in this research have been implemented and evaluated within the Epsilon family of languages. The reasons for this have been explained in Section 3.2.4. However, there is still the question of how generalisable such solutions (and challenges) are beyond Epsilon, and the extent to which a similar approach can be used to parallelise other languages. Whilst it can be argued that the identification of challenges (to parallelisation) and our solutions to them have been influenced by the design of Epsilon, fundamentally the principles should generalise to other languages by virtue of having similar goals regarding the model management tasks. As previously stated, Epsilon's languages offer more features and greater flexibility than popular alternatives, which in theory means there are more issues to consider. For example, EVL (discussed in Section 4.2) was created to address some limitations in OCL [66]. Given that OCL is a functional language and has fewer language features / capabilities, there should be fewer challenges, and only a subset of our solutions needs to be considered (for example, OCL doesn't support dependencies between invariants, whereas EVL does). This does not necessarily mean that in practice it will be easier to parallelise OCL, since the implementation of e.g. Eclipse OCL is very complex and may require re-architecting. However, the core algorithms and solutions we describe should be generalisable.

Given that Epsilon's languages are interpreted, one may wonder whether the performance gains from parallelisation will generalise to compiled languages. It may be argued that the gains in performance stem largely from improving the performance of interpreted code; that the proposed algorithms don't scale as well when the interpreter overhead is taken out of the equation. This is difficult to verify without implementing our solutions on top of a compiled language (or by developing a compiler for Epsilon's languages, which is well beyond the scope of this project). Our solutions target fundamental algorithms at the engine level, so our parallel approach can still be applied to compiled languages. We compare the relative performance of our parallel approach to the equivalent compiled Java code in Section 6.7.2. The experiments there show that the scalability of interpreted and compiled with regards to parallelisation are very similar.

We cannot be certain that our solutions generalise for all modelling technologies. Our solution relies on model access for (random) individual elements being relatively fast i.e. O(1) in time complexity. For simplicity, we have used in-memory EMF models for our performance benchmarks. However, for much larger models which cannot be stored in memory, model access may be significantly slower. Thus, a degree of in-memory caching may be necessary for optimal performance. Elaborate partial caching strategies of large models may also impact both correctness and performance gains from parallelism if the underlying model drivers are not thread-safe. That said, the design of Epsilon purposely separates model access and its languages through the EMC layer. Hence, it is the responsibility of EMC drivers to deal with multi-threaded access. Since these drivers implement interfaces which delegate to some underlying API exposed by the modelling technology (e.g. JDBC for relational databases), the responsibility also lies with the underlying storage (e.g. the database). Given that our parallel infrastructure is focused on languages and abstracts away from the model persistence layer, we can be confident in the generalisability of our solution in this regard (i.e. no changes needed at the language / engine level). With regards to performance optimisation, slow access to model elements may mean that the optimal number of threads should be increased due to I/O bottlenecks. Hence, this parameter is configurable by users in our solutions.

4 Parallel Model Validation

This chapter describes the design and implementation of a parallel model validation tool. This work builds on the prototype of Smith (2015) [10]; albeit with a different (i.e. heavily revised / overhauled) implementation. As discussed in the Literature Review chapter, the work of Smith (2015) identified notable challenges with applying parallelism to the Epsilon Validation Language (EVL), however due to the nature of the project's scope and time (a Masters' project), there remained some unresolved challenges and uncertainty regarding the correctness and performance. Whilst the work of Smith served as a useful starting point in understanding the challenges with parallelising EVL, the developed implementation was very much a prototype, and was not as well optimised. There was also a lack of a thorough test suite, and the author recognised some outstanding issues (bugs) with the implementation. The EVL implementation has also evolved during the course of this research project with further optimisations applicable to both the parallel and sequential execution engines. The work presented in this chapter represents a significant evolution of Smith's work, with a different parallelisation approach, more elaborate evaluation and a production-ready implementation with various optimisations. The main commonality between the two works is the use of serial thread confinement for internal engine structures, however even there the implementation is substantially different.

In this chapter, we explore the fundamental technical challenges (and solutions) with retrofitting concurrency support to a complex model validation language and devise a scalable parallelisation solution. Section 4.1 motivates the need for model validation conceptually. Section 4.2 introduces the Epsilon Validation Language (EVL), which will be used to demonstrate our contributions. Section 4.3 outlines the challenges with parallelising EVL and the solutions we adopted to address these. Section 4.4 describes several scalable parallel decomposition strategies of the execution algorithm. Section 4.5 evaluates our implementation, describing how it was tested for correctness along with some performance benchmarks. Section 4.6 concludes the chapter.

4.1 Model Validation

We already saw a brief overview of model validation in Chapter 2. The purpose of model validation is to perform consistency checks on models by enforcing *invariants* (also known as *constraints*). The primary motivation for model validation as a distinct model management task is traditionally routed in the deficiencies of popular metamodeling technologies like UML or Ecore. Whilst it's possible to enforce multiplicities in relationships directly from the metamodel, and even ranges of values for text using enumerations or bounds for integers, more advanced constraints need to be expressed in a different way. The most common solution is to use a textual language to programmatically express the invariants. The Object Constraint Language (OCL) [46] is the *de-facto* standard for this. A typical use case would be to embed OCL expressions on types in a UML class diagram. An invariant expressed in the context of a type in the metamodel can navigate the properties and references of an instance of the type using the "*self*" keyword / variable. Each invariant is expected to return a Boolean expression, indicating whether the given element is valid (i.e. satisfies the invariant) or not.

There are two ways to express invariants with respect to the metamodel. For most use cases, invariants are effectively part of the metamodel, and so it makes sense to embed the invariants directly in the metamodel. Semantically this implies that any model which does not satisfy all of the invariants does not conform to the metamodel. The *OCLinEcore* metamodeling language [259] provides a textual syntax allowing for the metamodel to be expressed along with invariants in a single document. This is compatible with regular Ecore metamodels, since the OCL invariants (and any helper operations) are encoded as string properties on the metamodel. In some cases however, the metamodel is always “valid” without any constraints, so the invariants depend on the context in which the model(s) are used. This is because the business logic may vary between organisations or use cases for a given (meta)model. In such cases, it would make sense to express invariants separately from the metamodel. The *CompleteOCL* language [260] allows for invariants to be expressed in a standalone document. Furthermore, it is even possible to express invariants for multiple metamodels by importing them and compartmentalising them into *packages*.

Conceptually, model validation can be viewed as either a query or transformation. One could view a model validation program as a series of filtering 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. Bézivin and Jouault (2006) [261] demonstrate how a model transformation language can be used to validate models and produce diagnostics. From either perspective, each invariant in a validation program is a predicate for which each applicable model element is tested. The output of a model validation program is a set of diagnostics for the unsatisfied constraints, with each diagnostic containing the name of the invariant, the elements which did not satisfy the predicate and an optional message detailing the reason for the failure.

4.2 Epsilon Validation Language

We have already introduced Epsilon and justified the rationale for using it as an implementation testbed in the previous chapter. In this section, we introduce the Epsilon Validation Language (EVL) in more detail.

In “*On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages*”, Kolovos, Paige and Polack (2009) [66] motivate the need for more advanced model validation capabilities and incorporate these features into the Epsilon Validation Language. Notable omissions from the *de-facto* Object Constraint Language (OCL) include the following:

- Poor user feedback (no customisable messages for failed invariants)
- No support for classifying invariant priorities (e.g. “warnings” vs “errors”)
- No support for dependencies between invariants
- No support for lazy invocation of invariants
- No support for rectifying failed invariants programmatically
- No support for pre and post-processing
- No support for imperative (i.e. non-functional) code

4.2.1 **Program Structure**

EVL programs are organised into *modules*; where each file contains one EVL module (an example is shown in the next section). The module can import other modules – for example, an EOL module – so that the operations (helper methods / functions) from the imported module(s) are also available to the current module.

An EVL module can declare any number of operations in the context of a type (or no type if it is a “context-less” operation). This is similar to programming languages like Kotlin, where additional functionality can be added to existing types without having to create a new type and extend through traditional inheritance. Operations which declare a type access the instance of the type through *self* (equivalent to “this” in other object-oriented languages). Furthermore, operations without parameters and a return result can be cached, which internally creates an association between the operation, the instance it was invoked on and the result. Subsequent invocations of the operation for the same instance will then return the cached result, avoiding any recomputation.

In all of Epsilon’s rule-based languages (internally known as ERL), prior to execution of rules (in the case of model validation, the constraints), users can run arbitrary imperative code, even invoking Java code (since EVL is interpreted, implemented in Java and uses reflection), in the *pre* block. This is useful for setting up the validation environment programmatically – for instance, writing to an output stream, declaring global variables which can be used by the validation constraints, pre-computing values and caching them to avoid unnecessary recomputations etc. Once all validation constraints have been checked, the *post* block is executed. This is again an arbitrary block of code which can be used to perform further processing on the results (e.g. writing to an output stream) or interact with the variables defined in the *pre* block.

The main body of an EVL module consists of invariants (declared using the *constraint* keyword), which are grouped by the model element types they validate (declared using the *context* keyword). A *ConstraintContext* declares a model element type and contains zero or more constraints. A ConstraintContext can also have a *guard* block, which filters the elements that are checked by the constraints. The guard block (or expression) returns a Boolean indicating whether the current element (*self*) should be validated or not. The absence of a *guard* implies that the invariant is applicable for all elements.

Each constraint may also have a *guard* block, which filters elements to be checked by the current constraint. The only mandatory construct in a constraint is the *check* block / expression, which returns a Boolean indicating whether the current element satisfies the constraint. If this block / expression returns false and a *message* block / expression is present, the returned value from the message block / expression is used as the diagnostic. All instances for which the check block returns false are added to the program results, which consist of a set of diagnostics containing [*model element*, *constraint*, *message*] tuples. A constraint may also declare any number of *fix* blocks, allowing the user to programmatically rectify the model in the event of a constraint being unsatisfied. The block allows access to the offending element and to make changes to the model as needed. However this step is optional and only performed after the program (including the *post* block) has been executed. As this is a post-processing feature, we will not consider fixes. However, it is important to ensure that the data required to perform fixes is made available.

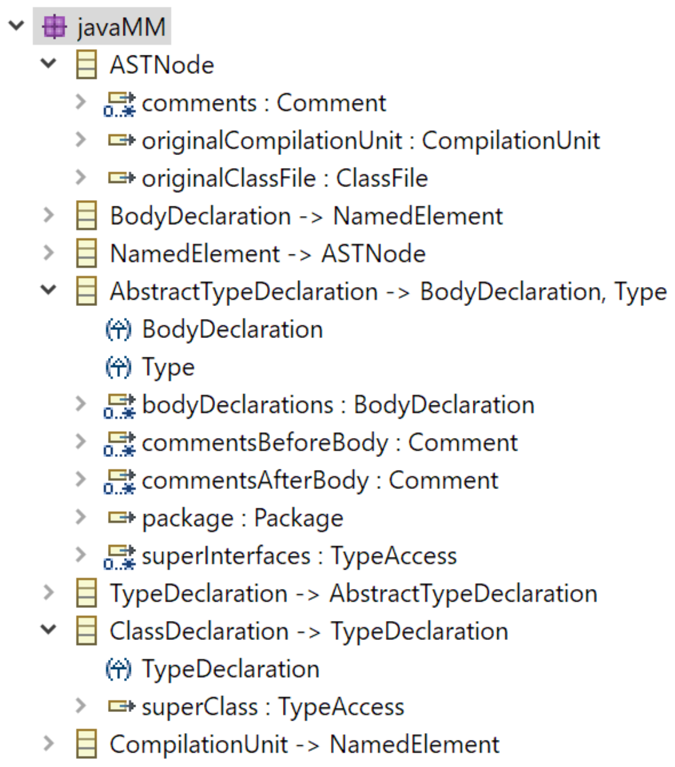
A constraint may be declared as *lazy* via an annotation. A lazy constraint is ignored by the execution engine unless it is explicitly invoked as the target of a dependency. Constraints may invoke each other via the *satisfies* operation which has two variants: *satisfiesOne* and *satisfiesAll*. Both operations take as input one or more constraint identifiers and return a Boolean by evaluating the named constraints for the given element, with the former operation requiring any of the constraints to be satisfied whilst the latter requires all of the specified constraints to be satisfied.

A constraint may be declared outside of a ConstraintContext, as there is a *GlobalConstraintContext*, which case it will be executed once by default (or never if declared as lazy), unless it is invoked as the target of a dependency by other constraints. It is worth noting that contextless constraints cannot access the *self* variable (as it would be meaningless) unless they are invoked by other constraints, in which case *self* will be set to whatever object the *satisfies* operation was invoked on (which may be a model element from a different ConstraintContext). Semantically, a lazy constraint declared in the global context is similar to an operation except that it is invoked via the *satisfies* operation, and its failure will be shown in the results (i.e. set of unsatisfied constraints).

4.2.2 Example EVL Module

```
@cached
operation AbstractTypeDeclaration getPublicMethods() : Collection {
  return self.bodyDeclarations.select(bd |
    bd.isKindOf(MethodDeclaration) and bd.modifier.isDefined() and
    bd.modifier.visibility == VisibilityKind#public
  );
}

context ClassDeclaration {
  constraint hasEquals {
    guard : self.satisfies("hasHashCode")
    check : self.getPublicMethods().exists(method |
      method.name == "equals" and
      method.parameters.size() == 1 and
      method.parameters.first().type.type.name == "Object" and
      method.returnType.type.isTypeOf(PrimitiveTypeBoolean)
    )
    message : "Class "+self.name+" violates hc/eq contract!"
  }

  @lazy
  constraint hasHashCode {
    check : self.getPublicMethods().exists(method |
      method.name == "hashCode" and method.parameters.isEmpty() and
      method.returnType.type.isTypeOf(PrimitiveTypeInt)
    )
  }
}
```

Listing 4.2.1 *hashCode* and *equals* contract example EVL program



```
javaMM
  ASTNode
    comments : Comment
    originalCompilationUnit : CompilationUnit
    originalClassFile : ClassFile
  BodyDeclaration -> NamedElement
  NamedElement -> ASTNode
  AbstractTypeDeclaration -> BodyDeclaration, Type
    BodyDeclaration
    Type
    bodyDeclarations : BodyDeclaration
    commentsBeforeBody : Comment
    commentsAfterBody : Comment
    package : Package
    superInterfaces : TypeAccess
  TypeDeclaration -> AbstractTypeDeclaration
  ClassDeclaration -> TypeDeclaration
    TypeDeclaration
    superClass : TypeAccess
  CompilationUnit -> NamedElement
```

Figure 4.2.1 Small subset of MoDisco Java metamodel

Suppose that we have a model of a Java program corresponding to the MoDisco Java metamodel [262] (the relevant subset is shown in Figure 4.2.1), and we want to ensure that every class overrides the *equals* method according to the contract "Equal objects must have the same hash code, but unequal objects do not necessarily have different hash codes". Listing 4.2.1 shows how this could be implemented in EVL. Note that by declaring the *hasHashCode* constraint as lazy, we only check it once per *ClassDeclaration* as a pre-condition for the *hasEquals*. If *hasHashCode* fails for the current element under consideration (referred to as *self*), then we avoid executing the *hasEquals* check expression for the current class. This means a class may fail to satisfy either *hasHashCode* or *hasEquals*, but not both. Therefore, our results will never contain the same class more than once. Also note that by declaring the *getMethods* operation as cached, we avoid re-evaluating it for a given model element, so that if a class satisfies *hasHashCode*, we do not need to find all of its methods again in the check block of *hasEquals*. For an alternative implementation of this program which demonstrates EVL's *pre* and *post* blocks, refer to Listing 4.3.2.

4.2.3 Execution Algorithm

The execution algorithm for sequential EVL is given in Listing 4.2.2.

```
execute module's pre block;
foreach ConstraintContext cc in EvlModule:
  foreach element in all matching the type described by cc:
    if not cc is lazy and cc's guard is satisfied for element:
      foreach Constraint c in cc's constraints:
        if not c in ConstraintTrace:
          if not c is lazy and c's guard is satisfied for element:
            satisfied = execute c's check block;
            (optional) add c and element to ConstraintTrace;
            if not satisfied:
              add c and element to set of UnsatisfiedConstraints;
execute module's post block;
```

Listing 4.2.2 Simplified sequential EVL execution algorithm

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

4.3 Thread-Safety: Challenges and Solutions

Our observation from Listing 4.2.2 is that each iteration of the three loops need not be performed sequentially, since there is no dependency between them (except for occasional constraint dependencies, discussed below). Fixes in EVL are performed optionally after validation and initiated by the user, so the model is only queried, never written to (we do not consider parallel execution of fixes). In principle, this makes the task of executing read-only operations (check blocks) within a loop inherently parallelisable. However, in practice this is complicated by a number of factors, to which we now turn.

4.3.1 Accessing Data Structures

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

A key challenge with retro-fitting concurrency into an existing program (such as the EVL interpreter) is handling of access to data structures. When multiple threads have shared access to the same mutable data, the non-deterministic nature of parallel execution can lead to inconsistent states (Readers-Writers problem). There are generally three solutions to this problem: Not sharing such data between threads, making the data immutable; or using synchronization whenever accessing the data [112]. Unfortunately, in many cases synchronization is adopted, which, from a programmer's perspective, is a convenient option if the language makes this trivial to achieve. This has a major impact on performance not only because a single thread can execute synchronized regions at a time, but also the overhead introduced by synchronization mechanisms. This is especially problematic for performance-critical data structures which are frequently accessed.

Before discussing the details of internal data structures, it may be useful for the observant reader to know that almost all of Epsilon's execution data structures are encapsulated within a single object. The core engine structures (such as declared variables, or *FrameStack*) are common to all languages, so the bulk of these is in an object called *EolContext* (and its interface, *IEolContext*). For EVL, additional data structures such as the set of unsatisfied constraints is stored in *EvlContext* (with access through *IEvlContext* interface). Code samples in future sections will contain references to this object. In short, the context is, as the name implies, the execution context holding all program state.

It is useful to classify access to internal data structures during execution of the script into one of three categories: read only, write only, and read/write. The latter category can be further classified into *read/write local* and *read/write global*. We discuss each of these in turn.

The first category is inherently thread-safe since it is immutable. Everything is assumed to be (and naturally falls into) this category by default, at least implicitly. Such structures include the script itself, the model, the constraints and constraint contexts. It should be noted that by "immutable", we mean anything that doesn't change during concurrent execution. In other words, even if a data structure or environment variable changes in the *pre* block, for example, then so long as this doesn't change until the *post* block we can safely treat it as immutable for the purposes of parallelisation. Put another way, we are only interested in changes which occur between (but excluding) the first and last lines in the algorithm shown in Listing 4.2.2, i.e. the main execution logic.

4.3.1.1 Unsatisfied Constraints

The second category will never be queried during execution. An effective solution for this is to create a per-thread data structure and then merge all of the thread-local data structures once execution has completed. Since no thread will ever attempt to read from the structure, we do not need to merge or synchronize access during execution. The set of unsatisfied constraints (i.e. the results data structure) belongs in this category. Note that this could also be replaced by a data structure in which writes are cheap but reads are expensive, though such a requirement is uncommon so a custom implementation of a Set would have to be hand-coded. Note also that since the structure will never be queried during execution, it need not be a Set. Writing to a HashSet can be expensive as each UnsatisfiedConstraint needs to be hashed and placed in the appropriate bucket, as well as checking for duplicates. We need a data structure that is very good for insertion of an unknown number of elements. A LinkedList is perhaps ideal since it is not array-based, so no restructuring or copies are needed, and also no need to check for duplicates or calculate the hash code. A shared data structure would have a lower memory footprint and be easier to manage from the application developer's perspective, however performance may be worse. By contrast, a thread-local solution requires additional book-keeping since the returned value when the data structured is accessed via a method – e.g. *getUnsatisfiedConstraints()* – would need to return a different value based on the calling thread. However since each thread has its own copy, no thread-safety issues can occur and there is no performance penalty for writes compared to the sequential case as no synchronization is needed.

A notable performance cost comes when merging all thread-local structures, because a Set requires all element to be unique. Depending on the merging algorithm and the internal structure of the set, this cost could outweigh the benefits of a shared data structure, however writing to a shared structure broadly imposes a similar cost. Either solution is plausible, however our experiments with both revealed the thread-local one to be superior when there are many unsatisfied constraints. Another observation is that a shared structure needs to grow more quickly whilst dealing with contended locks for writes, whereas a thread-local structure spreads the unsatisfied constraints, so there should be fewer resizing operations and, more crucially, no synchronization overhead. Another rationale for thread-local LinkedLists is that of dependencies. If the ConstraintTrace is disabled or a constraint that is depended on is executed simultaneous by multiple threads, there will be duplicate instances of the UnsatisfiedConstraint in different thread-locals, so even if thread-local structures were a Set, duplicates would still need to be resolved. Using thread-local LinkedList gives more consistent and predictable performance, leaving the merging and deduplication of results to the end. Performing this task throughout execution can be costly as the JVM is less able to optimise the code and also due to potential cache misses.

We have established that we need a collection which is optimal for insertion of an known number of elements during execution. This collection may be thread-local or shared, since it is not queried. It also need not be a Set during execution, only once execution is complete. Whilst using thread-local LinkedLists and merging them into a HashSet at the end is a fairly solid solution, we can do even better. Every time an UnsatisfiedConstraint is created, it needs to be added to the collection. Access to this collection comes from the execution context for each invocation of *`Constraint.check(…)`*. Each call to *`getUnsatisfiedConstraints()`* requires checking whether we are in parallel execution, and if so returning the thread-local value. Obtaining the thread-local involves an entry lookup into Java's (built-in) highly optimized *ThreadLocalMap* (see section 4.3.5 for more on this). The cost of this is clearly higher than simply returning a shared (not thread-local) field. Provided that we can write to a collection without blocking (synchronization), a shared collection would reduce the overall cost both during execution and after when converting to a Set. We therefore use a shared custom collection based on *ConcurrentLinkedQueue* [263] which is a non-blocking, lock-free queue (FIFO) structure based on linked nodes – similar to LinkedList but more memory efficient since it is singly linked (each node has a reference to its successor, not predecessor).

4.3.1.2 Caches and Internals

The third category is unsurprisingly the most complex to deal with. There are two approaches for dealing with this. The first approach applies to data structures where full, consistent visibility is required during execution. We refer to such structures in our categorisation as *read/write global*. In these cases, shared data between threads is inevitably required, either for correctness or performance. This category applies only to caches in the case of EVL.

One notable cache is the *ConstraintTrace*, which keeps track of the executed constraint-element pairs and their results. Note that this is different from the set of unsatisfied constraints, which are the results of the EVL program. The constraint trace keeps track of all executed constraint-element pairs, not just the failures. Therefore, this structure is written to every time a constraint-element pair is executed. Internally, this is modelled as a *Set<ConstraintTraceItem>*, where *ConstraintTraceItem* is a tuple of the constraint, model element and result (i.e. whether the constraint was satisfied for the element). This collection can be replaced with a *KeySetView* derived from ConcurrentHashMap [264], a high-performance thread-safe hash map which is lock-free (i.e. there is no synchronization) for reads and locking for writes is of high granularity. However, as we shall see in section 4.3.4, an optimisation can be made so that the ConstraintTrace is rarely used.

The other caches reside at the model level. Most implementations of EMC (Epsilon Model Connectivity) extend *CachedModel*, which provides a basic capability to lazily cache results when calling *getAllOfKind(String)* or *getAllOfType(String)*. These methods take as input the name of a model element type, where the former returns the type and all its sub-types and the latter returns only instances of the specified type. The *getAllOfKind* method is called in the execution algorithm for every ConstraintContext to obtain the model elements which will be validated (i.e. substituted as the *self* variable in constraints), as shown in line 3 of Listing 4.2.2.The reason this is not thread-safe is because the caches are populated lazily to avoid unnecessary upfront temporal and memory costs of caching the entire model regardless of whether it will be used. Thus, in most cases, every iteration of the outer-most loop (line 2) in Listing 4.2.2 will require a write to the cache. Furthermore, it is possible that the user navigates references of a model element in the *guard* or *check* blocks, and if the element navigated to has not been cached then another write will occur.

This cache is implemented as a custom multimap with the type/kind identifier (which could be any Object, but for simplicity let's assume it's a String – the exact type is not important and doesn't affect the solution) as key, and its instances as the value. The multimap is internally modelled as a *Map<K, Collection<V>>*, so we can use a ConcurrentHashMap as the underlying storage. So, when *getAllOfKind* is called, the cache is checked for the given key. If the key is present, the cached value (i.e. the collection associated with the key) is returned. Otherwise the *getAllOfKindFromModel* method is called, which retrieves all values of the specified kind from the underlying model resource. The returned values are then associated with the key in the cache. However these values need to be wrapped in a mutable thread-safe collection (if they are not already), because other operations may modify this collection. For example, creating or deleting elements (which are operations supported by EMC) requires the cache to be updated. When creating new elements for example, the individual element needs to be added to the (possibly existing) collection of cached types. Therefore, the values of each entry in the underlying map (i.e. the *Collection<V>*) also need to be thread-safe.

We use a modified *ConcurrentLinkedQueue* for this purpose (shown in Appendix II), which is suitable for frequent accesses (both reads and writes) since there is no synchronisation and, because it is based on linked nodes rather than a fixed-size array, structural modifications are relatively inexpensive. The main disadvantage is that determining the number of elements in the collection is not a constant-time operation, however the size is typically not queried frequently so such a trade-off is justifiable. Furthermore, we cannot use this data structure as-is due to some of its inherent shortcomings. Firstly, *null* is not supported, so for operations which modify the collection a substitution takes place if *null* is returned or put into the collection with a static object. Secondly, a notorious issue with this data structure is the *size()* method, which loops through all linked nodes to count the number of elements. However, due to the nature of the implementation this can lead to infinite looping – which we encountered and discovered during experiments – so a solution is clearly needed. For a detailed overview of this issue, see Java Specialists Issue 261 [265].

One possibility is to manually implement a counter so that additions and removals increment and decrement the size count respectively. Another solution is to synchronize calls to the method and any other method which structurally modifies the collection. Since this collection will in practice not be written to very often but queried frequently, we chose to adopt the size caching approach. Note that iterations over this collection are thread-safe ("weakly consistent") so there is no cost to querying this cache; which is by far the most common operation on this cache, especially in read-only model management tasks. The *size()* method is clearly useful, so it must be correct and performant, yet the documentation admits this is not the case. Our workaround ensures both correctness and consistent performance.

Another concern with the model cache is ensuring that multiple threads don't try to duplicate the (potentially expensive) process of populating the caches. Although for our use case the model is read-only, the caches are lazily populated to save time and memory. With multiple execution threads, request for values of a certain type or contents from the model may result in a race condition of threads populating the cache, which is wasteful. We would much rather have threads wait for one thread to populate the cache and then re-check the cache for the values. To illustrate, consider Listing 4.3.1, which shows how this is achieved in practice.

```
Collection<ModelElementType> values = null;
final Multimap<Object, ModelElementType> cache = isKind ?
      kindCache : typeCache;
final Object key = getCacheKeyForType(modelElementType);

if ((values = cache.getMutable(key)) == null) synchronized (this) {
   // Could have changed while we were waiting on the lock
   if (!isConcurrent() || (values = cache.getMutable(key)) == null) {
      values = wrap(isKind ?
         getAllOfKindFromModel(modelElementType) :
         getAllOfTypeFromModel(modelElementType)
      );
      cache.putAll(key, values);
   }
}
return values;
```

Listing 4.3.1 Code extract from code querying cache for all elements of a given type

There are two other caches of note. One is the cache used for operations when the @cached annotation is applied, as shown in line 1 of Listing 4.2.1. As a reminder, an operation which is declared as cached must have a context type declared and no parameters. On each invocation for a given value (i.e. the object which will be substituted as *self*), if the operation was previously called with the same object, then the cached result is returned, otherwise the result is calculated, added to the cache and returned. This cache is modelled on a per-operation basis as a *Map<Object, Object>*, where the key is the invoked element (i.e. the *self*) and the value is the result. To avoid unnecessary memory usage, this cache uses a *WeakHashMap* so that the references can be cleared if memory consumption exceeds the allocated amount. This makes sense since caching is only an optimisation and not critical to correctness. However, we cannot simply substitute a ConcurrentHashMap since it uses strong references. Instead we wrap the WeakHashMap into a synchronized one, so that all of the methods are synchronized. Thus, the cache can only be accessed by one thread at a time. However this shouldn't have a noticeable impact on performance if used correctly. The rationale for using cached operations is to prevent frequent recomputations. If the computation is expensive, the benefits of caching outweigh the cost of recomputation.

The other cache is for extended properties, which allow individual objects to have arbitrary properties associated with them. Internally this is modelled as a *Map<Object, Map<String, Object>*, where the key is the object and the value is its extended properties. The properties themselves are a mapping from the property name to property value. Since these properties can be created, accessed or modified at any time for any object, the safest solution is to synchronize access to the underlying map. The extended properties features is mostly for convenience and shouldn't be used frequently in performance-critical sections of the code. Given its specialist / niche use cases and the unpredictable ad-hoc nature, it is difficult to justify concerns for further optimisations.

Lastly, there are other "live" data structures in the engine which are a core part of the execution engine, owing to the fact that Epsilon's languages are interpreted. Despite being accessed frequently for both reads and writes, these data structures do not need to be globally visible or even persistent throughout program execution. In our categorisation these are *read/write local* structures, with three notable items: *FrameStack*, *ExecutorFactory* and *OperationContributor*s, which are discussed in turn in sections 4.3.2, 4.3.3 and 4.3.4 respectively.

4.3.1.3 Collections in the *pre* block

Even once we have taken care of engine internals, it is still possible for the user to shoot themselves in the foot if they mutate global state during parallel execution. A common way for this to occur is if the user writes to a collection declared globally. The *pre* block in Epsilon's rule-based languages is a convenient way to set up global state prior to executing rules and may contain arbitrary EOL code. Variables declared in the *pre* block are visible throughout the lifetime of the program. A common use case is to declare collections which may be written to or queried later on. Whilst in many cases the collections are read-only, in more advanced cases the user may want to perform additional post-processing in the *post* block and may manually populate a collection declared in the *pre* throughout program execution. To prevent invalid states (e.g. *ConcurrentModificationException*), one solution is to provide concurrent collections which are thread-safe for writing to.

Whilst one possible solution would be to modify existing collections to be thread-safe, this would have implications on backwards compatibility and reduce performance for the vast majority of use cases. Instead we provide three high-performance thread-safe types where synchronization is avoided where possible. For *ConcurrentMap* we use ConcurrentHashMap as the delegate, and so it is trivial to add a *ConcurrentSet* based on ConcurrentHashMap's *KeySetView*, mapping each key to a static Boolean value. The only downside to this (and hence incompatibility) is that null keys are not supported. However null values are extremely uncommon in such cases (since the keys must be unique), so we do not attempt to rectify this. In cases where a non-unique collection is required and null values should ideally be allowed, we re-use our custom ConcurrentLinkedQueue implementation from II, which supports null values and fixes the size issue as described in the previous section. We call this type *ConcurrentBag* – a regular *Bag* in Epsilon is specified as an unordered, unsorted, non-unique collection, with the underlying implementation being an ArrayList. Our custom queue implementation has the benefit of preserving ordering which, although not required, makes it more compatible with existing programs which rely on Bag types being ordered.

For a motivating example, consider Listing 4.3.2 which modifies our previous *hashCode* and *equals* contract program to manually record the offending *MethodDeclaration* instances for which an *equals* exists without a *hashCode*, and vice-versa. Here we move the logic of finding the methods into operations and use imperative code in the *check* block.

```
operation AbstractTypeDeclaration getEqualsMethod() : MethodDeclaration {
  return self.getPublicMethods().selectOne(method |
      method.name == "equals" and
      method.parameters.size() == 1 and
      method.parameters.first().type.type.name == "Object" and
      method.returnType.type.isTypeOf(PrimitiveTypeBoolean));
}

operation AbstractTypeDeclaration getHashCodeMethod() : MethodDeclaration {
  return self.getPublicMethods().selectOne(method |
      method.name == "hashCode" and
      method.parameters.isEmpty() and
      method.returnType.type.isTypeOf(PrimitiveTypeInt));
}

pre {
  var loneHCs = new ConcurrentSet;
  var loneEQs = new ConcurrentSet;
}

context ClassDeclaration {
  constraint hcAndEqContract {
    check {
      var hc = self.getHashCodeMethod();
      var eq = self.getEqualsMethod();
      var hasHC = hc.isDefined();
      var hasEQ = eq.isDefined();
      if (hasEQ and not hasHC) {
        loneEQs.add(eq);
      }
      if (hasHC and not hasEQ) {
        loneHCs.add(hc);
        return false;
      }
      else return true;
    }
  }
}

post {
  ("# of hashCode without equals: "+loneHCs.size()).println();
  ("# of equals without hashCode: "+loneEQs.size()).println();
}
```

Listing 4.3.2 Imperative EVL example of *hashCode* and *equals* contract adapted from Listing 4.2.1

4.3.2 Handling of Properties and Variables Scope

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

Our solution is to use a thread-local structure (serial thread confinement) with base delegation. The idea is that each thread has its own frame stack (which is only accessible from that thread) so that whenever the *getFrameStack()* method is called, we return the frame stack associated with the calling thread. Each thread-local frame stack also has a reference to the main thread's frame stack. Whenever a variable lookup is performed, we first check the thread-local stack and if it is not present, we then look in the base. Once parallel execution has finished (i.e. all constraint and element pairs have been checked), we merge the thread-local results back into the base frame stack (i.e. that of the main thread). Depending on the execution algorithm, the main thread may also need to write to the frame stack during execution, so we make the base structure thread-safe by using an appropriate collection. In all cases, this is either a *ConcurrentLinkedDeque* (for frame stack and execution trace); a lock-free double-ended queue structure (which can also work as a stack) where writes are based on atomic compare-and-swap operations or *ConcurrentHashMap*. We found that this approach eliminated many concurrency issues and is sufficient for supporting EVL's imperative features without introducing a major performance bottleneck due to excessive synchronization.

4.3.3 Traceability, Profiling and Exception Handling

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

By using a thread-local execution controller, each thread is able to keep track of its own execution trace so that when an error occurs, the cause can be identified in a similar manner to sequential execution. Exception handling is performed by overriding the "*afterExecute*" hook of ThreadPoolExecutor. This method is called after completion of each job, and can be used to perform further tasks, including exception handling. If an error occurred in the job, this will be available to the method as a parameter. Since the exception is wrapped, we use a recursive algorithm to find the original EOL exception and cache the EOL stack trace in the message. The exception is propagated to the main thread (which is waiting for completion), which can report the cached trace to the user. The reason this caching is required is that each thread's execution controller is tied to the lifetime of the thread, so it disappears once parallel execution ends.

The execution controller can be used for any other purposes as well to perform pre and post-processing tasks when executing an EOL abstract syntax tree (AST). A notable use case is profiling. In EVL for example, we can set up a custom listener such that the "beforeExecute" hook records the start time, and "afterExecute" the end time, for every Constraint invocation. Since each Constraint is executed many times (once per applicable element), we record a cumulative sum for each Constraint which can then be reported to the user to determine which are the most computationally expensive constraints. With multiple threads of execution, these before and after hooks would be called simultaneously, so we need to record execution times on a per-thread basis. Once parallel execution ends, we merge these execution times together. Note that the reported total execution time will be significantly higher than the "wall clock" execution time, scaling linearly with the number of threads, since each thread works independently of others. Thus the reported time needs to be divided by the number of threads to be meaningful. Nevertheless, since this applies to all Constraints, it is still a useful measure of relative computational expense, reported as CPU time rather than wall clock time.

4.3.4 **Operation Calls**

The extensible nature of Epsilon means users (and tool developers) can introduce new functionality through operations on any existing types. However there is an implicit inheritance hierarchy within the type system. For example, all types are "extensions" of *Any* (Epsilon's analogous type to java.lang.Object in Java), and the *OrderedSet* type is a subtype of *Set*, which is a subtype of *Collection* and so on. When operations in the context of a type of invoked, the engine resolves the operations based on the most specific type. The idea of an *OperationContributor* is to handle the resolution of operations for individual objects based on their type. Since Epsilon is interpreted and implemented in Java, operations are implemented as Java methods, which are then resolved through reflection. The *OperationContributorRegistry* holds a cache of all OperationContributors which essentially define built-in operations for different types. When invoking an operation on an object, the appropriate OperationContributor is resolved and a binding between the object and the method (operation) is made internally. Each OperationContributor also caches methods lazily to avoid the reflection overhead on each invocation. Furthermore, every time an operation is invoked that uses an OperationContributor's operations, the specific object it is invoked on is set as a field on the contributor. In short, the way Epsilon implements operations prohibits operations from being invoked simultaneously.

It is worth remembering that Epsilon is implemented in Java, and one of its key features is that its execution semantics are more aligned with Java than OCL. This close relationship also means EOL code can invoke arbitrary Java methods. EOL relies heavily on Java's reflection API, allowing objects in EOL programs to make use of functionality already available in Java, rather than reimplementing it or placing unnecessary restrictions on user code. The complex nature of operation resolution (see OperationCallExpression#execute [266]) and dispatch necessitates these intermediate structures.

The OperationContributorRegistry is one example of a low-level internal engine structure which is not thread-safe but does not need to share state. Therefore, it can be made thread-local similar to the FrameStack. With each thread having its own copy of the OperationContributorRegistry, setting the target object and populating the cache is a concern of the executing thread only. This solution is viable because the caller and consumer of an operation call reside in the same thread, so each thread can handle operations independently without affecting the correctness or global state of the program. Note however that this only applies to the process of invoking operations (and properties) on objects. The body of an operation may contain arbitrary imperative code which may have side effects. For example, an operation may mutate the value of a global variable declared outside of the operation or change properties of its parameters.

4.3.5 **Optimising access to Thread-Locals**

Our solution for making the data structures used during execution thread-safe appears to be efficient; after all, how can one improve upon serial thread confinement? One would assume that such a solution would be no worse in terms of performance than the sequential case. Indeed our expectation is that serial thread confinement of data structures is essentially equivalent to the sequential case as far as performance is concerned since each data structure is exclusively accessed and manipulated by the thread that created it; except for when/if execution data needs to be merged into the main thread's structure. The cost of this is also negligible since it happens infrequently. However, when profiling the execution time of sequential EVL vs. our parallel implementation with a single thread, we found this solution to be a significant bottleneck. Intuitively one would expect the bottleneck to be caused by the parallel decomposition of jobs and mapping these to threads. Our tests showed that this overhead is comparatively negligible (at least relative to our expectations). The performance cost of ThreadLocals comes from the extremely high frequency in which they are accessed, as for example discussed in section 4.3.1.1.

In Epsilon, the *execution context* is the main data structure which is passed around the interpreter (AST). Every statement, expression, operation call etc. takes as input the execution context, which contains references to the FrameStack, ExecutorFactory, OperationContributorRegistry amongst other things. All interpreted execution (i.e. any Epsilon AST) goes through the ExecutorFactory, and any time a variable is declared (even implicitly, such as when entering a new scope through an operation call) the FrameStack needs to be accessed. Our thread-safe context delegates each call to *`ThreadLocal.get()`*, which involves an entry lookup in a highly optimised ThreadLocalMap. Even despite the efforts of JDK developers, the cost of this lookup is significantly higher than the cost of accessing a field in class when it is performed many times. Therefore, we need a way to mitigate this cost so that every time a thread-local structure is accessed, it does not incur this cost.

Our solution is to use thread-local "shadow contexts". The idea is to create thread-local shallow copies of the execution context. During parallel execution, the top-level job would obtain this copy "shadow context" by calling the "*getShadow()*" method on the context. This method returns a thread-local copy of the context. This copies the references from the parent context using getter methods. Therefore, when calling *getFrameStack()*, *getExecutorFactory()* etc. the thread-local value is returned. This is then cached since the shadow context is the same as that used in sequential execution. The downside to this is that a new execution context object is created for every parallel job. Although in terms of memory this is not a major concern since these are shallow copies of the parent context, there is nevertheless an additional overhead of having to eagerly query the execution context's state to copy its references for every job. In practice this is extremely cheap since the non-concurrent getter methods are direct field accesses. Profiling shows that the benefits greatly outweigh this small cost, since data structures like the FrameStack and ExecutorFactory are accessed many times even for the smallest of jobs throughout the Epsilon AST, so ensuring that the ThreadLocalMap is only accessed once per parallel job is a worthwhile optimisation. Of course, another downside is that added maintenance cost, since extensions will have to also provide appropriate overrides and copy constructors etc., though in practice this is a relatively small burden.

4.3.6 **Lazy Constraints and Dependencies**

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

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

In practice, we optimise the checking of the constraints depended on every time a constraint is executed using a flag which indicates whether the constraint is a dependency. This flag is set to true on the constraint when it is first invoked by a *satisfies* operation. If this flag is true, we know to check the trace for a result, otherwise we proceed as usual.

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

```
public UnsatisfiedConstraint check(Object self, IEvlContext context) throws
EolRuntimeException {
    UnsatisfiedConstraint unsatisfiedConstraint = preprocessCheck(self, context);
    boolean result;

    if (isDependedOn() && context.getConstraintTrace().isChecked(this, self)) {
        result = context.getConstraintTrace().isSatisfied(this, self);
    }
    else {
        result = executeCheckBlock(self, context);
        if (!context.isOptimizeConstraintTrace()) {
            context.getConstraintTrace().addChecked(this, self, result);
        }
    }

    postprocessCheck(self, context, unsatisfiedConstraint, result);
    return result ? null : unsatisfiedConstraint;
}
```

Listing 4.3.3 Code extract from EVL *Constraint* checking logic

```
boolean valid;
if (constraint.isDependedOn() && constraintTrace.isChecked(constraint, source)) {
    valid = constraintTrace.isSatisfied(constraint, source);
}
else {
    valid = constraint.appliesTo(source, context) &&
            constraint.check(source, context) != null;

    if (context.isOptimizeConstraintTrace()) {
        constraintTrace.addChecked(constraint, source, valid);
    }

    constraint.setAsDependency();
}
```

Listing 4.3.4 Code extract from EVL *satisfies* operation logic

More concretely, consider the code in Listings Listing 4.3.3 and Listing 4.3.4. Every time a constraint's *check* block is invoked for a given element, the *isDependedOn* Boolean flag on the constraint is checked (Listing 4.3.3). This flag starts off as being false, and is only set to true if the constraint was invoked by a dependency at some point, as shown by the last line in Listing 4.3.4. If the flag is false, then the cache (*constraintTrace*) is never checked, and the check block is executed normally every time. However if the constraint is known to be the target of a dependency through a previous invocation (i.e. the flag is true), then the cache is checked to determine whether the element for this constraint has been evaluated. If a result exists, then it is returned, otherwise the check block is executed.

Note also the presence of another Boolean flag in the execution context: *optimizeConstraintTrace*. If set to true, this flag prevents unnecessary writes to the cache, saving both time and memory. If the flag is set to false, every time the check block is executed, the results are added to the cache (Listing 4.3.3) irrespective of whether the constraint was a dependency or not. This is useful in some cases where a complete execution trace is desirable (for example, in incremental execution), however in most cases the only time when the cache will be used is when there is a *satisfies* call. Hence, if the optimisation is enabled, the cache is only written to from the *satisfies* operation (Listing 4.3.4).

4.3.7 Testing for Correctness

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

Furthermore, debugging concurrent programs is also difficult, since the same tools and techniques used to detect issues with sequential programs may be inadequate or misleading when used for concurrent programs. For details, see the evaluation (section 4.5.1).

4.4 Parallelisation Strategy

Having dealt with thread-safety issues and ensuring the execution context is set up correctly, we now turn to the actual parallelisation of program execution. There are different levels of granularity for which we can introduce parallelism. From Listing 4.2.2, we can see there are three nested *for* loops, in the following order:

1. For all *ConstraintContext*s
2. For all model elements conforming to the type specified by the ConstraintContext
3. For all *Constraint*s declared within the ConstraintContext

An assumption we can make is that there will be at least as many *Constraint*s as there are *ConstraintContext*s (excluding the *GlobalConstraintContext*), since a Constraint must be contained within a ConstraintContext. Therefore it wouldn’t make sense to parallelise only the

ConstraintContexts when parallelisation of Constraints allows for more potential scalability. However note that although it is possible to parallelise only Constraints by refactoring the algorithm so that (2) and (3) are swapped in order, this would not be efficient.

Recall that the issues with scalability are, by definition, rooted in large models. It is entirely plausible to have only a single Constraint in an EVL program, especially for simple (meta)models. Therefore, a sensible parallelisation strategy should scale with the data. We only consider data-parallel strategies. Whilst efficient rule parallelisation is possible, this only makes sense in a limited number of scenarios. An ideal scenario in which rule parallelisation (as opposed to data parallelisation) is when there are many Constraints within different ConstraintContexts and, crucially, the number of model elements applicable to each rule (ConstraintContext) is relatively equal. If this is not the case, then the execution time will be dominated by the rule with the most elements. Another crucial factor is how demanding each Constraint is, not only the number of applicable elements. Regardless of how demanding or numerous the Constraints are, a data-parallel approach is almost always superior or at least as good as a rule-parallel one. To illustrate, suppose that there are *M* model element types, each with *C* associated Constraints and *N* model element instances, where *N* >= 1 for all *M*. A data-parallel approach will result in a potential parallelisation of *N* for all *M*. By contrast, a rule-parallel approach will only result in a potential parallelisation of *C*. Thus, the only time when a rule-parallel approach is superior to a data-parallel one is when *C* > *N*; i.e. when there are more Constraints than model elements. Needless to say, this is rarely the case, especially when dealing with large models!

4.4.1 Parallelisation infrastructure

We have discussed how a scalable parallel decomposition should, by definition, scale with the number of model elements. This allows us to take advantage of up to as many processing cores as there are model elements. That is, the number of individual parallel workloads (herein referred to as “jobs”) can be expressed at varying levels of granularity, with the finest being one model element per job. As there could potentially be millions of jobs, we need an efficient way to co-ordinate the mapping of jobs onto processors and efficient scheduling. This task is greatly simplified by abstracting away from using threads directly and instead relying on the notion of an *ExecutorService* [267]; an interface which allows for the submission of jobs without the caller needing to handle the low-level execution details.

A well-known implementation is a *ThreadPoolExecutor* [268], which uses a queue (FIFO) structure to buffer submitted jobs and map them onto a re-usable pool of threads. The implementation is configurable to allow for varying number of threads, their lifetime and how many jobs can be buffered. Since we are interested in maximising throughput and ensuring maximum utilisation of all processors, we opt for an unbounded queue and a fixed number of threads with unlimited lifespan, where the number of threads is equal to the number of logical cores on the system. This approach allows for an unbounded number of jobs without the expensive overhead of creating new threads for each job. As the task of model validation is CPU-intensive and we are assuming the model is can be queried quickly without side-effects, using as many threads as there are logical cores minimises memory usage and context-switching whilst still keeping all cores active. In opting for a high-granularity data-parallel approach, we ensure that regardless of how long it takes each job to finish, there will always be enough work to keep the (hardware) threads active: even in the worst-case scenario where a single job dominates execution time, we can still make use of all other cores to execute other jobs. However although this data-parallel approach and thread pool configuration maximises CPU utilisation, it is not necessarily the most efficient. The trade-off is the overhead of scheduling lots of jobs at a high frequency onto threads. If the jobs are computationally "cheap" (i.e. very fast) and there are many of them, the overhead of scheduling them onto threads from the queue could become a significant factor in performance. It is not inconceivable to imagine a scenario where the time spent creating the jobs, submitting them to the queue and moving them from the queue onto processing threads outweighs the time spent on actually processing the jobs. In practice however, we found this not to be a limiting factor, as will become clear in the evaluation. Even if the users' jobs are computationally light, which is not uncommon for model validation – for example, ensuring that an integer property of the *self* model element is greater than some integer expression – the fact that Epsilon is interpreted alone makes such jobs more demanding than the well-optimised scheduling algorithm of the thread pool implementation.

Another contributing factor to the relatively lightweight nature of a granular data-parallel approach is the practical definition of "job" in terms of the Java implementation. The ExecutorService API accepts jobs as objects implementing the *Runnable* [269] or *Callable* [270] interface. The former consists of a single method with no parameters or return type, whilst the latter can return a result or throw an exception. In Java, interfaces with a single abstract method are *functional interfaces* [271], which allows their implementations to be expressed as a lambda expression. This is not merely syntactic sugar for anonymous inner classes, but rather a more lightweight mechanism which allows the method to be invoked directly from a native virtual machine instruction without the same overhead of traditional objects / anonymous classes. Lambda expressions can therefore be optimised and in some cases lead to superior performance [272].

For our use case in EVL, we only need one thread pool executor, which we submit all of our jobs to and wait for termination (i.e. the main thread blocks). This termination mechanism is handled by the ExecutorService implementation, however as we shall see in a later chapter, there are more complex semantics and implementation details to consider.

4.4.2 **Elements-based**

Perhaps the simplest data-parallel strategy is to evaluate each model element independently. In other words, to parallelise the second *for* loop. This strategy scales with the number of model elements to be checked, and has the added benefit of keeping the origin algorithm structure intact, so no unnecessary complexity is introduced.

```
@Override
protected void checkConstraints() throws EolRuntimeException {
    EvlContextParallel context = getContext();
    for (ConstraintContext cc : getConstraintContexts()) {
        Collection<?> allOfKind = cc.getAllOfSourceKind(context);
        Collection<Callable<?> jobs = new ArrayList<>(allOfKind.size());
        for (Object element : allOfKind) {
            jobs.add(() -> cc.execute(element, context.getShadow());
        }
        context.executeAll(jobs);
    }
}
```

Listing 4.4.1 *EvlModuleParallelElements* execution algorithm

```
public void execute(Object modelElement, IEvlContext context) throws
EolRuntimeException {
   if (!isLazy(context) && appliesTo(modelElement, context)) {
      for (Constraint constraint : getConstraints()) {
         constraint.execute(modelElement, context);
      }
   }
}
```

Listing 4.4.2 *ConstraintContext* element execution algorithm

Listing 4.4.1 shows the EVL module implementing the purely data-parallel execution strategy. Here we can see that each job executes each ConstraintContext-element pair in parallel. The execution algorithm of each ConstraintContext for a given model element is shown in Listing 4.4.2. This approach is not only concise and intuitive to express, but also scalable and applicable to various extreme scenarios. Consider the case where there is only a single ConstraintContext and Constraint, but many elements. This approach will result in as many jobs as there are elements, as discussed previously. However this data-parallel approach also works when there are many ConstraintContexts (i.e. model element types to validate) each with only a single element, since a new job will be created for each element regardless of the type. The main limitation of this strategy is when there are many Constraints within a single ConstraintContext and relatively few model elements. As noted previously, this is an extremely rare case, so this strategy is the default as it handles the most common scenarios efficiently.

Note that the execution algorithm of this strategy will only ever evaluate a single ConstraintContext at a time, rather than submitting all of the jobs to the ExecutorService at once. The *context.executeAll* method is blocking: it takes as input a collection of jobs (Runnable or Callable), submits them to the ExecutorService and waits for them to complete (the precise details of this are explored in Section 6.2.6). That is, the call occurs within the loop, rather than outside of it.

This is to improve data locality and minimise memory allocation, since having lots of jobs from different contexts (model element types) would require fetching the elements from the model, caching them etc. along with the Constraints associated with each ConstraintContext.

The keen observer may notice that when executing each ConstraintContext for a given element, we do not pass in the original execution context, but rather a “shadow” copy. As discussed in section 4.3.5, the idea is to avoid the cost of ThreadLocal lookups, so before execution the ThreadLocal values are retrieved and cached for the currently executing thread (it may be a useful clarification for the reader that the code inside the lambda expression is executed in parallel).

4.4.3 Constraints-based

```java
@Override
protected void checkConstraints() throws EolRuntimeException {
    EvlContextParallel context = getContext();
    for (ConstraintContext cc : getConstraintContexts()) {
        Collection<?> allOfKind = cc.getAllOfSourceKind(context);
        Collection<Callable<?> jobs = new ArrayList<>(allOfKind.size());
        for (Object element : allOfKind) {
            if (!cc.isLazy(context) && cc.appliesTo(element, context)) {
                for (Constraint constraint : cc.getConstraints()) {
                    jobs.add(() ->
                        constraint.execute(element, context.getShadow())
                    );
                }
            }
        }
        context.executeAll(jobs);
    }
}
```

Listing 4.4.3 *EvlModuleParallelConstraints* execution algorithm

In the previous section, we noted that the main limitation of a purely data-parallel approach is the lack of scalability when there are very few total model elements but many rules (Constraints), especially when such rules are computationally expensive. An alternative solution is to parallelise not only the data but also the rules. This strategy is the most granular level of parallelism possible and is achieved by a simple modification of the algorithm in Listing 4.4.1.

Listing 4.4.3 shows how we can achieve both data- and rule-level parallelism by moving the job creation and submission one level down to the inner-most *for* loop. In this strategy, every constraint-element pair is executed in parallel. Whilst this approach appears to be superior to the previous one due to its greater scalability potential, there are disadvantages and complications.

The most obvious downside to this approach is that more work is performed outside of the job, so although there are more potential jobs, the total computation which occurs sequentially in the main thread has increased. Most notably is the if statement corresponding to the ConstraintContext is executed sequentially. This checks whether the ConstraintContext is lazy and whether its guard block (if present) returns true. Checking whether a ConstraintContext is lazy involves checking whether itself is annotated as lazy, or whether all of its Constraints are lazy. The result of this is cached after it is evaluated. Even with many Constraints and ConstraintContexts, such a check doesn't pose a significant cost due to caching. The demanding part comes from executing the guard block. Since the guard block of each ConstraintContext is checked for every applicable element and the block itself may contain arbitrarily complex code, this strategy could leave the ExecutorService starved of jobs, especially if the guard block returns false often. This strategy is therefore only suitable when the guard block is absent from the ConstraintContexts. Admittedly, this is usually the case: it is far more common to restrict the application of rules at the Constraint level than at the model element type level, so guard blocks are mostly found in Constraints. Placing a guard block on a ConstraintContext is, from the user's perspective, semantically identical to repeating the guard block in all of its contained Constraints. In theory, we could also make this transformation manually with relative ease, by executing the guard block of the ConstraintContext when checking each Constraint, and then if this block returns true, we proceed to check the guard block of the Constraint itself if it is present, then the check block and so on.

Even if we effectively remove the ConstraintContext guard block and inline in it into the Constraint's guard, we still have the overhead of handling additional jobs. In the purely data-parallel strategy, we created one job per model element, however this time we're creating one job per Constraint per element, or per Constraint-element pair. Although the additional overhead and memory consumption is likely to be minimal, we argue that such an approach is unlikely to be beneficial outside of rare cases where there are more constraints than there are model elements.

4.4.4 **Atom-based**

We have established that a data-parallel strategy allows for a high level of granularity, which is desirable as it allows for efficient CPU utilisation with automatic load-balancing when using a thread pool. However, our approaches so far have focused on direct modifications to the sequential algorithm, which limits the potential execution strategies. A more flexible and generic method is to consolidate jobs (for example, constraint-element pairs) into their own data structures. At first, this seems counter-productive because we are adding another unnecessary abstraction layer, which translates into an additional object for every job. However, if these new objects implement the appropriate Callable or Runnable Java interfaces, then they *are* the jobs, albeit with additional information which we can use. This structure consists of a (immutable) model element and executable rule, where the rule has a method which accepts a model element and optionally returns a result. As previously mentioned, the state of the execution engine is encapsulated in an *EvlContext* object and set upon construction. This approach can be factored out into an interface which is then implemented by both *Constraint* and *ConstraintContext* elements. For simplicity, we will refer to Constraint jobs as *ConstraintAtom*s and to ConstraintContext jobs as *ConstraintContextAtom*s. The execution (in simple terms, the `run` or `call` methods) of these atoms is equivalent to the jobs / lambda expressions in Listing 4.4.1 and Listing 4.4.3.

We can decompose an EVL program into a collection of “atoms” by iterating over all model elements and the appropriate rule construct. An example of how these jobs are obtained for the ConstraintContextAtom is shown in Listing 4.4.4. Each EVL module consist of a number of ConstraintContexts, each of each contains any number of Constraints. Since we are interested in data-parallelisation, we only need to obtain the model elements of type (and subtypes) described by the ConstraintContext. The ConstraintContext is therefore the rule, where its body is the Constraints it contains. Since we have the logic to execute (the Constraints contained within the ConstraintContext) and the data (the model element), Listing 4.4.4 returns a list of executable jobs.

When a ConstraintContextAtom is submitted to an ExecutorService, the `call` or `run` method will delegate to the `execute` method of the ConstraintContext (Listing 4.4.2). For reference, a simplified (i.e. inheritance / generics and utility methods removed) view of the ConstraintContextAtom is shown in Listing 4.4.5.

For a visual guide to the concept of this approach, see Figure 4.4.1. To make this decomposition clearer, Listing 4.4.6 shows an example EVL program with two ConstraintContexts. Assuming there are *Nc* instances of *ClassDeclaration* and *Nm* instances of *MethodDeclaration*, then the total number of jobs will be *2 Nm + Nc*, as there are two constraints applicable to the *MethodDeclaration* context and one for the *ClassDeclaration*. Note that with this atom-based execution strategy, all jobs are created eagerly and submitted to the ExecutorService at once. Unlike the Elements-based execution algorithm in Listing 4.4.1, we do not have multiple “rounds” of parallel execution: all jobs are created in advance. In theory the main difference is that by creating all jobs in advance, the cache of all model element types to be validated will have been populated prior to parallel execution. This should lead to more predictable execution times since there won’t be as much synchronization involved when populating caches, since in a sense this is performed eagerly rather than lazily.

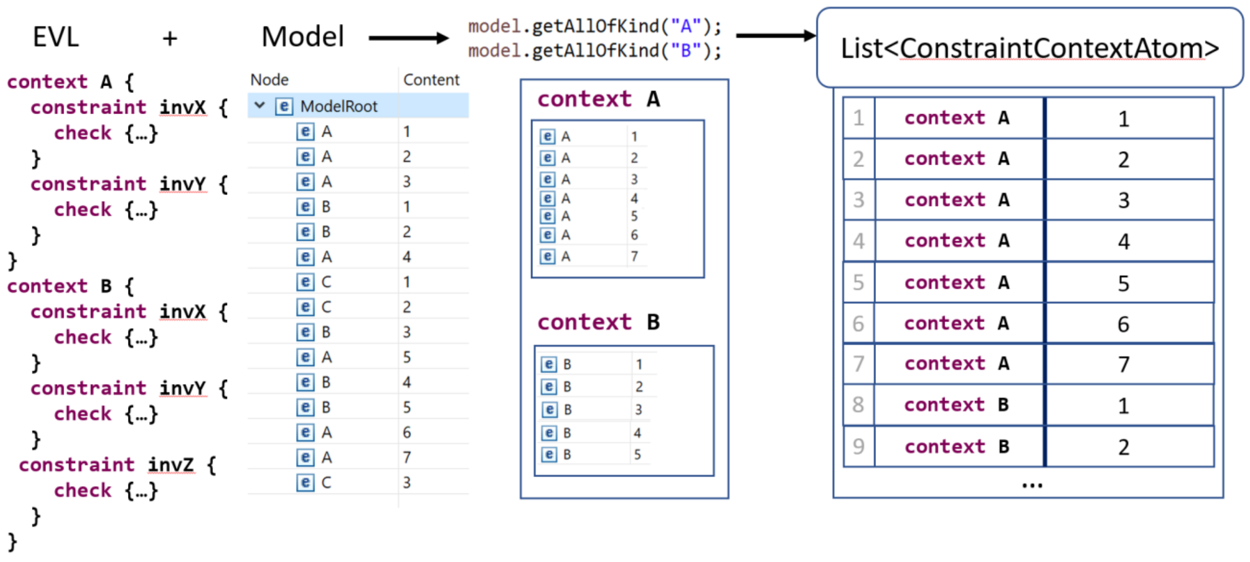
Note that we can easily adapt the ConstraintContextAtom execution algorithm to *ConstraintAtom*s, simply by changing the type of *rule* object in Listing 4.4.5 to Constraint, and modifying the job creating algorithm in Listing 4.4.4. Doing so would result in as many jobs as there are Constraint-element pairs, making the granularity equivalent to the Constraint-based algorithm in Listing 4.4.3.

```java
static ArrayList<ConstraintContextAtom> getAllJobs(IEvlModule module) {
   IEvlContext context = module.getContext();
   ArrayList<ConstraintContextAtom> atoms = new ArrayList<>();
   for (ConstraintContext cc : module.getConstraintContexts()) {
      Collection<?> allOfKind = cc.getAllOfSourceKind(context);
      atoms.ensureCapacity(atoms.size()+allOfKind.size());
      for (Object element : allOfKind) {
         atoms.add(new ConstraintContextAtom(cc, element, context));
      }
   }
   return atoms;
}
```

Listing 4.4.4 *ConstraintContextAtom* job creation algorithm

```java
public class ConstraintContextAtom implements Callable<Boolean> {

   public final ConstraintContext rule;
   public final Object element;
   protected IEvlContext context;

   public ExecutableRuleAtom(ConstraintContext cc, Object modelElement) {
      this.rule = cc;
      this.element = modelElement;
   }

   public Boolean execute(IEvlContext context) throws EolRuntimeException {
      return rule.execute(element, context);
   }

   @Override
   public final Boolean call() throws Exception {
      if (context == null)
         throw new IllegalStateException("Context cannot be null!");
      return execute(context);
   }
}
```

Listing 4.4.5 *ConstraintContextAtom* structure (non-generic)



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

Node	Content
ModelRoot	
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

```
model.getAllOfKind("A");
model.getAllOfKind("B");
```

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

Figure 4.4.1 Demonstration of atomic Rule-element decomposition of EVL programs

```
context MethodDeclaration {
   constraint shouldStartWithLowerCase {
      check: self.name.firstToLowerCase() == self.name
   }
   constraint localMethodIsUsed {
      guard: self.modifier.isDefined() and self.modifier.isLocal()
      check: self.usages.notEmpty()
   }
}
context ClassDeclaration {
   constraint comparatorImplementsSerializable {
      guard : self.implements("Comparator")
      check : self.implements("Serializable")
      message : self.name+" is not Serializable"
   }
}
```

Listing 4.4.6 Reference example for job creation algorithm

4.4.5 **Stage-based**

A radically different execution strategy inspired by Smith (2015) [10] is to execute EVL in three stages. The first stage loops through all ConstraintContexts and creates

ConstraintContextAtom instances for all elements which satisfy the guard block (if present). These atoms are then fed to the second stage, which loops through all of these ConstraintContextAtoms. With each iteration, the guard block of all Constraints of the atom's ConstraintContext are checked (only if the Constraint is not lazy of course) for the given element (from the ConstraintContextAtom), and for Constraints which "should be executed" (i.e. not lazy and the guard is satisfied), a flat collection of ConstraintAtoms is returned. This is fed into the third (final) stage which executes the check block for all constraint-element pairs (i.e. the ConstraintAtoms).

The main disadvantage with this strategy is the overhead of creating many intermediate objects; far more atoms than there are model elements to be checked infact. Consequently this approach is potentially more memory-intensive, and memory bandwidth is known to be a limiting factor in modern computing tasks. Furthermore, the intermediate objects pose a memory capacity overhead, which may be very high especially at the end of the second stage where there are both ConstraintAtom and ConstraintContextAtom instances. The only potential redeeming factor of this approach is that the execution path of the final stage – which is by far the most time-consuming – is much more predictable and so potentially easier for the JVM to optimise. This is because the decision for whether to evaluate the Constraint was made at an earlier stage, so each parallel job has the sole task of executing the check block for a given element unconditionally. By contrast, the other algorithms may result in many jobs terminating very quickly if the "should be checked" logic returns false, which results in more context-switching and greater burden on the thread pool's assignment / scheduling algorithm. The stage-based approach eliminates this uncertainty so that the final stage is simpler and guaranteed to be long-running – at least in relative terms – which may ultimately result in greater throughput.

4.4.6 Annotation-based

In all of the previous strategies, parallelisation was automatic and applied on at least a data-level granularity. This is suitable for most cases, however as discussed, EVL is a complex language given that it builds on EOL, which has imperative constructs and allows for arbitrary Java code to be invoked. In some complex EVL programs, correctness may not be guaranteed if, for example, mutation of global state (e.g. variables declared in the *pre* block) is performed during the execution of some constraints. The user may have written their programs with a particular order of execution in mind, and with our parallel execution strategies this is not always guaranteed. In such cases, it would be more appropriate if the user could explicitly specify where parallelism is applied.

Our solution is achieved through an annotation-based directive applied to rules, so that only rules specified by the user will be executed in parallel, leaving other rules free to take advantage of parallel collection operations both implicitly (i.e. the default invocation) and explicitly (i.e. by prepending them with "*parallel*"). Listing 4.4.7 shows an example of how this would look from the user's perspective. This example is based on Listing 4.2.1, but adapted so that whenever a *ClassDeclaration* instance which has a "hashCode" method but not an "equals", this class is added to a distinct, ordered collection declared in the *pre* block, which can then be processed later. We added an extra condition such that this collection is only populated if there are more than a threshold number of public methods on the class. Since we require deterministic ordering and the fact that the collection we're adding to is not thread-safe, we do not execute the "hasEquals" Constraint in parallel. Of course, there are ways around this, but the purpose of the example is to illustrate the principle. The rule(s) with the `@parallel` annotation would be executed in parallel, whilst rules without would execute sequentially. Note that we can annotate either Constraints or ConstraintContexts; in the latter case, all Constraints declared within the ConstraintContext will be executed in parallel, as shown in the *MethodInvocation* context. Note that we can parallelise lazy Constraints, as shown by the "hasHashCode" Constraint.

In our example, we have effectively "lost" some parallelisation potential since "hasEquals" is not parallelised due to the need to write to global state. However, this non-thread-safe operation only occurs for some elements, with the criteria specified in the *if* statement. It would be ideal if we could parallelise some or all Constraints for which the criteria does not hold; that is, in cases where it is guaranteed that the "hcButNoEq" collection will not be written.

A solution to this is to enable the user to specify the conditions under which a rule – or even rule-element pair – is parallelised, rather than having to change the script (or copy-paste with minor changes) depending on the input model(s). We can offer this facility using executable annotations, which can take arbitrary EOL expressions as a parameter. We assume that user-defined annotation expressions return a Boolean indicating whether parallelisation should take place or not. If no expression is provided, we assume *true*. If no annotation is provided, we assume *false* as before. Continuing with our example, Listing 4.4.8 shows how the user can exploit their knowledge of the program to parallelise Constraint-element pairs for which the conditionally non-thread-safe logic does not hold. We invert the condition as a pre-requisite for parallelisation, such that *ClassDeclaration* instances with fewer than 10 methods are parallelised.

```
pre {
   var hcButNoEq = new OrderedSet;
}

@cached
operation AbstractTypeDeclaration getPublicMethods() : Collection {
  return self.bodyDeclarations.select(bd |
      bd.isKindOf(MethodDeclaration) and
      bd.modifier.isDefined() and
      bd.modifier.visibility == VisibilityKind#public
  );
}

context ClassDeclaration {
  constraint hasEquals {
    guard : self.satisfies("hasHashCode")
    check {
      var methods = self.getPublicMethods();
      var hasEq = methods.exists(method |
        method.name == "equals" and
        method.parameters.size() == 1 and
        method.parameters.first().type.type.name == "Object" and
        method.returnType.type.isTypeOf(PrimitiveTypeBoolean)
      );
      if (not hasEq and methods.size() > 10) hcButNoEq.add(self);
      return hasEq;
    }
  }

  @parallel
  @lazy
  constraint hasHashCode {
    check : self.getPublicMethods().exists(method |
      method.name == "hashCode" and
      method.parameters.isEmpty() and
      method.returnType.type.isTypeOf(PrimitiveTypeInt)
    )
  }
}

@parallel
context MethodInvocation {
  guard: self.method.isDefined()
  constraint doesNotCallFinalize {
    check: not (self.method.name == "finalize" and
                self.method.parameters.isEmpty())
  }
  constraint doesNotCallExit {
    check: not (
      self.method.name = "exit" and self.method.parameters.size() == 1 and
      self.method.parameters.first().type.type.isTypeOf(PrimitiveTypeInt))
  }
}
```

Listing 4.4.7 Example of annotation-based parallelism

```
$parallel self.getPublicMethods().size() <= 10
constraint hasEquals {
  guard : self.satisfies("hasHashCode")
  check {
    var methods = self.getPublicMethods();
    var hasEq = methods.exists(method |
      method.name == "equals" and
      method.parameters.size() == 1 and
      method.parameters.first().type.type.name == "Object" and
      method.returnType.type.isTypeOf(PrimitiveTypeBoolean)
    );
    if (not hasEq and methods.size() > 10) hcButNoEq.add(self);
    return hasEq;
  }
}
```

Listing 4.4.8 Parameterised parallel annotation example (based on Listing 4.4.7)

The algorithm for the annotation-based EVL module – shown in Listing 4.4.9 – is longer and more complex than other execution strategies, but the main idea is relatively straightforward. Since either ConstraintContext or Constraint may be parallelised (as they are both "rules"), and the decision to parallelise or execute sequentially is made for each rule-element pair, we accumulate parallel jobs into a collection and execute them in the usual way. The fact that either ConstraintContext or Constraint may be parallelised (i.e. the fact that we have two Rule constructs) lengthens the algorithm, however the principle is the same in both cases. Thus the algorithm is a dynamic combination of the Elements-based and Constraints-based algorithms presented in sections 4.4.2 and 4.4.3. One feature to note is that the sequential rule-element pairs will always be executed before the parallel ones. This ensures that the parallel rules can safely depend on sequential ones, having assumed they have already been executed.

In Listing 4.4.10, we show the variables available to the user when deciding to parallelise a rule-element pair. Of course, the element itself is made available so that use cases such as in Listing 4.4.9 are possible. However the user may also want to make decisions based on how many model elements there are of the annotated ConstraintContext (or the ConstraintContext which an annotated Constraint belongs to). We also include the model which the ConstraintContext's elements belong to for convenience, so that for example the number of total model elements can be queried. In some cases the decision to parallelise may depend on how many processing threads are available and the user may know the threshold for which parallelisation of the specified rule brings performance benefits as opposed to parallelisation of collection operations (as discussed in section 6.2) or no parallelisation at all. These decisions may be dependent on the execution environment, where the user may not necessarily want 100% CPU usage in order to allow other background tasks to run efficiently. In any case, we have provided the user with the necessary information and tooling to be able to make informed decisions on parallel execution at the finest level of granularity possible. As previously discussed, users can take advantage of the advanced features available to them to optimise for their cases whilst ignoring features that they would not benefit from. Although from a user-centric perspective our approach may appear reasonable with no obvious drawbacks, we also need to consider the performance impact imposed by the more elaborate execution algorithm, which must cater for the most complex "worst-case" scenario. More specifically, a potential inefficiency is having to determine whether parallelisation should be applied for every rule-element pair, meaning potentially millions of *if* statements. In these cases the JVM cannot optimise to the same extent as the simpler parallel execution strategies, or even to the same extent as non-parameterised annotations – it is arguably at the mercy of the branch predictor. With simple (i.e. non-parameterised) annotations such as in Listing 4.4.7, there are only as many *if* statements as there are rules, however with parameterised executable annotations, there are many more. It should be noted however that we only populate the FrameStack with the variables if an annotation is present. Nevertheless, it is undoubtedly more efficient to use the simple (automatic) execution strategies than to manually annotate every Constraint with `@parallel`.

```java
for (ConstraintContext cc : getConstraintContexts()) {
   final Collection<Constraint> constraints = cc.getConstraints();
   final Collection<?> allOfKind = cc.getAllOfSourceKind(context);
   final int numElements = allOfKind.size();
   final IModel model = cc instanceof GlobalConstraintContext ?
      null : cc.getType(context).getModel();
   final Collection<Callable<?>> jobs = new LinkedList<>();

   if (cc.hasAnnotation(PARALLEL_ANNOTATION_NAME)) {
      for (Object object : allOfKind) {
         if (shouldBeParallel(cc, object, model, numElements)) {
            jobs.add(() -> cc.execute(constraints, object, context));
         }
         else {
            cc.execute(constraints, object, context);
         }
      }
      context.executeAll(jobs);
   }
   else {
      for (Constraint constraint : constraints) {
         for (Object object : allOfKind) {
            if (cc.appliesTo(object, context)) {
               if (shouldBeParallel(constraint, object, model, numElements)) {
                  jobs.add(() -> constraint.execute(object, context));
               }
               else {
                  constraint.execute(object, context);
               }
            }
         }
         context.executeAll(jobs);
      }
   }
}
```

Listing 4.4.9 *EvlModuleParallelAnnotation* execution algorithm

```
boolean shouldBeParallel(AnnotatableModuleElement ast, Object self, IModel
model, int numElements) throws EolRuntimeException {
   if (!hasParallelAnnotation(ast) || !getContext().isParallelisationLegal()) {
       return false;
   }
   return shouldBeParallel(ast, new Variable[] {
      createReadOnlyVariable("self", self),
      createReadOnlyVariable("NUM_ELEMENTS", numElements),
      createReadOnlyVariable("MODEL", model),
      createReadOnlyVariable("THREADS", getContext().getParallelism())
   });
}
```

Listing 4.4.10 Variables available to user for parallel annotations

We discuss further the motives for a user-guided approach to parallelisation and its implications in section 6.6.

4.5 Evaluation

In this section, we describe how we compared our parallel implementations to the status quo: namely, the sequential execution engine of EVL and Eclipse OCL.

4.5.1 Correctness

4.5.1.1 Preliminaries

Our automated testing framework consists of a series of parameterised JUnit tests, where the parameters consist of “modules” and “scenarios”. A *module* is an implementation (more precisely, a driver) of the execution engine. We use the original sequential module as the oracle for our tests – that is, the results and internal state of this module are assumed to be correct, so that the concurrent implementation(s) are expected to provide identical results to this module. A *scenario* provides the inputs to a module. In most cases, this consists of a script (written in the domain-specific language for the task), a model and metamodel. However, these are not the only inputs to the test. Each module may also have additional configuration parameters – for instance, we may specify the number of threads to use for the concurrent modules. The combination of different configuration parameters, modules, scripts and models can lead to a large input space which can be automatically tested for equivalence.

For the number of threads, we used 1, 2, “default” and “many”, as we believe these are exhaustive of all scenarios to be confident in the correctness of our solutions. Single-threaded checks to ensure that the implementation behaves correctly in concurrent conditions even when there is no parallelism, whilst two threads is the minimum for parallel execution. Our default value is equal to the runtime's number of logical cores (i.e. physical + SMT). This represents a typical use case from an end user's perspective. We also run our tests with "many" (specifically, 16383) threads in order to ensure that under extreme circumstances, a large thread pool does not lead to abnormal behaviour. Throughout development, we have experimented with various number of threads on a regular basis to ensure there was no abnormal behaviour.

We focus our testing efforts on providing more diverse inputs ("scenarios") than on examining the internal states of the modules; - after all, there are some fundamental differences between the concurrent and original solutions, so intermediate results may inevitably vary. From the end user's point of view, it's the end results (outputs) which matter, not the derivation. That said, we attempt to tests for equivalence in data structures affected by parallelism. For example, our solutions use a different frame stack for every thread which is then merged back into the main thread's frame stack once parallel execution completes. We therefore write a test to ensure that the frame stacks are "equivalent". In most cases, due to the non-deterministic execution order of any concurrent environment, we only need to check that they contain the same elements.

Prior to running the main body of our equivalence tests, we obviously need to ensure that both the oracle and test configurations have completed execution. For our test configuration, if any exception occurs, we fail immediately. This is perhaps one of the most fundamental tests – after all, if the program cannot complete without crashing or raising an exception, whether it is a checked or runtime (unchecked) exception, then the implementation is clearly incorrect. Furthermore, we check that we are comparing the same scenarios before proceeding. This is so that we can be confident to some degree that our testing infrastructure is correctly specified. The pre-amble, at the minimum, asserts the following statements in relation to the test and oracle configurations:

- They have same scenario ID (calculated as a hash of script, model and metamodel URIs)
- They have the same model (based on name and URI)
- They have the same script (based on name and URI)

Due to the non-deterministic execution scheduling of concurrent systems, a number of obscure bugs may become apparent only in certain conditions at seemingly random times. This may reveal a flaw in the solution's design and/or implementation. Since concurrency bugs are difficult to reproduce – and, in many cases, difficult to even detect in the first place – it is important to ensure that our tests are run many times. During development, most errors resulting from concurrency issues surfaced at least once in approximately ten to twenty runs. These were typically not triggered by a particular scenario, but by chance during execution of almost any model and script combination – even models with a handful of elements, for example. In order to increase our chances of detecting concurrency issues, we ran tests continuously on various computers throughout the development process. Every time a significant change was made to the code, the test suite and our advanced tests were run multiple times. Although various optimisations have been made throughout the project, the core concurrency infrastructure approach has been mostly the same for about two years. Throughout this time after running the suite thousands of times and also running thousands of performance experiments, we never encountered concurrency issues or unreproducible / non-deterministic behaviour. We hope this is sufficient to counteract any concurrency issues which may go undetected by chance. By contrast, when experimenting with alternative implementations, in our experience even the most concurrency obscure bugs can be reliably encountered in a few dozen runs, whilst more serious bugs can usually be reproduced within 10 runs or so on average. We should also note that we run all of our experiments (i.e. not just those for testing) with the `-ea` JVM option, and throughout the code at certain points where concurrency bugs have been known to occur or may be a potential source, assertions are placed where it also improves clarity of design / intent.

It should also be noted that the Epsilon project already has a large suite of tests. Where appropriate and/or necessary, additional tests have been added or modified to further test for correctness of program behaviour. We believe that a thorough unit testing approach – both from existing and new tests – enables us to be reasonably confident in the correctness of our solutions which, we argue, should be sufficient for all practical purposes. For reference, the Epsilon test suite (which has been expanded as part of this thesis to test certain features more extensively) has at the time of writing approximately 2800 test methods. The test suite for parallel EVL (discussed in the next section) has over 7000, even despite our efforts to reduce this number to more reasonable level. The tests are thorough, varied and try to cover all language features and complex interactions, as well as testing intermediate data structures used in execution where appropriate. For details, please see the source code. The remainder of this section will provide further details on the testing procedure for each model management program.

4.5.1.2 EVL test suite

There are four test classes for the Epsilon Validation Language; two of which are new additions. The main existing test class ("EvlTests") uses a barebones XML model and mainly focuses on testing the engine's frame stack and execution semantics. It does this by writing the tests in an EVL script and checking for an expected number of unsatisfied constraints for given model elements. These tests exercise all of the features of EVL, including lazy constraints, lazy contexts, contextless constraints, constraint dependencies, guards, operations, fixes etc. It also checks to ensure that there are no scoping issues (i.e. variables are visible/not visible as and when appropriate). Many of these tests were additions made for increasing our confidence of the correctness of our concurrent implementations. These tests are run for all IEvlModule implementations. In total, there are 33 tests in "EvlTests" run against two EVL scripts. We are confident that "EvlTests" and accompanying scripts exercise all combinations of features of the Epsilon Validation Language in the most concise and minimalistic manner; which allows developers to easily find and diagnose failures whilst providing substantial code coverage.

The additional classes are both equivalence tests. "EvlModuleEquivalenceTests" is a complex test class which uses the original EvlModule implementation (with original configuration) as the oracle. The original EvlModule can also be run with the constraint trace disabled, but this is considered as a test module rather than an additional oracle for simplicity. For each module configuration and scenario, the equivalence tests assert the following statements in relation to the oracle module:

- Same number of unsatisfied constraints
- Test module contains all unsatisfied constraints in the oracle module
- Frame stacks are the same size
- Test module contains all the frames in the oracle frame stack

- Same number of constraints
- Test module contains all the constraints in the oracle module
- Constraint traces are the same size
- Test constraint trace contains all trace items in the oracle constraint trace
- Constraints depended on are same size
- Test operation list contains all operations in the oracle operation list
- Test operation contributor registry has at least as many operation contributors as the oracle.

The rationale for testing data structures other than the output (unsatisfied constraints) is that these were common sources of problems and disparities between the oracle and concurrent modules during development, so we wanted to ensure that these problems are no longer present.

Our other equivalence test ("EvlOclTests") compares the result (i.e. unsatisfied constraints) of a given EVL module with an equivalent OCL (Object Constraint Language) one. Specifically, we use the Pivot API of Eclipse OCL (version 6.7). Since Eclipse OCL is designed to be integrated into Eclipse with a graphical editor, and its intended use is to embed constraints directly in EMF models, we needed to create a separate class for programmatically invoking OCL in a similar manner to EVL. Our façade takes as input three paths: the script, model and metamodel. More precisely, the "script" in this case is a Complete OCL document consisting of invariants for some contexts in a metamodel. Since the semantics of EVL are a superset of OCL with similar syntax, features and built-in operations, it is relatively straightforward to write identical scripts in EVL and OCL. However, OCL has limited support for imperative programming, and all operations must be free from side effects. Therefore, we tried to limit use of EVL's more complete language constructs to ensure the EVL and OCL scripts were as similar as possible. We even limited our use of declarative EVL features not present in OCL. For example, OCL does not support constraint and context guards in the same manner as EVL, so instead we used the "implies" operator in both cases.

Our test structure is similar to "EvlModuleEquivalenceTests", except that instead we are comparing results from two different tools. Our standalone OCL program uses the EMF EValidator API – specifically, a custom "Diagnostician". This allows us to get back a list of unsatisfied constraints, although the structure is different to Epsilon's representation. Nevertheless, we can get the messages for unsatisfied constraints from both EVL and OCL, which contain the name of the unsatisfied constraint. Since our scripts are almost identical in EVL and OCL, these names are the same and we can therefore compare them for equality. Furthermore, since Eclipse OCL only works on EMF and UML models, and our test models are EMF-based, we can even ensure that the unsatisfied constraints fail for the same model elements (EObjects) by using their URIs.

The choice of EVL module to compare against OCL should in theory be immaterial, since if we are confident in the equivalence between our oracle EVL module and the other implementations, then by transitivity we can also be confident that all of our modules will produce identical results in this test. For completeness, we chose to include all of our implementations, albeit with the default number of threads only and both with constraint trace enabled and disabled. For each module configuration and scenario, the equivalence tests assert the following statements in relation to the OCL-based program:

- Total number of unsatisfied constraints are the same
- Test module contains all unsatisfied constraints in OCL-based program

Note that in both “EvlModuleEquivalenceTests” and “EvlOclTests”, equality of unsatisfied constraints is based not only on their quantity and name of constraints but also on the model elements (specifically, EObjects). This is made possible because EMF provides identifiers for model elements, and model validation does not modify the input model (and none of our EVL scripts apply any fixes programmatically). An unsatisfied constraint in OCL is equal to an unsatisfied constraint in EVL if their constraint names are the same (this is determined by the scripts, which differ only in their syntax) and the associated element instances (EObject) are the same.

Our performance test suite for the Epsilon Validation Language (EVL) consists of three metamodels and three scripts (one for each metamodel). The models and metamodels were taken from [273], as this was a convenient resource designed for benchmarking model transformations. The resource provides models of varying sizes, which can be used to evaluate the scalability of our solution. The metamodels we used were IMDb (an extremely simple model of the Internet Movie Database), Java (specifically, reverse-engineered models of some parts of Eclipse projects obtained using MoDisco) and DBLP (which models a computer science bibliography).

Our choice of scripts and metamodels are complimentary in that they test different aspects of the execution engine. The IMDB metamodel allows us to test a simple metamodel with very simple constraints. Most of the validation logic simply checks a property of an element to ensure it meets a certain value.

The Java metamodel on the other hand, is extremely complex with lots of meta-elements and complex relationships between them (it is, after all, a metamodel of the Java programming language). We developed two identical scripts in both EVL and OCL (“java_findbugs.evl” and “java_findbugs.ocl” respectively). We took advantage of the complexity of the Java metamodel by making the script computationally expensive; with each constraint traversing a potentially large number of elements. Since we expect the obtained models to be semantically valid (i.e. that they compile), we did not simply re-implement some compiler logic. Instead, we aimed to write constraints which may be unsatisfied so that we could get some tangible results that can be compared. These constraints were inspired by a list of code smells and bad practices according to FindBugs descriptions [274]. Due to the complexity of the metamodel and time limitations, we tried to avoid constraints requiring heavy static analysis and context awareness. Nevertheless, we believe our constraints are of sufficient complexity and representative of a typically computation-heavy model validation scenario. Many of these constraints check that if an element has some property, then all or some of its parent or child elements must satisfy some other properties – in effect, leading to constraints of at least $O(n^2)$ complexity or similar. There are a total of 35 constraints across 18 different contexts, which we hope to be sufficient for testing both the performance and correctness of our solutions.

4.5.1.3 Issues detected by the test suite

Our unit tests highlighted a problem with disabling the constraint trace. The issue is that constraints can be invoked more than once due to dependencies (i.e. “satisfies” operation calls). In the absence of a constraint trace, the target of the satisfies operation must be rechecked. If the constraint depended on is unsatisfied, then this is added to the collection of unsatisfied constraints, resulting in duplication. One solution would be to check whether the constraint was invoked by a satisfies operation and if so, do not add the unsatisfied constraint to the results. However if the target of the satisfies operation is a lazy constraint (i.e. it can only be invoked by satisfies calls) then we want to add it to our collection of unsatisfied constraints, since it is being invoked for the first time. Even accounting for this, if the lazy constraint is depended on by multiple constraints, then the unsatisfied constraint may still be duplicated. Our solution to this problem is simple: we construct the unsatisfied constraint object; setting the model element instance, constraint and message, but we only add it to the results if it’s not already present. Although constructing the object and checking if it's already in the results can have a moderate impact on performance, it is necessary to ensure correctness; which is ultimately more important. The motive for disabling the trace is to improve performance, since most constraints are neither the target nor the source of satisfies calls; making the trace largely redundant. Since checking the presence of an unsatisfied constraint in the results is relatively expensive, we only do this if the constraint trace is disabled. This way we avoid checking whether a constraint has been executed twice when the trace is enabled.

Another problem found with parallelisation is lazy constraints. Such constraints are only evaluated if they are a dependency of another constraint (i.e. via a “satisfies” operation call). This poses a classic dependencies issue with concurrency. If multiple constraints depend on a lazy constraint, then this could lead to inconsistent state for the evaluation of the lazy constraint. There are three possible solutions to this problem. One possibility is to use static analysis and order the execution of constraints depended on so that their results can be re-used later. However, with a complex dependency graph, the scheduling and discovery can become complex. Another solution is to synchronize the execution of constraints depended on to avoid duplicate re-computation, but this poses a bottleneck by making execution single-threaded in such cases. Instead, our preferred solution is to simply allow duplicate computations to occur, so that if two constraints are being executed simultaneously and depend on the same constraint (which had not previously been evaluated), they each evaluate that constraint. Refer back to section 4.3.6 for more on this.

These are just a small sample of the problems detected by unit testing. Throughout the development process, many alternative implementations and optimisations were considered and the test suite helped not only to detect obvious fundamental flaws but also much more subtle concurrency issues.

In terms of coverage, the test suite achieves over 65% total coverage however it should be noted that the coverage for the actual execution logic in EVL (so excluding unused utilities, constructors, methods, auto-generated parser code etc.) is very close to 100%. The usual caveats with coverage metrics apply, however to our knowledge at the time of writing all features and semantics of EVL are tested thoroughly, with over 7000 JUnit tests and many more assertions. A detailed overview of the test suite can be found in Epsilon repository – specifically, in [275] at the time of writing. Equivalence tests with OCL can be found in [276].

4.5.2 **Performance**

As with correctness, the performance of EVL (sequential, parallel and distributed) has been tested extensively on many machines with different hardware and software (operating system JVM) configurations. In this subsection a relevant, representative sample of experiments are reported on and analysed. We primarily focus on the reverse-engineered Java models and IMDb models due to their sizes. The two metamodels (Java and IMDB) are two extremes: the former is extremely complex with many types and relations, whilst the latter is very simple and flat, with only two types and a single bidirectional reference relation between them. Note that for all performance graphs, the data labels show speedup relative to the original sequential EVL implementation. We report on more experiments with parallel EVL in section 5.9.

4.5.2.1 Java models

Whilst the *findbugs* script described earlier has 30 constraints, there is only a single constraint which is responsible for over 99% of the execution time. This constraint is shown in Listing 4.5.1. It is an inefficient algorithm for ensuring that all imports in all Java classes are referenced within the class.

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

Listing 4.5.1 Most demanding constraint in *findbugs* script

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.

Figure 4.5.1 shows the results for the *findbugs* script with 500K model elements. The results are disappointing to say the least: we did not exceed 5x speedup when using all 32 hardware threads with either of the atomic implementations. We see that performance scales with more threads, and SMT certainly makes a difference. Parallelisation for this script and model combination does not seem to be at all efficient. Interestingly, we see that neither of the three implementations is clearly superior here, and all perform similarly with 32 threads. This indicates a bottleneck elsewhere, since performance with fewer threads tends to vary more between the implementations.

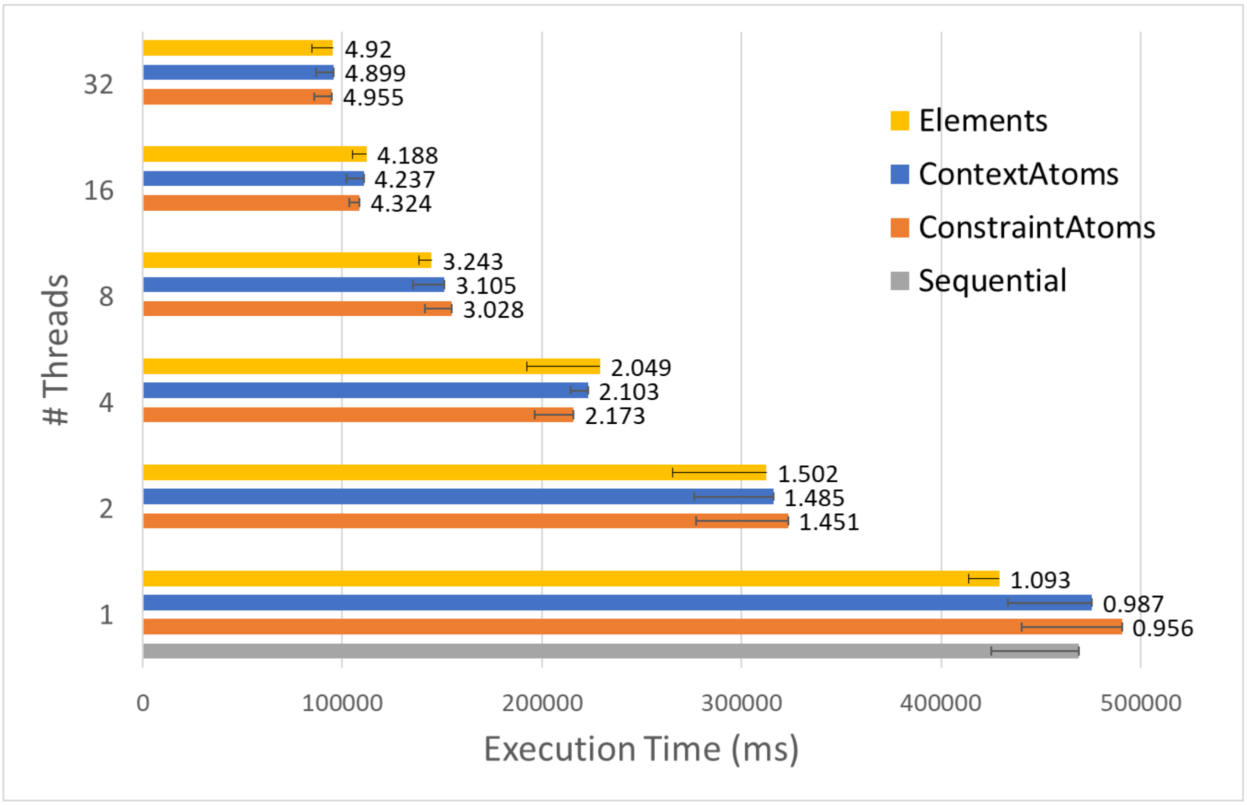


Figure 4.5.1 *java_findbugs* 0.5 million elements (TR-1950X system)

Table 4.5.1 shows the experiment with 3 million model elements. With a much larger model, the execution times are also increased, with the sequential implementation taking over two and a half hours to complete on average. Here we tested the Elements-based implementation, which executes each ConstraintContext sequentially, rather than putting all of the jobs (ConstraintContext-element pairs) into the queue all at once. The results are largely the same as before, however what is interesting is that with 32 threads, we see a big jump over 16 threads, going from just below 4x speedup to over 5.2x. This can be explained by the operating system's thread scheduling behaviour, which becomes immaterial here since all logical cores are used. Thus, any inefficiencies in allocation of software threads to hardware threads is effectively eliminated: the OS can't choose the "wrong" core which may introduce to cache misses, so data locality is improved. We also included the results for the Context-based and Constraint-based implementations, both of which perform similarly. It's also worth noting the unusually high variance in execution times for the Elements-based implementation, especially with 4 threads where it is over 22%.

Table 4.5.1 *java_findbugs* 3 million elements (TR-1950X system)

Implementation	Execute (ms)	Execute STDEV	Speedup	Memory (MB)	Memory STDEV
Sequential	9257074	663332	1	1477	2133
Elements (1)	9287420	405506	0.997	5441	1553
Elements (2)	6687856	783874	1.384	5437	1996
Elements (4)	4783562	1085070	1.935	1030	468
Elements (8)	3210668	539501	2.883	1494	493
Elements (16)	2328022	143205	3.976	2215	1131
Elements (32)	1759968	299532	5.26	1112	1707
ContextAtoms (32)	1719911	71541	5.382	159	3
ConstraintAtoms (32)	1742944	74099	5.311	198	1

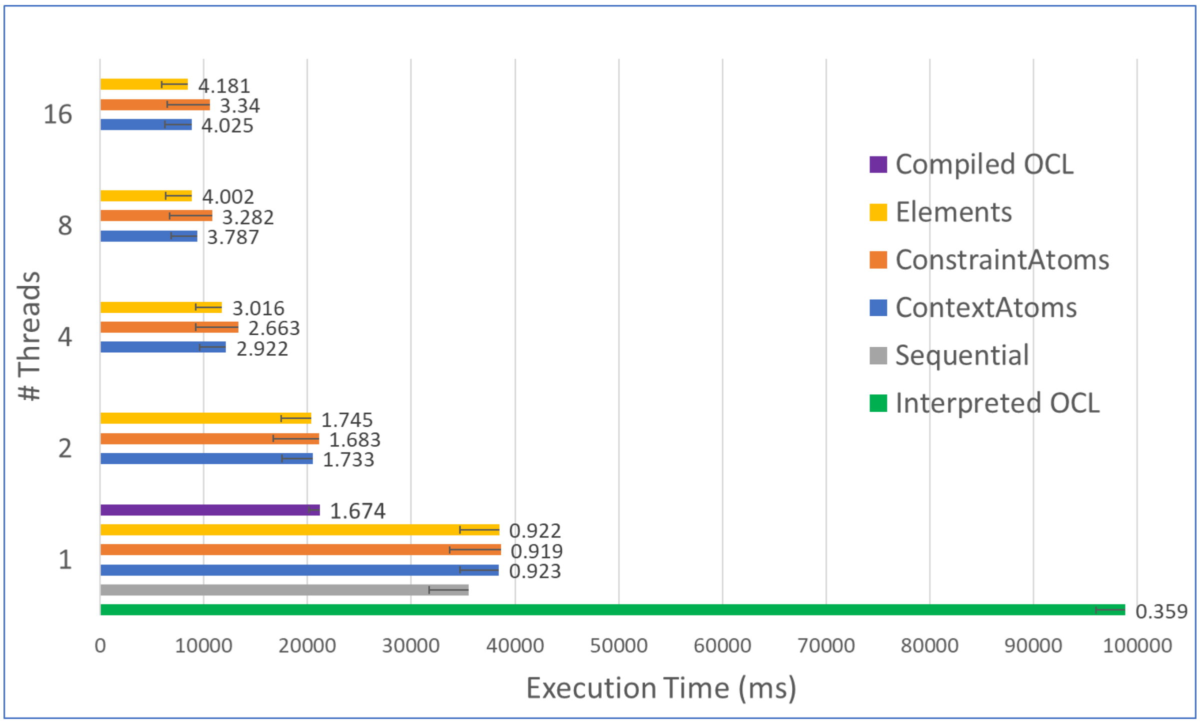


Figure 4.5.2 Results for *java_simple* script with 4.357 million elements (R7-3700X system)

In addition to testing the *findbugs* script, we also removed the most demanding constraint (i.e. the one in Listing 4.5.1) to assess the performance impact (we refer to this script as *java_simple* herein). As Figure 4.5.2 shows, even with the largest model (which has 4,357,774 model elements), the execution time for sequential EVL is less than 40 seconds on average. For context, the model took 24 seconds to load, so in practical terms the benefits of parallelisation are limited by Amdahl's Law regardless of the efficiency in this case. The main point of interest with this experiment is the large disparity between the initial and second run. For sequential EVL, the first run (where the model element type caches are uninitialized) took around 39 seconds, whereas the second run, with the caches populated, took less than 32 seconds. For parallel EVL with 16 threads, the initial run was 11 seconds whilst the warmed-up run was 6 seconds. Hence, we see quite large error bars and relatively poor speedup on our octa-core Ryzen system. However when compensating for this, comparing only the second run we see a speedup between 5.2 and 5.3x with the Elements-based implementation. The initial run fairs significantly worse, with a speedup of around 3.68x. Perhaps this is Amdahl's Law coming into play, since reading the model and populating the cache is performed sequentially in both cases.

We can also compare our results to Eclipse OCL, both interpreted and compiled. In the latter case, the validation constraints are embedded into the metamodel using *OCLinEcore*, and generated using EMF's facilities. For this program, we can see that interpreted OCL is almost three times slower than sequential EVL, however compiled OCL is approximately 4.47x faster than interpreted. Even the slowest ConstraintAtoms parallel implementation with two threads is faster than compiled OCL.

Further experiments for this program and with these models, including comparison to Eclipse OCL, are described and analysed in sections 5.9.2 and 5.9.3.

For completeness, we also experimented with the *java_equals* script shown in Listing 4.2.1 (refer to section 4.2.2 for a detailed explanation). The idea is to see how the presence of a lazy constraint which is executed via a dependency affects performance. Since there are only two constraints and the program took very little time to run in our preliminary experiments, we again used the largest model (with over 4.357 million model elements). We also wrote the program in OCL, using the *implies* operator to express the dependency and extracting the "*hashHashCode*" invariant to an operation. We have tested both the EVL and OCL implementations for equivalence.

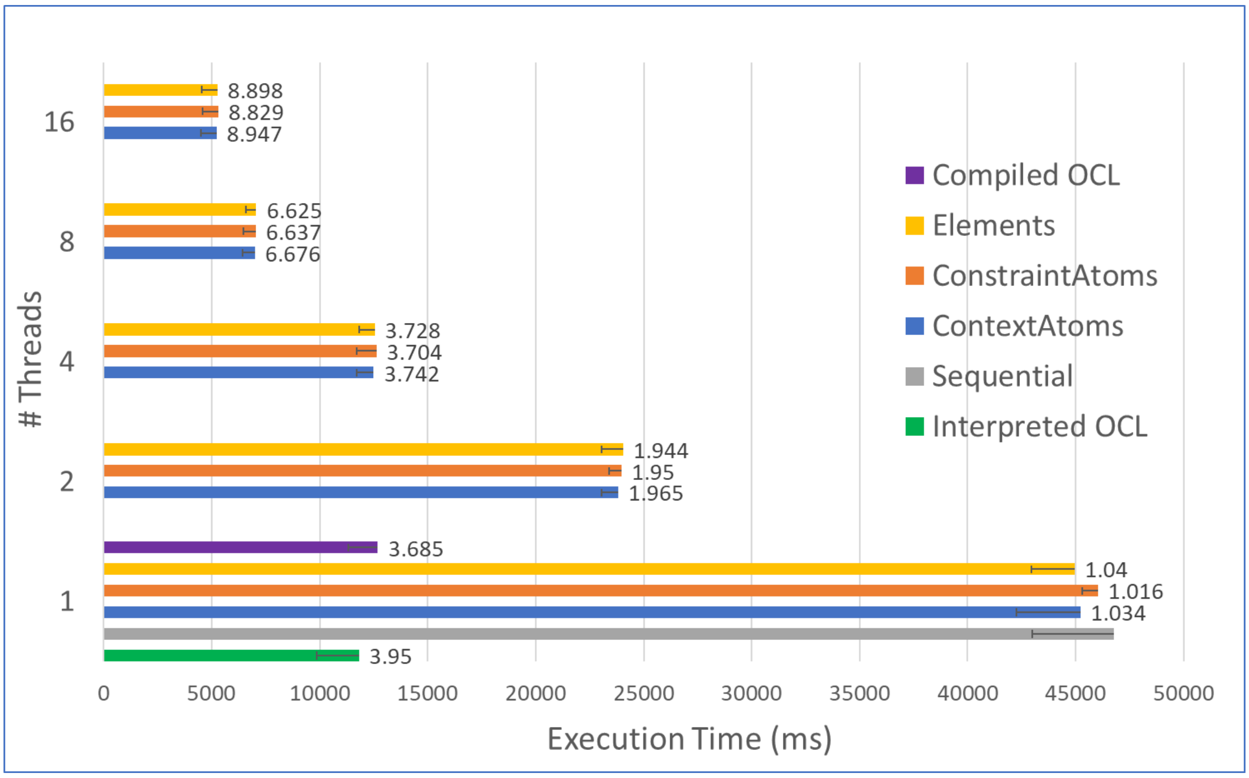


Figure 4.5.3 Results for *java_equals* script with 4.357 million elements (R7-3700X system)

The execution times are similar for the *java_simple* EVL script discussed previously. However, the relative performance of parallel EVL is superior, as shown in Figure 4.5.3. This is surprising given our approach to dealing with dependencies, which shows the extreme dependence of performance benefits on the nature of the program. We can see that all of the parallel implementations tested perform similarly, and deliver much strong results compared to those in Figure 4.5.2. With 4 or more threads, we see almost double the performance in some cases. The efficiency here is far superior for the same model, using the same hardware and software – only the program is different. The fact that we were able to achieve almost 9x speedup with an 8 core system shows that in most cases, the bottleneck really is in the nature of the validation script rather than the model or hardware. To measure the average efficiency for a given number of threads, we can take the sum of the three speedups and divide them by the number of threads multiplied by three. For example, with two threads, we see an average efficiency of $(1.944 + 1.95 + 1.965)/(2 * 3) = 97.65\%$. With 4 threads, this is 93.1%. This drops off to 83.1% with 8 threads, which isn't unexpected given the modular design of Ryzen combined with the limitations of dual-channel memory coming in to play. However it is surprising that SMT provides such a large performance gain: on average an additional 2.25x over 8 threads. Of course, we also cannot ignore the strong performance of Eclipse OCL in this benchmark, which is almost 4x faster than sequential EVL. Clearly, deeper optimisations in the evaluation of the program are performed, and perhaps the subtle difference in the way we expressed the program in OCL also has an effect. Curiously however, we can see that there is no improvement in OCL's compiled performance for this program: if anything, it is slightly slower than interpreted! It would be difficult to envisage how compiled OCL could be so much faster than sequential EVL based on the results from Figure 4.5.2, so the performance here is not unreasonable.

4.5.2.2 IMDb models

We designed a computationally expensive validation script for the IMDb metamodel, shown in Listing 4.5.2. The idea is to ensure that there are no unreferenced ("dangling") Movie or Person elements in the model. We compute a flattened cache of all Movie and Person instances in the *pre* block, so model navigation is minimised in the constraint logic, and the benchmark becomes a test of how quickly we can search collections in parallel. This allows us to measure the efficiency of our parallel implementation in its purest form, where the overhead from model navigation is minimised and the logic is as simple as possible, minimising the overhead of AST interpretation. This gives us a "best-case" scenario of the kind of performance gains we can expect from parallelisation with a flat model and very simple but expensive validation logic.

```
pre {
    var allActorsInMovies = Movie.all.collect(m | m.persons).flatten();
    var allMoviesInActors = Person.all.collect(p | p.movies).flatten();
}

context Movie {
    constraint ReferencedInActors {
        check : allMoviesInActors.contains(self)
    }
    constraint AllActorsReferencedInMovies {
        check : allActorsInMovies.containsAll(self.persons)
    }
}
```

Listing 4.5.2 *imdb_dangling* EVL script

Table 4.5.2 compares the Constraint-based and Context-based data-parallel implementations for this script with 2 million model elements. It is immediately clear that for this kind of program, the purely data-parallel (ContextAtoms) approach is superior for all thread counts, even with just a single thread. Perhaps more surprising is the fact that with 1 and 4 threads, the Context-based implementation is more efficient than the sequential approach. Whilst this may be partially attributable to margin of error, the main explanation for this and the relatively large standard deviation is attributable to the discrepancy between the initial execution and second run, with the model already cached. Also in the parallel version, the *collect* operations in the *pre* block are executed in parallel to make it a fairer comparison and limit Amdahl's Law – see section 6.2.7 for details. With sequential EVL, the execution time is very consistent between the runs, whereas parallel EVL is significantly faster in the second run than the first. We found this pattern to be the case in all five runs (10 in total). The fact that the Context-based implementation is faster than the Constraint-based one conforms to our expectations: the overhead of executing twice as many jobs (since there are two constraints) is greater than the benefit, since the execution time for each model element is relatively small. Still, the magnitude of the difference and consistency of results is slightly larger than we anticipated. In terms of speedup, the results exceed our expectations: it is surprising to see such linear scalability with 16 cores, although the system is equipped with quad-channel memory, helping to alleviate memory access bottlenecks. Things get more interesting with 32 threads. With 16 threads, it appears the two implementations are similar in performance, delivering 14x speedup over the sequential implementation. But when making full use of all hardware threads, the Context-based approach is an additional 1x faster than the Constraint-based one. This can largely be explained by the scheduling of jobs to threads: at this point all hardware threads are busy, and the thread pool has more threads to deal with, and twice as many jobs to handle compared to the Context-based implementation, hence the result.

Table 4.5.2 *imdb_dangling* 2 million elements (TR-1950X system)

Implementation	**Execute (ms)**	**Execute STDEV**	**Speedup**	**Efficiency**	**Memory (MB)**
Sequential	10216145	47139			3538
ContextAtoms (1)	10189836	87019	1.003	1.003	3579
ConstraintAtoms (1)	10228335	26183	0.999	0.999	2515
ContextAtoms (2)	5236765	33036	1.951	0.976	3661
ConstraintAtoms (2)	5591928	770997	1.827	0.913	2191
ContextAtoms (4)	2526550	388166	4.044	1.011	3336
ConstraintAtoms (4)	2696757	25494	3.788	0.947	2288
ContextAtoms (8)	1358233	249964	7.522	0.94	2357
ConstraintAtoms (8)	1589881	262634	6.426	0.803	2498
ContextAtoms (16)	725357	104894	14.084	0.88	2625
ConstraintAtoms (16)	732000	127906	13.956	0.872	2658
ContextAtoms (32)	666356	182355	15.331	0.479	2999
ConstraintAtoms (32)	709888	168348	14.391	0.45	3064

In Figure 4.5.4, we tested the IMDb script with a smaller model (500 thousand elements), and added in our default Elements-based implementation. As a reminder, this implementation differs from the Context-based one in that all ConstraintContexts are executed sequentially; that is, all active jobs will be from the same ConstraintContext. We can see that all implementations scale almost identically with more threads. The only notable exception is with 8 threads, where for some reason the Elements-based implementation is significantly slower than even the Constraint-based one. The gap between the Constraint-based and Context-based implementations is also more pronounced. We are not sure why performance varies so dramatically with 8 threads, as shown in both sets of results (2 million and 0.5 million model elements) for this script. We know that the CPU cores are spread across two dies, and we used the NUMA-aware JVM option. However, it still doesn't explain why the scheduling of jobs onto threads would be different between each of these implementations.

An interesting takeaway from these sets of results is that the parallel implementations are sometimes more efficient than the sequential one (i.e. speedup is greater than the number of threads). Whilst it may seem that the only explanation is margin of error, there is a tangible explanation. Recall that the parallel implementation uses different data structures, not only for the model cache but also for the results (UnsatisfiedConstraints). These data structures may be significantly more efficient in the case of this program – for example, we use a ConcurrentLinkedQueue in the parallel implementation whereas the sequential uses HashSet for the results. Similarly, for the model cache, our bespoke ConcurrentLinkedQueue and ConcurrentHashMap may be more efficient than ArrayList and HashMap.

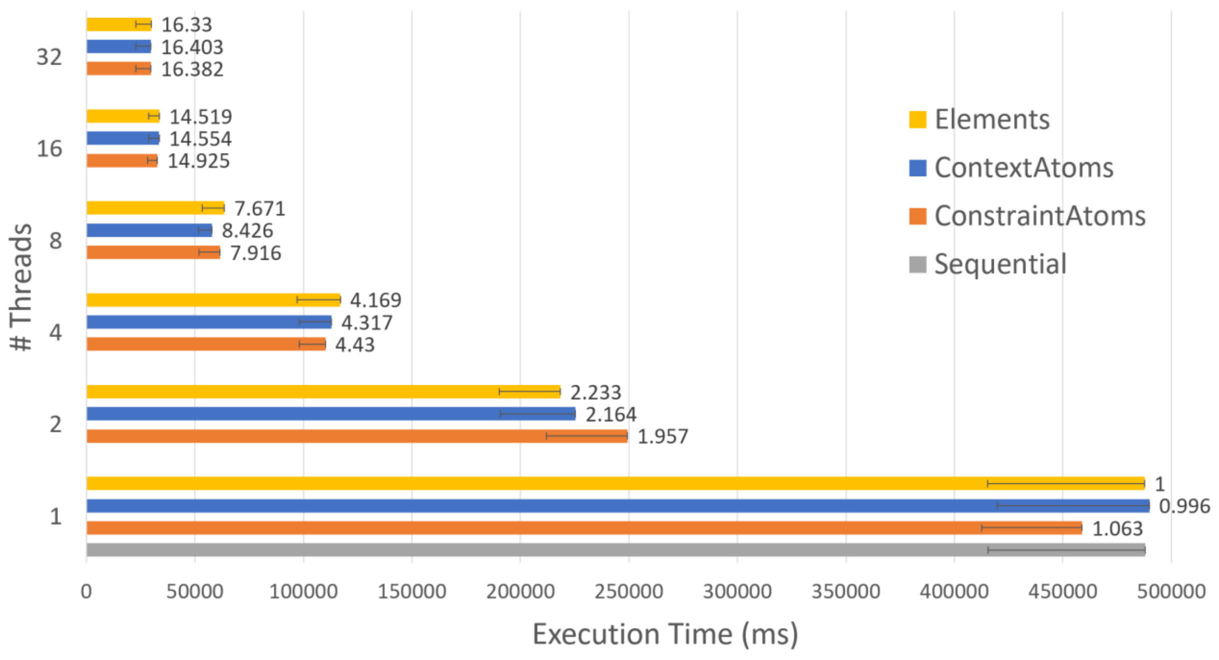


Figure 4.5.4 *imdb_dangling* 0.5 million elements (TR-1950X system)

4.5.2.3 DBLP model

Finally, we experimented with a large model of the DBLP bibliography, which has 5,654,916 elements and is 1.2 GB in its serialized form. The structure of the model is flat, similar to IMDb models but with different type names, so for example Actors are Authors, and Movies are Publications. Our validation script is quite long but not particularly complex, consisting of 7 invariants. One of these is a purely imperative algorithm for checking that the ISBN of Book instances are valid. There are two relatively demanding constraints, again inspired by the source of the model from the LinTra project [247], which finds authors who have previously published in a specific journal that have stopped publishing after a certain period. We simply placed this query into a validation context, since we are interested in evaluating our infrastructure with various models and programs: the specifics of what the program tries to do (i.e. the interpretation of the output, its usefulness in the context of the domain) is not our concern.

In our adaptation, we used a lazy constraint which is invoked by another constraint as a dependency. These two constraints are the most computationally expensive, so it is a good way to measure the performance impact of our constraint trace optimisation strategy for dealing with dependencies. The remaining constraints not discussed are trivial validations of model element attributes, such as making sure the month of a publication is one of 12 valid values.

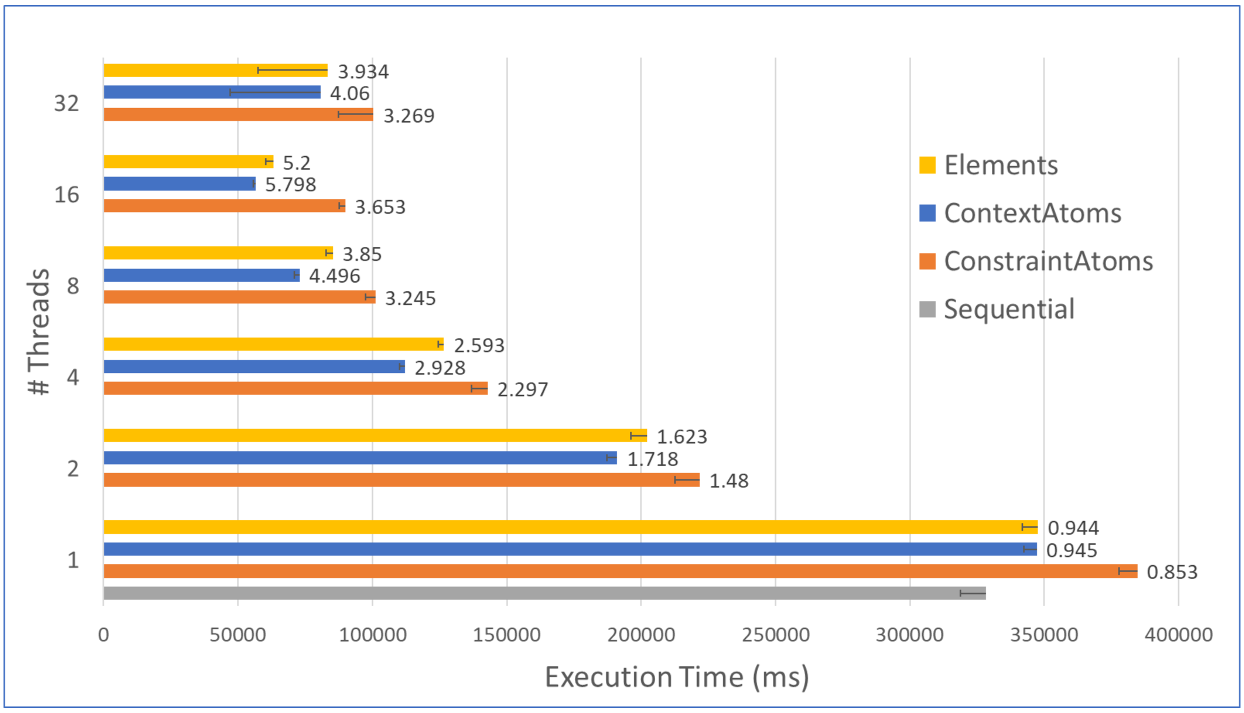


Figure 4.5.5 Results for *dblp_isbn* script with 5.65m elements (TR-1950X system)

Figure 4.5.5 shows the results for this program and model combination and contains some surprising revelations. Firstly, it conforms to our hypothesis that the ConstraintAtom implementation is inferior to the purely data-parallel one, as with the previous experiments. The differences here though are more pronounced and well beyond the margin of error, allowing us to conclude more confidently that a purely data-parallel approach is superior. Secondly, although we expected the difference between the Elements-based and ContextAtom-based implementations to be very minor, given they are both data-parallel approaches, this experiment has highlighted a more significant difference than the previous ones, which leads us to conclude that the strategy of pre-computing the jobs list and submitting them all at once to the ExecutorService is superior to the more sequential approach of the Elements-based algorithm. Thirdly, and perhaps most surprising, is that with 32 threads, the variance in execution time increases drastically, as shown by the error bars. This is unexpected, given the consistency of performance with fewer threads. We can see that the ContextAtom-based implementation is not only the fastest in all cases, but also the most consistent, with the smallest standard deviation (albeit by a very narrow margin). When moving up to 32 threads, we see that for all implementations, not only does performance degrade significantly over 16 threads, but that the variance for all implementations increases, making the ConstraintContextAtom implementation the most volatile with this many threads where previously it was the most consistent.

We also see that the peak speedup is quite poor – only 5.8x with the ConstraintContextAtoms and 16 threads. This is perhaps partly due to our strategy for dealing with dependencies, which may lead to the Constraint depended on being executed twice in the worst case for a given element. However judging by the results from *findbugs*, we can see that this is not an outlier case, and there are certainly more influential factors. What is difficult to explain is the dramatic degradation in performance and increase in variance when moving from 16 to 32 threads, since this was not the case in the IMDb experiments. Clearly there is a memory-related or thread scheduling (at the operating system level) bottleneck at play here, or maybe the *UseNUMA* flag has a detrimental effect. This all goes to show how sensitive the performance benefits of parallelisation are not only to the environment (hardware, operating system, JVM parameters etc.) but also the nature of the EVL program and the model(s) involved.

We ran this program on the old E5520 system, which consists of two quad-core Hyperthreaded CPUs. The results are even more surprising, as shown in Figure 4.5.6.

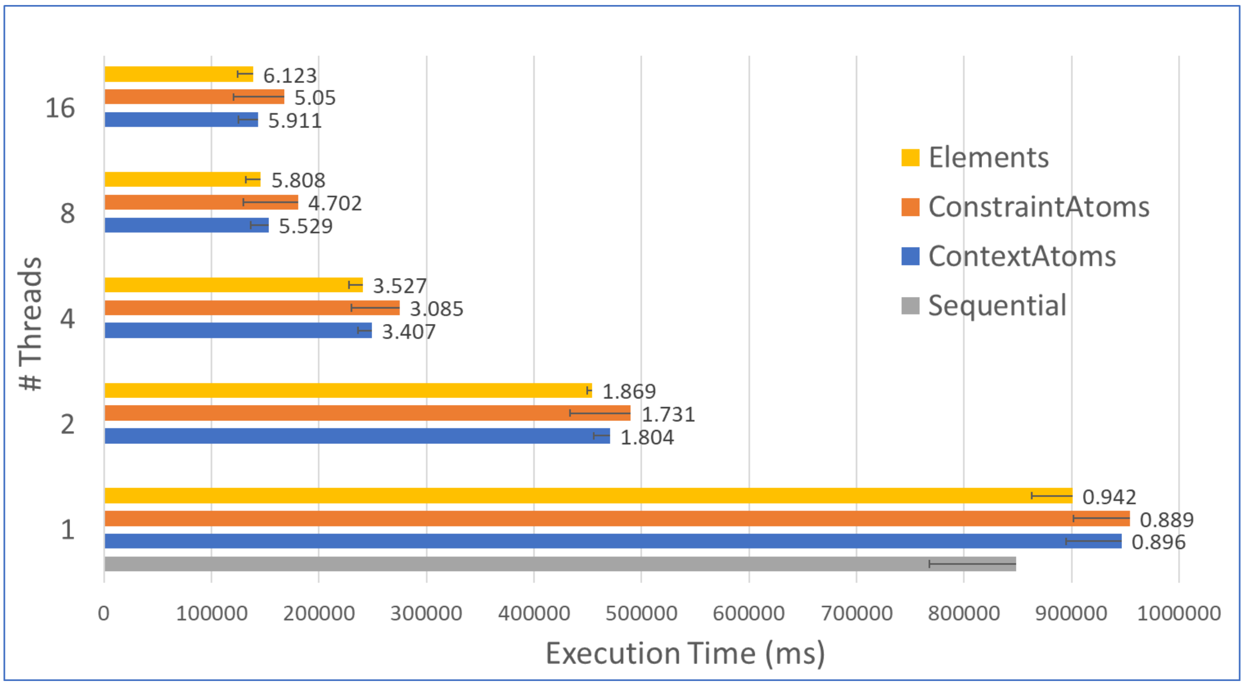


Figure 4.5.6 Results for *dblp_isbn* script with 5.65m elements (E5520 system)

Here we see that the more linear Elements-based execution algorithm outperforms the “all-at-once” ConstraintContextAtom-based implementation. Furthermore, it also has the least variance, judging by the error bars. Meanwhile, the ConstraintAtom-based implementation is clearly the worst performer. This is no surprise based on the previous experiments, however notice also that the variance in execution time is greater than the other implementations. Finally, we see that efficiency drops off sharply beyond four threads; where we observe a surprisingly strong 3.5x speedup with the Elements-based implementation. The sharply diminishing speedup is easily explained by the fact that we are dealing with a dual-socket CPU, so inevitably there is a greater communication cost due to NUMA with more than 4 threads. Since these CPUs have Hyperthreading, the extent of this depends on how the operating system chooses to allocate software threads to logical cores: whether with 8 threads we see a clean split of the workload between the two physical CPUs, or whether the load is placed on a single CPU. We would expect that both CPUs would be fully utilised with 16 threads, however as the results show, there is little gain in performance when going from 8 to 16 threads. Still, assuming all logical cores across both CPUs were in use, the Elements-based implementation achieves over 6.1x speedup, whilst the ConstraintAtom-based implementation only manages 5x. Notice also how the performance difference between the implementations widens with more threads. The discrepancy in results between Figure 4.5.5 and Figure 4.5.6 can be explained not only by the choice of hardware, but also the JVM and version of the Linux kernel.

4.6 Summary

In this chapter, we have presented a scalable data-parallel framework for a complex hybrid (declarative / imperative) model validation language. This framework includes solutions to concurrency challenges which not only apply to model validation but to other rule-based model management tasks. On top of this, we also designed a generalisable approach for distributed model validation, which we argue is also generalisable for other read-only model management tasks. Finally, we evaluated our solutions with a comprehensive test suite and detailed performance experiments which showed that our parallel and distributed approach scales with more processors and model elements, as expected based on the hypothesis.

Our parallel and distributed architecture is constructed with the Epsilon Model Validation language (EVL) as the focus, for reasons outlined in the previous chapter. The structure of an EVL program is a setup ("*pre*") and post-processing ("*post*") block which may contain arbitrary imperative code and are executed before and after the declarative rules. Each rule in EVL is specified in the context of a model element type, which acts as the domain from which the collection of data the rule will be applied to is drawn from. Each rule (in the case of EVL, a "*constraint*") can further filter elements from the domain, again with arbitrarily complex imperative code, before finally executing the main rule body. This body is a predicate (which can again be expressed using arbitrarily complex imperative code) for which each element is executed against and if the result is false, the rule-element pair is added to the global set of results.

Despite the complex language features and imperative constructs of EVL, and the fact that rules may have dependencies on each other, we have been able to demonstrate that each rule-element pair can be executed independently in parallel, making each EVL program scalable with as many processing cores as there are rule-element pairs. Along the way, we have identified many engineering challenges and provided efficient solutions based on highly optimised design, data structures and algorithms. Most notably, these include the use of serial thread confinement for inherently sequential aspects of the execution engine, as well as careful choices regarding shared mutable and immutable state such as caches and results.

We equivalence-tested our implementation against sequential EVL and Eclipse OCL with a large and comprehensive test suite, which was able to identify several obscure concurrency bugs even at late stages in the project. Our performance evaluation showed that our solution has potential to scale linearly with the number of cores in a system (achieving over 16x speedup on a 16 core / 32 thread CPU), however the extent of this is highly dependent on the nature of the program, where memory access and other bottlenecks come into play. We saw that performance improvements can be as poor as 5x speedup with 16 cores. In the next chapter, we will see how this infrastructure and parallel decomposition can be further developed to achieve significantly greater scalability.

5 Distributed Model Validation

In Chapter 4, we provided scalable data-parallel algorithms for EVL. Although we addressed the issues with concurrent execution in shared-memory systems, we did not show how our strategies can scale with multiple physical computers. In this chapter, we show how the parallel decomposition developed in the previous section can be realised on a multi-process (as well as multi-threaded) architecture. By virtue of being Java-based, this approach works with heterogenous architectures where participating computers may have completely different instruction sets, hardware and be located on distant networks.

5.1 Distribution challenges

Theoretically, any problem which can be parallelised can also be distributed. However the key difference between shared-data parallelism and distribution (amongst others [130]) is *state independence*. In other words, since each computer is physically different in its location, CPU, memory etc. then the only means of communication is through a network. By contrast, a shared-memory multi-core system can achieve parallelism within the same software process since all hardware is shared, and each thread can access the same data through. In a distributed system, sharing data is extremely expensive, as it requires transmission over a separate physical connection (e.g. through Ethernet or Wi-Fi) using different protocols. A high-performance distribution algorithm should therefore minimise communication (i.e. the amount of data) traffic between computers (herein referred to as “nodes”).

A corollary of the lack of shared memory is that any data which is transmitted between nodes needs to be *serializable* – that is, the data should be easily decomposable into (and reconstructable from) bytes. In practice, this means that all objects need to be flattened into primitive forms (Strings, integers, booleans, bytes etc.). Therefore we cannot directly re-use our atom-based approach from the previous section because each atom refers to non-serializable objects. For example, a ConstraintContext has references to its Constraints, its *ModelElementType*, the parent *EvlModule* etc. whilst a model element is an arbitrary object. In most cases, the model element object has references to its children and parent, as models are typically (but not always) graph-like structures. An efficient and generalisable approach is needed to achieve data-parallel distribution.

Finally, distribution is significantly more complex than single-machine parallelisation due to the co-ordination overhead and uncertainty involved in communication between (potentially heterogenous) nodes. In a shared-memory system, mapping jobs to threads is a relatively simple task: all threads are essentially identical and part of the same process. If a job fails, we can gracefully stop other threads and obtain any thread-local data for error reporting and diagnosis. By contrast, mapping jobs to nodes is not so simple, because each node is executing in a separate process on a different computer. Transporting data from one node to another is not always reliable, as there may be intermittent network failures or the target node could be disconnected / shut down etc. In other words, there are many more assumptions involved and the lack of shared state means execution is not easy to debug. The load-balancing is also more complex in distributed systems. In shared-memory parallelism, it is trivial to determine whether a thread is busy, whereas with distributed parallelism, each computer may have different processing capabilities (different number of cores, clock speeds, architecture, memory speed etc.) and so feedback from nodes is needed. Obtaining this feedback regularly can be costly due to the additional network traffic.

5.2 Serialization strategies

In this subsection we address the most immediate challenge posed by distributed execution; namely how to efficiently send data to and from slave nodes (herein referred to as “workers”). We propose two approaches, however in both cases we assume that the workers have knowledge of the model(s) involved in execution – i.e. they are able to load individual model elements into memory. We also assume that workers have full access to the program under execution (that is, the EVL script and its dependencies). Ensuring that each node has their own copy of the required resources is trivial and not a focus of the contributions.

The premise of a distributed execution strategy is in theory quite straightforward now that we have (multiple) atomic parallel decompositions. The master (we will discuss the master-slave architecture in a later section) sends jobs to workers for execution, workers execute these jobs and send back the results. Since we only need to report Constraint-element pairs which were unsatisfied as part of the results (in other words, the *ConstraintTrace* is not required), only serializable instances of *UnsatisfiedConstraint* are required. An UnsatisfiedConstraint consists of a Constraint, 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). Supporting fixes is beyond the scope of this work.

For simplicity and for reasons discussed in the previous chapter, we base our distribution strategies on the *ConstraintContextAtom* decomposition – that is, at the data-parallel level of granularity rather than Constraint-element pairs. However, our approach can also accommodate the *ConstraintAtom* decomposition if required, although we see no reason to support both for the purposes of this work.

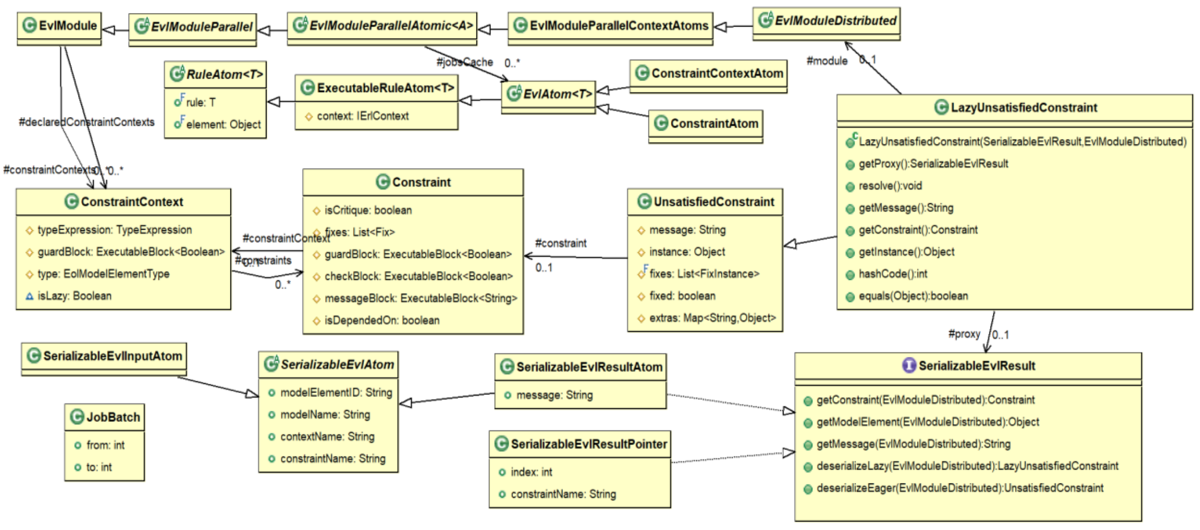

EvlModule
EvlModuleParallel
EvlModuleParallelAtomic<A>
EvlModuleParallelContextAtoms
EvlModuleDistributed
#jobsCache 0..*
#module 0..1
RuleAtom<T>
rule: T
element: Object
ExecutableRuleAtom<T>
context: IErlContext
EvlAtom<T>
ConstraintContextAtom
ConstraintAtom
LazyUnsatisfiedConstraint
LazyUnsatisfiedConstraint(SerializableEvlResult,EvlModuleDistributed)
getProxy():SerializableEvlResult
resolve():void
getMessage():String
getConstraint():Constraint
getInstance():Object
hashCode():int
equals(Object):boolean
#declaredConstraintContexts
#constraintContexts 0..* 0..*
ConstraintContext
typeExpression: TypeExpression
guardBlock: ExecutableBlock<Boolean>
type: EolModelElementType
isLazy: Boolean
#constraintContext
#constraints 0..1 0..*
Constraint
isCritique: boolean
fixes: List<Fix>
guardBlock: ExecutableBlock<Boolean>
checkBlock: ExecutableBlock<Boolean>
messageBlock: ExecutableBlock<String>
isDependedOn: boolean
#constraint 0..1
UnsatisfiedConstraint
message: String
instance: Object
fixes: List<FixInstance>
fixed: boolean
extras: Map<String,Object>
#proxy 0..1
SerializableEvlInputAtom
JobBatch
from: int
to: int
SerializableEvlAtom
modelElementID: String
modelName: String
contextName: String
constraintName: String
SerializableEvlResultAtom
message: String
SerializableEvlResultPointer
index: int
constraintName: String
SerializableEvlResult
getConstraint(EvlModuleDistributed):Constraint
getModelElement(EvlModuleDistributed):Object
getMessage(EvlModuleDistributed):String
deserializeLazy(EvlModuleDistributed):LazyUnsatisfiedConstraint
deserializeEager(EvlModuleDistributed):UnsatisfiedConstraint

Figure 5.2.1 Relevant data types in distributed EVL

5.2.1 **Atom-based**

Assuming that all workers have access to the script and model(s), we can convert the rule-element data atoms introduced in section 4.4.4 into efficiently serializable forms. We exploit the fact that the Epsilon Model Connectivity layer supports the notion of IDs for model elements. A subset of EMC's functionality is shown in Listing 5.2.1, which also includes the resolution and lookup of elements by identifiers. Although such support is optional, the ability to retrieve an element by a unique (String) identifier and to obtain an element's ID is supported by EMF, which is arguably the most commonly used modelling technology in MDE applications. Thus we can replace the model element field in a ConstraintAtom or ConstraintContextAtom with its String identifier. As for the ConstraintContext itself, we can use its type name, which consists of an optional model name and a model element type. As this is also a String and each worker has a copy of the program, we can simply substitute the rule (ConstraintContext) with its name and resolve it on the worker. The same approach can be used for ConstraintAtom, where we include the ConstraintContext name and the Constraint name. For serialization of UnsatisfiedConstraints, since we have already shown how to serializable model elements and constraints, the addition of a String message is trivial, so we can simply extend the input datatype to add this field. In Figure 5.2.1, the serializable job inputs for this strategy are based on *SerializableEvlAtom*. Note how the same data structure can represent both the inputs (jobs) and outputs (UnsatisfiedConstraint); albeit the latter requires an extension to include the message as shown by *SerializableEvlResultAtom*. Also note that the "constraintName" field will be null (unset) in the ConstraintContextAtom decomposition for inputs, since we are only interested in identifying model elements and their associated types.

Although the serializable variant of the atom-based job distribution strategy appears to be efficient, the size of each atom (in bytes) is still relatively large when considering that there will be millions of them to distribute. Furthermore, looking up individual model elements by ID can be expensive, even if the modelling technology keeps a cache of ID to element mappings. There is also the cost of finding individual Constraints and ConstraintContexts every time. Moreover, we are assuming that the modelling technologies used in the program all support IDs, which is not always the case.

```
public interface IModel {

  Collection<?> allContents();

  Object getElementById(String id);

  String getElementId(Object instance);

  Collection<?> getAllOfType(String type) throws
      EolModelElementTypeNotFoundException;

  Collection<?> getAllOfKind(String kind) throws
      EolModelElementTypeNotFoundException;

  Object getTypeOf(Object instance);

  boolean knowsAboutProperty(Object instance, String property);

  String getTypeNameOf(Object instance);
```

Listing 5.2.1 A small subset of *IModel* functionality

5.2.2 **Index-based**

We can further exploit our assumptions regarding the availability of resources on workers and the job creation algorithm. When parsing an EVL program, the order of rules is deterministic, so when looping through ConstraintContexts and Constraints in the program, we can guarantee a consistent ordering. If the modelling technology's implementation of *getAllOfKind* is also deterministic, then the entire program's job creation is deterministically ordered. In other words, if we were to execute these jobs sequentially in a single thread, the order of execution would be the same every time for a given model and EVL script. Since the number of model elements to be checked and the number of Constraints is finite and ordered, we can obtain a job **list**, which can be easily and efficiently constructed by all nodes after parsing the program and loading the model(s). For more on this, refer back to section 4.4.4.

5.2.2.1 Batching

Since we can assume all participating nodes are able to construct the job list and that this list is deterministic, it is not necessary to distribute any atoms; nor do we need to rely on the modelling technology supporting IDs. We can instead assign each node a subset of the list to evaluate using indices to specify the sublist (see *JobBatch* in Figure 5.2.1). This approach is much more efficient because each job consists of only two integers specifying the start and end index of the list, which is significantly smaller in size than serializing the atoms, which consist of potentially long Strings. As a result, the size of each job is constant and helps to ensure a more consistent flow of network traffic, as well as reducing the serialization overhead. Moreover, this approach allows us to directly control the granularity of jobs and thus the volume of traffic. This is achieved by adjusting the size (i.e. the difference between the start and end index) of each job (herein referred to as “batch”). One extreme is to divide the job list into as many batches as there are jobs, which allows for maximum granularity. The batching algorithm is shown in Listing 5.2.2. The *getBatches* function creates contiguous batches (where *from* is the start index and *to* is the end index) based on a given total number of jobs and the desired size of each batch (i.e. the difference between *to* and *from*), referred to as *chunks*. Thus, there will be $totalJobs\ /\ chunks$ number of batches, possibly with an additional batch containing the remainder if *totalJobs* is not exactly divisible by *chunks*.

The batch granularity (i.e. the number of jobs represented by each batch) 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 can be set as either a percentage 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), or a fixed (absolute) value if greater than or equal to 1. By default, this is set to the number of logical cores on the master; assuming the master has at least as many cores as workers. This maximises CPU utilisation at the finest level of granularity. To see why, consider the (simplified) execution algorithm of a batch in Listing 5.2.3 and imagine that this block of code is called each time a worker receives a batch. Since indices represented by a batch are contiguous, we start at the *from* index and finish at the *to* index. The numbers between these ranges represent the location of the ConstraintContext-element pairs in the job list to be executed. Since the list is built and cached (and backed by an array), accessing each atom is a constant-time operation. We simply retrieve these atoms, and submit them to the *ExecutorService*.

To provide some context, it is useful to know that when a worker receives a job, it can execute it independently from other jobs in parallel, just as with parallel EVL. 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 job to a separate thread and evaluate them in parallel. If however it receives a batch containing one job (e.g. index from 0 to 1), then execution will effectively be single-threaded. In effect, the batch size contains within it the parallelisation information: in other words, we do not need to rely on asynchronous execution of jobs when we send them. Note that this assumes the distribution framework waits for jobs to complete before sending them the next. 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. If the distribution is asynchronous (i.e. a worker can accept multiple batches at a time), then there is no need for batching: we could simply set the batch factory to 1 so that each batch represents a single job and rely on the distribution framework to do the parallelisation. Assuming we are using our own parallel EVL infrastructure (i.e. we control our own ExecutorService), then we can also explicitly set the parallelism of workers to a fixed value if needed (we will see why this is useful later on), rather than setting the batch factor equal to the number of hardware threads.

To further clarify the effect of the batch factor on the total number of jobs, consider Figure 5.2.2, which shows an example where the batch factor is set to 3. Refer back to Figure 4.4.1 for more context on this illustration.

```
public class JobBatch implements java.io.Serializable, Cloneable {

  public int from, to;

  public static List<JobBatch> getBatches(int totalJobs, int chunks) {
    assert totalJobs >= 1 : "Must have at least one job";
    assert chunks >= 1 : "Batch size (chunks) must be at least 1";

    final int
      modulo = totalJobs % chunks, division = totalJobs / chunks,
      batches = modulo > 0 ? 1 + division : division;

    ArrayList<JobBatch> resultList = new ArrayList<>(batches);

    for (int prev = 0, curr = chunks; curr <= totalJobs; curr += chunks) {
      resultList.add(new JobBatch(prev, prev = curr));
    }

    if (batches > 0 && modulo > 0) {
      resultList.add(new JobBatch(totalJobs - modulo, totalJobs));
    }

    assert resultList.size() == batches : "Expected number of batches met";
    assert resultList.get(batches-1).to == totalJobs : "All jobs retained";

    return resultList;
  }

}
```

Listing 5.2.2 Batch-based distribution data structure and splitting algorithm

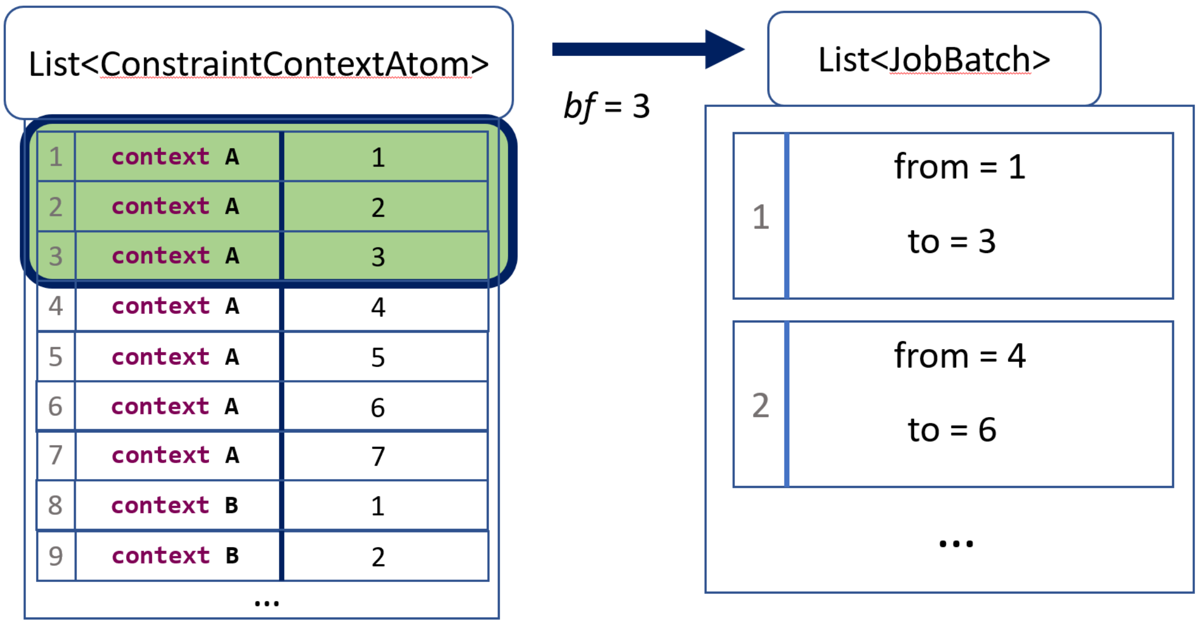


Figure 5.2.2 Batching example with *batch factor* set to 3

```
List<Callable<?>> executorJobs = new ArrayList<>(1+batch.to-batch.from);
for (int i = batch.from; i < batch.to; i++) {
    ConstraintContextAtom cca = jobList.get(i);
    executorJobs.add(cca);
}
evlContext.executeAll(executorJobs);
```

Listing 5.2.3 Simplified execution algorithm for *JobBatch*

5.2.2.2 Results

As for serialization of the results, we can similarly exploit the deterministic ordering of jobs. If we were splitting at the ConstraintAtom granularity, we could simply send back the indices of the constraint-element pairs which were unsatisfied. The master could then construct the UnsatisfiedConstraint by retrieving the ConstraintAtom from the job list, which contains a reference to the Constraint and model element. As with the input jobs, this has the advantage that the modelling technology does not need to support persistent (i.e. non-volatile) model element identifiers. Thus, we can avoid the potentially expensive process of resolving model elements by ID (and vice-versa) for each model element to be validated and/or resolved. Instead we can simply look up the item in the list. This simplicity is demonstrated in Listing 5.2.4.

Adapting this to the ConstraintContextAtom granularity is slightly trickier since each job (i.e. ConstraintContext-element pair) may output multiple UnsatisfiedConstraints, so we need a way to identify each one. We create a new data type (known as *SerializableEvlResultPointer* in Figure 5.2.1) 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. 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 deserialization process (performed on the master).

```
public Object modelElementAtIndex(int index) throws EolRuntimeException {
  return getAllJobs().get(index).element;
}

public int indexOfModelElement(Object element) throws EolRuntimeException {
  Iterator<? extends RuleAtom<?>> iter = getAllJobs().iterator();
  for (int i = 0; iter.hasNext(); i++) {
      if (iter.next().element == element) return i;
  }
  return -1;
}
```

Listing 5.2.4 Index-based resolution of model elements

5.3 Preparation

Distributed execution begins from the *master*; 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. 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). Thus, to maximise efficiency the execution sequence of events needs to be revised compared to single-node EVL. Furthermore, additional steps must be taken to co-ordinate the execution between workers and the master, which are heavily dependent on the implementation technology used to facilitate communication. The order of steps in preparation for distributed execution are as follows:

1) The master must establish communication channel with workers (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)
    c. The execution engine must be initialized
    d. The *pre* block of the EVL program must be executed
4) All workers must report when they are ready to begin processing jobs.

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 steps all the steps in (3), and the configuration is known in advance to the master, it makes sense to perform this asynchronously 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. Again, the details of this are dependent on the communication technology.

Of course, this requires the program configuration to be serialized. Since all workers have access to the required resources (i.e. their own local copy), the master need only to send pointers to these resources. Concretely, this is realized as a serializable Map, where the keys are String constants referring to the configuration data held by the corresponding value. This 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, what execution strategy to use 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. different operating systems or directory structures), some substitution is required for paths. We use the notion of a *base path* as an additional variable to relativize paths, so absolute paths are converted into relative paths, with the absolute paths being resolved by each worker.

Each worker is started with nothing more than its *base path* variable and information required to locate the master. When a worker connects to the master, it then receives its configuration data, and begins replicating the configuration of the master (i.e. loading the script and model(s)). Once this has been performed, an acknowledgement must be sent to the master – again this may be implicit, depending on the implementation technology. The worker can then begin accepting jobs.

5.4 Job Assignment

In this section, we outline further optimisations to maximise the efficiency of our approach according to the environment and parameters.

5.4.1 Exploiting information on number of workers

Before the master starts sending jobs to workers, 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.

Generally, 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 $1/(1 + \text{number of workers})$. 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, and the fact that the master is continuously processing results as they arrive (the details of this are discussed in section 5.5).

```
public abstract class EvlModuleDistributedMaster extends EvlModuleDistributed {

  final JobSplitter<?, ?> jobSplitter;

  @Override
  protected final void checkConstraints() throws EolRuntimeException {
    Collection<?> masterJobs = jobSplitter.getMasterJobs();
    Collection<? extends Serializable> workerJobs = jobSplitter.getWorkerJobs();

    if (masterJobs.isEmpty() && workerJobs.isEmpty()) {
      return;
    }
    else if (masterJobs.isEmpty() && !workerJobs.isEmpty()) {
      executeWorkerJobs(workerJobs);
    }
    else if (!masterJobs.isEmpty() && workerJobs.isEmpty()) {
      executeMasterJobs(masterJobs);
    }
    else {
      CheckedEolRunnable masterAsync = () -> executeMasterJobs(masterJobs);
      CheckedEolRunnable workerAsync = () -> executeWorkerJobs(workerJobs);
      try {
        CompletableFuture.runAsync(masterAsync)
          .thenCombine(
            CompletableFuture.runAsync(workerAsync),
            (v1, v2) -> null
          )
        .get();  // This is the blocking part
      }
      catch (InterruptedException | ExecutionException ex) {
        EolRuntimeException.propagateDetailed(ex);
      }
    }
  }
}
```

Listing 5.4.1 Asynchronous execution of master and worker jobs

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 the *CompletableFuture* [199] to pipeline these two tasks and wait for both to complete. The implementation becomes relatively trivial thanks to this API, as shown in Listing 5.4.1.

Furthermore, if we know the number of workers, we can better optimise the granularity of jobs. For example, in the extreme case where there is only one worker and the computational complexity of jobs is perfectly balanced, we can split the workload into two batches: one for the master and one for the worker. By default, 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 $Nj \ * \ (b/w)$ (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. We apply a correction to this value dividing it by the number of workers to ensure there will be at least one job per worker.

If the number of workers is not known in advance (or the master proportion is unspecified), then jobs are dynamically assigned to the master from the distribution framework. However, the implementation must check whether it is executing on the master so that the model(s) and program loaded into memory can be re-used, rather than spawning a separate process. Thus, if the master's jobs are dynamically assigned (that is, the master proportion parameter is 0) then the initial registration step usually taken by standalone workers on a separate process must be bypassed.

5.4.2 **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 – and without knowing the number of workers in advance, it is difficult to determine an optimal batch size. However, we cannot assume that the computational cost of jobs is uniformly distributed across the job list. 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. Hence, the optimal default is to set the batch size equal to the maximum number of hardware threads across all participating computers (but no larger) as discussed in section 5.2.2.1. 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.

5.4.3 Job splitting strategy

To demonstrate the strategies and concepts used in distinguishing between master and worker jobs, consider the base *JobSplitter* class in Listing 5.4.2, which shows how jobs are divided between master and workers based on a simple percentage. Note that worker jobs must be serializable, whilst master jobs can be in their original form. In the atom-based approach, *T* would be *ConstraintContextAtom* whilst *S* would be *SerializableEvlInputAtom*.

```java
public abstract class JobSplitter<T, S extends Serializable> {

  protected final EvlContextDistributedMaster context;
  protected final double masterProportion;
  protected final boolean shuffle;
  protected Collection<S> workerJobs;
  protected Collection<T> masterJobs;

  protected void split() throws EolRuntimeException {
    List<T> allJobs = getAllJobs();
    if (shuffle) Collections.shuffle(allJobs);

    int numTotalJobs = allJobs.size();
    int numMasterJobs = (int) (masterProportion * numTotalJobs);
    if (numMasterJobs >= numTotalJobs) {
      masterJobs = allJobs;
      workerJobs = Collections.emptyList();
    }
    else if (numMasterJobs <= 0) {
      masterJobs = Collections.emptyList();
      workerJobs = convertToWorkerJobs(allJobs);
    }
    else {
      masterJobs = allJobs.subList(0, numMasterJobs);
      List<T> workerSubset = allJobs.subList(numMasterJobs-1, numTotalJobs);
      workerJobs = convertToWorkerJobs(workerSubset);
    }
  }

  protected abstract List<T> getAllJobs() throws EolRuntimeException;

  protected abstract Collection<S> convertToWorkerJobs(Collection<T> jobs)
      throws EolRuntimeException;
```

Listing 5.4.2 Abstract job splitting between master and workers

Listing 5.4.3 shows how this splitter is extended to work with batches. Note that both the master and workers use the same data structure (the *JobBatch* – see section 5.2.2) so no conversion between the local and serializable forms is needed. We only need the job granularity (*batchSize*) to determine the number of jobs per batch. Notice how this batching approach can be applied to any list regardless of the type. Here we assume that the job list is based on ConstraintContext-element pairs (i.e. ConstraintContextAtoms), but this could be Constraint-element pairs as well – anything which can be reproduced in identical order to the master on all workers.

```
public class BatchJobSplitter extends JobSplitter<JobBatch, JobBatch> {

  protected final double batchSize;

  @Override
  protected List<JobBatch> getAllJobs() throws EolRuntimeException {
    List<ConstraintContextAtom> jobList = context.getModule().getAllJobs();
    final int numTotalJobs = jobList.size(), chunks;
    if (this.batchSize >= 1) {
      chunks = (int) batchSize;
    }
    else {
      final int adjusted = Math.max(context.getDistributedParallelism(), 1);
      chunks = (int) (numTotalJobs * ((1-batchSize) / adjusted));
    }
    return JobBatch.getBatches(numTotalJobs, chunks);
  }

  @Override
  protected Collection<JobBatch> convertToWorkerJobs(
      Collection<JobBatch> jobs) throws EolRuntimeException {
    return jobs;
  }
```

Listing 5.4.3 Batch-based job splitter

5.4.3.1 Annotation-based

There is also a third way to split jobs, giving more control to the user in the process. Rather than randomly assigning a proportion of jobs to workers, we can rely on users to declare which rules (and even elements) should be distributed through declarative annotations. The idea is that the user annotates the rules to be distributed, where the annotation can also be parameterised and return a Boolean indicating whether the rule-element pair should be sent to workers. As an example, consider Listing 5.4.4. The "*publicFieldsOnlyInPOJO*" will be executed only on the master. The *"hashCodeAndEquals*" constraint will be executed on both the master and workers depending on the individual element – the condition states that the rule-element pair will be distributed to workers only if the *ClassDeclaration* has more than 12 "*bodyDeclaration*" elements, otherwise it will be evaluated on the master. The "*allImportsAreUsed*" constraint will only be evaluated by workers.

```
context ClassDeclaration {
  constraint publicFieldsOnlyInPOJO {
    guard: self.getPublicFields().notEmpty()
    check: self.bodyDeclarations.select(bd |
      bd.isKindOf(MethodDeclaration) and
      (bd.modifier.isUndefined() or bd.modifier.static)
      and not (
        bd.isEquals() or bd.isHashcode() or bd.isToString()
        or bd.isFinalize() or bd.isClone() or bd.isCompareTo()
      )
    ).isEmpty()
  }

  $distributed self.bodyDeclarations.size() > 12
  constraint hashCodeAndEquals {
    check {
      var hasEquals = self.getMethods().exists(md | md.isEquals());
      var hasHashcode = self.getMethods().exists(md | md.isHashcode());
      return
        (hasEquals implies hasHashcode) and
        (hasHashcode implies hasEquals);
    }
  }
}

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

Listing 5.4.4 Example of annotation-based distribution EVL program

Implementation-wise, we only need to execute the annotation for each rule-element pair during the split to determine which jobs are to be executed on workers. If the `@distributed` annotation is present with no expression, all elements applicable for the annotated rule will be evaluated workers. If no annotation is present, the rule will be evaluated on the master only for all elements. If the annotation is present and parameterised, it will be evaluated for each element to determine whether the given rule-element pair should be distributed to workers. The annotation-based job splitting strategy – which uses the atom-based data types for distribution, although it can also be adapted to the index-based approach – is shown in Listing 5.4.5.

```
public class AnnotationConstraintAtomJobSplitter extends
   JobSplitter<ConstraintAtom, SerializableEvlInputAtom> {


  @Override
  protected void split() throws EolRuntimeException {
    List<ConstraintAtom> allJobs = getAllJobs();
    if (shuffle) Collections.shuffle(allJobs);
    int numTotalJobs = allJobs.size();
    masterJobs = new ArrayList<>(numTotalJobs/2);
    ArrayList<ConstraintAtom> workersLocal = new ArrayList<>(numTotalJobs/2);

    for (ConstraintAtom ca : allJobs) {
      Variable self = Variable.createReadOnlyVariable("self", ca.element);
      if (ca.rule.getBooleanAnnotationValue("distributed", context, self)) {
        workersLocal.add(ca);
      }
      else {
        masterJobs.add(ca);
      }
    }
    workerJobs = convertToWorkerJobs(workersLocal);
  }

  @Override
  protected List<ConstraintAtom> getAllJobs() throws EolRuntimeException {
    return ConstraintAtom.getConstraintJobs(context.getModule());
  }

  @Override
  protected Collection<SerializableEvlInputAtom> convertToWorkerJobs(
      Collection<ConstraintAtom> jobs) throws EolRuntimeException {

    List<SerializableEvlInputAtom> workerJobs = new ArrayList<>(jobs.size());

    for (ConstraintAtom ca : jobs) {
      EolModelElementType type = ca.rule.getConstraintContext().getType(context);
      SerializableEvlInputAtom seia = new SerializableEvlInputAtom();
      seia.modelElementID = type.getModel().getElementId(ca.element);
      seia.contextName = ca.rule.getConstraintContext().getTypeName();
      seia.modelName = type.getModelName();
      seia.constraintName = ca.rule.getName();
      workerJobs.add(seia);
    }

    return workerJobs;
  }
```

Listing 5.4.5 Annotation-based job splitter

5.5 Results processing

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. Furthermore, if the ConstraintContextAtom is used for specifying jobs, then the result of executing a ConstraintContext will be a Collection of UnsatisfiedConstraint instances, since a ConstraintContext may have many Constraints. In the absence of any UnsatisfiedConstraints, an empty collection is used. As for the Collection itself, we opt for an ArrayList, which has a small footprint and is serializable.

Although each job is unique in that it is only sent and processed once, there may still be duplicate results due to dependencies. In EVL, constraints may invoke one or more constraints through the *satisfiesOne* and *satisfiesAll* operations. These operations can be invoked on a given model element type, taking as input the name of the constraint(s) for which the given element should be validated against. The former checks that at least one of the constraints returns true, whilst the latter requires all of the constraints to be satisfied. 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. 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, no communication happens between workers: the dependencies are executed separately on each worker if and when required.

When the master receives a result (or results), it may deserialize the UnsatisfiedConstraint(s) by looking up the local Constraint instance based on the name, and the model element based on the ID. However when there are many results, this process (specifically finding an element by its ID in a large model) can be very expensive, at least with in-memory XMI-backed EMF models. Instead of eagerly performing this de-serialization, we create a *LazyUnsatisfiedConstraint* which contains the serialized result. That is, the components of an UnsatisfiedConstraint (e.g. the model element, message, Constraint etc.) are only resolved on demand individually when queried. This helps to minimise the workload on the master during execution. We also override *equals* and *hashCode* to ensure comparison of lazily resolved UnsatisfiedConstraints is based on the proxies to avoid unnecessary resolution when performing insertions into the results set on the master.

Depending on the implementation environment, it may also make sense to write the results directly to a file or database rather than sending them back to the master and incurring the serialization overhead. This would be a sensible decision if the results would only be processed at a later stage, or if the model validation exercise is intended for diagnostic purposes rather than to be used directly within a single monolithic process. However for simplicity and compatibility with all use cases, we collect the results 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.

5.6 *MapReduce* decomposition

The semantics of our distributed approach as described so far can also be expressed using the *MapReduce* model of computation [264], which is one of the most well-known approaches to distributed data processing. The approach operates on records of data organised into key-value pairs. As its name suggests, MapReduce has two (modern realisations such as Hadoop have more) main stages, with each stage performed by a different worker (there can be multiple workers for each stage). The data is then distributed by the master node and sent to *Map* workers, which apply a function to each record and produce an intermediate record (key-value pair). Before being passed on to the reduce function, records with the same key are grouped together. The *Reduce* function then takes these intermediate records and produces a (possibly filtered) collection of their values. These functions can be summarised in terms of the following Java data types:

```
List<Entry<K2,V2>> map (Entry<K1,V1> input);
```

```
List<V2> reduce (K2 key, List<V2> values);
```

In this regard, we can model our inputs and results to fit this domain by substituting the following values for each type (assuming the atom-based, rather than index-based distribution strategy):

```
K1 → ConstraintContext
```

```
V1 → ModelElementID
```

```
K2 → ConstraintContext
```

```
V2 → Collection<SerializableEvlResultAtom>
```

The inputs to the *map* function would be ConstraintContext – element pairs; with the ConstraintContext's name as the key and the ID of the model element as value. The function then evaluates all of the constraints for the given element, and produces intermediate values where the keys are the ConstraintContext names and the values are the unsatisfied instances (the constraint, message and model element ID). The *reduce* function then simply discards the key and returns the values. In other words, we are only interested in using the *map* function to transform each serializable ConstraintContext into a Collection of serializable UnsatisfiedConstraint instances (which may be empty).

Other strategies are also possible to accommodate the user's requirements. For example, we could formulate the solution so that each instance of *V2* corresponds to the model element ID of an UnsatisfiedConstraint, or the number of elements for which the n^{th} Constraint in the program were unsatisfied, in order of their appearance in the script.

5.7 Flink implementation

Perhaps the most straightforward and obvious way to realise our distributed implementation is to use an off-the-shelf distributed processing framework. As mentioned above, one potential choice is Hadoop, however there are many alternatives within the Hadoop ecosystem which offer better performance, are more modern, have richer functionality and/or a more user-friendly API. Most of the distributed processing frameworks are built on top of the Akka [124] actor system, and are primarily targeted towards distributed applications where low-latency rather than high throughput is a priority. Thus they adopt a stream-based approach, where each processing stage is asynchronous; that is, each data element goes through the entire pipeline one at a time rather than processing each function for all inputs before moving on to the next. Nevertheless, such frameworks are usually well-optimised and becoming increasingly mature such that they are also suitable for batch processing. In our case, we only need a single function which transforms ConstraintContext-element pairs into a collection of (serializable) UnsatisfiedConstraint instances, so there is practically no difference between a stream and batch-oriented approach.

One of the most well-known and widely used distributed processing frameworks is Apache Spark [277], which is suitable for both low-latency and high-throughput applications. Spark revolves around *Resilient Distributed Dataset*s (RDDs), which can conceptually be thought of as a kind of distributed Java Streams. They operate on a data source (which may be lazily produced and thus infinite), consist of a pipeline of operations and a final "sink" where the data stream ultimately ends – usually written to a file or database. In the Hadoop ecosystem, we could take advantage of HDFS – the distributed filesystem – to write the results from each node to a file, rather than sending them back to the master. However the functionality to collect all results on the master is also present.

Although Spark appears to be a good candidate for our requirements, it is missing one crucial capability: to run a one-time setup code on worker nodes. Like most distributed processing frameworks in the Hadoop ecosystem, Spark is written in and primarily geared towards the Scala programming language. The implicit assumption of Spark is that all processing functions are stateless: that is, when a process function is invoked on a worker, all of the data required to produce the output is contained in the parameters to the function. This is a logical and reasonable assumption – after all, distributed processing functions should be pure. Indeed in our case, this is somewhat true: for a given ConstraintContext-element pair, the same results would be sent every time. However the function itself is not stateless and requires setup. This setup would be extremely inefficient to perform upon each invocation – loading the model every time and parsing the script only to evaluate a single ConstraintContext-element pair would eliminate any performance benefits from distribution. Instead we need to be able to set up each worker so that it has the script parsed and model(s) loaded before it begins processing jobs. Unfortunately without abusing Classloaders this is not possible at the time of writing, although it is a long-requested feature [278].

Fortunately, a very similar and more modern distributed processing framework exists which does provide a way to run initialisation and teardown code before and after each process function. Apache Flink [279] is another distributed processing framework offering a rich API and processing capabilities similar to Spark. Flink treats all data as streams internally, however it offers a more optimised *DataSet* API for bounded data – ideal for our case. As previously mentioned, we only need a single function implementation. We use the *flatMap* function (one to many *map* function), where conveniently we output only the UnsatisfiedConstraints resulting from execution of a ConstraintContext-element pair. This avoids the need for sending an empty collection when no UnsatisfiedConstraints were encountered. Our implementation of this function is very concise and requires very little adaptation from our generic framework, as shown in Listing 5.7.1. The *getParameters* static method retrieves the configuration parameters (i.e. the HashMap containing paths to the script, models, parameters, output file etc.) from the Configuration object passed to the *open* function; which is invoked once per worker before jobs are sent to the *flatMap* method. We parse these key-value pairs into a container which contains all of the necessary state required for executing any ConstraintContext-element pairs.

```
public class EvlFlinkFlatMapFunction<IN extends Serializable> extends
RichFlatMapFunction<IN, SerializableEvlResult> {

  protected transient EvlModuleDistributedSlave localModule;
  protected transient DistributedEvlRunConfigurationSlave configContainer;

  @Override
  public void flatMap(IN value, Collector<SerializableEvlResult> out)
     throws Exception {

    localModule.executeJob(value).forEach(out::collect);
  }

  @Override
  public void open(Configuration additionalParameters) throws Exception {
    configContainer = EvlContextDistributedSlave.parseJobParameters(
      getParameters(
        getRuntimeContext(), additionalParameters
      ).toMap(), null
    );

    localModule = configContainer.getModule();

    configContainer.preExecute();
    localModule.prepareExecution();
  }
}
```

Listing 5.7.1 Distributed EVL Flink *flatMap* function implementation

In Listing 5.7.2 we show how the *flatMap* function is used as part of the execution. To complete the distributed processing pipeline, we only need a source of data – which may be the list of batches or individual serializable ConstraintContext-element pairs – and the sink – which is either a text file or a collection. The master only needs to provide Flink with the configuration parameters (i.e. the path to the models, script, parameters etc.) and the actual input data to the flatMap function itself. Notice that our abstract EVL module allows for either the index-based or atom-based approaches to be used by treating the data (*D*) as a Serializable. In the index-based implementation, *D* becomes a *JobBatch* (i.e. the object which contains the *from* and *to* indices of the job list) whilst in the atom-based implementation, *D* becomes *SerializableEvlInputAtom*. Likewise, the result will be an implementation of *SerializableEvlResult* (i.e. either *SerializableEvlResultAtom* for the atom-based or *SerializableEvlResultPointer* for the index-based implementation).

```java
public abstract class EvlModuleFlinkMaster<D extends Serializable> extends
EvlModuleDistributedMaster {

   private ExecutionEnvironment executionEnv;

   @Override
   protected final void prepareExecution() throws EolRuntimeException {
      super.prepareExecution();
      EvlContextDistributedMaster context = getContext();
      executionEnv = ExecutionEnvironment.getExecutionEnvironment();
      executionEnv.setParallelism(context.getDistributedParallelism());
   }

   protected abstract DataSource<D> getProcessingPipeline(ExecutionEnvironment
execEnv) throws Exception;

   @Override
   protected final void checkConstraints() throws EolRuntimeException {
      try {
         Configuration config = getContext().getJobConfiguration();
         String outputPath = getContext().getOutputPath();
         executionEnv.getConfig().setGlobalJobParameters(config);
         DataSet<SerializableEvlResult> pipeline =
            getProcessingPipeline(executionEnv)
            .flatMap(new EvlFlinkFlatMapFunction<D>());

         if (outputPath != null && !outputPath.isEmpty()) {
            pipeline.writeAsText(outputPath, WriteMode.OVERWRITE);
            executionEnv.execute();
         }
         else {
            deserializeResults(pipeline.collect());
         }
      }
      catch (Exception ex) {
         EolRuntimeException.propagate(ex);
      }
   }
```

Listing 5.7.2 Distributed EVL Flink master module implementation

5.8 JMS implementation

To further demonstrate the generalisability of our approach and dive deeper into the technicalities of the communication involved, we implement our distributed solution using a generic messaging API. This enables us to highlight and better understand the execution stages of our distributed approach. A custom lower-level implementation based on messaging allows us to optimise further, since we are not targeting low latency with infinite streams (the scenario which most distributed processing frameworks are geared towards) but high throughput with finite data. We can also control the exception handling semantics and have full control over message sequencing. We implement a custom distribution solution as our main implementation which we evaluate. This message-based implementation shows that even in the absence of a fully-fledged distributed processing framework, the communication between the master and workers is simple enough that it can be hand-coded using virtually any technology which supports communication over networks using standard protocols such as TCP.

5.8.1 **Java Message Service**

The Java Message Service (JMS) [125] is an API specification for enterprise Java applications designed to facilitate communication between processes (which may be on different computers or networks) through messaging. 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 [280]; a modern JMS 2.0 compliant broker.

5.8.2 **Architecture**

Our implementation follows a master-slave architecture. 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 creates a Message containing the configuration parameters, attaches the master's 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.

If the number of workers is known in advance (or the master proportion parameter is specified), the workload is divided between the master and workers as discussed in section 5.4.1. 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, 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, the 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 all jobs have been processed by a worker, it 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.

5.8.3 **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 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.

5.8.4 Termination Criteria

Another challenge 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 (i.e. per-message basis). Our solution is arguably simpler and only requires a one-time response from each worker.

5.8.5 Summary

To summarise this implementation and the sequence of events in a more understandable way, we present Figure 5.8.1, which shows the relevant communication channels and ordering of events. Note that the communication channels are asynchronous and reactive, with the direction of arrows signifying data flows. Figure 5.8.2 shows the order of events, although it should be noted that the timeline of individual workers and the master are separate due to the asynchronous design.

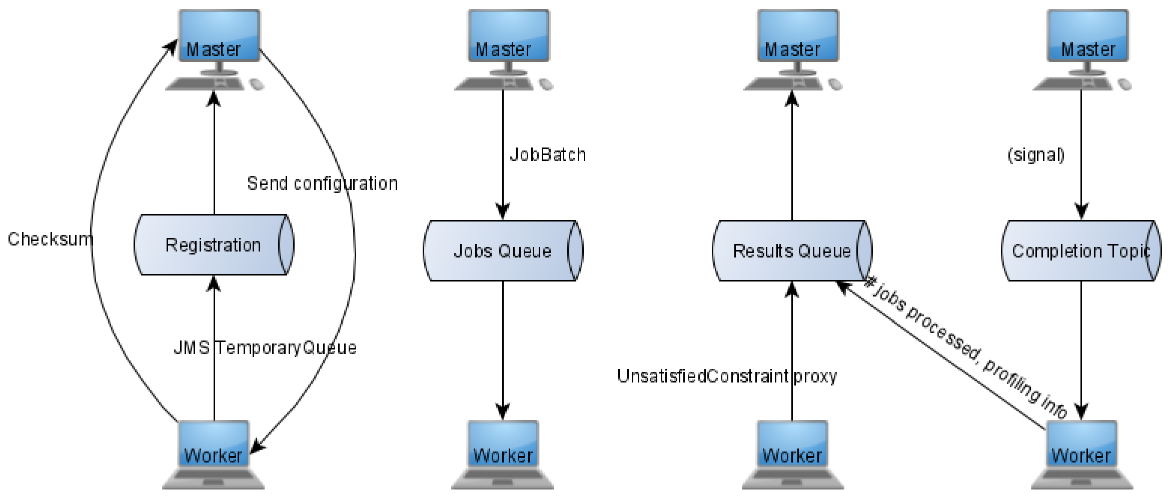


Figure 5.8.1 Overview of communication between master and workers

In conclusion, the main challenges with a manual message-based implementation stem from the need for correct sequencing of events during the setup phase, and responding to exceptions. We argue that the number of communication channels needed is small enough to be feasible to implement in the absence of more sophisticated distribution mechanisms such as Apache Flink. We hope that this demonstrates the generalisability and efficiency of our solution in terms of complexity and developer effort required to realise such an implementation.

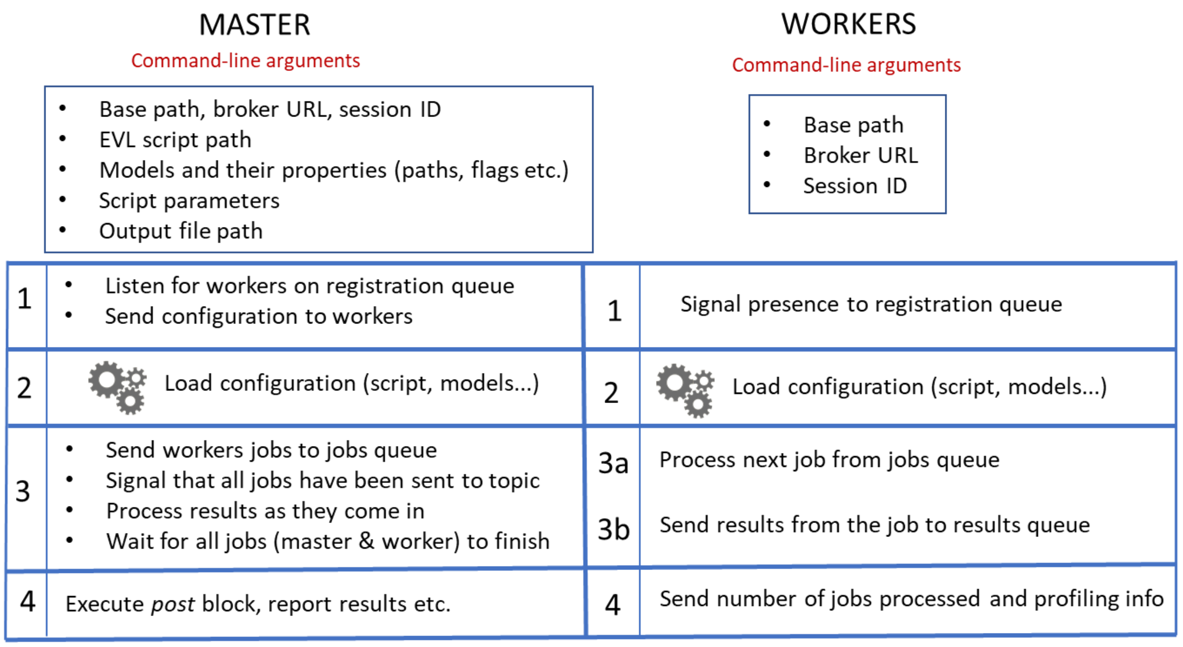

MASTER
Command-line arguments
Base path, broker URL, session ID
EVL script path
Models and their properties (paths, flags etc.)
Script parameters
Output file path
WORKERS
Command-line arguments
Base path
Broker URL
Session ID
1
Listen for workers on registration queue
Send configuration to workers
1
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

Figure 5.8.2 Summary of asynchronous execution steps in JMS implementation

5.9 Performance

We re-use the same models and programs from the parallel EVL benchmarks in evaluating distributed EVL, to allow for a comparison in performance. We focus on the *findbugs* script (and its subset derivatives) in these experiments. Before timing each experiment in distributed mode, we ensured all workers were started first. We used our base Threadripper system as the master node in all cases. We used Apache ActiveMQ Artemis 2.10 [280] as our broker with message persistence disabled. For reference, where applicable we used EMF 2.15 and Eclipse OCL 6.7.0. When recording execution time, we exclude the initial model loading time for the master. However since all workers must load the model before processing can begin, we do not exclude the loading time for workers. We repeated each experiment several times and took the mean average time in milliseconds. We only show results for the batch-based approach, since it is much more performant and makes more sense if the full model and program is available on all nodes. We use one system as a baseline for all benchmarks. The workers are running the i5-8500 system, and are all identical. The master will be the Threadripper 1950X system unless otherwise stated.

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. 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.

5.9.1 **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.

5.9.1.1 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.

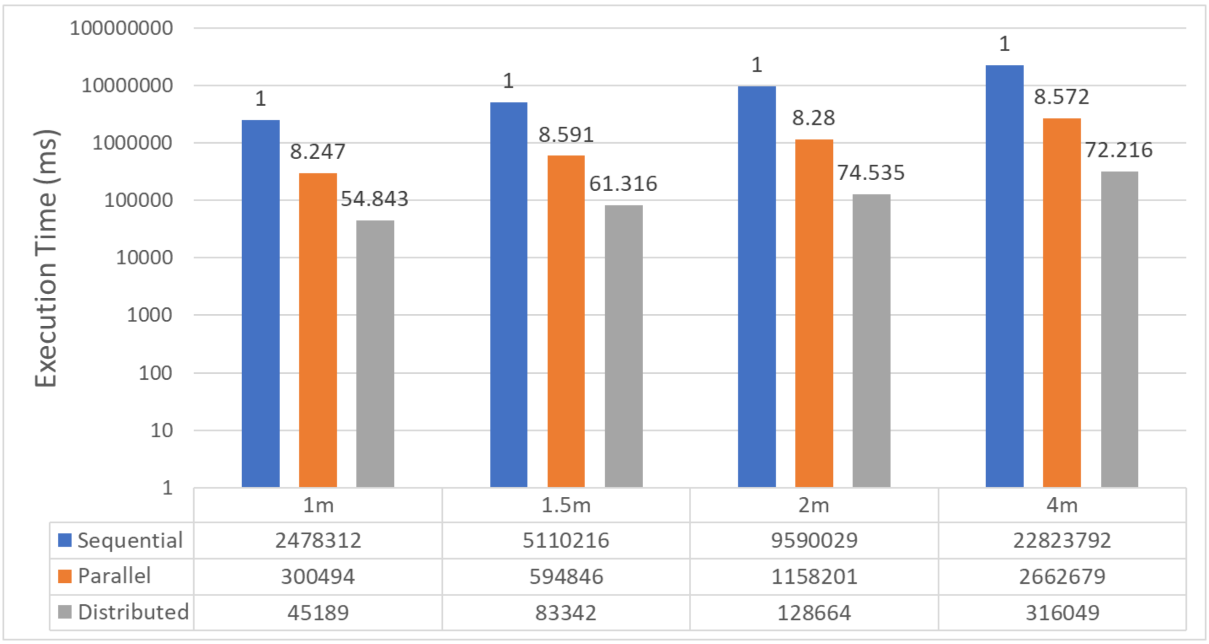
5.9.1.2 Batch Size

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 1, especially for unevenly distributed workloads like the *findbugs* script. 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 two minutes. A smaller factor 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.

5.9.2 Results for *findbugs*

Table 5.9.1 Results for simplified *findbugs* script with 16 workers (TR-1950X master)

# Elements	Implementation	Model Load (ms)	Execute (ms)	Speedup
1m	Interpreted OCL	7686	42803	--
1m	Compiled OCL	8312	8871	4.825
1m	Sequential EVL	9354	20308	2.108
1m	Parallel EVL	9317	4966	8.619
1m	Distributed EVL	9058	6777	6.316
2m	Interpreted OCL	15769	86659	--
2m	Compiled OCL	16774	16993	5.1
2m	Sequential EVL	18132	38741	2.237
2m	Parallel EVL	18179	8637	10.033
2m	Distributed EVL	18371	8299	10.442
4m	Interpreted OCL	31856	145540	--
4m	Compiled OCL	32337	29928	4.863
4m	Sequential EVL	35294	61482	2.367
4m	Parallel EVL	34623	14541	10.001
4m	Distributed EVL	34751	13736	10.596



	1m	1.5m	2m	4m
🟦 Sequential	2478312	5110216	9590029	22823792
🟧 Parallel	300494	594846	1158201	2662679
⬜ Distributed	45189	83342	128664	316049

Figure 5.9.1 *findbugs* 16 workers and 1-4 million elements (TR-1950X master)

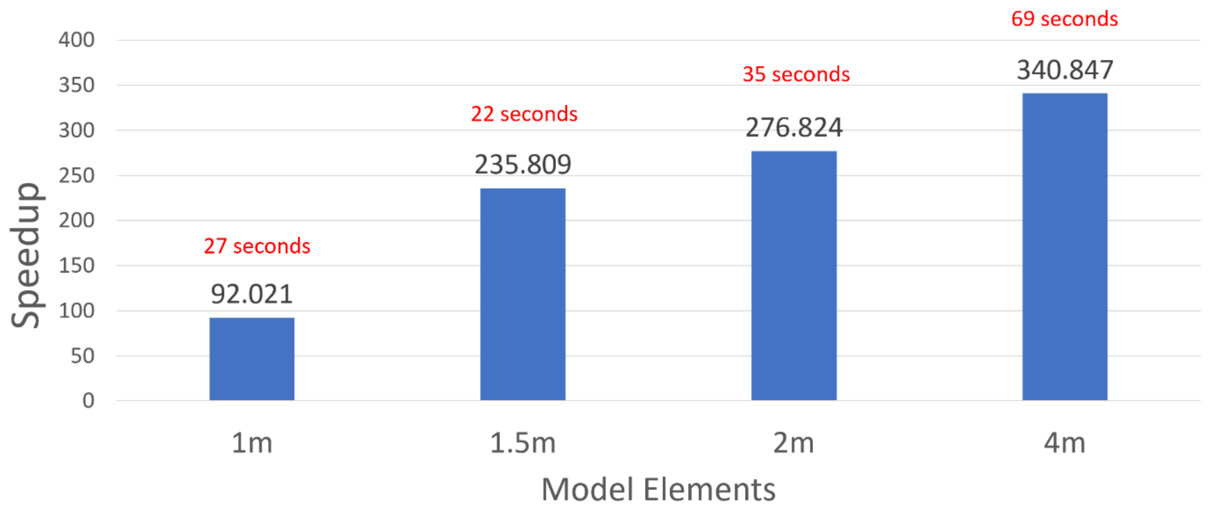


Figure 5.9.2 *findbugs* with 87 workers, 0.015 master proportion, 32 batch size (TR-1950X master)

Figure 5.9.1 shows the execution times (on a logarithmic scale) for the *findbugs* script with varying model sizes and sixteen workers. Although parallel EVL provides a speedup between 8.2 and 8.6x on our 16-core system, distributed EVL varies considerably more, scaling better with model size. At its peak, we see an improvement of almost 75x compared to the single-threaded engine with 2 million model elements. In absolute terms, this means a configuration which took approximately 2 hours and 40 minutes is reduced to just 2 minutes and 8 seconds. 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.

Although the potential speedup for distributed EVL compared to sequential is $(6 * 15) + 4 + 16 = 110$ based purely on number of cores, the overhead of distribution, the 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.

Table 5.9.1 shows the results for *findbugs* with the most demanding constraint (as shown in Listing 4.5.1) removed – readers of section 4.5.2.1 will remember this as the *java_simple* script. Note that the execution time is much lower, and in some cases 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 OCL.

In Figure 5.9.2 we attempt to assess the potential scalability of our solution with specific tuning parameters. We use 87 workers with a relatively small batch size of 32, although significantly larger than the number of logical cores on the workers. Nevertheless, we found this configuration to be a good 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 seconds, compared to the sequential case of over 6 hours and 20 minutes. However as with the results in Table 5.9.1, with smaller models the execution time becomes comparable to the model loading time. We can clearly see an upwards trend: the larger the model, the greater the speedup. Amdahl's Law likely plays a role here, along with the fact that a longer runtime and larger model results in more memory allocations and garbage collection overhead with a single machine. With more workers, this is spread out so that each computer does less work, with less overhead incurred by memory allocation and other JVM internals.

An interesting observation from Figure 5.9.2 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 seconds or 12 seconds (±1 second). 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.

5.9.3 Single constraint, 2 million elements

Now we take a deep dive, focusing on one specific configuration to further assess the scalability and efficiency of our solution. For this we chose the single most demanding constraint and ran it against the 2 million elements model. 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). The results are shown in Figure 5.9.3. 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 parentheses indicate the number of threads in the case of parallel implementation, and the number of workers in the case of distributed.

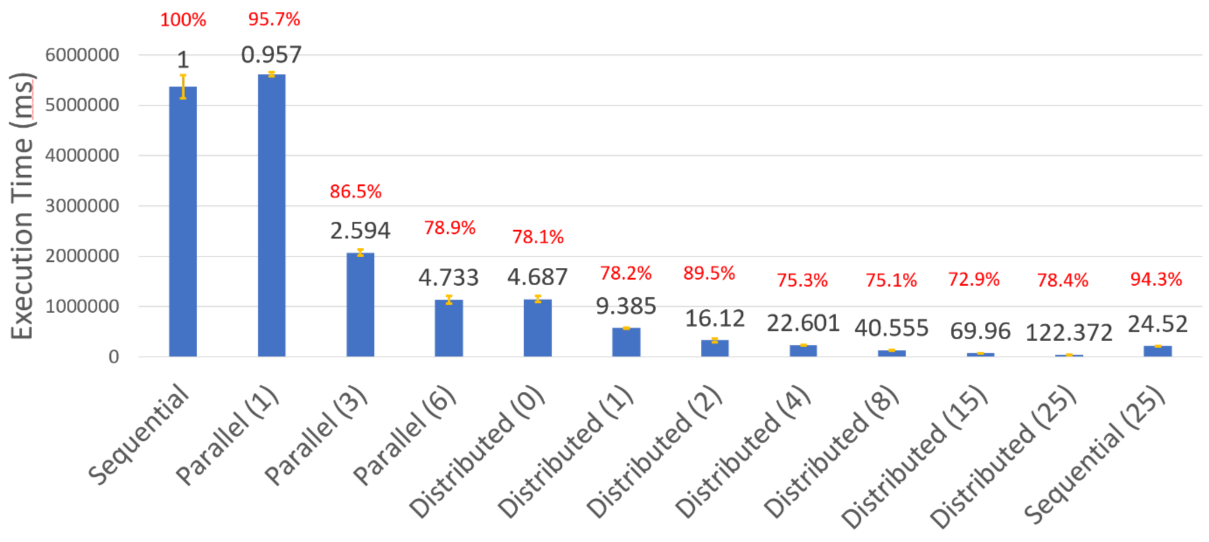


Figure 5.9.3 *1Constraint* with 2 million elements, 6 batch size (i5-8500 system)

The results here are in line with our expectations. The parallel and distributed implementations perform almost identically and are around 4.7x 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 96x, and our implementation achieves 70x. 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” 4.7x faster than the sequential implementation and we are using the same parallel execution infrastructure in the distributed case, then clearly 96x 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 6 percentage point drop in efficiency with a 16x 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 the jobs assigned to the master are done so beforehand and not dynamically load-balanced (unlike 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 minutes, 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.

5.9.4 Single-threaded parallelisation

In Figure 5.9.3 we also ran the benchmark with local (shared memory) parallelism set to 1 to be able to compare the efficiency of the distributed approach without any parallelisation overhead. We tried this with 25 workers – the result for single-threaded distributed EVL is labelled denoted on the chart as "Sequential (25)" – and observed a speedup of 24.5x over sequential EVL: very close to the theoretical limit of 26x. We also ran the benchmark with local parallelism set to the number of cores as usual for comparison. With over 122x speedup out of a theoretical 156x, 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 hour 30 minutes, with 25 workers this is reduced to just 44 seconds. 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 5x speedup compared to the sequential case with the same number of workers. This is slightly higher than the 4.7x 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. One of these is that with distributed EVL, we effectively eliminate the thread pool scheduling by setting the batch factor equal to the number of threads: there are only 6 jobs being processed at any given point by any given computer, since the jobs are pulled from the queue as workers are able to process them. As 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.

5.9.4.1 Simulink

We have established that our index-based batch distribution strategy delivers strong performance and scalability – far beyond what shared-memory parallelism can offer even with 16 cores. This is aided further by the fact that our distributed approach builds on the shared-memory parallel architecture, ensuring maximum CPU utilisation across all machines involved. Moreover, our distribution strategy can provide benefits in cases where shared-memory parallelism is not feasible due to technical limitations of the modelling technology. This is exemplified by the Simulink EMC driver [add citation]. Simulink [281] is a proprietary model-based simulation framework and a component of MATLAB. MATLAB is a very rich, long-standing and complex software with many components, and is mostly written in C++. The framework offers APIs for external tools from different ecosystems to leverage some of its functionality. Java is one of the supported platforms, so the Simulink driver in Epsilon uses the Java API of the MATLAB engine to interact with Simulink models. However despite the Simulink driver extending CachedModel, for reasons still unknown at the time of writing, this EMC driver – or at least the MATLAB engine – is not thread-safe. Regardless of the reason, it is inevitable that there will be modelling technologies which severely limit the potential benefits of shared-memory or even shared-system parallelisation. In these cases, rather than giving up on parallelisation or committing to spend a very long time on debugging and resolving thread safety issues – which may not be in the hands of EMC driver developers as they rely on external proletary tooling – our approach offers a plausible fool-proof solution which requires no additional code or understanding of how the EMC driver is implemented. The only requirement is that *getAllOfKind* returns model elements in a deterministic order every time.

The idea is simple: we invoke the master and workers with a parallelism of 1. This still uses the parallel EVL architecture, but with only a single worker thread executing the script or invoking the EMC driver at a time for any given machine. However, we can still have multiple workers, since each worker runs in its own process – infact, on its own hardware which is not shared with other workers. Therefore any issues with, for example, multiple processes interacting with the same MATLAB instance or reliance on intermediate files persisted in non-volatile memory – are impossible. Each worker executes a subset of the program with its own resources on its own machine. In these cases, we are relying on our job splitting strategy and distribution mechanism to co-ordinate independent processes executing a subset of the EVL program with full access to all resources. This is possible because model validation is essentially a specialised form of model querying, where the output is a Set of unsatisfied constraint-element tuples. Our approach decomposes this query into a fine level of granularity so that workers can execute only the relevant subset.

To demonstrate this, we used the EVL program and models from [282] (see Sanchez et al. (2019) [283]) , which was independently developed and not modified whatsoever for our experiments: we did not even look at the script or model or even attempt to understand the context to prevent any subconscious optimisation attempts within the script itself, the execution engine and/or distribution strategy. We treat the program and model and the EMC driver as a black box, which helps to validate our claim and avoid any undue biases.

Table 5.9.2 Single-threaded Simulink experiment, 6 batch size (i5-8500 master)

# Workers	Model load (ms)	Execute (ms)	Execute STDEV	Speedup
0	33165	484535	31546	--
15	14399	205766	17581	2.355

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 seconds) with only a single outlier at 21 seconds. Execution time was also more volatile than we expected, with sequential EVL showing a standard deviation of over 31 seconds. 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 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 seconds. 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 ConstraintContexts 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, so our theory still holds.

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.

5.9.5 **Single machine, multiple processes**

We have seen that, remarkably, our distributed approach is more *efficient* than our parallel one, even when combined with local parallelisation. That is, the speedup achieved for any given level of parallelism (number of computers multiplied by number of logical cores, assuming all participating nodes are homogenous) is greater under the distributed setting than under the purely single-machine case, as shown in Figure 5.9.3. To investigate this further, we can try to assess the extent to which our parallelisation infrastructure is a bottleneck, as opposed to hardware limitations (e.g. memory bandwidth) by running our distributed program on a single machine. That is, one computer hosts the broker, master and workers. Of course, there are limited resources and we do not want the total level of parallelism to exceed the number of logical cores. We balance this by adjusting the local parallelism parameter (i.e. the number of threads used by each process) and the number of workers. For example, if we have 16 logical cores, then we can experiment with any combination of workers and threads per worker such that the product of the two is equal to 16. For instance, we can have 16 workers each using one thread, or 2 workers using 8, or 4 using 4 and so on. The main limiting factor is memory: since each worker runs in its own process, it also hosts its own data structures and loads the model into memory. We therefore can't test with very large models and a large number of workers. Still, the Ryzen 7 3700X system has 32 GB of memory, which is enough for experimenting on models with up to one million elements. By experimenting with multi-process parallelism, we can assess the extent to which the data structures in our shared-memory approach are a bottleneck, since hardware is effectively taken out of the equation. By this, we mean to say that with a single computer, the number of memory channels is limited, whereas in our previous experiments, the distributed approach can be expected to be faster because the workload was spread across more computers, meaning more memory channels (since we had one worker process per computer). Now we can assess whether multi-process is truly more efficient (in terms of execution time) than shared-memory once factoring in the hardware limitation.

We tested with the *findbugs* script, since this was the most demanding and we were limited on model size due to memory capacity. We should also note that 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 and 0.25 with 3 workers and 0.125 with 7 workers. The results are shown in Table 5.9.3.

Considering the largest model (1 million elements) with four processes, the program was 7x faster than the sequential implementation, whilst with a single process speedup was just over 4.8x. This is well beyond the margin of error: over 76 seconds' difference whilst retaining the same total level of parallelism, where the standard deviation was between 4 and 7 seconds on average. However we can see that with 8 total processes, performance suffers and speedup drops to 6x. At this point, all processes combined were consuming around 20 GB memory. This trend continues with 500k elements, where we can see that with two processes, the program was over 5.2x faster than the single-threaded implementation, whereas the purely shared-memory parallel implementation was only around 4.6x faster. Further increasing the number of workers improved efficiency somewhat: we observed a speedup of over 5.4x. Doubling this again reduced performance, though with 8 processes the execution time was still superior to the single-process parallelism case. Curiously, we also saw the standard deviation fall drastically with multiple processes: perhaps due to the fact that the batch factor was a smaller number, leading to more predictable performance. However, whilst the numbers may appear damning for shared-memory parallelism, in absolute terms the best-case difference in execution time was only 10 seconds: a mere 3% of the total execution time of the sequential implementation. This becomes even less relevant when considering the standard deviation. With 200k elements, the results contradict the hypothesis that shared-memory parallelism is less efficient. Here it is clear that a single process is more efficient, with speedup decreasing as we add more workers. Although the difference in absolute terms is only a few seconds, the variance is also comparatively low (below 1 second), meaning the results obtained are beyond margin of error.

Table 5.9.3 *java_findbugs* with multiple processes (R7-3700X system)

Implementation	Model	Execute (ms)	Execute STDEV	Speedup
Sequential	200k	43302	1105	
ContextAtoms (16)	200k	10723	384	4.038
Distributed Local (1 worker)	200k	12548	339	3.451
Distributed Local (3 workers)	200k	12760	143	3.394
Distributed Local (7 workers)	200k	17084	414	2.535
Sequential	500k	319923	13484	
ContextAtoms (16)	500k	68676	5287	4.658
Distributed Local (1 worker)	500k	60922	2691	5.251
Distributed Local (3 workers)	500k	58609	847	5.459
Distributed Local (7 workers)	500k	65550	791	4.881
Sequential	1m	1192448	48753	
ContextAtoms (16)	1m	246599	7179	4.836
Distributed Local (1 worker)	1m	233434	3689	5.108
Distributed Local (3 workers)	1m	170410	5299	6.998
Distributed Local (7 workers)	1m	196487	2233	6.069

Based on these results, our conclusion is that the distributed approach is more efficient for larger models, which are more memory-intensive. With smaller models, the overhead of distribution becomes more significant, and so a single process is more efficient. The efficiency of multi-process is not always guaranteed: there is a balancing act between number of processes and number of threads per process. Based on the results in Table 5.9.3, it appears the optimal is 4 processes with 4 threads each. The optimum will differ depending on the particular hardware platform and program configuration (model and script). It should also be noted that this experiment is purely of "academic interest": the cost of loading the model multiple times into memory on the same computer is an unrealistic and extremely wasteful proposition, since the memory requirement is multiplied by the number of processes (with 4 processes, the memory required is at least 4x compared to the shared-memory parallel case) and only improves performance by a few percent compared to the standard parallel implementation. The takeaway from this is that shared-memory parallelism does indeed have limitations at the software level, not just in terms of hardware. Therefore, future work should attempt to identify the main source of these bottlenecks and propose solutions to close the gap in efficiency between shared-memory and mutli-process parallelism.

5.10 **Summary**

Having established a thread-safe and optimised infrastructure as well as a data-parallel execution algorithm for the Epsilon Validation Language, in this chapter we went further by showing how we can combine shared-memory parallelism with the scalability provided by multi-process distributed computing. 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. By incorporating lazy and asynchronous computing techniques into our distributed parallel architecture, we have been able to achieve performance improvements of a magnitude which is unheard of in the literature.

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 340x 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 24.5x speedup. Repeating this experiment where each worker leverages all of its cores, we saw a further 5x speedup, making it over 122x faster than the traditional single-machine, single-threaded execution engine. Overall, we found that our distributed execution approach is actually more efficient than shared-memory single-process parallelism, due to alleviation of the von Neumann bottleneck (memory access).

6 Parallel Iteration Operations

In Chapter 4, we introduced parallelism and highlighted the associated challenges and solutions with introducing concurrency to the Epsilon Model Validation Language (EVL). However model validation is just one task in a typical model management workflow. In order to bring the benefits of parallelisation to other model management tasks and more general workflow operations, in this chapter we attempt to generalise our data-parallel solution by factoring out a reusable parallelisation framework which can easily be applied to other model management tasks. Primarily, we achieve this by introducing parallelism to the declarative aspects of model management programs. As each model management language has a different set of associated complexities and challenges, we focus on model querying operations, which can be used by all model management tasks. Thus where the last chapter focused on EVL as the implementation target, in this chapter we use the Epsilon Object Language (EOL) to demonstrate our approach.

The remainder of this chapter is organised as follows. Section 6.1 further motivates the use of EOL rather than the Object Constraint Language (OCL), which is both the Object Management Group and de-facto language used for model querying. Section 6.2 provides details on the parallelised first-order collection operations. Section 6.3 introduces a novel parallel operation which can be used as a more efficient replacement for the common task of comparing a subset of a collection’s size to an integer constant. Section 6.4 introduces lazy and parallel operation chaining semantics to EOL by leveraging Java Streams. Sections 6.5 and 6.6 discuss the nuances with parallelisation from an engineering and user-centric perspective: namely, how to optimally apply parallelism automatically and the interplay between parallel operations and task-specific parallelisation. Section 6.7 evaluates the performance and correctness of the parallel operations. Section 6.8 consolidates the parallelisation approach from a language engineering perspective and summarises the chapter.

6.1 Motivation for Parallel EOL

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

As noted by Willink (2017) [284], who is the lead developer of Eclipse OCL, the specification unfortunately precludes some optimisations which would enhance the efficiency of collection-based operations. Most notably, this includes the inability to perform short-circuiting because all errors in OCL must be caught and propagated as *Invalid*. This means that if for example the last element in a collection is null and the predicate of a first-order operation dereferences the element, even though the last element may never be reached under short-circuiting evaluation the specification requires that all elements be evaluated / checked for validity. Furthermore, the specification forbids lazy evaluations of chained operations, and mutability. However as Willink claims, the underlying implementation need not be purposely wasteful so long as the observable end result (from the user's perspective) is consistent with the specification.

To demonstrate these shortcomings, let us consider a simple query. Suppose we have a transportation model of a city and we want to know whether there are any cars of a given brand registered after a particular year. One way to write this in OCL is shown in Listing 6.1.1. In a typical OCL execution engine, the following sequence of evaluations would occur:

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

Listing 6.1.1 Example of an inefficiently expressed OCL query

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

This is a very inefficient algorithm for expressing this query, even though from the user's perspective it is intuitive to write. A much more optimal representation is shown in Listing 6.1.2, which not only requires a single iteration but also avoids creating intermediate collections as well as partial short-circuiting of the expression, since OCL permits short-circuiting of Boolean values and expressions.

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

Listing 6.1.2 More optimal expression of the query in Listing 6.1.1

Even though Listing 6.1.2 is significantly more efficient, it can be further optimised. Firstly, it is not necessary to retrieve all instances of the *Car* type into memory before evaluating the *exists*, but rather iterating through them as and when required. Secondly, if the OCL specification was less strict about propagation of *Invalid* then *exists* could stop executing once a match (i.e. an element which satisfies the predicate criteria) has been found. Finally, the execution of *exists* need not happen sequentially, as there are no dependencies between iterations and elements used in the operation.

In order to develop efficient parallel model querying operations, we free ourselves from the restrictive OCL specification and semantics, opting instead to focus on the Epsilon Object Language. The same arguments can be made for using EOL as with EVL; namely the experimental nature of Epsilon and the lack of a formal specification along with its closer semantics to Java. Since EOL supports both imperative and declarative programming, as well as the ability to work with native code, a broader range of issues can be considered when incorporating parallelism into the language.

Moreover, since EOL is the base language for all other Epsilon model management languages, adding parallelism and improving performance of the base expression language will benefit all other languages which are able to take advantage of the developed parallel constructs. Finally, adding parallelism to EOL enables us to explore integration issues with parallelism of queries and task-specific parallelisation (such as parallel model validation as described in the previous chapter). This issue of where and how to apply parallelism under such circumstances is explored in Section 6.6.

6.2 Parallel first-order operations

The declarative collection operations in EOL provide an obvious and relatively "safe" point of entry for introducing parallelism. Unlike imperative looping constructs (which are also available in EOL), the functional style iteration operations are not only more concise but also easier to reason about due to their specialised purpose. As discussed in the literature review, functional programming is inherently suitable for parallelism due to its more constrained semantics. To illustrate, consider the example in Listing 6.2.1, which attempts to filter a collection of model elements according to some arbitrary criteria.

```
Collection<Car> selectCarsByYear(Collection<Car> cars, int year) {
   var results = new ArrayList<Car>(cars.size());
   for (Iterator<Car> cIter = cars.iterator(); cIter.hasNext();) {
      var car = cIter.next();
      if (car.year == year) {
         results.add(car);
      }
   }
   return results;
}
```

Listing 6.2.1 Imperative *select* operation example

In contrast, consider Listing 6.2.2, which serves the same purpose as the code in Listing 6.1.1 but is much more concise, readable and crucially for our purposes, optimisable.

```
Collection<Car> selectCarsByYear(Collection<Car> cars, int year) {
   return cars.select(car -> car.year == year);
}
```

Listing 6.2.2 Declarative *select* operation example

The problem with Listing 6.2.1 from an optimisation perspective is that without static analysis, we cannot reason about the thread safety of the operation. A manual looping construct allows users to potentially perform different operations in each iteration (for instance, by examining the loop count variable). The body of a loop can also contain arbitrarily complex code which modifies variables declared outside of the loop, which in most loops is the intended behaviour. In Listing 6.2.1, modifications to the result collection happen in every iteration, so such a collection needs to be thread-safe. By contrast, in Listing 6.2.2 the semantics of the operation are clear and the user has no control over the looping mechanism, they only declare their intent as opposed to the means by which it is realised.

Of course, the lambda expression in Listing 6.2.2 may still reference global variables or invoke operations containing arbitrarily complex code, however the clearly defined semantics of the iteration operation's purpose are what matter most for our purposes. In essence, we can treat the lambda expression as a black box operation which takes as input an element form the source collection and produces as output a Boolean. This is the definition of a predicate, and in true functional style, the evaluation of this predicate for a given element should be unaffected by the result or order of evaluation of other elements.

In this subsection, we exploit this data independence and the single instruction multiple data (SIMD) – or in this case, single expression, multiple elements – of these declarative operations to demonstrate how data parallelism can be introduced for almost all such operations. In doing so, we provide a library of useful operations for performing common queries and transformations on collections of data (model elements) with parallel execution. Furthermore, we aim to ensure that the parallel variants are functionally identical to their sequential counterparts in their results from the users' perspective for consistency and to maximise compatibility with a wide range of use cases.

Most first-order operations in Epsilon are "select-based" or "collect-based", as will become apparent in the subsequent subsections. In general, there are two types of operation executions: those which execute the given lambda expression on all elements, and those which execute it until a certain condition is met (e.g. the *exists* operation). The latter are *short-circuiting* predicate-based operations, which are more efficient but also more complex to parallelise whilst retaining the short-circuiting behaviour. We therefore need to devise a more complex parallel execution mechanism to retain short-circuiting behaviour. Table 6.2.1 shows the first-order operations and their properties. The type *T* refers to the type of the source collection, whilst other letters refer to derived types. Although not all of the operations in the table exist in OCL, we chose to implement these in order to deeply explore the challenges with converting from sequential to parallel execution algorithm.

Table 6.2.1 Summary of Epsilon declarative collection operations

Operation	Short-circuiting	Return Type	Lambda Type
select	✗ No	`Collection<T>`	`Predicate<T>`
selectOne	✓ Yes	`T`	`Predicate<T>`
reject	✗ No	`Collection<T>`	`Predicate<T>`
rejectOne	✓ Yes	`Collection<T>`	`Predicate<T>`
exists	✓ Yes	`Boolean`	`Predicate<T>`
forAll	✓ Yes	`Boolean`	`Predicate<T>`
collect	✗ No	`Collection<R>`	`Function<T, R>`
sortBy	✗ No	`List<T>`	`Function<T, Comparable<?>>`
mapBy	✗ No	`Map<R, List<T>>`	`Function<T, R>`
aggregate	✗ No	`Map<K, V>`	`Function<T, K>`, `Function<T, V>`
closure	✗ No	`Collection<R>`	`Function<T, R>`

6.2.1 *select / reject*

The *select* operation is a filter on the collection, returning a subset for which the given predicate is true. Since this operation's processing logic can be re-used, there are two additional internal parameters. The first of these is whether the operation should return when a match is found (i.e. the predicate evaluates to true), and the second is whether the operation is a reject or select. With these parameters, the first four operations in Table 6.2.1 can be handled by a single code base in the sequential implementation. *reject* can also be expressed using *select*, since `c.reject(i | `*`<p>`*`)` `=== c.select(i | not `*`<p>`*`)` for a predicate *p* and collection *c*. In the sequential implementation, it is also relatively trivial to implement *selectOne* and *rejectOne* in the same code as *select* and *reject*. *selectOne* (“*any*” in OCL) returns the first value which satisfies the predicate, whilst *rejectOne* adds all of the source collection to the results and removes the first value which does not satisfy the predicate. The implementation becomes more complex when executing in parallel, so the same approach and degree of re-use is not as simple. However since *reject* is effectively the same operation as *select* with negation of the predicate, we can apply the same approach for *parallelSelect* as in the sequential variant. In both cases, the predicate needs to be evaluated on the entire source collection and a collection is returned.

As with all other first-order operations, we can execute *select* in parallel by submitting a new job to an executor service such as a thread pool (which takes care of managing thread lifecycles and mapping jobs to threads), for each element in the collection, so that the predicate is then evaluated for each element independently. In the sequential implementation, every time a matching value is found it is added to the results collection. However when executing in parallel, this would require synchronization on the result collection for every write, which would greatly diminish the performance gains from parallelism. There is also the matter of ordering the results in a consistent manner with sequential *select*. For this reason, we considered two implementations: *parallelSelectUnordered* when order is immaterial, and *parallelSelect* when ordering is desired.

If encounter order is to be preserved, one solution is to use Futures. The idea is that when each job is submitted to the executor service, an object which encapsulates the result of the job's computation is returned, and can later be used to retrieve the result. We therefore add all of the futures to a collection and subsequently loop through this collection and get the result for each job in the order that they were submitted (ordering is guaranteed since we add the futures and request their results sequentially). Although getting the result from a Future is blocking, since *select* requires us to operate on every element of the source collection, all results must be obtained anyway.

Collecting the results is also not as straightforward as in the sequential case. Recall that a Future returns the result of an asynchronous computation. In the case of *parallelSelect*, this result is not always present. Although the type of the result is determined – the same as that of the source Collection's elements – we also need a mechanism to signal the absence of a result for values which the predicate is false. Note that this cannot simply be *null*, since an item for which the predicate is true may also be null e.g. in the case of `c.select(i | i == null)`. We therefore wrap the result of the computation into an *Optional* (or any other arbitrary container, such as a singleton collection). This way, if the Optional itself is null, we know there was no value present. If the value is present, it can be represented by an Optional (a null value is represented by an empty Optional). Once all jobs have completed and their values obtained, we can add the contents of all non-null Optionals to the results collection sequentially, eliminating the need for a thread-safe collection.

Although the solution with Futures guarantees ordering, it does not offer the best performance due to the overhead of creating additional wrapper objects. For improved throughput, an alternative for gathering results would be to use persistent thread-local collections – essentially, a map of threads to their respective collection of values – and merge results at the end. We use this implementation in *parallelSelectUnordered*. Another option would be to annotate objects with their encounter order and restore this order in the returned collection, however this would pose additional sorting and object creation overhead.

Our naming convention provides a hint as to which implementation is the default. Although theoretically the ordered variant has greater overhead, in practice we found no noticeable difference in performance between the ordered and unordered variants. Furthermore, the ordered variant provides greater compatibility with existing programs which may rely on ordering. Indeed, if the operation is invoked on an ordered collection, the returned collection will also be ordered in the sequential variant, so it makes sense to replicate this behaviour for the parallel version. Since the unordered variant is largely redundant, we do not show its implementation here and consider only the ordered variant in our evaluation.

6.2.2 ***selectOne***

The *selectOne* operation differs from *select* in that the return type is a single value, rather than a collection of values. This means that the operation may return without evaluating the predicate on the entire source collection, once a matching value is found.

This operation introduces an inconsistency with the sequential *selectOne* in that the returned value may potentially be different for each invocation, whereas the sequential variant will always return the first value which matches the predicate as returned by the source collection's iterator. This is because each computation occurs in parallel and it is non-deterministic which will finish first. However as the name of the OCL operation suggests (“*any*”), the chosen value need not necessarily be the first. If a “*selectFirst*” operation is required, or at least a deterministic result is desired, then the sequential implementation should be used. Alternatively, the *parallelSelect* operation could be used and the first element retrieved from the resulting collection; though this may result in an error (e.g. *ArrayIndexOutOfBoundsException*) if no element is found. Moreover, the benefits of short-circuiting would be lost if opting for *parallelSelect* over the sequential *selectOne*.

6.2.3 *rejectOne*

The *rejectOne* operation is a hybrid of *selectOne* and *select* in that it does not need to evaluate the predicate on the entire source collection, but it returns a collection instead of a single value. Consequently, *parallelRejectOne* is not a simple modification of *parallelSelect* or *parallelSelectOne*. However, since we only need to exclude a single value from the source collection which matches the predicate, we can delegate the predicate to a *parallelSelectOne* operation, and remove the returned value from the source collection. However we still have the issue of determining whether a result was found or not, so that we do not inadvertently remove a null value from the source. This can be circumvented by using a simple Boolean flag on the *parallelSelectOne* operation instance, which is set when a result is found. Since this value will only ever be set from false to true and never queried during parallel execution, there are no concurrency issues to resolve. As with *parallelSelectOne*, this operation is inconsistent with its sequential variant in that the returned collection may exclude an element which is not the first value for which the predicate is not satisfied.

6.2.4 *exists* and *forAll*

The *exists* operation returns true if there is at least one element in the source collection which satisfies the predicate (essentially a logical OR of the predicate on each element). This is therefore shorthand for whether a *selectOne* operation with that predicate has a result, so we delegate to the *parallelSelectOne* operation in our parallel implementation. As will be clear in next subsection, our implementation of *selectOne* returns a Collection. We can simply return whether the collection is not empty, since the presence of a value indicates that a match was found.

The *forAll* operation is similar to *exists* except that it requires the predicate to hold for all elements in the collection (i.e. logical AND instead of OR). We can therefore delegate to the *parallelExists* operation with a negated predicate. That is, if there is an element for which the predicate is not satisfied, we can return without evaluating the predicate for all elements. This proof by counter-example approach retains short-circuiting behaviour whilst maximising re-use.

6.2.5 Select-based operation implementation

Having discussed the semantics of the first-order predicate logic operations and the challenges with parallelisation, we now show how all of the select-based operations can be implemented in parallel in a single Java method, as shown in Listing 6.2.4. We now explain the code in detail.

We start by resolving the source collection from the object for which the operation was invoked on and creating the result collection which matches the type of the source collection. If the source collection is empty, the (empty) result collection is returned. We then allocate a new fixed-size ordered collection which is used as an intermediary for storing the eventual computation of all jobs (that is, each predicate-element pair). We also ensure that the execution context has the appropriate state for parallel execution (if not already so, we create a new context from the current one). We then convert the EOL AST of the operation's expression and iterator variable – i.e. the variable before the vertical bar in the `select(i | <p>)` syntax – into a Java lambda expression and assign it to a custom extension of java.util.function.Predicate variable which can throw an exception. This Java predicate encapsulates the actual execution of the expression for a given element – for example, it maps the iterator variable name (the *i* before the vertical bar in EOL syntax as mentioned previously) to the actual element and places it on the FrameStack. This declarative approach enables us to execute the EOL code without repetition by localising the mechanics into a single code base.

After the first few lines of preparation, we loop through all elements in the source collection and create a job for each one, where each job is responsible for executing the predicate for the given element. The job execution code itself is relatively straightforward since we converted the EOL predicate into a Java one. Since the operation's result may contain fewer elements than the source collection (since it is a filtering operation), we need a mechanism to distinguish between elements which satisfy the predicate and those that do not. For elements where the predicate returns true, the result of the job will be the element itself. However, the element itself may be null, hence the use of Optional as discussed.

If the operation is short-circuiting (i.e. *selectOne*), we need to stop the execution of jobs once an element satisfying the predicate has been found to prevent wasteful CPU usage. Originally, we developed a bespoke *ExecutionStatus* object which encapsulates the result, and could be used to signal to the waiting thread (i.e. the main thread in most cases) that a result was found and that the remaining jobs should be cancelled. Although this worked, the additional complexity introduced by this solution was unnecessary and more difficult to present, reason about and test. We found a much simpler solution which performs identically. Rather than cancelling the jobs in progress, which itself has some coordination overhead, we let them run but alter the execution path such that they would not do anything. This is where our boolean flag comes into play. Since the flag is modified once a result is found, it needs to be both non-final and atomic, so that the value is not cached by the CPU (i.e. it is retrieved from memory every time it is queried). This is an ideal use case for *AtomicBoolean* [285], which provides the volatile and atomic semantics we require. We initialise the `search` flag to true, and once we found a result, set this to false. Since this flag is only relevant when short-circuiting, we avoid unnecessarily querying it in the non-short-circuiting case. This optimisation is relevant because the boolean is both atomic and volatile, meaning a memory access is required as the value is shared amongst threads and not cached.

Finally, we need to perform some post-processing on the results. Since a null Optional indicates an absent result (and empty Optional represents a null value), we filter the null Optionals and map the empty Optionals to null. We add the processed values to the results collection and return it. For the short-circuiting case, we only need to add a single value, and so do not need to process all values.

To make clearer the need for a wrapper type an post-processing of the results, consider the example in Listing 6.2.3, which operates on a collection containing null elements. In the sequential implementation of the *select* operation's algorithm, we can simply continue looping if the criteria (namely, that the element is null) is unsatisfied for a given element. However for the parallel algorithm (as shown in Listing 6.2.4), each job needs to return a result, since a job is a java.util.concurrent.Callable. Since each job operates on a single element, we need a way of signalling whether the predicate was satisfied for the given element. If it is not satisfied, then we need to return "nothing". However, in this example, we cannot represent "nothing" as "null", because the predicate is explicitly requesting null elements. Therefore, we need to distinguish between "nothing" and "null". Hence, we use a wrapper to be able to represent the three possible outcomes. If the predicate is satisfied and the element is not null, then we wrap it as an Optional. If the predicate is satisfied and the element is null, we return an empty Optional. If the predicate is not satisfied, we return null. Once all jobs have completed, we can then filter the results to exclude elements which did not satisfy the predicate, based on this three-valued logic.

```
Sequence{0, null, 1, null, 2}.select(i | i == null);
```

Listing 6.2.3 Example of *select* operation with null elements

```java
public Collection<?> execute(boolean returnOnMatch, Object target,
NameExpression opNameExpr, Parameter iter, Expression expr, IEolContext ctx)
throws EolRuntimeException {

  Collection<Object> source = resolveSource(target, iterators, ctx);
  Collection<Object> resultsCol = EolCollectionType.createSameType(source);
  if (source.isEmpty()) return resultsCol;
  IEolContextParallel context = EolContextParallel.convertToParallel(ctx);
  Collection<Callable<Optional<?>>> jobs = new ArrayList<>(source.size());
  Predicate<?> predicate = resolvePredicate(opNameExpr, iter, expr, context);

  AtomicBoolean search = returnOnMatch ? new AtomicBoolean(true) : null;

  for (Object item : source) {
     jobs.add(() -> {
        Optional<?> intermediateResult = null;
        if ((search == null || search.get()) && predicate.test(item)) {
           intermediateResult = Optional.ofNullable(item);
           if (returnOnMatch) {
               search.set(false);
           }
        }
        return intermediateResult;
     });
  }

  Stream<Optional<?>> resStream = context.executeAll(jobs)
     .stream().filter(r -> r != null);

  if (returnOnMatch) {
     resStream.findAny().ifPresent(r -> resultsCol.add(r.orElse(null)));
  }
  else {
     resStream.map(opt -> opt.orElse(null)).forEach(resultsCol::add);
  }

  return resultsCol;
}
```

Listing 6.2.4 *parallelSelect* execution algorithm

6.2.6 **Parallel job execution**

So far, we have seen that we can implement the *select* operation in parallel whilst maintaining ordering of the source collection in the output, and that a short-circuiting variant is also possible. In this subsection, we explain in greater detail how exactly the `context.executeAll` method works. The (flattened) implementation is shown in Listing 6.2.5.

```
<T> List<T> executeAll(Collection<? extends Callable<? extends T>> jobs)
throws EolRuntimeException {

  initThreadLocals();
  if (executorService == null || executorService.isShutdown()) {
     executorService = newExecutorService();
  }
  isInParallelTask = true;
  ArrayList<T> results = new ArrayList<>(jobs.size());
  try {
     for (Future<? extends T> future : executorService.invokeAll(jobs)) {
        results.add(future.get());
     }
  }
  catch (InterruptedException | ExecutionException ex) {
     EolRuntimeException.propagateDetailed(ex);
  }
  clearThreadLocals();
  isInParallelTask = false;
  return results;
}
```

Listing 6.2.5 Algorithm for parallel job execution

We start by initialising the thread-local containers, which are initially null to save memory. We also make sure the ExecutorService is available to accept jobs. We then set a flag signifying that parallel execution is in progress (the significance of this will become apparent in section 6.5). We then create a new ordered collection for the results of executing each job. Note that since the method accepts a collection of Callables, each one may return a result. In the case of *select* operation for example, it is the element (although wrapped, as described in the previous section). The crucial part in the listing is `executorService.invokeAll`, which is a standard functionality in the ExecutorService API [286]. This method takes care of submitting the jobs to the ExecutorService and waiting for all of them to complete. As the documentation states, the return is "a list of Futures representing the tasks, in the same sequential order as produced by the iterator for the given task list, each of which has completed" [286]. Hence, we can loop through this list and get the completed result, adding it to our results collection. Finally, we clear the state of thread-local variables (so that they are ready for use next time without any "contaminated" / stale state left over), reset the parallel execution in progress flag and return the results.

Whilst the main abstraction here is provided by the ExecutorService API, our initial design was much more complex, using a bespoke implementation and mechanism for co-ordinating parallel execution and short-circuiting. The details of this alternative are described in Appendix III.

6.2.7 **Transformation operations**

Having covered the parallel execution mechanics of the first-order collection operations, we now turn to operations which perform transformations on the source Collection rather than querying them based on a predicate. Since the complex execution mechanics have already been described, for brevity we only show the novel differences, excluding the setup code.

6.2.7.1 *collect*

We start with perhaps the simplest and commonly used transformation operation: *collect*. The input expression to this operation returns an object from the given element – typically, a property derived from the input element. In this case the size of the result collection is the same as the input one – it is a one-to-one mapping. We therefore don't need to wrap values – we just execute the expression on each element in parallel and return the results, as shown in Listing 6.2.6.

```
Function<?,?> function = resolveFunction(opNameExpr, iter, expr, context);
Collection<Callable<?>> jobs = new ArrayList<>(sourceSize);
for (Object item : source) {
    jobs.add(() -> function.apply(item)));
}
return context.executeAll(jobs);
```

Listing 6.2.6 Code extract from *parallelCollect* operation

6.2.7.2 *sortBy*

An extension of the *collect* operation is *sortBy*, which returns a sorted copy of the collection according to a derived comparable property. Internally, this operation uses the decorator pattern and assumes the lambda expression returns a Comparable Java object (e.g. a String, integer etc.). The operation works by first obtaining the comparable properties by executing the expression on all source elements, and then creating a new object from each of these containing the element itself and the comparable property. These objects are placed into an array and sorted using the *java.util.Arrays.sort* utility method. The first stage of this operation delegates to *collect* to retrieve the comparable properties, so we can just replace the delegate with *parallelCollect*. The sorting itself can also be parallelised using *java.util.Arrays.parallelSort*, which uses the JVM's common Fork/Join pool to recursively sort sub-arrays in parallel. However below a certain size (at the time of writing, 8192 elements), the parallel sorting becomes sequential, since it uses a divide-and-conquer approach and it becomes inefficient to sort smaller arrays in parallel. Therefore we can use parallel sort in the sequential operation, and only change the delegate operation for *parallelSortBy*.

6.2.7.3 *mapBy*

Another transformation operation is *mapBy*, which returns a Map (key-value pairs) with the key being derived from the lambda expression and the value being a list of elements which share the same key. Since this operation returns a multi-map (that is, a key which can have multiple values associated with it), there is an inherent need for synchronization because the key may or may not be present in the results collection at any given time, and the collection of associated values also requires synchronization for writes. To avoid this synchronization requires a fundamentally different execution algorithm. Our solution once again achieves data-parallelism in a similar manner to the *parallelCollect* operation, however instead each job returns a tuple; representing a mapping from the derived result (as obtained by executing the expression for the given element) to the element itself. We can then merge the individual entries (tuples) afterwards, where we resolve duplicate mappings by joining the mapped values into collection for each duplicate entry key, which can be performed in a declarative manner using Java Streams and Collectors API. The algorithm is shown in Listing 6.2.7. The *Collectors.toMap* method takes as input four functions and outputs a Map from the input Stream. The first two take as input an element from the stream – in our case, an Entry. The first function derives the key for the map – in our case, this is just a reference to the existing entry's key. The second derives the value for the element. Since we want to build a *Map<K, List<V>>*, we create a new collection and add the entry's value to it. The third function acts as a merger for key collisions, taking as input the values of the colliding keys. Since these are both collections, we copy the values from one of these to the other and return the merged collection. The fourth function is actually a Supplier, returning a constructor reference for a new Map.

```
for (Object item : source) {
   jobs.add(() -> new SimpleEntry<>(function.apply(item), item));
}
Collection<Entry<?, ?>> intermediates = context.executeAll(jobs);
return intermediates.stream()
     .collect(Collectors.toMap(
        Entry::getKey,
        entry -> {
           EolSequence<Object> value = new EolSequence<>();
           value.add(entry.getValue());
           return value;
        },
        (oldVal, newVal) -> {
           oldVal.addAll(newVal);
           return oldVal;
        },
        EolMap::new
     ));
```

Listing 6.2.7 Code extract from *parallelMapBy* operation

6.3 Optimised sub-collection counting

So far, we have demonstrated parallel algorithms for both short-circuiting and non-short-circuiting operations. In this section, we present parallel algorithms for two new optimal operations which did not previously exist in EOL or OCL. The purpose of these operations is to improve the performance of expressions of the form `select(…)->size()`. This is a very common use case of the *select* operation, however just as `select(…)->get(0)` can be replaced with *selectOne*, so too can operations of this form. In these expressions, the resulting collection is not used, so an intermediate collection is not needed. It is also possible in some circumstances to make these expressions short-circuiting if the size is compared to a constant.

6.3.1 *count*

Consider the query in Listing 6.3.1, where the intention is to count the number of elements satisfying some arbitrary predicate. Naturally, a user may express this as a *select* and then query the size of the resulting collection.

```
Car.allInstances()->select(c | c.brand = 'BMW')->size()
```

Listing 6.3.1 Suboptimal expression of a counting operation

However, building a sub-collection can be wasteful, especially when the source collection is large and the sub-collection contains many elements. Since the user only needs a count of the number of elements which satisfy the specified criteria, an intermediate collection is not needed. Instead, we can simply increment an integer when a predicate returns true for a given element. This avoids the overhead of creating a new collection and copying elements over to it.

```
Car.allInstances()->count(c | c.brand = 'BMW')
```

Listing 6.3.2 Optimised expression of Listing 6.3.1

As for parallelisation, we can use an *AtomicInteger* [287], which uses direct memory access to perform compare-and-set operations on an in-memory operation. That is, increments to the integer happen atomically (a single JVM and CPU instruction) with reads and writes directly to and from memory, which eliminates the need for synchronization. Listing 6.3.3 shows the parallel execution algorithm (excluding the setup code).

```
final Collection<Runnable> jobs = new ArrayList<>(source.size());
final AtomicInteger result = new AtomicInteger();

for (Object item : source) {
   jobs.add(() -> {
      if (predicate.test(item)) {
         result.incrementAndGet();
      }
   });
}

context.executeAll(jobs);
return result.get();
```

Listing 6.3.3 *parallelCount* execution algorithm (extract)

6.3.2 *nMatch*

Although the *count* operation avoids building an intermediate collection, it does not fundamentally improve the execution time in the same way that *selectOne* improves a *select* operation which requires a single element. However in many cases, the result of a *count* (or *select* followed by *size*) is compared to another integer (or at least, an expression returning an integer), resulting in a Boolean. An example of this is shown in Listing 6.3.4.

```
Car.allInstances()->select(c | c.brand = 'BMW')->size() >= 9000
```

Listing 6.3.4 Suboptimal expression of a size requirement operation

The user's intention here is to determine whether there are at least *n* elements (in this case, 9000) which satisfy the predicate. Since the size of the source collection and *n* are known in advance, we can derive a new operation with short-circuiting behaviour. For example, suppose that the size of the source collection (referred to as *ssize* herein) is less than *n*. Immediately we can determine that the expression in Listing 6.3.4 will be false, which eliminates the need for evaluating the *select*. Similarly, if *ssize* is 10,000 and *n* is 9000, then if after evaluating 3000 elements we have found 1999 matches (instances satisfying the predicate), we can reason that even if the remaining 7000 elements satisfied the predicate, the total number of matches would be 8999, one short of the required 9000. Thus, we do not need to evaluate the predicate on the remaining elements – we can return false. Likewise, if *ssize* = 50,000 and we find 9000 matches after 16,000 evaluations, we can skip the remaining 34,000 elements and return true. Of course, the semantics of the operation can be extended to support not only a minimum number of matches, but also an exact number or even a maximum (upper bound). In all cases, short-circuiting is possible. After each evaluation, we can invoke a function which determines whether short-circuiting is possible based on the number of evaluations so far, the size of the source collection, target number of matches and the remaining number of matches. This logic is shown in Listing 6.3.5.

```
boolean shouldShortCircuit(int ssize, int tMatches, int cMatches, int cIndex) {
  // Short-circuit if we met / exceeded the number
   return // First check whether we've already met the criteria
   (cMatches > tMatches || (mode == MatchMode.MINIMUM && cMatches == tMatches))
||
   // # of remaining elements  vs # of remaining matches
   (ssize - cIndex) < (tMatches - cMatches);
}

boolean determineResult(int currentMatches, int targetMatches) {
   switch (mode) {
      case EXACT: return currentMatches == targetMatches;
      case MINIMUM: return currentMatches >= targetMatches;
      case MAXIMUM: return currentMatches <= targetMatches;
      default: return false;
   }
}
```

Listing 6.3.5 Code extract demonstrating *nMatch* operation semantic variants

The parallel execution algorithm shown in Listing 6.3.6 is similar to *parallelCount* (and *selectOne*) except that we keep track of the number of elements evaluated as well as matches found so that we can determine whether short-circuiting is possible. As with the short-circuiting variant of *parallelSelect* (i.e. *parallelSelectOne*), we use a concurrent flag to determine whether we should continue evaluating. Note that there is an implicit design decision here with the use of this flag. As previously mentioned, reading this flag is more expensive than a typical boolean, so an alternative would be to call the *shoulShortCircuit* method at the start of each job, since that only involves comparing integers. However since the parameters to this are also atomic and there are multiple AtomicIntegers, we end up doing more reads that way, hence the use of this flag.

```
final int ssize = source.size();
if (mode != MatchMode.MAXIMUM && ssize < targetMatches) return false;
AtomicInteger cMatches = new AtomicInteger();
AtomicInteger evaluated = new AtomicInteger();
Collection<Callable<?>> jobs = new ArrayList<>(ssize);
Predicate<?> predicate = resolvePredicate(opName, iter, expr, context);
AtomicBoolean search = new AtomicBoolean(true);

for (Object item : source) {
   jobs.add(() -> {
      if (keepSearching.get()) {
         final int cInt = predicate.test(item) ?
            cMatches.incrementAndGet() : cMatches.get(),
            eInt = evaluated.incrementAndGet();

         if (shouldShortCircuit(ssize, targetMatches, cInt, eInt)) {
            search.set(false);
         }
      }
   });
}

context.executeAll(jobs);
return determineResult(cMatches.get(), targetMatches);
```

Listing 6.3.6 *parallelNMatch* execution algorithm (extract)

The operation in Listing 6.3.4 from the user's point of view can be refactored as shown in Listing 6.3.7. Notice how by going further with the functional approach of using dedicated operations with a specific purpose, we not only make the code more concise but also improve performance. Where possible, we advocate the development of operations such as *nMatch* for other commonly used operation chains.

```
Car.allInstances()->atLeastNMatch(c | c.brand = 'BMW', 9000)
```

Listing 6.3.7 Optimised expression of Listing 6.3.4

6.4 Efficient operation chaining

In section 6.1, we showed how even a functional / declarative approach to model querying can leave the user with a toolset of powerful operations which can be misused, or at least suboptimal. In the previous section, we showed how further specialisation (i.e. more specific operations) can improve both performance and expressiveness. However from a user's perspective, optimisation of queries is the concern of the internal execution engine. In an ideal world, a truly declarative approach would allow the user to write their queries in any way which is intuitive to them and be automatically optimised by the engine. For instance, in the previous section we saw an example of how the generic *select* operation can be specialised to provide more efficient and even short-circuiting semantics. To take advantage of these optimisations, the user must be aware of the existence of the optimised version, which is difficult to expect when the operations are new and not present in other languages. In general, the more specialised an operation, the less likely it is to be widely known by users and supported by languages. Furthermore, languages such as EOL and OCL are arguably designed to be usable by domain experts and not expert software engineers; at least to a greater extent than general-purpose programming languages. Indeed, if OCL-like languages are not significantly more usable for non-technical people than general-purpose languages, then the performance overhead makes these languages more difficult to justify even in model-based projects. In recent years, mainstream object-oriented languages such as Java and C#, which have traditionally been associated with an imperative style of programming – have embraced the benefits of declarative programming, adding features from functional languages such as lambda expressions. Furthermore, modern languages such as Kotlin and Scala have been developed to make writing code on the JVM platform significantly less cumbersome, with features that make even the language itself extensible.

As discussed in the literature review, one of the cornerstones of functional programming is lazy evaluation, which enables greater efficiency by avoiding unnecessary computations. With the release of Java 8, the keenly anticipated lambda expressions were added to the language. Although this was a major feature for the Java language, the introduction of java.util.stream package has also had a large impact on how the language is used, with some arguing that Streams are a somewhat more magnificent feature [288]. On the surface, Streams appear to be very similar in style and functionality to the collection operations in EOL though with different names (e.g. "*collect*" is "*map*", "*select*" is "*filter*", "*exists*" is "*anyMatch*" etc.). However the key difference is in the execution semantics. The EOL / OCL operations are implemented in a traditional manner where the result is computed in full for a given input collection and lambda expression. Take for example the query in Listing 6.1.1 as described in section 6.1. The execution algorithm is as follows:

1. Load all instances of *Car* into memory
2. Iterate through (1) and return a subset for which the *year* property is greater than 2017
3. Iterate through (2) and return a collection of each *Car*'s *brand* property
4. Iterate through (3) and find any brand string that is equal to "BMW".

Although this algorithm appears to directly match the user's intention as expressed by the declarative operations, the fact that all of the operations are chained (and that they are functional, thus side-effect-free) indicates that the user only cares about the value of the overall expression, which in this is a Boolean. The intermediate operations are only there to aid with expressiveness.

From the user’s point of view, Listing 6.1.1 and Listing 6.1.2 are identical. Yet in practice, the execution algorithm and consequently computational cost are substantially different.

Java Streams addresses this by employing a “vertical” (as opposed to “horizontal”) execution strategy. The algorithm becomes as follows:

1. Take the *nth Car* instance
2. If the *Car*’s *year* property is greater than 2017, proceed to (3)
3. Proceed to (4) using the *Car*’s *brand* property as input
4. If the input is equal to “BMW”, return true, else go to (1)

In other words, the declaratively expressed query is transformed into a much more efficient, minimal computation algorithm as would be expressed by imperative means (see Listing 6.2.1). The mechanism by which this is achieved is to return a Stream of data at each stage rather than the result of executing the individual operation. Instead of evaluation the *select* for all *Car* elements, and then the *collect* for the subset returned from *select* and then finally the *exists*, each *Car* element is fed through the pipeline until a match is found. For example, the first car will be tested against the *select* predicate, and if it passes, the *collect* and *anyMatch* will be executed on it. If not, then the second *Car* will be tested and so on. By pipelining these computations, we effectively fuse the logic into a single operation, making it semantically equivalent to the more efficient expression in Listing 6.1.2 (or indeed Listing 6.4.1). This is only possible through laziness, so a consequence is that Streams need to distinguish between intermediate (lazy) and terminal (eager) operations; the latter of which trigger the computation. Intermediate operations such as *filter* and *map* effectively attach a computation stage to the processing pipeline, whilst operations which ask for data, such as *findAny*, *forEach* and *allMatch* trigger the processing. From the user’s point of view, the query in Listing 6.1.1 can be written using Streams as shown in Listing 6.4.2 with minimal changes and no impact to expressiveness. whilst being much more efficient.

```
for (Car car : Car.allInstances()) {
   if (car.year > 2017) {
      String b = car.brand;
      if ("BMW".equals(b))
         return true;
   }
}
return false;
```

Listing 6.4.1 Efficient imperative execution algorithm for Listing 6.1.1

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

Listing 6.4.2 Stream representation of Listing 6.1.1

Furthermore, Streams are inherently parallelisable thanks to their functional properties (and assumption that the lambda expression parameters do not depend on mutable global state). This parallelism is achieved through a divide-and-conquer (or Fork/Join) approach. Underpinning the fundamental mechanism of Streams is the Spliterator [289], which is essentially an Iterator which can be divided (or "forked") into sub-iterators. Spliterators also have certain characteristics such as mutability, null support, concurrency, size, uniqueness, ordering etc. Fundamentally it is the divisibility which enables the Fork-Join style of parallelism.

Listing 6.4.2 shows how we can use Streams in Epsilon. In theory, one would imagine that supporting Streams would be simple, given that EOL already has the infrastructure for invoking native (i.e. Java) code and is semantically Java-like (unlike OCL). However a non-trivial piece of language engineering is needed to make the experience of using Streams be as simple as it would be in Java. One of the challenges comes from modifying the parser and AST to recognise the first-order operation syntax as a parameter. Since all method invocations in Java take Objects as parameters, it is necessary to convert EOL lambda expressions into the appropriate *FunctionalInterface* [271] (i.e. an interface with a single abstract method) based on the operation being invoked. Furthermore, since the parameters to the single abstract interface method could be numerous, and the operation being invoked may take other parameters (i.e. not just a single lambda expression), it is necessary to modify the grammar to be able to distinguish between parameters to an interface (lambda expression) and parameters to the operation itself. To demonstrate, consider the *reduce* operation [290], which takes as input an ordinary object as the first parameter, and a *BinaryOperator* (i.e. a function which takes as input two parameters of the same type) as the second parameter. The example given in the documentation is a very simple use case of summing integers from a Stream:

```
Integer sum = integers.reduce(0, (a, b) -> a + b);
```

The complexity of parsing such an expression in EOL comes from distinguishing between the "`0`" (initial / identity value) and the `(a, b)` part, which are the parameters to the lambda expression. We could of course employ the same syntax as Java by requiring parentheses, however syntax and parsing is not the only issue. The more fundamental challenge is in identifying the appropriate operation to be called based on the parameters provided. Since EOL is dynamically typed and interpreted, the lambda expressions used as parameters to method invocations are anonymously typed – the only information we have is the text. With objects, we can easily determine the type at runtime and find the appropriate method to call based on the object types. With lambda expressions, we have no type information to work with – we cannot determine the return type nor the parameter types at runtime. To remedy this, we use the method being invoked and the object it is being invoked on as a hint. Since we know the object which the operation is being invoked on, we can look for operations with that name on the type of type of the object – in the above example, we know that "*integers*" is a Stream, and can find multiple methods with the name "*reduce*". Since Java methods must have unique signatures, we can use the number of parameters and the types, comparing them to the parameters supplied by the user. We can then deduce that for the object parameter and a lambda expression with two parameters, there is only one possible method. Once we have identified this method, we can then trivially infer the type of the lambda expression using reflection, albeit without knowing the type of parameters since generics are erased at runtime. However since we treat everything as Objects anyway, this information is not needed.

With the method signature at hand, the final step is to convert the user-defined EOL lambda expression(s) into the required Java *FunctionalInterface* type. In other words, we need to implement the interface(s) we have inferred at runtime, where the implementation executes the user-defined EOL lambda expression(s) with the specified parameters. This can be achieved by wrapping the implementation into a *Proxy* [291]. The actual implementation itself binds the parameters to the abstract interface method to the EOL variables (in the above example, to *a* and *b*), puts them onto the current FrameStack, executes the EOL expression and returns the result. Of course, we still have to deal with the complications of exception handling and reporting, which need to be legible for end users i.e. reported and handled identically to exceptions for ordinary EOL code. Since we converted EOL lambdas into Java FunctionalInterfaces in our bespoke first-order operations, we do not need to duplicate any logic or testing since the same code underpins the execution of both types of operations. However we still need to ensure that the semantics of Streams are upheld by our implementation, as well as ensuring the correctness of parallel Streams.

Since Streams can be initiated and materialised (triggered) at any point and can be parallel, we need to detect and prevent nested parallelism (the rationale for this is discussed in Section 6.5). The laziness of Streams requires special handling of reflective method calls to Stream operations. Specifically we need to determine an appropriate time to call the *beginParallelTask* method (with abrupt termination parameter set to *true*) in the case of parallel streams. Since a sequential Stream may be converted into a parallel one at any point in the chain by calling the *parallel()* method (although this takes effect for all operations), the fact that all non-terminal operations are lazy means we cannot simply call *beginParallelTask* when we encounter *parallel()* on a Stream. Instead we look at the method being invoked and if the return type is not a Stream then we know that it is a terminal operation. Only then we call *beginParallelTask* if the Stream is parallel. Further complicating matters is that nested parallel Streams are acceptable since they use the same Executor (specifically, the *ForkJoinPool.commonPool*). Nevertheless, we can detect whether a parallel Stream is being invoked within a parallel operation (such as *parallelSelect* or even within parallel EVL, for example) and convert the Stream into a sequential one. However it should be noted that since Streams do not use our custom ExecutorService and rely specifically on Fork/Join, we have less control over the internal parallelisation details. Furthermore, the current implementation of Streams is not extensible, although it is possible to write a custom implementation from scratch.

In summary, the ability to use Java Streams from EOL is arguably a convenient and powerful feature for end users in terms of both functionality and efficiency. One may wonder whether they make our bespoke operations effectively obsolete. Whilst on the surface the introduction of Streams to the underlying language that EOL is implemented is undeniably convenient, there are still advantages of our bespoke operations and drawbacks to using Streams, as will become evident in the performance evaluation. Furthermore, the “black box” nature of the Streams implementation and lack of extensibility make it difficult to provide the same level of support as our bespoke operations due to the aforementioned challenges. Moreover, Streams are not suitable for simple queries and transformations where operations are not chained. Whilst laziness is desirable in long and complex queries, our eagerly evaluated operations are more suitable (and arguably more efficient) than Streams when laziness is not required (i.e. the full results are needed).

6.5 Nested parallelism and re-use

In the previous section, we briefly encountered the issue of nested parallelism, where we had to detect and forbid parallel operations being invoked from within other parallel operations. However, this issue is part of a broader set of language engineering challenges involving parallelisation. In this section, we dive deeper into the technical limitations of our parallelisation infrastructure and outline remedies to these challenges, both from a technical and user-centric perspective.

Executing parallel operations within each other is a problem for numerous reasons. Without modifications to our current parallel infrastructure, the main issue is with the use of thread-locals. When we have a single level of parallelism running at a given time, we know that only one job can be running on a single thread at any given point in time. Since jobs rely on thread-local data structures, this strategy works. However when there are multiple layers of parallelism, the lower (deeper) layers depend on state from the layers above. Since each job has its own thread-local state, the nested layers need to be executed on the same thread as the layer(s) above it, otherwise the wrong state will be inherited. The problem is that we cannot guarantee that a nested parallel job will end up executing on the same thread as its parent. Even if we could ensure this was the case, we would still have a problem because the nested jobs would be modifying thread-local state, which are shared amongst other parallel jobs using the same thread. In other words, nested parallelism is fundamentally incompatible with our serial thread confinement approach.

One possible solution is to use “job-local” (or “nest-local”) rather than thread-local storage, so that each job has its own state space which cannot be accessed from outside of the job. The downside to this is the vastly increased memory consumption brought about by creating new objects for each job – of which there may be millions – as well as significant overhead of the data structures used for mapping jobs to stateful objects. To understand the problem, we need to consider two entities: the *ExecutorService* and *execution context* (referred to as *Context* herein). Although the ExecutorService is itself part of the execution context, for illustrative purposes it is useful to think of them as independent entities. The ExecutorService is the thread pool, which in our case holds a fixed number of threads and handles the execution of jobs. The Context holds engine data structures such as the FrameStack (where variables are stored), execution trace etc. Since most of the mutable data structures in the context are thread-local, there is an implicit dependency between the ExecutorService and execution context internals. So when starting a new parallel task (e.g. a parallel operation), we can either re-use the executor and context, re-use the context and create a new executor or create new context (and thus a new executor) for the parallel task. The problem with each of these is as follows:

If we re-use the context and executor service, we end up with jobs depending on state from the level above, but since we don’t choose which jobs map to which threads, we end up not having that state – everything is mixed together. This is the problem we’re trying to solve. If we re-use the context but not the executor service, we end up creating *parallelism ^ nesting level* number of threads (e.g. with running with 8 cores, a nesting depth of 3 ends up with 512 threads (and thread-locals). In the case of parallel EVL with parallel first-order, it can be much worse since a parallel first-order operation is executed for each element in the context type, and if there are multiple constraints using parallel first-order operations, there could be millions of threads! So, with just a single constraint which has only 1000 elements applicable to it, we could have 1000 * (8 ^ 2) = 64000 threads.

Making matters worse is that all of these threads not only slow down the system, but we end up persisting the thread-locals – at least in the case of parallel EVL – which consumes more memory too. Unlike more modern state-of-the-art concurrency constructs such as Fibers, Threads are comparatively heavyweight, so we should aim to maximise throughput with as few threads as possible. In CPU-bound tasks, this optimum is equal to the number of hardware threads, which is usually quite low (in the order of 10s or even single figures in most cases). Another solution is to just copy the parent context before entering a level of parallel nesting. So calling a parallel first-order operation, we copy the state of the current context into a new parallel context, and use that in the jobs. Once finished, the context will be disposed, along with all of its thread-locals and state (executor service, threads etc.). But we still have the problem of having new threads being created, rather than re-using what's already there, as well as the overhead of copying state potentially millions of times over. Copying is costly, both in terms of time and memory capacity, so this would lead to lower overall throughput.

Moreover, the question is not whether it is possible to support nested parallelism, but whether it is desirable. Since our parallel operations (and task-specific languages e.g. EVL) are data-parallel anyway, even if only a single parallel operation is running at any given time, the hardware threads will already be busy. The only time where this would not be the case is when there are fewer data elements than there are hardware threads; a highly unlikely scenario. Nevertheless, it is possible that the user initiates a parallel operation on a collection with very few elements but then within the lambda expression, performs another operation on a derived collection which has many more elements. Under these circumstances, CPU utilisation is not maximised and nested parallelism may provide performance benefits. However, a more performant solution would be to parallelise the inner operation(s) which have more elements. Unfortunately doing so is not possibly automatically since it is very much dependent on the nature of the application data, so even static analysis could not help us here. However, at runtime we could examine the size of the collection, and if it is smaller than the number of hardware threads, we could opt to not apply parallelisation so that parallelism can be applied when there are more elements. We discuss this further in the upcoming sections.

So far, we have established that our parallelisation infrastructure is incompatible with nested parallelism, which therefore means it is also unsuitable for the divide-and-conquer (or Fork/Join) style of parallel decomposition. Thus, we need a different approach for handling recursively parallel tasks. However, this should not be interpreted as a notion that parallel Streams cannot safely be used, as our evaluation shall reveal. Parallel Streams do not perform nesting naively – they re-use the common ForkJoinPool which is a fixed-size thread pool that is always available, and the threads are never destroyed and re-created. The issue with our approach is not limited to nested parallelism, but more generally to reusing the infrastructure for parallel tasks.

In section 6.2.6 (specifically Listing 6.2.5), we said that when beginning a parallel job, we set a boolean flag signifying that parallel execution is in progress. This is how we are able to detect nested parallelism. Although we did not show this check in Listing 6.2.5, the first thing we do before submitting jobs for parallel execution is check that this flag is false: i.e. that there is not already a parallel job in progress. With parallel Streams, we invoke a method which explicitly sets this flag to true, ensuring first that it is already false of course, otherwise an exception is thrown.

As previously mentioned, the main complication with our approach is the use of thread-locals, which creates a dependency on the threads (and thus the ExecutorService). Even in a non-nested scenario, when a parallel task finishes and another begins (e.g. a *parallelSelect* operation followed by a *parallelExists*), the threads from the ExecutorService are still alive, as are the thread-local data structures, such as the execution stack trace. We do not want to have old state (e.g. from a *parallelSelect*) polluting the new task (e.g. *parallelExists*). Thus, we need to clear all thread-locals and merge any required persistent state into the appropriate main thread structures. We also don't want to have any stale state left over from the ExecutorService. To guarantee that the thread-locals are disposed of, we need to destroy the corresponding threads. So when a parallel task ends, we shut down the ExecutorService and dispose of it, along with any thread-locals. This has the added benefit that no unnecessary objects are kept in memory, and when a new parallel task begins, a fresh state is initialised. Therefore, our parallel execution context only incurs additional memory overhead (as compared to the sequential execution context) when a parallel task is active. Not only does this approach save memory but also guarantees that parallel tasks are independent, which helps to prevent bugs from thread-local states unintentionally leaking or being persisted.

6.6 Automatic and User-defined parallelisation

In the previous section, we tackled the issue of nested parallelism by forbidding it. What we did not consider is how this could be mitigated from a user-centric viewpoint. After all, we have so far assumed that the user is to blame for invoking a parallel operation within another. In this subsection, we will discuss how we can intelligently deal with parallelisation in a way that helps to prevent illegal stages whilst also allowing the user a high degree of control if they desire.

Initially we designed the parallel collection operations not as a replacement for the sequential variants, but an alternative accompaniment when guarantees can be made about the thread-safety of the lambda expressions. Indeed, we assumed that in almost all cases, the lambda expressions used in collection operations would be side-effect-free, making parallelisation of these operations safe and optimal. Since the results produced by sequential and parallel variants are identical, except for *selectOne* where the chosen element is not guaranteed to be consistent – intuitively it seems appropriate to automatically replace sequential operations with their parallel variants. For example, instead of the default *select* operation delegating to the sequential implementation, we could instead delegate to the parallel implementation. However, this is not so simple due to nested parallelism as previously discussed.

In the absence of static analysis, automatic substitution of sequential operations with their parallel implementation can only be performed at runtime, where we have information from the execution context. More specifically, since our Context contains a flag indicating whether we are currently executing a parallel task, we can use this information to determine whether parallelisation is legal. More concretely, when an operation invocation is encountered in Epsilon, we first check for built-in operations before looking for other candidates through reflection. When we encounter *select* for example, we can also check whether parallelisation is legal and if so, delegate to the parallel implementation. Similarly, if the operation starts with "parallel" and there exists an operation where removing the prefix would yield a built-in operation (e.g. "*parallelSelect*" would become *"select*") then we know to delegate to the parallel implementation, assuming one exists for the desired operation of course. Since we're performing automatic substitutions, we still want to give users the ability to choose the implementation if they desire, so we can also do the same for sequential operations, e.g. "*sequentialSelect*". In other words, from the user's perspective there are three ways to invoke a built-in operation such as *select*. "*sequentialSelect*" and "*parallelSelect*" always delegate to the sequential and parallel implementations respectively, even if it is not safe to do so. We trust that the user knows what they are doing and understands its implications, even if it may result in an exception. This is also useful for testing purposes. The default invocation – "*select*" – would automatically choose the appropriate operation, so from the user's perspective, no changes to existing programs are needed in order to take advantage of the parallel operations. The rule is simple: if the Context is not currently executing a parallel task – i.e. the operation is not being invoked from within another parallel operation – then we delegate to the parallel implementation. Otherwise the sequential implementation is chosen.

One slight complication of this substitution approach is that Epsilon's operations are designed to be overridable at the engine level, so that a more optimal implementation can be used for specific applications or modelling technologies. For instance, Epsilon's JDBC driver [231] overrides the *select* and *collect* operations when invoked on model elements to provide lazy semantics akin to Java Streams and to delegate the querying to the actual underlying database containing the model using the SQL *SELECT* statement. This is far more efficient than loading all instances of the model into memory from the database only to throw them away again after the operation, whereas performing the query natively on the database maximises data locality, minimises memory usage and enables exploitation of how the data is structured, since the database will be able to perform optimisations that would otherwise not be possible with the generic EMC API. However, we should re-iterate that these optimisations are applied only when the operations are invoked on an applicable model element type, not to all operations, so deciding whether to delegate to the parallel operation becomes more complex. Making matters even more complicated is that delegation may be multi-layered. For example, the *forAll* operation delegates to *exists* with a negated predicate and *exists* delegates to *selectOne* whose logic is contained in *select*. So, when the user invokes e.g. an *exists* operation, we need to first determine whether the operation being invoked has an actual bespoke implementation or whether it relies on a delegate. This is relatively trivial since all delegate-based operations extend an abstract class. Therefore, if an operation has a delegate, we need to examine its delegate operation and look to replace the delegate with a parallel (or sequential) variant if available. In the case of *exists*, we replace the delegate with *parallelSelect* if it is being invoked from outside of a parallel task. If the delegate is not a known built-in implementation, we leave it untouched in the case of the default invocation. To be clear, if the default *select* operation is invoked on a model element coming from a JDBC model, then the optimised SQL delegate is invoked. However, if the user invokes *sequentialSelect* or *parallelSelect* on the same model element, we always delegate to the known sequential or parallel implementation respectively.

One might argue that the parallel variants of the operations should be scrapped entirely, to avoid issues with nested parallelism. Consider for example Listing 6.6.1. Clearly the user's intention here is to parallelise the inner *select* and *reject* operations, however our automatic application of parallel operations will see that the outer *forAll* is not currently nested within a parallel operation, and delegates to the parallel *forAll* implementation. When the engine encounters the *parallelSelect* operation, it forces delegation to the parallel implementation of the *select* operation, even though this would result in a *NestedParallelismException*. In this case, we're giving the user an illusion of control, when it is evident that naïvely combining automatic and user-defined parallelisation leads to undesirable execution state at runtime. Of course, we are assuming that no static analysis is involved – EOL is interpreted after all, and we are trying to perform these optimisations at runtime, since there is no "compile-time". Under these circumstances, it would be wise to let the user decide through a check box whether they want to specify parallelisation manually or to let the engine handle it. If the user opts for manual specification, no automatic substitution is performed, and the parallel variants become available. If the user opts for automatic parallelisation, the outer-most operations are parallelised, and the parallel variants become unavailable.

```
Person.allInstances().forAll(a |
   a.movies.parallelSelect(m | m.year > 1920)
   .containsAll(a.movies.parallelReject(m | m.year < 2132)
);
```

Listing 6.6.1 Problematic application of automatic parallelisation

6.6.1 Interplay of parallel operations and rule-based parallelism

Up to this point, we have been using the terms "parallel operation" and "parallel task" seemingly interchangeably, however the difference is that a parallel task may be a parallel operation (such as the ones discussed in this chapter) or it could be the parallel implementation of a rule-based execution engine such as parallel EVL. In this chapter, we have focused almost exclusively on EOL without much consideration that EOL serves as the base language for other task-specific model management languages. Although the discussion and operations developed so far are generalisable for all EOL-based model management languages, we have not addressed the interplay between parallel operations and parallel implementations of EOL-based languages, such as parallel EVL. For convenience and generalisability, we will refer to rule-based execution engines as ERL (an internal intermediary between EOL and other languages known as the Epsilon Rule Language) herein.

Whilst any parallel ERL can take advantage of the parallel collection operations and automatic substitution facilities of parallel EOL, given that they use the same parallelisation infrastructure they must also follow the same rules; most notably no nested parallelism. Consider for example the EVL program snippet in Listing 6.6.2, which shows a parallel operation explicitly being invoked within a Rule (we will use the terms "Rule" and "Constraint" interchangeably since we want to generalise to any ERL-based language). The behaviour of this program will be different depending on the execution engine used. If a parallel execution context is used from parallel EOL whilst retaining the sequential EVL execution algorithm, this program will work fine. However, if a parallel EVL execution algorithm is used, then the program will complain about nested parallelism, terminating abruptly with an exception when this operation is encountered. The reason being is that that the parallel EVL engine started a parallel task – albeit where the task is the execution of all Rules – which is (minus the *pre* and *post* blocks) the entire program! Therefore, in any data-parallel ERL implementation, parallel operations can only be used outside of Rules, i.e. within the one-time *pre* and *post* blocks where arbitrary EOL code is executed before and after the rules respectively.

```
@cached
operation AbstractTypeDeclaration getMethods() : Collection {
  return self.bodyDeclarations.parallelSelect(
     bd | bd.isKindOf(MethodDeclaration)
  );
}

context ClassDeclaration {
  constraint hashCodeAndEquals {
    check {
      var hasEquals = self.getMethods().exists(md | md.isEquals());
      var hasHashcode = self.getMethods().exists(md | md.isHashcode());
      return
        (hasEquals implies hasHashcode) and
        (hasHashcode implies hasEquals);
    }
  }
}
```

Listing 6.6.2 EVL program using a parallel operation within a Constraint

Listing 6.6.2 demonstrates the main issue with allowing users to explicitly invoke parallel operations. Assuming that the *getMethods* operation is invoked only from within the *hashCodeAndEquals* constraint, the *parallelSelect* is executed from within a Rule-element pair (i.e. a given *ClassDeclaration* model element and Constraint), so we end up with nested parallelism, at least in cases where the *getMethods* operation is invoked for the first time on a given *ClassDeclaration*. Of course, the behaviour in this Listing is not a problem since the user is explicitly invoking a non-default implementation, thereby declaring that they understand the implications. If the user were to stick with the sensible default *select* operation, the result would be a sequential *select* when a parallel execution engine is used. However, this effectively means we have made parallel operation redundant outside of pure EOL. This is where the annotation-based implementation becomes valuable – refer back to section 4.4.6 for details on this.

Note that the existence of parallel operations does not undermine the viability of rule-based parallelism. The default data-parallel strategies for rule-based languages is still optimal for the majority of cases, since it maximises the degree of parallelisation regardless of whether the user writes imperative (non-parallelisable) or declarative (parallelisable) code. In other words, a user may choose to write a *select* operation imperatively using *for* loops, but still be able to take advantage of automatic parallelisation offered by the engine's rule-based parallel execution algorithm. In most cases, the availability of parallel operations in the *pre* and *post* blocks can further help to improve performance where the rule-based execution engines cannot, due to the arbitrary black box nature of these expression blocks and the lack of common data inputs or structure. Thus, parallel operations and parallel rules are complementary by default, not substitutes.

Combining annotation-based parallelism (for rule-based languages) and declarative parallel operations optimally requires the user to be well aware of the expensive parts of the program and model structure, perhaps by having instrumented the program on numerous invocations with various models, or through strong intuition and expertise. However as previously mentioned, we are trying to go as far as possible with runtime optimisations in the absence of an advanced static analysis framework. Given that models and ERL programs may vary greatly in their structure and computational complexity, our job is to provide them with the necessary tooling to be able to optimise their own programs with relative ease. A trade-off between power (as in customisability) and usability (i.e. how difficult it is to understand and use the tooling for an end user) has to be made at some point. Our solution provides a powerful enough tooling for expert users without placing any burden on novice users; who can ignore the advanced fine-grained parallelisation features entirely and opt for an automated approach. In the latter case, we simply parallelise all rule-element pairs using any of the other strategies.

Ultimately it is down to the user to decide how best to use the tools: intuitively it should be obvious that using a complex execution strategy for simple tasks is suboptimal compared to using an out-of-the-box solution designed to handle the most common and simple cases. Thus we conclude that for most use cases, the annotation-based approach is unlikely to be optimal, however in cases where the user really knows what they're doing and has extensively profiled the program, knows the structure of the model and execution environment and all of the intricacies and parameters which affect performance, they at least have the option to optimally tune the execution in a declarative manner. Of course, one may wonder why an advanced user who understands all of these things would use interpreted languages like EVL in the first place. However, it should be noted that when we refer to "The User", we are not meaning a single person but rather a "Use Case". For instance, let us suppose that a domain expert initially writes an EVL program for validating their models instead of using Java due to the advantages offered by MDE tools. They test it with small sample models for correctness. The organisation then deploys their MDE-based workflow in production, where models are considerably larger. The organisation's performance engineers are then tasked with rectifying this. They then find that the EVL program is the bottleneck. Rather than having to re-engineer the entire toolchain, they can instead profile the EVL program using Epsilon's profiling tools in combination with Java profilers such as Mission Control or VisualVM to find which parts of the code are the most CPU-intensive. They can then use this information to make changes to the EVL script through annotations. This avoids introducing bugs due to domain logic being "lost in translation" when refactoring to a compiled language, whilst still allowing the software engineers to perform optimisations with relative ease. Moreover, since the engineers do not need to re-architect or re-implement anything, they can immediately see the impact of their tuning efforts. This approach is arguably more productive for the organisation as a whole, since the domain experts do not need to migrate to a different unfamiliar or less user-friendly technology, whilst the software engineers have the tooling they need to improve performance with relative ease without needing to modify the underlying runtime code.

6.7 Evaluation

In this subsection, we describe how our parallel operations were evaluated for correctness and performance.

6.7.1 Correctness

Since our parallel collection operations do not require models, they can be executed as pure EOL programs without additional parameters. Thanks to EOL's liberal nature with regards to mixing imperative and declarative programming, as well as its Java-based implementation and semantics, we were able to implement a multitude of thorough tests for the operations using EUnit [292] – a Junit-style testing framework for Epsilon. This allows us to write tests as operations and use assertions to test for equivalence between expected and actual values. An example is shown for the sequential *select* operation in Listing 6.7.1.

For each operation, we write tests for both the sequential and parallel variants. We copy-paste the tests for the sequential operations and change the operation to their parallel variants. These tests use two pre-defined values: the input collection and the expected outcome. When the operation is applied, the result is compared to the expectation. We also test for equivalence between the sequential and parallel operations using different inputs for all operations, with the sequential operations as the oracle (i.e. the expected result). By assuming the sequential variant is correct (having asserted this through previous tests), we can use more complex inputs and lambda expressions since there is no need for manual verification of the results.

```
@test
operation testSelect() {
    var data := Sequence {0..9};
    var selected = data.select(n | (n > 3 and n <= 7) or n == 2);
    var expected := Sequence {2, 4, 5, 6, 7};
    assertEquals(expected, selected);
    selected = data.select(n | n > 10);
    assertEquals(Sequence{}, selected);
    selected = data.select(n | n < 10);
    assertEquals(data, selected);
    data = Bag{};
    selected = data.select(e|e);
    assertEquals(selected.getClass(), data.getClass());
    assertTrue(data.isEmpty());
    assertEquals(selected.size(), data.size());
}
```

Listing 6.7.1 Test for sequential *select* operation

Aside from testing the correctness of results, we also test the execution semantics through more advanced tests. For instance, we check whether global variables are accessible from the operation expressions, and the use of cached operations. We also test the short-circuiting behaviour of sequential operations by using predicate expressions and inputs for which after a certain element, an exception would be thrown due to an invalid property access or null navigation. Furthermore, we test combinations of nested operations with the outer being parallel and inner being sequential and vice-versa, with both short-circuiting and non-short-circuiting operations in both cases. An example is shown in Listing 6.7.2. We also test for equivalence between the sequential and parallel module implementations for a given script to ensure that the internals (e.g. frame stacks) as well as the results are the same regardless of which implementation is used. We also test for exceptions, ensuring that the parallel operations (and parallel module when using sequential operations) report exceptions just as the sequential variants would.

```
var Thread = Native("java.lang.Thread");
var mainThread = Thread.currentThread();

@cached
operation Any lengthAsStr() : Integer {
    return self.toString().length();
}

@test
operation testNMatchScope() {
    var testData := Sequence {-25..16};
    var expected = testData.nMatch(n | Thread.currentThread() ==
            mainThread and n.lengthAsStr() == 2, 16);
    assertTrue(expected);
    var actual = testData.parallelNMatch(n | Thread.currentThread() <>
            mainThread and n.lengthAsStr() == 2, 16);
    assertEquals(expected, actual);
}
```

Listing 6.7.2 Example of “advanced” test for *nMatch* operation

Finally, we test the operations for equivalence with Java Streams in EOL, as well as testing the stream operations themselves in EOL for correctness of both the results and semantics, using the same approach for our custom operations. An example of this is shown in Listing 6.7.3. Since streams exhibit lazy execution semantics, we also test the short-circuiting behaviour when chaining stream operations. We test both sequential and parallel streams for equivalence with our own (both parallel and sequential) operations. We also test the stream exception handling / reporting as with our custom operations. Another advanced test involving streams is nested parallelism, so we test whether the engine can detect and report illegal nesting of parallel streams within parallel EOL operations and vice-versa. There is also a test for the correctness of automatic application of parallel operations, involving both built-in and stream operations, with short-circuiting and non-short-circuiting operations as well as manually specified parallel and sequential operations to ensure automatic parallelisation is applied correctly only when appropriate (i.e. no nesting).

In terms of coverage, the test suite covers 67.2% of EOL all-inclusive. Of the parallel operations, the test suite covered 100% of instructions, and for the base operations package (includes the abstract utilities), 91.9% were covered. As is usually the case with coverage metrics, the 67.2% figure is misleading in that much of what is not covered is not critical code. Based on eyeballing the coverage of the important classes and packages, the meaningful coverage is much closer to 75% and above, without accounting for the usual pitfalls of coverage metrics regarding non-critical code (e.g. *toString* and other trivial methods). The test suite for our operations can be found in the Epsilon repository at https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/tree/tests/org.eclipse.epsilon.eol.engine.test.acceptance/src/org/eclipse/epsilon/eol/engine/test/acceptance/firstOrder.

```
var Collectors = Native("java.util.stream.Collectors");
var testData = Sequence{-9999..99999};

var positiveOddsSquaredEol = testData
  .parallelSelect(i | i >= 0 and i.mod(2) > 0)
  .parallelCollect(i | i * i)
  .asSet();

var positiveOddsSquaredJava = testData.parallelStream()
  .filter(i | i >= 0 and i.mod(2) > 0)
  .map(i | i * i)
  .collect(Collectors.toSet());

assertEquals(positiveOddsSquaredEol, positiveOddsSquaredJava);
```

Listing 6.7.3 Stream and EOL operation equivalence test

6.7.2 Performance

To assess the performance of our parallel operations, we use the very simple and flat IMDb metamodel shown in Figure 6.7.1, with model sizes ranging from 100,000 to 3.53 million. Given that most of the operations are essentially variants of the same few operations, we opted to test only the fundamental operations, such as *select* and *exists* (which delegates to *selectOne*). This way we test both short-circuiting and non-short-circuiting operations for performance whilst minimising unnecessary analysis and benchmarks which detract from the main findings. Of course, we also evaluate our novel *nMatch* and *count* operations, as well as *mapBy* as it is a more advanced variant of the *collect* operation. As per usual, we compare performance to the sequential variant with various number of threads and model sizes to assess the scalability of our implementation.

However, we are also able to perform a broader comparison in light of Java Streams. Not only can we compare our bespoke operations to Streams in EOL, but we can also compare them to Streams written in Java. This gives us a good indication of the overhead of EOL's interpreter and heavy use of reflection vs. natively written Java code. Surprisingly, we found that hand-coding a Java version of the query using Streams is relatively concise, since we can still make use of Epsilon's Model Connectivity API for model accesses, and of course the declarative nature of Streams and lambda expressions in Java further help to reduce the code size and development effort. In this regard, we can question whether the performance delta between compiled and interpreted is large enough to justify the higher development effort required to write the query in Java.

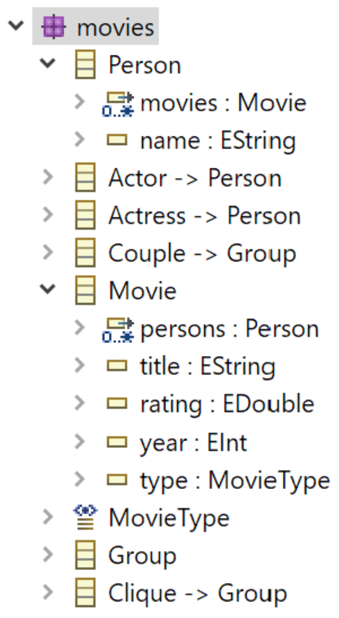


Figure 6.7.1 IMDb metamodel

Of course, we also compare our query to both interpreted and compiled variants of Eclipse OCL, although it is worth noting that such performance comparisons are mostly for motivational purposes, since OCL's semantics prohibit short-circuiting operations. Nevertheless, it also allows us to assess the benefits of migrating from OCL to EOL for an end user who does not need the strict OCL semantics of Invalid, but rather the declarative querying operations and syntax.

We focus our analysis and testing on the query in Listing 6.7.4, which is inspired by a transformation used to evaluate LinTra from the Atenea group at the University of Málaga [293]. It attempts to find couples who have played in at least three movies together. The query is complex enough in its use of operations and property accesses for our purposes and results in sufficiently long execution times to perform meaningfully scaled comparative analysis. We also implemented similar queries using the same IMDb "coactors" logic as in the *select* operation for the *atLeastNMatch* and *count* operations, to compare the performance of our bespoke operations to their closest Java Streams equivalents. The *nMatch* version of the query is shown in Listing IV.1 of Appendix IV.

```
def: coActorsQuery() : Integer = Person.allInstances()
    ->select(a | a.movies->collect(persons)->flatten()->asSet()
      ->exists(co |
        co.name < a.name and a.movies->size() >= 3 and
        co.movies->excludingAll(a.movies)
        ->size() <= (co.movies->size() - 3)
      )
    )->size()
```

Listing 6.7.4 Flattened OCL implementation of *select* operation benchmark

For the remainder of this section, the figures show speedup relative to the baseline (sequential EOL unless otherwise stated) as data labels, and the baseline scenario is labelled with the absolute execution time, for reference.

6.7.2.1 *select* and *count* operations

First, we begin with the non-short-circuiting *select* operation. We have three variants of this: one which uses the bespoke *count* operation (referred to as "count" in the tables), one which uses *select* (referred to as "select") and a Java Streams version which also uses the count operation built in to the API. We refer to the Java implementation of the count operation also as "count", however there is another possibility. We could collect the results and then call *size()*, similar to *select*. We refer to the Java implementation of this as "*select*". Finally, we have an EOL implementation which uses Java Streams, also using the *count()* terminal operation, which we refer to as "filter". It is important to note that all of these operations give the same result, it's just a matter of whether they use *count* or *select(…).size()*.

Table 6.7.1 *select* operation variants 1.5 million elements (R7-3700X system)

Implementation	**Operation**	**Execute (ms)**	**Execute STDEV**	**Speedup**	**Efficiency**	**Memory (MB)**
Interpreted OCL	select	1753972	27362			15046
Sequential EOL	select	1129938	56923	1.573	1.573	38
Parallel EOL (16)	select	191837	6602	9.263	0.579	24
Java (Sequential)	select	39224	763	44.717		3327
Java (Parallel)	select	7424	296	236.257		3958
Sequential EOL	filter	879400	143			250
Parallel EOL (16)	filter	187299	1077	4.695	0.293	1342
Sequential EOL	count	984200	113148			957
Parallel EOL (1)	count	1021882	113713	0.963	0.963	445
Parallel EOL (2)	count	515253	61702	1.91	0.955	892
Parallel EOL (4)	count	265697	20006	3.704	0.926	912
Parallel EOL (8)	count	198439	2197	4.96	0.62	1462
Parallel EOL (16)	count	184237	1631	5.342	0.334	2084
Java (Sequential)	count	36549	1284	26.928	26.928	1165
Java (Parallel)	count	7153	227	137.593	137.593	1236

In Table 6.7.1 (note that the speedup is relative to Interpreted OCL), we see how the same result for the same model can be achieved in vastly different times depending on the implementation. At worst, we have interpreted OCL, taking just under half an hour, and at best we have the hand-coded Java implementation leveraging parallel Streams, taking just over 7 seconds on average. Note that for the *select* operation, we show speedup relative to OCL. We see that EOL is well over 1.5x faster, though parallel EOL is 5.89x faster than sequential EOL in this case. This is slightly better than native Streams, where the parallel version is 5.28x faster. Still, this is well short of the 8x theoretical speedup possible on an octa-core CPU. Perhaps this is due to the complex design of Ryzen, which fundamentally is made up of quad-core clusters. This would explain the sharp decline in efficiency beyond four threads, as evidenced by the *count* operation. We see that speedup scales linearly with the number of threads for the count operation with minimal drop in efficiency: with four threads, we observe a speedup of 3.7x relative to sequential EOL. If we compare for example two threads to the single-threaded parallel EOL implementation, we see a speedup of 1.98x out of a maximum possible of 2x, and even then, this shortcoming can mostly be explained by the slightly lower multithreaded clock speeds. The sharp drop-off in efficiency with 8 threads is clearly a hardware limitation, as we failed to exceed even 5x speedup. That said, SMT clearly helps here (albeit marginally), with 16 threads pushing us to over 5.3x. Another interesting observation is how parallel Streams scale worse under EOL than native Java. We observe a 5.11x speedup in the native case for *count*, yet only 4.7x with EOL. It's also worth noting that in both the native and interpreted cases, the *count* operation (be it our bespoke implementation or Streams) scales slightly worse than the *select* variant.

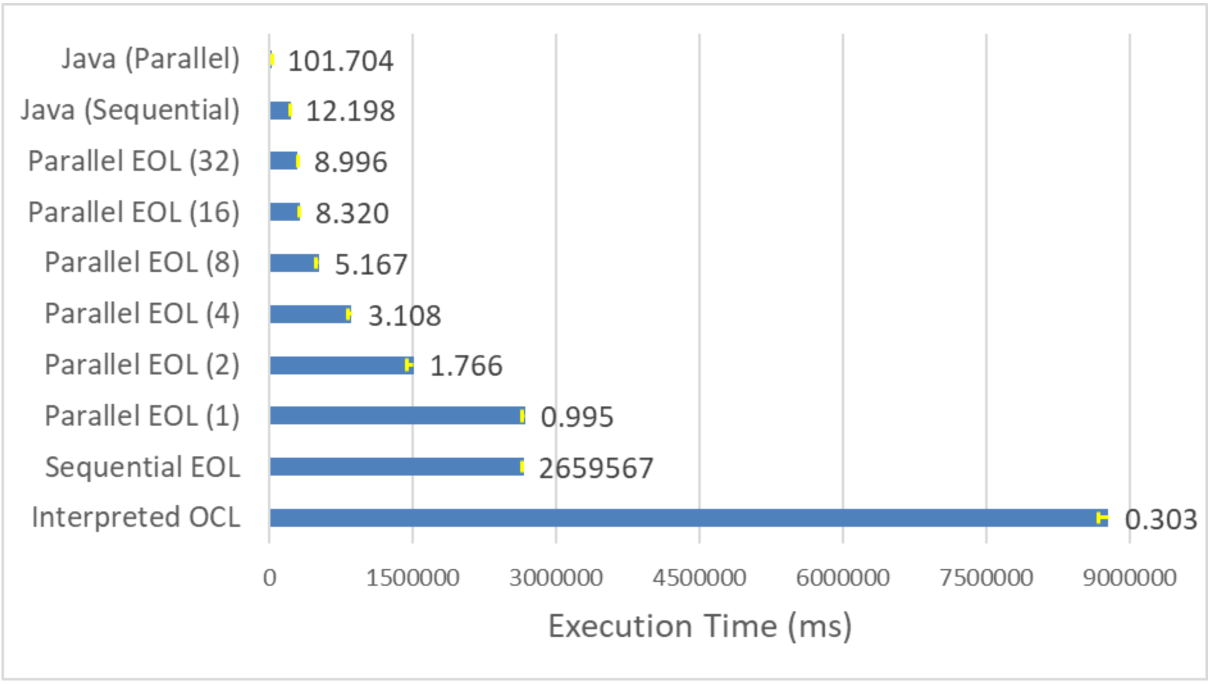


Figure 6.7.2 *select* operation 3.53 million elements (TR-1950X system)

Figure 6.7.2 shows the results for *select* with the largest IMDb model on our Threadripper system. Note that speedup here is relative to sequential EOL. The Java implementation achieves 8.34x speedup with parallel Stream relative to sequential Stream, which is slightly less than the 9x speedup achieved by parallel EOL over sequential. Note however the relatively poor efficiency with lower thread counts. The single-threaded parallel EOL is within margin of error in terms of execution time relative to sequential EOL (judging by the standard deviation), however even with just 2 threads the observed efficiency is lower compared to the Ryzen 3700X. This trend continues until we hit 8 threads, where the speedup is superior to the 3700X. There is a significant uplift in speedup when going from 8 threads to 16, where our intuition based on previous results would suggest rapidly diminishing returns. This can perhaps be explained by the Linux kernel's scheduling algorithm. We ran this experiment with the *-XX:+UseNUMA* JVM flag however we noticed that during the experiment, with fewer threads than there are cores, the logical cores which were at 100% load kept changing. This is normal to evenly spread the heat, however in a complex CPU like this 1950X, where there are two dies and four CCXs, it can negatively impact memory access times. The system uses quad-channel memory, however the NUMA design may be a disadvantage when not utilising all cores, at least with our chosen operating system.

Next, we turn to our laptop system for the *select* operation, where once again we encounter complications, this time from firmware. Specifically, the radically different clock speeds under sequential and parallel execution distort our results, so we compensated by multiplying the speedup by the clock speed differential factor – approximately 2.1x. Table 6.7.2 shows the results. Note that to avoid further complications in the speedup arithmetic, we did not vary the number of threads from the maximum, since the clock speeds would vary under different core utilisations and fluctuate even more unpredictably. This highlights a crucial issue with using laptops – which are far more common than desktops amongst most users and developers – for benchmarking.

Table 6.7.2 *select* operations 1 million elements (i7-8550U system)

Implementation	**Operation**	**Execute (ms)**	**Execute STDEV**	**Raw speedup**	**Effective Speedup**
Sequential EOL	count	1109682	16815		
Parallel EOL (8)	count	469298	18412	2.365	5.003
Java (Sequential)	count	31389	888	35.353	
Java (Parallel)	count	12468	859	89.002	188.273
Sequential EOL	select	1102431	16602		
Parallel EOL (8)	select	475194	8942	2.32	4.908
Sequential EOL	filter	1096682	10040		
Parallel EOL (8)	filter	470808	33039	2.329	4.927
Java (Sequential)	select	32748	2140	33.489	
Java (Parallel)	select	12352	1117	88.786	187.817

Looking at the results demonstrates this: without compensating for clock speed delta under all-core load, the speedups appear to be very disappointing compared to our desktop system: a maximum of less than 2.4x. However, once we compensate for the lower clocks when running the parallel implementations, we see speedups much closer to our expectation, if not perhaps even beyond. Here we observe approximately a 5x speedup over sequential EOL. Interestingly, looking that native Java Streams implementation, we observe an adjusted 5.6x speedup from sequential to parallel Stream. For a quad-core CPU, the parallel implementation appears to be rather efficient, with Hyper-Threading undoubtedly helping here to raise the effective speedup beyond the physical number of cores. This does the raise question on why the 5x effective speedup is so close to that achieved in the Ryzen 3700X system, which has twice the number of cores and threads. We suspect that this is a memory channel bottleneck: both systems use dual-channel memory, and clearly the inability to hit even 6x speedup on our 8-core system indicates a bottleneck, perhaps amplified by the CPU architecture. Since our 4 core / 8 thread is (albeit with adjusted clocks) able to deliver speedups beyond its physical core count, the fact that our 8 core / 16 thread system can't even hit 6x is certainly an indication of some upper limit. The fact that this limit is present in both our bespoke parallel operations and Java Streams is clearly a sign of a hardware bottleneck. This makes sense given the memory-intensive nature of the benchmarks. Also note the absence of interpreted OCL: we attempted to benchmark this but after almost 4 hours, the JVM ran out of memory, despite us allowing a maximum of 90% system RAM for the heap size.

For completeness and the curious reader, we conducted benchmarks for the same 1 million elements model with the Ryzen 3700X system to allow for a comparison between the two systems in question in terms of relative performance in Table 6.7.3. Immediately we can see that the desktop system is almost 2x faster for the sequential *count* operation, and with 8 threads, it's 3.6x faster. However the relative efficiency is still inferior, at least when compensating for clock speeds in the laptop system. Interestingly we see a more marked decline in efficiency with four threads in the 1 million elements model compared to the 1.5 million elements model. On a final note regarding these set of benchmarks, it's interesting to note the efficiency of the single-threaded parallel implementation, which when considering the standard deviation in execution time, has an extremely low overhead compared to the sequential implementation, at least relative to our expectations.

Table 6.7.3 *count* operation 1 million elements (R7-3700X system)

Implementation	**Operation**	**Execute (ms)**	**Execute STDEV**	**Speedup**	**Efficiency**	**Memory (MB)**
Sequential EOL	count	560002	6890			348
Parallel EOL (1)	count	567659	6577	0.987	0.987	465
Parallel EOL (2)	count	287915	4195	1.945	0.973	943
Parallel EOL (4)	count	159564	1813	3.51	0.877	1524
Parallel EOL (8)	count	130436	1775	4.293	0.537	2670
Parallel EOL (16)	count	121859	1249	4.595	0.287	2644
Java (Sequential)	count	18950	454	29.552	29.552	206
Java (Parallel)	count	4590	253	122.005	122.005	2462
Sequential EOL	filter	718409	19355			38
Java (Sequential)	select	21254	238	33.801	33.801	8
Java (Parallel)	select	4443	97	161.695	161.695	9

Out of curiosity, we ran a further experiment on the old Xeon system with the *select* operation and 2 million model elements for good measure. The results are promising, as shown in Table 6.7.4. Here we see a speedup which, when comparing the slowest sequential time and fastest parallel time, exceeds 10x, and averaging over 9.3x without accounting for clock speed differences. We also see that Java Streams benefit less from parallelism, resulting in a mere 6.24x speedup. This is particularly impressive when considering this is a multi-socket setup, so we have two quad-core CPUs with Hype-Threading delivering a greater speedup than the number of physical cores. These results perhaps demonstrate that our solution is particularly scalable in a cloud computing hardware context; or that HotSpot's *-XX:+UseNUMA* flag delivers impressive optimisation. It is puzzling that we have not observed the same calibre of results in more modern single-socket processors. Perhaps this reaffirms our theory that memory channels play a crucial role in non-distributed parallelism, and that Intel's QPI links are well-suited to this kind of work. Furthermore, since there is plenty of memory, we also benchmarked OCL which unsurprisingly performed similarly in relative terms, with sequential EOL being over 1.5x faster. However as previously noted, Eclipse OCL (at least at the time of writing) is extremely memory-hungry, not just slow. To be fair however, OCL's strict semantics are partially to blame, not just the unoptimized implementation. To put the results into practical terms, OCL took 1 hour 40 minutes on average, whilst the native parallel Streams implementation took just over 20 seconds. Meanwhile parallel EOL took less than 7 minutes.

Table 6.7.4 *select* operations 2 million elements (E5520 system)

Implementation	Operation	Execute (ms)	Execute STDEV	Speedup	Memory (MB)
Sequential EOL	count	3805923	35514		1074
Parallel EOL (16)	count	405708	28012	9.381	3454
Java (Sequential)	count	144158	2334	26.401	1371
Java (Parallel)	count	20366	825	186.876	3630
Sequential EOL	filter	3789355	67492		2436
Parallel EOL (16)	filter	406247	22351	9.328	2403
Interpreted OCL	select	5958586	137868		18642
Sequential EOL	select	3837988	161284	1.553	2622
Parallel EOL (16)	select	407806	19771	14.611	3916
Java (Sequential)	select	133770	13661	28.327	3211
Java (Parallel)	select	20335	625	186.346	1007

6.7.2.2 *selectOne* operation

Since other important first-order logic operations are based on *selectOne* (e.g. *exists* and *forAll*), we benchmarked this operation. We used the same logic as for *select* benchmark, except making the criteria stricter since this is a short-circuiting operation. We were also mindful not to use the largest model, since doing so would make the criteria easier to reach and short-circuiting would potentially happen sooner. The results in Figure 6.7.3 show near enough what we would expect based on the previous benchmarks with this hardware. Up to four threads, we observe relatively efficient CPU utilisation. Beyond this, the extra threads have a meaningful impact on speedup, but rapidly diminishing in impact given the increase. To exemplify, consider the jump from 4 to 16 threads: a quadrupling of parallelism with less than 24% improvement in execution time. Although it is somewhat disappointing that we failed to achieve even 5x speedup, clearly the memory-intensive nature of these benchmarks combined with the CPU architecture limit the potential gains. Once again this is not due to our implementation, but the program and hardware, as even the native Java Streams implementation could only attain a 4.47x speedup when using all 16 threads. Finally, it's also worth noting the relatively low standard error in execution times, which were below one second except for parallel EOL with one and two threads.

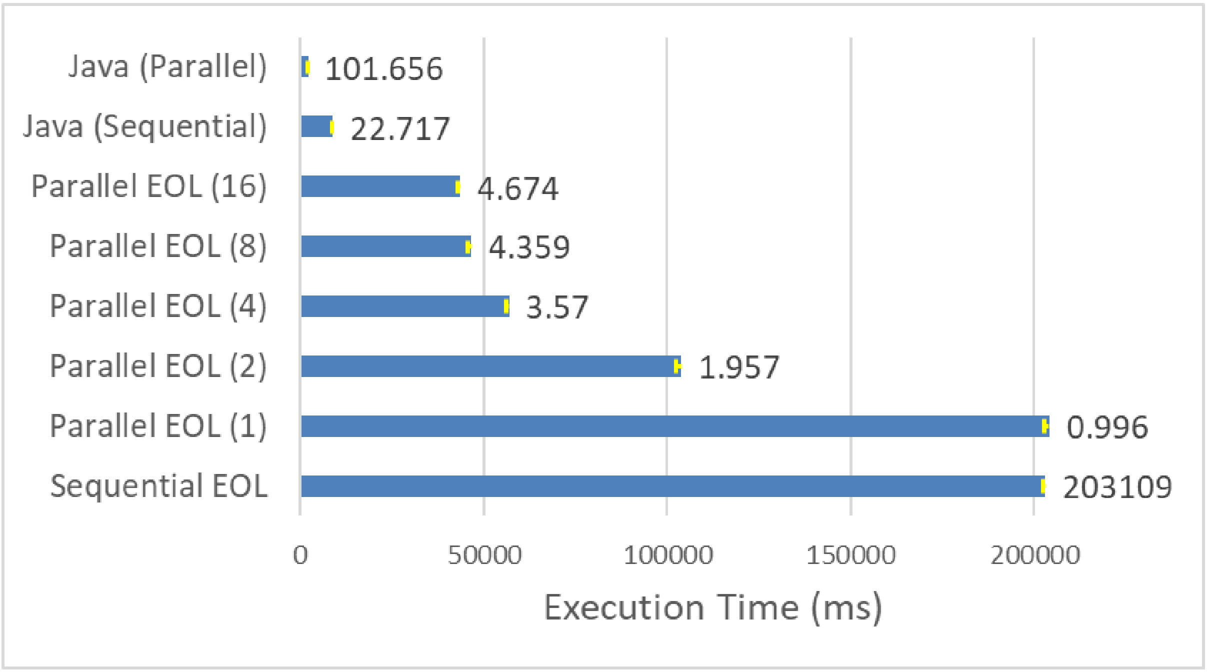


Figure 6.7.3 *selectOne* operation 1 million elements (R7-3700X system)

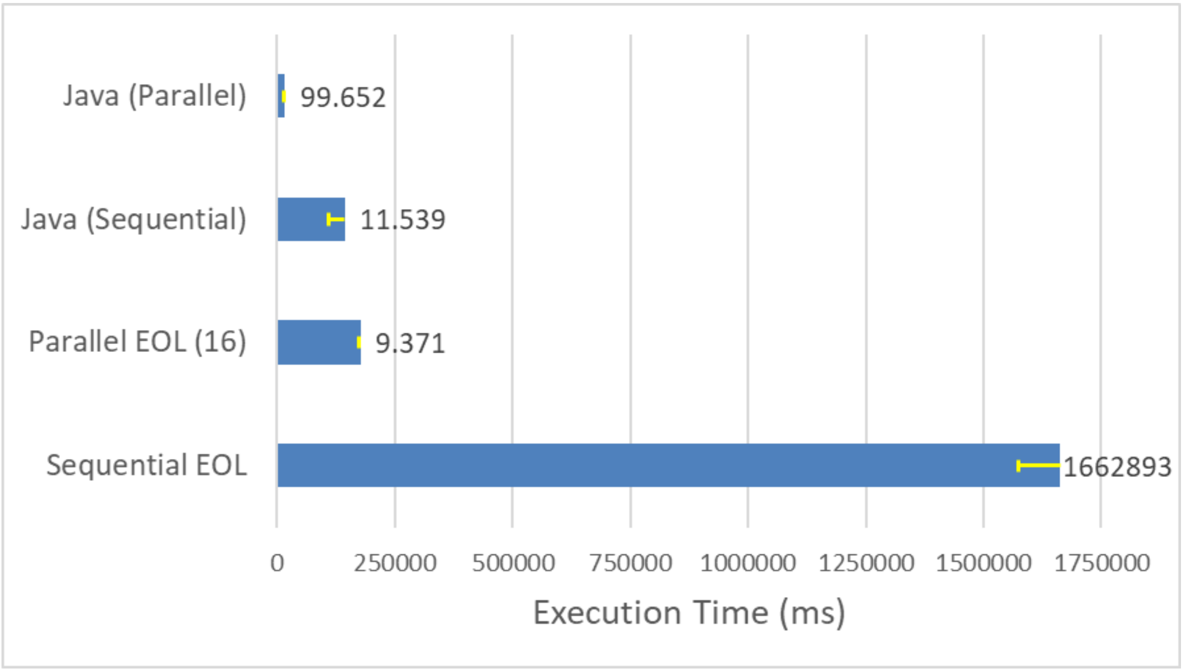


Figure 6.7.4 *selectOne* 3.53 million elements (E5520 system)

We also benchmarked the *selectOne* operation with the largest IMDb model available on the dual-socket Xeon system. As with the *select* operation, we see significantly better efficiency than with our Ryzen system, as shown in Figure 6.7.4. In fact, the speedup with 16 threads is exactly double that of the 3700X system, despite both systems having eight physical cores and with simultaneous multithreading. As previously discussed, it is somewhat perplexing that despite the Xeon system being dual-socket (two physical CPUs), the relative parallel performance is far superior. This is further evidence that the gains from parallelism are extremely hardware-dependent, since in theory there is nothing inherent in our implementation which would favour a particular CPU architecture. This discrepancy in efficiency comes down to a combination of the operating system, JVM and CPU, rather than any fundamental bias in our design.

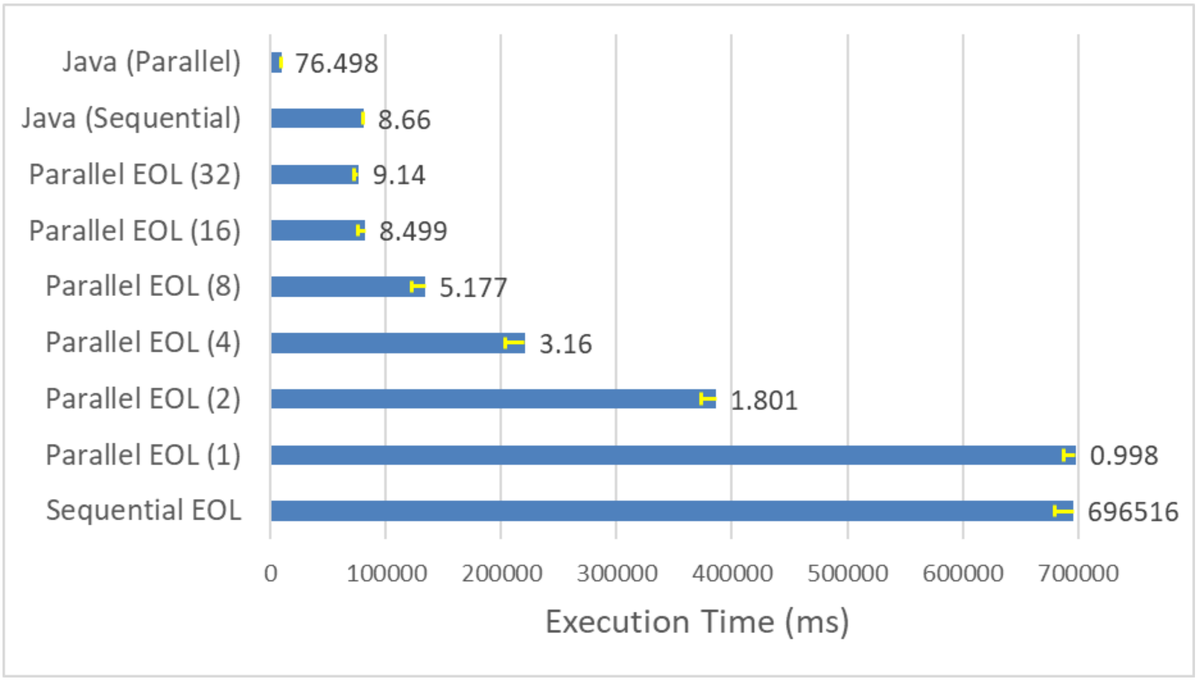


Figure 6.7.5 *selectOne* operation 3.53 million elements (TR-1950X system)

In Figure 6.7.5 we repeated the *selectOne* experiment for the largest IMDb model on the Threadripper system, with unsurprising results. Notice how close the single-threaded parallel implementation is in performance to sequential EOL. This is an indication that although the parallel short-circuiting approach is more complex than the sequential case from a design aspect, in practical terms there is virtually no significant performance penalty judging by this experiment, with an efficiency of 99.8%. The peak speedup is inferior to the Xeon system, despite having twice the number of cores, however the efficiency is very similar to *select*, with 16 threads giving 8.5x speedup and taking full advantage of SMT pushes this to over 9.1x. Meanwhile the Java implementation achieves similar, but slightly lower speedup as before, with 8.83x speedup over sequential Streams.

6.7.2.3 *nMatch* operation

We now turn to the short-circuiting *nMatch* operation. We benchmarked the *atLeastNMatch* variant on the IMDb model.

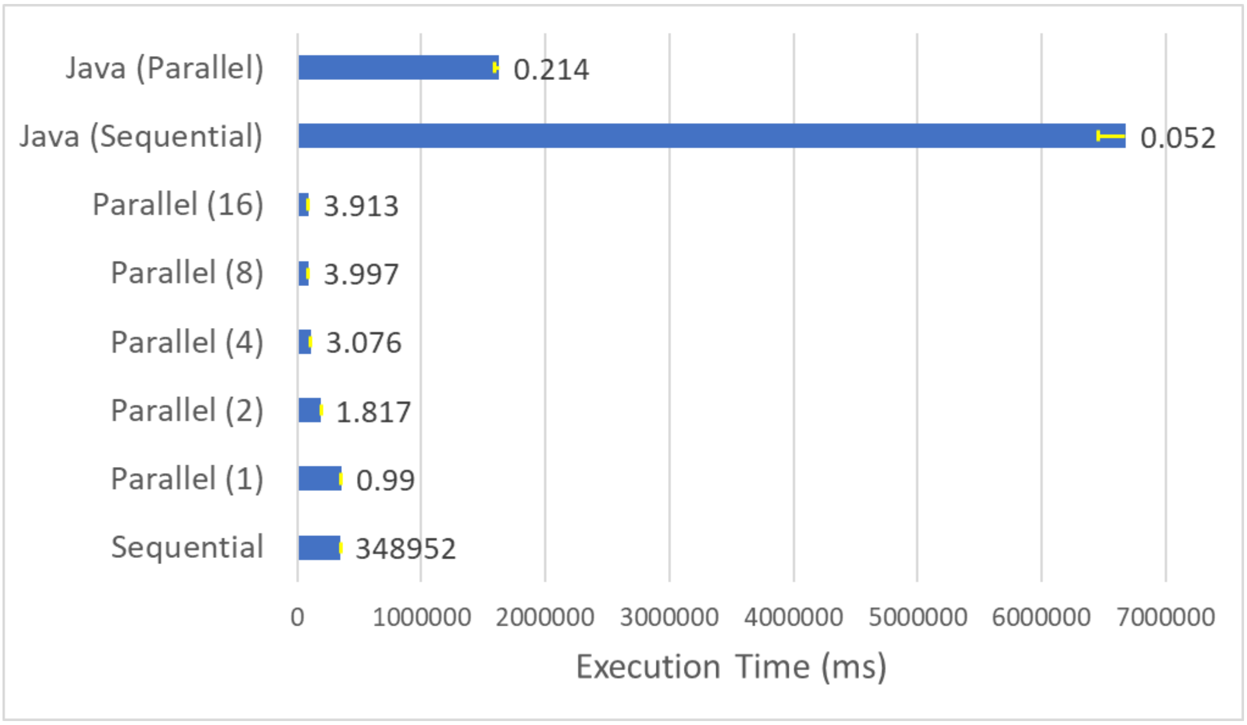


Figure 6.7.6 *atLeastN* 2 million elements (R7-3700X system)

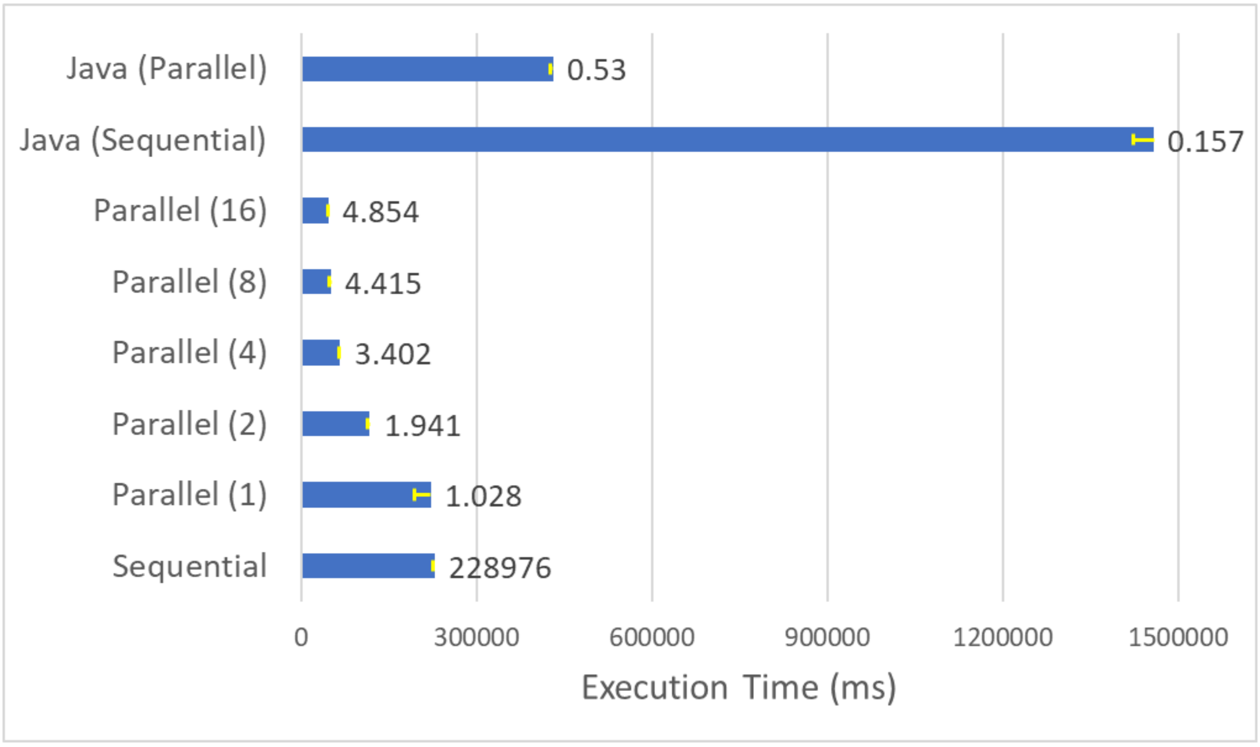


Figure 6.7.7 *atLeastN* 1 million elements (R7-3700X system)

By evaluating the *atLeastN* operation, we now see its true value. Having established in the previous experiments that native Java Streams is much faster than interpreted EOL, the lack of an efficient *nMatch* equivalent operation in the Streams API means the tables are turned: even native parallel Stream is almost two times slower than sequential EOL for both one (Figure 6.7.7) and two (Figure 6.7.6) million model elements. For context, with 2 million elements the native sequential Java code took approximately 1 hour 54 minutes, whereas parallel EOL took around 1 minute 30 seconds: almost a 75x difference! Compared to native parallel Stream, this is still over 18x. Comparing the sequential implementations yields similar results: sequential EOL is over 19x faster in this case.

It is interesting to note the speedup disparity in this short-circuiting operation compared to *select* and *selectOne*. With 1 million elements, the parallel implementation is relatively more efficient, whereas with 2 million elements the speedup never exceeds 4x, and actually there is a small drop when making use of all hardware threads. In this experiment, parallel EOL is over 9x faster than parallel Java, and sequential EOL is over 6x faster than sequential Java Stream. Perhaps the superior parallel performance can be attributed to the order in which jobs are executed, since they are created and submitted deterministically, and so it follows their scheduling is also deterministic. In a short-circuiting operation, we cannot expect a constant speedup because the termination criteria may be met at different points based on the model. Therefore, the large discrepancy in performance between the two models is not surprising. What is more difficult to explain is why performance declines with 16 threads in the larger model and improves in the smaller one.

Table 6.7.5 *atLeastN* 3.53 million model elements (TR-1950X system)

Implementation	**Execute (ms)**	**Execute STDEV**	**Speedup**	**Memory (MB)**
Sequential EOL	685966	5645		2567
Parallel EOL (1)	703117	15911	0.976	2895
Parallel EOL (2)	407932	19436	1.682	4481
Parallel EOL (4)	231100	16506	2.968	4579
Parallel EOL (8)	157209	9987	4.363	5789
Parallel EOL (16)	118791	3629	5.775	5025
Parallel EOL (32)	117488	1684	5.839	6314
Java (Sequential)	36776010	397612	0.019	3244
Java (Parallel)	5233314	49656	0.131	5312

We also tested *atLeastNMatch* with the largest IMDb model on the Threadripper system (as shown in Table 6.7.5), with largely similar results, and an even more pronounced difference. Due to the scale of the difference between our bespoke *nMatch* and Java Streams, we present the results as a table instead of a graph. We were unable to reach even 6x speedup with all cores, however using 4 threads we did get almost 3x speedup. Efficiency diminished rapidly beyond this point. This shows that even with quad-channel memory and many cores, there is still a limit to how much faster this short-circuiting operation can be when utilising all logical cores. The native Java Streams implementation is not short-circuiting, and so we see a greater speedup when introducing parallelism; in this case a factor of 7x. Still, this is significantly less than the speedup achieved for the *select* operation, so the nature of the computation plays a major role as well, not just the operation's implementation or even the model, since we've used the same model in other benchmarks. Nevertheless, it is interesting that we only managed to achieve a small improvement in speedup compared despite double the number of cores and memory channels compared to the 3700X system. That said, we did use the larger model, and we showed that the 2 million elements model was significantly worse in terms of speedup for this script than the 1 million elements model, so perhaps this trend continues. This is contrary to what we found with EVL, where benchmarks with larger models resulted in greater parallel speedups.

We also benchmarked *nMatch* over the large DBLP model, this time with the *atMostNMatch* variant. The logic here is a deliberately inefficient and somewhat convoluted attempt to find "Records" which have the same "Authors". We can vary the desired execution time almost directly by adjusting the value of *n*: a larger value results in a longer execution time, since to short-circuit (i.e. return false) requires more matches to be found. The program is shown in Listing 6.7.5.

```
return Record.allInstances()
      .atMostNMatch(r1 | r1.authors.exists(a1 |
            a1.records.exists(r2 | r2 <> r1 and
                  r2.authors.size() == r1.authors.size() and
                  r2.authors.containsAll(r1.authors)
            )
      ), Author.allInstances().size() / 17);
```

Listing 6.7.5 *atMostN* operation over DBLP metamodel

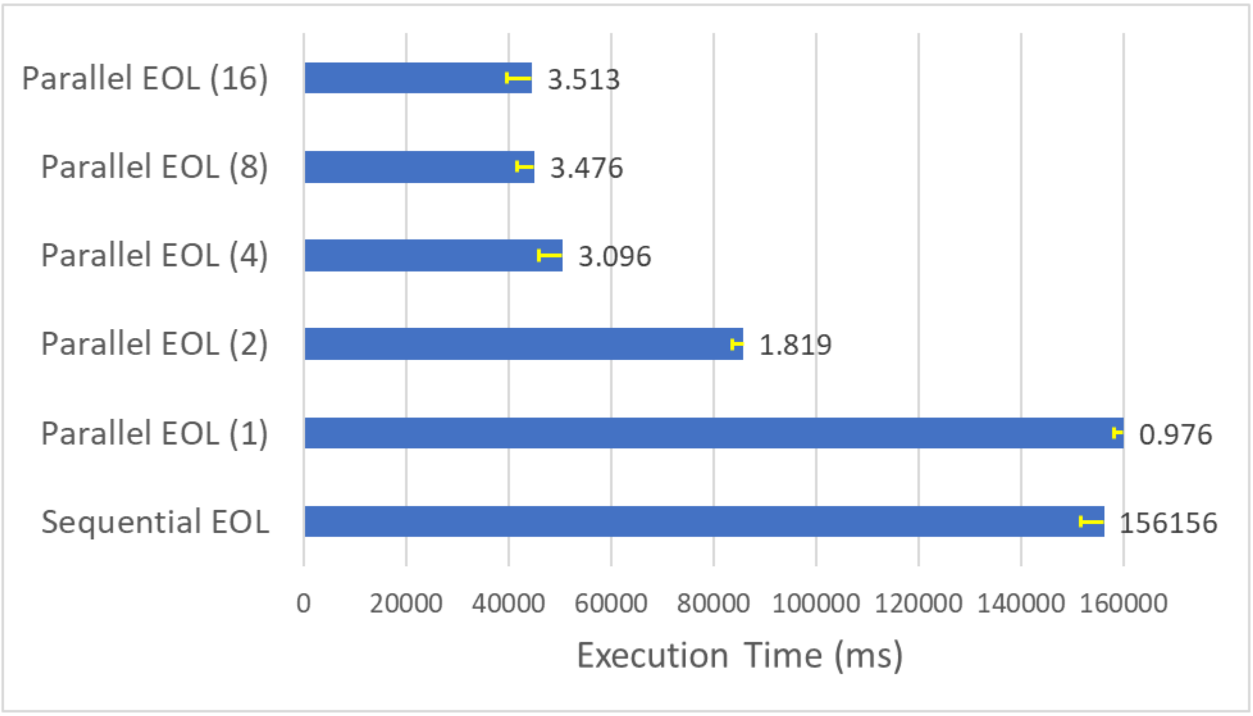


Figure 6.7.8 *atMostN* operation DBLP model (R7-3700X system)

There isn't anything particularly surprising to report from Figure 6.7.8. Just as with the *atLeastN* operation on the IMDb model, we observe relatively poor scalability with more threads, though this time it appears even worse, perhaps due to the larger model resulting in more "pointer-chasing" at the hardware level. There isn't anything in this program which is truly computationally intensive; rather, this is predominantly a memory access benchmark. This is further exaggerated by the use of AtomicInteger in the parallel implementation, which by definition directly accesses the in-memory value, meaning on each match the value cannot be retrieved from cache. Given the relatively large value of *n*, this becomes somewhat significant in an already bandwidth-constrained program, although in the single-threaded case the effect on efficiency is less than 3%. This may explain why we failed to achieve even 4x speedup with all 16 threads, though a threefold speedup with four threads is much closer to our expectations. Note also that the value of improving execution time beyond what is shown here is perhaps academic, given the model loading time is comparable at just under 40 seconds. With a smaller *n* parameter, the execution time would be even less than this.

6.7.2.4 *mapBy* operation

Aside from the first-order operations, we also benchmarked the *mapBy* operation. Recall that the parallel algorithm for this is different to the sequential, in that a collection of Map.Entry objects is obtained by executing the expression in parallel, and then merging these into a Map structure using Java Streams (also leveraging parallelism). To benchmark this operation, we used a simple script on the DBLP model, as shown in Listing 6.7.6. The idea is to map each Record instance by the combined hashcode of its properties. Since there are more Records than Authors, we chose to perform this in the context of Records.

```
return Record.allInstances()
      .mapBy(r | Sequence{
                  r.key, r.url, r.ee, r.mdate, r.month,
                  r.authors.collect(a | a.name)
            }.hashCode()
      )
      .size();
```

Listing 6.7.6 *mapBy* operation over DBLP metamodel

We benchmarked this on the 3700X system, with the results shown in Figure 6.7.9. Despite there being millions of Records and Authors, the script ran surprisingly fast, with sequential EOL taking less than 24 seconds on average. In fact, model loading took longer at 40 seconds! That said, we are "only" hashing Java Strings, which is of course highly optimised. The relative gains from parallelisation appear to be much more modest, if not somewhat disappointing compared to other operations. Clearly the overhead of creating individual entries and merging them later on, even in parallel, is significant. That said, the parallel version is clearly faster even with just two threads. At its peak, we see a speedup of over 3x over the sequential version. When compared to the single-threaded parallel implementation, the speedup is 3.58x with 16 threads and 3.41x with 8 threads.

Given the relatively large jump in speedup from a single thread to four threads, or even from two threads to four, it is surprising how sharply efficiency declines for this operation compared to the others, though in any case the parallel variant is clearly superior. The extent of this superiority does not seem to scale so well with more hardware threads, but nevertheless a 3x boost in performance is still a noticeable gain over the sequential operation.

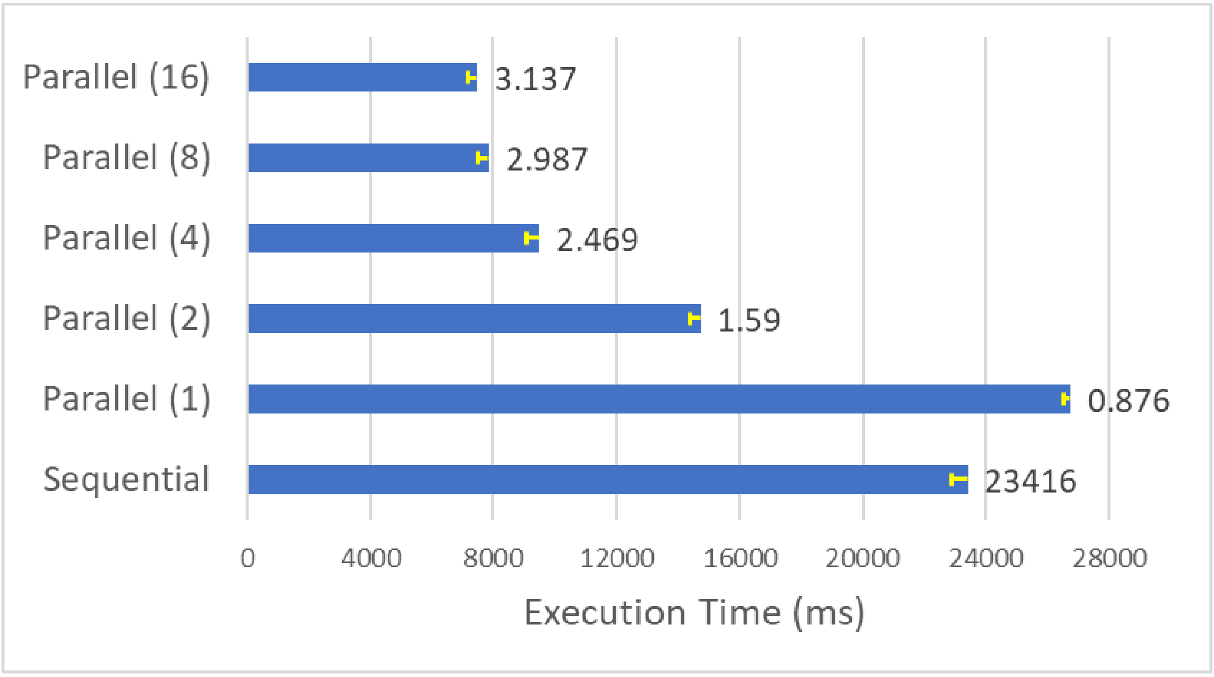


Figure 6.7.9 *mapBy* operation on DBLP model (R7-3700X system)

6.8 **Summary**

In this chapter, we have extended our data-parallel infrastructure for rule-based languages to more generic declarative model query operations. We recognised that all model management tasks depend on model querying, so it is important to ensure that common querying functionality is highly optimised. We evaluated our parallel implementation against not only the *de-facto* Object Constraint Language (OCL) but also against Java, which although is a general-purpose programming language, is also a suitable for model querying in isolation, especially if performance is a key concern. We also investigated the interplay between parallelisation of query operations and the parallel execution approach for EVL introduced in the previous chapter, highlighting some noteworthy challenges and solutions.

As in the previous chapter, we implemented our approach in the context of Epsilon – in this case specifically, the Epsilon Object Language (EOL). The motivation for introducing parallelism to EOL is that EOL underpins all other Epsilon languages, and so if we can generalise some of the data-parallel execution capabilities, given we already have the parallelisation infrastructure and thread-safe engine internals, we can deliver the performance benefits of parallelisation for more complex model management tasks and also for general-purpose model management utilities and querying capabilities. EOL has a mix of imperative and declarative (functional style) language constructs, so we focus on parallelisation of commonly used declarative first-order logic operations. These operations apply to collections of data and take as input a lambda expression, which although typically a very simple one-line expression, can also be arbitrarily complex by invoking an operation containing imperative code. In essence, these operations can be treated similarly to the main execution rule bodies of EVL Constraints, albeit with greater diversity in their purpose and semantics. As with EVL, our goal is to make parallelisation transparent to the user and compatible with the sequential implementations where possible.

We have been able to parallelise all of the declarative operations in EOL except for "*closure*" and "*aggregate*", which are the least commonly used operations, with fully compliant semantics to the sequential operations with regards to results ordering. We have also introduced two new operations: "*count*" which is a more optimal way to express *"select(…).size()*" and *"nMatch*" with three semantics, which is a short-circuiting variant of the *"count*" operation. To our knowledge, the "*nMatch*" operation is novel not only in the context of model management but also in general-purpose querying, and is absent from mainstream programming languages that we are aware of.

We also recognised the overlap between the declarative operations in EOL and the capabilities offered by the Java Streams API, which is more efficient (at least algorithmically) for chained operations due to its lazy execution semantics. We therefore modified the EOL internals to be able to support Java Streams whilst maintaining its semantics, highlighting some of the engineering challenges in doing so.

Introducing parallelisation in a more generalisable, re-usable manner inevitably highlighted some challenges and shortcomings with our original design, which we reformulated and improved further. Most notably, in our rule-based approach parallelisation was applied only once globally, where setting up and tearing down of engine internals was only performed once before and after the main loops, respectively. However the declarative operations can be invoked at any time in any syntactically correct, be nested, chained and used within rule bodies in model management languages such as EVL. We have highlighted some technical challenges, limitations and solutions to this in our approach. We have followed a policy which places minimum burden on the user such that parallelisation, applied at any level, is automatic, transparent, semantically correct (compatible with expectations) and requires minimal assumptions and changes to be performed. We recognise however that ultimately the optimal application of parallelism depends heavily on the program and models, which are exogenous. In the absence of static and/or runtime analysis, we offer a declarative annotation-based approach for fine-grained control of where parallelism is applied in rule-based model management languages, and also allow users to explicitly specify whether the sequential or parallel implementation of an operation should be used. Thus our approach to this fundamentally language engineering problem is a principle of sensible, optimal defaults with no user intervention required to benefit from parallel constructs, whilst also providing the necessarily tooling to allow for optimisation by advanced users so long as they understand the limitations.

We evaluated the main parallel operations on a range of model sizes and with varying number of threads, which showed somewhat stronger performance gains than parallel EVL. We also compared the performance of EOL to Java, leveraging the design of Epsilon's Model Connectivity Layer (EMC) to perform declarative queries using parallel Streams. Surprisingly, the hand-written Java code was much more concise than we expected, owing to the simplicity of EMC. Despite our optimisations, we found that parallel EOL could not beat even sequential Java Streams for performance, though the experiment served as a useful reference point and demonstrates the large overhead of interpreted languages compared to compiled ones. Based on this, we conclude that although parallelisation improves performance significantly, it is not as ground-breaking as more fundamental changes such as moving from interpreted to compiled execution.

7 Parallel Model-to-Text Transformation

This chapter further demonstrates the generalisability of our parallelisation approach by applying the infrastructure developed in the previous chapters (namely, Chapter 4) to another rule-based model management language. This is demonstrated in the context of Epsilon's co-ordination language for model-to-text transformations: EGX. The chapter is organised as follows. Section 7.1 introduces the EGX language, discussing its purpose, features and execution semantics. Section 7.2 shows how the execution algorithm can be parallelised in a similar manner to EVL (as introduced in Chapter 4). Section 7.3 evaluates the parallelised version of EGX against sequential EGX, both for correctness and performance. Section 7.4 concludes the chapter.

7.1 EGX Language

EGX is a rule-based co-ordination language designed for automating the parametrised execution of model-to-text template transformations. Although built on top of the Epsilon Generation Language (EGL), EGX can in principle work with any template-based model-to-text transformation language. The rationale for this co-ordination language comes from the need to invoke text generation templates multiple times with various parameters, usually derived from input models. To better understand EGX, it is helpful to be familiar with template-based text generation, as briefly reviewed in Section 2.1.5.3.

7.1.1 Epsilon Generation Language

The Epsilon Generation Language (EGL) [63] is Epsilon's model-to-text transformation language. EGL in principle is similar in purpose to server-side scripting languages like PHP (and can indeed be used for such purposes, as demonstrated in [294]). To recap, a *template* is a text file which has both *static* and *dynamic* regions. As the name implies, a static region is where text appears as-is in the output, whereas a dynamic region uses code to generate the output, often relying on data which is only available at runtime (hence, "dynamic"). Dynamic regions are expressed using EOL. One can think of an EGL template as a regular text file with some EOL embedded in it, or as an EOL program with the added convenience of verbatim text generation. Indeed, it is possible to use EGL without any static regions, relying on the output buffer variable to write the output text. In EGL, the output variable is called "`out`" and the markers for the start and end of dynamic regions are "`[%`" and "`%]`" respectively. For convenience, "`[%=`" outputs the string value of the expression which follows. EGL has many advanced features, such as recording traceability information, post-process formatting (to ensure consistent style in the final output) and *protected regions*, which allow certain parts of the text to be preserved if modified by hand, rather than being overwritten on each invocation of the template. EGL can handle merges, and also supports outputting text to any output stream.

As an example, consider a simple Library metamodel in Figure 7.1.1. Suppose each model may have multiple Libraries, and each Library has a name, multiple Books and Authors. Similarly, each Book has one or more Authors, and each Author has multiple Books, similar to the relation between

Actors and Movies in the IMDb metamodel used in previous chapters. Now suppose we have a single monolithic model and want to transform this into multiple structured files, such as web pages (HTML) or XML documents. One possible decomposition of this is to generate a page for each Library in the model, as shown in Listing 7.1.1.

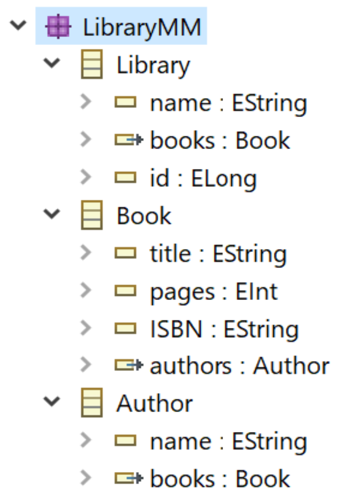


Figure 7.1.1 Library metamodel

```
<?xml version="1.0" encoding="UTF-8"?>
<library id=[%=lib.id%] name="[%=name%]">
[% for (book in books) {%]
  <book>
    <title>[%=book.title%]</title>
    <isbn>[%=book.isbn%]</isbn>
    <pages>[%=book.pages.asString()%]</pages>
    <authors>
    [% for (author in book.authors) {%]
      <author name="[%=author.name%]"/>
    [%}%]
    </authors>
  </book>
[%}%]
</library>
```

Listing 7.1.1 *Lib2XML.egl* example

Notice how the template refers to "*books*" (which is a collection of Book elements) without deriving them directly from the underlying model (i.e. there are no uses of *allInstances*). This is because the variables were provided to the template before invocation.

7.1.2 Template Orchestration

In the previous example, we stated that we want to invoke the template for all instances of Library in the model. To do this, we need to loop through all Library instances in the model(s), load the template, populate it with the required variables derived from the current Library instance and execute the template. However since we want each Library's contents to be written to a distinct XML file (perhaps identified by its name or id), we also need to set the output file for each template based on the current instance. In more complex cases, we may also want to have certain rules for whether a Library should be generated at all (e.g. if it does not have a threshold number of Books), and whether we should overwrite an existing file. For example, we may decide that for Libraries with a large number of books, we do not want to overwrite the file. Furthermore, we may want to have a different naming convention for certain libraries based on their name or ID, which may be decided based on an arbitrarily complex function. Also, we may not want to include all of the Books in the output file, but a subset, which requires additional processing logic. We may even have different templates for libraries based on the number of Books they hold – for example, with a large Library, we may want to inline all of the properties of each Book to save disk space, rather than having the title, pages, authors etc. enumerated as children. Or we may want to omit the authors. This can be achieved by modifying the template with conditionals, but this makes the template much less readable and harder to modify, so it can be easier to have a separate template instead.

All of these factors are tedious to implement manually and can be difficult to maintain and modify by domain experts using handwritten imperative code. Therefore, a more declarative way of achieving this is needed. This is precisely the purpose of EGX.

7.1.3 Features and Execution Algorithm

Like all of Epsilon's rule-based (ERL) languages, an EGX module consists of any number of named rules, as well as optional *pre* and *post* blocks which can be used to perform arbitrarily complex tasks using imperative code before and after the execution of rules, respectively. Since the purpose of EGX is to simplify and automate the process of invoking EGL templates, one could argue that EGX is perhaps the "purest" form of ERL: there are no dependencies between rules, and no intermediate or even results data structures to manage. The execution algorithm of EGX is quite simple, since the language itself is essentially a means to parameterise a *for* loop. EGX adds on top of ERL only a single top-level rule construct: the *GenerationRule*. The execution algorithm is thus as simple as executing all of these rules, in the order they are defined in the module. Thus, the remainder of this subsection describes the components and execution semantics of GenerationRule. Note that since variables declared in an earlier scope (executable block) within a GenerationRule are visible to later blocks, the order in which the engine executes each component block is important. Thus, we summarise each component block in execution order; which should also be the order in which they are declared by the user in the program. Note also that all of the component blocks of a GenerationRule are optional – that is, one can use any combination of them, including all or none.

- **`transform:`** A parameter (name and type), optionally followed by the collection of elements to run the rule over. The parameter name is bound to the current element, and this rule is executed for all elements in the specified collection. If the user does not specify a domain from which the elements are drawn using the **`in:`** construct, the engine will retrieve all model elements matching the type (but not subtypes) of the parameter type. To include all types and subtypes of the specified parameter, rule must be marked with the `@greedy` annotation, otherwise the entire rule must be repeated for each subtype.
- **`guard:`** True by default. If this returns false, the GenerationRule will skip execution of the remaining blocks for the current element (or altogether if the rule has no input elements).
- **`pre:`** Arbitrary block of code, can be used to set up variables or any other pre-processing.
- **`overwrite:`** Whether to overwrite the target file if it already exists. True by default.
- **`merge:`** Whether to merge new contents with existing contents. True by default.
- **`template:`** The path (usually relative) and name of the template to invoke.
- **`parameters:`** Key-value pairs mapping variable names to values, which will be passed to the template. That is, the template will be populated with variable names (the keys) and values based on the provided Map.
- **`target:`** The path of the file to which the output of the template should be written.

- **`post:`** Arbitrarily code block for post-processing. In addition to having access to all variable declared in previous blocks, a new variable called "generated" is also available, which is usually a reference to the generated file (so the user can call any methods available on java.io.File). If the EGL execution engine has not been configured to output to files, then this variable will be the output of the template as a String instead.

The only other noteworthy aspect of EGX's execution algorithm is that it keeps a cache of templates which have been loaded, to avoid re-parsing and re-initialising them every time. Of course, the variables for the template are reset and rebound every time, as they may be different. The purpose of the cache is only to avoid the potentially expensive process of parsing EGL templates.

7.1.4 **Example Program**

Returning to our example, we can orchestrate the generation of Libraries as shown in Listing 7.1.2, which demonstrates most of the features of EGX. This example is adapted from the article in [295]. Here we see how it is possible to screen eligible Library instances for generation, populate the template with the necessary parameters, invoke a different version of the template and direct the output to the desired file, all based on arbitrary user-defined criteria expressed declaratively using EOL. We can also compute aggregate metadata thanks to the *pre* and *post* blocks available both globally and on a per-rule basis. In this example, we simply compute the size of each file and print them once all transformations have taken place. Furthermore, we demonstrate that not all rules need to transform a specific model element: EGX can be used for convenience to invoke EGL templates with parameters, as shown by the "AuthorsAndBooks" rule. Here we only want to generate a single file from the Authors and Books in the model, where the logic for doing this is in a single EGL template. Although it wouldn't make much sense to use EGX purely for invoking single templates without parameters, the reader can perhaps appreciate that in large and complex models, there may be many different templates – e.g. one for each type – so all of the co-ordination in invoking them can be centralised to a single declarative file. EGX can thus be used as a workflow language in directing model-to-text transformations and is suitable for a wide range of use cases.

```
operation Book isValid() : Boolean {
  return self.isbn.isDefined() and self.isbn.length() == 13;
}
pre {
  var outDirLib : String = "../libraries/";
  var libFileSizes = new Map;
}

rule Lib2XML transform lib : Library {
  guard : lib.name.length() > 3 and lib.books.size() > 10
  pre {
    var eligibleBooks = lib.books.select(b | b.isValid());
    var isBigLibrary = eligibleBooks.size() > 9000;
  }
  merge : isBigLibrary
  overwrite : not isBigLibrary
  template {
    var libTemplate = "rel/path/to/Lib2XML";
    if (isBigLibrary) {
      libTemplate += "_minified";
    }
    return libTemplate+".egl";
  }
  parameters : Map {
    "name" = lib.name,
    "id" = lib.id,
    "books" = lib.books
  }
  target {
    var outFile = outDirLib + lib.name;
    if (isBigLibrary) {
      outFile += "_compact";
    }
    return outFile+".xml";
  }
  post {
    libFileSizes.put(generated.getName(), generated.length());
  }
}

rule AuthorsAndBooks {
  parameters : Map {
    "authors" = Authors.allInstances(),
    "books" = Book.allInstances()
  }
  template : "AuthorsAndBooks.egl"
  target : "AllAuthorsBooks.txt"
}

post {
  libFileSizes.println();
    ("Total: "+libFileSizes.values().sum()).println();
}
```

Listing 7.1.2 EGX example program over Library metamodel

7.2 Parallelisation

As previously stated, the execution algorithm of EGX is very simple, owing to its rule-based structure. Since we have already developed a parallel execution infrastructure and “solved” the concurrency issues with EOL, having demonstrated this both for EVL and declarative iterator operations in the previous chapters, as one would expect EGX is almost trivial – at least in principle. However we encountered numerous engineering difficulties due to the initial design of EGL, and EGX’s tight coupling with EGL-specific internals. Whilst the internal design of EGL is quite convoluted, the main issue essentially boils down to shared state between template invocations, which we were able to refactor without affecting their observable behaviour. However, at the time of writing, there are a few niche features of EGL which our parallel implementation does not support, such as fine-grained traceability (which may be used for incremental execution).

There are three strategies / algorithms we can use for parallel EGX, all of which are based around executing *GenerationRule-element* pairs independently. This is analogous to the *ConstraintContext-element* pairs decomposition we used for parallel EVL. A GenerationRule does not always transform elements from a collection – it may be run just once. Algorithmically, this is similar to the notion of *GlobalConstraintContext* in EVL, where the rule has no input elements and is executed once. The three strategies are as follows:

- **Elements-based**: See section 4.4.2.
- **Atom-based**: See section 4.4.4.
- **Annotation-based**: See section 4.4.6.

In all of the above cases, it should be sufficiently trivial for the reader to deduce the algorithm of each, simply by substituting *ConstraintContext* with *GenerationRule* in the associated listings of each section. From a parallel execution perspective, there is no fundamental difference between the two concepts: both constructs are invoked by the engine repeatedly for each applicable element, and their subconstructs (e.g. the *guard* block) are executed as part of that job for the specific element. In other words, the implementation of *GenerationRule.execute(Object element)* and *ConstraintContext.execute(Object element)* are “black boxes”: we need not concern ourselves with their constructs, since our parallel algorithm operates on the basis of rule-element pairs.

The main notable difference is that GenerationRule offers users more flexibility in the collection from which model elements are drawn. Where ConstraintContext retrieves all types and subtypes of the specified type (i.e. *getAllOfKind*), GenerationRule by default only retrieves the direct types (i.e. *getAllOfType*) by default, and offers the option to specify an expression returning the collection of elements to execute the rule with. It should be noted that this collection need not necessarily be model elements – it can be of any object type. There is also no concept of *self* in GenerationRule, since the parameter name is specified by the user. It is also more common to see rules without model element parameters in EGX than in EVL, and the cost of executing each rule is potentially higher since each rule invokes an EGL template. For this reason, it does not make much sense to use the Elements-based implementation, as it executes each rule one at a time, rather than submitting all rule-element pairs to the executor service at once. Thus, our default implementation is the atom-based one, which as a reminder pre-computes all rule-element tuples and executes them in parallel.

7.3 Evaluation

We evaluate parallel EGX in a similar manner to parallel EVL, reusing most of the infrastructure and methodology for both correctness and performance. However, EGX is slightly more complicated to evaluate because the output is (usually) persisted to files, which requires more engineering effort.

7.3.1 Correctness

We followed the same principle for testing as with parallel EVL: using the sequential implementation as the oracle and ensuring the parallel implementation produces identical results. We reused the testing infrastructure, since both EVL and EGX extend ERL. Therefore equivalence of internal data structures (e.g. the FrameStack) was assured, and we tested with varying number of threads as usual. Comparing the results is straightforward. Since the execution of EGX may result in multiple files in various directories, we build a map of output files to their contents in bytes. We can then ensure that the sequential and parallel implementations result in an identical byte array (same order, same size, same contents) for each generated file. After each test, we delete the output files to ensure they are regenerated every time.

We used two test programs, one which is relatively simple and another which is extremely complex, using almost every feature of EGL and EOL, with thousands of lines of code. The former is from Picto [296]: an Epsilon project which was originally for visualising Ecore metamodels in Graphviz, though it now supports a wider range of formats. The main idea behind Picto is to generate text which is then parsed by visualisation programs (such as PlantUML or Graphviz) to provide a graphical view of models. The example we used visualises Ecore models (i.e. EMF metamodels) in DOT format. Specifically, each EClass in the metamodel is transformed into a DOT file (see [297]).

The second comes from the author's own MSc project, which aimed to reimplement Apache Thrift using MDE technologies [8]. Thrift can generate RPC code for various languages, so for this project, two target languages were chosen: Java and Ruby. The code generation targeting Java is by far the most verbose and complex, whilst Ruby is the shortest and simplest. The generator takes as input models of Thrift definition files (which describe the communication interfaces, services and data structures to be generated) and produces them, along with the serialization / marshalling and deserialization code in the target language. Combined, there is over 3500 lines of EGL, EGX and EOL code for this project, making extensive use of cached operations, extended properties, imperative and declarative constructs. The generated output files, particularly for the Java generator, are quite large even for trivially sized Thrift services and structs. The complexity of the generator and the number of types involved makes this a good test for the engine's correctness. For the Java generator, there are 5 EGX rules, 4 of which transform elements (i.e. they invoke an EGL template for every model element of a given type) and 1 which invokes a template only once.

As expected, all 360 unit tests passed without errors. This test suite is very computationally expensive due to its thoroughness. For perspective, the entirety of the Epsilon test suite, which has nearly 2900 tests, takes 22 seconds on our Threadripper system, whereas this test suite for EGX takes 184 seconds; almost entirely due to the high computational cost of our Thrift generator. That said, we have seen these tests fail during development on occasions (i.e. failing perhaps 10% of the time, due to the non-deterministic nature of concurrent programs), helping to highlight obscure bugs in the implementation. These bugs were related to a leakage of state between EGL template invocations, not our parallel infrastructure or approach. As with our other tests, we have ran this suite hundreds of times without failures on various hardware platforms and operating systems, once we “finished” development (i.e. the version we used for evaluation was complete).

7.3.2 **Performance**

We used a different set of inputs for assessing the performance of EGX than our test suite. Specifically, we are interested in assessing the scalability benefits of parallel EGX when there are many template invocations, leading to many generated files, since we are assessing the performance of EGX rather than EGL. Furthermore, it is interesting to see how performance of EGX compares to EVL, given that this time we are potentially I/O bound. Conventional wisdom states that when bottlenecked by I/O, more threads should improve performance. Thus, we test not only with the maximum number of hardware threads, but up to quadruple that to see the extent to which we are I/O-bound or whether having too many threads negatively impacts throughput. Moreover, the choice of storage device becomes significant, since a mechanical hard drive will be slower than an NVMe SSD. To test this, we have three variants for the output of each experiment: one which targets an SSD, one which targets a hard drive and another where the output is not written to a file at all, but rather returned as a raw string. This will allow us to assess the extent to which sequential and parallel EGX are I/O bound and under which scenarios parallelisation provides the greatest benefits.

The program we are benchmarking uses the IMDb models. It is a very simple solution for mapping the entire model to files. Specifically, we generate a CSV file for every Actor in the model. This file contains the list of Movies the Actor has participated in – one per row – along with all available information about the movie in each column. The EGX program is shown in Listing 7.3.1, and the corresponding EGL template in Listing 7.3.2.

```
pre {
    var lengths = new ConcurrentMap;
}

@greedy
rule People2MovieFiles transform person : Person {
    parameters : Map{"person" = person}
    template : "personMovies.egl"
    target : "actors/"+person.name.replaceIllegalChars()+".csv"
    post {
        lengths.put(person.name, generated.length());
    }
}

post {
    var keys = lengths.keySet().size();
    keys.println("Number of files: ");
    var values = lengths.values().sum();
    values.println("Sum of characters in all files: ");
    var average = (values / keys).asInteger();
    average.println("Average characters per file: ");
}
```

Listing 7.3.1 *imdb2files* EGX performance evaluation script

```
YEAR, TITLE, RATING, PEOPLE, TYPE
[% for (movie in person.movies.sortBy(m | m.year)) {%]
[%=movie.year%], [%=movie.title%], [%=movie.rating%],
[%=movie.persons.size()%], [%=movie.type.name%]
[%}%]
```

Listing 7.3.2 *personMovies* EGL template

Figure 7.3.1 shows the results for the script when ran on 1.5 million model elements on our Threadripper system, both with persistence enabled and disabled. We used the *GenerationRuleAtom* implementation for parallelism. Note that speedup is relative to the sequential implementation in each case. The number of template invocations (and thus, generated files when persistence is enabled) with this many elements is a gargantuan 1,176,147, and a total cumulative size of 4.6 GB (based on the *du* command). We wrote these files to the NVMe SSD, near the root of the filesystem and also to the HDD in the user's home directory. Given the sheer number of files and the total number of bytes written, it is unsurprising that for all levels of parallelism, the program was faster with persistence disabled. However, this difference narrowed considerably when all hardware threads were utilised. In the sequential case, not having to write to disk reduced execution time by a minute and a half (25%), whereas with 32 threads, the difference in execution time was just 3 seconds (8%). It's also interesting how efficiency varies between the two modes. With two threads, the parallel implementation was twice as fast as the sequential when writing to SSD, whereas with persistence disabled this dropped to less than 1.7x. This can be explained the fact that the program is bottlenecked by writing to disk, hence the additional thread(s) provide performance benefits not just in terms of compute, but I/O. The delta between speedup achieved with persistence enabled vs. disabled gives us an indication of the extent to which we are I/O bound. With fewer threads, the application is more bottlenecked by writing to disk, whereas when we increase this, we are bottlenecked by compute performance. In all cases, peak performance is achieved with 32 threads, where we observe a peek speedup of 9.5x on the SSD.

It's interesting to note that the performance difference between writing to the hard drive and to solid-state storage appears to be non-existent, at least when using 4 or more threads. Clearly the performance difference becomes negligible, perhaps because we are writing lots of small files rather than a few large ones. What's most surprising is that in the sequential case, the hard drive was faster than the SSD. The reasons for this are beyond the scope of this thesis, however for reassurance, all of the experiments in Figure 7.3.1 were repeated (i.e. the original data was thrown out, and completely redone and triple-checked to ensure the data was correctly reported on).

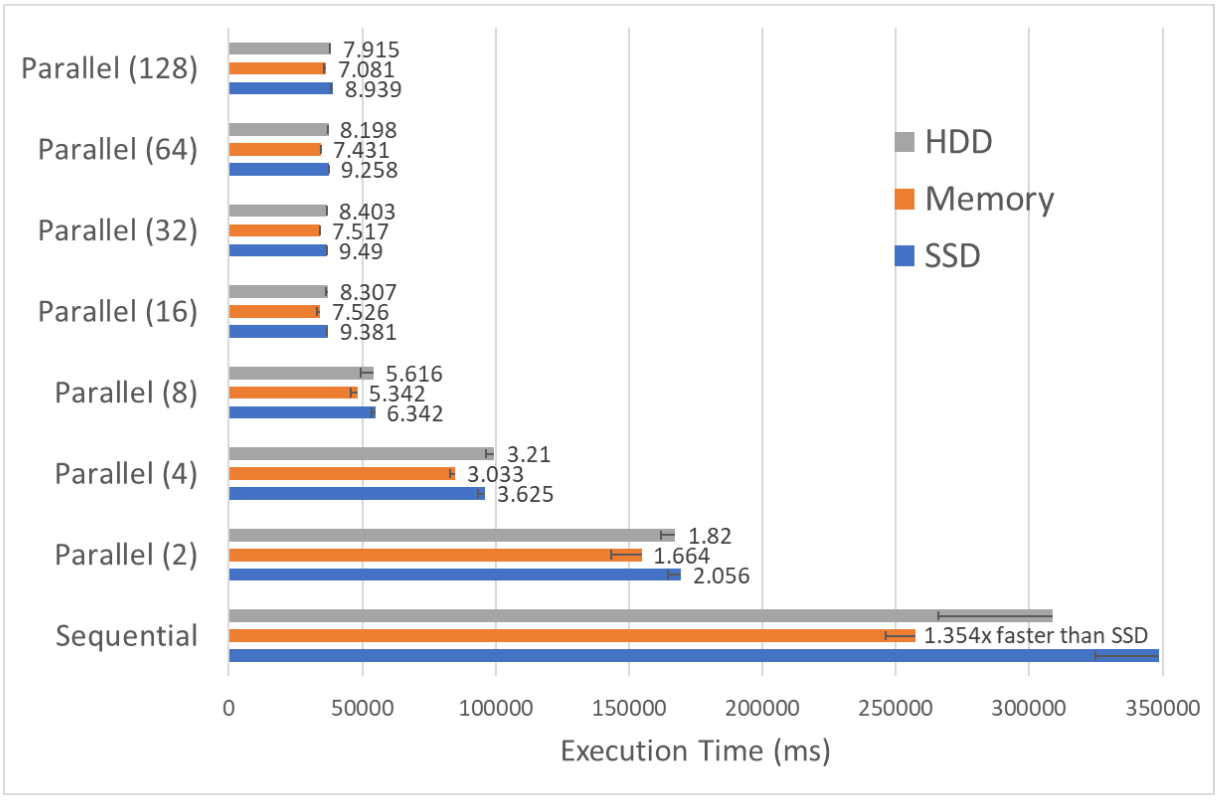


Figure 7.3.1 *imd2files* 1.5 million elements (TR-1950X system)

We experimented with a larger model, doubling the number of elements to assess the relative scalability with more files. This resulted in 2,196,849 files when writing to disk, weighing in at 8.5 GB when combined. The results in Figure 7.3.2 are more aligned with our expectations. With only a single thread, the sequential implementation is clearly bottlenecked by the storage, although this is by a relatively small margin when considering the speed difference between solid-state and spinning platter storage. At best, not having to write to disk at all is 1.5x faster. However, this difference quickly diminishes with multiple threads. At this point, the bottleneck is not I/O but executing the templates. Hence, performance peaks when utilising all 16 cores. It's also interesting that speedup is greatest when writing to the hard drive – when using 2 and 4 threads, efficiency is above 1, and practically equal to 1 with 8 threads. This is perhaps due to the I/O bottleneck of writing to the hard drive, where the extra threads not only improve compute performance but also make writing to disk faster. This is true, albeit to a lesser extent, for the solid-state drive. The actual compute speedup when not writing to disk is just under 8.2x. Surprisingly, SMT provides no performance benefit in this case, with performance virtually the same, if not ever so slightly worse when using all 32 hardware threads. We can also see that performance diminishes more when going beyond 16 threads when we are not I/O bound; hence the speedup for HDD remaining constant, if not slightly improving, whilst in pure compute terms performance degrades. Notice how relative to the 1.5 million elements model, multi-threaded speedup is improved considerably for all thread counts, peaking at over 11.3x with 32 threads when writing to the hard drive.

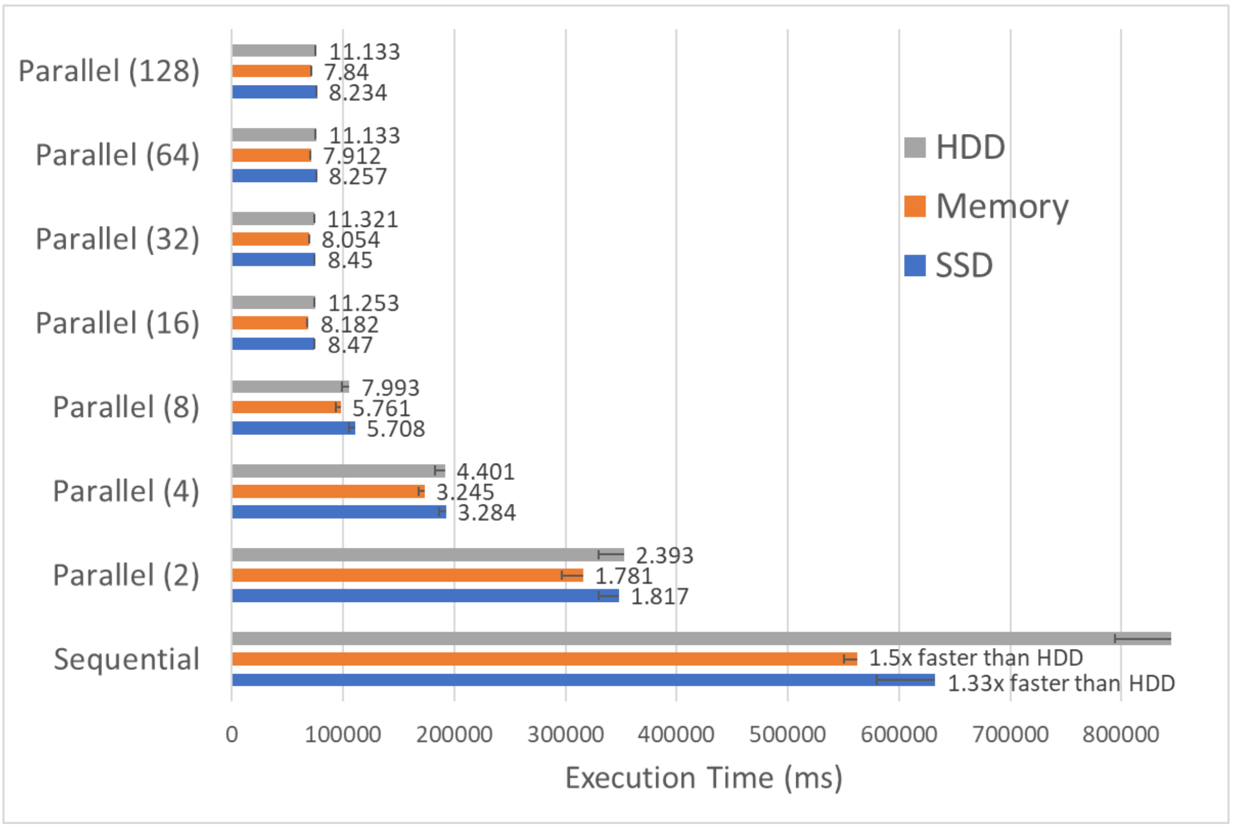


Figure 7.3.2 *imdb2files* 3 million elements (TR-1950X system)

For completeness, we also benchmarked with a smaller model – this time only half a million elements. The results – shown in Figure 7.3.3 – are structurally similar to those in Figure 7.3.1. The SSD is slower than the hard drive, and the I/O bottleneck is even less apparent, as expected.

Parallelism provides the greatest benefit when writing to solid-state storage, with up to 8.4x speedup when using all logical cores. Any benefits of multithreading for I/O become redundant when going beyond 32 threads, since at this point compute performance also degrades as evident by the no output scenario. Still, at best we were able to achieve over 7.6x speedup with 16 threads; again curiously SMT somehow degrading compute performance. Even with only 500k model elements, the number of generated files is large enough to show a significant disparity between the relative compute and I/O performance with multithreading.

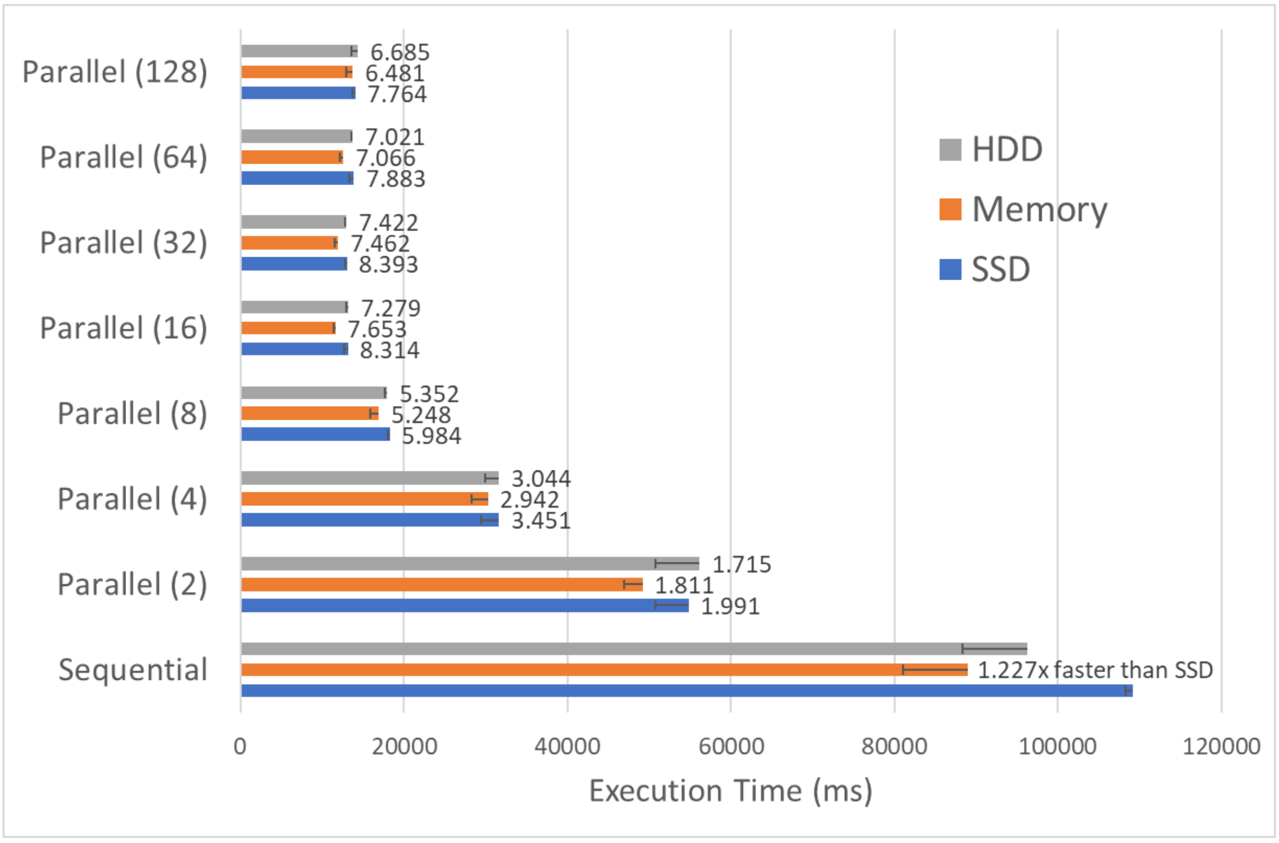


Figure 7.3.3 *imdb2files* 0.5 million elements (TR-1950X system)

Finally, we benchmarked the program using even smaller models on our laptop system. Figures Figure 7.3.4 and Figure 7.3.5 show the results with 200k and 100k model elements, respectively. Without repeating the caveats of interpreting speedup on laptops, where the sequential clock speed is over twice as fast as under all-core, the results are in line with our expectations. As usual, additional threads benefit the I/O-bound case more than the pure compute one, albeit by a smaller margin. Performance peaks when utilising all 8 hardware threads, though unlike the Threadripper system, Hyperthreading improves compute performance rather than degrading it (perhaps due to there being fewer cores in this system). For simplicity, let us assume that the "true" speedup, when compensating for the difference in clock speeds, is 2x greater than what the graphs show when using all 8 hardware threads. In that case, we would be looking at over 4.1x compute speedup with 200k elements, and over 5x when writing to disk. This is slightly worse at around 3.9x compute and

4.7x I/O respectively for the 100k model. In both cases, performance degrades when using more threads than can be executed simultaneously at the hardware level. Not only does it degrade compute performance, but it also provides no benefit in terms of I/O throughput, given we are writing very small files.

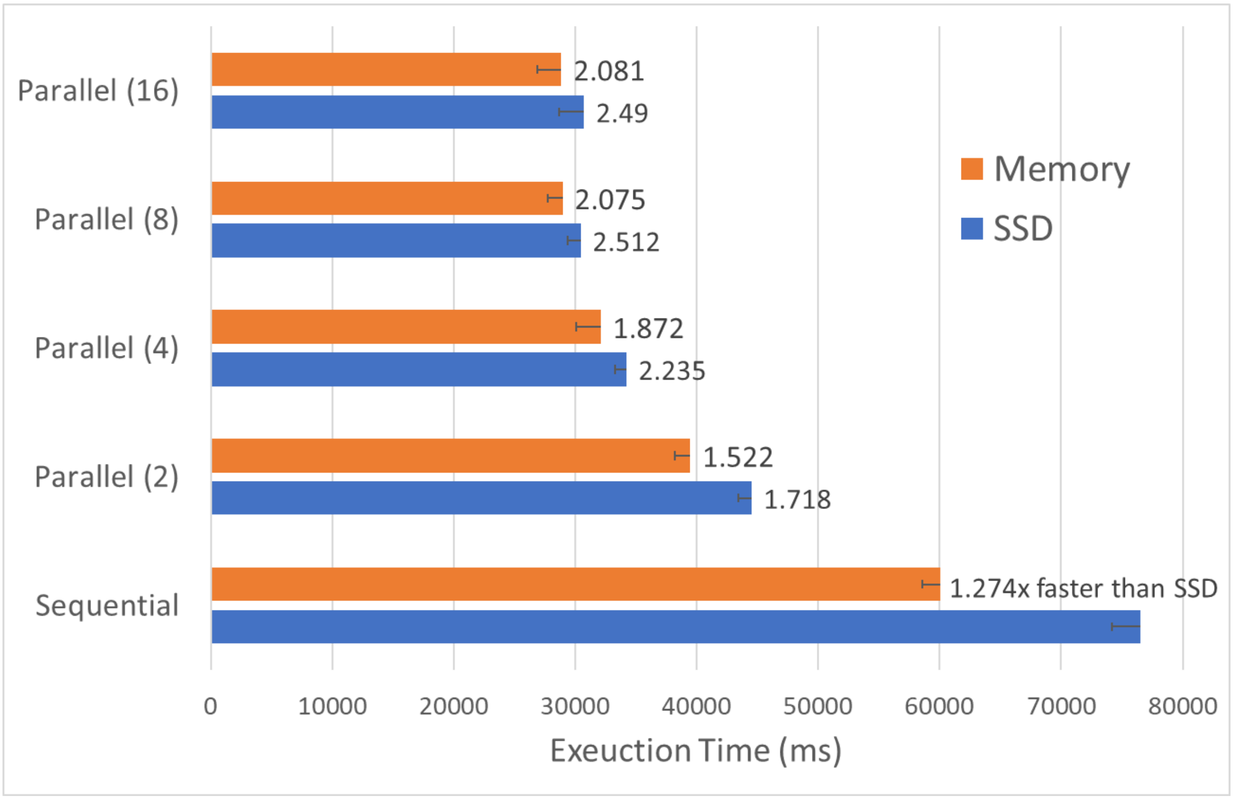


Figure 7.3.4 *imdb2files* 0.2 million elements (i7-8550U system)

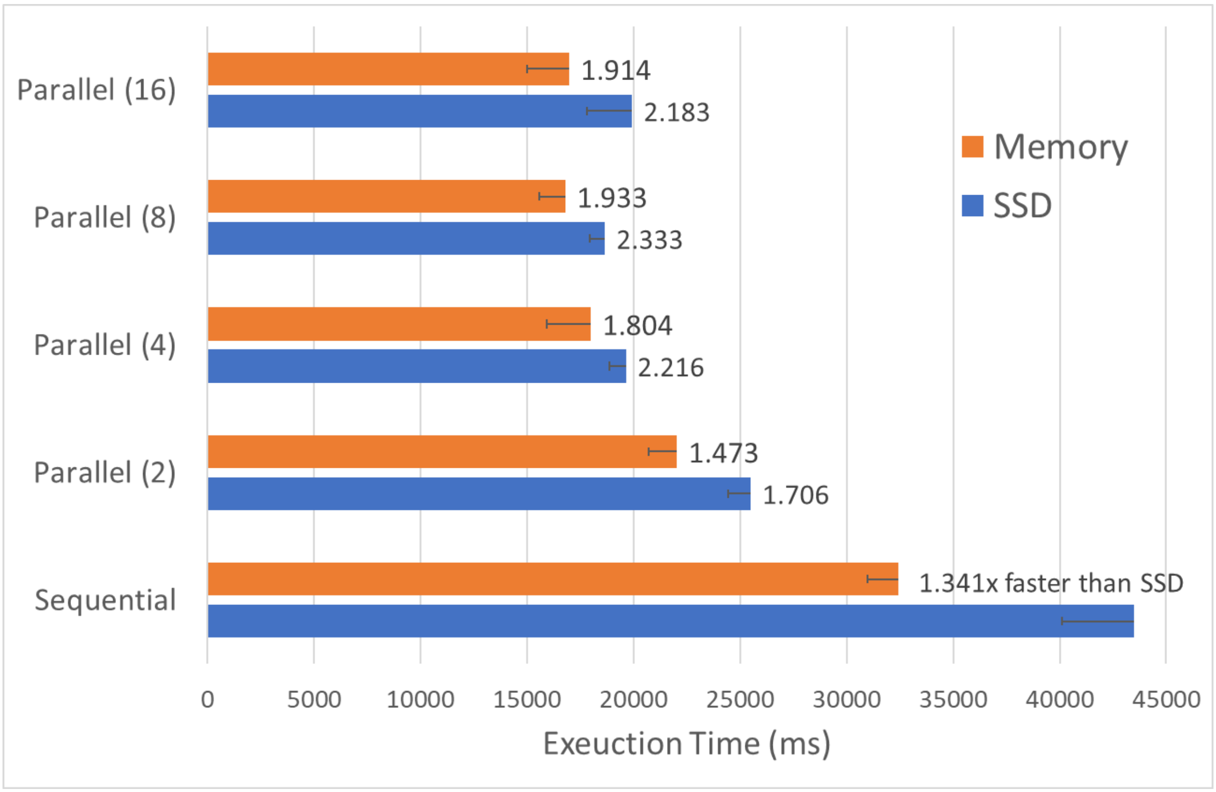


Figure 7.3.5 *imdb2files* 0.1 million elements (i7-8550U system)

7.4 Summary

In this chapter, we showed how our parallelisation approach can be applied to the task of model-to-text transformation. We motivated the need for a rule-based co-ordination language when working with parameterizable template-based model-to-text transformation languages, and how EGX provides many of the capabilities required by complex modelling projects to transform their models into textual artefacts. By virtue of being rule-based, we explained that applying the same approach to parallelisation and evaluation as outlined in Chapter 4 is almost trivial, requiring only minor additional code for dealing with the task-specific aspects / data structures used in the language. We equivalence-tested our parallel implementation with the sequential one with a few highly complex open-source EGX projects which exercised almost all the features of EOL and EGL.

In terms of performance, we have seen that multithreading provides performance benefits not only for compute performance, but also in I/O-bound scenarios. For this reason, one may argue that parallelisation of model-to-text transformations is of great significance when compared to other model management tasks due to the greater efficiency. That said, our experiments showed that even in extreme scenarios where millions of output files are created, the bottleneck imposed by writing to a slow disk is mostly alleviated even with just 4 or 8 threads, and compute performance becomes dominant. In any case, the experiments have shown that our data-parallel approach and infrastructure scale equally as well in terms of raw compute performance for model-to-text transformation as they did for model validation.

8 Conclusions

In this final chapter, we reflect on the overall aims, objectives and achievements of the thesis, as well as noting content which is absent from the thesis; thus motivating the need for further research. Section 8.1 provides a brief summary of this document. Section 8.2 consolidates the contributions of the thesis with the key takeaway messages. Section 8.3 describes preliminary / ongoing work which was not completed in full as part of this thesis, and also suggests directions for further complimentary research. Section 8.4 closes the thesis with the author's final thoughts on the relevance and tone of this research.

8.1 Summary

The goal of this thesis has been to improve the performance of model management programs by making better use of modern hardware in the hope of making a small but much-needed contribution to the multi-faceted concern of scalability in model-driven engineering. In Chapter 1, we briefly introduced the rationale for this thesis.

A detailed overview of the background to this research was provided in Chapter 2. Firstly, we introduced the relevant and fundamental concepts in model-driven engineering. We also performed a metanalysis of the literature regarding the adoption of MDE in industry, including the motivations and barriers to adoption. We then motivated the case for improving performance of model management programs. Essentially the argument is that model-driven engineering is often used in large, complex industrial projects, where models tend to be very large. Thus, the projects which would be ideal candidates for benefiting the most from a model-driven approach also bear the greatest burden with regards to scalability challenges, which includes poor performance of model management tools. Having established that inefficient tooling is an important problem in the field of MDE, we provided some background on methods used in general-purpose programming to improve application performance. Namely, these include laziness, incrementality and parallelism; which comes in two forms: shared-memory and distributed (multi-process). We described each of these techniques in sufficient detail, accounting the benefits of each, and justified in particular the motivation for choosing parallelism as the primary focus of this thesis. With the pre-requisites hopefully now clear, we thoroughly examined the existing literature on improving performance of model management programs. Our goal in this section was to provide a broad but detailed view of works, both classic and recent, of research into scalability of model management so that the reader can gain an appreciation of the breadth of concerns involved. It is also helpful to be able to compare this research to the works cited for gauging where this thesis' contributions lie in the grand scheme of things. The main findings of the literature review were that there is a lack of research into parallel and distributed execution of model management beyond model-to-model transformations. Specifically, there are no works (at the time of writing) that we were aware of which attempt to provide a generalisable, highly optimised parallelisation and distribution approach for rule-based and declarative model management tasks such as querying, validation and model-to-text transformation. That is the gap which this thesis aims to fulfil, at least partially.

In Chapter 3, we provided details of how the thesis would realise its intended contributions in practice. We motivated the case for targeting Epsilon, noting its extensible nature, support for various model management tasks which build upon a unified, highly complex and feature-rich core language, and the fact that the languages are independent from any specific modelling technology. We then explained the evaluation methodology in some detail.

Chapter 4 presents the main contribution of this thesis using the Epsilon Validation Language (EVL) as the implementation target. The chapter begins by introducing the language concepts and structure, then diving into the technical details of its implementation. Since EVL is built on top of EOL, most of the challenges regarding thread-safety are at the EOL level. These include data structures such as the frame stack (used for storing variables in various scopes), the execution trace (for reporting exceptions and other traceability information) and the mechanism by which operation calls are handled. On the language level, serial thread confinement is used for these structures, and other shared data structures, such as caches, use standard thread-safe collections. On the modelling technology side, the main thread-safety issues stem from caches, which are made thread-safe. Since the thesis focuses only on read-only model management tasks, no further concurrency issues were identified in this regard. The main concurrency challenges with EVL stem from dependencies between rules, and a novel solution is presented for handling these. Aside from that, the data structures used for storing results are made thread-safe and optimised. With thread safety issues dealt with, we then present both data-parallel and data-and-rule-parallel decompositions of EVL programs. We opt for a data-parallel strategy, for various reasons. We present our parallelisation algorithm, where essentially every rule-element pair is executed independently. The scheduling of jobs to hardware threads is handled by a fixed thread pool executor service. With a parallel decomposition at hand, we show how the deterministic ordering of job creation, which depends on the EVL script and modelling API, can be exploited to distribute the workload to multiple independent processes efficiently. Our distribution approach relies on each node having a full copy of the resources, and with the ability to recreate the jobs list, a subset of jobs can be dynamically assigned to each participating process using only indices to reference jobs in the list. We discussed various parameters and optimisations to maximise the efficiency of the approach in an implementation-agnostic manner. Finally, we describe how we evaluated both the correctness and performance in detail. For correctness, an extremely thorough test suite is used, with various combinations of complex EVL programs exercising the full capabilities and features of the language, on different (meta)models, with varying number of threads and other internal engine parameters. We test the internal data structures and results for equivalence with the sequential implementation, and also ensure equivalence with identical programs written in the Object Constraint Language – a *de-facto* standard. In terms of performance, our parallel approach scales almost linearly with more threads, however is severely bottlenecked by memory bandwidth after a certain point. We observed up to linear speedups with 16 cores. Our distributed approach is much more efficient, as it can leverage multiple processors on multiple computers, leading to linear scalability. At its best, with a 16-core master and 87 6-core workers, we observed a 340x speedup over the sequential implementation; reducing execution time from over 6 hours and 20 minutes to just 70 seconds. The speedup scales linearly with model size and number of workers.

In Chapter 6, we generalised the parallel execution capabilities developed for parallel EVL to declarative query operations. These are predominantly first-order logic function which come in two forms: short-circuiting and non-short-circuiting. We showed how the same data-parallel approach and existing infrastructure can be adapted to accommodate a much more diverse range of queries and transformations, all whilst maintaining equivalence in terms of ordering with their sequential counterparts. We also introduced a novel short-circuiting operation for determining whether a specified number of elements in a collection satisfy a predicate. In addition, we recognised the lazy semantics offered by the Java Streams API and incorporated this into EOL, exploring the engineering challenges associated with this. We established equivalence between all parallel and sequential operations as well as their Java Streams counterparts with a comprehensive test suite. Finally, we conducted a thorough performance evaluation of both short-circuiting and non-short-circuiting operations on various hardware configurations, assessing the scalability with model size and number of threads. We also compared the performance to OCL and native Java Streams, noting the inferiority of the former and superiority of the latter. Although native (compiled) hand-coded Java Streams code is in another league in terms of performance relative to EOL (even when comparing parallel EOL to sequential Java), one notable exception is the novel *nMatch* operation, which has no equivalent in the Streams API. In our benchmarks, the short-circuiting capabilities of this operation in EOL outperformed hand-coded Java Streams by the same magnitude that native Java code outperformed EOL in other operations. Our benchmarks also revealed the extent to which performance benefits are hardware dependent. Specifically, we found that the number of memory channels and CPU architecture are key factors. For example, dual-socket quad-core CPUs from 2009 with HyperThreading (8 cores / 16 threads in total) provided almost 10x speedup whereas a single-socket 8 core / 16 thread CPU from 2019 struggled to get beyond 5x speedup in most cases with dual-channel memory, despite the memory being 3x faster in clock speeds.

Chapter 7 demonstrated how our parallelisation approach can be applied to another rule-based model management task: the orchestration of model-to-text transformation templates. We saw how our atomic rule-element decomposition can be applied without any changes to the approach or parallelisation infrastructure. We followed our usual method of evaluation for both correctness and performance, with equivalence testing using a complex suite and assessing the performance via a carefully chosen program and models. The main findings from our benchmarks were mostly inline with our expectations: even when writing to a fast disk, parallelisation has greater benefit than when writing to memory. We were able to reduce execution time by up to 10x compared to the sequential case with a 16-core system.

8.2 Lessons Learnt

This thesis has focused on data parallelisation of read-only model management languages, particularly those of Epsilon. In Chapters 4 and 5, we showed how a data-parallel decomposition can be implemented for a complex model validation language to take advantage of both parallelism both within and between computers; that is, both shared-memory and distributed memory parallelisation. In Chapter 6 we generalised the parallelisation infrastructure, showing how generic model query operations on collections can be parallelised in an optimal and compatible manner, as well as the interplay between parallelisation at the language level and parallelisation of first-order operations. Finally in Chapter 7 we showed how our parallelisation strategy for EVL (as discussed in Chapter 4) could be generalised to the co-ordination of model-to-text transformations.

Whilst these chapters focused exclusively on Epsilon's languages, there is no reason why they ought to be limited to the scope of Epsilon or similar languages. The thesis relies on certain fundamental properties provided by the nature of certain model management tasks which Epsilon's design highlights conveniently. In this section, we explain these underlying fundamentals explicitly. This section also serves as an extension of the "Threats to Validity", since we discuss the assumptions we have made in our approach.

8.2.1 Rule-Element decomposition

The desire to achieve data parallelism stems from the presence of a large number of model elements. The nature of models as discussed in this thesis (and by common metamodelling technologies) implies that model elements can be grouped based on some shared attributes or relationships. When we say that a model conforms to a metamodel, we mean that its elements belong to a category of types (classes) described in the metamodel. It is in the definition of these types and their relationships which gives models their structure. Hence, the grouping of model elements by types is intuitive in the design of model access APIs and model management languages. Thus, when it comes to expressing the logic of model management tasks, these are often categorised based on the model element types. More concretely, in Epsilon's languages (and OCL), the program logic is defined within a *context*, where the context specifies the type of elements which said logic can operate upon. Of course, navigation is possible between elements, so the context is only a starting point. Nevertheless, the context provides a domain from which the data is drawn, though this need not be based purely on types. Further specialisation can be achieved by filtering elements (using the *guard* construct in Epsilon, for example). More generally, the collection may be explicitly specified, as is the case in EGX (through the *in* keyword).

The purpose of such groupings is to be able to define *rules*. A rule can be thought of as some arbitrary block of logic which takes as input an element conforming to some predefined criteria. As discussed, the most frequent and convenient way to group elements is based on their type, but the collection of applicable elements to the rule may be more complex, based on the structure of the program. In any case, the main point is that rules make certain assumptions about their input data, and hence a common logic can be defined to process elements (the input data) conforming to the criteria. The analogous hardware term for this is "Single Instruction Multiple Data" (SIMD).

Every element may have multiple rules in which they are applicable to (i.e. the element may satisfy the trigger condition for multiple rules in the program), so we can define the notion of a "rule-element pair" to be our base unit of "work". We have referred to this notion as a "RuleAtom" in the thesis – for example, "GenerationRuleAtom" in the case of EGX. We can then discuss the notion of *executing* a RuleAtom, because we have both parts of what is needed: the instruction (rule) and the data (element). Because, it does not make sense to execute a rule – basically a function – without its input parameter. Nor does it make sense to execute an element, since multiple rules may be applicable to a single element. Our parallelisation approach for model management tasks relies on this fundamental ability to decompose programs into rule-element pairs.

8.2.2 Independence of Rule-Element pairs

Having established an atomic unit of parallelisation, the next assumption is that these atoms can be executed independently of each other. More specifically, the outcome of the program should be the same regardless of the order in which these atoms are executed. If this property holds, then we can execute these atoms simultaneously without needing to perform any post-processing to ensure correct ordering of rule application. In other words, there should not be any conflict between rules, such that the result of one overwrites the other.

That is not to say that dependencies cannot exist, however. As we saw with EVL (see Section 4.3.6), dependencies between atoms or rules do not prohibit parallelisation. Rather, they increase the computational cost and overhead. Dealing with dependencies efficiently whilst allowing for parallel execution of dependent rules requires either caching, duplication (i.e. re-executing the dependency), or a bit of both. Our hybrid solution in EVL is an optimistic approach: we only perform caching after discovering that a rule is the target of a dependency. Under parallel execution, this means an atom which is dependent on may be executed twice (but no more). Whilst this solution achieves a good balance between memory consumption and avoiding re-execution, the nature of parallel execution means that a program with heavy use of dependencies will benefit less in terms of performance. The lesson learnt in this case is that programs should be written in a manner such that dependencies between rules are minimised. That said, if the rules depended on are lazy and invoked from only one other rule, then it is effectively like a function invocation, and there is no penalty under parallel execution in these circumstances. Thus, dependencies should be made lazy where possible.

8.2.3 Purity of Rules

We said that the body of a rule may consist of arbitrary logic. A rule takes as input elements from the model based on some user-specified criteria and/or type information. In this regard, a rule may be thought of as a function: it has an input, some logic and an output. The output is, in the case of EVL, the UnsatisfiedConstraint (i.e. the Constraint-element pair) if the body returns false. In the case of EGX, it is the text generated from the invoked EGL template. Since the execution engine manages the output of each rule, there isn't much "wrong" that the user can do. For instance, in EVL, we can make the Set of UnsatisfiedConstraints thread-safe, or have a thread-local collection, or even a thread-safe collection which is not a set, since the results are write-only. That is, the intermediate output of rules cannot be accessed at any arbitrary point during execution: it is deliberately insulated and managed by the engine.

However, this alone does not make invalid states and concurrency issues impossible within rules. If the language offers a means to mutate global variables or some other internal state, then mechanisms need to be put in place to prevent e.g. *ConcurrentModificationException* or non-deterministic behaviour caused by concurrent execution. This can be quite complex to cater for and degrade performance. Thus, our assumption is that the logic contained within rules are *pure functions*; they do not mutate global state, perform the same operation and produce the same result for a given input (model element).

8.2.4 Model access assumptions

To simplify the evaluation, we have assumed that models are loaded into memory so that they can be accessed quickly, especially given that we can also cache all model elements by their type. We have already discussed this in Section 3.4, however we are now in a position to state our recommendations regarding models more generally.

Given that our unit of parallelisation is rule-element pairs, it should be possible to navigate the model concurrently. That is, the underlying modelling technology should be able to deal with multiple threads of execution accessing arbitrary elements and attributes. Models are essentially typed and attributed graphs, and so model elements often have navigable relationships to other model elements. In this thesis, we have focused only on read-only model management tasks, so in theory this property should always hold. However, in practice, the implementation may not be able to deal with multi-threaded queries. For example, the org.w3c.dom API in Java is not thread-safe, so an XML model driver implemented using this will not work with our parallel execution approach unless model access operations are synchronized. This obviously degrades performance, especially given how frequent model accesses occur in model management programs. This can be remedied by navigating the entire model and caching it prior to parallel execution, but that is effectively the same as re-adapting the model to a different underlying technology.

To use our index-based distributed execution strategy outlined in Section 5.2.2, we require that the model access API returns elements belonging to a given type in a deterministic order. For simplicity we refer to types, but as previously mentioned, the collection of elements applicable to a rule may be based on more than purely type information. In any case, what we are really interested in is that the rule-element pairs can be constructed in a deterministic order. In other words, the list of atoms (jobs) is ordered predictably. Whilst this is straightforward from the model management program's perspective (rules and types are iterated over in declared order), a modelling API which uses a HashSet for its underlying elements storage may not return them in the same order each time, which therefore means each invocation of the program is unique despite the same inputs, and so we cannot rely on referring to atoms based on their index.

Moreover, the number of atoms (and thus, the number of model elements) needs to be *finite*. Our current implementation cannot be used in cases where the model is "live" – i.e. when it is modified during execution. The rule-element pairs must be countable prior to execution. This doesn't necessarily require that the model be loaded into memory, however. Rather, the index-based approach requires that random access to a specific model element is a constant-time operation. In our implementation, we achieved this by eagerly computing the list of atoms, where each atom contains a direct reference to the in-memory model element. If the model cannot be loaded into memory, then the direct reference could be replaced with a reference to the ID of the model element, assuming that the modelling technology supports unique IDs for elements. This is the case for EMF models, however just because we can locate an individual element by an ID, it does not necessarily guarantee constant-time access. The lookup of elements by ID may still be inefficient, if the ID to element mapping is not stored using a hash table or there are frequent collisions.

To summarise, modelling technologies and model access APIs should have the following properties to be able to take advantage of the performance optimisations presented in this thesis:

- Thread-safe (and non-blocking) for querying operations
- Deterministic ordering when querying its contents and/or types
- Constant-time access to random elements
- Finite and read-only during execution.

In-memory data structures make it convenient to achieve all of the above properties, hence our evaluation relied on models being loaded into memory prior to execution of the program. However, so long as these properties hold, there is theoretically no reason that e.g. database-backed models could not benefit from the parallel and distributed execution approaches described in this thesis.

8.2.5 Program and model structure

Our parallelisation approach works well when there are many rule-element pairs. Our decomposition was deliberately designed in a way to maximise the potential parallelism. This data-parallel approach arose from the observation that performance issues are prevalent due to a large number of model elements. Whilst this may seem obvious and intuitive, it is not guaranteed to provide performance benefits under some extreme circumstances.

For example, suppose that a very large model contains various types of elements, where these are all children (but not subtypes) of a single root element. If the user declares this root element as the context (i.e. the type which the rules applied to) for all rules, and the number of rules is less than the number of cores in the system, then clearly the maximum level of parallelism is limited by the manner which the user has structured their program and/or model. Where possible, *for* loops over types should be factored out to take advantage of the language constructs – i.e. by defining them such that rules operate over types which contain many elements. If this is not possible, then at least imperative *for* loops should be avoided, and instead one of the declarative operations (as discussed in Section 6.2) should be used to take advantage of parallelism that way.

There are various other scenarios one could imagine which limit the maximum potential parallelism – e.g. a heavy reliance on attributes rather than types. In any case, the important takeaway is to effectively use the language constructs and structure the model in a way which makes it easy to structure model management programs based on element types. A poorly structured program or ineffective use of types at the metamodel level can limit the effectiveness of parallelisation.

8.3 Future Work

As is usually the case with doctoral theses, there is not enough time to achieve everything that was initially within the scope of the project. Although the main fundamental aims and objectives of the thesis have largely been achieved, there remain some notable omissions which, if time allowed, would make the thesis' contributions more well-rounded. There is also a substantial amount of complementary work which can build on top of the ideas and artefacts developed as part of this project to further improve performance and efficiency of model management programs. That said, we have already made significant progress in some areas presented in this section.

8.3.1 **Parallel Model Comparison**

Continuing with the theme of parallelising rule-based Epsilon languages, a notable one which, to our knowledge, has no direct "competitor" with similar capabilities is the Epsilon Comparison Language (ECL), which allows users to declaratively compare models – even those conforming to different metamodels – with custom logic for defining equivalence between model elements. ECL is algorithmically very similar to EVL: rules are executed in the context of model elements and the logic in each rule is a Boolean expression or block. The main difference is that there are two model elements: one from the "left" model and one from the "right" model. That is, there are two "*self*" objects, rather than one. Instead of the ConstraintTrace and UnsatisfiedConstraints, we have a *MatchTrace*, which tracks which elements have been declared as being "equal" to each other based on the user-defined logic of the comparison rules. Instead of the "satisfies" operation, we have the "*matches*" operation which allows rules to invoke other rules by name. As with EVL, rules may be declared as lazy and have a guard to limit their applicability.

We have implemented both an annotation-based and automatic data-parallel algorithm for ECL. We have also implemented a parallel "*matches*" operation. However, we have not been able to thoroughly test and evaluate this at the time of writing.

8.3.2 **Parallel Pattern Matching**

Another important model management task is pattern matching, which is particularly computationally expensive even for relatively small models. The Epsilon Pattern Language (EPL) [62] is another rule-based language, where rules are patterns and cannot have dependencies between them. The language constructs and execution semantics are extremely complex, by far the most complex of any of Epsilon's languages. This is because each pattern can define any number of *roles* (essentially, variable bindings) which are drawn from *domains* (arbitrary collections of model elements), and each role and domain may have dependencies on previous roles and domains declared in the same pattern. There are also further constructs restricting the applicability of roles and what constitutes a match and not a match. Whilst the semantics and features of EPL are far too numerous to discuss here, the key takeaway is that EPL is fundamentally different in its execution compared to other ERL languages because of its combination generator: where other languages have a predetermined flat structure of rule-element pairs, in EPL these are only discoverable at runtime, so in effect the number of *for* loops is dynamic.

Of course, since each pattern is independent, we could execute each one in parallel, which only requires us to make the *PatternMatchModel* – the data structure used to hold pattern matches as an EMC-compliant model – thread safe in the data structures it uses. Failing that, we could even fully synchronize access. The model will only be written to during execution and read from after all patterns have been executed, so we could use a similar solution to that of the UnsatisfiedConstraints in EVL, making the intermediate data structures (or even the entire model) thread-local and merging them afterwards. The expensive part of EPL is generating the combinations and executing all of the intermediate expressions resulting from these. Since each pattern is expensive, a parallelisation solution should try to parallelise execution of roles. This appears to be an ideal use case for Fork/Join parallelism; where the problem is recursively divided into smaller subtasks. The same principle behind a recursive parallel execution algorithm for calculating the Fibonacci sequence can be used here, although undoubtedly due to the complexity of the language there are many factors to consider, so it is unlikely that our existing parallelisation framework would work without any modifications for this kind of problem.

At the time of writing, we have been able to parallelise the execution of patterns, since patterns do not have dependencies on each other. However, there are no performance benefits for EPL programs with a single pattern, or when there is only a single computationally expensive pattern.

8.3.3 Parallel and Incremental Model Validation

Since the beginning of the project, incremental execution has always been on the radar for improving performance, if not as a primary means, then as a complement to parallelisation. We have postulated that incrementality, laziness and parallelism are orthogonal means to improve performance – in theory, all three techniques can be combined. Given that incrementality is the most commonly researched solution in the MDE community, and given its obvious and large performance benefits, it is important to investigate the extent to which parallel and incremental execution can be combined. Incrementality is fundamentally about partial execution of a program, not necessarily a radically different execution algorithm. In terms of our architecture, it essentially requires keeping track of the rule-element pairs which have been executed and their results. In EVL for example, the *ConstraintTrace* already keeps track of this information – we just need to disable our optimisation which minimises writes to this so that the full results are written.

Incremental execution needs to persist the execution trace (in the case of EVL, the ConstraintTrace) in non-volatile memory so that subsequent invocations can determine which rule-element pairs to re-execute based on changes in the model or program. A scalable solution is to use a database for persistence, which may be written to during execution periodically so that if the program crashes, not everything is lost. Depending on the implementation, this may pose a significant challenge when combined with parallelism. Firstly, writing to a database during parallel execution would likely need to be handled asynchronously: perhaps by means of a separate thread which polls the ConstraintTrace and writes the results sequentially to the database. There is also the matter of whether execution should support live changes to the model and/or script. In our parallel architecture, we have (quite reasonably) assumed that the model and script are immutable. An incremental solution is *reactive* if it can process changes on-the-fly and update the execution accordingly. The non-deterministic scheduling can be a problem if it means that some jobs which have been submitted (or are in progress) are no longer valid. We could get around this by having an "isCancelled" flag associated with each job, but this requires additional bookkeeping. Another problem is that changes not only affect jobs directly, but also dependencies. Regardless of the implementation, an advanced incremental solution is unlikely to be straightforward to integrate with our existing parallel approach without ensuring that jobs can be cancelled immediately on-the-fly.

There is currently a prototype implementation of incremental EVL [298], however after speaking with the lead developer on numerous occasions, it is not in a "finished" state such that it can be considered stable, fully functional and production-ready. As such, given the timescales, we have unfortunately been unable to investigate the possibility if integrating these two works to create an "ultimate" model validation solution. To our knowledge at the time of writing, there are no works which attempt to combine incremental and parallel execution in any model management task.

8.3.4 **Parallel Model Merging, Migration and Transformation**

In this thesis, we have only considered read-only model management tasks – that is, tasks where the model is not modified. Our assumption of immutability extends beyond the model, to the program itself as well. Whilst these are reasonable first steps, it is important to note that some of the most useful model management tasks inevitably require models to be modified. Most notably, model-to-model transformations – by far the most widely studied model management task – poses many challenges for arbitrarily complex transformation rules and execution semantics as available in the Epsilon Transformation Language (ETL). Even if the output model is created rather than modifying the input, there are still fundamental challenges to be addressed if our approach is to be re-used for these tasks. In our initial attempt to parallelise ETL, we found the most immediate challenge to be the lack of thread-safety offered by EMF during out testing. Since Epsilon supports any modelling technology, we cannot generalise a thread-safe solution at the execution engine level. Future work could focus on developing a concurrent adapter or extension to the EMC API so that non-thread-safe modelling technologies can be supported "automatically". For example, rather than writing changes directly to the underlying model resource(s), we could buffer these Create/Update/Delete operations, each with an associated timestamp and/or priority. The adapter could then automatically infer the appropriate final state based on some algorithm and commit the changes sequentially. The solution would need to be much more elaborate if the modified / created model may also be read from during execution, but for "write-only" this could be a starting point.

As noted with the case of ECL, even a "read-only" model management task can face similar challenges to model transformation due to the order of rule applications. There will inevitably be fundamental concurrency challenges to solve for model migration and model merging tasks. Future work could begin by investigating the extent to which Epsilon Flock and the Epsilon Merging Language (EML) can be parallelised using our approach. We have already identified the parallelisable "atoms" for EML as it extends ERL: the *MergeRule*. This is similar in principle to the ECL *MatchRule* as it takes as input two parameters but produces an output parameter. With the exception of EPL, EML is perhaps the most complex ERL extension to parallelise due to this fact.

8.3.5 **More diverse evaluation**

We have evaluated our approach using a relatively small sample of (meta)models and model management scripts. Furthermore, our models have been EMF-based and fully in-memory. These limitations are not, in principle at least, of our approach, but of our inability to find suitable non-proprietary alternatives which can be freely published for other researchers to verify and run their own experiments with. Unfortunately, as alluded to in Chapter 2, many large models and complex / long-running model management programs, as well as modelling technologies, are proprietary.

There is also the issue of state explosion: even with our relatively small sample, we had to resort to generating our test scenarios as mentioned in Chapter 3 as there are thousands of scenarios we could meaningfully test. Nevertheless, further experiments with different hardware to assess the scalability of our distribution approach (as described in Chapter 5) when worker nodes have heterogenous hardware resources would help to strengthen the evaluation. It would also be interesting to compare the performance of our bespoke JMS-based implementation to e.g. the Flink implementation (or another distributed processing framework).

Whilst we argued that our solution is generalisable due to EMC, it would be beneficial to perform further experiments with not only different models and programs, but also different modelling technologies. Most notably, we would like to see the relative performance of our solutions when working with database-backed models. As a starting point, the existing EMF models used in our evaluation could be adapted / stored using various persistence technologies in the EMF ecosystem. Most notably: NeoEMF [29], Connected Data Objects (CDO) [27] and Hawk [299] are all good options. They also allow for evaluation of much larger models which would otherwise not fit into memory to be benchmarked against.

8.3.6 **Static Analysis**

In the beginning, we outlined three main optimisation techniques for improving performance not just of model management programs, but any non-trivial application or language: laziness, incrementality and parallelism. We explored how these techniques have been proposed and applied to model management programs in the literature. We also alluded to the wealth of research in the database community on query optimisation, where the kind of expressions which are optimised by even the most primitive query planners are far beyond the capabilities of most model querying tools. All three optimisations we identified can be improved by having more information about the program they're trying to optimise. This information can come at runtime or computed beforehand. Runtime information is useful for making optimisations on-the-fly, however where possible, is preferable to have as much information as possible prior to execution so that the execution algorithm / semantics can be optimised beforehand. This requires static analysis of the program.

With regards to the contributions of this thesis, static analysis can be used to enhance the optimisations in several ways. For instance, we could re-order the execution of rules so that rules which are depended on are executed first. Another use is query optimisation and suboptimal code detection, continuing the work of Wei and Kolovos' [235]. With the new *count* and *nMatch* operations, we can detect cases where these operations can be used in place of *select*. In keeping with the theme of making optimisations transparent to the user and not requiring modifications to existing programs, we propose that rather than detecting suboptimal code and warning the user, we instead automatically substitute the suggested optimisation, provided that the result is the same (i.e. the user cannot distinguish between the original and optimised expressions).

Static analysis is also necessary for further improving the efficiency of our distributed approach. In our architecture, we treat the model management program as a black box, decomposing it into rule-element pairs without any concern for data locality: all rules and all elements are treated homogenously. Whilst we realise that computation costs can vary significantly between each job, we have tried to compensate for this. The absolute runtime cost of each job is relatively small due to our fine-grained granularity, and we shuffle the jobs and use smaller batches to ensure a uniform distribution. However as shown in our experiments, relying on random distribution does not give consistent performance, and can in some cases create very unequal workloads. Nevertheless, with sufficient load balancing and a high granularity our approach works in the context we have presented it. With static analysis, we can go further by exploiting data locality. Rather than having each worker load the full program and model(s), we could exploit knowledge of the program prior to distribution to determine dependencies between rules, and the parts of the model(s) each rule accesses (both directly and indirectly). We can build a graph-like data structure (perhaps even a model) of the program and use this information to intelligently assign subsets of the program to workers, so that each worker only needs the required subset of the model and program to run. Whilst this approach can slightly improve runtime performance (or at least, improve consistency), the main benefit is the reduction in memory requirements. In our evaluation, we have only been working with in-memory models, however large models should typically be stored in more scalable backends such as databases. When model accesses become non-uniform in cost – perhaps even the model persistence is distributed many computers – data locality becomes a critical factor. In cases where models are too large to fit into memory and expensive to access specific model elements at random, we need to ensure that when we query the model, we obtain as much information as possible that is relevant for the program execution. That is, the goal is to minimise model access operations. With our current random assignment strategy, the opposite occurs. With static analysis, we can try to optimise the distribution to minimise the proportion of the model accessed by each worker. This is a common concern in distributed model management (and distributed execution in general), as evident by the works reviewed in Chapter 2 (for example [213]).

8.3.7 **Intelligent job distribution**

The distributed execution approach described in Chapter 5 could be improved and simplified by automatically determining the optimal proportion of jobs statically assigned to the master and batch factor parameters based on the operating environment. Future work could employ analysis of the model and script as well as reformulation of the parameters so that users can more easily determine optimal values without experimentation by, for example, providing normalised values for network speed and relative strength of master to workers.

It would also be interesting to experiment with alternative methods of job assignment and/or shuffling. Our current approach randomises the order of jobs due to lack of information regarding the execution cost of each job. However, it may also be possible to use more sophisticated shuffling strategies based on heuristics or from profiling at runtime, which can then be used to determine which jobs are likely to be time-consuming and thus can be used for fairer assignment.

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 [300] 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.

8.3.8 **Compiled code generation**

In evaluating our bespoke parallel operations and parallel streams, we compared the performance of EOL to Java to determine the extent of the overhead incurred by an interpreted language. We found this difference to be extremely large. Despite us implementing the program identically using Streams in both EOL and in Java, using Epsilon's Model Connectivity API directly and EOL's data structures, parallel EOL was outperformed even by sequential Java Streams. However, it should be noted that our bespoke parallel operations still outperformed parallel streams in terms of speedup relative to the sequential variants. What surprised us most is how concise the hand-written Java code turned out to be, due mainly to the clean and concise design of both EMC (for model querying) and Java Streams APIs. With EMC, we only needed a *PropertyGetter* which we can pass the name of the property to retrieve for a given model element. Whilst there is a lot of unsafe casting, the engineering effort required to manually implement an EOL query using Java Streams and EMC is not drastically different from writing the code in EOL, as demonstrated by the relatively similar source lines of code between EOL and Java.

Given the vast performance benefits of compiled code and based on our experiments of manually hand-coding EOL model query programs in Java, there is a clear case for ensuring that the final runtime code is compiled to Java rather than interpreted. Of course, hand-coding such queries requires a degree of expertise with the programming language and some APIs, which is not as user-friendly as EOL, since EOL was designed precisely for these kinds of tasks. We therefore strongly advocate that future work investigates the extent to which our parallelisation architecture can still be leveraged and applied to compiled EOL, once a suitable compilation infrastructure is in place. To some extent, creating a compiler for Epsilon languages also depends on having a static analysis infrastructure in place, so that errors can be reported at compile-time where possible. Whilst we cannot foresee any obvious issues with adapting our parallel architecture to a compiled setting, there will inevitably be engineering challenges associated with parallelising compiled code as oppose to interpreted code, especially if it means much of the AST is effectively eliminated or replaced. More interestingly however is whether the performance benefits of parallelisation will be greater or lesser in compiled code vs. interpreted. Given the complex nature of modern hardware and software platforms, it would be unwise to attempt to speculate on the efficiency of our approach in a compiled scenario. Nevertheless, in our view it is an important aspect to investigate given the undeniable performance superiority of compiled code over interpreted.

8.4 Concluding Remarks

This thesis has taken an engineering-centric approach to research. It has focused on a specific research topic and methodology: the parallelisation of model management tasks, with the goal of making a small but significant contribution in the area of improving the scalability of model management programs. As with all research and engineering problems, there are many ways one can approach a task, and at various levels of abstraction – even projects which are fundamentally focused on optimisation such as this one. The author's choice of methodology in approaching this problem is a reflection on their unique combination of personality and skillset: no two researchers, given the same problem statement, will produce identical solutions due to human nature.

For any given problem, there will always be a trade-off between an actionable, instantiable solution for a specific instance where the problem applies, and the solution being too specific to the given instance, making the generalisation a task for the reader. Abstract solutions can only be evaluated by concrete instantiations. Concrete solutions can be evaluated with relative ease, but their external validity is more difficult to assess. Even if a concrete solution is not widely adopted and generalised, this does not mean it has no value: indeed, by construction and by evaluation, a concrete solution can be demonstrated to have real-world value if it achieves its goals, even for its own specific instantiation. By contrast, an abstract solution has unknown value if it has no instantiation, and thus cannot be empirically evaluated. This research has taken a "bottom-up" approach of developing concrete solutions and arguing for their external validity, often by inference or implication.

In conclusion, I hope that this thesis stands the test of time in its solutions and approach and the reader finds the works presented in this thesis (and the accompanying artefacts) to be of value. Ultimately, I hope that, if successful, this thesis inspires others to be confident in their approach to research, without fear of it being labelled as "just engineering" in an academic context.

Appendices

This section contains listings which were deemed unsuitable for the main text.

I. Source code and artefacts

The full source code for this project can be found in the main Epsilon repository:

https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git

The work described in this thesis has been integrated into the 2.0 release of Epsilon.

The evaluation artefacts and datasets can be found in the following repository:

https://github.com/epsilonlabs/parallel-erl

Source code for distributed EVL (as described in Chapter 5) can be found at:

https://github.com/epsilonlabs/distributed-evl

It is possible that the material used to produce this thesis (source code, binaries, scripts, models etc.) may become unavailable or inaccessible. As a redundancy, a complete, self-contained repository containing all relevant artefacts and data from experiments has been compiled and is available at:

https://github.com/SMadani/PhD

Should the reader want to reproduce experiments or further develop these, the author is available to contact for any questions or assistance via e-mail: sinadoom@gmail.com (or alternatively, sina.madani@york.ac.uk).

II. Custom *ConcurrentLinkedQueue* implementation

```
public class SizeCachingConcurrentQueue<E> extends ConcurrentLinkedQueue<E> {

   protected static final Object NULL = new Object();

   protected static final Object replaceWithNull(Object o) {
      return o == null ? NULL : o;
   }

   protected static final <T> T convertToNull(Object o) {
      return o == NULL ? null : (T) o;
   }

   final AtomicInteger size;

   public SizeCachingConcurrentQueue() {
      size = new AtomicInteger();
   }

   public SizeCachingConcurrentQueue(Collection<? extends E> initial) {
      super(initial);
      size = new AtomicInteger(initial.size());
   }

   @Override
   public int size() {
      return size.get();
   }

   @Override
   public E peek() {
      return convertToNull(super.peek());
   }

   @Override
   public E poll() {
      E res = convertToNull(super.poll());
      size.decrementAndGet();
      return res;
   }

   @Override
   public boolean contains(Object o) {
      return super.contains(replaceWithNull(o));
   }

   @Override
   public boolean remove(Object o) {
      if (super.remove(replaceWithNull(o))) {
         size.decrementAndGet();
         return true;
      }
      else return false;
```

```
}

// Compiler-pleasing magic
protected final Consumer<? extends E> offerSuper = super::offer;

@Override
public boolean offer(E e) {
   ((Consumer) offerSuper).accept(replaceWithNull(e));
   return size.incrementAndGet() > 0;
}

@Override
public boolean equals(Object o) {
   if (this == o) return true;
   if (!(o instanceof Collection)) return false;
   Collection<?> that = (Collection<?>) o;
   if (this.size() != that.size()) return false;

   for (
      Iterator<?> thisIter = this.iterator(),
                  thatIter = that.iterator();
      thisIter.hasNext() && thatIter.hasNext();
   ) {
      if (!Objects.equals(thisIter.next(), thatIter.next()))
         return false;
   }
   return true;
}

@Override
public boolean addAll(Collection<? extends E> c) {
   boolean res = false;
   if (c != null && c != this) {
      Collection filtered = c.stream()
         .map(SizeCachingConcurrentQueue::replaceWithNull)
         .collect(Collectors.toList());

      res = super.addAll(filtered);
      size.getAndAdd(filtered.size());
   }
   return res;
}


@Override
public Iterator<E> iterator() {
   return new InternalIter(super.iterator());
}

private final class InternalIter implements Iterator<E> {
   Iterator<E> delegate;

   InternalIter(Iterator<E> delegate) {
      this.delegate = delegate;
```

```
            }

            @Override
            public boolean hasNext() {
                return delegate.hasNext();
            }

            @Override
            public E next() {
                return (E) convertToNull(delegate.next());
            }

            @Override
            public void remove() {
                delegate.remove();
                size.decrementAndGet();
            }
        }


        @Override
        public int hashCode() {
            return Objects.hash(size);
        }

    }
```

III. Original short-circuiting infrastructure solution

Listing III.1 shows the *ExecutionStatus* object (minus the public getter methods for the *result*, *exception* and *inProgress* fields). As its name suggests, the purpose of this class is to indicate whether task execution has completed, which may have an associated exception or result object. It also offers a convenience method *waitForCompletion*, which blocks the calling thread until the *inProgress* flag is set to false. The object synchronizes on itself, so it can only be used for a single task at a time, although re-use is possible. When a task begins, the *register* method is called, which sets the *inProgress* flag to true. Calls to the *complete* method make registration available again (setting *inProgress* to false).

```java
public final class EolConcurrentExecutionStatus {

  private volatile boolean inProgress = false;
  private Object result;
  private Throwable exception;

  public synchronized boolean register() {
    if (!inProgress) {
      exception = null; result = null;
      return inProgress = true;
    }
    else return false;
  }

  public synchronized void complete() {
    inProgress = false;
    notify();
  }

  public synchronized void completeWithResult(Object result) {
    this.result = result;
    complete();
  }

  public synchronized void completeExceptionally(Throwable exception) {
    this.exception = exception;
    complete();
  }

  public synchronized boolean waitForCompletion() {
    while (inProgress) {
      try {
        wait();
      }
      catch (InterruptedException ie) {}
    }
    return exception == null;
  }
```

Listing III.1 *ExecutionStatus* object definition (excluding getters)

We now turn to the simpler of the two Future completion methods: *collectResults* as shown in Listing III.2. Much of the code deals with manipulating the ExecutionStatus object based on its current status. The key part is that a separate "AwaitCompletion" thread is used to loop through all of the Futures and block until they are complete. A completed Future returns its result immediately, so here we are simply collecting the results if they are all already complete. Note also that the Future results are added to an ordered collection sequentially in encounter order, which is how the result of *parallelSelect* is deterministically ordered. Of course, whilst waiting for the jobs to complete, an exception may be encountered in one of the jobs, so the ExecutionStatus object must be polled for an existing exception if it was already encountered before the AwaitCompletion thread started. If an exception is encountered whilst waiting for the Futures, the ExecutionStatus is signalled which terminates the *waitForCompletion* method and populates the *exception* field. The main thread is blocked whilst the AwaitCompletion thread is running, which may also be blocked whilst waiting for the Futures. By design, only one of these threads can be running at given time: if all jobs complete successfully, we signal the ExecutionStatus, which wakes up the main thread, and similarly for exceptional completion. When the AwaitCompletion thread terminates, we can guarantee that the ExecutionStatus is also not *inProgress*. We also guarantee that the AwaitCompletion thread terminates before exiting the method with the results list.

The more complex short-circuiting operation is shown in Listing III.3. At first glance the algorithm is broadly similar, being bloated by the exception-handling code which may occur (or already have occurred) at any point. There are however two key differences. Firstly, since only a single result is needed which will be set by the caller on the ExecutionStatus object when required (see the previous section's handling of *parallelSelectOne*), we do not need a results collection and therefore the Futures can be of any type, since we don't care about the results. Infact, in most cases the Futures are obtained by submitting a Runnable as opposed to a Callable, which has no return type (so the Futures always return null). We still employ the same strategy of having an AwaitCompletion thread loop through the Futures though so that in the event of a result not being found (i.e. the ExecutionStatus.*completeWithResult* method never being called), we don't wait forever. The AwaitCompletion thread will signal the ExecutionStatus when all jobs have finished, or an exception occurs. The main thread waits for this signal by calling the *waitForCompletion* method as before, however this time there is additional work to be done. If the ExecutionStatus is signalled due to a short-circuit completion (i.e. the *completeWithResult* method is called), then there is no point in waiting for the remaining Futures to complete, so we must not only terminate the AwaitCompletion thread but also ensure that any running jobs are forcefully cancelled (the true flag passed to *Future.cancel* indicates the job should be terminated even if it is in progress) to avoid wasting CPU time. Finally, if no exception occurred, we return the result of the ExecutionStatus object.

One of the issues with short-circuiting is that because jobs may be terminated abruptly (by calling *`future.cancel(true)`*) once a result is found, this leaves the ThreadLocals in an inconsistent state. Variables put into the executing threads' FrameStacks will not have had chance to be cleared and the execution trace will be incomplete. This itself is not the issue, since we clear all thread-local state and the ExecutorService (and thus the threads it created) after a parallel task. The problem lies with the lack of distinction between abrupt and normal termination in our parallelisation context. Normally thread-local state is merged into the main thread – in this case the variables in the thread-local FrameStacks – to preserve compatibility with existing programs that relied on the availability of variables after a certain block of code has been executed. For example, variables declared in the *check* block of an EVL constraint should be made available in the *fix* and *message* blocks. In most cases however this is not required. When a code block or expression may terminate abruptly, no guarantees can be made with regards to consistency of available variables. Indeed, in most cases (such as *parallelSelectOne*), no consistency guarantees can even be made about the result. Our solution to this problem is to introduce an additional Boolean parameter to the *beginParallelTask* method. A *true* value will avoid merging thread-local state into the parent (main) thread, whilst *false* (the default value) will save thread-local state before disposing of them.

The clever part of this process is how ordering is guaranteed. Note that the collection we created to contain the eventual computation results is ordered, and that the looping is performed sequentially. When we submit a job to the ExecutorService, we get back a Future which is a container for the result of the job. The Future can be thought of as a box which will be populated once the computation is complete. This computation of course happens in parallel to other computations (jobs), however the “box” (Future) itself is added to the collection of results in the order of submission, which is always deterministic for ordered collections. Thus, when we come to retrieve the results later on, we can guarantee that the order in which the results collection is populated will be the same order that the jobs were submitted in, even though the order of completion for the jobs may be arbitrary. In other words, the order of the containers (Futures) is deterministic, but the order in which they are populated as a result of their corresponding job’s completion is not. Since we must wait for all jobs to complete in the *parallelSelect* case, the order of completion does not matter.

```
<R> List<R> collectResults(Collection<Future<R>> futures) throws EolRuntimeException {
  final EolConcurrentExecutionStatus status = getExecutionStatus();
  Throwable statusException = status.getException();
  if (statusException != null) EolRuntimeException.propagateDetailed(statusException);
  else if (futures.isEmpty()) {
    if (status.isInProgress()) status.complete();
    return Collections.emptyList();
  }
  else if (!status.isInProgress()) status.register();

  final List<R> results = new ArrayList<>(futures.size());

  final Thread compWait = new Thread(() -> {
    try {
      for (Future<R> future : futures) {
        if (status.isInProgress()) results.add(future.get());
        else return;
      }
      status.complete();
    }
    catch (Exception ex) {
      if (status.getException() == null || status.isInProgress())
        status.completeExceptionally(ex);
    }
    finally {
      if (status.isInProgress() && status.getException() == null)
        status.complete();
    }
  });
  compWait.setName(getClass().getSimpleName()+"-AwaitCompletion");
  compWait.start();

  try {
    if (!status.waitForCompletion()) {
       compWait.interrupt(); shutdownNow();
       EolRuntimeException.propagateDetailed(status.getException());
    }
  }
  finally {
    if (compWait.isAlive()) try {
      compWait.join();
    } catch (InterruptedException ie) {}
  }
  return results;
}
```

Listing III.2 *EolExecutorService.collectResults* implementation

```
Object shortCircuitCompletion(Collection<Future<?>> jobs) throws EolRuntimeException {
  final EolConcurrentExecutionStatus status = getExecutionStatus();
  Throwable statusException = status.getException();
  if (statusException != null) EolRuntimeException.propagateDetailed(statusException);
  else if (jobs == null || jobs.isEmpty()) {
    if (status.isInProgress()) status.complete();
    return status.getResult();
  }

  final Thread compWait = new Thread(() -> {
    try {
      for (Future<?> future : jobs) {
        if (status.isInProgress()) future.get();
        else return;
      }
      status.complete();
    }
    catch (Exception ex) {
      if (status.isInProgress()) status.completeExceptionally(ex);
    }
    finally {
      if (status.isInProgress() && status.getException() == null)
        status.complete();
    }
  });
  compWait.setName(getClass().getSimpleName()+"-AwaitCompletion");
  compWait.start();

  try {
    boolean success = status.waitForCompletion();
    compWait.interrupt();
    if (!success) {
      shutdownNow();
      EolRuntimeException.propagateDetailed(status.getException());
    }
    else for (Future<?> future : jobs) future.cancel(true);
  }
  finally {
    if (compWait.isAlive()) try {
      compWait.join();
    } catch (InterruptedException ie) {}
  }
  return status.getResult();
}
```

Listing III.3 *EolExecutorService.shortCircuitCompletion* implementation

```
public EolExecutorService beginParallelTask(ModuleElement entryPoint) throws
EolNestedParallelismException {

   if (!isParallelisationLegal() ||
executor.getExecutionStatus().isInProgress()) {
      throw new EolNestedParallelismException(entryPoint);
   }
   initThreadLocals();
   if (!executor.getExecutionStatus().register()) {
      throw new EolNestedParallelismException(entryPoint);
   }
   isInParallelTask = true;
   return executor;
}

public Object endParallelTask() throws EolRuntimeException {
   Object result = null;
   if (isInParallelTask) {
      EolConcurrentExecutionStatus status = executor.getExecutionStatus();
      if (status.isInProgress()) {
         throw new IllegalStateException("Execution is in progress!");
      }
      else if (status.waitForCompletion()) {
         result = status.getResult();
      }
      else {
         EolRuntimeException.propagateDetailed(status.getException());
      }
   }
   executor.shutdownNow(); executor = null;
   clearThreadLocals();
   isInParallelTask = false;
   return result;
}

public boolean isParallelisationLegal() {
   return !isParallel() && ConcurrencyUtils.isTopLevelThread();
}

public boolean isParallel() {
   return isInParallelTask;
}
```

Listing III.4 Code extracts used for initialisation and teardown of parallel tasks

IV. Parallel operation evaluation programs

```
return Movie.allInstances().atLeastNMatch(m | m.nestedActors(), getN());

operation Movie nestedActors() : Boolean {
  var actors = Person.allInstances().asSequence();
  var toIndex = actors.size() / getN();
  var subActors = actors.subList(0, toIndex);
  return self.persons
    .exists(ac |
      subActors.exists(mp |
        ac.hashCode() == mp.hashCode() and
        mp.movies.size() == ac.movies.size()
      )
    );
}

operation getN() : Integer {
  return 32;
}

operation Person hasCoupleCoactorsRare() : Boolean {
  var nTarget = self.movies.size() + getN();
  return self.coactors().count(co | co.areCoupleCoactors(self)) == nTarget;
}

operation Person hasCoupleCoactors() : Boolean {
  return self.coactors().exists(co | co.areCoupleCoactors(self));
}

operation Person areCoupleCoactors(co : Person) : Boolean {
  return
    self.name.compareTo(co.name) < 0 and
    co.movies.size() >= 3 and
    self.areCouple(co);
}

operation Person coactors() : Set(Person) {
  return self.movies.collect(m | m.persons).flatten().asSet();
}

@cached
operation Person areCouple(p : Person) : Boolean {
  return
     self.movies.excludingAll(p.movies).size() <= (self.movies.size() - 3);
}
```

Listing IV.1 *atLeastNMatch* EOL program for IMDb models

References

[1] S. Madani, “Parallel First-Order Operations (presentation),” 16 October 2018. [Online]. Available: https://www.slideshare.net/SinaMadani/parallel-firstorder-operations-119601295.

[2] S. Madani, “Towards Parallel and Lazy Model Queries (presentation),” 17 July 2019. [Online]. Available: https://www.slideshare.net/SinaMadani/parallel-queries.

[3] S. Madani, “The EGL Co-Ordination Language (EGX),” Eclipse Foundation, [Online]. Available: https://www.eclipse.org/epsilon/doc/egx. [Accessed 26 June 2020].

[4] D. S. Kolovos, R. F. Paige and F. A. Polack, “Eclipse Development Tools for Epsilon,” in *Eclipse Summit Europe, Eclipse Modeling Symposium*, 2006.

[5] S. Kent, “Model Driven Engineering,” in *Integrated Formal Methods Third International Conference*, Turku, 2002.

[6] G. Mussbacher et al., “The Relevance of Model-Driven Engineering Thirty Years from Now,” in *17th International Conference on Model-Driven Engineering Languages and Systems*, Valencia, 2014.

[7] P. Mohagheghi and V. Dehlen, “Where Is the Proof? - A Review of Experiences from Applying MDE in Industry,” in *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications (ECMDA-FA '08)*, Berlin, 2008.

[8] S. Madani and D. Kolovos, “Re-implementing Apache Thrift with MDE (guess what happened!),” 9 December 2016. [Online]. Available: https://modeling-languages.com/re-implementing-apache-thrift-with-mde/. [Accessed 27 February 2020].

[9] D. S. Kolovos, R. F. Paige and F. A. Polack, “The Grand Challenge of Scalability for Model Driven Engineering,” in *MODELS 2008*, Toulouse, 2009.

[10] M. Smith, *Parallel Model Validation,* University of York, 2015.

[11] “Who is using Epsilon?,” Eclipse Foundation, [Online]. Available: https://www.eclipse.org/epsilon/users/. [Accessed 29 February 2020].

[12] S. Madani, “Accompanying artefacts,” [Online]. Available: https://github.com/SMadani/PhD. [Accessed 29 June 2020].

[13] M. Brambilla, J. Cabot and M. Wimmer, Model-Driven Software Engineering in Practice, Morgan & Claypool Publishers, 2012.

[14] W. Kozaczynski and S. Sendall, “Model Transformation: The Heart and Soul of Model-Driven Software Development,” *IEEE Software,* vol. 20, no. 5, pp. 42-45, 2003.

[15] “First-class citizen,” Wikimedia Foundation, [Online]. Available: https://en.wikipedia.org/wiki/First-class_citizen. [Accessed 25 August 2016].

[16] A. Rodrigues da Silva, “Model-driven engineering: A survey supported by the unified conceptual model,” *Computer Languages, Systems and Structures,* vol. 43, no. C, p. 139–155, 2015.

[17] “XML Schema specification,” [Online]. Available: http://www.w3.org/2001/XMLSchema.xsd. [Accessed 30 April 2016].

[18] A. G. Kleppe, “A Language Description is More than a Metamodel,” *Fourth International Workshop on Software Language Engineering,* 1 October 2007.

[19] D. Moody, “The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering,” *IEEE Transactions on Software Engineering,* vol. 35, no. 6, pp. 756-779, 2009.

[20] “Eclipse Modelling Framework (EMF),” Eclipse Foundation, [Online]. Available: https://eclipse.org/modeling/emf/. [Accessed 8 May 2016].

[21] “Metaobject Facility,” Object Management Group, [Online]. Available: https://www.omg.org/mof/.

[22] Object Management Group, “XML Metadata Interchange,” [Online]. Available: http://www.omg.org/spec/XMI/. [Accessed 31 August 2016].

[23] A. Garmendia, E. Guerra, D. S. Kolovos and J. de Lara, “EMF Splitter: A Structured Approach to EMF Modularity,” *CEUR,* vol. 1239, pp. 22-31, 2014.

[24] “EMF Ecore Javadoc,” Eclipse Software Foundation, [Online]. Available: http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html#details. [Accessed 8 July 2016].

[25] G. Hinkel, “.NET Modelling Framework (NMF) repository,” [Online]. Available: https://github.com/NMFCode/NMF.

[26] “Eclipse Modeling Framework - Interview with Ed Merks,” Jaxenter, 16 April 2010. [Online]. Available: https://jaxenter.com/eclipse-modeling-framework-interview-with-ed-merks-100007.html. [Accessed 8 July 2016].

[27] “Eclipse CDO Model Repository,” The Eclipse Foundation, [Online]. Available: https://projects.eclipse.org/projects/modeling.emf.cdo. [Accessed 21 October 2017].

[28] E. Stepper, “Connected Data Objects presentation,” 18 March 2008. [Online]. Available: https://www.eclipse.org/cdo/documentation/presentations/EclipseCon_2008/CDO-Presentation.pdf. [Accessed 21 October 2017].

[29] “NeoEMF homepage,” AtlanMod, SOM Research, [Online]. Available: https://neoemf.atlanmod.org/. [Accessed 2 January 2020].

[30] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi and D. Launay, “Neo4EMF, a Scalable Persistence Layer for EMF,” in *European conference on Modeling Foundations and*, York, 2014.

[31] “Emfatic,” Eclipse Software Foundation, [Online]. Available: https://www.eclipse.org/emfatic/. [Accessed 20 September 2019].

[32] D. S. Kolovos, L. M. Rose, S. Bin Abid, R. F. Paige, F. A. Polack and G. Botterweck, “Taming EMF and GMF Using Model Transformation,” in *13th International Conference on Model Driven Engineering Languages and Systems*, Oslo, 2010.

[33] “Eclipse Sirius,” Obeo, [Online]. Available: https://www.eclipse.org/sirius/. [Accessed 24 March 2020].

[34] “Human-Usable Textual Notation specification,” Object Management Group, August 2004. [Online]. Available: https://www.omg.org/spec/HUTN.

[35] “Epsilon HUTN compliance,” [Online]. Available: https://www.eclipse.org/epsilon/doc/articles/hutn-compliance/. [Accessed 24 March 2020].

[36] L. M. Rose, R. F. Paige, D. S. Kolovos and F. A. Polack, “Constructing Models with the Human-Usable Textual Notation,” in *11th International Conference on Model Driven Engineering Languages and Systems*, Toulouse, 2008.

[37] “Meta Programming System (MPS),” JetBrains s.r.o., [Online]. Available: https://www.jetbrains.com/mps/. [Accessed 25 March 2020].

[38] “Xtext - LANGUAGE ENGINEERING FOR EVERYONE!,” Eclipse Foundation, [Online]. Available: https://www.eclipse.org/Xtext/. [Accessed 25 March 2020].

[39] “Xtext EMF Integration,” [Online]. Available: https://www.eclipse.org/Xtext/documentation/308_emf_integration.html. [Accessed 11 June 2020].

[40] “XML Metadata Interchange,” Object Management Group, [Online]. Available: http://www.omg.org/spec/XMI/. [Accessed 10 May 2016].

[41] D. S. Kolovos, R. F. Paige and F. A. Polack, “The Epsilon Transformation Language,” in *Theory and Practice of Model Transformations*, vol. 5063, Zurich, Springer Berlin Heidelberg, 2008, pp. 46-60.

[42] F. Jouault, F. Allilaire, J. Bézivin and I. Kurtev, “ATL: A model transformation tool,” *Science of Computer Programming,* vol. 72, no. 1-2, pp. 31-39, June 2008.

[43] “ATL Transformations,” [Online]. Available: https://www.eclipse.org/atl/atlTransformations/. [Accessed 23 October 2017].

[44] R. Heckel, “Graph Transformation in a Nutshell,” *Electronic Notes in Theoretical Computer Science,* vol. 148, no. 1, pp. 187-198, February 2006.

[45] G. Taentzer, K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Golas, D. Varro and S. Varró-Gyapay, “Model transformation by graph transformation: A comparative study,” in *International Workshop on Model Transformations in Practice*, Montego Bay, 2005.

[46] “Object Constraint Language specification,” Object Management Group, [Online]. Available: http://www.omg.org/spec/OCL/.

[47] “QVT specification,” Object Management Group, June 2016. [Online]. Available: https://www.omg.org/spec/QVT/About-QVT.

[48] “Eclipse Model-to-Model Transformation (MMT),” Eclipse Foundation, [Online]. Available: https://projects.eclipse.org/projects/modeling.mmt.

[49] “EMF Henshin,” Eclipse Foundation, [Online]. Available: https://www.eclipse.org/henshin. [Accessed 26 March 2020].

[50] “Eclipse OCL (Object Constraint Language),” The Eclipse Foundation, [Online]. Available: https://projects.eclipse.org/projects/modeling.mdt.ocl. [Accessed 20 October 2017].

[51] L. M. Rose, N. Matragkas, D. S. Kolovos and R. F. Paige, “A feature model for model-to-text transformation languages,” in *4th International Workshop on Modeling in Software Engineering (MISE)*, Zurich, 2012.

[52] “Epsilon Generation Language (EGL) overview,” [Online]. Available: https://www.eclipse.org/epsilon/doc/egl/.

[53] “Xtend language,” Eclipse Foundation, [Online]. Available: https://www.eclipse.org/xtend/documentation/index.html. [Accessed 27 March 2020].

[54] “Eclipse M2T,” Eclipse Software Foundation, [Online]. Available: http://www.eclipse.org/modeling/m2t/?project=xpand. [Accessed 27 July 2016].

[55] “Apache Velocity engine,” Apache Software Foundation, [Online]. Available: https://velocity.apache.org/engine/2.0/overview.html. [Accessed 27 March 2020].

[56] “Apache Freemaker,” Apache Software Foundation, [Online]. Available: https://freemarker.apache.org/. [Accessed 27 March 2020].

[57] “MOFM2T specification,” Object Management Group, January 2008. [Online]. Available: https://www.omg.org/spec/MOFM2T.

[58] “Eclipse Model To Text (M2T),” Eclipse Foundation, [Online]. Available: https://www.eclipse.org/modeling/m2t/. [Accessed 27 March 2020].

[59] “Acceleo,” Eclipse Foundation, [Online]. Available: https://www.eclipse.org/acceleo/overview.html. [Accessed 27 March 2020].

[60] L. M. Rose, R. F. Paige, D. S. Kolovos and F. A. Polack, “The Epsilon Generation Language,” in *Model Driven Architecture – Foundations and Applications*, Berlin, Springer Berlin Heidelberg, 2008, pp. 1-16.

[61] “Epsilon,” Eclipse Software Foundation, [Online]. Available: https://www.eclipse.org/epsilon/. [Accessed 7 July 2016].

[62] D. S. Kolovos and R. F. Paige, “The Epsilon pattern language,” in *9th International Workshop on Modelling in Software Engineering*, Buenos Aires, 2017.

[63] L. M. Rose, R. F. Paige, D. S. Kolovos and F. A. Polack, “The Epsilon Generation Language,” in *Model Driven Architecture – Foundations and Applications*, Berlin, 2008.

[64] D. S. Kolovos, R. F. Paige and F. A. Polack, “The Epsilon Transformation Language,” in *Theory and Practice of Model Transformations*, Zurich, 2008.

[65] D. S. Kolovos, R. F. Paige and F. A. Polack, “Model Comparison: A Foundation for Model Composition and Model Transformation Testing,” in *28th international conference on Software engineering* , Shanghai, 2006.

[66] D. S. Kolovos, R. F. Paige and F. A. Polack, “On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages,” in *Rigorous Methods for Software Construction and Analysis*, Springer Berlin Heidelberg, 2009, pp. 204-218.

[67] D. S. Kolovos, “Establishing Correspondences between Models with the Epsilon Comparison Language,” in *Model Driven Architecture - Foundations and Applications*, Enschede, 2009.

[68] L. M. Rose, D. S. Kolovos, R. F. Paige and F. A. Polack, “Model migration with epsilon flock,” in *Third international conference on Theory and practice of model transformations*, Malaga, 2010.

[69] D. S. Kolovos, R. F. Paige and F. A. Polack, “The Epsilon Object Language (EOL),” in *European Conference on Model Driven Architecture - Foundations and Applications*, Bilbao, 2006.

[70] “Epsilon Object Language,” Eclipse Software Foundation, [Online]. Available: https://www.eclipse.org/epsilon/doc/eol/. [Accessed 7 July 2016].

[71] “Epsilon Documentation,” The Eclipse Foundation, [Online]. Available: http://www.eclipse.org/epsilon/doc/. [Accessed 24 October 2017].

[72] J. Whittle, J. Hutchinson and M. Rouncefield, “The State of Practice in Model-Driven Engineering,” *IEEE Software,* vol. 31, no. 3, pp. 79-85, 2014.

[73] A. Kraus, A. Knapp and N. Koch, “Model-Driven Generation of Web Applications in UWE,” in *Proceedings of the 3rd International Workshop on Model-Driven Web Engineering MDWE 2007*, Como, 2007.

[74] J.-S. Sottet, G. Calvary, J. Coutaz and J.-M. Favre, “A Model-Driven Engineering Approach for the Usability of Plastic User Interfaces,” in *Engineering Interactive Systems*, Salamanca, Springer Berlin Heidelberg, 2008, pp. 140-157.

[75] J. Vanderdonckt, “Model-Driven Engineering of User Interfaces: Promises, Successes, and Failures,” in *5th Annual Romanian Conference on Human-Computer Interaction* , Iasi, 2008.

[76] J. Hutchinson, M. Rouncefield and J. Whittle, “Model-driven engineering practices in industry,” in *Proceedings of the 33rd International Conference on Software Engineering*, Honolulu, 2011.

[77] T. Weigert and F. Weil, “Practical experiences in using model-driven engineering to develop trustworthy computing systems,” in *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, Taichung, 2006.

[78] A. Floch, T. Yuki, C. Guy, S. Derrien, B. Combemale, S. Rajopadhye and R. B. France, “Model-Driven Engineering and Optimizing Compilers: A Bridge Too Far?,” in *Model Driven Engineering Languages and Systems*, Wellington, Springer Berlin Heidelberg, 2011, pp. 608-622.

[79] “Object Constraint Language (OCL),” Object Management Group, [Online]. Available: http://www.omg.org/spec/OCL/index.htm. [Accessed 27 July 2016].

[80] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden and R. Heldal, “Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem?,” in *Model-Driven Engineering Languages and Systems*, Miami, Springer Berlin Heidelberg, 2013, pp. 1-17.

[81] P. Baker, S. Loh and F. Weil, “Model-Driven Engineering in a Large Industrial Context — Motorola Case Study,” in *Model Driven Engineering Languages and Systems*, Montego Bay, Springer Berlin Heidelberg, 2005, pp. 476-491.

[82] G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh and A. Ökrös, “Incremental Evaluation of Model Queries over EMF Models,” in *International Conference on Model Driven Engineering Languages and Systems*, Oslo, 2010.

[83] “About AUTOSAR,” AUTOSAR, [Online]. Available: https://www.autosar.org/about/basics/. [Accessed 23 October 2017].

[84] M. Tisi, S. Martinez and H. Choura, “Parallel Execution of ATL Transformation Rules,” in *Model Driven Engineering Languages and Systems*, Miami, 2013.

[85] H. Brunelière, J. Cabot, G. Dupé and F. Madiot, “MoDisco: A model driven reverse engineering framework,” *Information and Software Technology,* vol. 56, no. 8, pp. 1012-1032, August 2014.

[86] M. Tisi, “Scalability in MDE - AtlanMod,” 19 May 2015. [Online]. Available: http://web.emn.fr/x-info/atlanmod/index.php?title=Scalability_in_MDE. [Accessed 24 October 2017].

[87] “MONDO Project,” [Online]. Available: http://www.mondo-project.org. [Accessed 24 October 2017].

[88] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, S. J. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi and J. Cabot, “A research roadmap towards achieving scalability in model driven engineering,” in *Scalability in Model Driven Engineering*, Budapest, 2013.

[89] A. Bucchiarone, J. Cabot, R. F. Paige and A. Pierantonio, “Grand challenges in model-driven engineering: an analysis of the state of the research,” *Software and Systems Modeling,* vol. 19, pp. 5-13, January 2020.

[90] J.-S. Sottet and F. Jouault, “Program Comprehension,” in *5th International Workshop on Graph-Based Tools*, Zurich, 2009.

[91] I. Dávid, I. Ráth and D. Varró, “Streaming Model Transformations By Complex Event Processing,” in *Model-Driven Engineering Languages and Systems*, Valencia, 2014.

[92] A. B. Bondi, “Characteristics of scalability and their impact on performance,” in *2nd international workshop on Software and performance*, Ottawa, 2000.

[93] “IBM’s Server Processors: The RS64 and the POWER,” 24 January 2011. [Online]. Available: http://www.cpushack.com/2011/01/24/ibms-server-processors-the-rs64-and-the-power/.

[94] “Intel Celeron G470 Prcoessor,” Intel, [Online]. Available: http://ark.intel.com/products/74390/Intel-Celeron-Processor-G470-1_5M-Cache-2_00-GHz.

[95] “Intel Product Specifications,” Intel, [Online]. Available: http://ark.intel.com/.

[96] "AMD Desktop Processors," Advanced Micro Devices, [Online]. Available: http://products.amd.com/en-us/search?k=DesktopProcessors.

[97] "Intel Processors (before December 2008)," Intel, [Online]. Available: http://www.intel.com/pressroom/kits/quickreffam.htm.

[98] J. Hruska, "Intel Core i7-7740K Shatters Overclocking Records, Hits 7.5GHz Thanks to Liquid Helium," 8 June 2017. [Online]. Available: https://www.extremetech.com/computing/250622-intel-core-i7-7740k-shatters-overclocking-records-hits-7-5ghz-thanks-liquid-helium. [Accessed 25 September 2017].

[99] "CPU Frequency: Hall of Fame," HWBOT, [Online]. Available: http://hwbot.org/benchmark/cpu_frequency/halloffame. [Accessed 25 September 2017].

[100] J. Kienberger, P. Minnerup, S. Kuntz and B. Bauer, "Analysis and validation of AUTOSAR models," in *2nd International Conference on Model-Driven Engineering and Software Development*, Lisbon, 2014.

[101] M. McCool, J. Reinders and A. D. Robison, Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufmann, 2012.

[102] "Intel CPU Innovation... or Lack Thereof?," Linus Media Group, 31 December 2016. [Online]. Available: https://youtu.be/jy1M0jkRWWk.

[103] B. Goetz, "From Concurrent to Parallel," Jfokus, 2016. [Online]. Available: https://youtu.be/NsDE7E8sIdQ.

[104] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller and M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal,* vol. 6, no. 1, 14 February 2002.

[105] G. Blake, R. G. Dreslinski and T. Mudge, "A survey of multicore processors," *IEEE Signal Processing Magazine,* vol. 26, no. 6, October 2009.

[106] "Processes and Threads - The Java Tutorials," Oracle, [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html.

[107] "Multithreaded Programming Guide," Sun Microsystems, 2001. [Online]. Available: http://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html.

[108] R. Pike, "Concurrency Is Not Parallelism," [Online]. Available: https://vimeo.com/49718712.

[109] C. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[110] E. G. Coffman, M. Elphick and A. Shoshani, "System Deadlocks," *ACM Computing Surveys,* vol. 3, no. 2, pp. 67-78, 1971.

[111] "Starvation and Livelock - The Java Tutorials," Oracle, [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html.

[112] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Lea and D. Holmes, Java Concurrency in Practice, Addison-Wesley, 2006.

[113] E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," *Communications of the ACM,* vol. 8, no. 9, p. 569, 1965.

[114] J. Bloch, Effective Java: Second Edition, Addison-Wesley, 2008.

[115] "java.util.concurrent.locks.Condition," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Condition.html.

[116] "Java Fork/Join introduction," Oracle, [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html. [Accessed 5 August 2019].

[117] "java.util.Arrays.parallelSort," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#parallelSort-T:A-. [Accessed 1 October 2019].

[118] D. L. Edward, "The problem with threads," *Computer,* vol. 39, no. 5, pp. 33-42, May 2006.

[119] "Project Loom," Oracle, [Online]. Available: https://openjdk.java.net/projects/loom/. [Accessed 3 June 2019].

[120] R. Pressler, "Project Loom presentation," Oracle.

[121] "Comparision of different concurrency models: Actors, CSP, Disruptor and Threads," Blogspot, 28 January 2014. [Online]. Available: http://java-is-the-new-c.blogspot.com/2014/01/comparision-of-different-concurrency.html. [Accessed 7 June 2019].

[122] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM,* vol. 21, no. 8, pp. 666-677, August 1978.

[123] C. Hewitt, P. Bishop and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *3rd international joint conference on Artificial intelligence*, Stanford, 1973.

[124] "Akka homepage," [Online]. Available: http://akka.io. [Accessed 6 June 2019].

[125] "Java Message Service API," Oracle, [Online]. Available: https://docs.oracle.com/javaee/7/api/index.html?javax/jms/package-summary.html}. [Accessed 4 September 2019].

[126] M. Cole, “Why Structured Parallel Programming Matters,” 4 September 2004. [Online]. Available: http://www.di.unipi.it/europar04/ColePisa04.pdf. [Accessed 30 September 2019].

[127] B. Meyer, “Systematic Concurrent Object-Oriented Programming,” *Communications of the ACM,* vol. 36, no. 9, pp. 56-80, September 1993.

[128] P. Nienaltowski, *Practical framework for contract-based concurrent object-oriented programming,* Department of Computer Science, ETH Zurich, 2007.

[129] F. Torshizi, J. S. Ostroff, R. F. Paige and M. Chechik, “The SCOOP Concurrency Model in Java-like Languages,” in *Communicating Process Architectures 2009*, vol. 67, IOS Press, 2009, pp. 7-27.

[130] M. Raynal, “Parallel Computing vs. Distributed Computing: A Great Confusion?,” in *European Conference on Parallel Processing*, 2015.

[131] G. C. Wells, “New and improved: Linda in Java,” *Science of Computer Programming,* vol. 59, no. 1-2, pp. 82-96, January 2006.

[132] J. Dongarra, 13 December 1996. [Online]. Available: http://www.netlib.org/utk/papers/comp-phy7/node3.html.

[133] W. Grop, E. Lusk, N. Doss and A. Skjellum, “A high-performance, portable implementation of the MPI message passing interface standard,” *Parallel Computing,* vol. 22, no. 6, pp. 789-828, September 1996.

[134] “MPI: A Message-Passing Interface Standard Version 3.1,” Message Passing Interface Forum, 4 June 2015. [Online]. Available: http://mpi-forum.org/docs/mpi-3.1/mpi31-report-book.pdf.

[135] J. Dursi, “HPC is dying, and MPI is killing it,” [Online]. Available: https://www.dursi.ca/post/hpc-is-dying-and-mpi-is-killing-it.html.

[136] J. Dean and S. Ghemawhat, “MapReduce: simplified data processing on large clusters,” in *6th conference on Symposium on Opearting Systems Design & Implementation*, San Francisco, 2004.

[137] “Hadoop,” Apache Software Foundation, [Online]. Available: https://hadoop.apache.org/.

[138] “OpenMP ARB - About Us,” OpenMP Architecture Review Boards, [Online]. Available: http://www.openmp.org/about/about-us/.

[139] P. Bělohlávek and A. Steinhauser, “OMP4J - OpenMP for Java,” [Online]. Available: http://www.omp4j.org/.

[140] “OpenMP - Frequently Asked Questions,” OpenMP Architecture Review Boards, [Online]. Available: http://www.openmp.org/about/openmp-faq.

[141] “SequenceL at a glance,” Texas Multicore Technologies, [Online]. Available: https://texasmulticore.com/technology/sequencel-at-a-glance/.

[142] “SequenceL documentations and specification,” Texas Multicore Technologies. [Online]. Available: https://texasmulticore.com/documentation/3.0/0700ref_docs.html.

[143] “Why SequenceL,” Texas Multicore Technologies, [Online]. Available: https://texasmulticore.com/technology/why-sequencel/.

[144] “SequenceL data types,” Texas Multicore Technologies, [Online]. Available: https://texasmulticore.com/documentation/3.0/0712a_data_types.html.

[145] D. Spinellis, “Another Level of Indirection,” in *Beautiful Code: Leading Programmers Explain How They Think*, Sebastopol, California: O'Reilly, 2007, pp. 279 - 291.

[146] “Java 8 Stream,” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html.

[147] Y. Chan, A. Wellings, I. Gray and N. Audsley, “A Distributed Stream Library for Java 8,” *IEEE Transactions on Big Data,* vol. 3, pp. 262-275, September 2017.

[148] “10 Years of AMD Video Cards BENCHMARKED!,” Linus Media Group, 21 March 2017. [Online]. Available: https://www.youtube.com/watch?v=2Fh9Sznf6Rs. [Accessed 29 October 2019].

[149] “10 YEARS of NVIDIA Video Cards Compared!,” Linus Media Group, 22 May 2017. [Online]. Available: https://www.youtube.com/watch?v=q0Kn2hL9ZJQ. [Accessed 29 October 2019].

[150] “Tesla V100 Data Center GPU,” NVIDIA Corporation, [Online]. Available: https://www.nvidia.com/en-us/data-center/tesla-v100/. [Accessed 29 October 2019].

[151] “OpenCL website,” Khronos Group, [Online]. Available: https://www.khronos.org/opencl/. [Accessed 01 November 2019].

[152] “CUDA homepage,” NVIDIA Corporation, [Online]. Available: https://developer.nvidia.com/cuda-zone. [Accessed 1 November 2019].

[153] “CUDA Programming documentation,” NVIDIA Corporation, 19 August 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Accessed 1 November 2019].

[154] B. He, F. Wenbin, Q. Luo, N. K. Govindaraju and T. Wang, “Mars: a MapReduce framework on graphics processors,” in *17th international conference on Parallel architectures and compilation techniques*, Toronto, 2008.

[155] C. Hong, D. Chen, W. Chen, W. Zheng and L. Haibo, “MapCG: writing parallel program portable between CPU and GPU,” in *19th international conference on Parallel architectures and compilation techniques*, Vienna, 2010.

[156] M. K. Elteir, H. Lin, W.-c. Feng and T. R. Scogland, “StreamMR: an optimized MapReduce framework for AMD GPUs,” in *IEEE 17th International Conference on Parallel and Distributed Systems*, Xiamen, 2011.

[157] M. Grossman, M. Breternitz and V. Sarkar, “HadoopCL: MapReduce on Distributed Heterogeneous Platforms through Seamless Integration of Hadoop and OpenCL,” in *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, Cambridge, 2013.

[158] “Java bindings for CUDA,” [Online]. Available: http://jcuda.org/. [Accessed 1 November 2019].

[159] “Java bindings for OpenCL,” [Online]. Available: http://www.jocl.org/. [Accessed 1 November 2019].

[160] “Aparapi,” [Online]. Available: http://aparapi.com/. [Accessed 1 November 2019].

[161] “Project Sumatra,” Oracle, [Online]. Available: http://openjdk.java.net/projects/sumatra/. [Accessed 1 November 2019].

[162] K. Ishizaki, A. Hayashi, G. Koblents and V. Sarkar, “Compiling and Optimizing Java 8 Programs for GPU Execution,” in *24th International Conference on Parallel Architectures and Compilation Techniques*, San Francisco, 2015.

[163] J. Fumero, M. Steuwer and C. Dubach, “A Composable Array Function Interface for Heterogeneous Computing in Java,” in *ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, Edinburgh, 2014.

[164] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Efe, M. Guney and A. Shringarpure, “On the limits of GPU acceleration,” in *2nd USENIX conference on Hot topics in parallelism*, Berkeley, 2010.

[165] M. A. Hammer, K. Y. Phang, M. Hicks and J. S. Foster, “ADAPTON: Composable, DemandDriven Incremental Computation,” in *Programming Language Design and Implementation*, Edinburgh, 2014.

[166] “Samsung NVMe SSD 960 PRO/EVO,” Samsung Electronics, 21 September 2016. [Online]. Available: http://www.samsung.com/semiconductor/insights/news/25661/960pro-960evo. [Accessed 17 April 2017].

[167] V. Subramaniam, “Lazy Evaluations and its Impact on Efficiency,” Warsaw JUG, 20 October 2015. [Online]. Available: https://youtu.be/8srdPlPfFIk.

[168] “OneDrive in Windows 8.1,” Microsoft, 22 July 2013. [Online]. Available: https://blogs.office.com/2013/07/22/have-all-your-skydrive-files-with-you-without-using-all-your-storage-or-bandwidth/. [Accessed 18 April 2017].

[169] S. Noursalehi, “Announcing Git Virtual File System (GVFS),” Microsoft, 3 February 2017. [Online]. Available: https://blogs.msdn.microsoft.com/visualstudioalm/2017/02/03/announcing-gvfs-git-virtual-file-system/. [Accessed 18 April 2017].

[170] F. P. Brooks, “No Silver Bullet - Essence and Accidents of Software Engineering,” *Computer,* vol. 20, no. 4, pp. 10-19, April 1987.

[171] V. Subramaniam, “Functional Programming: Technical Reasons to Adapt,” in *JavaDay*, Kiev, 2015.

[172] R. Hickey, “Values and Change: Clojure’s approach to Identity and State,” [Online]. Available: https://clojure.org/about/state.

[173] G. Hutton, “Lambda Calculus,” [Online]. Available: https://youtu.be/eis11j_iGMs.

[174] A. Vakil, “Learning Functional Programming with JavaScript,” in *JSUnconf*, Hamburg, 2016.

[175] V. Subramaniam, “Get a Taste of Lambdas and Get Addicted to Streams,” in *Devoxx*, Belgium, 2015.

[176] “Haskell homepage,” haskell.org, [Online]. Available: https://www.haskell.org/.

[177] “Referential transparency - Haskell Wiki,” haskell.org, 29 March 2009. [Online]. Available: https://wiki.haskell.org/Referential_transparency.

[178] B. Goetz, M. Reinhold and S. Marks, “Ask the JDK Architects,” Belgium, 2016.

[179] S. Wlaschin, “F# - Expressions vs Statements,” [Online]. Available: https://fsharpforfunandprofit.com/posts/expressions-vs-statements/.

[180] “java.time,” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html.

[181] O. Šelajev, “Functional data structures with Java 8,” in *Devoxx*, Poland, 2016.

[182] R. Gransberger, “Functional Libs for Java 8,” in *Jfokus*, Stockholm, 2017.

[183] D. Schmitz, “Javaslang - Functional Java The Easy Way,” in *Devoxx*, Belgium, 2016.

[184] V. Subramaniam, “The Power and Practicality of Immutability,” Zurich, 2018.

[185] V. Subramaniam, “Let's Get Lazy: Explore the Real Power of Streams,” in *Devoxx*, United States, 2017.

[186] “java.util.function,” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html.

[187] A. Davies, Async in C# 5.0, O'Reilly, 2012.

[188] P. Roberts, “What the heck is the event loop anyway?,” in *JSConf*, Edinburgh, 2014.

[189] “What is the difference between a Future and a Promise?,” StackExchange, [Online]. Available: http://softwareengineering.stackexchange.com/questions/207136/what-is-the-difference-between-a-future-and-a-promise.

[190] “java.util.concurrent.Future,” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html.

[191] T. Nurkiewicz, “CompletableFuture in Java 8, asynchronous processing done right,” in *Joker<?>*, St. Petersburg, 2015.

[192] J. Paumard, “CompletableFuture for Asynchronous Programming in Java 8,” Oracle, 2016. [Online]. Available: https://community.oracle.com/docs/DOC-995305.

[193] “Promises/A+,” Promises/A+ Organization, [Online]. Available: https://promisesaplus.com/.

[194] M. Might, “By example: Continuation-passing style in JavaScript,” [Online]. Available: http://matt.might.net/articles/by-example-continuation-passing-style/.

[195] “watt,” mappum, [Online]. Available: https://github.com/mappum/watt.

[196] “Project Loom,” Oracle, [Online]. Available: https://wiki.openjdk.java.net/display/loom/Main. [Accessed 30 September 2019].

[197] “CompletionStage,” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html.

[198] J. de Jong, “The Future is Completable in Java 8,” 26 September 2015. [Online]. Available: http://www.jesperdj.com/2015/09/26/the-future-is-completable-in-java-8/.

[199] “CompletableFuture,” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html.

[200] J. Paumard, “Asynchronous programming in Java 8: how to use CompletableFuture,” in *Devoxx*, Belgium, 2015.

[201] V. Subramaniam, “Reactive Programming,” in *Devoxx*, Belgium, 2016.

[202] “java.util.Observable,” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/Observable.html. [Accessed 1 October 2019].

[203] “java.util.Observer,” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/Observer.html. [Accessed 1 October 2019].

[204] L. Burgueño, J. Troya, M. Wimmer and A. Vallecillo, “On the Concurrent Execution of Model Transformations with Linda,” in *Scalability in Model Driven Engineering*, Budapest, 2013.

[205] L. Burgueño, “Concurrent and Distributed Model Transformations based on Linda,” in *Model Driven Engineering Languages and Systems*, Miami, 2013.

[206] L. Burgueño, J. Troya, M. Wimmer and A. Vallecillo, “Parallel In-place Model Transformations with LinTra,” in *3rd Workshop on Scalable Model Driven Engineering*, L'Aquila, 2015.

[207] T. Vajk, Z. Dávid, M. Asztalos, G. Mezei and T. Levendovszky, “Runtime model validation with parallel object constraint language,” in *8th International Workshop on Model-Driven Engineering, Verification and Validation*, Wellington, 2011.

[208] C. Clasen, M. Didonet del Fabro and M. Tisi, “Transforming Very Large Models in the Cloud: a Research Roadmap,” in *First International Workshop on Model-Driven Engineering on and for the Cloud*, Copenhagen, 2012.

[209] J. Cabot, “Transforming Very Large Models in the Cloud: a Research Roadmap,” 10 July 2012. [Online]. Available: http://modeling-languages.com/transforming-very-large-models-in-the-cloud-a-research-roadmap/.

[210] G. Mezei, T. Levendovszky, T. Meszaros and I. Madari, “Towards truly parallel model transformations: A distributed pattern matching approach,” in *EUROCON*, St. Petersburg, 2009.

[211] G. Szárnyas, B. Izsó, I. Ráth, D. Harmath, G. Bergmann and D. Varró, “IncQuery-D: A Distributed Incremental Model Query Framework in the Cloud,” in *Model-Driven Engineering Languages and Systems*, Valencia, 2014.

[212] A. Benelallam, A. Gomez, M. Tisi and J. Cabot, “Distributed model-to-model transformation with ATL on MapReduce,” in *ACM SIGPLAN International Conference on Software Language Engineering*, Pittsburgh.

[213] A. Benelallam, M. Tisi, S. J. Cuadrado, J. de Lara and J. Cabot, “Efficient model partitioning for distributed model transformations,” in *ACM SIGPLAN International Conference on Software Language Engineering*, Amsterdam, 2016.

[214] A. Benelallam, “Efficient Model Partitioning for Distributed Model Transformations,” 24 October 2016. [Online]. Available: http://modeling-languages.com/model-partitioning-for-distributed-transformations/.

[215] C. Jeanneret, M. Glinz and B. Baudry, “Estimating Footprints of Model Operations,” in *33rd International Conference on Software Engineering*, Waikiki, 2011.

[216] C. Clasen, F. Jouault and J. Cabot, “VirtualEMF: a Model Virtualization Tool,” in *Workshops of the 30th International Conference on Conceptual Modeling*, Brussels, 2011.

[217] O. Babur, L. Cleophas and M. G. J. van den Brand, “Towards Distributed Model Analytics with Apache Spark,” in *Special Session on Model Management and Analytics (MODELSWARD)*, Funchal, 2018.

[218] “Connected Data Objects,” Eclipse Foundation, [Online]. Available: https://www.eclipse.org/cdo/. [Accessed 20 April 2017].

[219] A. Gómez, A. Benelallam and M. Tisi, “Decentralized Model Persistence for Distributed Computing,” in *3rd Workshop on Scalability in Model Driven Engineering*, L'Aquila, 2015.

[220] K. Barmpis, S. Shah and D. S. Kolovos, “Towards Incremental Updates in Large-Scale Model Indexes,” in *European Conference on Modelling Foundations and Applications*, L`Aquila, 2015.

[221] A. Garcia-Dominiguez, “Hawk project,” [Online]. Available: https://github.com/mondo-project/mondo-hawk. [Accessed 29 May 2019].

[222] J. Cabot and E. Teniente, “Incremental Evaluation of OCL Constraints,” in *Advanced Information Systems Engineering*, Luxembourg, 2006.

[223] F. Jouault and M. Tisi, “Towards Incremental Execution of ATL Transformations,” in *Theory and Practice of Model Transformations*, Malaga, 2010.

[224] A. Razavi and K. Kontogiannis, “Partial evaluation of model transformations,” in *34th International Conference on Software Engineering*, Zurich, 2012.

[225] B. J. Ogunyomi, *Incremental Model-to-Text Transformation,* University of York, 2016.

[226] F. Jouault and O. Beaudoux, “On the Use of Active Operations for Incremental Bidirectional Evaluation of OCL,” in *15th International Workshop on OCL and Textual Modeling*, Ottawa, 2016.

[227] M. Tisi, S. Martínez, F. Jouault and J. Cabot, “Lazy Execution of Model-to-Model Transformations,” in *Model Driven Engineering Languages and Systems*, Wellington, 2011.

[228] M. Tisi, R. Douence and D. Wagelaar, “Lazy Evaluation for OCL,” in *15th International Workshop on OCL and Textual Modeling*, Ottawa, 2015.

[229] E. D. Willink, “Deterministic Lazy Mutable OCL Collections,” in *17th International Workshop in OCL and Textual Modeling*, Marburg, 2017.

[230] D. S. Kolovos, R. Wei and K. Barmpis, “Towards Scalable Querying of Large-Scale Models,” in *2nd Extreme Modeling Workshop, ACM/IEEE 16th International Conference on Model Driven Engineering Languages & Systems*, Miami, 2013.

[231] “Epsilon JDBC Driver,” [Online]. Available: https://github.com/epsilonlabs/emc-jdbc. [Accessed 31 May 2019].

[232] G. Daniel, “Efficient Validation of Large Models using the Mogwaï Tool,” in *18th International Workshop on OCL and Textual Modeling*, Copenhagen, 2018.

[233] G. Bergmann, I. Ráth, G. Varró and D. Varró, “Change-driven model transformations,” *Software & Systems Modeling,* vol. 11, no. 3, p. 431–461, July 2012.

[234] S. M. Pérez, M. Tisi and R. Douence, “Reactive Model Transformation with ATL,” *Science of Computer Programming,* vol. 136, pp. 1-16, August 2015.

[235] R. Wei and D. S. Kolovos, “Automated Analysis, Validation and Suboptimal Code Detection in Model Management Programs,” in *Proceedings of the 2nd BigMDE Workshop*, York, 2014.

[236] R. Wei, D. S. Kolovos, A. García-Domínguez, K. Barmpis and R. F. Paige, “Partial loading of XMI models,” in *19th International Conference on Model Driven Engineering Languages and Systems*, Saint-Malo, 2016.

[237] J. S. Cuadrado, “A verified catalogue of OCL optimisations,” *Software & Systems Modeling,* July 2019.

[238] “Gremlin language,” Apache Software Foundation, [Online]. Available: https://tinkerpop.apache.org/gremlin.html. [Accessed 12 February 2020].

[239] Y. E. Ioannidis, “Query optimization,” *ACM Computing Surveys,* vol. 28, no. 1, pp. 121-123, March 1996.

[240] Microsoft, “Query Processing Architecture Guide - SQL Server,” 24 February 2019. [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/query-processing-architecture-guide?view=sql-server-ver15. [Accessed 13 November 2019].

[241] Oracle, “How Parallel Execution Works (Oracle Database),” [Online]. Available: https://docs.oracle.com/cd/E11882_01/server.112/e25523/parallel002.htm. [Accessed 13 November 2019].

[242] S. Chaudhuri, “An overview of query optimization in relational systems,” in *Seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* , Seattle, 1998.

[243] Microsoft, “Language Integrated Query (LINQ),” 2 February 2019. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/. [Accessed 15 November 2019].

[244] “AMD Ryzen™ Threadripper™ 3990X Processor,” Advanced Micro Devices, [Online]. Available: https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3990x. [Accessed 12 March 2020].

[245] “Epsilon Users,” Eclipse Foundation, [Online]. Available: https://www.eclipse.org/epsilon/users/. [Accessed 12 June 2020].

[246] S. Sen, B. Baudry and J.-M. Mottu, “Automatic Model Generation Strategies for Model Transformation Testing,” in *International Conference on Model Transformation*, Zurich, 2009.

[247] “LinTra Resources,” University of Málaga, [Online]. Available: http://atenea.lcc.uma.es/projects/LinTra.html. [Accessed 26 January 2020].

[248] E. W. Dijkstra, “On The Reliability of Mechanisms,” in *Notes on Structured Programming*, 1970.

[249] E. W. Dijkstra, “Software Engineering Techniques,” 1969.

[250] “EclEmma,” [Online]. Available: http://www.eclemma.org/index.html. [Accessed 18 September 2017].

[251] D. Hawkins, “Java Performance Puzzlers,” in *Devoxx*, Krakow, 2017.

[252] “Java Microbenchmarking Harness,” Oracle, [Online]. Available: http://openjdk.java.net/projects/code-tools/jmh/. [Accessed 24 August 2017].

[253] E. Berger, “Performance Matters,” in *Strange Loop*, St. Louis, 2019.

[254] P. Zemtsov, “Turbo Boost and multi-threading performance,” 8 December 2016. [Online]. Available: http://pzemtsov.github.io/2016/12/08/turbo-boost-and-performance.html. [Accessed 24 August 2017].

[255] P. Alcorn, “Our Tests Show Not All Ryzen 3000 Cores Are Created Equal,” Tom's Hardware, 30 July 2019. [Online]. Available: https://www.tomshardware.com/uk/reviews/amd-ryzen-3000-turbo-boost-frequency-analysis,6253.html. [Accessed 23 January 2020].

[256] “Zen - Microarchitectures - AMD,” [Online]. Available: https://en.wikichip.org/wiki/amd/microarchitectures/zen. [Accessed 23 January 2020].

[257] A. Smith, “YARCC - York Advanced Research Computing Cluster,” University of York, [Online]. Available: https://wiki.york.ac.uk/display/RHPC/YARCC+-+York+Advanced+Research+Computing+Cluster. [Accessed 25 August 2017].

[258] P. Lathan and S. Burke, “Explaining AMD Ryzen Precision Boost Overdrive (PBO), AutoOC, & Benchmarks,” Gamers Nexus, 15 July 2019. [Online]. Available: https://www.gamersnexus.net/guides/3491-explaining-precision-boost-overdrive-benchmarks-auto-oc. [Accessed 23 January 2020].

[259] E. D. Willink, “OCLinEcore Language documentation,” [Online]. Available: https://help.eclipse.org/photon/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FOCLinEcore.html. [Accessed 9 June 2019].

[260] E. D. Willink, “CompleteOCL Language documentation,” [Online]. Available: https://help.eclipse.org/photon/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FCompleteOCL.html. [Accessed 9 June 2019].

[261] J. Bézivin and F. Jouault, “Using ATL for checking models,” *Electronic Notes in Theoretical Computer Science,* vol. 152, no. 1, pp. 69-81, March 2006.

[262] “MoDisco Java Metamodel documentation,” Eclipse Foundation, [Online]. Available: https://help.eclipse.org/photon/index.jsp?topic=%2Forg.eclipse.modisco.java.doc%2Fmediawiki%2Fjava_metamodel%2Fuser.html. [Accessed 10 June 2019].

[263] “ConcurrentLinkedQueue documentation,” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html. [Accessed 17 June 2019].

[264] “ConcurrentHashMap documentation,” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html. [Accessed 17 June 2019].

[265] H. M. Kabutz, “Concurrent Queue Sizes and Hot Fields,” 20 September 2018. [Online]. Available: https://www.javaspecialists.eu/archive/Issue261.html. [Accessed 1 November 2019].

[266] “org.eclipse.epsilon.eol.dom.OperationCallExpression,” [Online]. Available: https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/tree/plugins/org.eclipse.epsilon.eol.

engine/src/org/eclipse/epsilon/eol/dom/OperationCallExpression.java. [Accessed 23 February 2020].

[267] "Java ExecutorService documentation," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html. [Accessed 8 July 2019].

[268] "Java ThreadPoolExecutor documentation," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html. [Accessed July 8 2019].

[269] "Java Runnable documentation," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html. [Accessed 8 July 2019].

[270] "Java Callable documentation," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html. [Accessed 8 July 2019].

[271] "Java FunctionalInterface documentation," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html. [Accessed 8 July 2019].

[272] S. Kuksenko, "JDK 8: Lambda Performance study," Oracle, 2013. [Online]. Available: http://www.oracle.com/technetwork/java/jvmls2013kuksen-2014088.pdf. [Accessed 8 July 2019].

[273] "LinTra Models," Atenea, University of Malaga, 10 November 2015. [Online]. Available: http://atenea.lcc.uma.es/index.php/Main_Page/Resources/LinTra. [Accessed 25 August 2017].

[274] "FindBugs Bug Descriptions," 3 June 2016. [Online]. Available: http://findbugs.sourceforge.net/bugDescriptions.html. [Accessed 26 August 2017].

[275] "EVL test suite," [Online]. Available: https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/tree/tests/org.eclipse.epsilon.evl.engine.test.acceptance/src/org/eclipse/epsilon/evl/engine/test/acceptance. [Accessed 11 March 2020].

[276] S. Madani, "EVL OCL equivalence tests," [Online]. Available: github.com/epsilonlabs/parallel-erl/tree/master/tests/org.eclipse.epsilon.evl.engine.test.ocl.. [Accessed 11 March 2020].

[277] "Apache Spark overview," Apache Software Foundation, [Online]. Available: https://spark.apache.org/docs/latest/. [Accessed 6 September 2019].

[278] "Apache Spark worker intitialization feature request," [Online]. Available: https://issues.apache.org/jira/browse/SPARK-650. [Accessed 6 September 2019].

[279] "Apache Flink," Apache Software Foundation, [Online]. Available: https://flink.apache.org/. [Accessed 6 September 2019].

[280] "Apache ActiveMQ Artemis," Apache Software Foundation, [Online]. Available: https://activemq.apache.org/components/artemis. [Accessed 4 September 2019].

[281] "Simulink," Mathworks, [Online]. Available: https://uk.mathworks.com/products/simulink.html. [Accessed 5 November 2019].

[282] B. A. Sanchez Pina, "Simulink EMC driver experiments repository," Github, [Online]. Available: https://github.com/beatrizsanchez/icse2019-experiments. [Accessed 6 November 2019].

[283] B. A. Sanchez Pina, A. Zolotas, H. Hoyos Rodriguez, D. S. Kolovos and R. F. Paige, "On-the-fly Translation and Execution of OCL-like Queries on Simulink Models," in *ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems*, Munich, 2019.

[284] E. D. Willink, "Deterministic Lazy Mutable OCL Collections," in *17th International Workshop in OCL and Textual Modeling*, Marburg, 2017.

[285] "java.util.concurrent.AtomicBoolean," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicBoolean.html. [Accessed 17 June 2020].

[286] "java.util.concurrent.ExecutorService," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#invokeAll-java.util.Collection-. [Accessed 26 February 2020].

[287] "java.util.concurrent.AtomicInteger documentation," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html. [Accessed 2 August 2019].

[288] V. Subramaniam, "Get a Taste of Lambdas and Get Addicted to Streams," Antwerp, 2015.

[289] "java.util.Spliterator documentation," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html. [Accessed 5 August 2019].

[290] "Java Stream reduce operation," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#reduce-T-java.util.function.BinaryOperator-. [Accessed 6 August 2019].

[291] “java.lang.reflect.Proxy documentation,” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html. [Accessed 6 August 2019].

[292] A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige and I. Medina-Bulo, “EUnit: a unit testing framework for model management tasks,” in *14th international conference on Model driven engineering languages and systems*, Wellington, 2011.

[293] “Atenea Modeling Software Systems,” Universidad de Málaga, [Online]. Available: http://atenea.lcc.uma.es/. [Accessed 26 July 2019].

[294] “Using EGL as a server-side scripting language in Tomcat,” [Online]. Available: https://www.eclipse.org/epsilon/doc/articles/egl-server-side/. [Accessed 16 February 2020].

[295] S. Madani, “EGX example,” Eclipse Foundation, [Online]. Available: https://www.eclipse.org/epsilon/doc/articles/egx-parameters/. [Accessed 23 December 2019].

[296] D. Kolovos, “Visualising Models with Picto,” [Online]. Available: https://www.eclipse.org/epsilon/doc/articles/picto/. [Accessed 25 March 2020].

[297] D. Kolovos, “ecore2dot Picto project,” [Online]. Available: https://git.eclipse.org/c/epsilon/org.eclipse.epsilon.git/tree/examples/org.eclipse.epsilon.examples.picto.ecore. [Accessed 25 March 2020].

[298] H. Hoyos Rodriguez and J. Co, “Incremental EVL,” [Online]. Available: https://github.com/epsilonlabs/incremental-evl. [Accessed 24 December 2019].

[299] A. Garcia-Dominguez, K. Barmpis and D. Kolovos, “Eclipse Hawk,” Eclipse Software Foundation, [Online]. Available: https://projects.eclipse.org/proposals/eclipse-hawk. [Accessed 2 January 2020].

[300] K. Barmpis and D. S. Kolovos, “Hawk: towards a scalable model indexing architecture,” in *Workshop on Scalability in Model Driven Engineering*, Budapest, 2013.

[301] S. Akhmechet, “Functional Programming For The Rest of Us,” 19 June 2006. [Online]. Available: http://www.defmacro.org/ramblings/fp.html.

[302] M. Staron, “Adopting Model Driven Software Development in Industry – A Case Study at Two Companies,” in *Model Driven Engineering Languages and Systems*, Genoa, Springer Berlin Heidelberg, 2006, pp. 57-72.

[303] L. Day, “Programming Paradigms,” August 2013. [Online]. Available: https://youtu.be/sqV3pL5x8PI.