"This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 813884"

Project Number: 813884

Project Acronym: Lowcomote

Project title: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms

Multi-Paradigm Distributed Transformation Engine

Project GA: 813884

Project Acronym: Lowcomote

Project website: https://www.lowcomote.eu/

Project officer: Thomas Vyzikas

Work Package: WP5

Deliverable number: D5.4

Production date: September 30th 2022

Contractual date of delivery: September 30th 2022

Actual date of delivery: October 22th 2022

Dissemination level: Public

Lead beneficiary: Institut Mines-Télécom

Authors: Benedek Horváth, Jolan Philippe, Apurvanand Sahay, Qurat ul ain Ali

Contributors: The Lowcomote partners

Reviewers: Jean-Marie Mottu, Luca Berardinelli

Project Abstract

Low-code development platforms (LCDP) are software development platforms on the Cloud, provided through a Platform-as-a-Service model, which allows users to build completely operational applications by interacting through dynamic graphical user interfaces, visual diagrams and declarative languages. They address the need of non-programmers to develop personalised software and focus on their domain expertise instead of implementation requirements.

Lowcomote will train a generation of experts that will upgrade the current trend of LCDPs to a new paradigm, Low-code Engineering Platforms (LCEPs). LCEPs will be open, allowing to the integration of heterogeneous engineering tools, interoperable, allowing for cross-platform engineering, scalable, supporting very large engineering models and social networks of developers, smart, simplifying the development for citizen developers by machine learning and recommendation techniques. This will be achieved by injecting in LCDPs the theoretical and technical framework defined by recent research in Model Driven Engineering (MDE), augmented with Cloud Computing and Machine Learning techniques. This is possible today thanks to recent breakthroughs in the scalability of MDE performed in the EC FP7 research project MONDO, led by Lowcomote partners.

The 48-month Lowcomote project will train the first European generation of skilled professionals in LCEPs. The 15 future scientists will benefit from an original training and research programme merging competencies and knowledge from 5 highly recognised academic institutions and 9 large and small industries of several domains. Co-supervision from both sectors is a promising process to facilitate the agility of our future professionals between the academic and industrial worlds.

Deliverable Abstract

Low-Code Development Platforms (LCDPs) have emerged as the next-generation Cloud-based development platforms that utilize recent theoretical and practical advancements of Model-Driven Engineering. On these platforms, non-technical users build models of their applications using visual diagrams, domain-specific editors and graphical workflows and can automatically generate source code to realize them as fully operational applications. Therefore, LCDPs help in speeding up the development process and shortening the time-to-market and time-to-product cycles.

Since LCDPs are cloud-based platforms deployed in a Platform-as-a-Service (PaaS) model, they have specific needs. They need to complete complex operations with low response time to satisfy users' needs with efficiency. Although there exist several technologies which follow a single execution strategy for model-management operations, there is no technology that would automatically choose the most efficient one, even by the combination of several others, for a given goal. Besides, to achieve responsive low-code platforms, we need scalable reactive model transformations that can react quickly to events which occur on the platform. Moreover, in LCDPs, users can create complex model-management workflows that should be executed in the most efficient way possible while maintaining some properties and constraints. Specifications of complex workflows within and across multiple platforms are elaborated in understanding the process builder mechanism in an LCDP. This raises a broad research goal about using customized modelling constructs so that a citizen developer can use the constructs to specify the desired workflows at a high level of abstraction.

In this deliverable, we propose ways to address the aforementioned challenges. We present a distributed model transformation engine on Spark, a scalable framework for developing distributed applications. We provide an analysis of several solutions for evaluating OCL expressions, conducting a distributed model transformation and implementing a transformation engine. After that, we propose a benchmark framework to find the best parameterization of multi-parameter Spark applications according to a goal metrics function. To improve the quality of industrial systems engineering models, we propose a cloud-based model transformation, validation and verification framework that takes benefit of the elastic scalability of cloud resources to enhance the performance of resource-intensive formal verification tasks. Finally, we propose a workflow for efficiently executing model transformation composition scenarios across several platforms. The identification of model transformation chains is made by analyzing the transformation language in the (meta)model repository. An optimal chain is selected based on different objectives, such as coverage and complexity. Lastly, an optimal transformation chain is deduced by finding out the exact elements that are needed to be transformed across the chain.

Contents

1 Introduction

Model-Driven Engineering (MDE) [1] provides a method to develop software using domain models at a higher level of abstraction. The Object Management Group [2] proposed the Model Driven Architecture (MDA), which is part of MDE. The MDA-based software starts by building platform-independent models, which are transformed into one or more platform-specific models, which can further be transformed to code for specific software. Model transformation is key in determining the interoperability [3] with other software and the reusability [4] of the artefacts within or outside a particular software scenario.

Model transformation creates new target models from source models, which conform to target and source metamodels. During this process, a trace model is created that tracks which target model elements were created from which source model elements. Thereby enabling the incremental update of the target model according to the changes in the source model or back-propagating the target model to the source model if we want to trace the analysis results in the target model to their origins in the source model. Researchers have implemented several model transformation engines in the past (e. g., ETL [5], ATL [6], Viatra [7]), each implementing a different approach to achieve this goal.

In recent years, Low-Code Development Platforms (LCDP) have emerged, allowing citizen developers with no or little prior programming experience to design and implement full-fledged applications. LCDPs adopt the recent theoretical and practical advancements of Model-Driven Engineering. On these platforms, citizen developers [8] build software models by refining their operation on diagrams with different levels of abstraction, using domain-specific editors. Moreover, they use model transformations to derive platform-specific source code, tests, or configuration artefacts from the models to realize their systems as fully operational applications [9].

LCDPs allow citizen developers to design and implement complex systems that can process a large amount of data. Let's consider an application which processes content from a social media platform, where millions of comments, videos and photos are posted every day. To process and transform such a large amount of content efficiently, we need many computational resources (CPU, memory, disk, network) and a scalable, distributed platform.

Cloud computing provides easy access to a large number of computational resources. Increasing the number of available resources enables elastic, on-demand, horizontal and vertical scalability, i. e., the application remains scalable if the number of dedicated resources, or the size of input data, increases. In other words, the performance will not be impacted if the number of computational units or the size of the input data increases. The only challenge then is to build a scalable program. MapReduce, a data-distributed-based framework designed for big data processing, offers a highly scalable and parallel solution. Also, other frameworks, such as Spark [10] are very popular in the cloud [11].

To push the boundaries of state-of-the-art (SOTA) research further and to take benefit of such software platforms as scalable solutions for model management operations, we propose SparkTE, a multi-paradigm distributed transformation engine (Section 2) with a benchmark framework (Section 3) to find the best parametrization of multi-parameter applications similar to SparkTE. Besides multi-paradigm model transformations, the cloud can also be used to improve the quality of software and system models. Therefore, in Section 4, we introduce a cloud-based model transformation, validation and verification workflow to check the correctness of industrial systems engineering models.

In a complex data-processing and model manipulations pipeline usually, many transformations are chained after each other. However, these transformation chains are often more complex than they should be, i. e., contain model transformations that are unnecessary from the final result's perspective. Therefore, in order to help LCDP users' work and improve the SOTA research, in Section 5, we introduce novel solutions for model transformation composition, chain identification, selection and optimization.

Finally, we conclude the report and present future work in Section 6.

Contributions

In this report, we present the following contributions:

1. SparkTE, a multi-parameter model transformation engine on Spark, proposes a comparison of strategies for evaluating expressions and conducting model transformations based on different semantics,
2. a framework to benchmark multi-parameter applications on a Spark cluster,
3. a scalable, cloud-based model transformation, validation and verification workflow for industrial systems engineering models,
4. approaches to create a model transformation composition engine in the Epsilon Transformation Language (ETL). The approaches are related to identifying and selecting the appropriate chain based on multiple objective criteria. Also, optimization of the transformation chain execution is done by discarding those elements in the intermediate transformation which are not used in the target model.

2 Cloud-based Scalable Model Management Operations

In this section, we propose SparkTE, a model transformation engine based on Spark[1], proposing different implementations at different levels of the engine. Proposing a unified solution nesting several implementations and strategies allows experiments to compare different semantics and strategies of execution. It also includes an attempt to compare engineering choices that are part of the design space of model transformation engine definitions. Most of these works have been published in [12] and [13].

2.1 Research objectives

A model transformation (MT) is defined as a set of rules and is executed in an engine. The left-hand side of the rule (LHS) is an expression describing what part of the model must be matched, while the right-hand side (RHS) expresses its related output. Evaluating these expressions results in running queries on the input model. The LHS is an expression evaluated into a boolean value, while the RHS constructs output elements from the matched elements and the input model. In a larger scope, rules are executed in an engine. Each research effort in developing a model management engine exploits a single strategy for optimizing model management operations [12]. Typically, the strategy is applied in an additional implementation layer for the model management language, e. g., an interpreter or compiler.

We say that a transformation engine performs *multi-strategy model management* if it automatically considers different strategies in order to manipulate models efficiently. To the best of our knowledge, such an approach does not exist in the literature yet.

In this section, we exemplify the multi-strategy approach by implementing an OCL expression in different ways, using different strategies of parallelism. Secondly, we propose a full model transformation engine based on formal semantics: one is designed for reasoning, while the other increases parallelism opportunities. Finally, we discuss engineering decisions to implement such an engine.

Our prototype is built on top of Spark, an engine designed for big data processing in the Cloud. The goal of this section is not to provide the most efficient solution for solving the given problem. Instead, it aims at illustrating the diversity of solutions, each having its own advantages depending on the use cases.

2.2 Motivating example

Social network vendors often provide specific development platforms used by developers to build apps that extend the functionality of the social network. Some networks are associated with marketplaces where developers can publish such apps, and end-users can buy them. Development platforms typically include APIs that allow analyzing and updating the social network graph.

As a running example for this section, we consider a scenario where a vendor adds a Low-Code Development Platform (LCDP) to allow end-users (also called *citizen developers* in the LCDP jargon) to implement their own apps. Such LCDP could include a WYSIWYG editor for the app user interface and a visual workflow for the behavioural part. In particular, the LCDPs would need to provide mechanisms, at the highest possible level of abstraction, to define expressions to update the social graph.

In Figure 1, we show the simple metamodel for the social graph that we will use in the chapter. The metamodel has been originally proposed at the Transformation Tool Contest (TTC) 2018 [14] and used to express benchmarks for model query and transformation tools. In this metamodel, two main entities belong to a `SocialNetwork`. First, the `Post`s and the `Comment`s that represent the `Submission`s, and second, the `User`s. Each `Comment` is written by a `User`, and is necessarily attached to a `Submission` (either a `Post` or another `Comment`). Besides commenting, the `User`s can also like `Submission`s.

The TTC case states that the impact of a post depends on how much it is debated. As evaluation, a `score` function is applied to each `Post`. The execution of this scoring function represents our use case for evaluating the different implementations of a given expression. To evaluate the different semantics for executing a model transformation, we use a simple identity transformation on the model.

2.3 Architecture and approach

2.3.1 Expression evaluation on cloud architectures

In addition to parallel features of Spark on data structures, called Resilient Distributed Datasets (RDDs), the Scala implementation of Spark proposes several APIs, including a MapReduce-style one, an API for manipulating graphs (GraphX [15] that embeds the possibility to define Pregel programs [16]), and a SQL interface to query data structures. Because the framework proposes different parallel execution strategies, we only focused on parallel approaches to illustrate the need for a multi-strategy approach. Comparing solutions that include laziness and incrementality aspects is part of our future work. In our implementation example, we use GraphX in addition to its provided Pregel function and MapReduce features. We represent instances of `SocialNetwork` as a GraphX graph where each vertex is a couple of a unique identifier

[1] https://spark.apache.org/

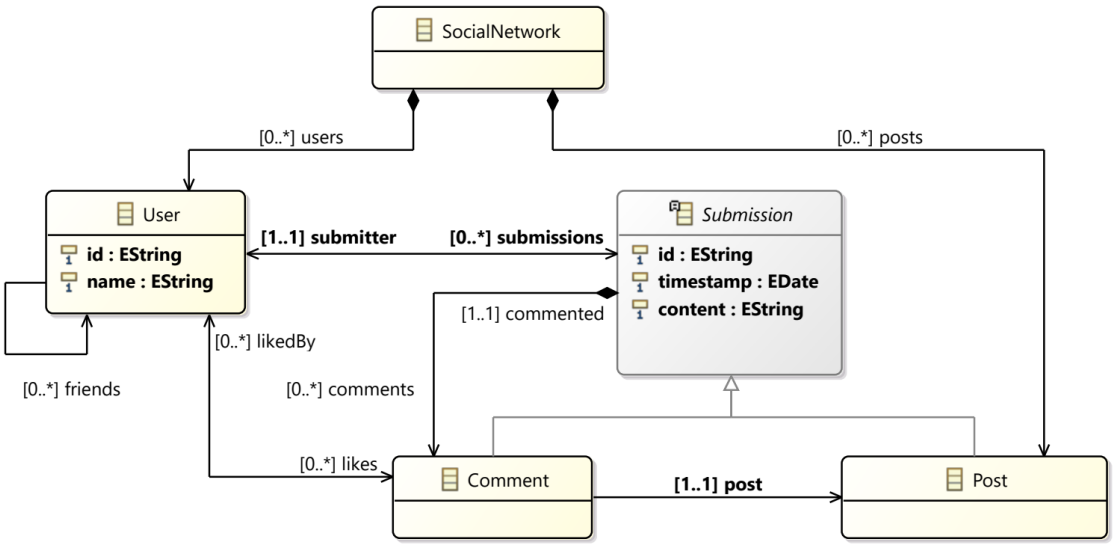


Figure 1: The metamodel of a social network (TTC 2018).

and an instance of either a `User` or a `Submission` (`Comment` or `Post`). Edges represent the links of elements of a model conforming the metamodel presented in Figure 1, labeled by a `String` name. We keep exactly the same labels from the metamodel for [0..1] or [1..1] relations, but we use singular names for [0..*] relations (e. g., one edge "like" for each element of the "likes" relationship). For the rest of this section, we consider `sn` a GraphX representation of a `SocialNetwork`.

Considering that there exists an implementation for the function `score`, that will be detailed later in this section, the OCL query `topPosts` of Listing 1 can be rewritten using Spark, as presented in Listing 2.

```
query topPosts = SN!Post.allInstances()
                         →sortedBy(e | -e.score)
                         →subSequence(1, 3);

helper context SN!Submission def: allComments =
   self.comments→union(self.comments
                            →collect(e | e.allComments)
                            →flatten() );

helper context SN!Post def: countLikes =
   self.allComments
       →collect(e| e.likedBy.size())
       →sum();

helper context SN!Post def : score =
   10*self.allComments→size() + self.countLikes;
```

Listing 1: An OCL query for the first task of the TTC 2018.

```
1 sn.vertices.filter(v => v.isInstanceOf[Post])
2    .sortBy(score(_._2), ascending=false)
3    .collect.take(3)
```

Listing 2: Spark implementation of a query from TTC 2018.

First, the `SN!Post.allInstances()` statement of the OCL specification is translated into the application of a filtering function on the vertices of the graph `sn` (line 1). Sorting with a decreasing order is then applied to the score values (computed by the `score` function) of each vertex. The projection `_._2` returns the second element of the vertex values, that is an instance of `Post`, while `_._1` would have returned its identifier within the graph. At the end of line 2, the current structure is still an RDD. Because of the small number of values we aim at finally obtaining, the structure is converted into a sequential array of values (function `collect`), from which we get the first three values. We can notice a similar structure between the Spark and OCL queries. Hence, the global query can almost be directly translated from one language to the other.

However, the scoring function can be implemented in many different ways with many different strategies. We illustrate this through three implementations: *direct-naive*, and *highly-parallel*, *pregel*.

The first implementation, namely *direct-naive* directly follows the OCL helpers from Listing 1. It uses parallelism coupled with the lazy evaluation provided by Spark. Each helper is written using Spark-equivalent functions.

The second implementation with MapReduce proposes a solution with a higher level of parallelism, namely *highly-parallel*. The purpose of this solution is to process as many operations in parallel for the first time as possible and then go through the graph to reduce these values. The first step counts the number of direct sub-comments and the number of likes for each element of the model using a map operation. Then a graph-traversal operation calculates the total number of belonging comments and likes for a given post. We do not expect to gain performance with this approach because the operations are not costly enough. However, having a highly parallel approach largely increases the scalability of the program. One disadvantage of this implementation is its non-reactive aspect. Indeed, without an additional mechanism, all the initial map functions must be re-executed in case of a change in the model.

The third implementation, namely *pregel*, is a Pregel-based one. The main idea of this solution is, starting from a `Post`, to count the number of comments and the number of likes for these comments by propagating messages through the edges of the graph by using Pregel. The propagation is processed using the Pregel support of GraphX, which works as follows. At each iteration, a function *mergeMsg* accumulates the incoming messages into a single value. The messages are stored in an iterable structure from the previous iteration (with an initial message defined for the first iteration). The accumulated value is used by *vprog* with the previous vertex v_n to generate the new vertex data v_{n+1}. In addition, to this new vertex data, messages are generated by *sendMsg* and sent to vertices through edges for the next iteration. The program stops when no message is produced for the next iteration. In our implementation, messages are tuples of two values. The first is a boolean, specifying if the sending vertex has been reached during the *pregel* execution. The second one aims at making more precise what value must be incremented (either the number of comments (`false`), or likes (`true`)). The initial step of the execution initializes the graph creating a tuple from the vertex values with a boolean specifying if the vertex has been reached. At the end of the execution, the score of a post is calculated using the accumulator values.

Comparison of solutions First, the complexity of the solutions *direct-naive* and *pregel* can be compared. On the one hand, the complexity in time of the direct implementation of the OCL query, can be given as the sum of the complexity of `allComments` and `countLikes`. Considering n the number of nodes, these two complexities are defined as follows. First, `allComments` is a depth-first search of complexity $O(n + m)$ with m the number of `"comment"` edges (i. e., the depth of belonging comments). Second, `countLikes` is composed by a depth-first search, and the `map` of a function whose complexity is $O(n)$. Then, the complexity of the mapping part is given by $O(n^2)$. Since the complexity of the `sum` operation is negligible, we do not consider it in the calculation of the global complexity. By summing these values, we obtain a complexity of $O(n^2 + m)$ for the direct implementation of the scoring function. On the other hand, the Pregel implementation complexity is bounded by $O(n^2)$ in the case of all comments belonging to the same post. Naturally, the second solution will be preferred since its complexity is lower. However, if the model has a small depth of belonging comments (i. e., a small value for m), the two solutions are not significantly different.

The Pregel solution has, nonetheless, an important weakness. Indeed, for optimization reasons, *vprog* is only applied to vertices that have received messages from the previous step. Then, considering the case where the comments are all commented once, the *vprog* function will be applied to only one vertex. Hence, the parallelism level strongly depends on the number of siblings in each comment. With Pregel, only active vertices, i. e., vertices which received a message from the previous iteration, compute the *vprog* function. Thus, the number of operations concurrently executed in Pregel varies from the less to the most commented and liked element. On the contrary, the highly parallel implementation executes the processing operations on every element of the model. In the latter, the parallelism level of graph-traversal has the same limitation as the Pregel implementation but always performs a less complex operation (i. e., a reduction as a sum of integer values).

The three parallel approaches mentioned above solve the same problem, but their efficiency depends on external parameters. We have illustrated how the topology of the model may impact the complexity, but also how the level of parallelism may become important, and what impact the cost of parallel operations may have. Thus, choosing the right paradigms can have a huge impact on performance.

Mix of solutions As mentioned at the beginning of the section, our proposed solutions do not claim to be the most efficient ones. They are based on three parallelism strategies to illustrate the variability of possible solutions for a given problem. Considering all the presented strategies, a more robust solution could include reactive aspects. For this particular example, mixing incrementality and parallelism would avoid useless calculations when the score of a single post has changed. For instance, the independent scores could be calculated once using parallelism, and when a change occurs, use incrementality to avoid

the recomputation of unchanged elements. Considering a possible deletion of a part of the model (e. g., deletion of a user and then of all his posts and comments), laziness could be incorporated into the solution only to recompute potential new most-debated posts.

Finally, the first two strategies (i. e., direct-naive and highly parallel) adopted a depth-first search strategy for their traversal functions. The functions are implemented as recursive functions that use functional patterns implemented in parallel. Nonetheless, it is possible to use a higher level of implementation for these functions. For instance, the iterative aspect of Pregel totally fits in this case. The different implementations can be found in a remote directory[2].

In the rest of this section, we also conducted experiments on these multi-strategy implementations. To execute the `topPosts` query, a multi-strategy engine would compile it to:

- the direct-naive implementation, if the depth of belonging comments is small;
- the highly-parallel solution, if the score computation needs big calculation on the vertices themselves;
- the Pregel solution, if the environment has few resources for parallelism;
- the mix of direct-naive, or the highly-parallel solution, with Pregel features, if several conditions are respected.

2.3.2 Model transformation semantics

In MDE, model management frameworks propose dedicated languages to transform models, like the AtlanMod Transformation Language [17] (ATL) or the Epsilon Transformation Language [18] (ETL). Good scalability and facilities for formal reasoning are among the intended benefits for users of model-transformation languages. On the one hand, researchers have proposed transformation engines designed to effectively perform computationally or memory-intensive transformations [19]. On the other hand, the community has extensively worked on formal reasoning and verification tools for model transformation languages. Among these solutions, the CoqTL language [20] allows users to write transformation rules, define contracts and certify the transformation against them within the Coq proof assistant [21].

In this section, we introduce SparkTE, a transformation engine that addresses at the same time distribution and certification. To do so, we propose a refinement of the CoqTL semantics, named Parallelizable CoqTL, increasing parallelism and distribution opportunities. But Coq specifications are not designed to be executed. Then we also propose an implementation on top of Spark. We illustrate the increased performances of our new semantics in Section 2.4.2.

2.3.2.1 CoqTL

The CoqTL language [20] allows users to write transformation rules, define contracts and certify the transformation against them within the Coq proof assistant [21]. A transformation is written in CoqTL, an internal DSL for model transformation within the Coq theorem prover. The transformation primitives are newly-defined keywords (by the notation definition mechanism of Coq), while all expressions are written in Gallina, the functional language used in Coq. The CoqTL semantics is heavily influenced by ATL [22] notably in the distinction between a match/instantiate, and an apply function), and its original design choices focus on simplifying proof development.

The `execute` function shown in Listing 3 is the entry point of the transformation execution in the standard CoqTL semantics. First, the `allTuples` function (line 2) produces all the tuples of elements that can possibly be matched by the rules. `allTuples` computes a list of $\sum_{a=0}^{A} n^a$ tuples, with n the number of elements in the input model, and A the maximum arity of the transformation rules. Then, the `instantiatePattern` function (line 3) considers each tuple to find if it matches with any rule, and for each match, it constructs the corresponding output pattern elements. Internally it iterates on each rule, executes its guard function and if the result is positive, executes the element creation function for each output pattern element of that rule. The resulting elements are gathered by the `flat_map` in a single output list. Finally, the `applyPattern` (line 4) function is executed on each tuple to create target links. Similarly to the function `instantiatePattern`, the function internally iterates on all rules and checks if the rule matches the given pattern. In a positive case, the element creation functions for that rule are executed and then the link creation functions. The resulting links are gathered by the `flat_map` in a single output list.

```
1 Definition execute (tr: Transformation) (sm: SourceModel): TargetModel :=
2     let tuples := allTuples tr sm in
3     let elements := flat_map (instantiatePattern tr sm) tuples in
4     let links := flat_map (applyPattern tr sm) tuples in
5     Build_Model elements links.
```

Listing 3: Execution definition of MT in CoqTL.

[2] https://tinyurl.com/yxb86zev

```
1 Definition execute (tr: Transformation) (sm: SourceModel) (sm: SourceMetamodel): TargetModel :=
2     let tuples := allTuplesByRule tr sm mm in
3     let (elements, tls) := flat_map (tracePattern tr sm mm) tuples in
4     let links := flat_map (fun sp ⇒ applyPatternTraces tr sm sp tls) (allSourcesPattern tls) in
5     Build_Model elements links.
```

Listing 4: `execute` function in the Parallelizable CoqTL specification.

2.3.2.2 A new semantic: Parallelizable CoqTL

Parallelizable CoqTL contains three optimizations to the base CoqTL specification:

- to increase parallelization, the algorithm is split into two consecutive phases, *instantiate* and *apply*, that are built on parallelizable functional patterns (`flat_map`);
- to improve load balancing of the instantiate phase, only possibly useful tuples are generated and then distributed;
- to improve load balancing of the application phase, a set of trace links is produced by the instantiate phase and the apply phase iterates only on those trace links.

Note that similar optimizations (among others) are already implemented in well-known transformation engines, like ATL [22] or ETL [23]. Differently from previous work, we formalize the optimizations, interactively prove that they do not affect the transformation output and assess their impact on distributed execution. The entry point of Parallelizable CoqTL is presented in Listing 4.

Optimization 1: Two phases In standard CoqTL, the `applyPattern` function performs all the computations of the links generated by a matched input pattern by the rule that matches it. However, the computation of the links of a rule is not independent of the computation of other rules. This dependency is caused by the `resolve` function that searches for the output of another rule in order to set the target of the created link. In general, because of this dependency, two executions of the apply function can not be run in parallel without replicating some matching and instantiation within each call to `resolve`.

We refactor the computation to split it into two phases, similarly to ATL [24]. This is visible in Listing 4. In the first phase (lines 3), we compute the tuples, and we run the matching and instantiation by a new function named `tracePattern`. The first phase produces the list of generated elements and trace links connecting them to their corresponding source patterns. Differently from Listing 3, here the second phase (line 4) can only start computing output links after the full first phase has finished computing the trace links since the `flat_map` expects the `tls` structure as a parameter.

In Listing 4, every execution of `tracePattern` can be run in parallel. When the first phase is over, every execution of `applyPatternTraces` can be run in parallel, too, since the calls to resolve can be computed immediately on the trace-link structure. This greatly improves the parallelization of the algorithm.

Optimization 2: Tuple generation by rule Matching a pattern to a rule happens in two consecutive steps. First, the types of the pattern are checked against the types expected by the rule. Then, if the types are correct, a guard condition is evaluated. The type checking is very fast; hence it acts as the first filtering. Instead, the evaluation of the guard condition can potentially be very long or navigate large parts of the model. So it is executed only for the few tuples that pass the type check.

Since the tuples that require an evaluation of the guard condition are a small subset of all the possible tuples, arbitrarily distributing all tuples among the cores can potentially lead to imbalanced partitions. In particular, it would not be uncommon to have partitions that do not require any guard evaluation, as opposed to partitions that need to evaluate several expensive guards. In such cases, idle workers would wait for the synchronization barrier to start new computations. The imbalance impacts the scalability of the program.

To limit the imbalance in the initial sequential tuple generation phase, we generate only tuples whose type matches at least one rule of the transformation. This is shown in Listing 4 by the use of the `allTuplesByRule` for tuple generation (line 2). `allTuplesByRule` iterates on rules and produces only combinations of elements of the types listed in the rule input pattern. This improves the load balancing of the first phase since all produced tuples require a guard evaluation.

Optimization 3: Apply iterates on traces Executing the apply phase on the tuples generated by `allTuplesByRule` would cause an imbalance among partitions, similar to the one discussed in Optimization 2. Indeed, among these tuples, only very few have passed the guard condition in the first step. A partitioning of `allTuplesByRule` would produce partitions that do not require any computation, together with partitions that need to evaluate several expensive link creation functions. In this optimization, we make the apply phase iterate only over the source patterns that passed the guard evaluation in the instantiate phase. We retrieve these patterns by collecting them from the list of trace links. This is performed by the function `allSourcePatterns` at line 4 of Listing 4.

Optimization	Specification size (LOC)	Certificate size (LOC)	Proof effort (man-days)
twoPhases	69	484	10
byRule	42	487	7
iterateTraces	69	520	4

Table 1: Sizes of new specifications and certification proofs for each optimization with proof effort.

Proof Besides the executable functional specification, CoqTL is also described by an axiomatic specification. Certifying against the axiomatic specification involves providing 10 types, 27 semantics functions and proving 15 theorems. The specification is fully illustrated in [13], together with a proof engine that can execute the certification against the axiomatic one.

We prove that engines implementing Parallelizable CoqTL certify the axiomatic specification, too. For this step, we naturally reuse the types, functions and certification proofs of the base executable specification that are not impacted by the optimization. Each optimization is proved independently. Table 1 shows the size (in lines of code, LOC) of new specifications and proofs required for describing and certifying each optimization, plus the human effort (in man-days) to complete the proofs. The refined specifications and their proofs are available online[3].

2.3.2.3 SparkTE

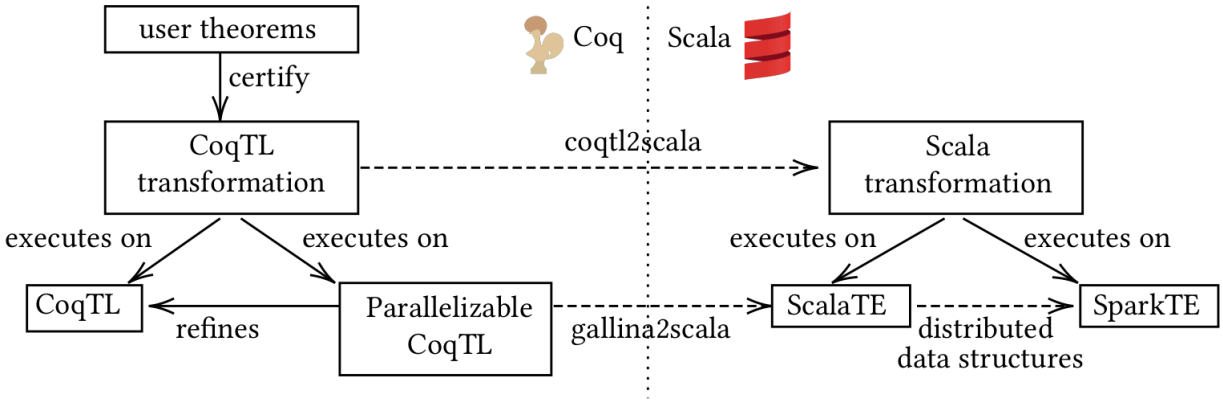


Figure 2: Global overview of our workflow to execute certified model transformations on Apache Spark.

Coq to Scala The Coq environment includes an extraction mechanism targeting functional languages: OCaml, Haskell, or Scheme. Although an automatic extractor to Scala is available [25], it supports only a subset of Gallina on an outdated version of Coq. Hence, we opted to perform the extraction manually. We perform manual extraction on two levels: first, to create the core engine (*gallina2scala* in Figure 2), then to obtain Scala rules representing a CoqTL transformation (*coqtl2scala*).

- **Gallina to Scala**. The executable CoqTL specification can be seen as a functional program that interprets the transformation code. We produce a literal translation of this interpreter in Scala. For extracting Scala code from Parallelizable CoqTL, we translate Gallina types and (pure) functions into their corresponding types and pure functions in Scala.
- **CoqTL to Scala**. The CoqTL parser translates the concrete syntax of the transformation into Coq code to construct an abstract syntax tree. Obtaining the same transformation in Scala requires constructing the same abstract syntax tree as Scala objects. Note that Scala constructors for the abstract syntax are the literal translation of the corresponding Gallina constructors in CoqTL.

Distributed Data Structures Spark RDDs are data structures that are automatically partitioned, resulting in the distribution of the computation operations on a Scala sequence of serializable elements. From a user point of view, RDDs can be manipulated as lists using the same primitive functions, and parallelism is implicit. The advantage of using such abstraction for parallelism is the semantics preservation of the operations on the distributed structures. Because of the popularity of Spark and its support, we assume the correctness of parallel operations on the data. The efficient use of RDDs requires an effective partitioning

[3] https://github.com/atlanmod/SparkTE_public/

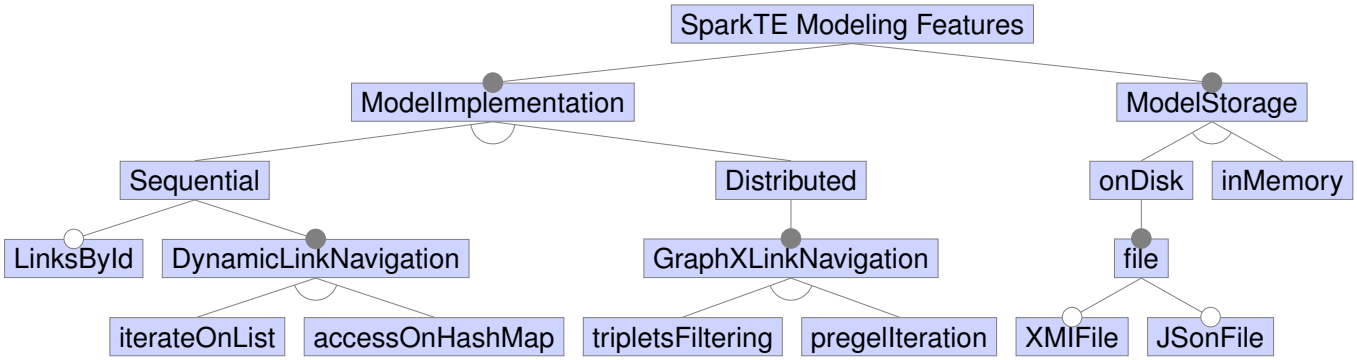


Figure 3: SparkTE feature diagram for its modelling solution.

of data. For instance, to take advantage of the internal multi-threading mechanism, it is typically recommended in Spark to assign four data partitions to each core. Each independent computation of a partition is referred to as a task. The Spark task scheduler makes the distribution following a round-robin approach, optimizing load-balancing: once a task is ended on a core, a new one can be assigned from the waiting list.

2.3.3 Feature diagrams for Parametrizable SparkTE

Next to a correct implementation of Parallelizable CoqTL in SparkTE, using functional structures (e. g., *List*) and pure functional features, we introduced additional solutions in SparkTE. Firstly, the representation of the model and its navigation present several approaches based on different data structures and different approaches for representing links. Secondly, each part of the execution might follow a different strategy. As illustrated previously in the new specification of CoqTL, the instantiation of elements and links might depend on the purpose of having a solution for reasoning or a solution for increasing parallelism opportunities. Finally, since SparkTE is based on Spark, some features are Spark-related. We will briefly discuss the latest later in the section.

Modeling approaches In SparkTE, models follow a very generic specification stating that models must only implement two functions: one to get elements and one to get links. The concrete implementation is a couple of two sets: one for the elements and one for the links. Figure 3 gives an overview of the features described below. To tackle memory issues that can be faced when dealing with very large models (VLM), it is possible to distribute these two sets using RDDs and create a graph by GraphX. Once the model is distributed, it can be transparently queried by the user, using expressions as explained in Section 2.3.1.

On the one hand, the sequential solution offers two possible ways to navigate among the links of the model. First, links can be reached by iteration on the full set. This operation only shows benefits on a very small model since the browse of data is made instantly. A second approach is to store links in a *HashMap*, using elements and types of links as keys. Accesses are direct, but creating such a structure requires additional operations with a CPU cost. Also, in Scala, *HashMap* keys are managed in a non-linear structure (a tree). It aims at improving the speed of accessing data but requires an additional amount of memory. Finally, the links can either be represented as a tuple of elements with a label or only using their IDs. The latest stores less information and does not duplicate the objects during the distribution, only their IDs, but it leads to the need of resolution when users want to access the content of the objects from a link.

On the other hand, distributed models take advantage of the distributed graph structure and allow users to use several computational models for link navigation. It can either be using Pregel iteration, with the propagation of messages to get only a subset of links, or by filtering the distributed set of edges represented as *triplets* composed by a source, a label, and a target. Note that in the distributed case, links are represented by edges and are always built from the IDs of elements. However, their representation as triplets in Spark gives direct access to both the element ID and the element itself.

Finally, the model can be stored at different levels on the machine. In a sequential implementation of the model, i. e., when the model is fully duplicated on each computational node, the storage must be explicitly handled. SparkTE supports two modes: the model can be fully loaded in memory or kept on disk using XMI or JSON files. Distributed data with Spark can be natively stored at different levels by specifying a *StorageLevel*. This approach is discussed later. In future work, we consider storing the model in a remote database to limit the amount of data loaded in memory, but this approach would increase computation time because of the necessary communications.

Execution strategies Section 2.3.2.2 has shown there exist different semantics for performing a model transformation in SparkTE. In the implementation of Parallelizable CoqTL, we put a lot of importance on correctness. In the following features, we do not consider correctness aspects but only focus on performances. Figure 4 gives an overview of the features described below.

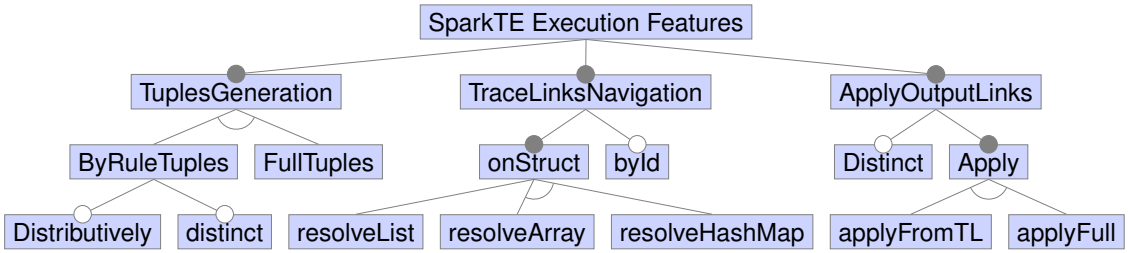


Figure 4: SparkTE feature diagram for its execution strategy.

The first feature, the tuples generation, can follow different strategies. As explained before, generating all the tuples is not always necessary. However, it tackles an imbalance in the distribution of data since all tuples have the same weight. In the other case, generating the tuples from the types of rules reduces the total number of tuples. Note that if several rules have the same input types, duplicates might be found in the generated tuples. As an option, we also propose adding a distinct operation which might add an additional cost of computation. Lastly, if the input models are very large, tuples can be generated using a Cartesian product of RDDs. This approach starts to be very inefficient for tuples of size bigger than 2.

The second feature of choice is the type of structure used for storing the generated trace links at the end of the instantiate phase. These trace links will be used to resolve output elements in the application phase to create links. We propose three data structures:

- a List that is naturally made by construction during the instantiate phase. This approach is simple but terribly inefficient since elements cannot be accessed by index. Resolution is conducted by exploring the full list;
- an Array that is collected from the RDDs during the distribution of computation in the instantiate phase. As for lists, the resolution is costly and inefficient since the index of the array does not consider the stored elements, only their positions. However, Java uses less memory for storing an array instead of a list;
- a HashMap with the input element as key, and a list of output tuples as value, each containing a rule name, a pattern name, and a list of output patterns. This solution is the fastest to use but necessitates additional computation to build and more memory for storing the keys.

By default, all these structures store the elements and their associated output. It is also possible to only consider the identifiers of the elements to save memory.

Finally, the second part of the computation, designed in the apply phase, can be executed from trace-links or from scratch, as described in Section 2.3.2.2. The first solution reduces the global amount of computation by avoiding the second instantiation of output elements. However, it imposes a synchronization barrier called a gather operation to collect all trace links in the master node. Depending on the available amount of memory on the master node, this solution can slow down the computation. Recomputing the full links from scratch, including the application of instantiating part of rules to input patterns, allows a fully distributed computation. This approach takes advantage of a large number of computational resources available in a cluster. The two approaches might lead to duplicate results. That's why we also propose to apply a distinct operation on the resulting links. It adds more computation time but reduces the size of the result.

Additional Features Spark has more than 200 parameters to configure an execution environment. Existing work has illustrated how challenging it is to configure such an environment [26]. In SparkTE, we propose two internal configurations of Spark:

- The storage level of the distributed structures varies according to the available resources. Indeed, a small amount of memory would force a user to prefer disk usage instead of keeping all the data in memory. Spark proposes to define, for each RDD, a storage level which can be: only memory, only disk, or a hybrid solution. The latest favours memory usage, but start swapping on disk to avoid out-of-memory errors. In addition, it is possible to duplicate the data on several nodes (up to 3), ensuring higher fault tolerance and reliability. Finally, a user can specify Spark to serialize the content of RDDs on nodes to reduce its size. This additional computation increases the distribution time of data (caused by serialization operations) but profits to machines with a small amount of memory.
- Communications in Spark are mostly implicit. Contrary to other libraries (e. g., MPI [27]), the library gives a very high level of abstraction, where managing parallelism details such as the concrete distribution of chunks among nodes or the concrete communication are not doable from the user perspective. More generally, all the underlying computations are optimized by taking advantage of functional composition equivalences [28]. However, the broadcast and the gather operations, that is, sending data to all nodes and getting back all the results into a single node, are managed from the user perspective. The first can be implicit (by calling a variable from the sequential scope in an RDD) or explicit using the *broadcast* Spark operator. The second is usually processed by calling a *collect* operation at the end of a chain of distributed computation. This operation simply gathers all results

into an array on the master node of the cluster. It is also possible to use a reduction operation, called *fold*, to gather results in parallel into a single value. The latter can also be used as a *collect* operation.

2.4 Experiments

In this section, we aim to experiment with the different approaches we have implemented on the different levels of design. However, because of the complexity of experimenting on an engine with many possible parameters, this deliverable will not show results for this work. A preliminary approach to do so can be found in Section 3.

2.4.1 Expression evaluation

We present here the preliminary results of experiments we processed using the five different implementations presented above. Each example ran 30 times. The relative speed-up of the different solutions, compared to a naive sequential implementation, reported in Table 2 is the average over the 30 experiments of the maximum value of the execution times of all the Spark processes. Note that we only collect the execution times of `score` in its different implementation. The experiments have been processed on a shared memory machine (32 GB) with an Intel(R) Core(TM) i7-8650U processor having 8 cores at 1.90 GHz. We used the following software: Ubuntu 16.04, Java 1.8 with Scala 2.13.2 (Spark 3.0.1). The experiments have been conducted on 8 different data sets that can be found in a GitHub repository[4].

Dataset							Speed-up (compared to naive sequential)					
#	name	#users	#posts	#comments	#likes	size	Naive Sequential	Naive Parallel	MapReduce	Pregel	Naive + Pregel	MapReduce + Pregel
1	1	80	554	640	6	154 KB	**x 1**	x 0.40	x 5.82	x 10.30	x 9.40	x 4.63
2	2	889	1,064	118	24	251 KB	**x 1**	x 0.39	x 0.46	x 0.36	x 0.44	x 0.46
3	4	1,845	2,315	190	66	537 KB	**x 1**	x 0.51	x 0.85	x 0.68	x 0.66	x 0.71
4	8	2,270	5,956	204	129	983 KB	**x 1**	x 0.51	x 2.34	x 0.35	x 0.15	x 2.96
5	16	5,518	9,220	394	572	2 MB	**x 1**	x 4.25	x 4.17	x 5.21	x 4.68	x 4.03
6	32	10,929	18,872	595	1,598	4.22 MB	**x 1**	x 4.68	x 2.39	x 2.83	x 1.97	x 3.91
7	64	18,083	39,212	781	4,770	8.42 MB	**x 1**	x 4.07	x 4.58	x 4.12	x 5.17	x 3.27
8	128	37,228	76,735	1,158	13,374	17.1 MB	**x 1**	x 7.28	x 7.61	x 9.52	x 9.66	x 9.22

Table 2: Preliminary performance results of queries on a single machine.

It appears clearly that all the solutions do not provide the same speed-up depending on the data set. The first observation is that using a parallel solution on a small data set is not worth it. It can be explained by the difference between the computation and the communication cost. Having communications between processors increases the execution time of a program. To decrease the computation time of a parallel program, the part that is executed independently on each processor must propose enough speed-up to balance with the communication time. Second, no unique solution is better for all the data sets. For the 8^{th} data set, Pregel looks to provide a better speed-up than the other solution. On the contrary, in experiment 6, using Pregel seems not to be the best solution. Finally, we can observe that mixing approaches can largely increase the performance of the query (e. g., "Naive + Pregel" on the data set 7).

The observations do not provide formal proof to decide which strategy is better than another one for a given case. Additional experiments should be conducted with the following criteria:

- the use of data sets with more specific topology (e. g., high-number of comments, and sub-comments, for each submission),
- larger data set (the TTC18 provides three additional, larger data sets),
- the use of a distributed architecture with several nodes (e. g., Grid'5000 [29]).

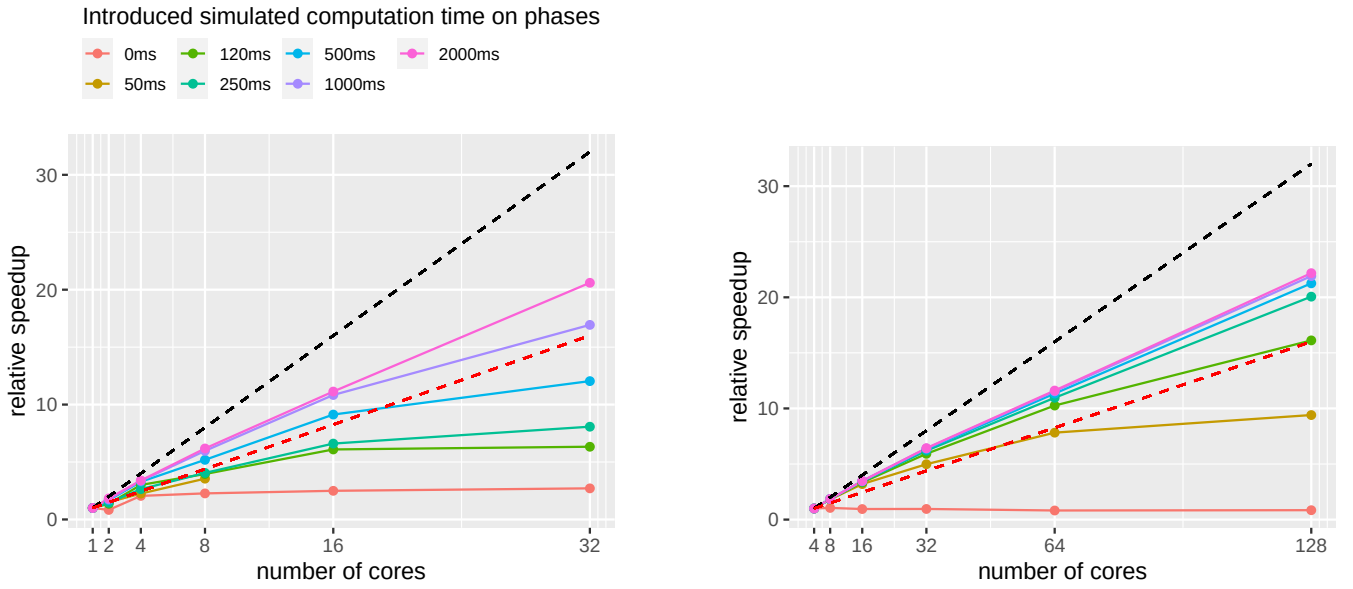
2.4.2 Transformation semantics

All results presented in this section have been executed ten times on the same hardware configuration, and an average is presented. Furthermore, all our experiments have been conducted on the French experimental platform for distributed computing Grid'5000. Grid'5000 (G5k) is made of geographically distributed clusters of machines in France, each one with its specific hardware setup. G5k is a large-scale and flexible testbed for experiment-driven research with a focus on parallel and distributed computing, including cloud, big data and AI [30]. It provides access to a large number of computational resources, physically located at 8 sites in France and Luxembourg. The resources are distributed to computational clusters from which several nodes can be booked. A node is a virtual machine deployed on a physical machine (server). This platform also facilitates the reproducibility of the experiments by offering a way to build the same environment for any researcher. However, Grid'5000 makes us dependent on available

[4] https://tinyurl.com/y63tcl6b

machines (i. e., nodes) for provisioning, which is why we use two different clusters of Grid'5000 in our experiments [29].

For each experiment, the number of machines and the number of cores is specified. One can note that the number of machines corresponds to the number of workers instantiated in Spark and that one additional dedicated machine is provisioned to host the master of Spark. All our codes, raw results, and analysis scripts are publicly available online[5]. The first model, designated as M1, is composed of 150 elements and 300 links, while the second (M2) is four times bigger: 600 elements and 1,200 links.



(a) Relative speed-ups of SparkTE for identity on M1 up to 32 cores on 4 machines.

(b) Relative speed-ups of SparkTE for identity on M2 up to 128 cores on 8 machines.

Figure 5: Relative speed-ups of SparkTE with sleeping times.

The transformations specified in CoqTL, subsequently translated to Scala and Spark, are always correctly computed by SparkTE, i. e., with the expected output of the transformation, for all our experiments. We have used from one to eight machines of the cluster `gros` (2x18 cores Intel Xeon Gold 5220CPU @ 2.2 GHz-3.9 GHz, 96 GB of memory, and interconnected systems at a bandwidth of 2x25Gbps). We want to show the performance gain obtained thanks to our three optimizations. For this purpose, we execute the same identity transformation on the direct implementation of the CoqTL specification (without optimizations) on M1. On one core naive is computed in 27 seconds by SparkTE and in 52 seconds by the CoqTL implementation.

To experiment with the scalability of the new semantic of SparkTE, we modelled computation time within the identity rules of the transformation. We explicitly called sleeping times on three parts of the rules: the guard condition, the instantiate expression and the apply expression. The shown results consider setting the same sleeping time for all three. More details about the experiments can be found in [13].

Figure 5 shows the speed-ups observed for each model according to the sleeping time and compared to the theoretical ideal speed-up. The black dashed line represents 100% of the ideal speed-up, while the red dashed line represents 50%. When increasing the sleeping time, i. e., the execution time of the transformation, the speed-up is enhanced and is closer to the ideal one. By increasing the size of the data set, one can note a slight increase in the speed-up by comparing B1 and B2 results, respectively, in Figure 5a and Figure 5b. Indeed, in Figure 5a, at 32 cores, more than half of the points are below the 50% optimal value, while in Figure 5b only one point is below this theoretical value. We observe that at 128 cores, the transformation of M2 only needs a sleeping time of 120 ms to reach 50% of optimal speedup.

2.5 Conclusion

In this Section, we have illustrated the variety of solutions for designing a distributed model transformation engine. We proposed SparkTE, a distributed model transformation on top of Spark, proposing correctness and good vertical scalability. Experiments have shown the scalability of the engine and the impact of the different configuration choices. In future work, we want to conduct experiments on all combinations of parameters for the SparkTE features on inputs with different topologies to illustrate the performance gain while having a well-configured engine, especially on VLMs.

[5] https://github.com/atlanmod/SparkTE_public

3 Multi-Parameter Benchmark Framework

Every non-trivial application has a large number of parameters, each of them having varying sizes of value sets, i. e., different numbers of values for those parameters. Some of these parameters influence the application's performance, i. e., execution time, memory, disk or network use, which is of interest to the application's users. They would like to have the application finish in the shortest time, use the least amount of memory or limit the network traffic to a certain extent. Often these goals conflict with each other; however, finding the best parametrization[6] is crucial to improve the user experience of the application.

To make the general goal more specific, we focused our research on Spark applications, i. e., software that can be deployed on a Spark cluster. SparkTE, that was introduced in Section 2, is such an application.

3.1 Research objectives

Finding the best parametrization in a naive way is a time-consuming endeavour due to the exponential number of combinations that have to be checked (cross-product of the parameter value sets). A way to reduce these combinations is to use good filter functions that remove the unnecessary combinations, e. g., those that do not have an influence on the optimization goal or might yield equivalent results. Another way is to leverage easy access to large amounts of computational capacity in the cloud, so we can yield the benchmark results faster by testing more parametrizations in parallel. However, this approach has high-cost implications that we have to pay. To summarize, our research objectives are:

RO1 Develop a multi-parameter benchmark framework to find the best parametrization of a Spark application according to a goal (i. e., execution time, memory or network use),

RO2 Use cloud infrastructure to test the parameter combinations to yield the fastest results.

3.2 Motivating example

Let's take a simple word count application that counts the number of occurrences of each word in a text (corpus), illustrated by Listing 5: Line 1, the instruction reads the file into an RDD in Spark. After that, in line 2, the content of the file is splitted by white space, groups the words by occurrence, and returns them in descending order of occurrence. Finally, line 3, the most frequent word is printed to the standard output. The parameters of the application are: the name of the file (`filename`), the replication factor (`replication`, on how many nodes the RDD will be copied on the cluster), the number of partitions (`partition`, how many RDDs the file's content will be split into).

```
1 val file: RDD[String] = nFile(filename, replicate, spark,
    partition)
2 val res = file.flatMap(line => line.split(" ")).map(word =>
    (word.replaceAll("[-+.^:,;)(_]", ""), 1)).reduceByKey(_
    + _).sortBy(e => e._2, ascending = false).collect()
3 println(res(0))
```

Listing 5: Word count example Spark application.

In our example, we used a corpus with 857,116 words (`bible.txt`), experimented with replication factors `1-2` and number of partitions `1-3`, as shown in Table 3. We measured both the execution time and the memory use of the application in six different scenarios to check all possible combinations of `replication` and `partition`.

File name	Replication factor	Number of partitions
bible.txt	1, 2	1, 2, 3

Table 3: Word count parameters.

[6] Best parametrization is a concrete binding of the parameters to values with which the application has the best performance.

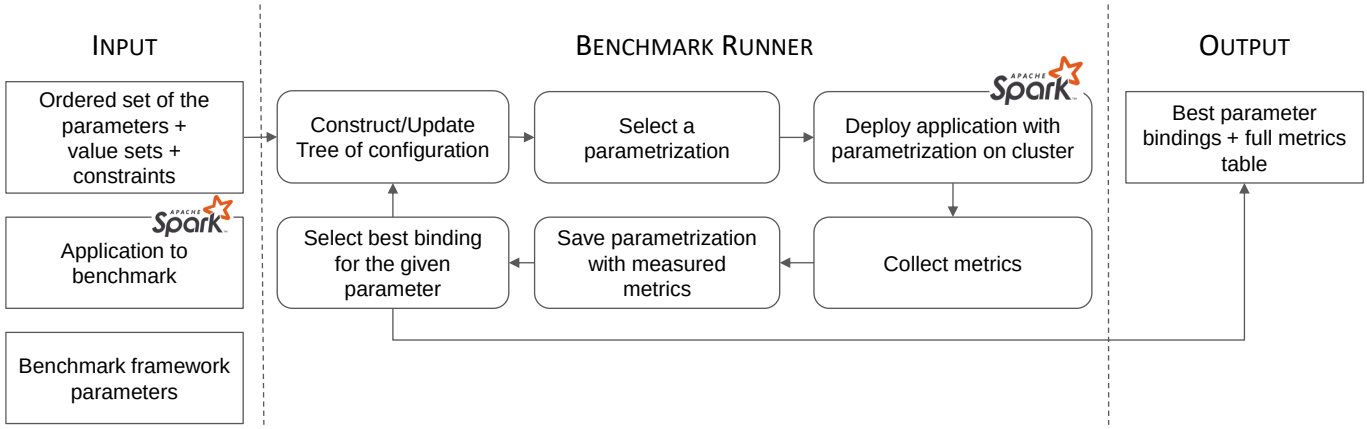


Figure 6: Overview of the approach.

3.3 Architecture and approach

Figure 6 provides an overview of our benchmark approach. It takes as input:

- the Spark application to benchmark,
- the ordered set of parameters with their value sets, and some constraints describing the prohibited combinations of the parameter values (e. g., `Parameter A` cannot have value `a`', if `Parameter B` has value `b`'),
- the benchmark framework parameters: Spark cluster configuration, the number of warm-up and measurement rounds with each combination and the number of parameter combinations we want to benchmark.

We need the input parameters to be sorted by priority, i. e., the parameter with the highest impact on the measurement goal (e. g., memory use, execution time or network traffic) should have the highest priority. Although several methods exist that help decide the prioritization (e. g., in an empirical way by learning from previous measurements, following theoretical assumptions, following industrial or research best practices), it was out of the scope of our research to decide which is the best way to set the priorities. In our case, we used the experiences gained from previous measurements to decide the priorities among the parameters.

After the prioritization, we built a Monte Carlo tree to perform a Monte Carlo tree-like search [31] on the different combinations of the parameter values. During the tree construction, we removed those nodes that would give a forbidden combination of the parameter values (due to the constraints defined between the parameters in the input step).

In each benchmark round, we (1) select the next parametrization from the tree, (2) deploy the application with this parametrization on the cluster, (3) run the application and collect the metrics, (4) save the parametrization with the measured metrics, (5) when we have enough data collected then we save the concrete value (binding) of the parameter that yielded the best result for the given metric, and (6) update the tree with this information and move on to get the next parametrization. In order to avoid concluding results from just one measurement, each parametrization is measured in m number of measurement rounds, preceded by n number of warm-up rounds. The average of the metrics values measured in the m number of measurement rounds will be the metrics values saved for that parametrization. We repeated the benchmark loop until we experimented with all valid combinations of the parameters or the number of combinations we wanted to benchmark. Finally, this algorithm returns the best parameter binding(s) for the given metrics and a CSV table with all measurement results.

Comparing benchmark results when multiple metrics are measured is a difficult task. Therefore, we give the opportunity to the user to define the comparison function to compare which measurement result is better in these cases. In our example (Section 3.2), we measured both the execution time and the memory use of the application. However, we gave higher weight to the execution use, i. e., the lower it is, the better, despite possibly yielding a higher memory use.

The benchmarking framework is independent of the execution environment (computation cluster) on which the application is deployed in Spark. In the following subsection, we introduce a cloud infrastructure, Grid'5000, that can be used to run the benchmark.

3.4 Evaluation

On G5k, we booked a single node with 64 GB RAM and Intel Xeon E5-2660 CPU at the Nantes site of the cluster and installed Spark on it. After that, we ran the benchmark workflow on the motivating example (Section 3.2) that concluded the following measurement results:

File name	Replication factor	Number of partitions	Memory use (MB)	Execution time (ms)
bible.txt	1	1	3,156	33
bible.txt	1	2	3,267	32
bible.txt	1	3	3,494	44
bible.txt	2	1	3,878	44
bible.txt	2	2	3,436	43
bible.txt	2	3	3,334	44

Table 4: Benchmark results on the word count example.

From the results in Table 4, we can conclude that replication factor 1 with two partitions is the best parametrization of the running example in the given deployment environment because it yields the shortest execution time (32 ms) despite a larger memory use (3,267 MB) than the smallest one (3,156 MB).

3.5 Related work

Several benchmarks and benchmark frameworks have been defined and implemented in the past to measure the performance of applications. One of them is MONDO-SAM, a framework to systematically assess MDE scalability. Izsó et al. [32] proposed the framework to enable systematic and reproducible benchmarking across different domains, scenarios and workloads in Model-Driven Engineering. The framework offers four extensible components for the benchmark: the model instantiator (to create the input models), the benchmark component (to measure the performance of different tools for given cases), the metrics evaluator (the tool-specific way of measuring a given metrics), the result reporting and analysis (to compare the measurement results and conclude the scalability of the measuring tool on the given cases). Similar to us, they were also benchmarking model-driven applications. However, their focus was on providing a generic, extensible framework to measure the performance of different tools on certain models and model-driven use cases (e. g., query, transformation). Compared to them, we focused on finding the best parametrization(s) of (model-driven) applications that can be deployed on Spark clusters.

Program autotuning is a paradigm that enables the software to tune itself to its environment, so it performs the best under the given circumstances [33]. Ansel et al. implemented OpenTuner [34], an extensible framework for program autotuning. The framework (*i*) employs different search techniques to find the best configuration (parametrization) of the given application, (*ii*) offers user-defined measurement functions that can be adapted to the applications' needs to correctly measure the metrics, and (*iii*) provides a results database to store the measured metrics. Compared to our solution, OpenTuner is a more generic and mature framework for finding the best parametrization of an application for given metrics. However, it does not support constraints to remove invalid parameter combinations, and its focus is not on Spark applications.

3.6 Conclusion

We introduced a multi-parameter benchmark framework to find the best parameterization of a Spark application. We used a cloud infrastructure (G5k) to test the parameter combinations and found the best parametrization for our motivating example application. The prototype implementation of the framework is available on GitHub[7].

In future work, we plan to extend the framework so that it is able to run the measurements in parallel on different machines in the cluster. Thereby speeding up the overall execution time of the benchmark workflow. Besides, we will experiment with different applications and advertise the framework for a broader audience in the MDE and software engineering communities so that other software engineers can also benefit from our work.

[7] https://github.com/lowcomote/multi-parameter-benchmark

4 Model Transformations and Scalable Model Checking in the Cloud

The elastic scalability of the cloud can be leveraged to deploy a large number of computational resources (CPU, memory, disk) in a short time and to be able to scale the system up and down according to the current service load. This means we are able to add new computational nodes to the system and deploy the application on them according to the user demand. In order to be able to leverage the elastic scalability, the cloud providers must provide appropriate APIs to deploy the new nodes at runtime, and the applications must be able to take benefit of the extra available resources; either by starting a new instance of a part of the software (component) that is under heavy load or to add the extra resource to the already running software system. This way, the end-user experience will improve as the software system will be responsive, despite the growing number of user requests. Of course, the cloud has its physical scalability limits due to the finite number of servers and computers building up the cloud infrastructure, but usually, it is much bigger than the applications in reality need.

In this section, we will introduce a cloud-based model validation and verification workflow that is able to verify the correctness of SysML state machines and activity diagrams with respect to reachability properties. SysML (Systems Modeling Language) is a standardized modelling language by OMG [35] to describe the structure and behaviour of systems. Several vendors and products are adopting the requirements, design, simulation, and code generation aspects of the standard.

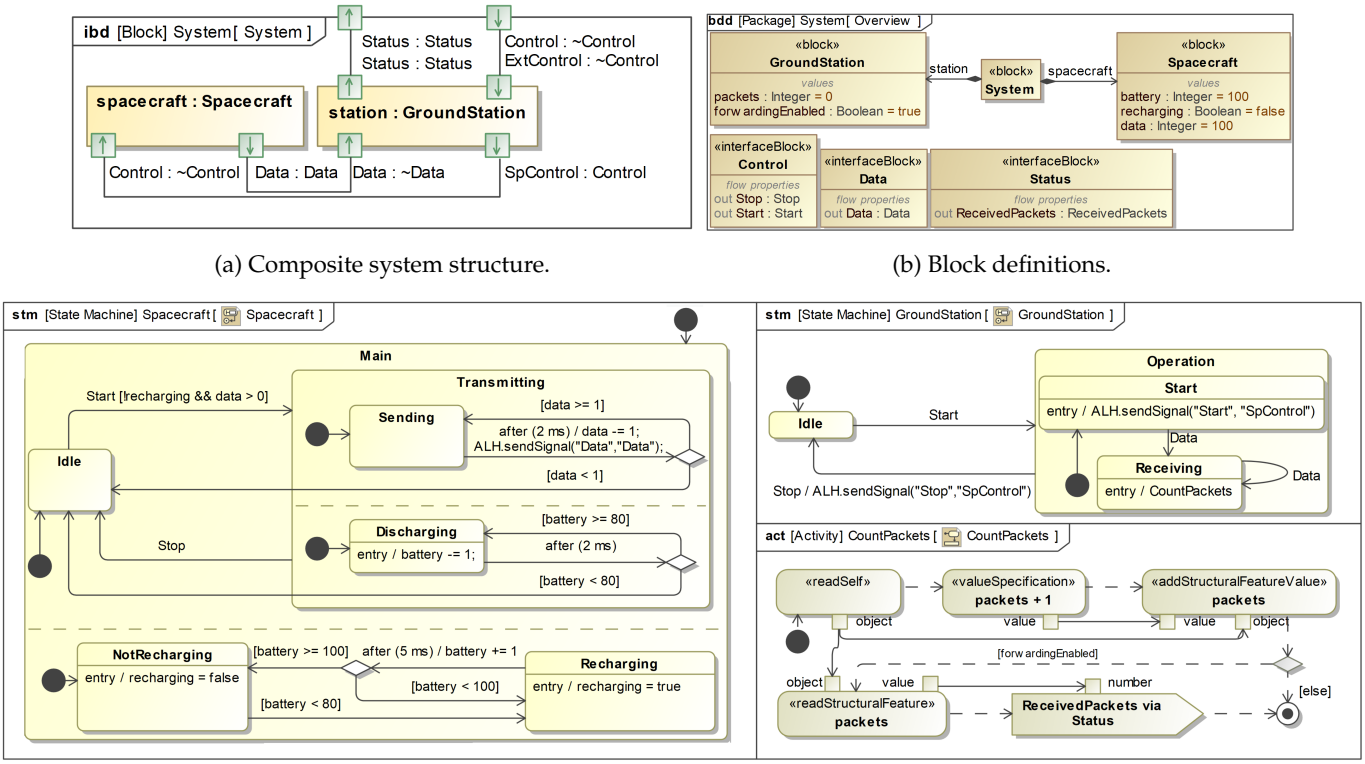
A predecessor of this work was published at the OpenMBEE workshop of the MODELS conference in 2020 [36]. A recent work version is under review in a renowned systems engineering journal.

4.1 Research objectives

We aimed to leverage the cloud with the following questions that we use as research objectives:

- *RQ1*: Can the workflow verify reachability properties on artificial and industrial models?
- *RQ2*: What are the benefits of using a cloud-based approach in the deployment if multiple users concurrently use the workflow on an industrial model?

4.2 Motivating example



(a) Composite system structure.

(b) Block definitions.

(c) Behaviour definitions.

Figure 7: A simple SysML model describing a simplified Spacecraft and Ground Station to illustrate the scope of our approach.

Consider the composite system of a simplified Spacecraft and Ground Station model in Figure 7 adopted from the OpenSE Cookbook [37]. As soon as the Ground Station is in `Operation`, it notifies the Spacecraft to `start` sending `Data`. The station counts and forwards the incoming data packets via its `Status` port.

The Spacecraft is waiting `Idle` until it receives a `Start` signal from the station to start sending data in packets. Data transmission consumes 1% of `battery` power per packet, and if the battery level falls below 80%, ongoing data transmission is paused until full recharge.

There are several properties that the system design has to fulfil. Let's consider that the Spacecraft (*a*) should only start transmitting after receiving a `Start` signal, and (*b*) should never transmit when the `battery` is below `80%` and the Ground Station has already received at least `20 packets`.

While Property (*a*) could be checked in principle by reviews or model validation rules, Property (*b*) is much harder as we have to consider all feasible paths in the model.

These properties can be captured as *reachability properties* on state machines of the composite system. Such properties describe (un)desirable state configurations whose existence shall be proven by model checkers. State configurations (state predicates) consist of states of the state machines and logical expressions over their variables. If the state configuration is reachable, the model checker returns an execution trace, proving how the system gets into this configuration starting from the initial configuration. If the state configuration is desirable, it is correct if it may exist in the system. On the other hand, if it is an undesirable state configuration, it is an error if such a configuration exists.

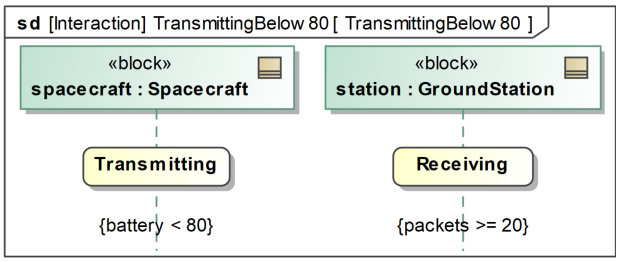


Figure 8: Reachability property.

SysML sequence diagrams can be used as a syntactic means to define the reachability property. The diagram consists of lifelines representing the part properties of the composite block. Each lifeline can contain several state invariants defining the state configuration to be reached by the respective state machine. A subset of JavaScript can be used to express logical predicates over variables. The reachability property of the composite system is the conjunction of the state predicates defined over the lifelines. The property, illustrated by Figure 8, defines the undesired configuration where the `Transmitting` state of the Spacecraft is active, the `battery` level is below `80%`, the Ground Station is in the `Receiving` state and has already received at least `20 packets`, corresponding to Property (*b*).

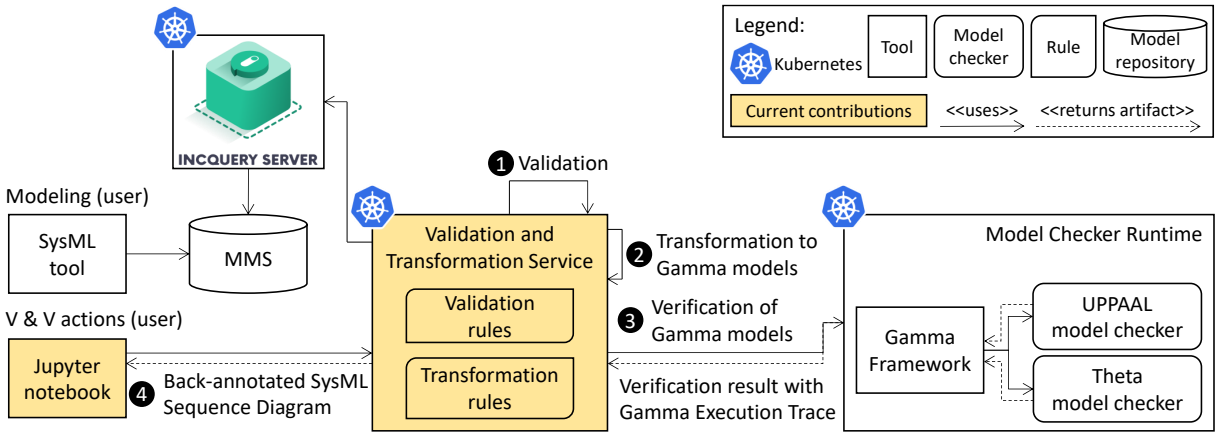


Figure 9: Overview of the architecture.

4.3 Architecture and approach

We implemented a verification and validation approach using proprietary and open-source tools. For modelling, we chose Cameo Systems Modeler[8], a proprietary SysML design tool, due to its popularity in the industry. For the verification, we used the open-source Gamma Statechart Composition and Verification Framework [38] due to its precise formal semantics and automatic translation to different model checker back-ends, i. e., UPPAAL and Theta. The architecture of the workflow is depicted in Figure 9.

Systems engineers design the state machines activity diagrams and define the reachability properties of interest in SysML modelling tools, and push them to the OpenMBEE MMS[9] model repository. Then,

[8]https://www.nomagic.com/products/cameo-systems-modeler
[9]https://github.com/Open-MBEE/mms-alfresco

Element	Severity	Message
4a609931-cd5a-4716-ba45-6f8579f97ad5	WARNING	Regions OperationRegion and BatteryRegion are conflicting regions. They, or their transitively contained regions, use the same variable with read-write or write-write access. Verification will choose a fixed scheduling for these regions and might not discover behaviors reachable with a different scheduling.
fc2bbbe9-8f26-4d5f-bf3d-d001146abc81	WARNING	Regions BatteryRegion and OperationRegion are conflicting regions. They, or their transitively contained regions, use the same variable with read-write or write-write access. Verification will choose a fixed scheduling for these regions and might not discover behaviors reachable with a different scheduling.
cd8743ff-855b-4bec-a918-13f6557f6007	WARNING	Orthogonal regions of the Transmitting state will be scheduled according to their names during verification. Scheduling order is lexicographic.
b786da69-3013-41cb-8ccd-b7874bc40e88	WARNING	Orthogonal regions of the Main state will be scheduled according to their names during verification. Scheduling order is lexicographic.

Figure 10: Validation results in the Jupyter notebook.

users open a web browser to perform verification and validation (V&V) actions in a *Jupyter notebook*[10]. The frontend is connected to the *Validation and Transformation Service (VTS)* in the backend, which validates the structural correctness of the SysML models ❶ before transforming them to Gamma models ❷ and verifying them with the Gamma Framework ❸. The verification result, including a possible Gamma Execution Trace that is back-annotated to a SysML sequence diagram, is presented in the browser ❹.

The *Jupyter notebook* is an open-source web application that allows users to execute code and show visualizations in a structured way in the browser. A notebook is divided into different steps, each of them having a description and a runnable code fragment. We use Python in the notebook to demonstrate the API use of the *VTS* and its integration possibilities with other tools as well. Figure 10 illustrates the result of the validation step in the notebook. The user can see the validation messages and their severity accompanied by the ID of the erroneous element. By clicking on the element ID, the user can navigate to the element directly in the browser using an appropriate model viewer or, alternatively, a desktop SysML tool.

On the backend, the Gamma Framework translates the intermediate models to formal models and queries to be checked by model checkers, i. e., UPPAAL [39] or Theta [40]. The philosophy of Gamma is to have a portfolio of model checkers. Each tool implements different algorithms tailored to specific classes of problems. Therefore the larger variety increases the chance of successful verification. For example, UPPAAL uses explicit model checking and is efficient for timed systems, while Theta uses a wide array of abstraction-based symbolic techniques that have a larger overhead on simpler problems but have a higher chance of verifying harder ones where an explicit algorithm would not scale.

In order to have a scalable architecture to validate and verify industrial SysML models, we moved the processing from the client machines to the cloud. Most components in the architecture, marked by a logo on the architecture figure, can be deployed on Kubernetes[11], an open-source cloud orchestration, deployment and scalability tool for containerized applications. Applications can be deployed into different execution units, called pods, with a certain amount of computational resources allocated. This way, they can be scaled up with a high amount of CPU and memory (vertical scalability), or new pods can be started on demand (horizontal scalability) as long as the cluster has enough computational resources. The ability to scale out in the cloud allows the adaptive allocation of a high amount of computational resources [41], that can address the high resource demand of the *Model Checker Runtime (MCR)* component. Moreover, the portfolio of model checkers philosophy of Gamma can also benefit from the cloud-based setting because we can execute multiple model checker configurations in parallel (and stop when one of them gives a result). Ultimately, if we accumulate enough data about the target models, we may be able to reduce the executed configurations to a few promising ones and achieve a relatively good price/performance ratio.

As model checking is the most resource-intensive task in the workflow, we serve each verification task in a separate pod, running the *MCR*. This way, the long-running tasks can be served in parallel, and as the concurrent pods can be deployed on physically different machines, they do not hinder the performance of a single computer. Therefore, the workflow can serve many users running their verification tasks concurrently and enable its use as a common verification service for a team of systems engineers.

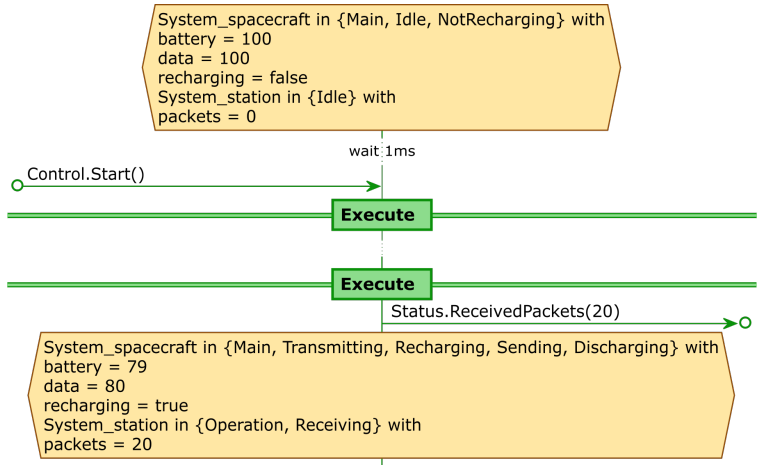
Moreover, to improve the performance of the static validation and transformation phases, we use IncQuery Server (IQS), a scalable model query middleware on top of model repositories [42]. IQS builds an in-memory index from the model and efficiently evaluates model queries implemented in Viatra Query Language [7]. These queries are used by model validation and transformation rules. We implemented several model transformation rules that work on the index and build the target and traceability models. The transformed models are persisted and used in the verification and back-annotation phases of the workflow.

Besides scalability, another advantage of our approach is the separation of concerns for the engineering and the formal verification domains: systems engineers design both the models and the reachability properties in a high-level engineering language they are familiar with. The formal models are automatically derived from these high-level design models by a series of transformations hidden from the users. The ver-

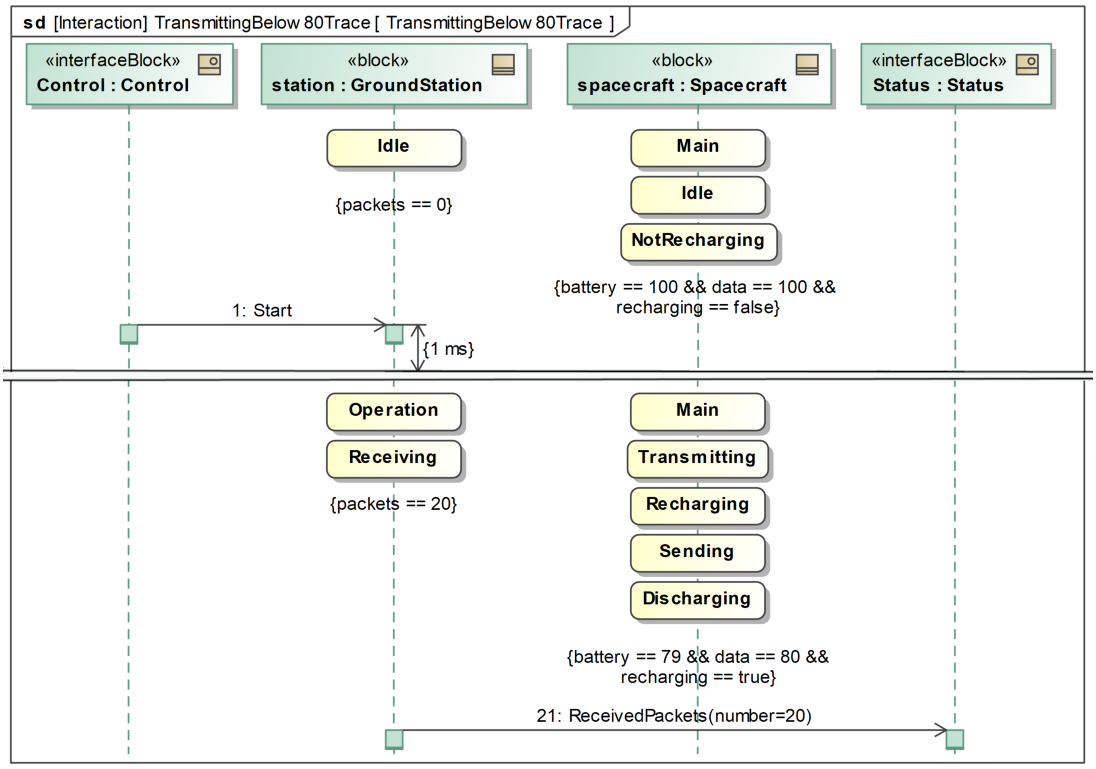
[10] https://jupyter.org

[11] https://www.kubernetes.io

ification result is back-annotated to the original engineering language, thus making it easier to understand without expertise in formal methods. Besides, the workflow helps engineers to inspect the execution in detail or to derive further test cases.



(a) Execution Trace in Gamma.



(b) SysML sequence diagram trace.

Figure 11: Execution Trace to SysML sequence diagram mapping.

Demonstration of the workflow

The workflow can be illustrated by checking Property (*b*) on the motivating example (Figure 8). The verification result trace (Figure 11) proves that the property is violated, because in the last state `battery` is `79%`, the Spacecraft is still transmitting and the Ground Station has already received `20 packets`. Simulating the sequence diagram trace in Cameo Simulation Toolkit[12], one can find that the entry action of `Discharging` state causes the issue (Figure 7c). The faulty model can be *fixed* by moving the action to the effect of the outgoing timed transition. Rerunning the verification proves that the undesired state is not reachable anymore.

Due to space limitations, the details of the Gamma Execution Trace to SysML sequence diagram mapping are skipped here. Interested readers may find them in the journal paper that is under review in a journal.

[12]https://www.3ds.com/products-services/catia/products/no-magic/cameo-simulation-toolkit/

Name	Target state	Number of states	Number of activity actions
TMT1	Minimize Sensor Readings	5	213
TMT2	M3 Alignment 3	8	533
TMT3	Broad Band Phasing 3um	9	636
TMT4	Broad Band Phasing 1um 3	10	739

Table 5: Reachability properties in TMT and metrics on the shortest path to reach the target state.

4.4 Evaluation

We evaluated our proposed V&V workflow in different scenarios imitating real-world use to answer the research questions posed in Section 4.1. In order to answer these research questions, we set up an evaluation environment on Amazon Elastic Kubernetes Service (EKS[13]). The infrastructure consisted of two EC2 nodes: one m5.xlarge instance (4 vCPUs, 16 GB RAM) and one m5.2xlarge instance (8 vCPUs, 32 GB RAM). The components of the architecture in Figure 9 were deployed as follows: *IncQuery Server (IQS)* and the *Validation and Transformation Service (VTS)* were running on the m5.xlarge instance for each research question; the *Model Checker Runtime (MCR)* was deployed in a varying number of pods on the m5.2xlarge instance, depending on the scenario.

For benchmark purposes, we used the running example *Spacecraft model* (Figure 7) with two versions and a modified version of OpenMBEE's Thirty-Meter Telescope (TMT)[14] model. In the *faulty* version of the Spacecraft model, the reachability property (Figure 8) is satisfied; in the *fixed* version, the reachability property is unsatisfied due to corrections described in Section 4.3. Both the *faulty* and the *fixed* versions of the model contain 11 states, 23 transitions and 5 activity actions.

In the *TMT model*, we chose the Procedure Executive and Analysis Software (PEAS) block and adapted its state machine and activities to the elements supported by our workflow (depicted on Figure 7). We removed do-behaviours from the state machine, removed unsupported Actions from the Activities, replaced Float variables with Integers, and resolved data type inconsistencies in Activity actions. This was necessary to make the model conform to our approach. Nevertheless, the resulting model is still quite complex and represents the general modelling patterns used in the OpenSE Cookbook. We specified four reachability properties for the TMT model, denoted as *TMT1–4* in Table 5. The table contains the target state and the number of states and activity actions for the shortest paths satisfying the respective property. Altogether the PEAS block contains 61 states, 93 transitions and 2,310 activity actions.

4.4.1 Research question answers

We describe the scenarios in which the workflow was executed and the obtained results.

Answers to RQ1

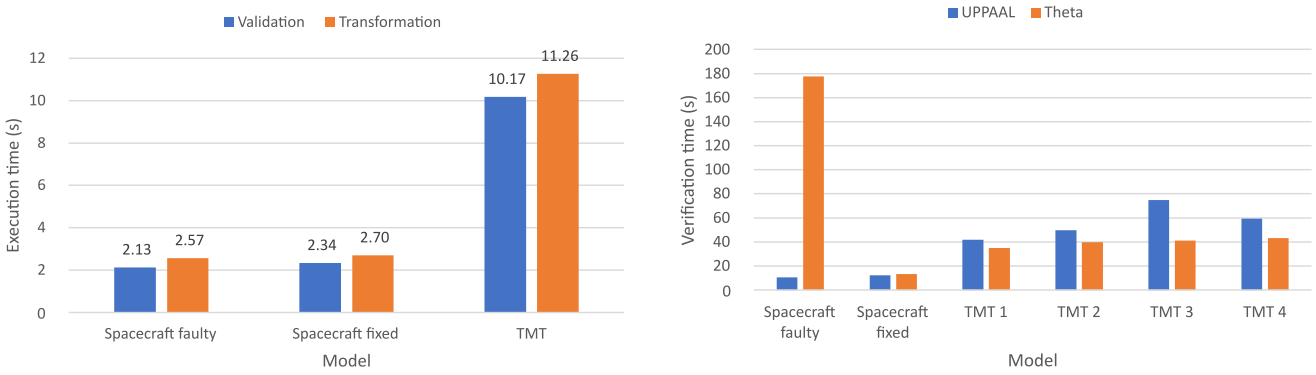
Scenario: We deployed the *MCR* in one pod, and we only sent one validation, transformation and verification request at a time for each model and reachability property. This scenario serves as a high-level validation of the approach and a baseline for the other scenarios.

Results: Figure 12 depicts the measurement results. Figure 12a shows that the model validation and transformation were completed in an acceptable time, despite running more than 130 structural validation rules. The model sizes are reflected in the execution times: while the simpler Spacecraft model is validated in 2 seconds and transformed in 2.5 seconds, the more complex TMT model is validated in 10 seconds and transformed in 11 seconds.

Regarding the verification times, depicted in Figure 12b, it can be seen on the hand that UPPAAL outperforms Theta for the *Spacecraft faulty* model, which contains timed transitions. On the other hand, Theta performs better for the TMT model, which contains many data variables due to the large number of activity actions whose instructions are transformed into variable assignments. The large difference in verification times by UPPAAL and Theta on the *Spacecraft faulty* model is due to the fact the abstraction-refinement algorithm implemented by Theta tracks the values of the timer variables in the model, which results in a large number of combinations that need to be checked by the model checker. However, in the case of the *Spacecraft fixed* model, where the property is unsatisfied, Theta finishes in about the same time as UPPAAL due to the abstraction domain used in verification.

[13] https://aws.amazon.com/eks/

[14] https://github.com/Open-MBEE/TMT-SysML-Model



(a) Validation and transformation times.

(b) Verification times.

Figure 12: Validation, transformation and verification times of the Spacecraft and TMT models.

Answering RQ1: As Figure 12 shows, the approach can transform and verify properties on artificial and complex industrial models within an acceptable time. Besides, the differences between explicit and abstraction-based model checkers could also be observed: UPPAAL may handle timed systems better than Theta. However, further models and verifiable properties are needed to draw a definitive conclusion in this respect.

As Figure 12 shows, the verification times are the most influential in the whole validation and verification workflow, therefore in the subsequent research question, we only measure the verification time.

Answers to RQ2

Scenario: This scenario investigates that if we have multiple verification requests, possibly coming from different users, then how does the approach scale if more than one *MCR*s are running verification? Therefore we deployed the *MCR* on one, two, and four pods. Each pod was handling only one verification request at a time. We sent four verification requests in a fixed order directly after each other asynchronously to the workflow. The verification requests were waiting in a queue until being processed by a free pod. We repeated the measurements on each model checker (UPPAAL and Theta) separately, resulting in six sub-scenarios (3 pod configurations × two model checkers).

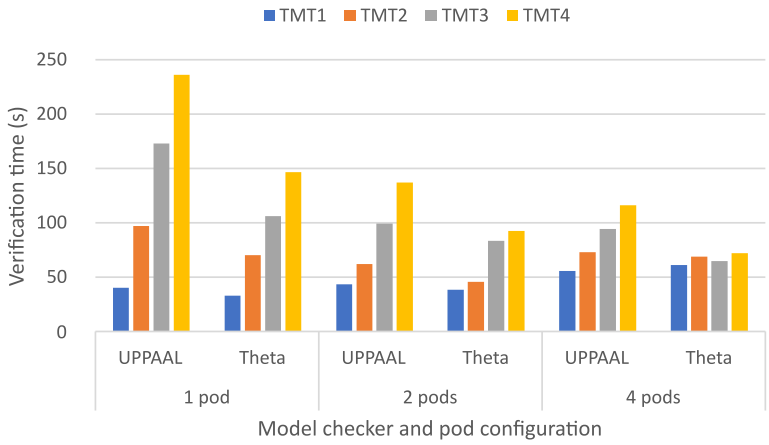


Figure 13: TMT model verification times.

Results: By looking at Figure 13, we can see the execution order and also the effect of the queuing times of the requests (*TMT1–4*). As the number of processing pods grows from one to two, so is the overall execution time of the requests reduced. However, in the case of four pods, we can see that the concurrent verification requests give a higher load to the only processing m5.2xlarge node, which results in longer verification times than in Figure 12b when only one request was processed by the node at the same time.

Besides, we can also observe that Theta verified the models faster than UPPAAL, confirming our observations from *RQ1*.

Answering RQ2: To conclude, the benefit of using a cloud-based approach in the deployment is, as the number of pods grows, so can we serve more users concurrently with the same resources, which can result in lower round-trip times for long-running verification tasks and better resource utilization. However, we shall also consider the computational capacity of the processing nodes when allocating pods to them to avoid starvation between them.

4.4.2 Threats to validity

To address threats to the *internal validity* of the measurements, we executed 20 rounds of warm-up requests in each phase, followed by the seven rounds of measurements whose median represents a data point on each figure. In order to address the *external validity* concerns, we used an industrial model (TMT) apart from a syntactic model (motivating example). However, our results might not be generalizable for models built from elements that are not part of the supported elements (construct validity; Figure 7). Nevertheless, it is important to note that industrial models are generally confidential and hard to access, so having a solution presented in this report is vital to move forward to industrial use.

4.5 Related work

To put our research in a broader context, we collected related work about the practical advantages and perceived challenges of using *model-driven engineering and formal methods in industry* (Section 4.5), and applications of *hidden formal methods* in verifying system models (Section 4.5).

MDE and formal methods in industry

Bucchiarone et al. [43] described tool and implementation challenges hindering the widespread use of Model-Driven Engineering (MDE). Scalability in terms of size and diversity of artefacts, i. e., models, metamodels, model transformations and dependencies in any non-trivial project, has been denoted as one of the open challenges that tools tried to address in the last decade. We encountered similar challenges and therefore focused on an integrated, scalable tooling environment.

Garavel et al. [44] surveyed 130 high-profile experts in different aspects of formal methods for the 25th international conference on Formal Methods for Industrial Critical Systems. One aspect was the industrial adoption of formal method practices. 67.7% of the responders believe that formal methods are now ready to be used in industry to a limited extent. The reasons for the limited applicability are often related to the domains, tool maturity, and people's skill and willingness to transfer and adopt academic research results to industrial case studies. According to the survey, the most mentioned limiting factor of wider adoption of the formal methods by industry is the improper integration of formal methods in the industrial design life-cycle, the lack of proper training of FMs and its steep learning curve. According to the research participants, more collaborative projects between research and industry and increased support for academic researchers developing tools can contribute to addressing these challenges. We contributed to overcoming these limitations by developing a workflow that is tightly integrated into the engineering environments.

Gleirscher and Marmsoler [45] surveyed 216 participants from industry (78%) and academia (22%) about the use of formal methods (FM) in mission-critical software domains. Their results indicate an increased intent to apply FMs in the industry across all application domains, suggesting a positively perceived usefulness. Besides, the intrinsic motivation to use FM is stronger than the regulatory one. Scalability, skills, and education were perceived as the toughest challenges of applying FMs in practice. More experienced respondents more often rated these challenges as highly difficult compared to less experienced ones. Finally, past experience with formal methods was positively correlated with future usage intent. We observed a similar situation in NASA JPL, where successful previous projects [46, 47] opened the way for working in an environment to use formal verification as a service.

Verifying systems with hidden formal methods

Software and systems model checking is widely researched, with many tools and approaches available. *Ciccozzi et al.* [48] performed a systematic review of solutions to execute UML models. The closest one to ours is *Kölbl et al.* [49] who proposed an approach to translate SysML models to the language of NuSMV, Prism, and Spin model checkers. Similar to us, they used an intermediate metamodel between the engineering and formal domains, and the counter-examples are returned as SysML sequence diagrams. In contrast, they verified Linear Temporal Logic (LTL) expressions specified as OCL state invariants in SysML, supported only send signal actions in activities, and used only a small model.

Calvino and Apvrille [50] proposed the direct model-checking of SysML state machine models. In their paper, they used AVATAR to design the SysML models that are directly verified by TTool. They support verifying a broader set of formal expressions specified in the text. The verification results of reachability and liveness properties are back-annotated directly to the state machines, but a trace of the original model checker representation is also returned.

Gibson et al. [46] verified properties on SysML statecharts by combining code generation with software model-checking techniques. They translated the state machines to Java code by the COMODO model-to-text transformation tool and evaluated certain properties by Java Pathfinder [51]. The verifiable property was directly inserted in the generated code, the guards of the transitions were transformed manually, and the result (trace) was not annotated to the original model. They used depth limits as a trade-off between performance and the ability to verify properties [47].

4.6 Conclusion

In this section, we presented a cloud-based, scalable model transformation and verification workflow to verify reachability properties on SysML state machines and activity diagrams, back-annotating the verification results to the original domain, thereby adopting the so-called hidden formal methods approach [36].

One way to improve the performance of the workflow is to adopt the parallel-reactive model transformation proposed [9, 11]. It enables live, incremental model transformations, i. e., if the source model changes, then only the impacted parts of the target model will change, which results in a shorter transformation time. Another future work is to extend the supported set of SysML state machine and activity diagram elements to be able to verify more complex models that are useful for systems engineers.

5 Composition of Model Transformations

This section discusses model transformation compositions and their associated services i. e., chain selection and chain optimization.

5.1 Research objectives

The general research objective is to compose and execute model transformations that enable the development of complex transformations by reusing and composing simpler and smaller ones [52]. There are three major activities performed in composing model transformation. First, we have to identify the possible transformation chain by using a cloud-based search engine [53] or through static analysis of the transformation language such as Epsilon [54]. The second major activity is to select the transformation chains by estimating the optimal criteria such as transformation coverage, transformation complexity and the number of transformation hops. Lastly, another activity involves optimizing a transformation chain by identifying the usage of the elements that are propagated till the final target model.

5.2 Model Transformation Chain Identification

Identification of possible model transformation chains can be made using the following steps. The first step is to check if the direct model transformation exists that transforms the source model to the target model. This is the base condition. Then we identify all the available model transformations that transform the source model into any of the available intermediate models. Further, we reuse the intermediate model as the source model with the help of base condition (using depth-first search recursively), and we repeat this step till we reach the target model. Then, we check all the available model transformations that transform the identified intermediate model into the target model. Lastly, the source, intermediate and target models are composed and returned as a model transformation chain.

The input for identifying possible transformation chains is the source and target metamodel. From Figure 14, the input metamodels are KM3.ecore, XML.ecore. After applying the logic mentioned in the Listing 6, the output would give a list of lists of chained ecore which is [[KM3.ecore, Ecore.ecore, JavaSource.ecore, Table.ecore, HTML.ecore, XML.ecore], [KM3.ecore, JavaSource.ecore, Table.ecore, HTML.ecore, XML.ecore], [KM3.ecore, XML.ecore].

```
Input: sourceModel, sourceMM, targetModel, targetMM
Output: List of possible transformation chains from sourceModel to targetModel
identifychain(sourceModel, sourceMM, targetModel, targetMM) {
//returns true if a tranformation transforms sourceMM to targetMM
findEtl(sourceMM, targetMM)
if(findEtl(sourceMM, targetMM) is true)
    Store sourceMM, targetMM in an ArrayList A1
else
Traverse the folder/repository that contains metamodel files
Store each metamodel as an intermediate metamodel MM_inter and give a name to
it's model M_inter is the same as the name of the metamodel
if(findEtl(sourceMM, MM_inter) is true)
    identifychain(sourceModel, sourceMM, M_inter, MM_inter)
    Store MM_inter in the ArrayList A2
    sourceModel <- M_inter
    sourceMM <- MM_inter
if(findEtl(MM_inter, targetMM) is true)
    Store target<< in ArrayList A2
Add A1 and A2 in the list of ArrayList A3 and remove duplicates, if any.
Return list of Arraylists of chained transformations that can transform from
sourceMM to targetMM
}
```

Listing 6: Algorithm for identifying possible chains.

5.3 A motivating example of the selection of a model transformation chain

When chaining model transformations and multiple chains are available in our setting, different criteria can be defined to characterize the selection of an, in certain ways, optimal chain. The possible criteria mentioned in [55] are transformation coverage and information loss, but also other parameters may be considered or even combined. An example of the problem of chaining model transformation is borrowed

from [55] and it is depicted in Figure 14. Given a user input, composed of the input model, i. e., sample-km3 which conforms to the source metamodel KM3, and the required metamodel in output, i. e., XML, the process of finding optimized chains can be summarized as follows.

First, a discovery phase starts by inspecting the available transformations in the repository, which can also be a local folder, selects the needed transformations and derives the needed transformation chains from reaching the result. In Figure 14, steps involving identifying possible transformation chains are represented as filtering from the repository in which the transformations and the associated metamodels can be chained. In our scenario, the selected metamodel nodes and transformations are reported in dark grey, excluding the rest of the repository in light grey.

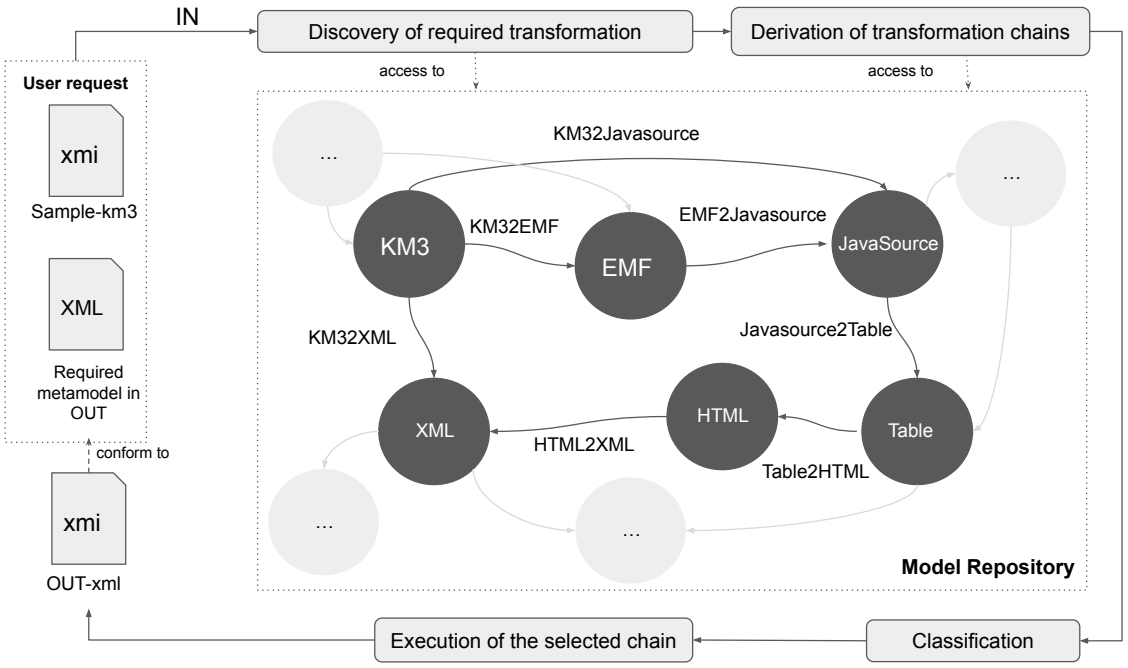


Figure 14: Model transformation chaining scenario.

In this scenario, the list of available chains is composed of three available chains:

$Ch_1 : KM3 \rightarrow EMF \rightarrow JavaSource \rightarrow Table \rightarrow HTML \rightarrow XML$

$Ch_2 : KM3 \rightarrow JavaSource \rightarrow Table \rightarrow HTML \rightarrow XML$

$Ch_3 : KM3 \rightarrow XML$

This list of available chains is given to the classification process that ranks the list according to the selected quality criteria or parameters, and the selected chain is then executed. The execution of the chain returns the required model, i. e., a model conforms to the XML metamodel. As said, the quality criteria that can be considered in this scenario may be various. For instance, if we consider (i) transformation coverage, (ii) transformation complexity or (iii) the number of transformation hops, we may have different results in the selection process. Transformation coverage is defined as the degree of completeness of a transformation, which means how many metamodel elements (i. e., metaclass and attributes) of the source model to be transformed are consumed by the transformation [56]. This may affect the transformation process since the more the transformation covers constructs from the metamodel, the less the result should lose output elements. The transformation complexity can be estimated by how much the rules, operators and expressions are used in the transformation chain. This is a new criterion to define a chain as the best one by using a heuristic which describes the complexity of the transformation by counting the static elements of the transformation. This may affect the result in terms of performance, for example, since the more the transformation engine has to interpret complex operations, the slower the execution will be. When the repository contains large graphs of available transformations, the user may be interested in getting the result faster. The third criterion considered is the number of transformation hops that are used to achieve the target model. This is calculated as the width of the graph visit for each transformation chain. Again, the performance may be affected as well as the output model to be generated since reducing the number of hops may reduce the chance of encountering a bottleneck transformation. In this case, it would make sense to combine the coverage and complexity with these criteria so that the selection becomes a multi-criteria optimization problem.

The problem of selecting the optimal chains according to different criteria is an open issue that has been partially covered in [55, 57], where these criteria are hard-coded in algorithms implementing the classification mechanism. This results in limiting, first for the effort required to add new criteria in the evaluation, second for the lack of support in defining an optimization strategy considering multiple criteria.

To overcome these limitations, in the next section, we present how we have used MOMoT, a model-driven optimization framework [58] using search-based techniques to optimize different criteria and, there-

fore, support selecting optimal transformation chains.

MOMoT[15] is a framework that uses MDE principles to solve complex multi-objectives problems by using search-based optimization. The problems are represented as Ecore metamodels, and particular instances of the metamodel are used to solve a specific problem. This problem represented by the metamodel is manipulated by an in-place graph model transformation expressed in Henshin [59]. To that end, the framework targets optimal transformation sequences leading to optimal models rather than direct model manipulation. The output model is characterized by different constraints and objectives which are written in OCL or a Java-like expression language (Xbase) [58]. Finally, a sequence of transformation and parameters are executed, and the Pareto-optimal solution is found by using search-based optimization [60] that includes different algorithms such as Random Search, NSGA-II [61], NSGA-III [62], etc., which are defined in MOEA[16] framework.

5.4 Approach: Chain selection with MOMoT

In Figure 15, we outline how we use MOMoT to support the selection of optimal chains. It shows the workflow used to run the selection process, starting from the definition of the needed artefacts, the configuration of the existing MOMoT modules, and the obtained results. In the following, we walk through the process and describe the artefacts used following the labels in Figure 15. As elaborated in Section 5.3, we anticipate the user input to include an input model and the required output metamodel. In addition, the repository that will be analyzed by looking for chains is also given as a parameter. In our case, we use a local folder containing Ecore models (metamodels) and ETL transformations. ETL is the transformation language part of the Epsilon [5] framework that we have used to test the implementation, but the approach can be easily re-applied for other transformation languages, e. g., ATL [6].

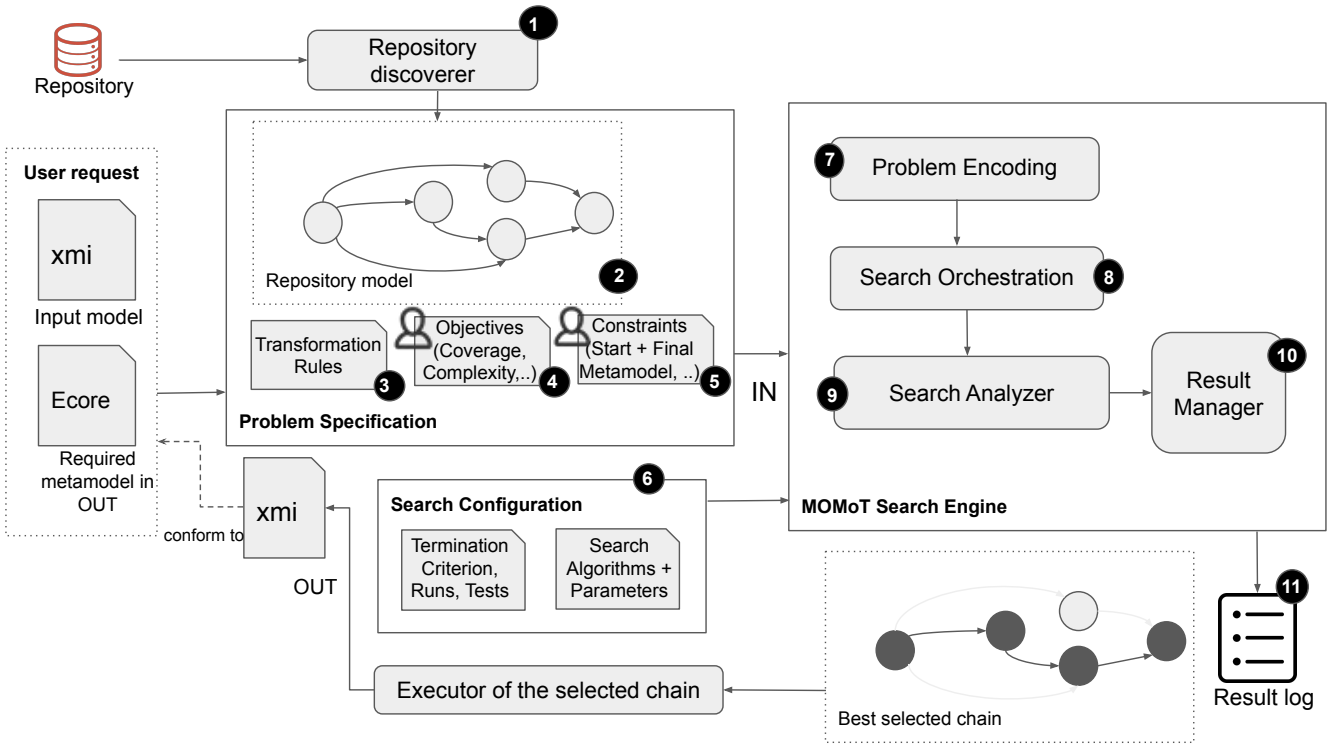


Figure 15: The proposed MOMoT extension for best chain selection.

In the following paragraphs, we address the different parts of the problem specification shown in Figure 15 and describe how optimal chain selection is supported with the individual components of MOMoT. First, the process for retrieving available transformations from the repository is described. We then present the metamodel, which describes the problem domain, hence carries elementary information of the chain selection task and maintains the current solution state, i. e., the transformation path with respect to the input model to be transformed. Afterwards, the transformation model used for assembling feasible chains through evolving the model-based problem representation is described. Thereafter, the quality criteria that we consider and evaluate in the chains are presented along with the validity conditions. Then, the configuration of the search-based evaluation process is explained, including the algorithms used. Finally, the MOMoT arrangements are discussed, including (1) the approaches to rewrite the problem model

[15] http://martin-fleck.github.io/momot

[16] http://moeaframework.org/

by rule applications to determine optimal transformation chains and (2) the means provided for comparing and processing the resulting chain solutions.

Repository Discoverer A repository/folder is first analyzed by the Repository discoverer ❶, which may be implemented based on the tools developed in [63] or programmatically finding out the possible chains. This module analyzes the repository given as input in order to create the Repository model in ❷, which is needed by MOMoT as part of the problem specification. We summarize the discovery strategy as follows: for each Ecore model, we programmatically create a node of the graph, and for each transformation, we add an edge. We highlight that ETL transformation modules do not include links to the source and target metamodels which are bound at the configuration phase. For this reason, in order to run the discovery, we consider as a pre-requisite that each ETL module must include in the header section the declaration of the source and target metamodels (this is optional in ETL, but it simplifies the discovery phase). In this way, the discoverer is able to retrieve the metamodels and set the edges of the graph. This is a requirement needed to use the static analysis [64] of the ETL transformation files, which helps to find out the source and target metamodel of the transformation.

Modeling the problem in MOMoT MOMoT leverages a graph-based representation as a simplified representation of the problem, which it mutates by applying search strategies as a matter of exploring possible model variations that constitute the search space. Such a model for our purpose encompasses vital task details like discovered transformations in our repository model, the currently selected path to transform the input model, or transformation-related metrics. Therefore, we propose the metamodel reported in Figure 16.

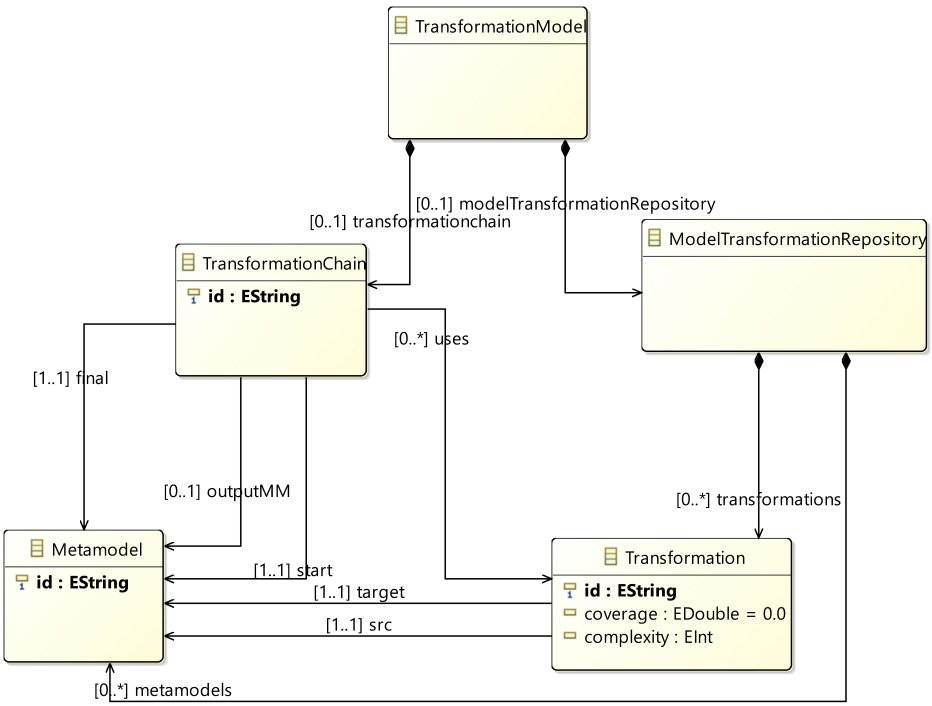


Figure 16: Metamodel for modelling the search problem.

This metamodel allows deriving representations of the transformation chain selection problem, ones which serve as input to the MOMoT tool. This, in turn, leads to the use of search strategies to achieve an optimal model evolution and, therefore, the involved transformation chain (TransformationChain). Models conforming to this metamodel are automatically created and persisted by the discoverer in ❶ after analyzing the given repository input. Indeed, this metamodel allows the creation of instances of TransformationModel in which ModelTransformationRepository and TransformationChain are contained. Consequently, ModelTransformationRepository is composed of Metamodels and Transformations. Each transformation has two references to src and target metamodel. Also, a TransformationChain can be referred to as an ordered set of transformations that will be composed to satisfy the required input, i. e., the required start metamodel (derived by the given input model) and the required final metamodel indicating the targeted transformation output, with an additional (optional) outputMM that reflects the chains current output metamodel. Consequently, for a non-empty chain, outputMM denotes the current output metamodel. It is required to realize the constraint that ensures feasible chaining solutions in that the chain leads to the output model as per user definition. Hence it is used by MOMoT in the search process to match it against

the chains final instance. As long as the discoverer has persisted in the *Repository model*, MOMoT can process it with the rest of the required artefacts.

Transformation Model for Chain Composition As part of the problem specification, transformation rules ❸ facilitate the composition of a chain as an ordered sequence of (chain selection) transformations identified between the source and the target metamodels. Note that the term "transformation" here is used ambiguously as it refers to (1) the mutation of the instance model that conforms to the domain model in Figure 16, e. g., using Henshin units [59], which we use as part of the problem specification, and (2) the domain model element (which is an EClass) Transformation (cf. Figure 16) where each instance implies a mapping between two metamodels, respectively. The former takes place with MOMoT's search engine applying rules to the model instance in order to explore its design space in terms of different chaining paths. The latter refers to possible hops within a solution to the chaining problem, which is reflected in the model by the Transformation instances referenced through uses relations from TransformationChain (*TC*). Altogether, MOMoT's solution to a task results from the changes it imposes on the input problem model by means of rule transformations. Here, those reflect a selected Transformation instance to be executed on the model which we intend to derive a chain for.

MOMoT uses Henshin [59], a graph-based transformation approach, to set the scope of possible changes for a domain model defined in EMF[17]. The toolset includes a rule-based model transformation language to induce changes by exploiting a model's graph representation and a transformation engine to execute them. Using the concepts in our problem-describing model (Figure 16), we can define patterns to match and imply changes in the model graph with so-called units and rules. Matches and the legality after the change are hereby determined through formal reasoning.

Figure 18 shows the Henshin transformation rules we defined to imitate the selection of a transformation *T* for a transformation chain *TC*. For the generation of a feasible chain, two cases can be distinguished in terms of matching semantics. Naturally, a *T* that is available in the repository can be added to *TC* with rule addTransformation. Per rule definition, *T* is limited to candidates that guarantee executability of the resulting chain i. e., *T* needs to take as input that chains current outputMM. For the first application, however, the input of *T* needs to conform to the metamodel to be transformed, i. e., source of *T* corresponds to start of *TC*, as per user declaration. Taking the chaining example from Figure 14, an excerpt of the instance model after selecting KM32EMF as first selected *T* is provided in Figure 17.

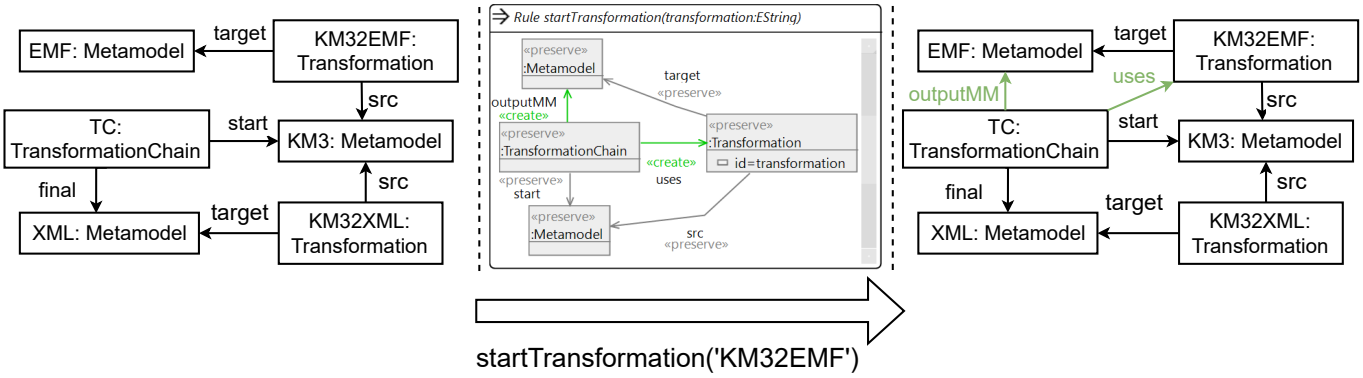


Figure 17: Henshin rule for selecting transformation *KM32EMF* and resulting model instance (excerpt).

Following the rule application, the uses reference signifies KM32EMF to be part of the chain and EMF as thereafter emerging metamodel for the chain output, while metamodels defined essentially for the chains first and last *T*, start and final, are maintained. Consequently, rule startTransformation ensures that the first candidates' input corresponds to the language of the model to be transformed. In any case, the scope for applicable subsequent candidates is established with target of the appended *T* becoming outputMM of *TC*. The described conditional behaviour is expressed with a ConditionalUnit, which applies one of the rules depending on whether a *T* is already part of the chain (rule checkHasTransformation). The same behaviour could be achieved with different control structures incorporated in other Henshin units [65].

Defining Objectives and Constraints The *Quality Criteria* that can be used are various, but to demonstrate the approach, we have chosen the ones anticipated in Section 5.3, i. e., coverage, complexity and transformation hops, and provide them with the specification in ❹. These objectives support chain selection from three different points of view, each of which adds a dimension to the fitness function that is used to evaluate derived chains.

We have used static analysis [64] of EOL and ETL (Epsilon Languages) to traverse through the elements of the used metamodels and model transformations, respectively [66] and then calculate the three structural quality criteria/objectives for the transformation chains. The transformation coverage is determined through

[17] https://www.eclipse.org/modeling/emf

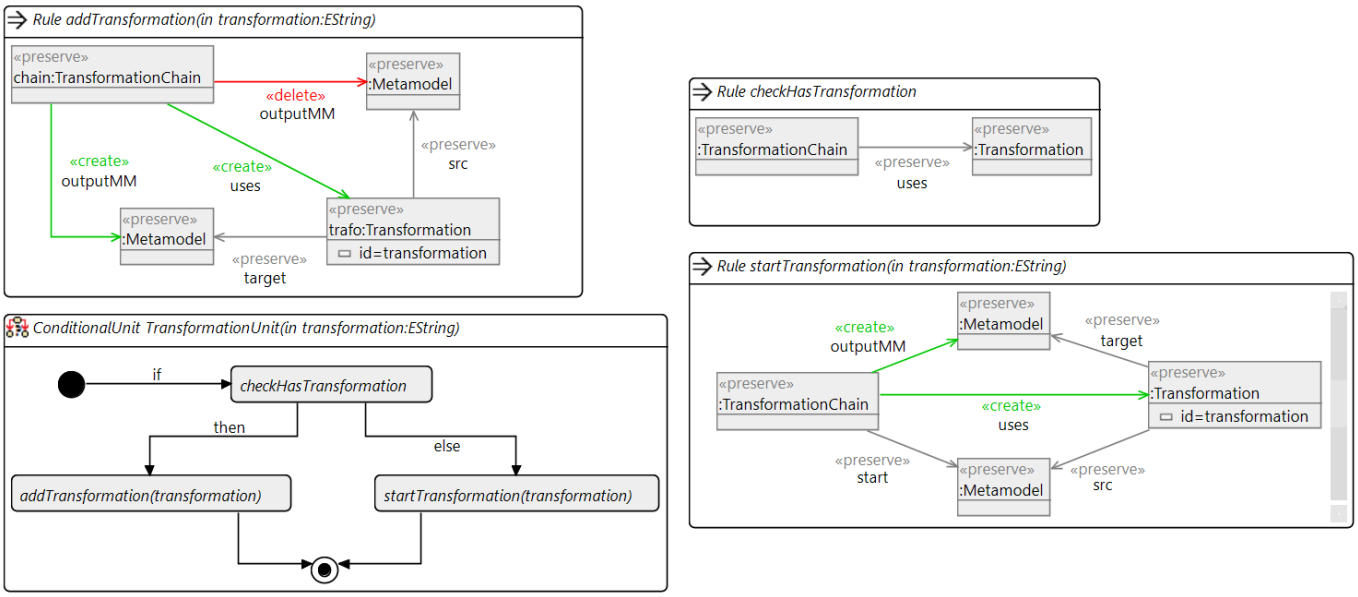


Figure 18: Model transformation defining the chaining problem in Henshin.

static analyses based on involved source and target models, i. e., srcMMCoverage and trgMMCoverage, and recorded in the intended attributes in the metamodel afterwards. It is defined in coherence with the criteria discussed by Basciani et al. in [55]. According to them, the weight considered for a transformation rule covering a metaclass in input/output is unitary, whereas the individual binding of the transformation, predicting a structural feature (in/out), weighs 0.5. Also, the transformation complexity is calculated by counting the number of constructs or elements used in a transformation module. These elements can be a predicate or reference to a metamodel element or an EOL construct such as keywords or variables. Any fitness function can be defined using a configuration language provided by MOMoT, which is based on XBase [67], a Java-like, statically typed expression language. By using this language, the user can define quality criteria/objectives as part of the fitness function, which will be evaluated on the problem model to calculate a chain's optimality. The evaluation process was executed multiple times to estimate the mentioned indicators (such as Hypervolume and Generational distance) taken from the MOEA framework based on multiple algorithms such as Random Search, NSGA-II, NSGA-III, etc.

Constraints in ❺ enrich the specification to ensure the validity in TC (i) in general and (ii) with respect to the input model to transform. Accordingly, the output of the T having been most recently added to a chain (outputMM) poses a domain-specific constraint that limits the selection for the next T to those from the repository with a corresponding metamodel source (src). Moreover, the first Ts src needs to comply with the users input model to transform, which is delineated by start of a chain TC. Likewise, a valid chain ends with a T having the final output metamodel (target) corresponding to TCs final metamodel. Remarkably, our rule definition in Figure 18 ensures that these constraints are satisfied for the chains that are eventually delivered in the result set.

Configuring the Search and Evaluation Next to the problem specification derived in previous paragraphs, the input to model-driven optimization tools like MOMoT usually entails configuring the search-based optimization process. Objective and constraint definitions elicit the extent to which a derived model reflects a desirable solution, whereas, for employed optimization techniques, parameters and evaluation metrics have to be decided to support model mutation and facilitate collective, comparative quality assessment. Therefore, problem-related specifications were previously established upon the metamodel in Figure 16 to define the search space and declare the fitness and legality of derived transformation chains. The experimental setup is now complemented with SBO-related (search-based optimization) settings (❻) to facilitate reasonable exploration of chaining selection options. User-defined parameters thereby consist of the following: (1) Algorithms and associated parameter settings, (2) Termination criterion and run, (3) Set evaluation measures and statistical test settings.

MOMoT allows to choose from a palette of generic multi-criteria approaches for targeted rule orchestration, ranging from evolutionary algorithms to local search to reinforcement learning techniques [58, 68]. In our evaluation, we let different algorithms compete against each other to demonstrate the algorithm-agnostic nature of this approach and do so multiple times to compare their performance under statistical support on our selected example. In general, the generated output contains feasible chaining solutions in terms of a Pareto-optimal solution set, so the user has to reason about trade-offs in terms of quality criteria. Our goal is to find the best chain in terms of a singular quality or a set and to learn more about the trade-offs that apply to each of the several feasible transformation chains. Therefore, we run through a fixed number of evaluations in each run and with each algorithm. Nevertheless, the framework supports

quality-based termination criteria when a particular fitness level is of interest for the desired solution. The search configuration we use is described in more detail in the following sections.

Problem Encoding and Search Orchestration As mentioned earlier, MOMoT's optimization incentive is to determine rule sequences for the problem instance model to arrive at an objectively optimal model state with respect to the fitness function. The used Problem Encoding ❼ is based on rules specifically designed to operate on model constituents of the particular domain. In our use case, each rule application represents a mapping between metamodels and acts as a decision variable as part of a solution candidate for a chain *TC*. The effective arrangement of these applications is now subject to the Search Orchestration ❽. Naturally, how new sequences are generated depends on the used methods of exploration and exploitation capabilities. A local search, for instance, is concerned with determining the nearest neighbours by adding a transformation or replacing one in the current chain to spawn new solutions. For the broadly adopted GAs (genetic algorithms), the framework initializes the population with legitimate chaining sequences at random. Moreover, the solution length in terms of the sequence of transformations (*T*) is limited to facilitate operators for alteration. In this respect, several operators for mutation and crossover are available. Note that transformation chaining poses a highly interdependent endeavour. Thus chains emerging from recombination carry a high potential of invalidity. For this reason, repair mechanisms are foreseen to restore feasibility, e. g., by replacing non-executable transformations in the sequence.

As established earlier, a validity constraint ensures the chain ends at the final output metamodel to conform to the problem specification. In fact, any *TC* concluding with a *T* having a target metamodel other than finalOutputMM represents an infeasible chaining path. These intermediary solutions, however, are maintained to be considered for further advancements and meanwhile marked as invalid to be later omitted when deriving the final solution set.

Search Analyzer and Result Manager Through monitoring capabilities at runtime, experimental setups in MOMoT are susceptible to clear-cut solution requirements and performance analysis. Information on the search process is collected and processed in the Search Analyzer ❾ to enable premature termination settings, posing the option, e. g., prioritize finding a chain that yields no attribute loss. Under consideration of multiple search optimization techniques, the MOEA framework is furthermore utilized to support performance evaluation. Chaining solutions evolved by different algorithms can be ranked using dedicated indicators like Hypervolume or Generational Distance. By default, they are computed post-search with respect to the Pareto set holding the objective trade-offs. As a result, the best searcher can be established for the chain selection task with the support of statistical tests.

Upon search termination, the Result Manager ❿ provides listings of the best found selection operations and therewith resulting output models, along with the optimal chains (*TC*) respective quality criteria. While the ordered Henshin unit instantiations leading to founding chains and objectives are provided as textual output, the transformed model conforms with the metamodel Figure 16 is persisted as (.xmi) model. Hence the chain transformation steps can be traversed programmatically and become subject to post-processing steps. This raises further options, such as the immediate transformation of a model based according to one of the found chains, visualization of identified chaining paths with further details, and additional analysis effort. Indeed, we can extract the chains and depict them and add objective annotations for each mapping/translation step. This allows for identifying bottlenecks in transformation quality, e. g., the mapping definitions responsible for most lost features/attributes.

Result Utilizing the described concepts in terms of the problem specification and with a search configuration available, MOMoT's search engine can finally be employed to determine feasible chaining solutions as a matter of evolving the model instance. The output for the scenario described in Section 5.3 (c.f. Figure 14) is shown in Figure 19. It includes the Pareto set of transformation chains produced by each algorithm and with respect to evaluated objectives, i. e., transformation coverage, complexity, and the transformation steps, i. e., hops. Note that for the coverage, due to expressing a maximization target, the additive inverse has to minimize. Therefore they are negative values in MOMoT. Moreover, the third available chain, Ch_3, is omitted from the output due to expressing a worse fitness in all respects. Apart from this, Ch_1 and Ch_2 have been determined by all three algorithms and depict optimal solutions depending on the intended use. Accordingly, Ch_2 (*KM*3 → *JavaSource* → *Table* → *HTML* → *XML*) is lower in complexity than Ch_3 whereas Ch_3 (*KM*3 → *XML*) has higher coverage and takes one transformation step only.

5.5 Experimenting and evaluating chain selection with MOMoT

In this section, we propose an evaluation of the approach based on two research questions:

- *RQ1*: Is the approach able to retrieve the best chain based on the user-defined objectives: coverage criteria, complexity and number of hops?

```
------------------------------------------------------
Results
------------------------------------------------------
Algorithm: Random
  Chain 1: KM3 -> XML
    TransformationCoverage: -0.491, Complexity: 247, Hops: 1
  Chain 2: KM3 -> JavaSource -> Table -> HTML -> XML
    TransformationCoverage: -0.002, Complexity: 173, Hops: 4

Algorithm: NSGAII
  Chain 1: KM3 -> XML
    TransformationCoverage: -0.491, Complexity: 247, Hops: 1
  Chain 2: KM3 -> JavaSource -> Table -> HTML -> XML
    TransformationCoverage: -0.002, Complexity: 173, Hops: 4

Algorithm: NSGAIII
  Chain 1: KM3 -> XML
    TransformationCoverage: -0.491, Complexity: 247, Hops: 1
  Chain 2: KM3 -> JavaSource -> Table -> HTML -> XML
    TransformationCoverage: -0.002, Complexity: 173, Hops: 4
```

Figure 19: MOMoT results to compute the best transformation chain.

- *RQ2*: How the performance of the proposed approach is affected with respect to the size of the repository and variations of the input?

In the following, we describe the experimental setup and discuss the results and threats to validity. All the experiments are run on a Windows 10 machine with 12 GB RAM that has i7-7500U CPU @ 2.70 GHz-2.90 GHz. Some of the fixed search configurations are taken as follows. The population size of the experiment is taken as 6, and the maximum evaluation is taken as 12. In order to calculate execution time, we run the experiment 20 times for each of the considered algorithms, which as Random Search, NSGA-II and NSGA-III.

Experimental Setup

In order to answer RQ1, we set up an experimental evaluation based on the ground truth established in [55]. In [55], the dataset used for the experimental evaluation is the same that we use in this paper. The graphical representation of the dataset is depicted in Figure 14. We compare the results for best chains in [55] with the proposed approach. We have used the same coverage formalization, and since the transformations used in [55] are implemented with ATL, we have re-implemented the same transformations as ETL modules. The entire corpus of transformations, models and metamodels is available on GitHub[18].

Results

The results of the first experiment that answers RQ1 are reported in Table 6.

Chain	Coverage	Complexity	Number of hops
Ch_1	0.001175	388	5
Ch_2	0.004820	173	4
Ch_3	0.49123	247	1

Table 6: Results for RQ1.

For each chain identified by the approach, the best chain, selected based on the maximum coverage value only, is chain Ch_3 with 0.49123, as in the ground truth in [55].

By also considering complexity and number of hops, the approach would consider minimum complexity with the minimum number of hops. According to Table 6, chain Ch_2 has minimum complexity (173) and chain Ch_3 has minimum number of transformation hops, i. e., 1. Since chain Ch_3 has the highest coverage and the minimum number of hops, we can assume it to be the optimum transformation chain available, considering the weights of the two quality (objective) criteria in our approach are the same. The weights are given by the user based on chain quality requirements. For simplicity, we consider them equal.

In order to answer RQ2, we have run another experiment in which we have iteratively increased the dataset from a single transformation to include the entire dataset used in the first experiment, varying the given source and required target metamodels. This assures that the size of the repository changes and the possible retrieved chains can vary too. We evaluate how the selection time for chains is affected

[18]https://github.com/lowcomote/chainselection_momot/tree/master

by also including different objective configurations. From Figure 22, it is shown that the coverage takes much longer to be computed than the complexity. The transformation coverage is a static value that can be calculated once, probably when a new transformation and related metamodels are added to the repository. Therefore, in order to reduce the overall execution time, coverage values of all the transformations are stored in a separate model and are retrieved during fitness calculation by MOMoT when necessary. This does not ultimately affect the results, but the computational costs for transformation coverages get reduced to a single execution.

The used objectives configuration (Oc) for this experiment are reported below:

Oc_1: Min(Complexity) and Min(number of hops), i. e., Min(Complexity) and Min(#hops);

Oc_2: Max(Coverage) and Min(#hops);

Oc_3: Max(Coverage) and Min(Complexity);

Oc_4: Max(Coverage) and Min(Complexity) and Min(#hops).

The results of the second experiment are reported in Table 7 and summarized in the chart in Figure 20.

	Input				Selection time in sec			
Data	**Source**	**Target**	**Available chains**	**Hops**	**Oc1**	**Oc2**	**Oc3**	**Oc4**
url1[19]	KM3	XML	[KM3,Ecore,Java,Table,HTML,XML], [KM3,Java,Table,HTML,XML], [KM3,XML]	[5,4,1]	16.23	11.35	23.84	20.95
url2[20]	KM3	XML	[KM3,Ecore,Java,Table,HTML,XML], [KM3,XML]	[5,1]	12.03	8.94	20.53	19.88
url3[21]	KM3	XML	[KM3,Ecore,Java,Table,HTML,XML], [KM3,Java,Table,HTML,XML]	[5,4]	18.39	15.49	30.84	29.52
url4[22]	KM3	XML	[KM3,Ecore,Java,Table,HTML,XML]	[5]	18.95	17.13	38.20	38.01
url5[19]	KM3	Table	[KM3,Ecore,Java,Table] [KM3,Java,Table]	[3,2]	11.22	8.29	24.07	21.77
url6[19]	Ecore	Table	[Ecore,Java,Table]	[2]	10.95	7.94	21.44	17.35
url7[19]	Ecore	XML	[Ecore,Java,Table,HTML,XML]	[4]	13.86	12.49	31.50	31.10

Table 7: Results for RQ2.

The first column in Table 7 provides the link to the GitHub repository in which the variations of the dataset are stored in separate folders. The given source metamodel and required target metamodel, influencing the column *available chains* is given in the second and the third columns of the table. The fourth column reports the set of the number of hops of a particular chain. The last columns report the chain selection time in seconds executed by our approach for the four different options of objectives. The execution time of the chain selection in MOMoT depends on transformations involved in the available chains. For example, a chain with ten intermediate transformations is detected slower in MOMoT than three chains with three intermediate transformations each. This execution time is calculated by running the MOMoT search 20 times and then taking the average value of those timings.

Figure 20 shows the execution time of MOMoT framework to run on different objective configurations considering varied datasets. The Figure 21 shows that the objective configuration Oc_3 took longer time to be executed followed by Oc_4, Oc_2 and Oc_1 sequentially. This time difference between Oc_3 and Oc_4 is that the configuration Oc_4 contains the calculation of the number of hops in an almost negligible transformation chain. This property is highlighted in Figure 21 where the average execution time for objective configuration Oc_3 is a little bit higher than Oc_4 when compared to different datasets as shown in Figure 20. In the objective configuration Oc_3, the coverage and the complexity are calculated for all the iterations in order to compute the MOMoT search. Whereas, in objective configuration Oc_4, the quick number of hops is computed within the solution model along with coverage and criteria objectives. Calculating coverage and complexity take a longer time, as it is shown in the Figure 21. This means that Oc_1 and Oc_2 takes much less time than Oc_3 and Oc_4. Also, calculating coverage is a much more expensive operation than calculating complexity, as shown in the Figure 22. The coverage and the complexity values are stored in the problem metamodel (as shown in the Figure 16), which is used in MOMoT search to find out the optimal chain based on the sets of these two objective criteria along with the number of hops in a transformation chain.

Figure 20 also highlights that datasets url3, url4 and url7 take longer execution time, followed by url1, url2, url5 and url6. The difference in their execution time is a result of the varying number of hops between all the possible chains from the source metamodel to the target metamodel, as shown in the Figure 14. In this figure, it is shown that there are three possible transformation chains with 5, 4 and 1 transformation hops. Therefore, considering the number of iterations run by MOMoT, it is evident that chains with 5 and 4 hops (such as in url3, url4 and url7) take approximately equal or longer time to execute than the other set of transformations. This is because, in each iteration used in an algorithm in MOMoT, there has to be a multiple counting of the objective criteria for each of the numerous transformations in the chain. Also, it can be explained that the other URLs, such as the one with 3, 2 and 1 hop, takes lower execution time as compared to the higher number of hops within a transformation chain. Thus, it is evident that the longer transformation chain would take more execution time, and it doesn't depend on the number of possible chains that transform the source model to the target model. The size of such a possible chain is shown in

[19]https://github.com/lowcomote/chainselection_momot/tree/master

[20]https://github.com/lowcomote/chainselection_momot/tree/master2

[21]https://github.com/lowcomote/chainselection_momot/tree/master3

[22]https://github.com/lowcomote/chainselection_momot/tree/master4

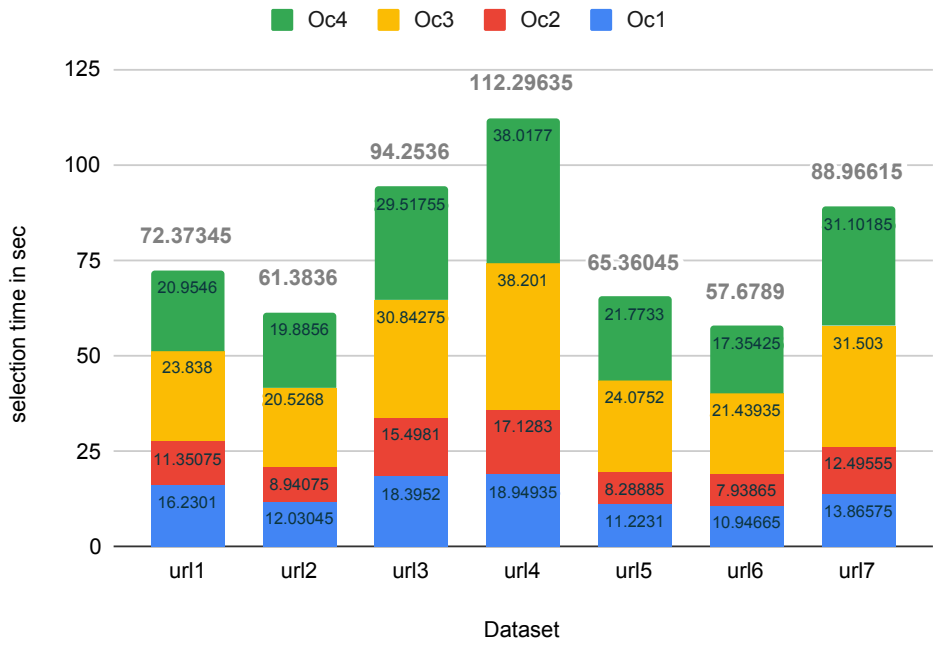


Figure 20: Execution time in MOMoT.

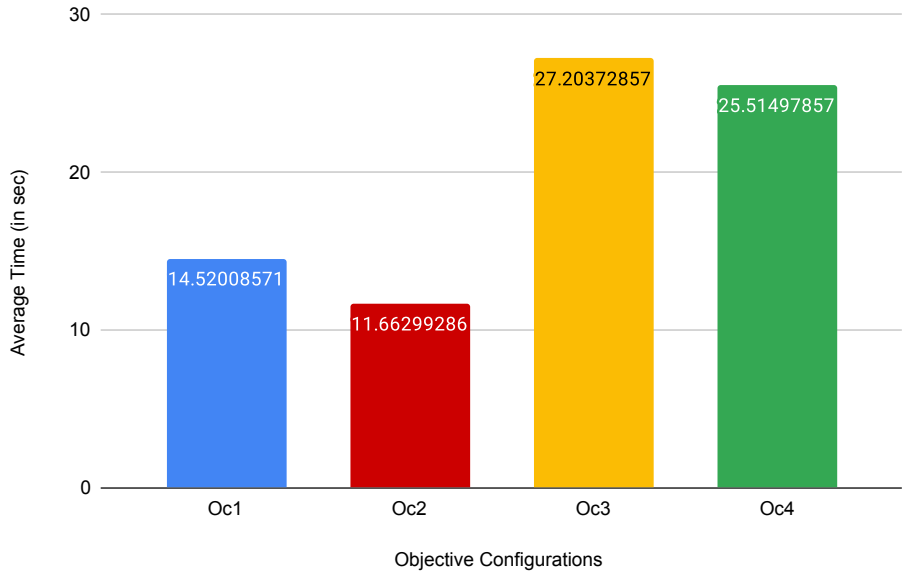


Figure 21: Average execution time for objective configurations.

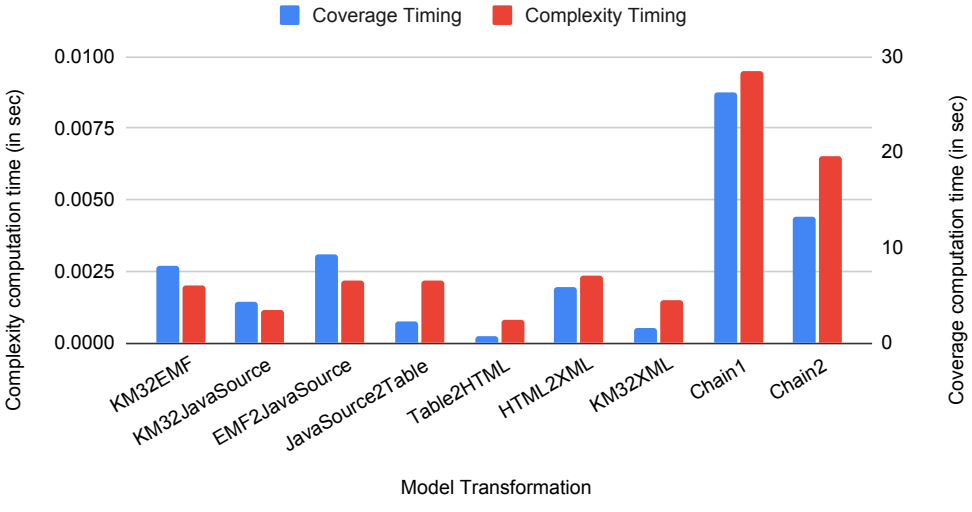


Figure 22: Chain Complexity and Coverage Computation Time.

the second column of the Table 7, which indicates a varying execution time without any relation to the known size of the possible chains.

Threats to validity

In this section, we discuss the threats to the validity of our experiments by distinguishing them between internal and external.

Internal Validity Such threats are the factors that could have influenced the final results of the performed experiments. We attempted to avoid any bias in the definition of the quality criteria, we have used them since they can influence the chain selection. To mitigate this possible threat, we have considered the same coverage formalization as in the ground truth paper [57, 55]. Moreover, we have built our dataset by translating the existing transformations in the dataset in [55] from ATL to ETL. This can include different transformation constructs used by the two transformation languages, e. g., operations in ETL vs helpers in ATL. We think that using only declarative aspects of both the transformation languages would limit this threat of varying quality criteria between ATL and ETL files. However, we have written almost syntactical equivalent ATL constructs into the ETL file.

We are aware that we have used a relatively small dataset of transformations, but it is based on real transformations with very different complexity, definitions, constructs and domains of application. We have also tried to mitigate this aspect by applying a mutation of the dataset, which is the result of changing the required input and artefacts included in the repository. The datasets need to be expanded, as well as further objectives can be added, such as model coverages, similarities between the metamodel, etc. These parameters for MOMoT would be tested to identify the importance of intelligent algorithms such as NSGA-II and NSGA-III over the Random descent algorithm.

External Validity The external validity discusses whether we can generalize our results. The first aspect we need to highlight is that the experiment has been based on an existing dataset of transformations, which has been rebuilt starting from a snapshot of the ATL transformation zoo [69]. The second aspect we need to discuss is that our approach is based on ETL transformations static analysis, but the approach is completely generalizable to other transformation languages on which quality criteria can be defined. To confirm that in the first experiment, we have compared the coverage-based selection with the paper in [55], where the transformations were defined with ATL. We mitigated this aspect by also having modularized the static analyzer of the transformation that can be replaced with another one in order to create the repository model. Moreover, in our approach, the static analysis[23] operates with a pre-requisite that the ETL module must contain the source and the target metamodels' linked in the header of the transformation. This avoids extra operations to retrieve the graph of the possible chains. We think that this threat is not influencing the results of the experiments since, by using the approaches presented in [63, 70], the recovery of the existing chains is possible with multiple technologies.

5.6 A motivating example of model transformation chain optimization

In this section, we introduce the tools and the languages used for the demonstration of the proposed approach. Model transformations are the heart and soul of MDE. Model-to-model (M2M) transformations transform a source model into a target model in the same or different abstraction level, whereas model-to-text transformations transform models into source code or generically text. In this paper, we concentrate on M2M transformations.

ETL [24], is the transformation language for model transformation tasks provided by the Epsilon family [5]. ETL takes in a number of source models and transforms them into a number of target models. An ETL module can contain a number of transformation rules which transform a source model element to one or more target model elements. An ETL module can also have pre and post-block to be executed before and after the execution of transformation rules, respectively. Every rule is composed of an internal body in which the user can specify the *bindings*. Binding is used to set the features of an instance transformed by the current rule by using the values of the features of the source instance. This aspect is mostly supported by the declarative nature of the transformation language, even if the user can also use imperative constructs. In order to support reuse and maintainability, model transformations can be composed as small transformation modules in order to get a larger transformation. The composition can be external or internal. External composition deals with composing model transformations together by passing models from one transformation to another. Internal composes two model transformation definitions into one new model transformation, expressed in the same transformation language [71, 72]. The external composition can be enabled if some pre-requirements are respected, e. g., the target metamodel of a transformation T_1 is contained, or it is the same as the source metamodel of T_2.

[23] https://github.com/epsilonlabs/static-analysis

[24] https://www.eclipse.org/epsilon/doc/etl/

In general, the steps executed to compose transformation chains are multiple [55]. The input model and the required output metamodel are given as user input. In order to get the required output model, the activities are (i) identification of available transformation chains, (ii) selection of one of the chains (by considering user-based preferences or quality parameters), and (iii) execution of the entire chain of transformations.

In a transformation chaining scenario, as the one represented in Figure 23, we have the criteria to chain the three existing transformations, i. e., *A2B* and *B2C* satisfied, giving place to a chain $C_1 = A2B \rightarrow B2C$.

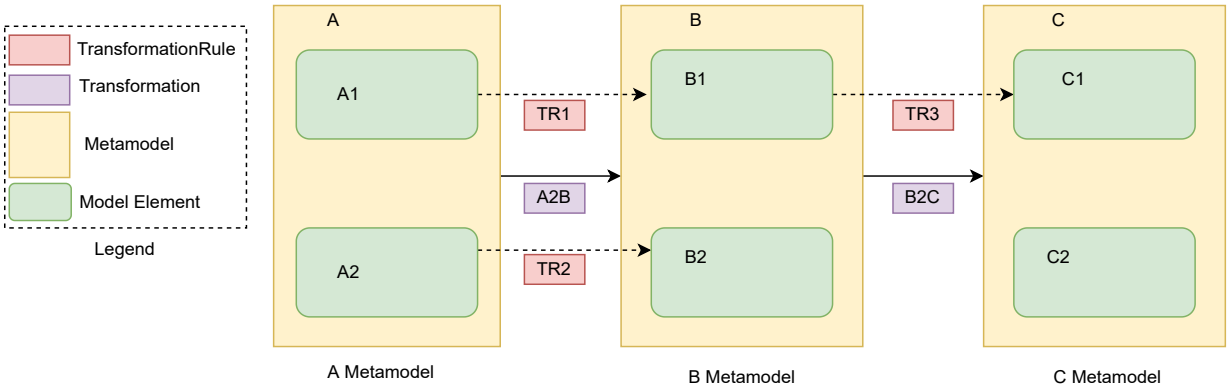


Figure 23: An example model transformation chain.

Transformation *A2B* is composed of two transformation rules *TR1* and *TR2*.*TR1* transforms metaclass *A1* instances into *B1* instances, while *TR2* transforms *A2* instances to *B2* instances. Transformation *B2C* has one transformation rule *TR3* that transforms *B1* model elements to *C1* model elements. Now, if we analyze chain C_1 from start to end, we can notice that rule *TR2* is generating *B2* model elements which are not propagated in the final transformation *B2C*, since there is not a specific rule matching B2 elements.

In this case, the available chain is only one, i. e., C_1, then given a model conforming to metamodel A, the chain can be executed to get in output a model conforming to metamodel C. If we execute the chain by composing the two transformations, all the rules and bindings in the transformations will be executed by trying to match all the elements declared in the transformations chain, even if they are not propagated by the intermediate or following transformation. For instance, in this case, the first transformation will execute rule TR2, producing elements, even if in the following transformation TR3, elements of type B2 are not considered nor propagated. This might cost an unnecessary burden in terms of execution time, especially in larger chains. This case is quite trivial for sake of simplicity, but it could be the case of complex transformations, in which multiple rules are declared, and multiple transformations are composed. This leads to the need for an optimization phase that can be performed before running the chain, and we propose this in the following section.

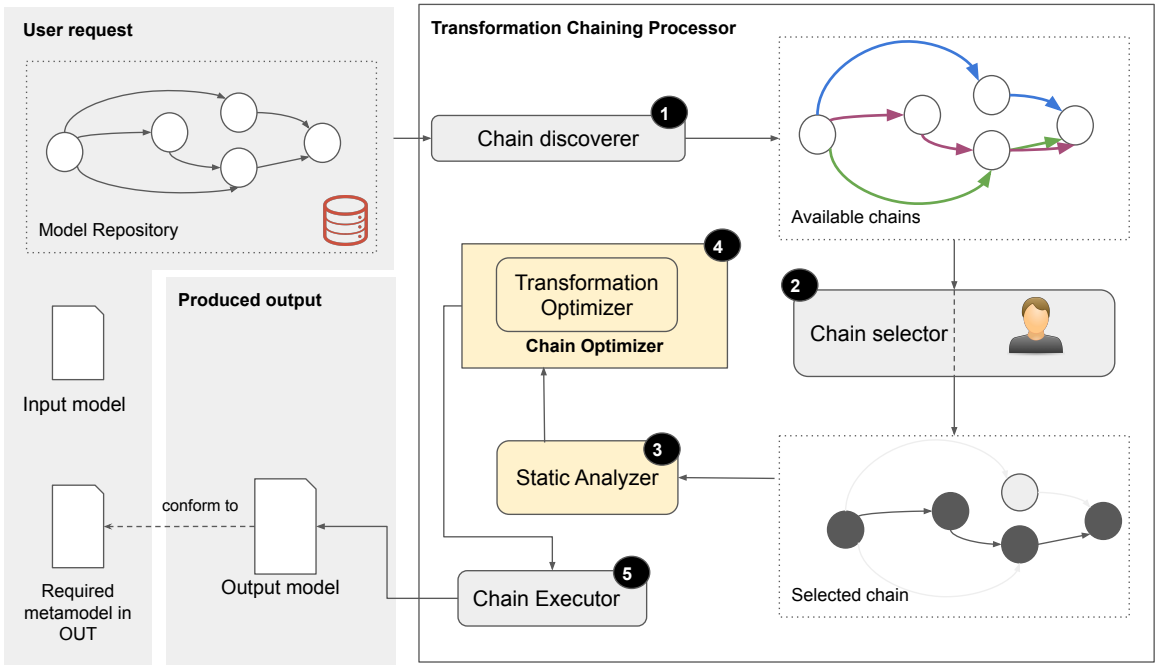


Figure 24: Proposed approach.

5.7 Approach: Chain execution optimization with Epsilon

In this section, we propose an extension of the architecture proposed in [55] and illustrated in Figure 24 that is able to optimize the selected chain of transformations before running it. The expected benefits for this application are in terms of memory allocation and, thus, reduction of the execution time.

The architecture proposed in [55] is composed of the components in grey, whereas the yellow ones have been added to support the optimization. The user input is composed of three elements: 1) a model repository (that might also be a local folder containing modelling artefacts, 2) an input model and a 3) required output metamodel. The *Chain Discoverer* ❶ is the component that, given the user input, explores the graph-based representation of the repository looking for available transformation chains satisfying the user request. If multiple chains are available, the *Chain Selector* selects one of the available chains by using user preferences, criteria or directly the user selection. If only one chain is available for that request, ❷ is skipped. When the approach has the selected chain, it can be executed to return the model in output, but a new component is then invoked, i. e., the *Chain Optimizer* ❹, that is in charge of optimizing the selected chain using the *Static Analyser* ❸ and pass it to the *Chain Executor* ❺ that will run the chain and save the output model.

In the following, we detail the optimization phase and explore the components used for this purpose. The optimization that we propose executes only those transformation constructs that are needed and propagated to the intermediate model elements, which are required to generate target model elements. This concept is illustrated in Figure 23, where we showed that rule TR2 is not needed since the chain at the second step does not consider the target metaclass of that rule. This concept can be easily extended to internal bindings of the rules. The required rules are derived based on the extracted typed information from the *Static analyzer* component ❸. The main idea is to analyze the transformation and then the entire chain from the initial source to the final target. The static analyzer is a helper module based on Java and Epsilon, which allows interaction with a model transformation as a model, so it can be queried and managed by using EOL scripts. For every transformation rule, it is checked if the target parameter(s) of the source model is the source parameter of any rule in the next intermediate or target model. Only the matched transformation rule is chosen to be the required rule. Otherwise, that transformation rule is removed from the transformation to speed up the execution. We can extend this logic to every statement of the transformation rule in which the transformed reference and attributes of a particular metaclass are checked to be present in the next transformation, i. e., a binding. Alternatively, if the transformed element is not present in the next transformation, then that particular statement of the rule is deleted.

The static analyzer first invokes an algorithm, presented in [73], in charge of building the dependency graph between the rules based on the equivalent(s) operator used in a statement of the considered rule. The equivalent operator is a built-in operator of ETL which automatically resolves source elements to their transformed counterparts in the target models. The output of this algorithm gives the HashMap in which the *key* contains all the rules in a given transformation while the *values* contain the dependent rule(s) of the corresponding keys(rules).

The HashMap output of this Algorithm is the input for the Algorithm 1 that traverses each statement of the given rule in the transformation file.

This algorithm checks the target bindings of a statement in a transformation and compares it with the source bindings of the next transformation. If the target binding (in a transformation) matches with the source binding (in the next transformation), we can store it in `RulesToKeep` array. Otherwise, we can store it in `RulesToDelete` array. The Algorithm 1 now checks the *values* given in the `HashMap` and compare it with the values stored in `RulesToDelete` array. If they are equal, then the reference of the current rules from `RulesToDelete` array will be removed from it. Otherwise, the current rules are not dependent on any other rule; therefore, such a rule can be part of the `RulesToDelete` array so that it can later be deleted to optimize the overall transformation chain.

One challenging thing here is that sometimes some rules would be altering the required model elements. The use of "equivalent" expression in the Epsilon Transformation Language would refer to a transformation rule that shows some dependency between two or more rules of a model transformation. To handle this issue, we propose to construct a dependency graph to find out such rules and will perform the program rewriting after analyzing the dependency graph.

The logic of this optimization stage is shown in the algorithm in Algorithm 1. This algorithm checks if a model element (attribute or reference) in a particular input pattern or binding of a transformation rule of the current transformation is used in any transformation rule of the next transformation. Once the chain is optimized, we calculate the number of bindings in the rules of the optimized transformation. If there are no bindings in the rules of an optimized transformation, then that rule is deleted from the optimized transformation. If the binding in the current transformation is not used in the next transformation, then this binding will be deleted as it is unused for that specific couple of transformations. Once all the unused bindings from transformations are removed, we have the optimized transformation chain, which will be executed to retrieve the same result with lesser execution time.

Finally, the rewritten transformations containing only the required transformation rules are executed within a chain of model transformations.

Algorithm 1 Pseudocode for optimization of transformation chains.

```
1: procedure OPTIMIZE(HashMap hm)
Require: DG = Dependency Graph given in FindDependencyGraph(a)
2:     Let a = chain containing transformations a0, a1, ..., an
3:     Traverse each transformation in the given chain a
4:     Take consecutive pairs of transformations from the end to start of the chain (an, an − 1)
5:     Source = an − 1
6:     Target = an
7:     BindingsToDelete, BindingsToKeep  ▷ Arrays for keeping the bindings that need to be removed and
   the bindings to keep, respectively
8:     for all rule=rules in Source do
9:         for all binding=bindings in rule do
10:            ref_type = EReferenceType of binding
11:            TP = types of target parameters of the binding in the rule in transformation a(i − 1)
12:            SPs = types of source parameters of the binding in the rule in the transformation ai
13:            dependent_rule = values given in key rule stored in HashMap hm
14:            if SPs equals TP then
15:                add binding in BindingsToKeep
16:            else
17:                if ref_type = dependent_rule then
18:                    add binding in BindingsToKeep
19:                else
20:                    add binding in BindingsToDelete
21:                    remove binding from Source
22:        if (#binding in the rule = 0) then
23:            delete rule
```

5.8 Experimenting and evaluating model transformation chain optimization

In this section, we evaluate the approach by answering the following research questions:

- *RQ1:* Is the approach able to produce correct results w.r.t. non-optimized chains?
- *RQ2:* Is the approach effective in optimizing the execution time of the available chains varying model size or transformation chain hops?

Experimental Setup

The experiment is based on a case study borrowed from [55], *KM3* → *XML* and reported in the dataset on GitHub[25]. The case study is composed of 6 metamodels and 7 transformations. The user request is made of given a KM3 model and requests an XML model as output; the repository is represented in Figure 25.

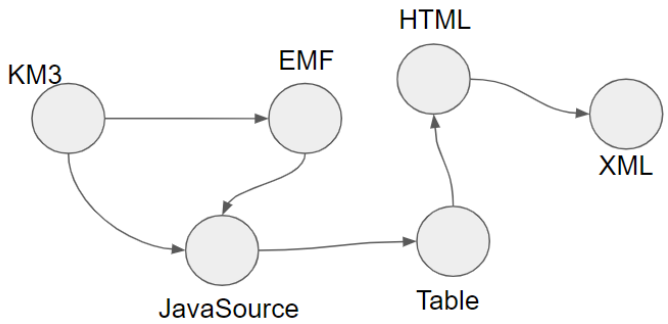


Figure 25: Graph-based representation of the KM32XML experiment.

We have executed the approach on this case study by providing the user request, i. e., the initial model and required metamodel, as well as the repository of the case study. We executed the available chains 10 times and measured the average of those runs. We reported all the execution times and calculated the entire chain execution time with/without the optimization component. If more than one chains were available for

[25]https://github.com/lowcomote/chain-optimisation

the case study, we calculated all the possible chains for both versions, optimized and unoptimized. If the results showed an improvement in terms of execution time, we compare the resulting models produced by the two executions in order to check if the models are exactly the same. This would confirm that the approach is able to optimize the execution based only on the removal of unneeded constructs of the transformation. This demonstrates the correctness of the implemented optimization algorithm. In order to compare the output models, we have used EMFCompare [74] to automate the tasks, and we inspected the results to confirm that the two models are exactly the same. If the models are the same, we can check the execution time. Otherwise, we mark the result of the transformation chain as unexpected. Also, the cache memory to load the EMF models has not been considered in order to exclude possible wrong evaluations based on caching features. The models used for the experiments are original models provided by the case study and a set of randomly generated models. The generated models are obtained by using EMG [75], which is the Epsilon Model Generation language supporting the semi-automated generation of models. With this tool, it is possible to create instances of metaclasses in the metamodel(s), assign values to the instance's attributes, and create links between instances to assign values to references. We generated 50 models for each case study to have in total of 102 models to test the approach with different-sized models. The experiments are run on a Windows 10 machine with 12 GB RAM that has i7-7500U CPU @ 2.70 GHz-2.90 GHz.

		Execution time			
		Unoptimized		**Optimized**	
Input model	**Size**	Ch_1	Ch_2	Ch_1	Ch_2
original_4	788	5.99	0.49	5.81	0.39
generated #1	1,150	10.76	3.36	8.90	3.13
generated #2	2,300	15.69	3.84	14.14	3.02
generated #3	3,450	19.98	4.10	18.61	4.47
generated #4	4,600	26.85	4.57	24.71	3.30
generated #5	5,750	36.433	3.96	33.69	4.61
generated #6	6,300	48.92	5.20	44.32	4.0
generated #7	7,450	61.0	4.83	58.21	4.25
generated #8	9,200	70.79	5.44	62.97	4.15
generated #9	10,350	91.79	5.62	87.03	4.39
generated #10	11,500	110.22	5.36	85.23	4.95
generated #11	13,800	116.43	5.29	103.83	4.37
generated #12	16,100	139.87	7.06	133.32	6.10
generated #13	18,400	133.07	5.82	130.85	5.21
generated #14	20,700	262.09	6.15	248.66	5.98
generated #15	23,000	221.18	7.94	216.53	7.57
generated #16	27,600	309.70	10.24	292.42	9.05
generated #17	32,200	416.54	11.17	392.56	10.81
generated #18	36,800	882.66	13.73	815.137	10.99
generated #19	41,400	1,033.70	16.53	1,005.30	13.73
generated #20	46,000	1,379.26	15.71	1,283.45	14.70

Table 8: Results for the KM32XML experiment.

Results

In this subsection, we discuss the obtained results of our experiments by graphically reporting the results for the first model (original) used as input, and the complete results are reported in Table 8. First of all, the resulting models with the optimized and unoptimized components resulted in exactly the same, positively confirming RQ1 and letting us proceed with the evaluation of the results for RQ2.

The experiment for the original model is reported in the first row of Table 8, and based on the user input, two available chains are returned by the algorithm:

CH_1: *KM3* → *EMF* → *JavaSource* → *Table* → *HTML* → *XML*

CH_2: *KM3* → *JavaSource* → *Table* → *HTML* → *XML*

Chains C_1 and C_2 have been executed for 20 model sizes. The optimized chain C_1 gives better results (less execution time) than the normal chain. There are some exceptions in calculating the optimized chain (C_2) compared to the normal chain. However, the average and median values of the optimized chain would be less than those of the normal chain.

We reported the execution time for the entire chains on the last node as well. The result confirms a 6.06% reduction in execution time on C_1 and 10.99% on C_2. The results confirm that the approach has been able to optimize the execution time, still producing correct models on the output, confirming the conclusions foreseen for RQ1 and RQ2.

Threats to validity

In this section, we discuss the internal and external threats to validity.

Internal threats Internal threats are aspects influencing the results of the evaluation. One of the aspects that need to be mentioned is that not all the transformation constructs are considered in the analysis. The `operation` in ETL files are not considered for optimizing the transformation chain. Moreover, executing transformation chains on a single machine can be influenced by other tasks in execution. For this reason, we executed the chains for both the unoptimized and optimized version 10 times with multiple runs, and we used the average of the results. As optimization is defined as the use of structural elements in the binding of the transformation to the binding of the next transformation, it is henceforth evident that there won't be any optimization case for the last transformation in a chain or even in the case of the direct transformation. The dataset of models used for the experiment is composed of 50 models, which could be seen as a limited size for a dataset, but it is quite hard to find online resources in which we have model transformations and available chains of them. For this reason, we tried to mitigate this threat by randomly generating models from a given seed model. Future work will include a more extended experiment with other chains and a bigger generated dataset of input models. The larger datasets need to be used to compare the execution time of the optimized and unoptimized transformation chains.

External threats The external factors influencing the conducted experiment's validity outside the used setting are multiple. We tested the approach on the Epsilon framework, and specifically with ETL transformations, but the generalizability of the approach is based on the fact that ETL is a declarative rule-based transformation language, and thus all transformation languages falling in this category are candidates for applying this approach. The static analyzer must be re-implemented in order to be able to analyze other types of transformations, e. g., ATL.

5.9 Related work

This subsection elaborates on the related works on the composition of model transformation, quality criteria in model transformation, search-based approaches in model transformation and how the execution of model transformations was optimized.

Model transformation composition approaches

Basciani et al. (2018a, 2018b) [55, 57] took user input as source model and target metamodel. It can be described as retrieving source metamodel and detecting all the available chain transformations. Then, find out the best chain transformation by calculating the optimal coverage of every chain transformation and information loss in a customized Dijkstra algorithm for every chain transformation. The processes in this method are (i) finding the source model and target meta-model. (ii) Finding available transformation chain lists. (iii) Select the optimal chain and execute it. Steps (ii) and (iii) comprise Model Transformation Composition Language (MTCL).

Basciani et al. [76] took user input as the source model and target metamodel. It can be described as retrieving source metamodel and discovering all possible chain transformations. It checks if the source and target metamodels are incompatible, and then an intermediate adapter is automatically generated to fill the gap between the inconsistencies between the metamodels. The processes in this method are (i) finding the source model and target meta-model. (ii) Build MTCL by creating an intermediate adapter between incompatible metamodels.

Etien et al. [77] took user input as very large models based on UML, Ecore, etc. It can be described as decomposing the models based on the separation of concerns and then using localized transformation to check the desired outcomes according to the objectives of the application. The processes in this method are (i) finding out the granularities of the large model. (ii) Build localized transformations and combine those transformations with the help of MTCL.

Aranega et al. [78] took user input as large models. It can be described as preparing feature models by dividing the business logic of a group of elements of a model. These feature models are used to automate the consistent set of model transformations and generate an executable chain of model transformations to implement the desired objectives. The processes in this method are (i) finding out the granularities of the large model. (ii) Build MTCL for a consistent set of model transformation chains.

Etien et al. [71] took user input as model transformation chains. It can be described as determining which chaining of the model transformation gives the desired result by determining pre-conditions, post-conditions, and behaviour of individual rules of different model transformations. Commutativity of the chaining of model transformations is also used to detect identical results by using both sides of the transformation. The process of this method is to find out the best possible model transformation chain.

Etien et al. [79] took user input as a model. It can be described as combining independent model transformations that jointly work to achieve the same objective that does not handle compatible source and target metamodels. The process in this method is to build MTCL for independent model transformation with incompatible metamodels.

Wagelaar et al. [80, 81] took user input as two model transformation languages (ATL and QVT-R). It can be described as proposing an internal composition technique called model superimposition that allows for extending and overriding rules in different transformation modules that provide executable semantics and proper implementation in one of the model transformation languages. The process in this method is to build the internal composition of model transformation.

Chenouard et al. [82] took user input as model transformation chains. It can be described as automatically discovering some more detailed information so that the actual complete chaining constraints can be fulfilled by statically analyzing transformation. This method aims to find out the best transformation chain by statically analyzing the transformation that comprises MTCL.

Rivera et al. [83] took user input as models and model transformations. It can be described as introducing a graphical executable language for orchestrating ATL transformation to modularize the transformation composition based on some mechanism and execute the chaining of model transformation. The process in this method is to find out proper mechanisms such as conditional, parallel, and looping of transformation composition for identifying the appropriate output model known as Orchestration Engine.

Vanfooff et al. [84] took user input as metamodel. It can be described as proposing metamodels for a transformation chain modelling language that enables the implementation-independent composition of transformations in the concrete syntax based on the UML activity diagram. This composition of transformation chains can be applied to the models used to implement a concrete implementation of the desired result. The process in this method is to create metamodels that support transformation chains modelling language that comprise an MTCL.

As a future goal, a parallel distributed cloud-based environment is envisioned to perform the model transformation and its composition. For a system to be scalable, all the involved artefacts, such as model transformations, need to be available in a distributed manner so that a user can efficiently access any reusable artefact according to their needs from remote locations.

Quality criteria in model transformation

Syriani et al. [85] elaborated on the design pattern in the context of model transformation that satisfies quality criteria identified before the execution of the transformation. The quality criteria identified to assess and validate the model transformation design pattern are correctness, re-usability, efficiency, reliability, maintainability and interoperability. The verification and validation done on the design patterns allow for assessing whether the catalogued design patterns are complete with respect to the quality criteria. This helps to detect bad design and improve the design pattern of a model transformation.

Selimet al. [86] elaborated on transformation testing, which is used to estimate the quality criteria of the model transformation. Some of the estimated quality criteria which are considered for transformation testing are metamodel coverage, input contract coverage of the model transformation, etc. These quality criteria are calculated using mutation analysis which computes the value of quality criteria of the original model and compares it with the value of quality criteria when the model is mutated or changed.

Bauer et al. [87] presented a coverage analysis approach to measure the test suite quality for model transformation chains. Their approach combines different coverage criteria such as class coverage, attribute coverage, association coverage, feature coverage and transformation contract coverage. The combination of these coverage metrics gives detailed coverage information that is used to identify missing and redundant test cases of model transformation or model transformation chains.

Erginet al. [88], a new model transformation design pattern is introduced that improves the quality of model transformation. The design pattern focuses on three quality metrics of the transformation. They are the number of rule applications, the size of the rule and the number of auxiliary elements. These three metrics are related to the efficiency quality criteria, and therefore, improvements in these metrics would lead to the reduction of time complexity. This paper finds out that the normal usage of these metrics would lead to quadratic time complexity while the improved solution would lead to linear time complexity.

Mkaouer et al. [89] elaborated on the objectives of the model transformation, which are to provide rules that generate the target model without any error and to minimize the complexity of the transformation rules (by reducing the number of rules and bindings in the same rule) while maximizing the quality of the target models. This paper focuses on the transformation mechanism as a multi-objective problem in order to find the best rules that maximize the target model quality and minimize the rule complexity. The quality

of the transformation rules and bindings is iteratively improved by using the multi-objective optimization process. The objectives are the number of rules and matching metamodels in each rule and assessing the quality of generated target models using a set of quality metrics. An optimization algorithm such as NSGA-II is used to automatically generate the best transformation rules satisfying the two conflicting objectives. By achieving the best possible solution with two conflicting objectives, the paper claims to provide a well-organized target model with a minimal set of rules.

Basciani et al. [55], two quality criteria such as transformation coverage and information loss, are considered in a model transformation chain scenario where multiple chains are possible between the source and the target model. A customized Dijkstra algorithm is used to individually consider the two criteria that consider the best chain. Since the information loss is considered in the model generated, the paper claims that information loss is a better quality criterion as compared to the transformation coverage, which is considered by calculating the static element in the metamodel and the transformation without considering the generated models.

Search-based approaches in model transformation

Kessentini et al. [90] framed a transformation technique as a combinatorial optimization problem where the end goal is to find a better transformation that starts from a small set of available examples. The search-based model transformation by example has been also further elaborated [91]. This approach is called model transformation as optimization by example (MOTOE) that combines transformation blocks extracted from examples in order to generate a target model. A modified version of particle swarm optimization (PSO) is used where different transformation solutions are modelled as particles which exchange transformation blocks to converge to achieve an optimal transformation solution.

Fleck et al. [92] identified the problem of modularizing a model transformation program and using it as a model in an automated search-based approach. The application and the execution of the problem are managed by the search framework that combines an in-place transformation language (in Henshin) and using a search-based algorithm framework. This calculates the Pareto-optimal solution based on four objectives or quality attributes. The four objectives are the number of modules in the transformation, the difference between the lowest and highest number of responsibilities in a module, the cohesion ratio and the coupling ratio. This approach uses MOMoT framework that can be used to model a problem, and by using in-place transformation and search-based algorithm (such as NSGA-II, NSGA-III, etc.), a Pareto-optimal solution is found based on minimization or maximization of the quality objectives.

Sahin et al. [93] proposed to handle model transformation testing as a bi-level optimization problem which combines the generation of test cases with mutation testing. This paper divides the problem into two parts: the upper level and the lower level. The upper-level problem generates a set of test cases that are used to maximize the coverages of metamodels used and the errors introduced by the lower-level problem to the transformation rules. This bi-level formulation of the problem provides a statistical analysis of the obtained result that shows the competitiveness and the outperformance of the proposed approach as compared to the precision over co-evolution and non-search-based methods.

Optimization of model transformation

Jordi Cabot et al. [94] propose a metric to measure the complexity of the OCL expression. The metric is based on the syntactic structure of the expressions (number of referred attributes, number of navigations, etc.) and on the constructs used in their definition (such as the number of forAll and select iterators). This traversal of each expression to determine the number of objects involved in calculating the expressions aims to give a precise complexity of the execution of an OCL expression.

Wimmer [95] have proposed an approach for parallel execution of model transformations along with some optimisations both at the transformation level and at the OCL expression level using static analysis. First, ordering of matched rules to identify the matched rule sooner. Second, the footprinting of the transformations filters the model elements by only keeping the ones that are matched with any rule. Third, handling bindings to be resolved at compile-time to speed up the execution at run-time. The fourth is to make trace links reduction by only keeping track of those links that are to be used during the transformation. This approach doesn't take chain optimisation into account.

An approach compiles a subset of ATL code to generate efficient Java code [96]. A similar approach [97] to enable incremental execution of model transformations uses partial evaluation to pre-compute a part of model transformation.

VIATRA [98] provides an incremental engine based on the RETE algorithm. The incremental engine of VIATRA computes the pattern matches and caches them, thus, executing the transformations efficiently. But due to caching the memory consumption is high so you get a faster execution at the cost of more memory consumption.

6 Conclusion

In this deliverable, we introduced low-code development platforms (LCDPs) as the next-generation development environments. These environments employ the recent practical advancements of MDE, with the benefit of using models at higher abstraction levels to define complex software systems as fully operational applications. We focused on model transformations as underpinning technologies and their requirements for scalability and quick response time to meet the users' needs.

In Section 2, we introduced SparkTE, a multi-paradigm, distributed model transformation engine in Spark. SparkTE is based on Spark and allows users to express transformations with different approaches, each having its benefits and drawbacks. We covered three main topics: (i) how expressions can be evaluated in transformation rules, (ii) what semantics can be used and implemented to conduct a model transformation, and (iii) what engineering choices in the design space of a distributed model transformation engine.

In Section 3, we presented a benchmark framework to measure the performance of multi-parameter Spark applications and find the best parametrization according to several goal metrics, e. g., execution time, memory or disk use, network traffic, etc. We introduced the benchmark workflow and demonstrated its applicability to a running example application. We measured the execution time and the memory use on a cloud infrastructure (G5k) and found the best parametrization of the application in the given context. The prototype implementation of the framework is open-source and available on GitHub[26].

In Section 4, we presented a cloud-based model transformation, validation and verification workflow to check the correctness of industrial systems engineering (SysML) models. The workflow adopts the hidden formal method approach, i. e., (*i*) users provide the input models and the verifiable (checkable) property in the SysML language they are familiar with; (*ii*) the details of the transformation and the formal verification are hidden from the users; (*iii*) the verification results are returned in the original domain, on a SysML sequence diagram. We demonstrated the approach and measured its scalability in the cloud (Amazon Elastic Kubernetes Service) on a running example and an industrial model.

In Section 5, we elaborated on approaches for identifying, selecting and optimizing model transformation compositions. These steps enable the efficient composition of transformation chains by combining several more minor transformations that are available in the model repository.

Future work

In future work for SparkTE, we will evaluate all the proposed features and their combinations. The purpose will be to show how the combination of the different strategies can enhance, or diminish, the performance of computation. The study will be based on two main criteria: computation time and memory use. Ideally, we want to highlight that there is no single perfect configuration, but it depends on the use case, the execution environment and the characteristics of input (e. g., model size, topology, and rule complexity).

We are planning to extend the multi-parameter benchmark framework to make it able to run the measurements in parallel on different machines in the cluster. Thereby speeding up the overall execution time of the benchmark workflow. Besides, we will experiment with different applications and advertise the framework for a broader audience in the MDE and software engineering communities so that other software engineers can also benefit from our work.

We will improve the performance of the cloud-based model transformation, validation and verification workflow. One way to achieve this goal is to adopt the parallel-reactive model transformation proposed in [9, 11]. It enables live, incremental model transformations, i. e., if the source model changes, then only the impacted parts of the target model will change, which results in a shorter transformation time. Another future work is to extend the supported set of SysML state machine and activity diagram elements to be able to verify more complex models that are useful for systems engineers.

The presented model transformation composition method should be used on a larger dataset so that many intelligent algorithms, such as NSGA-II, can find the best transformation chain. Also, several larger models could be tested in order to compare the optimized and unoptimized transformation chains. These approaches could be applied in a cloud-based model repository where the transformation chain can be structured as a graph, and upon deletion or addition of any metamodel or model transformation, the newly optimal transformation chain could be identified and selected. The graph data structure in the cloud-based repository could also help manage chains of transformations on larger models.

[26] https://github.com/lowcomote/multi-parameter-benchmark

References

[1] Douglas C Schmidt. Model-driven engineering. *Computer-IEEE Computer Society-*, 39(2):25, 2006.

[2] Mariano Belaunde, Cory Casanave, Desmond DSouza, Keith Duddy, William El Kaim, Alan Kennedy, William Frank, David Frankel, Randall Hauch, Stan Hendryx, et al. Mda guide version 1.0. 1, 2003.

[3] Frédéric Jouault and Ivan Kurtev. On the interoperability of model-to-model transformation languages. *Science of Computer Programming*, 68(3):114–137, 2007.

[4] Juan de Lara and Esther Guerra. From types to type requirements: genericity for model-driven engineering. *Software & Systems Modeling*, 12(3):453–474, 2013.

[5] Dimitris Kolovos, Louis Rose, Antonio Garcia-Dominguez, and Richard Paige. *The Epsilon Book. Eclipse.* 2010.

[6] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.

[7] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Softw. Syst. Model.*, 15(3):609–629, 2016.

[8] Massimo Tisi, Jean-Marie Mottu, Dimitrios S. Kolovos, Juan De Lara, Esther M Guerra, Davide Di Ruscio, Alfonso Pierantonio, and Manuel Wimmer. Lowcomote: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms. In *STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019)*, CEUR Workshop Proceedings (CEUR-WS.org), Eindhoven, Netherlands, July 2019.

[9] Benedek Horváth, Ákos Horváth, and Manuel Wimmer. Towards the next generation of reactive model transformations on low-code platforms: three research lines. In Esther Guerra and Ludovico Iovino, editors, *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings*, pages 65:1–65:10. ACM, 2020.

[10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[11] Benedek Horváth, Jolan Philippe, and Apurvanand Sahay. Concepts for multi-paradigm distributed transformation. Technical report, Institut Mines-Télécom, 11 2020.

[12] Jolan Philippe, Massimo Tisi, Hélène Coullon, and Gerson Sunyé. Towards Transparent Combination of Model Management Execution Strategies for Low-Code Development Platforms. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems*, (Virtual Conference), Canada, October 2020.

[13] Jolan Philippe, Massimo Tisi, Hélène Coullon, and Gerson Sunyé. Executing certified model transformations on apache spark. SLE 2021, page 36–48, New York, NY, USA, 2021. Association for Computing Machinery.

[14] Antonio García-Domínguez, Georg Hinkel, and Filip Krikava, editors. *Proceedings of the 11th Transformation Tool Contest, co-located with the 2018 Software Technologies: Applications and Foundations, TTC@STAF 2018, Toulouse, France, June 29, 2018*, volume 2310 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.

[15] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 599–613. USENIX Association, 2014.

[16] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[17] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[18] Dimitris S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The epsilon object language (eol). In *Proceedings of the Second European Conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA'06, pages 128–142, Berlin, Heidelberg, 2006. Springer-Verlag.

[19] Dimitris S. Kolovos, Richard F Paige, and Fiona AC Polack. Scalability: The holy grail of model driven engineering. In *ChaMDE 2008 Workshop Proceedings: International Workshop on Challenges in Model-Driven Software Engineering*, pages 10–14, 2008.

[20] Massimo Tisi and Zheng Cheng. CoqTL: an Internal DSL for Model Transformation in Coq. In *ICMT 2018 - 11th International Conference on Theory and Practice of Model Transformations*, volume 10888 of *LNCS*, pages 142–156, Toulouse, France, Jun 2018. Springer.

[21] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[22] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1-2):31–39, 2008.

[23] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon object language (eol). In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 128–142. Springer, 2006.

[24] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl: a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720, 2006.

[25] Youssef El Bakouny and Dani Mezher. Scallina: Translating verified programs from coq to scala. In Sukyoung Ryu, editor, *Programming Languages and Systems*, pages 131–145, Cham, 2018. Springer International Publishing.

[26] Palden Lama and Xiaobo Zhou. AROMA: automated resource allocation and configuration of MapReduce environment in the cloud. In Dejan S. Milojicic, Dongyan Xu, and Vanish Talwar, editors, *9th International Conference on Autonomic Computing, ICAC'12, San Jose, CA, USA, September 16 - 20, 2012*, pages 63–72. ACM, 2012.

[27] E Schikuta. MPI: A message-passing interface standard. *Techn. Ber., University of Tennessee, Knoxville, Tennesee*, 30, 1994.

[28] Jeremy Gibbons. The school of squiggol: A history of the bird-meertens formalism. in formal methods. In *FM 2019 International Workshops: Porto, Portugal, October 7-1, 2019, Revised Selected Papers, Part II*, pages 35–53, 2019.

[29] Grid'5000. https://www.grid5000.fr/. Last accessed on 2022-09-29.

[30] Raphael Bolze, Franck Cappello, Eddy Caron, Michel J. Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stéphane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quétier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *Int. J. High Perform. Comput. Appl.*, 20(4):481–494, 2006.

[31] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intell. AI Games*, 4(1):1–43, 2012.

[32] Benedek Izsó, Gábor Szárnyas, István Ráth, and Dániel Varró. MONDO-SAM: A framework to systematically assess MDE scalability. In Dimitris S. Kolovos, Davide Di Ruscio, Nicholas Drivalos Matragkas, Juan de Lara, István Ráth, and Massimo Tisi, editors, *Proceedings of the 2nd Workshop on Scalability in Model Driven Engineering co-located with the Software Technologies: Applications and Foundations Conference, BigMDE@STAF2014, York, UK, July 24, 2014*, volume 1206 of *CEUR Workshop Proceedings*, pages 40–43. CEUR-WS.org, 2014.

[33] Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda. *Software automatic tuning: from concepts to state-of-the-art results*. Springer Science & Business Media, 2010.

[34] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman P. Amarasinghe. Opentuner: an extensible framework for program autotuning. In José Nelson Amaral and Josep Torrellas, editors, *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 303–316. ACM, 2014.

[35] OMG. *System Modeling Language (SysML)*, 2019. formal/19-11-01.

[36] Benedek Horváth, Bence Graics, Ákos Hajdu, Zoltán Micskei, Vince Molnár, István Ráth, Luigi Andolfato, Ivan Gomes, and Robert Karban. Model Checking as a Service: towards pragmatic hidden formal methods. In Esther Guerra and Ludovico Iovino, editors, *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings*, pages 37:1–37:5. ACM, 2020.

[37] Robert Karban, Amanda G Crawford, Gelys Trancho, Michele Zamparelli, Sebastian Herzig, Ivan Gomes, Marie Piette, and Eric Brower. The OpenSE cookbook. In *Modeling, Systems Engineering, and Project Management for Astronomy VIII*, volume 10705, page 107050W. International Society for Optics and Photonics, 2018.

[38] Bence Graics, Vince Molnár, András Vörös, István Majzik, and Dániel Varró. Mixed-semantics composition of statecharts for the component-based design of reactive systems. *Softw. Syst. Model.*, 19(6):1483–1517, 2020.

[39] Gerd Behrmann, Alexandre David, Kim G. Larsen, John Hakansson, Paul Petterson, Wang Yi, and Martijn Hendriks. UPPAAL 4.0. In *Proc. of the 3rd Int. Conf. on the Quantitative Evaluation of Systems*, page 125–126. IEEE, 2006.

[40] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In *Formal Methods in Computer-Aided Design*, pages 176–179, 2017.

[41] Amir Molzam Sharifloo and Andreas Metzger. Mcaas: Model checking in the cloud for assurances of adaptive systems. In *Software Engineering for Self-Adaptive Systems III*, pages 137–153.

Springer, 2013.

[42] Ábel Hegedüs, Gábor Bergmann, Csaba Debreceni, Ákos Horváth, Péter Lunk, Ákos Menyhért, István Papp, Dániel Varró, Tomas Vileiniskis, and István Ráth. IncQuery Server for Teamwork Cloud: Scalable query evaluation over collaborative model repositories. In *Proc. of the 21st Int. Conf. on Model Driven Engineering Languages and Systems*, page 27–31. ACM, 2018.

[43] Antonio Bucchiarone, Jordi Cabot, Richard F. Paige, and Alfonso Pierantonio. Grand challenges in model-driven engineering: an analysis of the state of the research. *Softw. Syst. Model.*, 19(1):5–13, 2020.

[44] Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. The 2020 expert survey on formal methods. In *Formal Methods for Industrial Critical Systems*, volume 12327 of *LNCS*, pages 3–69. Springer, 2020.

[45] Mario Gleirscher and Diego Marmsoler. Formal methods in dependable systems engineering: a survey of professionals from europe and north america. *Empir. Softw. Eng.*, 25(6):4473–4546, 2020.

[46] Corrina Gibson, Robert Karban, Luigi Andolfato, and John Day. Formal validation of fault management design solutions. *Softw. Eng. Notes*, 39(1):1–5, 2014.

[47] Corrina Gibson, Robert Karban, Luigi Andolfato, and John C. Day. Abstractions for executable and checkable fault management models. In *Systems Engineering Research*, pages 146–154. Elsevier, 2014.

[48] Federico Ciccozzi, Ivano Malavolta, and Bran Selic. Execution of UML models: a systematic review of research and practice. *Software & Systems Modeling*, 18(3):2313–2360, 2018.

[49] Martin Kölbl, Stefan Leue, and Hargurbir Singh. From SysML to model checkers via model transformation. In *Proc. of the 25th International Symposium on Model Checking Software*, volume 10869 of *LNCS*, pages 255–274. Springer, 2018.

[50] Alessandro Tempia Calvino and Ludovic Apvrille. Direct model-checking of SysML models. In *Proc. of the 9th Int. Conf. on Model-Driven Engineering and Software Development*, pages 216–223. SCITEPRESS, 2021.

[51] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA finder. *Int. J. Softw. Tools Technol. Transf.*, 2(4):366–381, 2000.

[52] Apurvanand Sahay, Davide Di Ruscio, and Alfonso Pierantonio. Understanding the role of model transformation compositions in low-code development platforms. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 63:1–63:5. ACM, 2020.

[53] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Exploring model repositories by means of megamodel-aware search operators. In *MoDELS (Workshops)*, pages 793–798, 2018.

[54] Sorour Jahanbin. Efficient model loading through static analysis. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 660–665. IEEE, 2021.

[55] Francesco Basciani, Mattia D'Emidio, Davide Di Ruscio, Daniele Frigioni, Ludovico Iovino, and Alfonso Pierantonio. Automated selection of optimal model transformation chains via shortest-path algorithms. *IEEE Transactions on Software Engineering*, 2018.

[56] Marcel F van Amstel and Mark GJ Van Den Brand. Model transformation analysis: Staying ahead of the maintenance nightmare. In *International Conference on Theory and Practice of Model Transformations*, pages 108–122. Springer, 2011.

[57] Francesco Basciani, Davide Di Ruscio, Mattia D'Emidio, Daniele Frigioni, Alfonso Pierantonio, and Ludovico Iovino. A tool for automatically selecting optimal model transformation chains. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 2–6, 2018.

[58] Robert Bill, Martin Fleck, Javier Troya, Tanja Mayerhofer, and Manuel Wimmer. A local and global tour on momot. *Softw. Syst. Model.*, 18(2):1017–1046, 2019.

[59] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: advanced concepts and tools for in-place emf model transformations. In *International Conference on Model Driven Engineering Languages and Systems*, pages 121–135. Springer, 2010.

[60] Martin Fleck, Javier Troya, and Manuel Wimmer. Search-based model transformations with momot. In *International Conference on Theory and Practice of Model Transformations*, pages 79–87. Springer, 2016.

[61] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[62] Hisao Ishibuchi, Ryo Imada, Yu Setoguchi, and Yusuke Nojima. Performance comparison of nsga-ii and nsga-iii on various many-objective test problems. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 3045–3052. IEEE, 2016.

[63] Juri Di Rocco, Davide Di Ruscio, Johannes Härtel, Ludovico Iovino, Ralf Lämmel, and Alfonso Pierantonio. Systematic recovery of mde technology usage. In *International Conference on Theory and Practice of Model Transformations*, pages 110–126. Springer, 2018.

[64] Static Analysis built-on-the-top of Epsilon. https://github.com/epsilonlabs/static-analysis.git.

[65] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2010.

[66] Qurat ul ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. Efficiently querying large-scale heterogeneous models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–5, 2020.

[67] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: implementing domain-specific languages for java. In Klaus Ostermann and Walter Binder, editors, *Generative Programming and Component Engineering, GPCE'12, Dresden, Germany, September 26-28, 2012*, pages 112–121. ACM, 2012.

[68] Martin Eisenberg, Hans-Peter Pichler, Antonio Garmendia, and Manuel Wimmer. Towards reinforcement learning for in-place model transformations. In *24th International Conference on Model Driven Engineering Languages and Systems, MODELS 2021, Fukuoka, Japan, October 10-15, 2021*, pages 82–88. IEEE, 2021.

[69] Atl transformations. https://www.eclipse.org/atl/atlTransformations/. Accessed: 2022-05-28.

[70] Juri Di Rocco, Davide Di Ruscio, Johannes Härtel, Ludovico Iovino, Ralf Lämmel, and Alfonso Pierantonio. Understanding mde projects: megamodels to the rescue for architecture recovery. *Software and Systems Modeling*, 19(2):401–423, 2020.

[71] Anne Etien, Vincent Aranega, Xavier Blanc, and Richard F Paige. Chaining model transformations. In *Proceedings of the First Workshop on the Analysis of Model Transformations*, pages 9–14, 2012.

[72] Dennis Wagelaar. Composition techniques for rule-based model transformation languages. In *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*, ICMT '08, page 152–167, Berlin, Heidelberg, 2008. Springer-Verlag.

[73] Qurat ul ain Ali, Dimitris Kolovos, and Konstantinos Barmpis. Efficiently querying large-scale heterogeneous models. In *Proc. of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[74] Lorenzo Addazi, Antonio Cicchetti, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Semantic-based model matching with emfcompare. In *Me@ models*, pages 40–49, 2016.

[75] Saheed Popoola, Dimitrios S Kolovos, and Horacio Hoyos Rodriguez. Emg: A domain-specific transformation language for synthetic model generation. In *International Conference on Theory and Practice of Model Transformations*, pages 36–51. Springer, 2016.

[76] Francesco Basciani, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Automated chaining of model transformations with incompatible metamodels. In *International Conference on Model Driven Engineering Languages and Systems*, pages 602–618. Springer, 2014.

[77] Anne Etien, Alexis Muller, Thomas Legrand, and Richard F Paige. Localized model transformations for building large-scale transformations. *Software & Systems Modeling*, 14(3):1189–1213, 2015.

[78] Vincent Aranega, Anne Etien, and Sebastien Mosser. Using feature model to build model transformation chains. In *International Conference on Model Driven Engineering Languages and Systems*, pages 562–578. Springer, 2012.

[79] Anne Etien, Alexis Muller, Thomas Legrand, and Xavier Blanc. Combining independent model transformations. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2237–2243, 2010.

[80] Dennis Wagelaar, Ragnhild Van Der Straeten, and Dirk Deridder. Module superimposition: a composition technique for rule-based model transformation languages. *Software & Systems Modeling*, 9(3):285–309, 2010.

[81] Dennis Wagelaar. Composition techniques for rule-based model transformation languages. In *International Conference on Theory and Practice of Model Transformations*, pages 152–167. Springer, 2008.

[82] Raphaël Chenouard and Frédéric Jouault. Automatically discovering hidden transformation chaining constraints. In *International Conference on Model Driven Engineering Languages and Systems*, pages 92–106. Springer, 2009.

[83] José E Rivera, Daniel Ruiz-Gonzalez, Fernando Lopez-Romero, José Bautista, and Antonio Vallecillo. Orchestrating atl model transformations. *Proc. of MtATL*, 9:34–46, 2009.

[84] Bert Vanhooff, Stefan Van Baelen, Aram Hovsepyan, Wouter Joosen, and Yolande Berbers. Towards a transformation chain modeling language. In *International Workshop on Embedded Computer Systems*, pages 39–48. Springer, 2006.

[85] Eugene Syriani and Jeff Gray. Challenges for addressing quality factors in model transformation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 929–937. IEEE, 2012.

[86] Gehan MK Selim, James R Cordy, and Juergen Dingel. Model transformation testing: The state of the art. In *Proceedings of the first workshop on the analysis of model transformations*, pages 21–26, 2012.

[87] Eduard Bauer, Jochen M Küster, and Gregor Engels. Test suite quality for model transformation chains. In *International Conference on*

Modelling Techniques and Tools for Computer Performance Evaluation, pages 3–19. Springer, 2011.

[88] Hüseyin Ergin and Eugene Syriani. Identification and application of a model transformation design pattern. In *ACM southeast conference, ACMSE*, volume 13, 2013.

[89] Mohamed Wiem Mkaouer and Marouane Kessentini. Model transformation using multiobjective optimization. In *Advances in Computers*, volume 92, pages 161–202. Elsevier, 2014.

[90] Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. Model transformation as an optimization problem. In *International Conference on Model Driven Engineering Languages and Systems*, pages 159–173. Springer, 2008.

[91] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Omar Ben Omar. Search-based model transformation by example. *Software & Systems Modeling*, 11(2):209–226, 2012.

[92] Martin Fleck, Javier Troya, Marouane Kessentini, Manuel Wimmer, and Bader Alkhazi. Model transformation modularization as a many-objective optimization problem. *IEEE Transactions on Software Engineering*, 43(11):1009–1032, 2017.

[93] Dilan Sahin, Marouane Kessentini, Manuel Wimmer, and Kalyanmoy Deb. Model transformation testing: a bi-level search-based software engineering approach. *Journal of Software: Evolution and Process*, 27(11):821–837, 2015.

[94] Jordi Cabot and Ernest Teniente. A metric for measuring the complexity of ocl expressions. In *Model Size Metrics Workshop co-located with MODELS*, volume 6, page 10. Citeseer, 2006.

[95] Jesús Sánchez Cuadrado, Loli Burgueño, Manuel Wimmer, and Antonio Vallecillo. Efficient execution of atl model transformations using static analysis and parallelism. *IEEE Transactions on Software Engineering*, PP:1–1, 07 2020.

[96] Théo Le Calvar, Frédéric Jouault, Fabien Chhel, and Mickael Clavreul. Efficient atl incremental transformations. *The Journal of Object Technology*, 18:2:1, 07 2019.

[97] Áli Razavi and Kostas Kontogiannis. Partial evaluation of model transformations. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 562–572, 2012.

[98] Daniel Varro, Gábor Bergmann, Abel Hegedus, Ákos Horvath, István Ráth, and Zoltán Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the viatra framework. *Software Systems Modeling*, 15, 07 2016.