Towards Transparent Combination of Model Management Execution Strategies for Low-Code Development Platforms

Jolan Philippe
jolan.philippe@imt-atlantique.fr
IMT Atlantique, LS2N
Nantes, France

Héléne Coullon
helene.coullon@imt-atlantique.fr
IMT Atlantique, Inria, LS2N
Nantes, France

Massimo Tisi
massimo.tisi@imt-atlantique.fr
IMT Atlantique, LS2N
Nantes, France

Gerson Sunyé
gerson.sunye@ls2n.fr
Université de Nantes, LS2N
Nantes, France

ABSTRACT

Low-code development platforms are taking an important place in the model-driven engineering ecosystem, raising new challenges, among which transparent efficiency or scalability. Indeed, the increasing size of models leads to difficulties in interacting with them efficiently. To tackle this scalability issue, some tools are built upon specific computational strategies exploiting reactivity, or parallelism. However, their performances may vary depending on the specific nature of their usage. Choosing the most suitable computational strategy for a given usage is a difficult task which should be automated. Besides, the most efficient solutions may be obtained by the use of several strategies at the same time. This paper motivates the need for a transparent multi-strategy execution mode for model-management operations. We present an overview of the different computational strategies used in the model-driven engineering ecosystem, and use a running example to introduce the benefits of mixing strategies for performing a single computation. This example helps us present our design ideas for a multi-strategy model-management system. The code-related and DevOps challenges that emerged from this analysis are also presented.

CCS CONCEPTS

•Software and its engineering →Software creation and management; •Computing methodologies →Parallel algorithms;

KEYWORDS

Multi-strategy, Low-code development, Model-Driven Engineering, OCL, Spark

ACM Reference format:
Jolan Philippe, Héléne Coullon, Massimo Tisi, and Gerson Sunyé. 2020. Towards Transparent Combination of Model Management Execution Strategies for Low-Code Development Platforms. In *Proceedings of ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, October 18–23, 2020 (MODELS '20 Companion),* 10 pages.
DOI: 10.1145/3417990.3420206

MODELS '20 Companion, Virtual Event, Canada

978-1-4503-8135-2/20/10…$15.00
DOI: 10.1145/3417990.3420206

1 INTRODUCTION

Research in software engineering has produced several high-level abstractions to facilitate application development. Following this line, recent low-code development platforms (LCDPs) [39] propose visual interfaces for software development, distributed as platforms-as-a-service (PaaS), which minimize the need for users to write code. Most LCDPs are based on the description of application behavior with models, as promoted by model-driven engineering (MDE). In the MDE approach, models are the central and unifying point of the conception: they can represent knowledge, architectures, data, and so on. To be useful, models must be manageable by adding, removing, updating or querying information. The performance of these operations represent a field of study in the MDE community.

More specifically, model-management in LCDPs has a significant need for automatic and transparent efficient and scalable operations, for manipulating, querying and analyzing models. We identify three main reasons for this need. First of all, LCDPs need to provide complex visual development environments with low response time. For LCDPs that use models in the development phase, most of the model-management operations are executed at design time, e. g., for editing, validating, transforming the model. The required time for responding to a graphical command is a quality factor of the LCDP tool and has an influence on the developer's comfort [33], and on her efficiency. Cloud-based LCDPs have specific needs. For instance, they can integrate recommendation systems that may need to perform queries over the whole LCDP repository, to propose useful patterns to the user. Optimizing such design-time operations is important and challenging, especially when they require processing large-scale design models (or unions of models).

A second scalability issue arises when LCDPs need to manipulate large instance models of data, e. g., as it happens today in several (automotive, aeronautics, civil) engineering domains. In this paper we will consider a running case where a (fictional) company in social networking provides a LCDP to its users. Through the LCDP, users will be able to write their own apps over a huge social graph [15], represented as a model. Because of the sheer size of the model, providing an efficient solution is necessary.

The third reason for the need of efficient model-management in LCDPs is the number of concurrent operations on the platform. Due to the potential massive use of LCDPs, there is a need to run a big number of operations in parallel for many users. In the context of a PaaS, numerous customers may query models, thus servers or shared databases. Hence, efficient concurrent execution of model-management operations is necessary.

To improve efficiency and scalability, recent research on model-management studied parallel and concurrent programming as well as specific execution models for model-management languages. These techniques range from implementing specific execution algorithms (e. g., RETE) to compiling toward distributed programming models (e. g., MapReduce). In this paper we use the term *execution strategies* and (or just strategy) as a general way to denote these techniques. These techniques are sometimes qualified as *paradigms* in the literature, but this term may lead to confusion with programming paradigms (functional, logic, etc.).

The diversity of strategies that have been employed poses several scientific challenges. Most model-management languages implement a single execution strategy with specific strengths and weaknesses depending on the use case. Some existing solutions in MDE offer more than a single execution strategy but the choice is left to the user which requires expertise on parallelism or distribution [26]. Moreover, it appears that performance for some use cases could be improved by the combination of different strategies. To abide by low-code philosophy, the configuration of these development platforms should be as transparency as possible, even automatic.

In this paper we illustrate the variability of existing strategies, and emphasize the need for a multi-strategy vision for model-management where strategies can be automatically switched and combined to efficiently address the given model-management scenario. Furthermore, we stress the need for automatic choice and configuration of strategies to enhance performance of LCDPs. We outline code-related and DevOps challenges of a such approach and provide hints for technical solutions to these problems.

The rest of the paper is organized as follows. We motivate our work with an example in Section 2. Section 3 presents the necessary background and analyzes the existing computational strategies in model management. We introduce the *multi-strategy* approach in Section 4, and exemplify the variability of parallel execution strategies for the use case. In Section 5 we describe the main challenges for achieving a multi-strategy model-management engine. We finally conclude in Section 6.

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 paper, we consider a scenario where a vendor adds a LCDP to allow end-users (also called *citizen developers* in the LCDP jargon) to implement their own apps. Such LCDP could include a WYSIWG editor for the app user-interface, and a visual workflow for the behavioral part. In particular, the

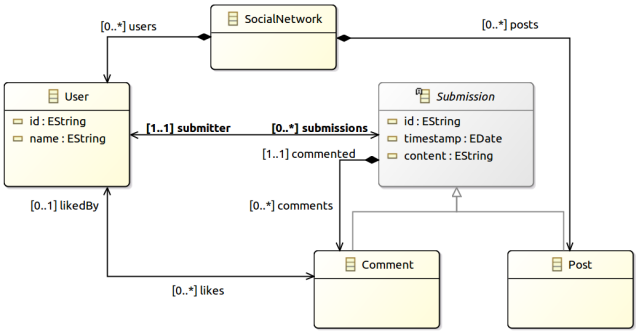

Figure 1: The metamodel of a social network (TTC 2018)

LCDPs would need to provide mechanisms, at the highest possible level of abstraction, to express queries and updates on the social graph.

In Fig. 1 we show the simple metamodel for the social graph that we will use in the paper. The metamodel has been originally proposed at the Transformation Tool Contest (TTC) 2018 [21], and used to express benchmarks for model query and transformation tools. In this metamodel, two main entities belong to a `SocialNetwork`. First, the `Posts` and the `Comments` that represent the `Submissions`, and second, the `Users`. Each `Comment` is written by a `User`, and is necessarily attached to a `Submission` (either a `Post` or another `Comment`). Besides commenting, the `Users` can also like `Submissions`.

As an example, in this paper we focus on one particular query, also defined in TTC2018: the extraction of the three most debated posts in the social network. To measure how debated is the post, we associate it with a numeric score. The LCDP will have to provide simple and efficient means to define and compute this score.

We suppose the vendor to include a declarative query language for expressing such computation on the social graph, and storing scores as a derived properties of the graph (i.e. new properties of the social graph that are computed on demand from other information in the graph).

In Listing 1 we implement the query to get the top-three debated posts in a model conforming to the presented metamodel, using the formula defined in TTC2018. The query is written in OCL, the most used declarative query language in MDE. In particular we use the ATL flavor of OCL.

In this code, a score of 10 is assigned to the post for each comment that belongs to it. Comments belong to a post in a recursive manner: a comment belongs to a post if it is attached either to the post itself, or to a comment that already belongs to the post. Then, a score of 1 is also added every time a belonging comment is liked.

The query is defined using three (attribute) helpers, that can be seen as derived properties. The first helper, `allComments` (line 7 to 11), collects recursively all the comments of a `Submission`. The second helper, `countLikes` counts how many times a comment that belongs to the given post has been liked. Then, the `score` of a `Post` is calculated by summing the result of `countLike` and the number of its belonging comments multiplied by ten. Finally the top three posts are obtained by the query `topPosts` sorting the posts by decreasing score, and selecting the first three.

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

```
1  query topPosts =
2      SN!Post.allInstances()
3          →sortedBy(e | -e.score)
4          →subSequence(1, 3);
5
6  helper context SN!Submission def : allComments =
7      self.comments
8          →union(
9              self.comments
10             →collect(e | e.allComments)
11             →flatten());
12
13 helper context SN!Post def : countLikes =
14     self.allComments
15         →collect(e| e.likedBy.size())
16         →sum();
17
18 helper context SN!Post def : score =
19     10*self.allComments→size() +
20         self.countLikes;
```

The simple declarative query in listing has not been defined with efficiency concerns in mind. Indeed, since we cannot make assumptions on the background of citizen developers, our LCDP cannot presume that they will structure the query for satisfying any performance requirement. As a result, when the number of users increases, soon the size of the social graph makes the computation of this query challenging. First of all, the list `Post.allInstances()` (line 2) becomes too large to manipulate. Especially the full sorting of posts (line 3) seems prohibitive. Without an efficient mechanism, the naive recomputation of `allComments` each time it is called, is a further performance waste. If we consider the typical frequency of updates for social network graphs, keeping the list of top posts up-to-date by fully recomputing this query at each update could consume a significant amount of infrastructure resources.

Moreover, the most efficient way to execute the query does not depend only on the given query definition and metamodel structure, but on several characteristics of the usage scenario. A technique to optimize a particular use case typically has significant overhead in other use cases. Factors that can influence this choice in our example can be related to the model size (e.g. order of magnitude for the number of `Users`), frequency of updates (e.g. of new `Submissions`), average model metrics (e.g. average number of `Comments` per `Post`), acceptable response time for the final query (`topPosts`), infrastructure constraints and resources (e.g. available memory, CPUs) and so on. In some cases techniques can be combined, further complexifying the search for the optimal solution.

Finally, while in this paper we will focus exclusively on this example, it is not difficult to identify similar issues for update operations (e.g. removal of all information for an unsubscribing user) or transformation (e.g. for storing the graph in a particular persistence format).

3 EXECUTION STRATEGIES IN MODEL MANAGEMENT

In this section we outline the execution strategies that are commonly used to enhance the efficiency of model-management. The below presented strategies have been identified with their use in MDE. In this Section, we only focus on the strategies, regardless of the chosen language for their implementations. We also give an overview of the existing applications of these strategies in model-management tools.

The two main categories of model-management tools we consider are model transformation (MT) and query (MQ). On the one hand, model transformation is the conversion process of one or more input models to output models (model-to-model) or text (model-to-text). A model transformation that produces a model as output can be either an in-place (i.e., direct modification of the input model) or an out-place transformation (i.e. production of a new model from the input one). On the other hand, a model query analyzes source models to compute the desired data value. Finally, some general key concepts (e.g., matching), that can be used both in MT and MQ are using strategies to improve the performances of engines. These concepts are also discussed in the current Section.

3.1 Avoiding computations

Incrementality and *laziness* are the main strategies used in MDE for minimizing the sequence of basic operations needed to perform a query or transformation. They have been classified as strategies for reactive execution in [33], since they foster a model of computation where the model-management system reacts to update and request events, (note that the term is only inspired by the reactive programming paradigm in the sense of [23], that we wont discuss here).

We classify existing applications of these strategies to model-management tools in the columns of Table 1, depending on their scope:

- **MQ** or **MT** if the strategy is applied to the whole model query or transformation;
- **Matching** if the strategy is only applied to the matching phase (the subgraph isomorphism of the pattern to query/transform, over the full model) of the model query/-transformation;
- **Collections** if the strategy is only applied to the computation of collections during the query/transformation.

Table 1: Reactive strategies for model-management in literature.

	MQ	MT	Matching	Collection
Incrementality	[12]	[29]	[6, 45]	
Laziness	[38]	[41]		[7, 46]

3.1.1 Incrementality. Incrementality is an event-based pattern, whose goal is to reduce the number of needed operations when a change happens within the input model. Instead of applying from scratch the whole set of operations on the new input model, incrementality allows the system to apply only the operations impacted

by updates. Since the system needs to apply a subset of operations, a trace to relate the output pieces to input elements is necessary. The approach leads then to an additional memory cost, with a good trade-off only if changes occur often enough.

To achieve incremental execution of transformation rules, Calvar et al. designed a compiler to transform a code written with ATL [29], a QVT-like (Query View Transformation) language to Java code. The output program takes advantage of active operations of the language. The active mechanism works as an observer pattern: the values are defined as mutable, and changes are notified to an external observer. From there, it is easy to isolate what part of the model has been changed, and then to deduce what rules must be operated again. To illustrate their proposal, they applied their evaluation to two cases including on social media models to illustrate the efficiency of the strategy for querying models that have strong user activity. This is not the single attempt of integrating incremental aspects in ATL.

In [12], Cabot et al. present an incremental evaluation of OCL expressions that are used to specify elements of a model in ATL. They used a such approach to state integrity preservation of models at runtime. Instead of testing the whole integrity of a model every time it is changed, the proposed system is able to determine when, and how, each constraint must be verified.

For example, the RETE algorithm for pattern matching, presented in [45], constructs a network to specify patterns and, at runtime, tracks matched patterns. Instead of matching a whole pattern, the RETE algorithm will match the subparts of the pattern until getting a full match. Incrementality is here used to update the incomplete patterns, without fully recalculating the matching for all the present candidates. In MDE, the Eclipse VIATRA framework has an implementation of the RETE algorithm to achieve an incremental pattern matching [6]. The choice of using an incremental algorithm is due to the focus of the tool. Indeed, the VIATRA platform focuses on event-driven and reactive transformations thus an efficient solution, for handling multiple changes, has been chosen.

3.1.2 Laziness. Laziness is also commonly used by model management tools. In general, laziness reduces computations by removing the ones that are not needed to answer to the user requests. Indeed, by using laziness, pieces of output are calculated only when they are needed by the user. This "call-by-need" approach is mainly used on big models, known as Very Large Models (VLMs). Since users may want to get only a part of the output, computing the whole query/transformation is unnecessary.

In [41], Tisi et al. extended the model transformation mechanism of ATL with laziness. Elements of the target model are firstly initialized, but their content is generated only when a user tries to access it. To do so, the model navigation mechanism has a tracking system, which provides, for a target element, the rules that must be executed. In addition, the tracking system keeps information about already executed rules to avoid recomputation. Other engines, such as ETL (Epsilon Transformation Language), from the Epsilon framework, implements a similar approach.

Besides model transformation, laziness is also used in model querying. In [38], Tisi et al. redefine OCL features with laziness aspects. For instance, operations of the language are redefined to be evaluated with a lazy strategy. Also, the work proposes lazy collections that respect the OCL specification. The latter is similar to the collections proposed by Willink in [46]. The OCL collections are implemented as generic Java classes, with lazy operators. These approaches aim at tackling OCL related efficiency issues. For example, because of the OCL collections are immutable, the successive add of elements in a collection would create intermediate data structures. More generally, the composition of operation calls would cause an evaluation of a cascade of operations. The proposed implementation of a lazy evaluation optimizes such common cases.

3.2 Parallelizing computations

Parallelism designates the use of several processing units in order to achieve a global operation. There exist many kinds of parallel architectures, from multi-cores to clusters of GPUs. In this paper, we only focus on the parallelism strategies that may be used to take advantage of parallel architectures.

In Table 2, we classify how parallelism has been applied to model management in literature, by the following columns:

- **MQ** or **MT** if the whole model query or transformation is parallelized;
- **Matching** if the work only parallelizes the matching phase of the model query/transformation;
- **Performance** whether the work pays particular attention to the impact of data distribution or task distribution on performance.

We classify the strategies into three categories: data-parallelism (Section 3.2.1); task-parallelism (Section 3.2.2), both of them being synchronous strategies; and one example of asynchronous strategy (Section 3.2.3).

Table 2: Parallelism for model-management in literature.

	MQ	MT	Matching	Perf.
Task-parallelism	[30, 43]	[24, 40]	[34]	
Data-parallelism		[4, 26, 42]	[26]	[5]
Asynchronism		[8–10]		[9]

3.2.1 Data-Parallelism. In a data-parallel approach, data is split and distributed across several computation units. Then, the same piece of program (from a single basic operation, to a complex function) is applied simultaneously on each part of data by each processing unit without synchronization. Furthermore, additional synchronizations and communications may be needed between processing units to correctly compute the overall result. For instance, data may need to be merged into a single result. This computation strategy is the one followed by the parallel algorithmic skeletons [16] on data structures [19, 35].

MapReduce [20] is a an example of programming model, designed for parallelism, that takes advantage of this strategy. However, MapReduce is mainly adapted and implemented for distributed arrays or lists, and the approach is not directly suitable for all types of data structures. For instance, *Pregel* [31] is a strategy that aims at easing parallel computations on graphs by using a vertex-centric approach. In Pregel, graphs are specified by their vertices, each of them embedding information on their incoming and outgoing

edges. A Pregel program is iterative, and is decomposed in three main phases: a computation on top of a vertex value, a generation of messages, and the send of messages through the edges of the vertex. This process is simultaneously applied to each vertex of a graph (such as a `map` in MapReduce).

Data-parallelism is adapted and adopted in case of large datasets. Indeed, to make profitable the parallel execution of a single computation on data, the data chunks must be large enough, otherwise an overhead has to be paid without much benefits from the parallelization effort [2, 17].

Benelallam et al. [4] use data-parallelism for distributing models among computational cores to reduce computation time in the ATL model transformation engine. The MapReduce version of ATL makes independent transformations of sub-parts of the model by using a local "match-apply" function. Then, the reduction aims at resolving dependencies between map outputs. The proposed approach guarantees better performance on basic cases such as the transformation of a class diagram to a relational schema. In a more recent work [5], the same authors highlight the good impact of their strategy for data partitioning. Instead of randomly distributing the same number of elements among the processors, they use a strategy based on the connectivity of models.

[25] illustrates how a model can be considered as a typed graph with inheritance and containment. Considering a model as a graph data-structure, the graph technologies can directly be applied to models. For instance, Imre et al. efficiently use a parallel graph transformation algorithm on real-world industrial-sized models for model transformation [24]. In [34], Mezei et al. use graph rewriting operations based on task-parallelism to distribute matching operations in large models in their transformation tool Visual Modeling and Model Transformation (VMTS). The Henshin framework [26] proposes to extract the matching part of its transformation rules into vertex-centric code (i.e., Pregel). Another possibility to use Pregel in model transformation is by using a DSL, such as [42] for graph transformation. The proposed compiler transform the code written with the DSL into an executable Pregel code.

3.2.2 Task-Parallelism. A *task-parallel* program focuses on the distribution of tasks instead of data. According to [36], "a task is a basic unit of programming that an operating system controls" within a job. This concept is often associated to multi-threading. The grain size of tasks depends on the context of the execution. At the operating system level, tasks may be entire programs while at the program level, they may be a single request, or a single operation. Because of concurrency, and the limited number of processing units, tasks executions must be ordered by considering both priorities, and dependencies across tasks. Ordering tasks in parallel are similar to the workflow concept. Task-parallelism will be preferred to data-parallelism when tasks are complex enough, or when the number of tasks is large enough to exploit parallelism capacities of the underlying parallel architecture (i.e., hardware).

[43] proposes a formal description of parallelism opportunities in OCL. Two main kinds of operation are targeted: the binary operations that can have their operands evaluated simultaneously, and the iterative processes of independent treatments. In [30], Madani et al. use multi-threading for "select-based" operations in EOL, the OCL-like language of the Epsilon framework, for querying models. The extension of the language with parallel features for selective operations have shown a non-negligible speed-up (up to 6x with 16 cores) in their evaluations on a model conform to the Internet Movie Database (IMDb) metamodel[1]. Next to query evaluation, multi-threading is also used for model transformation. In [40], Tisi et al. present a prototype of an automatic parallelization for the ATL transformation engine, based on task-parallelism. To do so, they just use a different thread for each transformation rule application, and each match, without taking into account concurrency concerns (e.g., race conditions).

3.2.3 Asynchronism. Both data-parallelism and task-parallelism can be defined as synchronous strategies where synchronizations are explicitly performed through communication patterns, or tasks dependencies. *Asynchronism* is another way of programming parallelism where synchronism is not explicitly coded but implicitly handled by an additional mechanism between processing units. For example, the Linda approach [13], is based on the treatment of asynchronous tasks or data, shared in a common knowledge base, the "blackboard" [11]. More specifically, in Linda several processes access a shared tuple space representing the shared knowledge of a system. The processing units interact with the shared space by reading, and/or removing tuples.

LinTra is a Linda-based platform for model management and has several types of implementation. First, on a shared-memory architecture (i.e., a same shared memory between processors, typically multi-threading solutions), LinTra proposes parallel in-place transformations [10] and parallel out-place transformations [8]. Both strategies have significant gains in performance, compared to sequential solutions. Nonetheless, shared-memory architecture are fine for not too big models. Indeed, since the memory is not distributed, a too big model could lead to a out-of-memory errors. This phenomenon happens more concretely in an out-place transformation since two models are involved during the operation. The first prototype of distributed out-place transformations in LinTra, is presented in [8], and works with sockets for communicating the machines. This first proposal remains naive. That is why, Burgueno et al. proposes a more realistic prototype for transformations on distributed architecture [9]. But the use of a distributed architecture raises new questions: how to distribute data and, how to distribute tasks? They applied different strategies mixing both the evaluation of tasks on a single or on multiple machines, and storing the source and target models on the same, or on different machines. The study was conducted for the specific IMDb test case only, and then does not provide a general conclusion about the benefits of a such solution.

One can note from Table 2 that only two papers of the related work on parallelism in MDE offer a detailed performance analysis according to the data or tasks distribution. However, both these papers clearly show that many factors can influence performance such as the size of models, their reading/writing modes (e.g., in-place), the distribution of the models and the distribution of the operations to perform on them and so on.

[1] http://www.imdb.com/interfaces

4 MULTI-STRATEGY MODEL MANAGEMENT

Each of the research efforts presented in Tables 1 and 2 exploit a single strategy for optimizing model-management operations. 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 query or transformation engine performs **multi-strategy model management** if it automatically considers different strategies, instead of a single one, in order to manipulate models in an efficient way. According to Section 3 and to the best of our knowledge, such approach does not exist in the literature.

In this section, we exemplify the multi-strategy approach by implementing the OCL query of Listing 1 in different ways, using different strategies of parallelism. The goal of this Section is not to provide the most efficient solutions for solving the given problem. Instead, it aims at illustrating the diversity of solutions, that each have its own advantages depending on the use cases. To do so, we implemented several solutions using different parallel strategies and compared them. Also, this section only illustrates the variability of single solution, and not their possible combination.

Our prototype is built on top of Spark[2], an engine designed for big data processing. 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 [22] that embeds the possibility to define Pregel programs), 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 of a multi-strategy approach. Comparing solutions that include laziness and incrementality aspects is a part of our future works. 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 and an instance of either a `User` or a `Submission` (`Comment` or `Post`). Edges represent the links of elements of a model conforming the meta-model presented in Figure 1, labeled by a String name. We keep exactly the same labels from the meta-model 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.

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

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

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). A 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 a 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 the 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 in the rest of this section: *direct-naive*, *pregel*, and *highly-parallel*. Then we discuss these three implementations and open to the multi-strategy approach.

4.1 Direct naive implementation

The first implementation, namely *direct-naive*, shown in Listing 3, directly follows the OCL helpers from Listing 1. The first auxiliary function `countLikes`, corresponding to the homonym helper, sums the number of `"like"` relations for each comment of a given post (lines 17 to 21). The second auxiliary function `score` (lines 23 and 24) is also a direct Spark translation from the OCL query. It uses parallelism, coupled with the lazy evaluation provided by Spark. Indeed, the execution of operations on RDDs is not started until an action is triggered. In our example, `collect` and `count` are these actions. Finally, the `allComments` function is defined recursively using GraphX features. The direct-naive implementation of `score` uses three functions that are inspired by functional languages: `filter` which removes all the elements of a list that do not respect a given predicate; `map` that applies a function to every element; and `flatMap` which is a composition of `map` and `flatten`. The latter is equivalent to `flatten` from Listing 1. The implementation first gets the direct comments of a post (lines 10 and 11), and, using an auxiliary function `getComments`, recursively gets all the belonging comments (lines 13 and 14). The method `flatMap` of lines 8 and 13 transforms the list of lists, into a list of comments.

4.2 Pregel implementation

The second solution, namely *pregel*, proposed in Listing 4, is a Pregel-based implementation. The main idea of this solution is, starting from a `Post`, counting the number of comments and the number of likes for these comments by propagating messages through edges of the graph by using Pregel. To do so, we declare two variables, `nbComments`, and `nbLikes`, that can be seen as aggregators, i.e., global accumulator of values. The propagation is processed using the Pregel support of GraphX that works as follows. At each iteration, the function *mergeMsg* accumulates into a single value the incoming messages (line 20), that are stored in a iterable structure, from the previous iteration (with an initial message defined for the first iteration). This value is used by *vprog* with the previous vertex v_n to generate the new vertex data v_{n+1}. With this value, messages are generated with *sendMsg* and sent to vertices through edges for the next iteration. Because the stucture which stores the message must be iterable, all the messages must be of type Iterator. An

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

Listing 3: Direct implementation of score.

```
1
2  def allComments (p : Post) = {
3    // recursive function
4    def getComments (co : Comment) : List[Comment] =
5      List(co).union(sn.triplets
6          .filter(t => t.srcAttr == co
7                       & t.attr == "comment")
8          .flatMap(_.dstAttr compose getComments).collect)
9
10   sn.triplets
11       .filter(t => t.srcAttr == p & t.attr == "comment")
12       .flatMap(_.dstAttr compose getComments).collect
13 }
14
15 def countLikes (p: Post) =
16   allComments(p)
17     .map(c => sn.triplets.filter
18                 (t => t.attr == "like" & t.dstAttr == c)
19         .count).sum
20
21 def score (p : Post) =
22     10 * allComments(p).size + countLikes(p)
```

Listing 4: Pregel implementation of score.

```
1  def score(p: Post) = {
2    var nbComment, nbLike = 0L // Aggregators
3    val fstId = sn.vertices
4                  .filter(v => v._2 == p).first._1
5
6    sn.mapVertices((_, v) => (fstId, v)).pregel
7      (initialMsg = (fstId, false))
8      (vprog = (id, value, merged) =>
9        if (merged_msg._1 == id || merged_msg._1 < 0)
10         if (merged_msg._2) nbLike += 1L
11         else nbComment += 1L
12       (merged_msg._1, value._2),
13     sendMsg = t =>
14       if (t.srcId == fstId | t.srcAttr._1 == -1L)
15         if (t.attr == "comment")
16           Iterator((triplet.dstId, (-1L, false)))
17         if (t.attr == "likedBy")
18           Iterator((t.dstId, (-1L, true)))
19       Iterator.empty,
20     mergeMsg = (m, _) => m)
21
22   10 * nbComment + nbLike
23 }
```

empty message is then produced by `Iterator.empty`. The program stops when no message is produced for the next iteration. In our implementation, messages are tuples of two values. The first one is an integer value for specifying which vertices should compute and send messages. Besides, if the integer is negative, then all the vertices should compute. The second one aims at precising what value must be incremented (either the number of comments (`false`), or likes (`true`)). The initial step of the execution questions the model to get the id of the vertex containing the `Post` we want to score (line 3 and 4). This identifier is added to every vertex to provide them a global view on the computation status (line 6). Then, messages are propagated through the edges to belonging comments of computed vertices, or to the users who likes the scoped comment (line 13 to 19). At the message reception, the computation will increment the aggregator according to the second value of the message (line 9 to 11). After the execution of the `pregel` function, a score value is calculated using `nbComments` and `nbLikes`.

4.3 MapReduce implementation

Listing 5 illustrates a solution with a higher level of parallelism, namely *highly-parallel*, that uses a MapReduce approach. The purpose of this third solution is to process as much as possible operations in parallel in a first time, and then go through the graph to reduce these values. The first step counts the number of direct subcomments, and the number of likes, for each element of the model, using a map and reduce-by-key composition (line 3 to 7). Because the number of likes has not the same importance than the number of belonging comments in the score calculation, two keys are created for a single element: one for counting each property (i.e., number of comments and number of likes). Then a graph-traversal operation calculates the total number of belonging comments and likes for a given post. However, the keys are only created if a comment, or a like, exists. Then, to initialize values, we use a composition of `find` that returns an option, and `getOrElse` in the case of the absence of the key. The latter returns the value of the option if it exists, and a default value otherwise We do not expect to gain performances with this approach because the operations are not costly enough. However, having a highly parallel approach largely increase the scalability of the program.

4.4 Discussion on multi-strategy

First, the complexity of the solutions *direct-naive* and *pregel* can be compared. On the one hand, the complexity on 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 are all 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.

Listing 5: Highly parallel implementation of score.

```
def score(p: Post) = {
  // number of likes and comments per element
  val scores = sn.triplets
      .filter(t => t.attr == "likedBy"
                | t.attr == "comment")
      .map(t => ((t.attr, t.srcAttr), 1L))
      .reduceByKey((a,b) => a + b).collect

  def getScore(s: Submission) = {
    val default = ((_,_), 0L)
    var nbLike = // 0 if s is not liked
        scores.find(e => e._1._2 == s
                       & e._1._1 == "likedBy")
            .getOrElse(default)._2
    var nbComment = // 0 if s is not commented
        scores.find(e => e._1._2 == s
                      & e._1._1 == "comment")
            .getOrElse(default)._2
    // recursive call
    val subScores = sn.triplets
        .filter(t => t.srcAttr == s
             & (t.attr == "likedBy"
               |t.attr == "comment"))
        .map(_.dstAttr compose getScore).collect
    // sum of all score from belonging comments
    for (score <- subScores) {
        nbLike += score._1
        nbComment += score._2
    }
    (nbLike, nbComment)
  }
  val score_p = getScore(p)
  10L * score._1 + score._2
}
```

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 of 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 elements of the model. In the latter, the parallelism level of graph-traversal has the same limitation than the Pregel implementation, but always process a less complex operation (i.e., a reduction as a sum of integer values).

The three above parallel approaches can solve the same problem, but their efficiency depends on external parameters. For executing the `topPosts` query, a multi-strategy engine would compile it to:

- the direct-naive implementation if the depth of belonging comments is small;
- the pregel solution if the environnement has few resources for parallelism;
- the highly-parallel solution if the score computation needs big calculation on the vertices themselves.

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 the all presented strategies of Section 3, 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 occur, 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 to the solution, to only recompute potential new most-debated posts.

5 CHALLENGES IN MULTI-STRATEGY MODEL-MANAGEMENT

In the perspective of low-code platforms, a multi-strategy engine should be fully automated, from the automatic strategy selection to the automatic configuration and deployment on distributed infrastructures. Our approach is different from the multi-strategy approach proposed in [3] which is focused on languages and their salient features. The conception of a multi-strategy engine leads to many scientific challenges that we divide in two parts in the rest of this section: the challenges related to the code and the challenges related to DevOps.

5.1 Code-related challenges

A first scientific challenge that arises from the multi-strategy approach is the automatic and transparent selection of the most adapted strategy for a given model-management operation. The motivating example of Section 2 shows the large variability to take into account to make the right choice. We divide this variability in different properties that should be considered:

- properties on the input model: size, meta-model, topology, etc.;
- properties on the operation to perform: update, launch, request, read or write, the frequency of the operation, etc.;
- properties on the available infrastructures: type of frameworks compatible or already deployed on the infrastructure;

This variability results in a combinatorial choice that could be solved by using constraint programming, or by leveraging machine learning techniques to automatically learn how to associate these properties together.

The second challenge related to the code aspect is that an initial code written with a MDE solution may need to be rewritten to follow the chosen execution strategy, while guaranteeing the same expected output. In other words, a code rewriting or code generation challenge is raised by the multi-strategy approach. In particular, formal semantics for the model-management engine may be of high importance to guarantee a correct output code [37].

As example, Listings 3, 4, and 5 described in Section 4 all present a different way of writing a code from the initial OCL solution shown in Listing 1. The complexity and the level of parallelism of the solutions have been discussed in Section 4.

5.2 DevOps-related challenges

In the context of low-code platforms, automatically handling the strategy selection and the code generation is not the only concern. Once generated, the code must be deployed and run on complex distributed infrastructures. These tasks should be as transparent as the previous code-related challenges. Hence, a first challenge is automatically handling the deployment of a set of model operations that potentially use different strategies, onto the associated infrastructures that could themselves be very heterogeneous (e. g. different public Cloud solutions such as AWS, or private Clouds, hybrid Clouds, etc.). This complexity should be handled at the LCDP level, which requires safe and efficient deployments [14, 18].

Furthermore, choosing a given strategy, often involves deploying code on existing frameworks or platforms that implement that strategy. For instance, when choosing the MapReduce (respectively Pregel) strategy, Hadoop[3] (resp. Giraph[4] or Spark[5]) should be used to benefit from efficient implementation. All these frameworks are highly configurable, e. g., MapReduce has more than one hundred parameters [28]). Because of their large number of parameters, finding their optimal configuration is a difficult problem. This additional layer of configuration represents an additional combinatorial challenge. Several solutions could provide a good trade-off. For instance, instead of providing a full configuration for the tools, which is very costly, a performance prediction built from configuration samples could be used. This solution has been adopted by Pereira et al. [1]. Another approach would be to make a full cost estimation but only considering critical parameters. For example, give the right level of parallelism by providing an approximation of the optimal number of mapper and reducer in a MapReduce job[6]. More formal approaches can also be used to estimate the cost of parallel programs (e.g., cost model for GraphX [27]), and compare the different solution using additional parameters such as hardware configuration (e.g., the bridging Bulk Synchronous Parallel cost model [44]).

Using formal approaches to estimate the cost of a parallel program such as BSP cost model [44] or Pregel cost estimation [27].

Finally, as for the combinatorial problem of choosing the right strategy, machine learning techniques could be adopted [32].

6 CONCLUSION

In this paper, we made an overview of what, and how, execution strategies can be used for model driven engineering. In the context of developing low-code platforms for managing models, these strategies might be used for optimizing performances. However, a wrong use of a computational model can have a bad impact on calculation efficiency. The motivating example presented in Section 2 and the implementations of Section 4 illustrate that by using different strategies and different combinations of paradigms for a given input model, different advantages could be observed, such as complexity, and parallelism level. Different paradigms may be chosen, according to different properties: the type of input model, its size, its topology, the type of computation to perform, and the available infrastructure. The future goal of a such prototype is to drive a complete study of how the paradigms can be used and combined, and to classify them depending on use cases.

[3]http://hadoop.apache.org/
[4]https://giraph.apache.org/
[5]http://spark.apache.org/
[6]http://wiki.apache.org/hadoop/HowManyMapsAndReduces

ACKNOWLEDGMENTS

This paper disseminates results from the Lowcomote project, that received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skffodowska-Curie grant agreement No 813884.

REFERENCES

[1] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, and Jean-Marc Jézéquel. 2020. Sampling Effect on Performance Prediction of Configurable Systems: A Case Study. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (Edmonton AB, Canada) *(ICPE fi20)*. Association for Computing Machinery, New York, NY, USA, 277–288. https://doi.org/10.1145/3358960.3379137

[2] G. M. Amdahl. 2007. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *IEEE Solid-State Circuits Society Newsletter* 12, 3 (Summer 2007), 19–20. https://doi.org/10.1109/N-SSC.2007.4785615

[3] Moussa Amrani, Dominique Blouin, Robert Heinrich, Arend Rensink, Hans Vangheluwe, and Andreas Wortmann. 2019. Towards a Formal Specification of Multi-paradigm Modelling. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019*, Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel (Eds.). IEEE, 419–424. https://doi.org/10.1109/MODELS-C.2019.00067

[4] Amine Benelallam, Abel Gómez, and Massimo Massimo Tisi. 2015. ATL-MR: model transformation on MapReduce. In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel Systems, SEPS SPLASH 2015, Pittsburgh, PA, USA, October 27, 2015*, Ali Jannesari, Skiegfried Benkner, Xinghui Zhao, Ehsan Atoofian, and Yukionri Sato (Eds.). ACM, 45–49. https://doi.org/10.1145/2837476.2837482

[5] Amine Benelallam, Abel Gómez, Massimo Tisi, and Jordi Cabot. 2018. Distributing relational model transformation on MapReduce. *Journal of Systems and Software* 142 (2018), 1–20. https://doi.org/10.1016/j.jss.2018.04.014

[6] Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, and Gergely Varró. 2008. Incremental Pattern Matching in the Viatra Model Transformation System. In *Proceedings of the Third International Workshop on Graph and Model Transformations* (Leipzig, Germany) *(GRaMoT fi08)*. Association for Computing Machinery, New York, NY, USA, 25–32. https://doi.org/10.1145/1402947.1402953

[7] Gerth Stølting Brodal and Rolf Fagerberg. 2002. Cache Oblivious Distribution Sweeping. In *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2380)*, Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan J. Eidenbenz, and Ricardo Conejo (Eds.). Springer, 426–438. https://doi.org/10.1007/3-540-45465-9_37

[8] Loli Burgueño, Manuel Wimmer, and Antonio Vallecillo. 2016. A Linda-based platform for the parallel execution of out-place model transformations. *Information & Software Technology* 79, C (Nov 2016), 17–35. https://doi.org/10.1016/j.infsof.2016.06.001

[9] Loli Burgueño, Manuel Wimmer, and Antonio Vallecillo. 2016. Towards Distributed Model Transformations with LinTra. *Jornadas de Ingeniera del Software y Bases de Datos* (2016), 1–6. http://hdl.handle.net/10630/12091

[10] Loli Burgueño, Javier Troya, Manuel Wimmer, and Antonio Vallecillo. 2015. Parallel In-place Model Transformations with LinTra. In *Proceedings of the 3rd Workshop on Scalable Model Driven Engineering part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 23, 2015. (CEUR Workshop Proceedings, Vol. 1406)*, Dimitris S. Kolovos, Davide Di Ruscio, Nicholas Drivalos Matragkas, Juan de Lara, István Ráth, and Massimo Tisi (Eds.). CEUR-WS.org, 52–62. http://ceur-ws.org/Vol-1406/paper6.pdf

[11] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns.* Wiley Publishing. 71–95 pages. https://ff.tu-sofia.bg/~bogi/knigi/SE/Wiley%20-%20Pattern-Oriented%20Software%20Architecture%20-%20Volume%201,%20A%20System%20of%20Patterns.pdf

[12] Jordi Cabot and Ernest Teniente. 2009. Incremental integrity checking of UML/OCL conceptual schemas. *Journal of Systems and Software* 82, 9 (2009), 1459–1478. https://doi.org/10.1016/j.jss.2009.03.009

[13] Nicholas Carriero and David Gelernter. 1989. Linda in Context. *Commun. ACM* 32, 4 (1989), 444–458. https://doi.org/10.1145/63334.63337

[14] Maverick Chardet, Hélène Coullon, Dimitri Pertin, and Christian Pérez. 2018. Madeus: A formal deployment model. In *4PAD 2018 - 5th International Symposium on Formal Approaches to Parallel and Distributed Systems (hosted at HPCS 2018)*. Orléans, France, 1–8. https://hal.inria.fr/hal-01858150

[15] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-scale. *Proc. VLDB Endow.* 8, 12 (Aug 2015), 1804–1815. https://doi.org/10.14778/2824032.2824077

[16] Murray Cole. 1988. *Algorithmic skeletons : a structured approach to the management of parallel computation*. Ph.D. Dissertation. University of Edinburgh, UK. http://hdl.handle.net/1842/11997

[17] Hélène Coullon, Julien Bigot, and Christian Pérez. 2017. Extensibility and Composability of a Multi-Stencil Domain Specific Framework. *International Journal of Parallel Programming* (Nov. 2017). https://doi.org/10.1007/s10766-017-0539-5

[18] Hélène Coullon, Claude Jard, and Didier Lime. 2019. Integrated Model-checking for the Design of Safe and Efficient Distributed Software Commissioning. In *IFM 2019 - 15th International Conference on integrated Formal Methods (Integrated Formal Methods)*. Bergen, Norway, 120–137. https://hal.archives-ouvertes.fr/hal-02323641

[19] Hlne Coullon and Sbastien Limet. 2016. The SIPSim implicit parallelism model and the SkelGIS library. *Concurrency and Computation: Practice and Experience* 28, 7 (2016), 2120–2144. https://doi.org/10.1002/cpe.3494

[20] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (San Francisco, CA) *(OSDI'04)*. USENIX Association, Berkeley, CA, USA, 137–149. http://dl.acm.org/citation.cfm?id=1251254.1251264

[21] Antonio García-Domínguez, Georg Hinkel, and Filip Krikava (Eds.). 2019. *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*. CEUR Workshop Proceedings, Vol. 2310. CEUR-WS.org. http://ceur-ws.org/Vol-2310

[22] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 599–613. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[23] D. Harel and A. Pnueli. 1989. *On the Development of Reactive Systems*. Springer-Verlag, Berlin, Heidelberg, 477–498.

[24] Gábor Imre and Gergely Mezei. 2012. Parallel Graph Transformations on Multicore Systems. In *Multicore Software Engineering, Performance, and Tools*, Victor Pankratius and Michael Philippsen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 86–89.

[25] Stefan Jurack and Gabriele Taentzer. 2010. A Component Concept for Typed Graphs with Inheritance and Containment Structures. In *Graph Transformations - 5th International Conference, ICGT 2010, Enschede, The Netherlands, September 27 - - October 2, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6372)*, Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr (Eds.). Springer, 187–202. https://doi.org/10.1007/978-3-642-15928-2_13

[26] Christian Krause, Matthias Tichy, and Holger Giese. 2014. Implementing Graph Transformations in the BulkSynchronousParallel Model. In *Fundamental Approaches to Software Engineering*, Stefania Gnesi and Arend Rensink (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 325–339. https://doi.org/10.1007/978-3-642-54804-8_23

[27] Rohit Kumar, Alberto Abelló, and Toon Calders. 2017. Cost Model for Pregel on GraphX. In *Advances in Databases and Information Systems - 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10509)*, Marite Kirikova and George A. Norvaag, Kjetiland Papadopoulos (Eds.). Springer, 153–166. https://doi.org/10.1007/978-3-319-66917-5_11

[28] Palden Lama and Xiaobo Zhou. 2012. AROMA: automated resource allocation and configuration of MapReduce environment in the cloud. In *9th International Conference on Autonomic Computing, ICAC'12, San Jose, CA, USA, September 16 - 20, 2012*, Dejan S. Milojicic, Dongyan Xu, and Vanish Talwar (Eds.). ACM, 63–72. https://doi.org/10.1145/2371536.2371547

[29] Théo Le Calvar, Frédéric Jouault, Fabien Chhel, and Mickael Calreul. 2019. Efficient ATL Incremental Transformations. *Journal of Object Technology* 18, 3 (Jul 2019), 2:1–17. https://doi.org/10.5381/jot.2019.18.3.a2 The 12th International Conference on Model Transformations.

[30] Sina Madani, Dimitris S. Kolovos, and Richard F. Paige. 2019. Towards Optimisation of Model Queries: A Parallel Execution Approach. *Journal of Object Technology* 18, 2 (July 2019), 3:1–21. https://doi.org/10.5381/jot.2019.18.2.a3 The 15th European Conference on Modelling Foundations and Applications.

[31] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[32] Hugo Martin, Juliana Alves Pereira, Mathieu Acher, and Paul Temple. 2019. Machine Learning and Configurable Systems: A Gentle Introduction. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC fi19)*. Association for Computing Machinery, New York, NY, USA, 325–326. https://doi.org/10.1145/3336294.3342383

[33] Salvador Martínez Perez, Massimo Tisi, and Rémi Douence. 2017. Reactive model transformation with ATL. *Sci. Comput. Program.* 136 (2017), 1–16. https://doi.org/10.1016/j.scico.2016.08.006

[34] Gergely Mezei, Tihamer Levendovszky, Tams Mszros, and Istvn Madari. 2009. Towards truly parallel model transformations : A distributed pattern matching approach. *IEEE EUROCON 2009, EUROCON 2009*, 403–410. https://doi.org/10.1109/EURCON.2009.5167663

[35] Jolan Philippe and Frédéric Loulergue. 2019. PySke: Algorithmic Skeletons for Python. In *The 2019 International Conference on High Performance Computing & Simulation (HPCS)*. Dublin, Ireland. https://hal.archives-ouvertes.fr/hal-02317127

[36] Margaret Rouse. [n.d.]. Task, Definition. https://whatis.techtarget.com/definition/task. Accessed: 2020-07-14.

[37] Massimo Tisi and Zheng Cheng. 2018. CoqTL: an Internal DSL for Model Transformation in Coq. In *ICMT 2018 - 11th International Conference on Theory and Practice of Model Transformations (LNCS, Vol. 10888)*. Springer, Toulouse, France, 142–156. https://doi.org/10.1007/978-3-319-93317-7_7

[38] Massimo Tisi, Rémi Douence, and Dennis Wagelaar. 2015. Lazy Evaluation for OCL. In *Proceedings of the 15th International Workshop on OCL and Textual Modeling co-located with 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Canada, September 28, 2015 (CEUR Workshop Proceedings, Vol. 1512)*, Achim D. Brucker, Marina Egea, Gogolla Martin, and Frédéric Tuong (Eds.). CEUR-WS.org, 46–61. http://ceur-ws.org/Vol-1512/paper04.pdf

[39] Massimo Tisi, Jean-Marie Mottu, Dimitrios S. Kolovos, Juan De Lara, Esther M Guerra, Davide Di Ruscio, Alfonso Pierantonio, and Manuel Wimmer. 2019. 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. https://hal.archives-ouvertes.fr/hal-02363416

[40] Massimo Tisi, Martínez Salvador Perez, and Hassene Choura. 2013. Parallel Execution of ATL Transformation Rules. In *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8107)*, Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke (Eds.). Springer, 656–672. https://doi.org/10.1007/978-3-642-41533-3_40

[41] Massimo Tisi, Salvador Martínez Perez, Frédéric Jouault, and Jordi Cabot. 2011. Lazy Execution of Model-to-Model Transformations. In *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6981)*, Jon Whittle, Tony Clark, and Thomas Kühne (Eds.). Springer, Berlin, Heidelberg, 32–46. https://doi.org/10.1007/978-3-642-24485-8_4

[42] Le-Duc Tung and Zhenjiang Hu. 2017. Towards Systematic Parallelization of Graph Transformations Over Pregel. *Int. J. Parallel Program.* 45, 2 (April 2017), 320–339. https://doi.org/10.1007/s10766-016-0418-5

[43] Tamás Vajk, Zoltán Dávid, Márk Asztalos, Gergely Mezei, and Tihamér Levendovszky. 2011. Runtime Model Validation with Parallel Object Constraint Language. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation* (Wellington, New Zealand) *(MoDeVVa)*. Association for Computing Machinery, New York, NY, USA, Article 7, 8 pages. https://doi.org/10.1145/2095654.2095663

[44] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. https://doi.org/10.1145/79173.79181

[45] Varró, Gergely and Frederik Deckwerth. 2013. A Rete Network Construction Algorithm for Incremental Pattern Matching. In *Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7909)*, Keith Duddy and Gerti Kappel (Eds.). Springer, 125–140. https://doi.org/10.1007/978-3-642-38883-5_13

[46] Edward D. Willink. 2017. Deterministic Lazy Mutable OCL Collections. In *Software Technologies: Applications and Foundations - STAF 2017 Collocated Workshops, Marburg, Germany, July 17-21, 2017, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10748)*, Martina Seidl and Steffen Zschaler (Eds.). Springer, 340–355. https://doi.org/10.1007/978-3-319-74730-9_30