"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

# Concepts for Multi-Paradigm Distributed Transformation

---

**Project GA**: 813884

**Project Acronym**: Lowcomote

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

**Project officer**: Dora Horváth

**Work Package**: WP5

**Deliverable number**: D5.2

**Production date**: November 29th 2020

**Contractual date of delivery**: November 30th 2020

---

**Actual date of delivery**: November 30th 2020

**Dissemination level**: Public

**Lead beneficiary**: Institut Mines-Télécom

**Authors**: Benedek Horváth, Jolan Philippe, Apurvanand Sahay

**Contributors**: The Lowcomote partners

**Project Abstract**

Low-code development platforms (LCDP) are software development platforms on the Cloud, provided through a Platform-as a-Service model, which allow 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 integrate 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 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 agility of our future professionals between the academic and industrial world.

**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 and their 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 are able to quickly react 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. Specification 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 a customized modeling 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 motivate our work with a running example on a fictional company in social networking, which provides an LCDP to millions of users, therefore urging the need for a highly scalable solution. We introduce state-of-the-art single-strategy model-management solutions and urge the need for a multi-strategy approach that would automatically choose and configure the strategies for the best execution possible. Besides, to achieve scalable reactive model transformations, we propose the parallel extension of a state-of-the-art reactive model transformation engine. Finally, we propose a workflow for the efficient execution of model transformation composition scenarios across several external platforms.

# Contents


**1 Introduction** **4**
- 1.1 On the need of scalable operations . . 4
- 1.2 On the need of reusable and interoperable workflows . . . . . . . . . . . . 5
- 1.3 Contributions . . . . . . . . . . . . . . 5
- 1.4 Outline . . . . . . . . . . . . . . . . . . 5

**2 Motivating Example** **6**
- 2.1 Querying a social media model . . . . 6
- 2.2 Transforming and checking a social network at runtime . . . . . . . . . . . 7
- 2.3 Composing model transformations . . 9

**3 Cloud-based Scalable Model Management Operations** **11**
- 3.1 Single-strategy model management . . 11
- 3.2 Multi-strategy model management . . 14

**4 Live Model Transformations in Reactive Applications** **20**
- 4.1 Reactive transformations in VIATRA . 20
- 4.2 Parallel reactive model transformations in VIATRA . . . . . . . . . . . . . 21
- 4.3 Related work . . . . . . . . . . . . . . . 24

**5 Composition of Model Transformations** **25**
- 5.1 Models to be transformed . . . . . . . 25
- 5.2 Proposed steps to achieve workflows using model transformation composition . . . . . . . . . . . . . . . . . . . . . 26
- 5.3 Related work . . . . . . . . . . . . . . . 27

**6 Conclusion** **30**

# 1 Introduction

Model-Driven Engineering (MDE) [1] is a method to develop a software by using domain models at a higher level of abstraction. The model is either interpreted, or code is generated from it, to produce the desired software. The Object Management Group [2] proposed the Model Driven Architecture (MDA) which defines MDE. The MDA-based software starts by building platform-independent models which are transformed to one or more platform-specific models, which can further be transformed to code for a specific software. This model transformation plays a key role to determine the interoperability [3] with other softwares along with the reusability [4] of the artifacts within or outside a particular software scenario.

Low-Code Development Platform (LCDP) [1] adopts the recent theoretical and practical advancements of MDE and produces the software by either interpreting the model or by compiling the code generated from the model. Both methods reduce the manual coding and thus help increase the productivity of the application [5]. Some of the model-interpreting LCDPs are Mendix[2], Appian[3], Salesforce App Cloud[4], while some of the code-generating LCDPs are OutSystems[5], Kony[6], Alpha Software[7].

## 1.1 On the need of scalable operations

Research in software engineering has produced several high-level abstractions to facilitate application development. Following this line, recently emerged LCDPs propose visual interfaces for software development, which enables citizen developers with little or no prior knowledge in programming to realize fully operational applications [6]. As promoted by MDE, all LCDPs describe the application's behavior in models. In the MDE approach, models are the central and unifying point of the conception: they can represent knowledge, architectures, and data. Models can be managed by adding, removing, updating or querying information from them. The performance of these operations represents a research line in the MDE community.

More specifically, model management in LCDPs has a significant need for automatic, transparent, efficient and scalable operations for manipulating, querying and analyzing models. We identify three main reasons for this need. First of all, 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 [7], and her efficiency. Therefore, LCDPs need to complete complex operations with low response time. Moreover, since LCDPs are Cloud-based platforms deployed in a Platform-as-a-Service (PaaS) model, they 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.

A second scalability issue arises when LCDPs need to manipulate, process, transform large instance models, as it happens today in several engineering domains. Both Kolovos et al. in [8] and Bucchiarone et al. in [9] have listed scalability of model transformations as one of the key challenges in MDE. Efficiently running these operations, in terms of memory use and execution time, on very large models with millions of elements is a challenging task. The challenge is multiplied in case of distributed models and distributed execution environments. Although model transformation is a widely researched area, state-of-the-art tools lack in performance even for mid-sized models [9].

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 [10]) to compiling towards distributed programming models (e. g., MapReduce [11]). In this deliverable, we use the term *execution strategies* (or strategy in short) 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 [12].

[1] Hereafter, the terms *low-code platform* and *low-code development platform* are used interchangeably and are abbreviated as *LCDP*.

[2] https://www.mendix.com/

[3] https://www.appian.com/

[4] https://developer.salesforce.com/platform

[5] https://www.outsystems.com/

[6] https://www.kony.com/solutions/low-code/

[7] https://www.alphasoftware.com/

Cloud computing provides a distant architecture that tackles the up-front IT infrastructure limits (e. g., limited number of resources). Increasing the number of available resources allows horizontal and vertical scalability, i. e., the application remains scalable either 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 that is designed for big data processing, offers a highly scalable and parallel solution. Also, other frameworks, such as Spark are very popular in the Cloud. They offer scalable solutions for model management operations.

### 1.2 On the need of reusable and interoperable workflows

An LCDP integrates different tools and components in order to easily develop new applications and fulfill non-functional properties like reusability and interoperability [13]. An application consists of several process flows which take data from different sources. To allow complex workflows within and across LCDPs and other external platforms, the process for building a new application using customized modeling language(s) (e. g., building a BPMN[8] like modeling language) is aimed to be designed. Such a modeling language represents workflows as models. Each step of the workflow can be regarded as a data fetching operation (model query) or as a data combination (model transformation) operation. Besides, several workflows can be composed into a workflow system to model complex applications. Using a process builder to construct the complex workflows plays an important role in creating an easy-to-use platform to build complex applications. Such workflows can be reused by storing the smaller transformations in model repositories and automatically executing those transformations specified in the overall application's workflows. Also, an application can interoperate with external sources such as IFTTT or Google Maps. Therefore, the reusability and interoperability in an LCDP is aimed to be achieved by using several compositions of model transformations defined in different workflows and process flows within and across different platforms.

### 1.3 Contributions

In this report, we present three main contributions.

1. We first 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 address the given model-management scenario efficiently. Furthermore, we stress the need for automatic choice and configuration of strategies to enhance the performance of LCDPs.
2. Then, we formulate reactive model transformations for LCDPs as a means to improve the response time of these platforms. We propose an extension of a state-of-the-art reactive model transformation engine to achieve higher throughput and lower response times on low-code platforms with frequently occurring events.
3. Finally, all the possible workflows are defined within and across LCDPs and other external services. Also, a research path is proposed to build a modeling language and their metamodels along with an application-builder metamodel. Further, a composition reasoner is planned to reason about the need for model transformation composition for different goal-to-workflow transformations. Such workflows will use a workflow engine to orchestrate flows of data. Also, a connector is supposed to be built to facilitate the interoperability of different LCDPs. Lastly, the literature survey on model transformation compositions and the use of model transformations in a distributed environment is done.

### 1.4 Outline

The rest of the report is organized as follows. We motivate our work with a running example in Section 2, where a fictional company in social networking provides an LCDP to its users. After that, in Section 3 we elaborate on the use of Cloud-based infrastructures for model transformations. Section 4 introduces live and reactive transformations for LCDPs and proposes a new approach to improve their scalability. In Section 5 the possible composition scenarios of model transformation compositions and the proposed work on LCDPs are described. Finally, we conclude the report and present future work in Section 6.

[8] http://www.bpmn.org/

# 2 Motivating Example

Social network vendors often provide specific development platforms, used by developers to build applications (or apps in short) that extend the functionality of the social network. Some networks are associated with market-places 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. In this report, we will consider a running case where a fictional company in social networking provides an LCDP to its users. Through the LCDP, users will be able to write their own apps over a huge social graph [14], represented as a model. Because of the sheer size of the model, providing an efficient solution is necessary.

Social networks are built in a graph structure, representing a model of relationships. Since these networks are based on the interaction of people, the data is often led to be modified. The LCDP should then consider reactive mechanisms, especially for the mutable parts of the model. Also, the size of the model depends one the number of users. First, the maximum number of relationships among $n$ number of users is equal to $\binom{n}{2} = \frac{n!}{2!(n-2)!}$. Trivially, this value increases with an increasing value of $n$. Second, the number of submissions in the network is related to the number of users. For instance, in 2010, the Facebook graph represented a trillion of interactions (friendships, likes, shares, etc.) [15]. Since the number of user has been multiplied by 5 in 10 years, we theoretically expect a way more large set of data [16]. A scalable solution is then inevitably needed.

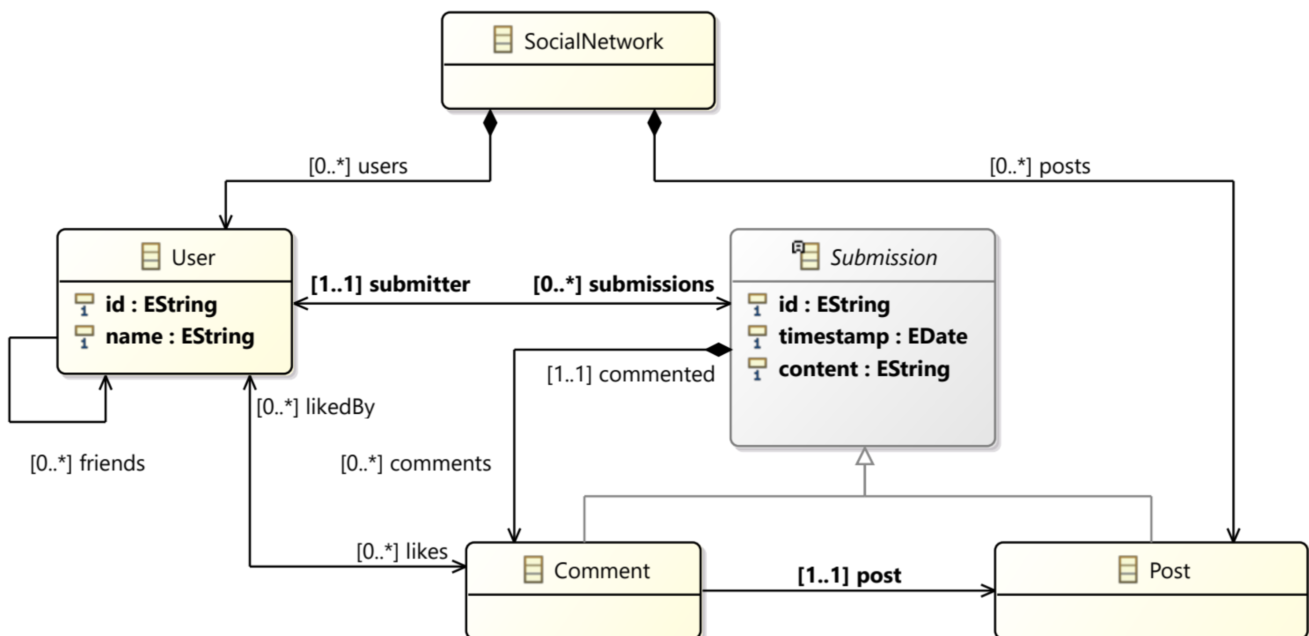


Figure 1: The metamodel of a social network (TTC 2018)

In Figure 1 we show the simple metamodel for the social graph that we will use in the report. The metamodel has been originally proposed at the Transformation Tool Contest (TTC) 2018 [17], 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`.

## 2.1 Querying a social media model

In this report, 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 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 Object Constraint Language (OCL) [18], the most used declarative query language in MDE. In particular, we use the ATL flavor of OCL [19].

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

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

In this code, a score of 10 is assigned to a `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` (lines 5–8), collects recursively all the comments of a `Submission`. The second helper, `countLikes` (lines 10–13) 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 `countLikes` and the number of its belonging `Comments` multiplied by ten (lines 15–16). Finally, the top three `Posts` are obtained by the query `topPosts` (lines 1–3) sorting the `Posts` by decreasing score, and selecting the first three.

The simple declarative query in Listing 1 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 1) becomes too large to manipulate. Especially the full sorting of posts (line 2) 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 on 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.

## 2.2 Transforming and checking a social network at runtime

As users use social media platforms, its content evolves over time. They post new pictures, share stories and videos, comment on posts of their friends, etc. Although free speech allows citizens to share their thoughts without restrictions, there are some sensitive topics about which it is inappropriate to talk or most people find them disturbing. Therefore as `Submissions` evolve over time, its content should be regularly checked to identify `Posts` and `Comments` that contain inappropriate content (e. g., prohibited pictures, hate speech, forbidden symbols, fake news, etc.). As the number of `Posts` exponentially grows each year [16], it is impossible to check them manually, due to the large cognitive effort and the sheer number of the `Posts` that are sent each day. Therefore, we need automated methods to keep up with this magnitude of information and to keep the platform safe and useful for its users.

To achieve this goal, there are some machine learning and AI-based approaches which apply image recognition or linguistic (Natural Language Processing) techniques to identify and recognize such contents, to mark them as inappropriate and to automatically remove them from the platform. These techniques can be categorized as *static checks*, that are triggered as soon as the `User` submits the `Comment` or `Post`

on the platform. In case of text-based contents, these static checks can be performed quickly and with high reliability, however for the image recognition and video processing some further research is needed to improve the confidence of the classification.

Although the static checks are able to capture a static picture of the social graph model, they are not able to check its dynamic evolution over time. For example, they are not able to recognize a trend, that every time a `User` submits a `Post` on the platform, then one of her friends will also submit a `Post` in one hour. In order to recognize such trends in a short time, we need to continuously monitor the evolution of the social network graph on a large scale.

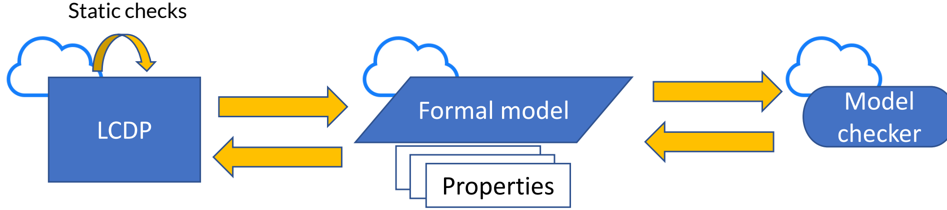


Figure 2: Model Checking as a Service workflow

Therefore we propose a Cloud-based Model Checking as a Service (MCaaS) workflow [20], depicted in Figure 2, that employs state-of-the-art model checking algorithms to formally prove that the model satisfies certain properties. If the model does not satisfy the property, then they return a counter-example which demonstrates how the violation occurs, which helps engineers identify and fix the cause of the problem.

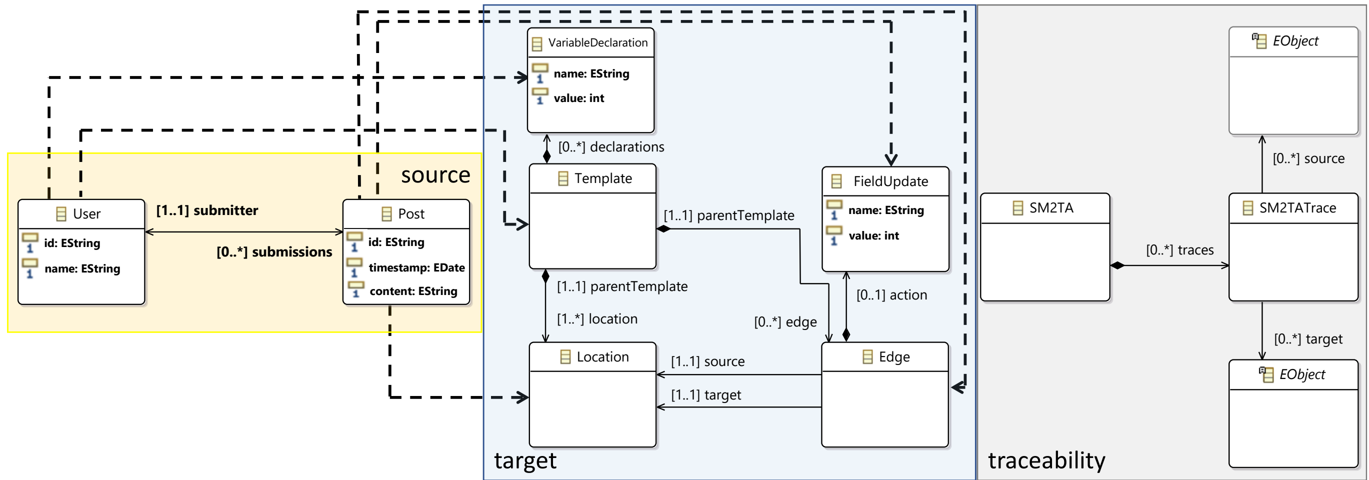


Figure 3: Source, target and traceability metamodels excerpt

Figure 3 depicts an excerpt of the source, target and traceability metamodels that are used in the verification workflow. The source metamodel contains the `User` and the `Post`. The target metamodel is timed automata formalism. A timed automaton is a `Template` that has several `VariableDeclarations` (fields). Besides, it contains several `Locations` and `Edges`. `Edges` connect the `Locations` and they can have an `FieldUpdate` that sets a `VariableDeclaration` of the `Template` for a given `value`. From each `User`, a `Template` is instantiated in the target domain. Each `Post` that is submitted to the social media platform is transformed to a `Location`, an `Edge` and a `FieldUpdate`. The `Edge` connects the newly created `Location` to the last one. The `FieldUpdate` indicates that the `timestamp` field of the `Template` has to be set for the time epoch of the `Post`. The translation of the corresponding elements is illustrated by dashed arrows. During each transformation action, a traceability link (`SM2TATrace`) is created, in order to incrementally update the target model according to changes in the source model.

From the target model, a model-to-text transformation creates the textual representation that can be processed by UPPAAL, a model checker that verifies CTL expressions on timed automata [21, 22]. In order to prove that if *Alice* submitted a `Post` on 13.11.2020 at 8:37 PM, then her friend, *Bob* also submitted a `Post` within one hour, the *Alice.timestamp* == 1605299835 → (*Bob.timestamp* <= 1605299835 && *Bob.timestamp* <= 1605303435) CTL expression is evaluated by UPPAAL on the serialized target model. If the expression is not satisfied by the target model, then the sequence of `Posts` sorted by their `timestamp` can be used as an example trace demonstrating the violation.

The MCaaS workflow has several advantages. First of all, it utilizes the elastic scalability of the Cloud, to adaptively tackle the high computation and memory demand of model checkers. A second advantage is, different model checkers can be attached to the workflow and they can be started in parallel for a given verification task, and their results can be combined together. Therefore, the most suitable (quickest) prover can be selected for the given task, which results in a shorter verification time and a faster feedback loop.

## 2.3 Composing model transformations

Model transformation is a key concept in MDE which is applicable in an LCDP. Some of the key features of low-code development that require model transformation are pre-built forms, reports and pages, interoperability with external sources such as APIs, IFTTT[9], zapier[10], etc. along with built-in workflows and converting report view from grid to kanban to CSV, etc. [23].

Many smaller and simpler model transformations are chained together to realize a complex transformation available in an LCDP. In such chaining of model transformations, pre- and post-conditions need to be ensured while the metamodels must be chained properly as per the compatibility of model transformations.

The usability of model transformation is defined in relation with the LCDP. In this aspect, multiple chainings of model transformations are offered so that all kinds of possible artifacts within an application can be realized. These artifacts are models that are achieved by one or more model transformations. Also, model transformation and its composition can be applied for the interoperability with external sources, which is one of the key aspects in a software system that uses some kind of bridging mechanism to transform artifacts from one software to another.

The model transformation composition can be shown using the motivation example of a social network that can be transformed to the tabular view which could further be transformed to the HTML format. These kinds of multiple transformations require an external chaining of transformations so that we can reuse those transformations as and when required. Such chaining must preserve the syntactic characteristics of the composed transformations (i. e., the source, and the target metamodels) and the semantic aspects that will drive the automatic selection of the intermediate transformations that have to be retrieved from a repository of existing model transformations.

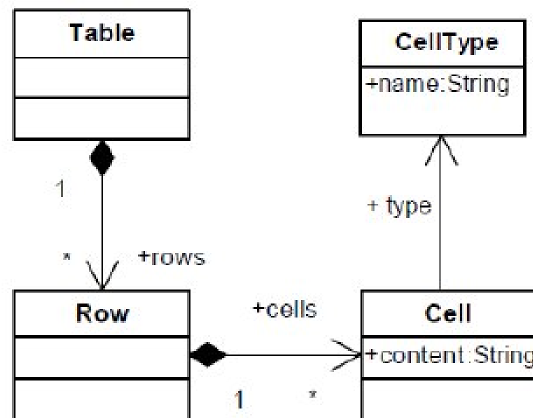


Figure 4: Metamodel for table [24]

To transform the model from social network metamodel (Figure 1) to the table metamodel (Figure 4), we need to know the logic of the source model and how it can be transformed into the target model based on the target metamodel. For example, considering the social network metamodel, a model is created with three `Users`: ”Adam”, ”Smith” and ”Evan”. Adam submitted two posts with content as ”Flood” and ”Covid19”. Smith commented on both of them. Adam replied to the comment on the content ”Covid19”. Evan commented on Adam’s comment on ”Flood”. Meanwhile, Evan submitted one post with content as ”Earthquake”. Smith commented on this content which Adam replied swiftly followed by Evan’s reply on the same comment. Evan also separately commented on the content ”Earthquake”.

This social network model can be transformed into tables by using corresponding rules that translate `User` to `Row` and `Submission` to `Cell`. These transformation rules are implemented using various transformation languages [25]. The table will be differentiated by the `Posts` sent by a `User` and it is shown in Table 1 and Table 2.

| | Flood | Covid19 |
|---|---|---|
| Adam | Post, Reply3 | Post, Reply2 |
| Smith | Comment1 | Comment2 |
| Evan | Comment3 | |

Table 1: Table for Adam’ post

| | Earthquake |
|---|---|
| Adam | Post, Reply4, Reply5 |
| Smith | Comment4 |
| Evan | Comment5, Reply6 |

Table 2: Table for Evan’ post

Furthermore, we can transform the transformed tables into HTML code by model-to-text transformations [26]. For example, a table is transformed to HTML code by using Eclipse Generation Language

[9]https://ifttt.com/

[10]https://zapier.com/

(EGL[11]) as it is shown in Listing 2.

Listing 2: Table to HTML transformation EGL code

```
1 <table>
2     [% for (Row in r.row) { %]
3      <tr>
4         [<% for (cell in c.cell) { %]
5             <td>[% = c.content %]</td>
6         [%}%]
7      </tr>
8     [%}%]
9 </table>
```

These two model transformations from social network to tables and from tables to the HTML web page should be stored in a model repository. The research objectives in relation to model transformation composition are the followings. First, transforming multiple models and chaining them according to their optimized execution. In this regards, model queries are optimized in order to optimize the execution of a model transformation. Secondly, a mechanism is required to orchestrate two or more stored transformations so that the composition of model transformation can be done by automatically orchestrating the necessary intermediate transformations.

[11]https://www.eclipse.org/epsilon/doc/egl/

# 3 Cloud-based Scalable Model Management Operations

In this Section, we first introduce approaches for model management that can take advantage of a distributed infrastructure: either by avoiding computations, or by parallelizing them. Then, we motivate the use of an adaptive engine based on a multi-strategy approach for model management operations. Most of these works have been published in [27].

## 3.1 Single-strategy model management

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 analysis, 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 below.

**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 [7], since they foster a model of computation where the model management system reacts to update and request events [28].

**Incrementality**

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 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 [30], 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 [31], 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 [32]. 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 (see Section 4) thus an efficient solution has been chosen to handle multiple changes.

**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 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 [33], 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 the rules that must be executed to produce the target element. In addition, the tracking system keeps information about already executed rules to avoid recomputation. Other engines, such as Epsilon Transformation Language (ETL) [12], from the Epsilon framework, implements a similar approach.

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

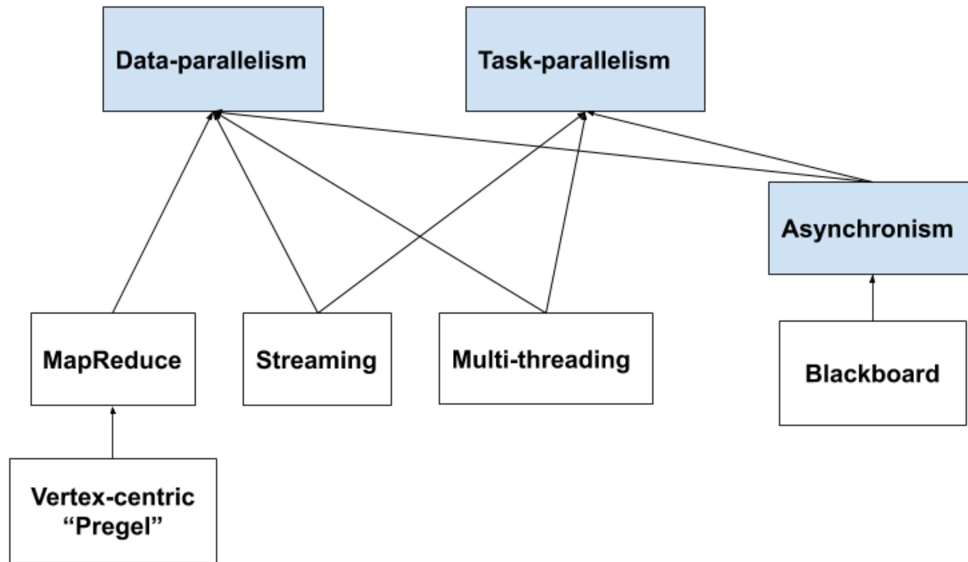


Figure 5: Metamodelisation of distributed approaches

Besides model transformation, laziness is also used in model querying. In [34], 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 [35]. 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 insertion 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.

**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 section, we only focus on the parallelism strategies that may be used to take advantage of parallel architectures, such as Cloud infrastructure. We classify the strategies into three categories: *data-parallelism*; *task-parallelism*, both of them being synchronous strategies; and one example of *asynchronous strategy*. Figure 5 presents a global view of these approaches. Note that the works of [36, 37, 38] presented below also serve as related work for Section 4.

**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 [39] on data structures [40, 41]. *MapReduce* [11] is 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* [42] 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 sending of messages through the edges of the vertex. This process is simultaneously applied to each vertex of a graph (such as the *map* phase of a MapReduce process). Data-parallelism is often 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 [43, 44].

Benelallam et al. [45] 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 [46], 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. [47] 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 [48]. In [49], 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 [12] 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 [50] for graph transformation. The proposed compiler transforms the code written in the DSL into an executable Pregel code.

All the aforementioned model query techniques can leverage only the computational power of a single machine. To overcome this limitation, Szárnyas et al. proposed IncQuery-D, a distributed incremental model query framework in the Cloud [51]. The framework implemented a distributed RETE network, where each machine stores a subset of the Rete nodes which communicate with each other to update their local caches. They proposed a distributed termination protocol to know if a model change has propagated through the whole network.

**Task-parallelism**

A *task-parallel* program focuses on the distribution of tasks instead of data. According to [52], "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, task executions must be ordered by considering both priorities, and dependencies across them. 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).

[53] 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 [36], Madani et al. use multi-threading for "select-based" operations in Epsilon Object Language (EOL) [13], 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[14]. Next to query evaluation, multi-threading is also used for model transformation. In [37], 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).

**Asynchronism**

Both data-parallelism and task-parallelism can be defined as synchronous strategies where synchronizations are explicitly performed through communication patterns, or task 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 [54], is based on the treatment of asynchronous tasks or data, shared in a common knowledge base, the "blackboard" [55]. 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 [38] and parallel out-place transformations [56]. 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 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 [56], 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 [57]. But the use of a distributed

[13] https://www.eclipse.org/epsilon/doc/eol/
[14] http://www.imdb.com/interfaces

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 does not provide a general conclusion about the benefits of a such solution.

## 3.2 Multi-strategy model management

Each of the research efforts presented above 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 in order to manipulate models in an efficient way. To the best of our knowledge, such approach does not exist in the literature yet.

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. Our prototype is built on top of Spark[15], an engine designed for big data processing in the Cloud. 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 having its own advantages depending on the use cases. To do so, we implemented several solutions using different parallelism strategies and compared them. Also, this section only illustrates the variability of single solution, and not their possible combination. Finally, we present first experimental results to illustrate what performance benefits Spark can provide.

### Implementations for 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 [58] 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 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 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 6.

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

Figure 6: 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). 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*, and *highly-parallel*, *pregel*. Then we discuss these implementations and open to the multi-strategy approach.

**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

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

to the homonym helper, sums the number of `"like"` relations for each comment of a given post (lines 13 to 18). The second auxiliary function `score` (lines 20 and 21) 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 uses an auxiliary, and recursive, function `traversal`. It first gets the direct comments of a submission (line 6), and apply the same process to its the belonging comments (lines 7 and 8). The method `flatMap` of lines 8 transforms the list of lists, into a list of comments.

```
1  def allComments (p : Post) = {
2   // recursive function
3     def traversal (s : Submission) : List[Submission] = {
4        List(s).union(sn.triplets
5              // Get all direct comments of p as vertices
6              .filter(t => t.srcAttr == s & t.attr == "comment"))
7              .map(_.dstAttr).collect // Collect the sub comments
8              .flatMap(a => traversal(a)) // recursive application
9     }
10    traversal(p).drop(1) // Remove the post itself of the result
11 }
12
13 def countLikes (p: Post) =
14  allComments(p)
15    .map(c => sn.triplets
16                .filter(t => t.dstAttr == c & t.attr == "like")
17                .count)
18    .sum
19
20 def score (p : Post) =
21    10 * allComments(p).size + countLikes(p)
```

Listing 3: Direct implementation of score

**MapReduce implementation** Listing 4 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 sub-comments, and the number of likes, for each element of the model, using a map and reduce-by-key composition (line 1 to 6). Because the number of likes do not the have same importance as 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 (lines 10 to 24). 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. One disadvantage of this implementation is its non-reactive aspect. Indeed, without additional mechanism, all the `getScore` function must be re-executed in case of change in the model.

**Pregel implementation** The third solution, namely *pregel*, proposed in Listing 5, 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

```
def getScore(): Array[((String, VertexId), Long)] = {
   sn.triplets.filter(t => t.attr == "like" || t.attr == "comment")
        .map(t => if (t.attr == "like") ((t.attr, t.srcId), 1L)
                else ((t.attr, t.srcId), 10L))
        .reduceByKey((a, b) => a + b).collect
}

def score(p: Post) = {
   val individual_scores = getScores(sn)
   def traversal(s: Submission): Long = {
      val default = ((_, _), 0L)
      val valLike = individual_scores
                       .find(e => e._1 == ("like", pid))
                       .getOrElse(default)._2
      val valCom = individual_scores
                       .find(e => e._1 == ("comment", pid))
                       .getOrElse(default)._2
      var current_score = valCom + valLike
      sn.triplets
        // Get all direct comments of s as vertices
        .filter(t => t.srcAttr == s
                   & (t.attr == "like" || t.attr == "comment"))
        .map(_.dstAttr).collect
        // recursive application
        .map(traversal)
        // accumulation of sub scores
        .foreach(score => current_score = current_score + score)
      current_score
   }
   traversal(p)
}
```

Listing 4: Highly parallel implementation of score

works as follows. At each iteration, the function *mergeMsg* accumulates into a single value the incoming messages (lines 31 and 32), that are stored in an 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 $v_{n+1}$. In addition to this new vertex data, messages are generated with *sendMsg* and sent to vertices through edges for the next iteration. An empty message is 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 boolean, specifying if the sending vertex has been reached during the pregel execution. 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 (line 34) initialize the graph by tupling the vertex values with a boolean specifying if the vertex has been reached. At the first step, only the source (the input `Post`) is reached. Then an initial message `(false, false)` is sent to every vertex of the graph before the execution of `pregel`. At the end of the execution, the score of a post is calculated using the accumulator values.

### Discussion

**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

```
def score(p: Post) = {
  val nbComment = longAccumulator("comment_" + p.id)
  val nbLike = longAccumulator("like_" + p.id)

  def vprog(vid: VertexId, value: (Boolean, VertexType), msg:
     (Boolean, Boolean)) = {
    if (merged_msg._1 & !value._1)
      // reached for the first time
      if (merged_msg._2)
        // True -> it is a like
        nbLike.add(1L)
      else
        // False -> it is a comment
        nbComment.add(1L)
      // the vertex is now reached
      (true, value._2)
    else
      // Either it is still not reached, or already reached
         before.
      value
  }

  def sendMsg(triplet: EdgeTriplet[(Boolean, VertexType),
     EdgeType]): Iterator[(VertexId, (Boolean, Boolean))] = {
    var res =
    if (triplet.srcAttr._1 & !triplet.dstAttr._1) {
      if (triplet.attr == EDGE_COMMENT)
        res = Iterator((triplet.dstId, (true, false)))
      if (triplet.attr == EDGE_LIKE)
        res = Iterator((triplet.dstId, (true, true)))
    } else { Iterator.empty }
  }

  def mergeMsg(m1: (Boolean, Boolean), m2: (Boolean, Boolean))
     : (Boolean, Boolean) = m1

  val initGraph = sn.mapVertices((id, v) => (id == pid, v))
  initGraph.pregel(initialMsg = (false, false))(vprog, sendMsg,
     mergeMsg)
  nbComment.value * 10 + nbLike
}
```

Listing 5: Pregel implementation of score

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 belong 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 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 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, what impact the cost of parallel operations has. Thus, choosing the right paradigms can have a huge impact on performances.

**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 of Section 3.1, 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.

Finally, the two first strategies (i. e., naive and highly-parallel) adopted a depth-first search strategy for their traversal functions. The functions are implemented as recursive functions, that uses 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[16].

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.

## Experiments

We present here 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 3 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 [17].

| Dataset | | | | | | | Speed-up (compared with 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 | 1064 | 118 | 24 | 251 KB | **x 1** | x 0.39 | x 0.46 | x 0.36 | x 0.44 | x 0.46 |
| 3 | 4 | 1845 | 2315 | 190 | 66 | 537 KB | **x 1** | x 0.51 | x 0.85 | x 0.68 | x 0.66 | x 0.71 |
| 4 | 8 | 2270 | 5956 | 204 | 129 | 983 KB | **x 1** | x 0.51 | x 2.34 | x 0.35 | x 0.15 | x 2.96 |
| 5 | 16 | 5518 | 9220 | 394 | 572 | 2 MB | **x 1** | x 4.25 | x 4.17 | x 5.21 | x 4.68 | x 4.03 |
| 6 | 32 | 10929 | 18872 | 595 | 1598 | 4.22 MB | **x 1** | x 4.68 | x 2.39 | x 2.83 | x 1.97 | x 3.91 |
| 7 | 64 | 18083 | 39212 | 781 | 4770 | 8.42 MB | **x 1** | x 4.07 | x 4.58 | x 4.12 | x 5.17 | x 3.27 |
| 8 | 128 | 37228 | 76735 | 1158 | 13374 | 17.1 MB | **x 1** | x 7.28 | x 7.61 | x 9.52 | x 9.66 | x 9.22 |

Table 3: 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

[16]https://tinyurl.com/yxb86zev
[17]https://tinyurl.com/y63tcl6b

balance with the communication time. Second, there is no unique solution that 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. At the contrary, on experiment 6, using Pregel seems to not 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 a formal proof to decide what 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, and larger, data-sets),
- the use of a distributed architecture with several nodes (e. g., Grid5000[18]).

[18]https://www.grid5000.fr/

# 4 Live Model Transformations in Reactive Applications

Reactive programming provides abstractions to express *event-driven* applications in which data and computation dependencies are managed automatically [59]. These applications react to *events* emitted by external *event sources* without an explicit notion of time or prior knowledge of the sequence of events.

LCDPs can be reactive applications, in which components of the platform communicate via *events* emitted by *event sources*. As illustrated in Figure 7, an *event source* can be a form on which the user is modeling the behavior of the application. An *event* is created by the form, as soon as the user adds a new input field, therefore extending the list of properties the corresponding entity has. This event is broadcasted to a diagram in the LCDP, that shows the architectural overview of the application, and automatically updates the figure based on the change received via the event.

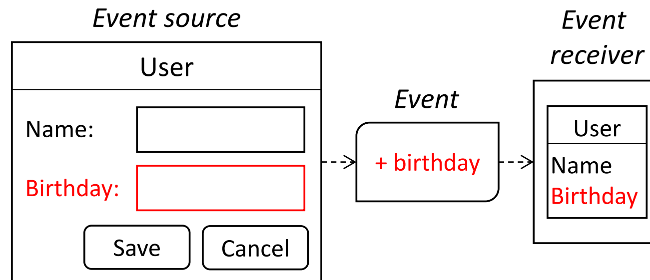


Figure 7: Reactivity example

In order to support this scenario, reactive model transformations are needed in LCDPs. Reactive model transformations adopt principles from reactive programming for model transformations. A model transformation rule consists of a *precondition*, that can be a graph pattern or model query, whose match on the source model activates the transformation action which translates the source model elements into target model elements. In the modeling environment, *events* are created from changes in the model, which cause new *matches* for the transformation *precondition* (pattern), which in turn *activates* the transformation *action*.

To realize reactive transformations live or incremental transformations have been adopted for reactive applications. Live or *incremental* model transformations are model transformations that update the target model, based on changes in the source model by maintaining a trace model between the source and target models (see Figure 8), and by caching the source model in memory. Therefore, if the source model changes, then the corresponding parts of the target model can be automatically updated, by minimizing the parts of the source model that need to be reexamined. This results in a faster execution time, compared to *batch* transformations where the whole model needs to be retransformed, upon a change in the source model.

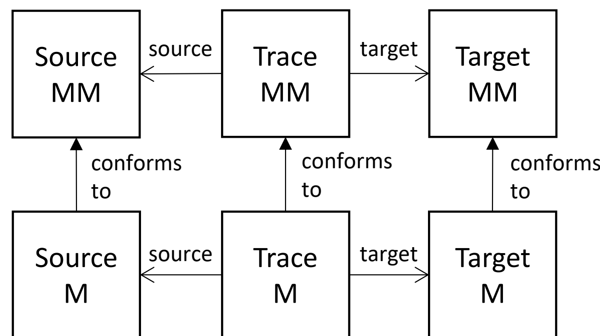


Figure 8: Relations between source, target, trace models (M) and metamodels (MM)

In this section we introduce how reactivity is realized in the VIATRA model transformation language (Section 4.1). After that, we propose an extension of reactive transformations to make them beneficial in low-code platforms, by addressing scalability challenges there (Section 4.2). Finally, we present the related work on incremental, reactive and parallel model query and transformation approaches (Section 4.3). Most of the work presented in this section have been published in [60].

## 4.1 Reactive transformations in VIATRA

Bergmann et al. proposed the Event-driven Virtual Machine (EVM) concept for reactive model transformations in VIATRA [61]. Figure 9 provides a black-box overview of EVM. The forthcoming detailed description of EVM is based on the work done by Bergmann et al. in [61].

The Event-driven Virtual Machine needs the *rule specifications*. Each *rule specification* consists of a model transformation *action*, that defines the commands to execute, and a *precondition* as a graph pattern, that should be found in the source model to activate the transformation. Besides, the EVM is waiting for *events*

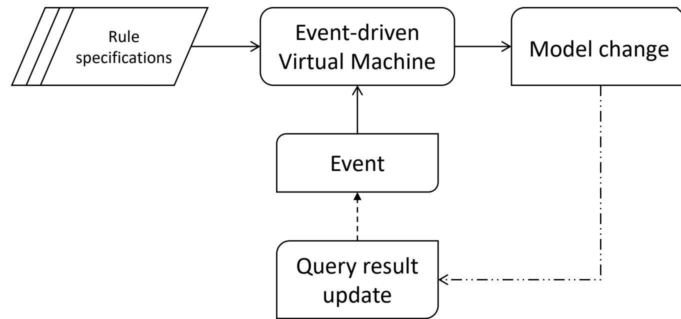


Figure 9: Black-box overview of the Event-driven Virtual Machine in VIATRA

from the application. Such events can be like the ones mentioned above, but they can also be events emitted by an incremental model query engine that is continuously looking for matches of the rule *preconditions* in the source model. If the query engine finds a match, then it updates its *query result set*, which results in the creation of an *event* that is propagated to EVM. If the EVM executes a transformation *action* then it results in a *change* in the target model. In case of endogenous transformations, this may result in a *query result update*, if the incremental query engine finds a match for a graph pattern in the source model, or if a previously found match is invalidated and disappears.

There is a scheduler inside EVM, which fires the transformation action, depending on the life-cycle of the *activation* (a match of a *precondition* in the source model). The *life-cycle* is a transition system, that consists of different *phases* (states), and event-triggered *transitions*. Each *activation* is associated with a *phase* in which in resides. If an event occurs, which triggers a transition, then the transformation action associated with this transition will be executed and its effects can be observed on the model. EVM provides two typical *life-cycles* by default: one with the events that are created in incremental pattern matching (e. g., match appeared, disappeared, a field of a graph pattern parameter got updated), and one that resembles batch transformations (e. g., rules are executed at most once, as soon as their precondition matches).

EVM provides an automated mechanism to resolve conflicting *activations* of a rule. Two activations of a rule are conflicting, if both of them are in an enabled state. In these cases the *conflict resolver* has to choose, for which *activation* should the rule be executed. Built-in conflict resolvers are: FIFO, LIFO, fair random choice, rule priority, interactive choice, custom conflict resolution algorithm [61].

Model transformation engineers can define custom *life-cycles*, using the events of the application for which they need to adopt the reactive transformations. Therefore, events which occur in the application can be directly use to trigger the execution of the transformations, which result in a reactive application.

## 4.2 Parallel reactive model transformations in VIATRA

### Motivation

Due to the online nature of low-code platforms, users expect them to be responsive, to complete complex operations in a short time. To develop responsive low-code platforms, we need scalable reactive model transformations that are able to quickly react to events which occur on the platform, e. g., derived views of the model need to be updated due to a change in the model. On the one hand, these transformations are executed automatically, if a triggering event occurs. On the other hand, if many events concurrently occur, and the transformation actions are long-running tasks, then congestion in their processing can arise quickly, which hinders the performance of the platform. To achieve scalable reactive model transformations on LCDPs, state-of-the-art reactive transformation approaches need to be improved.

In the model checking workflow, introduced in Section 2.2, `Users` and `Posts` have to be continuously synchronized with the target, formal model to check that the specified formal property is always satisfied by the model. In this way, we can recognize soon, if the social network evolves differently from expected.

Since the social media platform is used by hundreds of millions of users, the transformation of the `User` and their `Posts` should be separated from each other. Moreover, since each users submits vast amount of content (`Post`, `Comment`) on the platform, their transformations should be distributed between different processing units (transformation engines) to speed up the transformation process, keep the source and target models synchronized with each other.

Moreover, each interaction (e. g., user registration, posting `Submissions`, sharing `Posts`, liking `Comments`, etc.) on the social media platform, can be represented as an *Event*, and therefore processed by an event-driven transformation engine. It receives the change in the source model as an *Event*, checks if there is a *rule specification* whose *precondition* is satisfied by the change, and if so, then it executes the transformation *action*. In the following subsections we propose a scalable, parallel reactive model transformation engine to achieve this goal.

### Proposed approach

Parallelism is a frequently used means to speed-up the execution of independent data processing. In order to achieve better execution time of reactive model transformations in parallel, we are going to extend the EVM for task-parallel execution mode, as depicted in Figure 10.

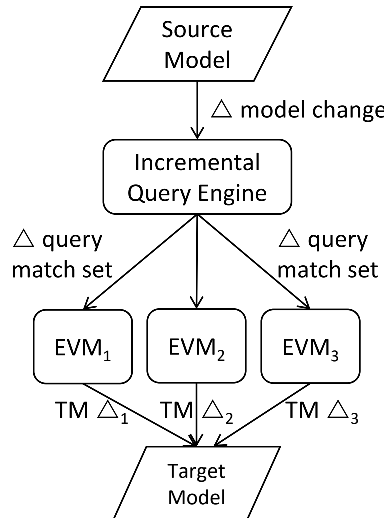


Figure 10: Parallel reactive model transformation in VIATRA

An incremental model query engine will be continuously looking for matches of transformation rule preconditions in the source model. To process the frequently occurring events in the application, we will have separate EVM instances. Each of them will be responsible for the execution of certain transformation *rule specifications* (tasks), which have different preconditions. The query engine incrementally propagates the matches of the preconditions in the source model, to every EVM instance. If an EVM instance has a *rule specification* with the same precondition as the one that is found, then it executes the corresponding transformation *action* which results in a *change* in the target model. These changes in the target model happen concurrently among the EVM instances, therefore conflicting changes need to be resolved and the target model needs to be synchronized to avoid inconsistencies.

Results obtained from the proposed approach will be published in future papers, including the refined execution mechanism, transformation examples and performance measurements to evaluate the scalability of the solution.

### Concrete example

Listing 6 shows the rule specification of a `User` using the Xtend language[19] in the VIATRA transformation framework. As a precondition to the rule, the `userInstance` simple graph pattern is referred to, that matches if a `User` is created in the source model. As an action, it initializes the `Template` with a new `VariableDeclaration` called `timestamp`. As the `Users` cannot delete themselves from the platform, there is no corresponding transformation action. In order to note which `Template` is created from the `User`, a traceability link is created. This link will be used in Listing 7 to find the corresponding `Template` in the target model.

```
1  val userRule = createRule()
2   .name("userTransformation")
3   .precondition(userInstance)
4   .action(ActivationStateEnum.CREATED) [
5    // create a Template and VariableDeclaration
6    val template = create(Template);
7    val variable = createChild(template, Template.Declarations, VariableDeclaration);
8    variable.name = "timestamp";
9
10   // create traceability link
11   val trace = createTrace(user, template);
12   trace.target.add(variable);
13  ].build();
```

Listing 6: User's rule specification

Listing 7 depicts the rule specification of a `Post`. As a precondition to the rule, the `postInstance` simple graph pattern is referred to, that matches if a `Post` is created or deleted in the source model. If a `Post` is submitted to the platform, then a new `Edge` is created which connects the last `Location`

[19] http://www.xtend-lang.org

of the Template with the newly created one. Besides, a FieldUpdate is also created which sets the timestamp VariableDeclaration for the UNIX time epoch of the Post creation. Finally, traceability links are instantiated between the Post and the newly created elements in the target model. If a Post is deleted from the platform, then the connections between the Locations are corrected, due to the to-be-removed Location. Finally, all the elements that were created from the Post are removed from the target and the traceability models.

```
val postRule = createRule()
 .name("postTransformation")
 .precondition(postInstance)
 .action(ActivationStateEnum.CREATED) [
  // create a new Template and VariableDeclaration in the target model
  val template = getTrgTrace(post.submitter) as Template;
  val edge = createChild(template, Template.Edge, Edge);
  val fieldUpdate = createChild(edge, Edge.Action, FieldUpdate);
  fieldUpdate.name = "timestamp";
  fieldUpdate.value = toUnixEpoch(post.timestamp);
  val newLocation = createChild(template, Template.Location, Location);
  edge.target = newLocation;
  edge.source = findLastLocation(template);

  // create traceability link
  val trace = createTrace(post, edge);
  trace.target.addAll(location, fieldUpdate);
 ].action(ActivationStateEnum.DELETED) [
  // find edge in target model
  val edge = getTrgTrace(post) as Edge;
  val location = edge.target;
  val nextEdge = findSourceEdgeOf(location)
  if (nextEdge != null){
     nextEdge.source = edge.source;
  }

  // remove edge and location
  val template = edge.parentTemplate;
  template.edge.remove(edge);
  template.location.remove(location);

  // remove traceability link
  val trace = getTrace(post);
  removeTrace(trace);
 ].build();
```

Listing 7: Post's rule specification

To illustrate the task-parallel reactive approach proposed above, the aforementioned rule specifications can be assigned to different EVM instances, each running on a separate thread. Separate instances are going to be monitoring the userInstance and postInstance precondition patterns' match set and run the corresponding actions. Since the update transformation actions of the postRule depend on the userRule, namely the Template of the User must exist in the target model, thus their executions should be scheduled accordingly. Besides, the target model should be handled in transactions to guarantee its consistency.

### Challenges and open questions

To realize the proposed parallel reactive model transformations in VIATRA, there are several practical and theoretical challenges that need to be overcome:

- Dependencies between the model transformation rules should be discovered to find the independent ones [62], that can be applied in parallel;
- The concurrent editing of the target model from multiple EVM instances requires transactional model processing and locking mechanisms ([63]) to guarantee its consistency;
- Alternatively, different lock-free mechanisms should be adopted for reactive transformations, e. g., Operational Transformation and Conflict-Free Replicated Data Types [64, 65] should be adopted for reactive transformations;
- The allocation of *rule specifications* to the EVM instances is crucial to achieve equal load distribution and avoid resource starvation between them;
- If the task-parallel execution does not provide satisfactory results, then the data-parallel approach has to be studied for reactive transformations.
- Finally, declarative languages and advanced static analysis techniques should be exploited to derive efficient imperative transformation code (e. g., Listing 6 and Listing 7).

| | Model Query | Model Transformation |
|---|---|---|
| Incremental | [66] | [29, 67] |
| Reactive | - | [7, 61] |
| Parallel | [68, 36] | [37, 38, 69, 70] |

Table 4: Incremental, reactive, parallel query and transformation approaches

### 4.3 Related work

In order to address scalability in model query and transformation, several execution models have been proposed and implemented. From the possible approaches, Table 4 enumerates the incremental, reactive and parallel directions as they are the most related ones to the approach proposed in Section 4.2.

#### Model query

In order to achieve scalable model transformations for very large models, scalable model queries should be employed. To this end, several approaches have been developed in the literature.

Bergmann et al. proposed incremental graph pattern matching on EMF models [66] by implementing the RETE algorithm [10]. In the RETE algorithm, a network of nodes is constructed from the graph pattern. In the network, each node caches the matches of the subpattern they are assigned to. The benefit of the algorithm is the incrementality and the ability to react to changes in the source model. However, intensive caching causes a large memory footprint, and updating the RETE network has computation complexity also.

In order to improve the performance of the RETE network, Bergmann et al. proposed a parallel implementation of incremental pattern matching [68]. They split the RETE network into containers, where each of them is responsible for matching a set of subpatterns. Each container runs on a separate thread and communicates via message queues. The advantage of this approach is, the update propagation of the network is spread between the containers, thus the computation can complete faster, than in the single-threaded implementation.

To improve the evaluation of OCL queries in EOL [71], Madani et al. provided parallel implementations for first-order OCL operations in a data-parallel approach by creating a job for each model element and submitting it to a thread pool executor [36]. Furthermore, they extended these operations with short-circuiting thus further improving the processing time.

#### Model transformation

In *target incremental* transformations target models are updated according to changes in the source model [29, 67]. Incremental transformations are usually executed faster, than batch transformations, which recompute the whole target model. Reactive model transformations are executed as reactions for events emitted by event sources, and they can combine incrementality with lazy evaluation [7].

Model transformations are computation heavy operations. In order to make them more scalable, different parallelization techniques have been proposed.

Tisi et al. implemented a *task-parallel* engine for ATL in which each thread executes a different transformation rule and works on the whole source and target models [37]. Due to several constraints of the ATL language, the application of transformation rules is highly independent of each other, which is beneficial for the parallel execution of transformations. However, due to technical restrictions of the Eclipse Modeling Framework they used, target element creation, adding values to multi-valued collections, creating and reading traceability links need to be synchronized.

Burgueno et al. introduced the LinTra framework for model-to-model (M2M) transformations [38, 69]. The framework adopts the principles of the Linda coordination language that follows the Blackboard approach. In this approach, processes communicate via tuples in shared memory. LinTra cuts the source model into partitions and transforms them in a *data-parallel* way in a master-slave architecture. The master process coordinates the work of the slaves that execute the transformation rules on the model partitions [72].

Mezei et al. proposed a parallelization approach based on the offline dependency check of transformation rules [70]. Two transformation rules are in *metaconflict* if their match may conflict according to the metamodel items used in the rules. Those rules that are not in *metaconflict* can be grouped into independence blocks because they can be executed in parallel. Consecutive blocks may not be conflict-free, thus they implemented several heuristics to avoid and resolve conflicts in VMTS [73].

# 5 Composition of Model Transformations

This section focuses on the data and service workflows while elaborates on the proposed steps of utilizing the concept of model transformation compositions in defining the specified workflows within and across LCDPs and other external services.

## 5.1 Models to be transformed

The area in an LCDP where the model transformations are applied are discussed in the paper [74].

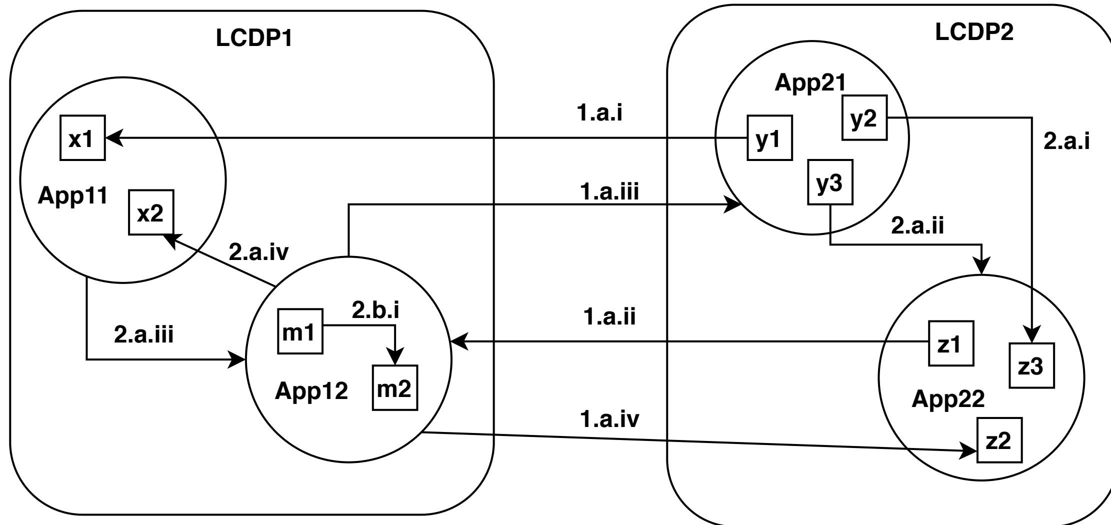


Figure 11: Model Transformation Area in LCDP/external services

Model transformations and its compositions can be used in the following situation as described in Figure 11. In this Figure, two LCDPs are considered as LCDP 1 and LCDP 2. In LCDP 1, there are two applications shown as circles called App11 and App12. Inside these applications, there will be few artifacts which is a page or a part of a page or a workflow of an application. Inside App11, there are two artifacts shown as squares named as x1 and x2. Inside App12, there are also two artifacts shown in squares called m1 and m2. Likewise, in LCDP 2, there are two applications shown as circles called App21 and App22. Inside App21, there are three artifacts show in squares called as y1, y2 and y3. Similarly in App22, there are also three artifacts shown in squares called as z1, z2 and z3. Different types of areas in a low-code platform where the model transformation may takes place are as follows.

1. Inter LCDP: An application or artifact made in one LCDP is transformed into another LCDP or other external services. Such LCDPs should be open source as it can be strategically used to handle lock-in, interoperability and long-term maintainance of software [75]. It must take place as:
   (a) Inter application: There are four sub-cases in which model transformation can take place across applications through different low-code platforms. They are:
      i. Artifact to artifact - An artifacts in one application within one LCDP can be transformed or moved into an artifact in another application in different LCDP. The transformation arrow is shown from y1 to x1.
      ii. Artifact to app - An artifact in one application within one LCDP can be transformed or moved into an application in different LCDP. The transformation arrow is shown from z1 to App12.
      iii. App to app - An application within one LCDP can be transformed or moved into another application of different LCDP. The transformation arrow is shown from App12 to App21.
      iv. App to artifact - An application within one LCDP can be transformed or moved into an artifact in another application in different LCDP. The transformation arrow is shown from App12 to z2.
2. Intra LCDP: Model Transformation takes place within a low-code platform. They are of two types.
   (a) Inter application: An application or a subset of an application is reused to build different applications within the same LCDP. In this case also, there are four different sub-cases. They are:
      i. Artifact to artifact - An artifacts in one application can be transformed or moved into an artifact in another application in the same LCDP. The transformation arrow is shown from y2 to z3.
      ii. Artifact to app - An artifact in one application can be transformed or moved into an application in the same LCDP. The transformation arrow is shown from y3 to App22.

iii. App to app - An application can be transformed or moved into another application within the same LCDP. The transformation arrow is shown from App11 to App12.
iv. App to artifact - An application can be transformed or moved into an artifact in another application in the same LCDP. The transformation arrow is shown from App12 to x2.

(b) Intra application: A reusable artifact such as prebuilt forms, reports, etc., used in an application is transformed from one view to another. These views are grid view, Kanban view, CSV, PDF, etc. It is of only one type.
i. Artifact to artifact - An artifact can be transformed or moved to another artifact within the same application in an LCDP.

## 5.2 Proposed steps to achieve workflows using model transformation composition

This subsection elaborates on the steps required to compose several model transformations in order to support the specification of complex workflows in LCDPs as referenced in the paper [74]. Specifically, we look at the possible interactions occurring *intra* and *inter* LCDPs as presented in Section 5.1 as proper orchestrations of different services. If we can manage such services as model transformations, then we can reuse the theories underpinning existing composition approaches for model transformations. In this respect, Figure 12 depicts the main components of envisioned approach which is detailed in the following.

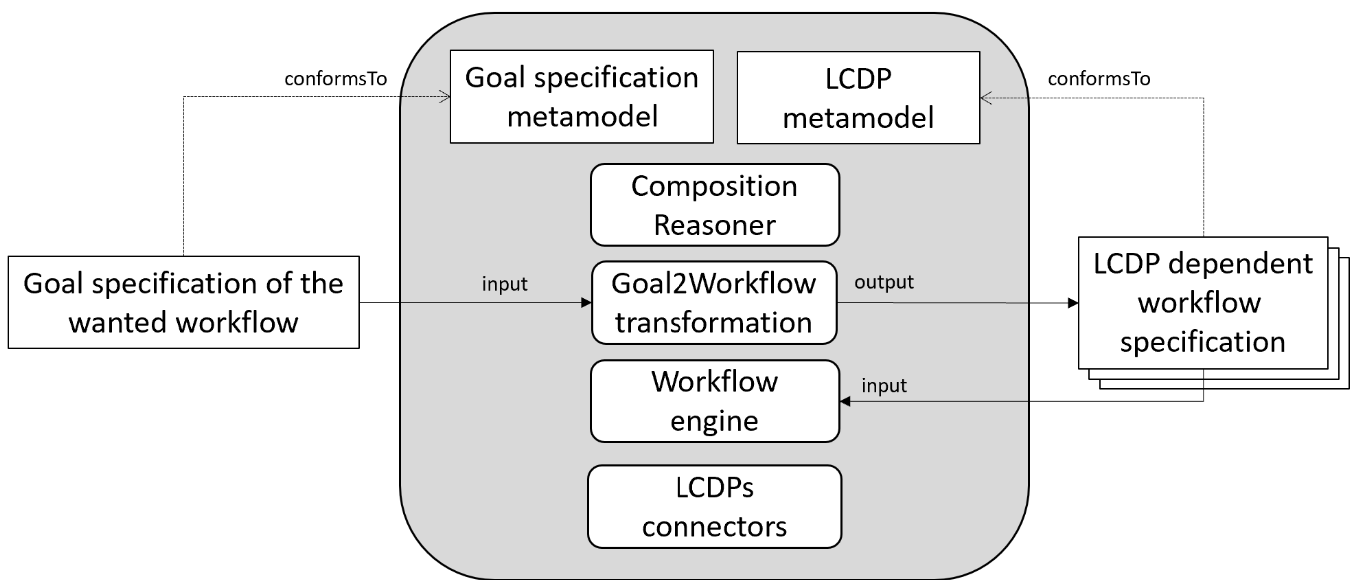


Figure 12: Proposed approach

*Goal specification metamodel:* A technique to chain model transformation is employed similar to the proposed technique in [76]. In that case, the desired model conforms to the goal specified by the user that consists of target metamodel. The approach is able to identify any possible transformation chains considering the given goal and the input model. Likewise, we aim to provide the users of LCDPs with the means to specify the properties of the targeted workflows at a higher level of abstraction. The tools and languages will allow to specify constraints, functional, and non-functional requirements that the desired workflow should satisfy. Example of goal is that the user wants to take some input model and visualize it by means of two target views, i.e., grid view and Kanban view. It is necessary to define a customized modeling language such as BPMN (Business Process Model and Notations) that provides users with all the modeling constructs formalized in the metamodel in order to enable the adoption of the metamodel.

*LCDP metamodels:* We, further plan to define metamodels for specifying properties of the supported LCDPs. The aim is to specify workflow models which can be executed by the corresponding LCDP. The specification of such metamodels mandates the analysis of different LCDPs with the aim to identify the distinct characteristics with regards to the provided mechanisms to specify and execute complex workflows [23]. The metamodels are classified as follows. The main application metamodel of the whole LCDP is mapped to view-specific metamodels. They are design-time view and run-time view, that correspond to the static analysis of the application model before the deployment, and to run-time analysis of the application model after its deployment, respectively. Such views store only those data that are relevant to its specific view [77]. The citizen developers should only see these view models individually at the design time or the run-time and not the whole of LCDP's application model. These separation of views allow the citizen developer to focus on either of the views without much worrying about the overall expressiveness and flexibility of the application model of a particular LCDP.

*LCDPs connectors:* They are the software components that allow the system to connect to the different LCDPs by relying on provided APIs.

*Composition Reasoner:* This component checks the feasibility of the input goal with respect to the available services that are provided by the LCDPs which the proposed system is able to connect. The list of such services is retrieved and kept updated by relying on the available LCDPs connectors. For instance, considering simple goal specification given previously, the composition reasoner would check if the available LCDPs can manage services' view types like grid and Kanban views.

*Goal2Workflow transformation:* By depending on the outcome of the *Composition Reasoner* and considering the available LCDPs connectors, this component will generate possible workflows that are compatible with the specified goals. Like model transformation compositions, multiple solutions are possible and the proposed system will provide the user with a ranked list. By considering the previous example, the component would show all the possible workflows that allow to generate grid and Kanban views out of a source model. In case there are more than one services (even provided by different LCDPs) that are able to manage grid and Kanban view, the component would produce all the possible compositions.

*Workflow engine:* It executes models generated by the *Goal2Workflow transformation*. The engine interacts with different LCDPs that allows the execution of workflows via some exposed APIs.

To sum it up, Figure 12 specifies different goal specifications which will be recognized by the source model, the intermediate model and the target model that need to be transformed. This *goal specification metamodel* is used to implement modeling languages. Further, in order to develop an application landscape that also includes static and run-time perspectives of the LCDP. This enables to specify goal-specific workflows. The third point of the figure shows the *composition reasoner* which defines the need for composing different model transformations to achieve a workflow or a part of workflow. The *Goal2Workflow transformation* helps to achieve the targeted workflows that are intended to be implemented. These dependent workflows need to be orchestrated across applications of different LCDPs using a *workflow engine*. Lastly, an LCDP connector is built to interoperate different services offered by different platforms. Many services such as APIs need to be published or consumed within or across LCDPs.

Therefore, this research approach helps to identify some of the prominent research challenges as to how one service in an LCDP can be considered as a model transformation, and how a workflow of services can be achieved by different model transformation compositions. This aims to achieve better reusable and interoperable services in an LCDP.

## 5.3 Related work

This subsection elaborates the related works on composition of model transformation, usage of model transformation in distributed environment and how composition of models are linked to distributed environment.

#### Model transformation composition approaches

Basciani et al. (2018a, 2018b) [76, 78] 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, finding 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 source model and target meta-model. (ii) Finding available transformation chain lists. (iii) Select the optimal chain and execute it. Step (ii) and (iii) comprises of Model Transformation Composition Language (MTCL).

Basciani et al. (2014) [79] took user input as source model and target metamodel. It can be described as retrieving source metamodel and finding out all the possible chain transformations. It checks if the source and target metamodels are incompatible, then an intermediate adapter is automatically generated to fill the gap in between the inconsistencies between the metamodels. The processes in this method are (i) find source model and target meta-model. (ii) Build MTCL by creating an intermediate adapter between incompatible metamodels.

Etien et al. (2015) [80] 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 use localized transformation to check the desired outcomes according to the objectives of the application. The processes in this method are (i) Find out the granularities of the large model. (ii) Build localized transformation and combine those transformations with the help of MTCL.

Aranega et al. (2012) [81] 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 transformation to implement the desired objectives. The processes in this method are (i) Find out the granularities of the large model. (ii) Build MTCL for a consistent set of model transformation chains.

Etien et al. (2012) [82] 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 behavior 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 in this method is to find out the best possible model transformation chain.

Etien et al. (2010) [83] took user input as models. It can be described as combining independent model transformation that jointly works 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. (2010, 2008) [84, 85] took user input as two model transformation language (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 provides 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. (2009) [86] 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. The process in this method is to find out the best transformation chain by statically analyzing transformation that comprises MTCL.

Rivera et al. (2009) [87] 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. (2006) [88] took user input as metamodel. It can be described as proposing metamodels for a transformation chain modeling language that enables the implementation-independent composition of transformation 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 modeling language that comprise an MTCL.

**State-of the-art in model transformation in a distributed environment**

The work in a PhD thesis done by Amine Benelallam [89] focuses on handling the scalability issues that exploit the wide availability of distributed clusters in the Cloud for the distributed execution of model transformations and their decentralized model persistence. The thesis proposes an approach for scalable model transformation and persistence by exploiting the high-level abstraction of relational model transformation languages and the well-defined semantics of existing distributed programming models to provide a relational model transformation engine with implicit distributed execution [46, 90]. This proposed approach is extended with an efficient model distribution algorithm based on the analysis of relational model transformation and results on balanced partitioning of streaming graphs [91]. This approach is applied to ATL transformation language on top of MapReduce distributed programming model. Finally, this approach also proposes a multi-persistence backend for manipulating and storing models in NoSQL databases according to the modeling scenario.

The research paper written by Jurack et al. [92] consider an approach to composite models for largely independent teams and their transformation based on graph transformation concepts. In this paper, Eclipse Modeling Framework (EMF) sets up a setting of composite models that can be distributed over several sites. Further, the paper shows composite models with explicit and implicit interfaces using concepts of distributed graph tranformation and outlines different kinds of composite model transformation.

The paper done by Mallet et al. [93] proposes a model-driven framework (to explicitly specify distributed architectural styles such as client-server, publish-subscribe, and peer-to-peer) as independent models of the application functionalities. Based on the process calculus, a formal design process is shown that enables architectural solutions to be generated by an endogeneous transformation model. Based on an expandable repository of architectural styles, a functional model of an application could be systematically composed with alternative styles for further analysis before development of an application.

A journal paper written by Burgueno et al. [94] presents a solution that provides concurrency (parallel) and distributed execution to model transformation. A novel coordination language called Linda is elaborately discussed in the paper [95]. In paper [94], a novel Java-based execution platform is introduced to achieve parallelization by parallel execution of the core features of the transformation. This parallel execution will be an out-place (have a distinct source model, target model and a file for storing the status of the transformation execution) transformation that can be used as a target for high-level transformation language compiler. This Linda-based tool enables the concurrent execution of model transformations that can serve as a platform for their scalable and efiicient implementation in parallel and distributed environment.

The research paper done by Rabbi et al. [96] introduces a new web-based metamodelling and model transformation tool called WebDPF (Web Diagram Predicate Framework). WebDPF supports multilevel diagrammatic development and analysis of model transformation that exploits auto-completion of partial

models which enhances modelling efficiency, and provides execution sematics for workflow model. This WebDPF involves a scalable model navigation facility that helps users to inspect and query large models.

**How distributed environment is linked with model composition**

Composition of model transformations is expected to be executed by heterogeneous tools and techniques across different platforms. These platforms can be remotely accessed distributively. Reusing and composing smaller model transformations distributively is a challenging task in preserving the syntactic and semantic characteristics across different platforms. The smaller model transformations need to be maintained in a model repository by applying the techniques of distributed systems. The model transformation is also a model that leads to clear system architecture, efficient implementation, high scalability and good flexibility [97].

One of the distributed techniques of using model compositions are described in the paper [98]. In paper [98], the problem of model compositions are addressed when data sources including models and data are distributed across multiple sites and have different scopes. Decision making is used when these data and model resources are leveraged to support the composition and execution of the sequence of models in response to a particular decision making situation. Also, in this paper, a system architecture is presented which facilitates automation during model composition and execution while enabling the distribution and implementation of the data/model to be transparent to the user.

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 artifacts, such as model transformations, need to be available in a distributed manner so that an user can efficiently access any reusable artifacts according to their needs from remote locations.

# 6 Conclusion

In this deliverable, we introduced low-code delevopment 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 level to define complex software systems as fully operational applications. We focused on the underpinning techniques, such as model queries and transformations, and their requirements for scalability and quick response time to meet the users' needs. In Section 2 we showed these needs in a motivating case study from social media.

In Section 3, we made an overview of what, and how, execution strategies can be used for model driven engineering. We mainly focused on distributed approaches that can be used to improve scalability of programs. 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 2.1 and the implementations of Section 3.2 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. These differences are more illustrated with the experiments conducted in Section 3.2. 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.

Section 4 introduced reactive model transformations for LCDPs as a means to improve the response time of these platforms. First, it elaborated on the state-of-the-art reactive model transformation engine (EVM) in the VIATRA model transformation framework. After that we motivated the need for a scalable reactive transformation engine in LCDPs by adopting the running example. Finally, we proposed a task-parallel extension of EVM, by distributing the model transformation tasks between the EVM instances with a goal to achieve higher throughput and lower response times on a platform with frequently occurring events.

Section 5 refered the four subparts that comprise of the specification of complex workflows within and across applications to be applied within or across platforms or external services. The goal is to support citizen developers by providing them modeling constructs that permit to specify the goal of the desired workflows at a high-level of abstraction. By relying on the techniques and tools developed for composing model transformations, the idea is to generate possible workflows that satisfy the initial goal. The second subpart showed the proposed plan on applying model transformation composition in a distributed low-code environment. The plan focuses on the modeling languages such as goal and workflow specification languages. They have to be defined in an iterative manner by specifying real situations and refine the available constructs in case of errors or to cover unforeseen requirements. The third subpart elicits on a related survey on internal and external compositions of model transformations while the fourth subpart briefly cites some of the state-of-the-art works on model transformations in a distributed environment. It is very important to consider the parallel and distributed execution of compositions of model transformations so as to handle complex workflows over a large set of models in a huge applications within or across different platform.

### Future work

As future work for the multi-paradigm strategies, we will create an engine that provides multiple execution strategies. The goal of creating a prototype of distributed queries is to drive a complete study of how paradigms can be used and combined to classify them depending on use cases. Also, additional experiments will be conducted with increasing input size on distributed architectures.

For reactive model transformations, we aim to realize the proposed parallel reactive model transformation engine for VIATRA, to achieve better response times and higher throughput on a platform with frequently occurring events.

As future works for model transformation compositions in relation to LCDPs, we will focus on the development of the workflow engine and all the dependent components including the reasoner, and the goal to workflow transformation of the proposed plan. We aim to implement the workflow (transformation) engine to compose various models in a scalable cloud-based environment such as IoT systems, social network platforms, etc. Further, we need to orchestrate the transformation chains that implements the interoperability features of a platform and managing different paradigms of model transformation languages applied in a distributed environment.

# 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] Robert Waszkowski. Low-code platform for automating business processes in manufacturing. *IFAC-PapersOnLine*, 52(10):376–381, 2019.

[6] 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.

[7] Salvador Martínez Perez, Massimo Tisi, and Rémi Douence. Reactive model transformation with ATL. *Sci. Comput. Program.*, 136:1–16, 2017.

[8] Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos Matragkas, Richard F. Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan de Lara, István Ráth, Dániel Varró, Massimo Tisi, and Jordi Cabot. A research roadmap towards achieving scalability in model driven engineering. In *Proc. of the Workshop on Scalability in Model Driven Engineering*, page 2. ACM, 2013.

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

[10] Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligences*, 19(1):17–37, 1982.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 137–149, Berkeley, CA, USA, 2004. USENIX Association.

[12] Christian Krause, Matthias Tichy, and Holger Giese. Implementing graph transformations in the bulk synchronous parallel model. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering*, pages 325–339, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[13] Raquel Sanchis, Óscar García-Perales, Francisco Fraile, and Raul Poler. Low-code as enabler of digital transformation in manufacturing industry. *Applied Sciences*, 10(1):12, 2020.

[14] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, Aug 2015.

[15] Derrick Harris. Facebook's trillion-edge, hadoop-based and open source graph-processing. Aug 2013.

[16] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM Workshop on Online Social Networks*, WOSN '09, page 37–42, New York, NY, USA, 2009. Association for Computing Machinery.

[17] 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.

[18] Dimitris S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. On the evolution of OCL for capturing structural constraints in modelling languages. In Jean-Raymond Abrial and Glässer Uwe, editors, *Rigorous Methods for Software Construction and Analysis, Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday*, volume 5115 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2009.

[19] 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).

[20] 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.

[21] Stefano Schivo, Bugra M. Yildiz, Enno Ruijters, Christopher Gerking, Rajesh Kumar, Stefan Dziwok, Arend Rensink, and Mariëlle Stoelinga. How to efficiently build a front-end tool for UPPAAL: A model-driven approach. In *Proc of the 3rd International Symposium on Dependable Software Engineering*, volume 10606 of *LNCS*, pages 319–336. Springer, 2017.

[22] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick P Bloem. *Handbook of model checking*. Springer, 2018.

[23] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 171–178. IEEE, 2020.

[24] Javier Troya, José M Bautista, and Antonio Vallecillo. A rewriting logic semantics for atl (extended version).

[25] Nafiseh Kahani, Mojtaba Bagherzadeh, James R Cordy, Juergen Dingel, and Daniel Varró. Survey and classification of model transformation tools. *Software & Systems Modeling*, 18(4):2361–2397, 2019.

[26] Edward D Willink. A text model-use your favourite m2m for m2t. In *MODELS Workshops*, pages 89–102, 2018.

[27] 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.

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

[29] Théo Le Calvar, Frédéric Jouault, Fabien Chhel, and Mickael Clavreul. Efficient ATL incremental transformations. *Journal of Object Technology*, 18(3):2:1–17, Jul 2019. The 12th International Conference on Model Transformations.

[30] Jordi Cabot and Ernest Teniente. Incremental integrity checking of UML/OCL conceptual schemas. *Journal of Systems and Software*, 82(9):1459–1478, 2009.

[31] Varró, Gergely and Frederik Deckwerth. A rete network construction algorithm for incremental pattern matching. In Keith Duddy and Gerti Kappel, editors, *Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*, volume 7909 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 2013.

[32] Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, and Gergely Varró. Incremental pattern matching in the viatra model transformation system. In *Proceedings of the Third International Workshop on Graph and Model Transformations*, GRaMoT '08, pages 25–32, New York, NY, USA, 2008. Association for Computing Machinery.

[33] Massimo Tisi, Salvador Martínez Perez, Frédéric Jouault, and Jordi Cabot. Lazy execution of model-to-model transformations. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, volume 6981 of *Lecture Notes in Computer Science*, pages 32–46, Berlin, Heidelberg, 2011. Springer.

[34] Massimo Tisi, Rémi Douence, and Dennis Wagelaar. Lazy evaluation for OCL. In Achim D. Brucker, Marina Egea, Gogolla Martin, and Frédéric Tuong, editors, *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*, volume 1512 of *CEUR Workshop Proceedings*, pages 46–61. CEUR-WS.org, 2015.

[35] Edward D. Willink. Deterministic lazy mutable OCL collections. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations - STAF 2017 Collocated Workshops, Marburg, Germany, July 17-21, 2017, Revised Selected Papers*, volume 10748 of *Lecture Notes in Computer Science*, pages 340–355. Springer, 2017.

[36] Sina Madani, Dimitris S. Kolovos, and Richard F. Paige. Towards optimisation of model queries: A parallel execution approach. *Journal of Object Technology*, 18(2):3:1–21, July 2019. The 15th European Conference on Modelling Foundations and Applications.

[37] Massimo Tisi, Martínez Salvador Perez, and Hassene Choura. Parallel execution of ATL transformation rules. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke, editors, *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, volume 8107 of *Lecture Notes in Computer Science*, pages 656–672. Springer, 2013.

[38] Loli Burgueño, Javier Troya, Manuel Wimmer, and Antonio Vallecillo. Parallel in-place model transformations with LinTra. In Dimitris S. Kolovos, Davide Di Ruscio, Nicholas Drivalos Matragkas, Juan de Lara, István Ráth, and Massimo Tisi, editors, *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.*, volume 1406 of *CEUR Workshop Proceedings*, pages 52–62. CEUR-WS.org, 2015.

[39] Murray Cole. *Algorithmic skeletons : a structured approach to the management of parallel computation*. PhD thesis, University of Edinburgh, UK, 1988.

[40] Jolan Philippe and Frédéric Loulergue. PySke: Algorithmic Skeletons for Python. In *The 2019 International Conference on High Performance Computing & Simulation (HPCS)*, Dublin, Ireland, July 2019.

[41] Hélène Coullon and Sébastien Limet. The sipsim implicit paral-

lelism model and the skelgis library. *Concurrency and Computation: Practice and Experience*, 28(7):2120–2144, 2016.

[42] Grzegorz Malewicz, Matthew H. Austern, Aart J.C 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*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[43] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *IEEE Solid-State Circuits Society Newsletter*, 12(3):19–20, Summer 2007.

[44] Hélène Coullon, Julien Bigot, and Christian Pérez. Extensibility and Composability of a Multi-Stencil Domain Specific Framework. *International Journal of Parallel Programming*, November 2017.

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

[46] Amine Benelallam, Abel Gómez, Massimo Tisi, and Jordi Cabot. Distributing relational model transformation on MapReduce. *Journal of Systems and Software*, 142:1–20, 2018.

[47] Stefan Jurack and Gabriele Taentzer. A component concept for typed graphs with inheritance and containment structures. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *Graph Transformations - 5th International Conference, ICGT 2010, Enschede, The Netherlands, September 27 - - October 2, 2010. Proceedings*, volume 6372 of *Lecture Notes in Computer Science*, pages 187–202. Springer, 2010.

[48] Gábor Imre and Gergely Mezei. Parallel graph transformations on multicore systems. In Victor Pankratius and Michael Philippsen, editors, *Multicore Software Engineering, Performance, and Tools*, pages 86–89, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[49] Gergely Mezei, Tihamer Levendovszky, Tamás Mészáros, and István Madari. Towards truly parallel model transformations : A distributed pattern matching approach. pages 403–410, 05 2009.

[50] Le-Duc Tung and Zhenjiang Hu. Towards systematic parallelization of graph transformations over pregel. *Int. J. Parallel Program.*, 45(2):320–339, April 2017.

[51] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. Incquery-d: A distributed incremental model query framework in the cloud. In *Proc. of the 17th International Conference on Model-Driven Engineering Languages and Systems*, volume 8767 of *LNCS*, pages 653–669. Springer, 2014.

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

[53] Tamás Vajk, Zoltán Dávid, Márk Asztalos, Gergely Mezei, and Tihamér Levendovszky. Runtime model validation with parallel object constraint language. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa, New York, NY, USA, 2011. Association for Computing Machinery.

[54] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[55] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996.

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

[57] Loli Burgueño, Manuel Wimmer, and Antonio Vallecillo. Towards distributed model transformations with LinTra. *Jornadas de Ingeniería del Software y Bases de Datos*, pages 1–6, 2016.

[58] 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.

[59] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Computing Surveys*, 45(4):52:1–52:34, 2013.

[60] 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.

[61] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. Viatra 3: A reactive model transformation platform. In Dimitris S. Kolovos and Manuel Wimmer, editors, *Theory and Practice of Model Transformations*, pages 101–110, Cham, 2015. Springer International Publishing.

[62] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars (A survey). In *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69. Springer, 1978.

[63] Csaba Debreceni, Gábor Bergmann, István Ráth, and Dániel Varró. Property-based locking in collaborative modeling. In *MODELS*, pages 199–209. IEEE Computer Society, 2017.

[64] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proc of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *LNCS*, pages 386–400. Springer, 2011.

[65] David Sun, Chengzheng Sun, Agustina Ng, and Weiwei Cai. Real differences between OT and CRDT in correctness and complexity for consistency maintenance in co-editors. *PACMHCI*, 4(GROUP):021:1–021:30, 2020.

[66] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. Incremental evaluation of model queries over EMF models. In *MODELS*, volume 6394, pages 76–90. Springer, 2010.

[67] 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, July 2016.

[68] Gábor Bergmann, István Ráth, and Dániel Varró. Parallelization of graph transformation based on incremental pattern matching. *Electronic Communications of the EASST*, 18, 2009.

[69] Loli Burgueño, Manuel Wimmer, and Antonio Vallecillo. A linda-based platform for the parallel execution of out-place model transformations. *Information and Software Technology*, 79:17–35, 2016.

[70] Gergely Mezei, Tihamer Levendovszky, Tamas Meszaros, and Istvan Madari. Towards truly parallel model transformations: A distributed pattern matching approach. In *IEEE EUROCON 2009*, pages 403–410. IEEE, 2009.

[71] 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.

[72] Loli Burgueño, Eugene Syriani, Manuel Wimmer, Jeffrey G. Gray, and Antonio Vallecillo. Lintrap: Primitive operators for the execution of model transformations with lintra. In *Proc. of the 2nd Workshop on Scalability in Model Driven Engineering*, volume 1206 of *CEUR-WS Proceedings*, pages 23–30. CEUR-WS.org, 2014.

[73] Tihamer Levendovszky, László Lengyel, Gergely Mezei, and Hassan Charaf. A systematic approach to metamodeling environments and model transformation systems in VMTS. *Electronic Notes in Theoretical Computer Science*, 127(1):65–75, 2005.

[74] 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.

[75] Björn Lundell, Jonas Gamalielsson, Stefan Tengblad, Bahram Hooshyar Yousefi, Thomas Fischer, Gert Johansson, Bengt Rodung, Anders Mattsson, Johan Oppmark, Tomas Gustavsson, et al. Addressing lock-in, interoperability, and long-term maintenance challenges through open source: How can companies strategically use open source? In *IFIP International Conference on Open Source Systems*, pages 80–88. Springer, Cham, 2017.

[76] 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.

[77] Nick Jansen. Exploring interactive application landscape visualizations based on low-code automation. Master's thesis, 2019.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] Amine Benelallam. *Model transformation on distributed platforms: decentralized persistence and distributed processing*. PhD thesis, Nantes, Ecole des Mines, 2016.

[90] Amine Benelallam, Abel Gómez, Massimo Massimo Tisi, and Jordi Cabot. Distributed model-to-model transformation with ATL on MapReduce. In Richard R. Paige, Davide Di Ruscio, and Markus Völter, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*, SLE 2015, pages 37–48. ACM, 2015.

[91] Amine Benelallam, Massimo Tisi, Jesús Sánchez Cuadrado, Juan de Lara, and Jordi Cabot. Efficient model partitioning for distributed model transformations. In Tijs van der Storm, Emilie Balland, and Dániel Varró, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, SLE 2016, pages 226–238. ACM, 2016.

[92] Stefan Jurack and Gabriele Taentzer. Towards composite model transformations using distributed graph transformation concepts. In *International Conference on Model Driven Engineering Languages and Systems*, pages 226–240. Springer, 2009.

[93] Julien Mallet and Siegfried Rouvrais. Style-based model transformation for early extrafunctional analysis of distributed systems. In *International Conference on the Quality of Software Architectures*, pages 55–70. Springer, 2008.

[94] Loli Burgueno, Manuel Wimmer, and Antonio Vallecillo. A linda-based platform for the parallel execution of out-place model transformations. *Information and Software Technology*, 79:17–35, 2016.

[95] Dolores Burgueño Caballero et al. On the quality properties of model transformations: Performance and correctness. 2016.

[96] Fazle Rabbi, Yngve Lamo, Ingrid Chieh Yu, and Lars Michael Kristensen. Webdpf: A web-based metamodelling and model transformation environment. In *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 87–98. IEEE, 2016.

[97] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[98] Kaushal Chari. Model composition in a distributed environment. *Decision Support Systems*, 35(3):399–413, 2003.