# Executing Certified Model Transformations on Apache Spark

Jolan Philippe
IMT Atlantique, LS2N (UMR CNRS 6004)
Nantes, France
jolan.philippe@imt-atlantique.fr

Massimo Tisi
IMT Atlantique, LS2N (UMR CNRS 6004)
Nantes, France
massimo.tisi@imt-atlantique.fr

Hélène Coullon
IMT Atlantique, Inria, LS2N (UMR CNRS 6004)
Nantes, France
helene.coullon@imt-atlantique.fr

Gerson Sunyé
University of Nantes, LS2N (UMR CNRS 6004)
Nantes, France
gerson.sunye@ls2n.fr

## Abstract

Formal reasoning on model transformation languages allows users to certify model transformations against contracts. CoqTL includes a specification of a transformation engine in the Coq interactive theorem prover. An executable engine can be automatically extracted from this specification. Transformation contracts are proved by the user against the CoqTL specification and guaranteed to hold on the transformation running on the extracted implementation of CoqTL. The design of the transformation engine specification in CoqTL aims at easing the certification step, but this requirement harms the execution performance of the extracted engine.

In this paper, we aim at providing a scalable distributed implementation of the CoqTL specification. To achieve this objective we proceed in two steps. First, we introduce a refined specification of CoqTL that increases the engine parallelization. We present a mechanized proof of the equivalence with standard CoqTL. Second, we develop a prototype implementation of the refined specification on top of Spark. Finally, by evaluating the performance of a simple case study, we assess the speedup our solution can reach.

***CCS Concepts:*** • **Computing methodologies** → **MapReduce algorithms**; • **Software and its engineering** → **Correctness**; **Domain specific languages**.

***Keywords:*** Model-Driven Engineering, Model transformation, Spark, Coq


*SLE '21, October 17–18, 2021, Chicago, IL, USA*

ACM ISBN 978-1-4503-9111-5/21/10…$15.00
https://doi.org/10.1145/3486608.3486901

**ACM Reference Format:**
Jolan Philippe, Massimo Tisi, Hélène Coullon, and Gerson Sunyé. 2021. Executing Certified Model Transformations on Apache Spark. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3486608.3486901

## 1 Introduction

In model-driven engineering (MDE), model management frameworks propose dedicated languages to transform models, like the AtlanMod Transformation Language [10] (ATL) or the Epsilon Transformation Language [12] (ETL). Good scalability and facilities for formal reasoning are among the intended benefits for users of model-transformation languages. On the one hand, researchers have proposed transformation engines designed to effectively perform computationally or memory-intensive transformations [11]. For instance, transformation languages have been equipped with implicit parallel/distributed modes of execution to automatically multiply the number of resources allocated to a transformation [3, 12]. To achieve this, some engines have been built on top of distributed programming frameworks (e. g. Apache Hadoop in [3]). On the other hand, the community has extensively worked at formal reasoning and verification tools for model transformation languages. Among these solutions, the CoqTL language [21] allows users to write transformation rules, define contracts and certify the transformation against them, within the Coq proof assistant [7].

In this paper, we introduce a transformation engine that addresses at the same time distribution and certification. This is not trivial. Distributed programming typically involves non-deterministic execution order, complicating the proof of properties on parallel programs. Interactive theorem proving is already a very costly activity on sequential model transformations. Certifying transformations by formal reasoning on the complexities of modern distributed frameworks would require unmanageable proofs. We propose a solution to allow users to certify transformations in Coq and execute them over a modern data-analytics framework. The core of our

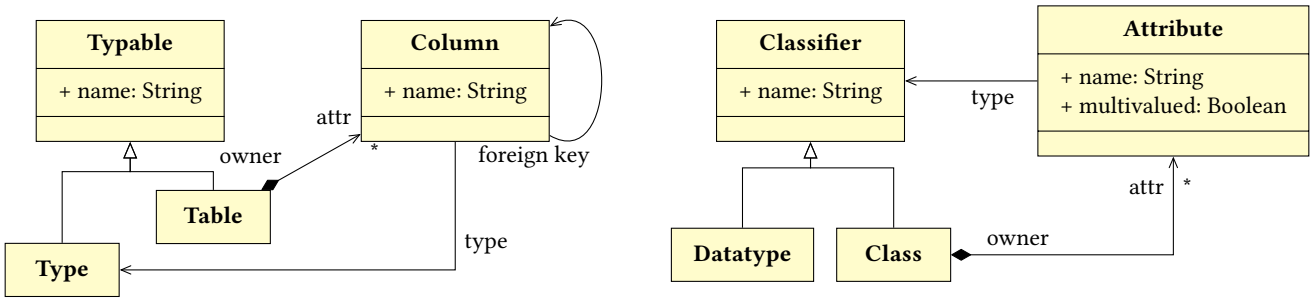



**Figure 1.** Relational and Class diagram metamodels

proposal is SparkTE, a distributed implementation of the CoqTL specification, built on top of Apache Spark, a widely-deployed framework. Besides scalability, Spark provides an ecosystem of libraries, e. g. for connection to heterogenous data sources. Importantly, the Spark programming interface mirrors a functional/higher-order model of programming. This is a key property for our solution, since it enables a straightforward and high-confidence extraction of parallel code from functional Coq specifications.

Notice that we designed the parallelizable specification for a specific data analytic framework, and targeting another back-end solution would necessitate additional manual changes.

This paper has two main contributions:

- A refinement of the standard CoqTL specification, including three major optimizations to increase parallelism opportunities. The refined specification, here named Parallelizable CoqTL, is written in Coq, and is validated by formally proving in Coq the input/output equivalence to the standard CoqTL specification.
- A transformation engine that implements Parallelizable CoqTL on top of Apache Spark, named SparkTE. By evaluating the performance of a simple case study, we assess the speedup that SparkTE can reach.

The rest of the paper is organized as follows. We first introduce a running example in Section 2. Our approach overview is illustrated in Section 3. Section 4 presents the new Parallelizable CoqTL specification and the SparkTE engine. The performance experiments on SparkTE are presented in Section 5. We finally conclude the paper by summarizing related work in Section 6 and drawing conclusions in Section 7.

## 2 Motivation and Background

### 2.1 Running Example

We choose a simple transformation as a running case to illustrate the approach and perform preliminary performance assessments. Listing 1 shows an excerpt of the code of a CoqTL transformation named `Relational2Class`, which ideally reverse-engineers class diagrams from given relational schemas, and Figure 1 shows its source and the target metamodels. The transformation is a simplified inverse of the well-known Class2Relational transformation [22], often used in the community for exemplifying new contributions. We choose the inverse direction because it is representative of reverse-engineering transformations, where scalability problems frequently arise [2].

The transformation is written in CoqTL, an internal DSL for model transformation within the Coq theorem prover. The transformation primitives are newly-defined keywords (by the notation definition mechanism of Coq), while all expressions are written in Gallina, the functional language used in Coq. The CoqTL semantics is heavily influenced by ATL [10] (notably in the distinction between a match/instantiate and an apply function), and its original design choices focus on simplifying proof development.

In Listing 1, the `Relational2Class` transformation is defined via four rules, composed of two parts: (i) a matching section (type and guard condition) and (ii) an output section, which contains a definition for created target elements and optional references. To keep a trace of which expression is used for mapping a source to a target element, each element of the output section of the rules is named.

The first rule (lines 5–11) maps all relational `Types` to class-diagram `Datatypes`. We specify that a datatype is constructed using `BuildDataType` with the same `id` and the same `name` as the matched type (line 10). The `Table2Class` rule (lines 5–11) translates all tables to a corresponding class, with the exception of tables that persist multivalued attributes. To filter these tables, a guard condition, introduced by the **`when`** keyword, calls a user-defined `isClassTable` function (line 14), that we will discuss later. The created target class is constructed from the `id` and the `name` of a matched table (line 18). The third rule (lines 21–40) generates `Attributes` and links them to `Classifiers`. `Columns` are transformed into single-valued `Attributes` by a call to `BuildAttribute` with the last parameter (`multivalued`) equal to `false` (line 27). Again, columns that are contained by tables that persist multi-valued attributes are not transformed, thanks to a guard `isClassTable` (line 14). Additionally, two links are defined for the generated `Attribute`. First,

```
1  Definition Relation2Class :=
2  transformation from RelationalMetamodel to
3    ClassMetamodel with m as RelationalModel :=
4
5    rule Type2Datatype
6      from element t class Type
7      to [
8        output "type"
9          element dt class Datatype :=
10         BuildDataType (getTypeId t) (getTypeName t)
11     ];
12
13   rule Table2Class
14     from element t class Table when isClassTable t
15     to [
16       output "class"
17         element c class Class :=
18           BuildClass (getTableId t) (getTableName t)
19     ];
20
21   rule Column2Attribute
22     from element c class Column when
23       isClassTable (getOwner c m)
24     to [
25       output "svattr"
26         element a class Attribute := BuildAttribute
27             (getColumnId c) (getColumnName c) false ;
28         links [
29           reference AttributeType :=
30             ty <- getColumnType c m;
31             dt <- resolve Relational2Class m "type"
32                    DataType [[ ty ]];
33             return BuildAttributeType a dt;
34           reference AttributeClass :=
35             ta <- getColumnTable c m;
36             cl <- resolve Relational2Class m "class"
37                    Class [[ ta ]];
38             return BuildAttributeClass a cl
39         ]
40     ];
41
42   rule MVAttributeTable2Attribute
43     from
44       element t class Table; element o class Table
45       when isMVAttributeTable t o
46     to [
47       output "mvattr"
48         element a class Attribute := ...
49     ]
```

**Listing 1.** Excerpt of the Relational2Class Transformation in CoqTL

a link from the target `Attribute` to its `type` (lines 29–33). The corresponding `type` is defined in `BuildAttributeType` by resolving the output of the `Type` of the source `Column` (line 32). The second reference, from the `Attribute` to its `owner` (lines 34–38), is defined in the same manner (i. e., by resolving the output of the owner `Table` of the source `Column`) (line 37). `MVAttributeTable2Attribute` (line 42–49), that is the last rule of our transformation, matches two tables: a table `t` generated from a multi-valued attribute and the table `o` that corresponds to its owner. The two references are built as in the `Column2Attribute` rule: one reference to the type of the attribute and one to its owner.

```
Lemma tr_r2c_inverse: ∀ (m : ClassModel),
execute Relational2Class (execute Class2Relational m)
  = m.
```

**Listing 2.** The tr_tracePattern_source lemma

Once the transformation has been defined in CoqTL, it is possible to prove that it has a given property, e. g. it respects a given contract. For instance, CoqTL users can interactively prove a properties of `Relational2Class` and `Class2Relational`, by proving the lemma in Listing 2 using Coq tactics. Several strategies are possible to distinguish tables that were generated from classes, from tables that correspond to multivalued attributes. These strategies would correspond to different implementations of the functions `isClassTable` and `isMVAttributeTable`. In our simple example, we assume that *Class2Relational* translates multivalued attributes into tables with a specific name pattern (e.g., name of the owner class followed by an underscore and the name of the multivalued attribute). Under this assumption `isClassTable` and `isMVAttributeTable` just contain a string check, with constant complexity. In the experimentation (Section 5) we will introduce more computational time in these functions to simulate increasing complexity, to better estimate the potential parallelization of our solution.

### 2.2 Objective

CoqTL includes an executable semantics for the transformation engine. An implementation of the engine in OCaml or Haskell can be automatically obtained by the standard extraction mechanism of Coq. However, since the executable semantics is designed to be reasoned about in proof terms, it is kept very simple (the core semantics is formalized in 196 Coq LOC) and does not include any efficiency optimization. This has a significant impact on the performance of the extracted engine for CoqTL. For instance , the execution of the Relational2Class transformation on a sequential version of the CoqTL engine for a model of a thousand of model elements takes more than 4 hours on a recent laptop (Intel Core i7-8650U CPU @ 1.90 GHz, 32 GB of RAM).

Considering the size of realistic code-bases for reverse-engineering projects, we aim for a solution that allows users to deploy and run the transformation on a state-of-the-art distributed data-analytics framework. Such frameworks are often already deployed in companies that perform large computations, or in their cloud-based services portfolio.

Apache Spark is a data-analytics framework aiming at querying or manipulating large-scale data. In a local mode, the computation is run using a single machine, while in a cluster mode a master dispatches some partitions and tasks

to several other additional machines: the workers. Such architecture allows to use more resources (memory, number of processors) which lead to the opportunity of running computations on larger datasets. The data-parallel approach promoted by Spark consists in using a specific data structure called RDDs (Resilient Distributed Dataset) that is partitioned by the master. Partitions are distributed among the workers as well as tasks to execute on RDDs. A task in Spark is a set of operations performed on a RDD. Spark supports several programming paradigms. The use of these paradigm for model-management operations has been investigated in [19]. In this paper, we use Spark with the MapReduce paradigm [14].

We aim at designing a solution that executes on Spark a CoqTL transformation (like `Relational2Class`), certified against a contract. On the performance side, we aim at improving two kinds of scalability: 1) The capacity of scaling with additional computational resources, referred to as vertical scalability; 2) The capacity of dealing with increasing datasets, referred to as horizontal scalability. On the reliability side, while the whole execution stack can not be certified end-to-end (e. g. the formal semantics of Apache Spark is not available), we aim nonetheless to produce a solution that gives users high confidence that the proved contract will hold on the distributed transformation.

## 3 Approach Overview

Figure 2 shows the global workflow of our approach, separating the formal part, written in Coq (left side), from the executable engine in Scala and Spark (right side). Starting from the Coq part, the workflow must be interpreted as follows.

The CoqTL language allows users to define model transformations, theorems on their behavior and machine-checked proofs of these theorems in Coq. These transformations can be executed directly on the executable CoqTL specification. We introduce *Parallelizable CoqTL*, a refinement of the executable CoqTL specification that increases its parallelization opportunities. We use the Coq theorem prover to prove that this new specification is a refinement of the original one (see Section 4.2). This entails that, for a given source model, any transformation produces the same output model when it runs on standard CoqTL or Parallelizable CoqTL. The Parallelizable CoqTL specification is written in Gallina, the functional language used in Coq.

Then we provide an implementation of Parallelizable CoqTL in Scala. To obtain the Scala implementation, that we name ScalaTE, we manually extract the Gallina functions into corresponding Scala functions. Section 3.1 details this extraction.

ScalaTE uses data structures that are the direct correspondent of Gallina data structures (e. g. Scala `List` for Gallina `list`). In a final step we replace these data structures with distributed data structures (RDDs) from the Spark library, as described in Section 3.2. We name the resulting distributed engine SparkTE.

To run a transformation on SparkTE, the CoqTL transformation rules (e. g., Listing 1) have to be translated into their corresponding Scala version. We currently perform this step manually, but its automatization is possible and left for future work. Since Spark does not change the functional interface of standard Scala, the obtained Scala transformation can run on both ScalaTE and SparkTE.

### 3.1 Coq to Scala

The Coq environment includes an extraction mechanism targeting ML languages: OCaml, Haskell, or Scheme. Although an automatic extractor to Scala is available [8], it supports only a subset of Gallina on an outdated version of Coq. Hence, we opted to perform the extraction manually. We perform manual extraction at two levels: first to create the core engine (*gallina2scala* in Fig. 2), then to obtain Scala rules representing a CoqTL transformation (*coqtl2scala*).

***Gallina to Scala.*** The executable CoqTL specification can be seen as a functional program that interprets the transformation code. We produce a literal translation of this interpreter in Scala.

For extracting Scala code from Parallelizable CoqTL we translate Gallina types and (pure) functions into their correspondent types and pure functions in Scala. Listings 3 and 4 show an example of CoqTL executable specification (in Gallina) and its implementation in SparkTE (in Scala). The example well illustrates the one-to-one translation among types and functions that we adopt for all the extraction.

We implicitly assume that Scala functions (e. g., `flatMap` for Scala `List`) implement the same semantics than their Gallina correspondent (e. g., `flat_map` for Gallina `list`).

***CoqTL to Scala.*** The CoqTL parser translates the concrete syntax of the transformation (e. g., from Listing 1) into Coq code to construct an abstract syntax tree. Obtaining the same transformation in Scala requires constructing the same abstract syntax tree as Scala objects. Note that Scala constructors for the abstract syntax are the literal translation of the corresponding Gallina constructors in CoqTL.

Listing 5 shows the translation of the *Table2Class* rule from Listing 1. The rule constructor (`RuleImpl`) requires a name, a list of types for an input pattern (the `types` argument), a guard condition (the `from` function), a list of output pattern elements (the `to` argument). Each output pattern element has a name, and a function for creating output elements from input pattern elements. While not shown in this example, it can be also accompanied by a list of functions for creating links in the output model.

As illustrated by the example, the only non-trivial part of this extraction is the translation of the body of expressions for guards and output element creation (e. g. the body of the

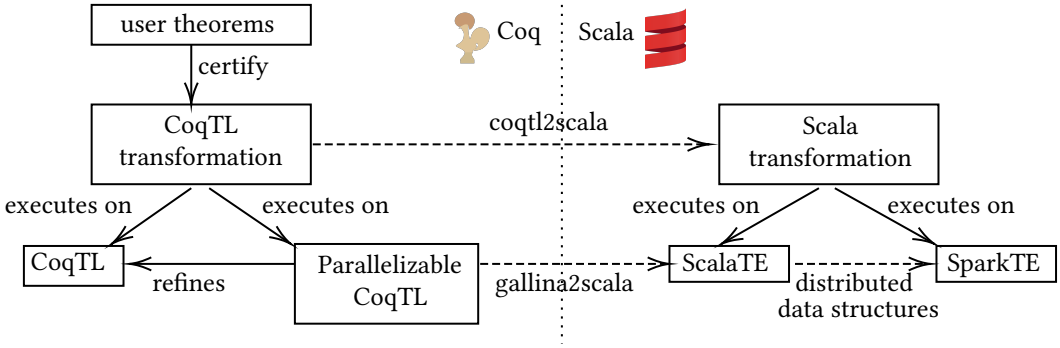



**Figure 2.** Global overview of our workflow to execute certified model transformations on Apache Spark.

```
Fixpoint tuples_of_length_n {A : Type} (s1: list A)
(n : nat): list (list A) :=
match n with
| 0 ⇒ nil::nil
| S n ⇒ prod_cons s1 (tuples_of_length_n s1 n)
end.
```

**Listing 3.** CoqTL definition for generating all combination of elements

```
def tuples_of_length_n[A] (s1: List[A], n: Int)
: List[List[A]] =
n match {
    case 0 =>
        List(List())
    case n1 =>
        prod_cons(s1, tuples_of_length_n(s1, n1-1))
}
```

**Listing 4.** Extracted Scala code from the Coq function `tuples_of_length_n`

```
new RuleImpl (
  name = "Table2Class",
  types = List(RelationalMetamodel.TABLE),
  from = (m, l) => {
    val t = l.head
    Some(t.isClassTable)
  },
  to = List (
    new OutputPatternElementImpl(
      name = "class",
      elementExpr = (i, m, l) =>
        if (l.isEmpty) None else {
         val t = l.head.asInstanceOf[RelationalTable]
         Some(new ClassClass(t.getId, t.getName,
           multivalued=false))}
  )))
```

**Listing 5.** Scala implementation for the Table2Class rule

anonymous functions in Listing 5). An automatic compiler from CoqTL rules to Scala is a part of future work (Section 7).

### 3.2 Distributed Data Structures

Spark RDDs are data structures that are automatically partitionned and resulting in the distribution of the computation operations on a Scala sequence (e. g., `List`) of serializable elements. From a user point of view, RDDs can be manipulated as lists, using the same primitive functions, and parallelism is implicit. The advantage of using such abstraction for parallelism is the semantics preservation of the operations on the distributed structures. Because of the popularity of Spark and its support, we assume the correctness of parallel operations on the data.

An efficient use of RDDs requires an effective partitioning of data. For instance, to take advantage of the internal multithreading mechanism, it is typically recommended in Spark to assign four data partitions to each core. Each independent computation of a partition is referred as a task. The Spark task scheduler makes the distribution following a round-robin approach, optimizing load-balancing: once a task is ended on a core, a new one can be assigned from the waiting list. Section 4 gives more detail about how we use RDDs in SparkTE.

## 4 Parallelizable Semantics for CoqTL

The `execute` function shown in Listing 6 is the entry point of the transformation execution in the standard CoqTL semantics.

First, the `allTuples` function (line 3) produces all the tuples of elements that can be possibly matched by the rules. `allTuples` computes a list of $\sum_{a=0}^{A} n^a$ tuples, with $n$ the number of elements in the input model, and $A$ the maximum arity of the transformation rules.

Then, the `instantiatePattern` function (line 5) tests each tuple to find if it matches with any rule and for each match it constructs the corresponding output pattern elements. Internally it iterates on each rule, executes their guard function and if the result is positive, executes the element creation function for each output pattern element of that

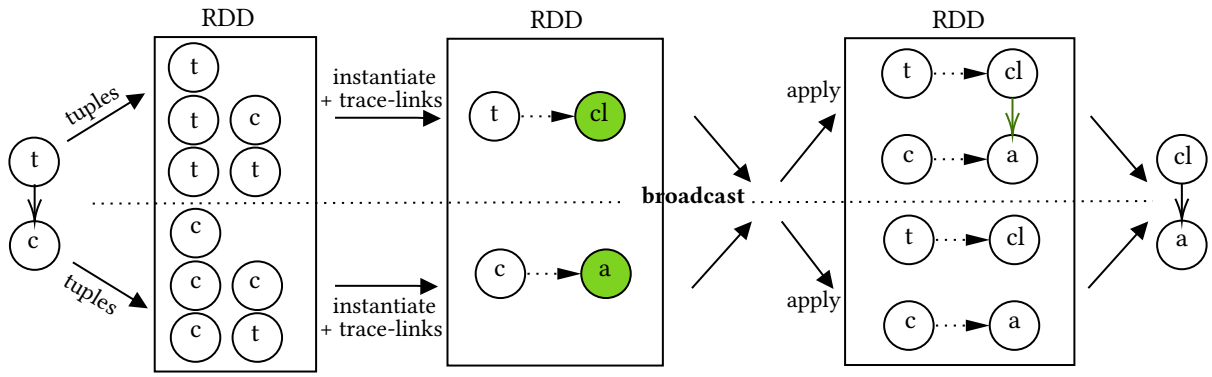



**Figure 3.** Distributed computation of a model transformation on SparkTL. *c* denotes a column, *t* a table, *a* an attribute, and *cl* a class. Green elements and links illustrate the entities created by the transformation in the output.

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

**Listing 6.** `execute` function in the base CoqTL specification

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

**Listing 7.** `execute` function in the Parallelizable CoqTL specification

rule. The resulting elements are gathered by the `flat_map` in a single output list.

Finally, the `applyPattern` (line 6) function is executed on each tuple to create target links. Similarly to the function `instantiatePattern`, the function internally iterates on all rules and checks if the rule matches the given pattern. In a positive case, the element creation functions for that rule are executed and then the link creation functions. The resulting links are gathered by the `flat_map` in a single output list.

### 4.1 Parallelizable CoqTL

Parallelizable CoqTL contains three optimizations to the base CoqTL specification:

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

Note that similar optimizations (among others) are already implemented in well-known transformation engines, like ATL [10] or ETL [12]. Differently from previous work, in this paper we formalize the optimizations, interactively prove that they do not affect the transformation output and assess their impact on distributed execution.

The entry point of Parallelizable CoqTL, is presented in Listing 7. Figure 3 illustrates the global behavior on a minimal example with one table and one column.

***Optimization 1: Two phases.*** In standard CoqTL the `applyPattern` function performs all the computation of the links generated by a matched input pattern, by the rule that matches it. However the computation of the links of a rule is not independent from the computation of other rules. This dependency is caused by the `resolve` function (e. g., lines 32 and 37 in Listing 1) that searches for the output of another rule in order to set the target of the created link. In general, because of this dependency, two executions of the apply function can not be run in parallel, without replicating some matching and instantiation within each call to `resolve`.[1]

We refactor the computation to split it in two phases, similarly to ATL [10]. This is visible in Listing 7. In the first phase (lines 5) we compute the tuples and we run the matching and instantiation by a new function named `tracePattern`. The first phase produces the list of generated elements, and trace links connecting them to their corresponding source patterns. Differently from Listing 6, here the second phase

[1]This is exactly how the standard CoqTL specification computes resolve. This simple solution keeps the specification compact and has no negative impact on proofs. However it has a big impact on performance and parallelization.

(line 8)) can only start computing output links after the full first phase has finished computing the trace links, since the `flat_map` expects the `tls` structure as parameter.

In Listing 7 every execution of `tracePattern` can be run in parallel. When the first phase is over, every execution of `applyPatternTraces` can be run in parallel too, since the calls to resolve can be computed immediately on the trace-link structure. This greatly improves the parallelization of the algorithm.

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

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

To limit imbalance, in the initial sequential tuple generation phase, we generate only tuples whose type matches with at least one rule of the transformation. This is shown in Listing 7 by the use of the `allTuplesByRule` for tuple generation (line 2). `allTuplesByRule` iterates on rules and produces only combinations of elements of the types listed in the rule input pattern. This improves load balancing of the first phase since all the produced tuples require a guard evaluation.

***Optimization 3: Apply iterates on traces.*** Executing the apply phase on the tuples generated by `allTuplesByRule` would cause an imbalance among partitions, similar to the one discussed in Optimization 2. Indeed, among these tuples only very few have passed the guard condition in the first step. A partitioning of `allTuplesByRule` would produce partitions that do not require any computation, together with partitions that need to evaluate several expensive link creation functions. In this optimization we make the apply phase iterate only over the source patterns that passed the guard evaluation in the instantiate phase. We retrieve these patterns by collecting them from the list of trace links. This is performed by the function `allSourcePatterns` at line 8 of Listing 7.

**Table 1.** Size (in LOC) of new specification and certification proofs added for each optimization, with proof effort (in man-days).

| Optimization | Spec. size | Cert. size | Proof effort |
|---|---|---|---|
| twoPhases | 69 | 484 | 10 |
| byRule | 42 | 487 | 7 |
| iterateTraces | 69 | 520 | 4 |

### 4.2 Refinement Proof

Besides the executable functional specification, CoqTL is also described by an axiomatic specification. Certifying against the axiomatic specification involves providing 10 types, 27 semantics functions and proving 15 theorems. The specification is fully illustrated in [6], together with a proof that engines implementing the executable specification certify against the axiomatic one.

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

```
Lemma exe_preserv :
∀ (tr: Transformation) (sm : SourceModel),
  twophases. execute tr sm = execute tr sm.

Lemma In_by_rule :
∀ (sp: list SourceModelElement) (tr: Transformation)
  (sm: SourceModel),
  In sp (allTuplesByRule tr sm)
    → In sp (allTuples tr sm).

Lemma In_by_rule_instantiate :
∀ (sp: list SourceModelElement) (tr: Transformation)
  (sm: SourceModel),
    In sp (allTuples tr sm)
      ∧ instantiatePattern tr sm sp <> nil
        → In sp (allTuplesByRule tr sm).
Lemma In_by_rule_apply : ...

Lemma tr_tracePattern_source:
∀ (tr: Transformation) (sm : SourceModel)
  (tl : TraceLink) (sp: list SourceModelElement),
  In tl (tracePattern tr sm sp)
    → sp = TraceLink_getSourcePattern tl.
```

**Listing 8.** Certifying optimizations lemmas

available online[2]. All the discussed lemmas below can be found in Listing 8.

The key step for certifying the *twoPhases* optimization, is proving that it does not change the output of the full transformation (`exe_preserv`). This proof has two parts: 1) for the instantiation phase, we prove that the additional computation of the trace, does not have any effect on the computation of the instantiated elements; 2) for the apply phase we need to prove that the new apply function is equal to the old one. Coq is able to prove automatically the second step, by full unfolding and simplification of the old and new apply functions. Note that in this proof we use the axiom of functional extensionality (two functions are equal if their values are equal at every argument), notably to compare the body of inner anonymous functions.

The *byRule* optimization changes the order of the element and link creation in the transformation output, hence a lemma similar to `exe_preserv` would not hold. We prove its equivalence in two steps: 1) we prove that tuples computed by rule are a subset of the all tuples (`In_by_rule`), 2) we prove that tuples computed by rule include all tuples that produce elements or links (`In_by_rule_instantiate`). The second step is the most challenging and is performed by case analysis on matchPattern: given any pattern, if the pattern does not match then it cannot produce anything, if it matches then it is one of the patterns generated by the rule `allTuplesByRule`.

The *iterateTraces* optimization does not change any function in the instantiate phase, hence we can easily prove a similar theorem to (`exe_preserv`), but limited to the result of instantiate. The apply phase does not change the computation of a single link, but skips some source patterns, and produces links in a different order w.r.t. the base version. To prove that the same links are generated, we proceed by case analysis on the trace generated for a given source pattern: 1) if the instantiate phase did not generate any trace for that pattern, then the standard apply phase would not have produced any corresponding link; 2) if there is a trace, then the apply phase applies the same function to the same source pattern, hence producing the same output (`tr_tracePattern_source`).

### 4.3 Implementation on Spark

Distributed computation of our Spark implementation revolves around the use of RDDs. First, the input patterns, represented as a list of tuples, are distributed with an RDD among cores. Each core independently applies the instantiate function to every tuple of its partition. Implicit communications are operated by Spark to scatter the tuples from the master process to the workers. After the computation, the resulting elements and their trace-links are all gathered to the master process.

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

**Table 2.** Hardware setup of the two clusters used during experiments. These clusters are part of the Grid'5000 experimental platform for distributed computing.

| Cluster | CPU | RAM | Netw. |
|---|---|---|---|
| Rennes<br>*paravance* | 2× 8 cores/CPU<br>*Intel Xeon E5-2620* | 128GB | 2×<br>10Gbps |
| Nancy<br>*gros* | 2× 18 cores/CPU<br>*Intel Xeon Gold 5220* | 96 GB | 2×<br>25Gbps |

**Table 3.** Description of the three datasets used in the experiments with the number of elements and links.

| Dataset | D1 | D2 | D3 | D4 | D5 |
|---|---|---|---|---|---|
| model type | Relational | | | IMDb | DBLP |
| elements | 150 | 300 | 600 | 440 | 700 |
| links | 290 | 580 | 1060 | 1968 | 1886 |

For the second phase, the trace-links are distributed with an RDD. Each core is in charge of generating the links for a partition of output elements and their associated applied rule. Since this second phase needs a global knowledge of the trace-links to resolve output elements, we used a broadcast communication to share the whole set of trace-links to all cores.

A global view of where RDDs are used and what communications are operated is illustrated in Figure 3.

### 4.4 Limitations

SparkTE implements the complete CoqTL specification but some limitations remain. First, all the translations from Coq to Scala are manual. In particular, the transformation rules need to be manually translated by the user. However the correspondence between the abstract syntax in CoqTL and ScalaTE/SparkTE makes the translation of the rule structure trivial. Hence, our solution reduces the complex problem of translating to Scala a CoqTL transformation, into a simpler problem: translating to Scala separately each side-effect-free expression for guards, output pattern elements creation, and output pattern links creation. While simpler, this task is still tedious and error prone, hence we aim at making the translation fully automatic in future work.

## 5 Experiments

In the previous sections, we have presented the three optimizations adopted for Parallelizable CoqTL to increase the efficiency and propensity to parallelization of the extracted engine. In this section, we evaluate the performance of SparkTE and the quality of its speedup compared to the ideal speedup that divides the execution time by the number

of cores (relative to a reference version). As none of the contributions of the related work have studied the use of Spark for parallel transformations, we restrict our comparisons to this theoretical ideal speedup. All jobs are run with Spark in a standalone mode.

The first subsection presents the results we get from the use-case previously described in Section 2.1, i. e. the *Relational2Class* transformation , and similar experiments on two additional cases.

We show that the three optimizations introduced in Section 4 improve significantly the performance of the transformation in Spark. The results also show a relatively poor speedup compared to the ideal one, but that can be improved when having more complex operations within the transformation. To investigate this result, the second subsection illustrates that a speedup close to the ideal can be observed with a high number of cores if the computation time and the size of the dataset is big enough to offset the overheads of Spark. Finally, a performance analysis by phase is presented to show the very good speedup obtained by the parallelized steps of SparkTE.

All results presented in this section have been executed ten times on the same hardware configuration, and an average is presented. The standard deviation of our measurements are never higher than 10 % of the total execution time, thus guaranteeing our results to be stable, and the average to be meaningful. Furthermore, all our experiments have been conducted on the French experimental platform for distributed computing Grid'5000. Grid'5000 is made of geographically distributed clusters of machines in France, each one with its specific hardware setup. This platform also facilitates the reproducibility of the experiments by offering a way to build the same environment for any researcher. However, Grid'5000 makes us dependent on available machines (i. e. nodes) for provisioning which is why we use two different clusters of Grid'5000 in our experiments. For each experiment the number of machines and the number of cores are specified. One can note that the number of machines that is specified correspond to the number of workers instantiated in Spark and that one additional dedicated machine is provisioned to host the master of Spark. All our codes, raw results, and analysis scripts are publicly available online.[3]

### 5.1 Evaluation of SparkTE on Use-cases

In this section we apply our running example to a first model, denoted D1 in Table 3. We also consider two additional transformations: the IMDb case [9], aiming at finding couples of actors who recurrently played together, and queries on a DBLP model to find active authors who published in specific journals [1]. We run the two additional transformations on model instances from the IMDb and DBLP metamodel, respectively denoted D4 and D5 in Table 3.

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

The transformations specified in CoqTL, subsequently translated to Scala and Spark, are always correctly computed by SparkTE, i. e. with the expected output of the transformation, for all our experiments.

We have used from 1 to 2 machines of the cluster *paravance* detailed in Table 2. Table 4 presents the execution times and speedups of the three transformations from 1 to 8 cores with 1 and 2 machines. The results show a relatively poor speedup. It is caused by the lack of complexity in the operations used in the guard condition, the instantiate function and the apply function.

Spark's overheads are more particularly observable with the poor speedup measured with 2 cores. Indeed, in this case the gain of parallelizing with Spark is completely swallowed by the overheads, but then an improvement is obtained with 4 cores. Overall the obtained speedup is not satisfying as the speedup for 8 cores is almost the same as with 4 cores, and far from the expected ideal speedup of 8. The additional cases show that the performance of the approach depends on the computation time of the rules. In the IMDb case, the better growing speedup is caused both by the rules themselves that have a high-level of complexity, and the high connectivity of the input model. On the contrary, in the case of the DBLP transformation, the rules with a low computational cost lead to a poor speed-up. We will show in the next subsection that Spark's overhead is fully compensated by high computation time of the transformation and/or large size of the dataset.

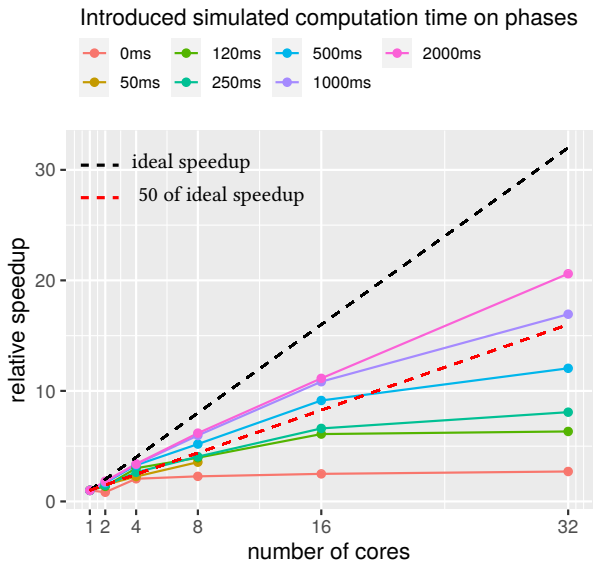
One problem with computation on large models is the lack of memory. Distributed solutions, such as the one we propose here, allow users to increase the resources allocated to a transformation. For instance, a fixed allocated memory for one core is not enough for processing the DBLP example, causing disk operations which drastically slow the computation. This phenomenon disappears with 2 cores, and we observe a hyper speedup.

Finally, we want to show the performance gain obtained thanks to our three optimizations. To this purpose, we execute the same *naive* transformation on the direct implementation of the CoqTL specification (without optimizations). On 1 core *naive* is computed in 27 s by SparkTE and in 52 s by the CoqTL implementation.

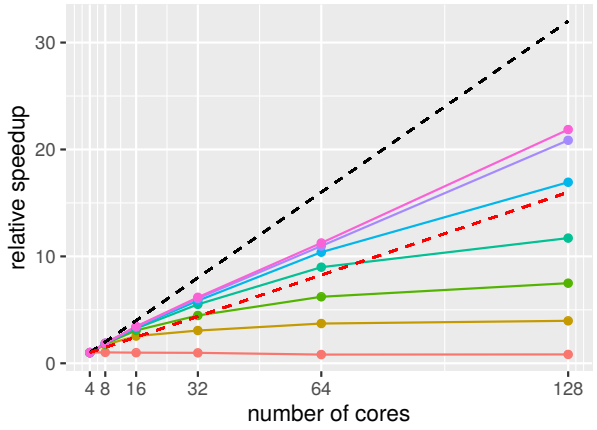
### 5.2 Performance Analysis by Complexity and Datasets

In the previous subsection, we have shown that the performance of SparkTE is enhanced by our three optimizations in terms of execution time and speedup. However, compared to the ideal speedup our use-cases are disappointing. In this subsection, we demonstrate that these results are due to the small computation time in the transformation and the small size of the dataset.

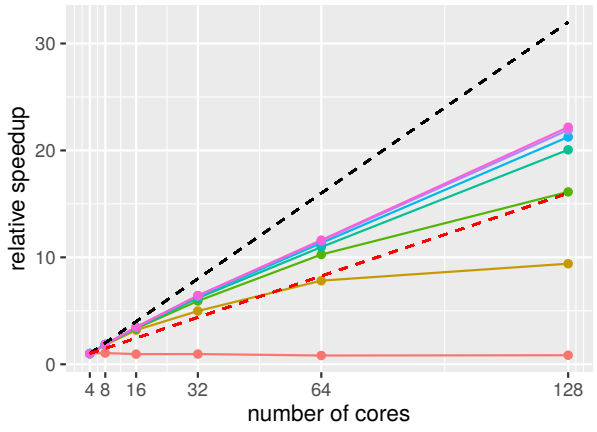
**Experimental setup.** To this purpose, we have built a version of the transformation with additional fictitious processing time in the different phases and steps. More precisely,




**(a)** Relative speedups of SparkTE for D1 up to 32 cores on 4 machines




**(b)** Relative speedups of SparkTE for D2 up to 128 cores on 8 machines




**(c)** Relative speedups of SparkTE for D3 up to 128 cores on 8 machines

**Figure 4.** Relative speedups of SparkTE with sleeping times

**Table 4.** Execution times and speedups (relative to 1 core) for the executions with SparkTE of *Relational2Class* (R2C), the IMDb findcouple transformation, and the DBLP case. Experiments respectively conducted with the dataset D1, D4, and D5, on the cluster *paravance*.

| | ♯ cores (♯ machines) | 1 (1) | 2 (1) | 4 (2) | 8 (2) |
|---|---|---|---|---|---|
| R2C | time (s) | 27.02 | 32.50 | 13.17 | 11.91 |
| | speedup | 1.00 | 0.84 | 2.13 | 2.31 |
| IMDb | time (s) | 38.35 | 22.01 | 18.33 | 11.61 |
| | speedup | 1.00 | 1.74 | 2.09 | 3.30 |
| DBLP | time (s) | 0.83 | 0.35 | 0.56 | 0.84 |
| | speedup | 1.00 | 2.37 | 1.49 | 0.99 |

we incorporate sleeping functions to different parts of the rule evaluation: the guard condition to simulate complex checking functions; the instantiate part to simulate complex instantiation of element; and the apply function to simulate a long resolution for links. As we discussed in Section 2, user-defined functions can have several implementations with different complexities. Proposing an evaluation based on the computation time of these functions offers an infinite set of benchmarks to accurately estimate the speedup of SparkTE. Furthermore, we additionally vary the size of the dataset in the obtained benchmarks.

Table 5 summarizes the set of benchmarks explained hereafter. The first benchmark (B1) uses the same data introduced in the previous subsection on (D1) with 1, 2, and 4 machines, and up to 8 cores per machine on the *paravance* cluster of Grid'5000. We compare the computation time of the transformation on 2, 4, 8, 16, and 32 cores relative to the execution time on 1 core. The second and third benchmarks, (B2) and (B3), illustrate both the horizontal scalability and the vertical scalability of SparkTE by increasing the size of the dataset with (D2) and (D3) detailed in Table 3, and the number of machines (i. e. nodes, workers) and cores detailed in Table 5. The two benchmarks are evaluated on 1 to 4 machines, up to 16 cores per machine. As increasing the size of the dataset and the fictitious sleeping time also increases the total execution time of experiments, we compute our speedup relative to 4 cores instead of 1. As a result, the ideal speedup for 16 cores, for instance, is 4. Experiments on (B2) and (B3) are conducted on the *gros* cluster of Grid'5000. Furthermore, as indicated in Table 5, the three benchmarks are evaluated on the following sleeping times: 0, 50, 100, 250, 500, 1000, and 2000 ms. Finally, for the three benchmarks the overall execution time is measured, thus including some small sequential parts of the code, and the broadcast phase.

**Results.** Figures 4a, 4b, and 4c show the speedups observed for each benchmark according to the sleeping time and compared to the theoretical ideal speedup. When increasing the sleeping time, i. e. the execution time of the

**Table 5.** The set of experiments described in Section 5.2 with different fictitious processing time in the transformation, and different sizes of datasets. The number of cores used by each benchmark is also indicated, as well as the number of cores per node (i. e. worker). Finally, the Grid'5000 cluster used for the benchmark is given in the last column.

| | dataset | sleeping | ♯ cores | ♯ machines | ♯ cores per node | cluster |
|---|---|---|---|---|---|---|
| B1 | D1 | {0, 50, 120, 250, 500, 1000, 2000} | {1, 2, 4, 8, 16, 32} | {1, 2, 4} | 4 to 8 | *paravance* |
| B2 | D2 | same as B1 | {4, 8, 16, 32, 64, 128} | {1, 2, 4, 8} | 4 to 16 | *gros* |
| B3 | D3 | same as B1 | same as B2 | same as B2 | same as B2 | *gros* |

transformation, the speedup is enhanced and is closer to the ideal one. By increasing the size of the dataset, one can note a slight increase of the speedup by comparing B2 and B3 results respectively in Figure 4b and Figure 4c. Indeed, in Figure 4b, at 128 cores, more than half of the points are below the 50 % *optimal* value, while in Figure 4c only two points are below this theoretical value. In other words, at 128 cores, (D2) needs a sleeping time of 500 ms to reach 50 % from optimal speedup while (D3) only needs a sleeping time of 120 ms.

### 5.3 Performance Analysis by Phase

This third subsection aims at analyzing the impact of the two phases on the global speedup of the program. To do so, we processed the same benchmark as before (i. e., (B1), (B2) and (B3)), but with additional counters within the program to record the computation time of each distinct part: (1) the tuples generation; (2) the instantiate phase; (3) the broadcast intermediate step; and (4) the apply phase.

***Results.*** Our results on the three benchmarks show that the total computation time is mainly driven by the parallel parts. Indeed, in the case of the biggest dataset, with a sleeping time of 0 ms, the sequential part represents only 3.5 % of the total computation time which is the maximum percentage of the sequential parts. For instance, at the opposite extreme, i. e. in the case of the smallest dataset, with a sleeping time of 2000 ms, the sequential part represents less than 1 % of the total execution time. Hence, in the following, we restrict our analysis to the speedup of the parallel parts.

Table 6 illustrates the impact of each phase, by comparing their relative speedup to the optimal one. For reading convenience, we only show the results for sleeping times equal to 50 ms and 2000 ms. Table 6 confirms our previous assumption about the overhead of Spark that can be offset by increasing the computation time (i. e. sleeping duration) or the size of the dataset. Also, one can note that the *apply* phase offers better scalability than the *instantiate* phase. This result can be explained by the remaining imbalance on the tuples distribution. Let us remind first that the first *instantiate* phase is composed of two steps: (1) a guard condition; (2) the instantiation of the tuples that have satisfied this condition. Hence, even if the partitions are better balanced by our second optimization (see Section 4.1), not all the tuples are necessarily computed. As a matter of fact, the guard condition is always evaluated, but not the instantiation that depends on the result of the guard condition. On the contrary, in the *apply* phase apply patterns to all entries which are nearly perfectly balanced.

## 6 Related Work

Since models are basically typed graphs, several research efforts rely on graph theory, especially for transformations. For instance, in their Visual Modeling and Model Transformation (VMTS) tool, Mezei et al. use graph rewriting operations based on task-parallelism to apply distribute matching operations to large models [18]. Recently, the Pregel [17] paradigm, a strategy that aims at easing parallel computations on graphs on top of MapReduce architectures, has been used as a solution for model transformation. This is the case of Krause et al., who describe how transformation rules from the Henshin framework can be partially extracted to Pregel code [13]. A second use of Pregel in model transformation is proposed by Thung and Hu [23]. In their approach, graph transformations specified in a DSL are transformed into executable Pregel code.

Benelallam et al. introduce ATL-MR, a tool that uses the MapReduce paradigm to distribute the execution of ATL transformation rules [2]. Their data-parallel implementation follows a similar strategy as our SparkTE implementation: sub-parts of the model are independently transformed in a map phase and then dependencies are resolved in a reducing phase. In addition to distributing computation, authors have recently highlighted the good impact of their strategy for data partitioning. For instance, Burgueño et al. propose LinTra, a prototype for transformations on distributed architectures that tackles memory issues they face with shared-memory solutions [4]. They applied different strategies mixing both the a distribution of tasks and data, on a single or on multiple machines.

Both, ATL-MR and LinTra show better performances than SparkTE for executing a model transformation. However, they do not propose a verification counter part for certifying their distributed transformations.

**Table 6.** Relative speedups of SparkTE parallel phases on B1, B2 and B3 for sleeping times equals to 50 ms and 2000 ms. The percentage of the observed speedup compared to the theoretical ideal speedup is indicated for each result (higher the better).

| | Cores | Instantiate phase (50ms) | Apply phase (50ms) | Instantiate phase (2000ms) | Apply phase (2000ms) |
|---|---|---|---|---|---|
| B1 | 1 | 1 (100%) | 1 (100%) | 1 (100%) | 1 (100%) |
| | 2 | 1.48 (74%) | 1.62 (81%) | 1.81 (90.5%) | 1.87 (93.5%) |
| | 4 | 2.40 (60%) | 1.83 (45.75%) | 3.21 (80.25%) | 3.85 (96.25%) |
| | 8 | 3.40 (42.5%) | 4.50 (56.25%) | 5.78 (72.25%) | 7.15 (89.375%) |
| B2 | 4 | 1 (100%) | 1 (100%) | 1 (100%) | 1 (100%) |
| | 8 | 1.68 (84%) | 1.90 (95%) | 1.83 (91.5%) | 1.99 (99.5%) |
| | 16 | 2.39 (59.75%) | 3.18 (79.5%) | 3.30 (82.5%) | 3.84 (96%) |
| | 32 | 2.74 (34.25%) | 4.66 (59.25%) | 5.86 (73.25%) | 7.07 (88.375%) |
| | 64 | 3.17 (19.813%) | 7.34 (45.875%) | 10.87 (67.938%) | 12.34 (77.125%) |
| | 128 | 3.33 (10.406%) | 8.87 (27.719%) | 20.92 (65.375%) | 24.70 (77.188%) |
| B3 | 4 | 1 (100%) | 1 (100%) | 1 (100%) | 1 (100%) |
| | 8 | 1.75 (87.5%) | 1.97 (98.5%) | 1.80 (90%) | 1.99 (99.5%) |
| | 16 | 3.04 (76%) | 3.68 (92%) | 3.31 (82.75%) | 3.95 (98.75%) |
| | 32 | 4.58 (57.25%) | 6.47 (80.875%) | 5.97 (74.625%) | 7.87 (98.375%) |
| | 64 | 7.00 (43.75%) | 11.47 (71.688%) | 10.59 (66.188%) | 15.04 (94%) |
| | 128 | 8.17 (25.531%) | 15.86 (49.563%) | 19.94 (62.312%) | 30.14 (94.188%) |

Interactive mechanized proof for model transformations properties is an active research area. Calegari et al. encode ATL model transformations and OCL contracts into Coq to interactively verify whether the transformation is able to produce target models that satisfy the given OCL contracts [5]. Stenzel et al. propose a Hoare-style calculus, developed in the KIV prover, to analyze transformations expressed in (a subset of) QVT Operational [20]. UML-RSDS is a tool-set for developing correct-by-construction model transformations [15]. It chooses well-accepted concepts in MDE to make their approach more accessible to model transformation developers. Once the development is achieved, transformations are verified against contracts by translating both into interactive theorem provers. None of these research efforts addresses proving the equivalence of the sequential and the distributed executions of a transformation.

Finally, one can argue that we could have chosen any other back-end language or framework instead of Spark. In particular, some back-ends automatically extracted from Coq, which would enhance the automatic certification of our pipeline. For instance, we could have chosen to extract Haskell code from CoqTL and then use the *Haskell distributed parallel Haskell* (HdpH) language to introduce parallelism and distribution. Similarly, we could have extracted OCaml code from the CoqTL specifications and then use the BSML [16] library for parallelization. This is indeed possible and could be the subject of future work. Note that the three optimizations introduced in this paper, and proven equivalent to the initial CoqTL specification, would be useful for any back-end that introduces parallelism and distribution. However, we want to highlight that the goal of our work is to bridge the gap between certified model transformations and data analytics frameworks, such as Spark. Indeed, as explained in the introduction, choosing Spark paves the way towards large-scale model transformations. Furthermore, Spark is broadly adopted by companies and is available on most Cloud platforms.

## 7 Conclusion and Future Work

In this paper, we presented a refinement of the CoqTL specification, designed for optimizing the parallel execution of model transformations on Spark. We have illustrated the benefits and the scalability of our proposed optimizations through the Relational2Class example.

In future work, we plan to continue experiments with other use cases (e. g., the IMDB case study). We plan to write a compiler from CoqTL to Scala, to automatically obtain an executable transformation from a Coq specification. Finally, we will study other optimizations, leveraging the vertex-centric paradigm supported by Spark (i. e., GraphX), and the integration with persistence solution (e. g., HDFS).

## Acknowledgment

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

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several universities as well as other organizations (see https://www.grid5000.fr). These experiments also used the EnosLib library (see https://gitlab.inria.fr/discovery/enoslib).

## References


[1] University of Malaga Atenea team. 2018. Lintra. http://atenea.lcc.uma.es/projects/LinTra.html.

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

[3] Amine Benelallam, Massimo Tisi, Jesús Sánchez Cuadrado, Juan de Lara, and Jordi Cabot. 2016. Efficient model partitioning for distributed model transformations. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016 (SLE 2016)*, Tijs van der Storm, Emilie Balland, and Dániel Varró (Eds.). ACM, 226–238. https://doi.org/10.1145/2997364.2997385

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

[5] Daniel Calegari, Carlos Luna, Nora Szasz, and Álvaro Tasistro. 2011. A Type-Theoretic Framework for Certified Model Transformations. In *13th Brazilian Symposium on Formal Methods*. Springer, Natal, Brazil, 112–127. https://doi.org/10.1007/978-3-642-19829-8_8

[6] Zheng Cheng, Massimo Tisi, and Joachim Hotonnier. 2020. Certifying a Rule-Based Model Transformation Engine for Proof Preservation. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems*. Montreal, Canada. https://doi.org/10.1145/3365438.3410949

[7] The Coq development team. 2004. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr Version 8.0.

[8] Youssef El Bakouny and Dani Mezher. 2018. Scallina: Translating Verified Programs from Coq to Scala. In *Programming Languages and Systems*, Sukyoung Ryu (Ed.). Springer International Publishing, Cham, 131–145.

[9] Tassilo Horn, Christian Krause, and Matthias Tichy. 2014. The TTC 2014 Movie Database Case.. In *TTC@ STAF*. Citeseer, 93–97.

[10] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. 2008. ATL: A model transformation tool. *Sci. Comput. Program.* 72, 1-2 (2008), 31–39. https://doi.org/10.1016/j.scico.2007.08.002 Special Issue on Second issue of experimental software and toolkits (EST).

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

[12] Dimitris S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. 2006. The Epsilon Object Language (EOL). In *Proceedings of the Second European Conference on Model Driven Architecture: Foundations and Applications* (Bilbao, Spain) *(ECMDA-FA'06)*. Springer-Verlag, Berlin, Heidelberg, 128–142. https://doi.org/10.1007/11787044_11

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

[14] Ralf Lammel. 2008. Google's MapReduce programming model - Revisited. *Science of Computer Programming* 70, 1 (2008), 1 – 30. https://doi.org/10.1016/j.scico.2007.07.001

[15] Kevin Lano, T. Clark, and S. Kolahdouz-Rahimi. 2014. A framework for model transformation verification. *Formal Aspects of Computing* 27, 1 (2014), 193–235. https://doi.org/10.1007/s00165-014-0313-z

[16] Frédéric Loulergue, Frédéric Gava, and David Billiet. 2005. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In *Computational Science – ICCS 2005*, Vaidy S. Sunderam, Geert Dick van Albada, Peter M. A. Sloot, and Jack J. Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1046–1054.

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

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

[19] Jolan Philippe, Héléne Coullon, Massimo Tisi, and Gerson Sunyé. 2020. Towards Transparent Combination of Model Management Execution Strategies for Low-Code Development Platforms. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (Virtual Event, Canada) *(MODELS '20)*. Association for Computing Machinery, New York, NY, USA, Article 72, 10 pages. https://doi.org/10.1145/3417990.3420206

[20] Kurt Stenzel, Nina Moebius, and Wolfgang Reif. 2015. Formal verification of QVT transformations for code generation. *Software & Systems Modeling* 14 (2015), 981–1002.

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

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

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