On the Concurrent Execution of Model Transformations with Linda

Loli Burgueño
Universidad de Málaga
ETSI Informática
Bulevar Louis Pasteur, 35. (29071)
Malaga, Spain
loli@lcc.uma.es

Javier Troya
Universidad de Málaga
ETSI Informática
Bulevar Louis Pasteur, 35. (29071)
Malaga, Spain
javiertc@lcc.uma.es

Manuel Wimmer
Business Informatics Group
Vienna University of Technology
Favoritenstrasse 9-11. (1140)
Vienna, Austria
wimmer@big.tuwien.ac.at

Antonio Vallecillo Universidad de Málaga
ETSI Informática
Bulevar Louis Pasteur, 35. (29071)
Malaga, Spain
av@lcc.uma.es

ABSTRACT

Nowadays there exists a wide variety of model transforma-tion languages. However, all of them present limitations, mainly performance issues, when the complexity and size of model transformations and models grow. The problems arise due to the in-memory allocation of large models as well as the time taken by the execution engines for producing the output models. This restricts the benefits of using model transformations in different application fields of model en-gineering where the complexity of the transformation tasks exceeds the capabilities of sequential execution engines. In this paper we tackle these limitations by introducing concur-rency for model transformations to effectively improve the execution performance. Instead of reinventing the wheel, we base our approach on Linda, a mature coordination lan-guage for parallel processes. We explore how model trans-formations fit into Linda and show a set of basic mechanisms to enable concurrent model transformations. Initial results of applying our approach show a great potential of using Linda to improve the execution performance with respect to existing approaches.

Keywords

Model transformation, concurrency, Linda, tuple spaces

1. INTRODUCTION

In model engineering [2], models are used as the main artifacts for capturing knowledge about different domains. One prominent example for model engineering is model-driven software engineering [3, 24], where models are used in a generative manner to produce software systems in a semi-automatic way. Model transformations (MTs) are so to speak at the heart of MDE, by providing the mechanisms for manipulating and transforming models. MTs can be classi-fied according to different characteristics [7, 19]: abstraction level of input and output models (i.e., horizontal vs. verti-cal transformations), directionality (i.e., uni-directional vs. bi-directional transformations), manipulation of input and output models (i.e., in-place vs. out-place transformations), etc. In model-driven software engineering, out-place trans-formations are very popular for implementing model compil-ers that produce concrete code from more abstract models. However, transformations are not limited to be applied for producing code. For instance, model transformations are becoming also a promising approach to deal with data inte-gration, especially when complex data structure are involved such as in Social Web data management [31].

Because of this increasing variety of model transformation scenarios, there exists a wide range of different languages with which model transformations can be developed, each of them comprises different characteristics [6]. Some examples of transformation languages from different language cate-gories are GrGen (graph transformation) [14], Kermeta (im-perative) [20], QVT-R (declarative) [12], ATL (hybrid) [15] and UML-RSDS (general purpose MDE tool) [18]. How-ever, all of them present limitations, mainly performance is-sues, when the complexity and size of model transformations and models grow [5, 16]. The problems arise due to the in-memory allocation of large models as well as the time taken by the execution engines for producing the output models. This restricts the benefits of using models and model trans-formations in different application fields of model engineer-ing, e.g., biology, medicine, and sociology, where the com-plexity of the models and model transformation tasks is such that sequential execution engines are not sufficient anymore.

In this paper we tackle these limitations by introducing concurrency for model transformations to effectively improve the execution performance. However, instead of reinventing the wheel, we base our transformation approach on Linda [10], a mature coordination language for parallel processes. Linda supports to read and write data to distributed tuple spaces as its basic primitives. For running transformations on such architecture, we explore how model transformations fit into the Linda framework. In particular, we report on the representation of metamodels and models as tuples, how trace links are encoded to allow for efficient retrieval, and how the execution of transformation rules is distributed over tuple spaces. Initial results of applying our approach to the classical Class2Relational case study show already a great potential of using Linda to improve the execution performance with respect to existing approaches.

The outline of this paper is as follows. In Section 2 we introduce the Linda framework and how model transformations are embedded in this framework. In Section 3 we apply the Linda-based transformation approach to the well-known Class2Relational case study and investigate on the execution performance. In particular, we compare our approach against the ATLAS Transformation Language (ATL) [15]. While we present related work in Section 4, we conclude the paper with an outlook on future work in Section 5.

2. LINDA MEETS MODEL TRANSFORMATIONS

In this section we show the basics of Linda and how model transformations can be embedded into the Linda approach.

2.1 Linda Basics

Linda is a coordination language for parallel and distributed processing and provides a communication mechanism based on a logically shared memory space called *tuple space*. It was first proposed by David Gelernter at Yale University in the mid-1980's [9] and in recent years there has been a resurgence in interest, particularly with regard to Java implementations of Linda [28, 29]. On distributed memory systems, such as networks of workstations, the tuple space is usually distributed among the processing nodes. Independent of the implementation strategy employed, the tuple space is structured as a bag of tuples. An example of a tuple with four fields is `("circumference", 3, 47, 53)`, where 3 is the radio, and 47 and 53 indicate the position (x and y coordinates) of the circumference represented by this tuple. As a coordination language, Linda is conceived to be integrated with a sequential programming language, which is called the host language. Linda provides operations to place tuples into tuple spaces (`write` operations) and to retrieve tuples from them (`read` operations). Read operations can be either blocking or non-blocking. Also, the specification of the tuple to be retrieved makes use of an associative matching technique whereby a subset of the fields in the tuple have their values specified.

For implementing our approach, we have used the Java implementation of an in-memory data grid offered by Gigaspaces Technologies [11]. It is called *XAP Elastic Caching Edition* (we will refer to it from here on as XAP) and supports the basic Linda operations as well as a wide range of new features that offer good results with low latency, such as fast data access, performance, and scalability. Linda, and consequently XAP, allows several machines to work simultaneously over a tuple space that can be distributed transparently to the user. Since XAP deals with Java code, Java objects can be introduced and extracted from tuple spaces. Internally, XAP serializes the objects before introducing them in a tuple space and deserializes them when extracting them.

XAP allows to create as many distributed tuple spaces as desired. These spaces can allocate big loads of information in RAM memory, since they can be distributed in different machines, where smaller chunks of the information are stored in each machine. Different execution threads, which may come from different machines, can read and write the information on these tuple spaces. XAP–internally and in a transparent way to the user–deals with the concurrent mechanisms that offer this parallelization.

2.2 Model Transformations based on Linda

In this subsection, we first show how metamodels and models are encoded in tuple spaces, and subsequently, we present how transformations are implemented based on this encoding.

2.2.1 Implementing Metamodels and Models

The first ingredients needed to define a MT are the metamodels of the source and target domains. In our approach, since we are dealing with Java code, metamodels are represented by means of Java classes. XAP requires these classes to implement the Java `Serializable` interface. Furthermore, `getter` and `setter` methods have to be defined in order to have access to the attributes afterwards. Our models, consequently, are composed of instances (objects) of these classes. Transformation languages such as ATL [15] or QVT [21] require metamodels to conform to Ecore [4]. Models conforming to these metamodels are typically stored in XML Metadata Interchange (XMI) files. We plan as future work to obtain our Java representation from such metamodels and models by means of model-to-text (M2T) transformations. Besides, the representation used throughout this paper has been adopted considering the particular example shown in Section 3.

We define two tuple spaces, one for storing the input model and one for the output model. From here on we refer to them as source tuple space and target tuple space, respectively. As described before, these tuple spaces can be distributed on several machines. This means that some parts of the model, i.e., some Java objects, can be in different machines than others as is also depicted by Figure 1.

We provide every object with an identifier, which is stored in an attribute of type `String` named `id`. The value of the identifier has the following form: `<package_name>.<class_name>_Integer`. By this convention, identifiers have information about the metamodel, in particular, about the metaclass and its container package, and they also contain an integer number that makes the value universally unique. For instance, the identifiers for two objects of type `Attribute` of the metamodel shown in Figure 2 may be "`classMM.Attribute_42`" and "`classMM.Attribute_43`".

Attributes of the meta-classes are represented in the corresponding Java also as attributes (i.e., fields), because this is enough to represent simple values stored in Java objects.

Finally, we need a mechanism to store information about links between objects. As in Ecore, we consider two types of relationships between classes: references and containment references. In our representation, each class has an attribute of type `String` for every outgoing relationship with the name `<relationship_name>ID`, where the identifier of the referenced object is stored. Additionally, there is another attribute of `Boolean` type for each relationship, with the name `<relationship_name>IsComposed`, which keeps a `true` value if the reference is of type containment and `false` otherwise.

2.2.2 *Implementing Model Transformations*

As first step in the transformation's execution, the input model, i.e., a set of Java objects, is loaded into the source tuple space. The identifiers of the objects of each class go from `<package_name>.<class_name>_1` until `<package_name>.<class_name>_n`, where `n` is the number of instances of each type. In our current naive implementation, input models are generated from scratch with the help of an external Java program. For future versions, we plan on receiving input models in XMI (or similar) format and translating them into our representation by means of M2T transformations. Thus, identifiers for objects would be automatically created by the transformation.

Identifiers of the objects in the output model are considered to be slightly different in our approach. They comprise the following form: `<package_name>.<class_name>_<rule_name>_Integer`, where the integer value is determined by the input model. By this convention, information about the rule that creates an output object is also contained because of the following two reasons:

1. **Uniqueness**: Taking into account that the integer value at the end of the identifier is the same as that of the input object from which the element is created, it may happen that two target objects have the same integer value if the input objects from which they are created are of different type. Consider, for example, the `Class` and `Relational` metamodels (cf. Figures 2 and 3), and let us suppose that objects of type `Class` and `Attribute` produce objects of type `Column` by means of `Rule1` and `Rule2`, respectively. If no information about the rule is added in the identifier of output objects, then the class `ClassMM.Class_3` and the attribute `ClassMM.Attribute_3` produce both an object with the same identifier, `RelationalMM.Column_3`. However, if we introduce information about the rules, the identifiers of the output objects are `RelationalMM.Column_Rule1_3` and `RelationalMM.Column_Rule2_3`.

2. **Tracing**: It allows to keep track of from which input object a specific output object is created. This is similar as the trace link concept used in ATL [15] or the trace model used in [26]. In this way, if we see an object with the identifier `RelationalMM.Column_Rule1_3` in the output model, and we know that `Rule1` receives objects of type `Class` as input, then we know that such object was created from object `ClassMM.Class_3` by means of `Rule1`

When more than one output object is created from an input object, an additional field is added in the identifiers to avoid ambiguities. The main element created, e.g., in an ATL rule, it is the first element created in a matched rule, has an identifier as described so far. The remaining elements have an extra integer number that gives the order in the creation. For example, if object with id `ClassMM.Class_3` produces a table and two columns by means of `Rule1`, the identifiers of these three objects are `RelationalMM.Table_Rule1_3`, `RelationalMM.Column_Rule1_3-1` and `RelationalMM.Column_Rule1_3-2`. In this initial version of our paper, we deal with output objects created only from one input object, i.e., one-to-many transformation rules.

A strong point of our approach is that several execution threads can deal with the transformation at the same time as illustrated by Figure 1. We have based our initial implementation presented in this paper on ATL. There is a Java class for every matched rule in an ATL transformation (for a concrete example see Section 3.2). Each of these classes defines how a specific set of target objects are created from a specific set of source objects. In fact, we use the same matching and filtering strategies as used for ATL matched rules. Consequently, the behavior of the transformation in our implementation is equivalent to the ATL implementation. Source objects are read in blocks and target objects are produced in blocks, too. Thus, a specific thread reads a set of objects of the same type (with the `takeMultiple` operation) from the source tuple space, creates a set of output objects, and introduces them in the target tuple space (with the `writeMultiple` operation). In this first implementation, each Java class is assigned to a specific number of threads, as we describe in more detail in Section 3.3. The reading and writing in blocks is more efficient than reading and writing objects one-by-one. The internal mechanisms of the XAP implementation of Linda assure that no two threads take the same elements from the source tuple space at the same time.

For reasons of navigability, input objects should remain in the source tuple space all the time. Otherwise, if a transformation rule requires to navigate from the input object to another input object which is no longer in the tuple space, a `NullPointerException` would be fired. However, at the same time, if objects remain always in the source tuple space, special care has to be taken not to fire transformation rules again and again for the same input objects. Here we have to consider how the tuple space is accessed. In particular, there are two types of operations for consulting a tuple space, `read` operations and `take` operations. The former keep the consulted tuples in the space, while the latter take them out. Therefore, there are two possibilities for extracting input objects from the source tuple space in order to transform them into output objects. On the one hand, if we use `read` operations to read these objects, then the transformation never stops, as mentioned before, because there are always objects matching the preconditions. On the other hand, if we use `take` operations, we remove objects from the source tuple space that may be needed afterwards for navigation because they are referenced by other objects being transformed. For this reason, we have added a `Boolean` attribute named `transformed`, which is nothing but a *flag*, to every class. This attribute is `false` at the beginning of the transformation's execution for every object, meaning that they have not been transformed. Then, input objects that have not been transformed can be extracted from the source tuple space with the `takeMultiple` operation. Straightaway, these objects are stored locally and they are written again in the source tuple space, now with the `transformed` attribute set to `true`, so that these objects are not taken again by any execution thread. Finally, the copy of the objects stored locally is used to create the output objects. Our Java classes (those corresponding to ATL rules), consequently, take into account this flag in order to select which input objects are to be transformed.

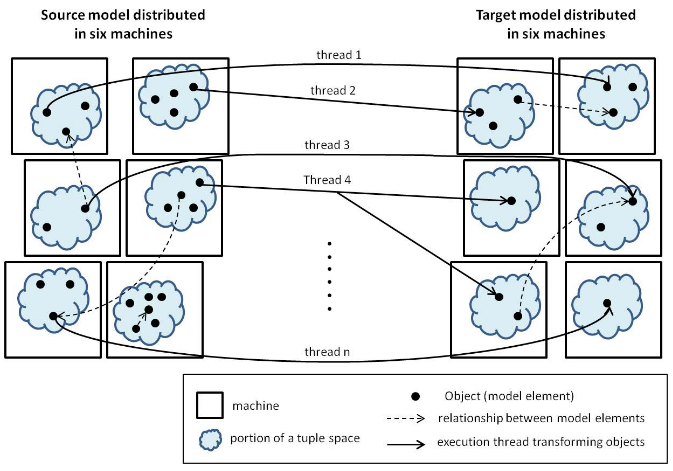
Source model distributed in six machines
Target model distributed in six machines
thread 1
thread 2
thread 3
Thread 4
thread n
machine
portion of a tuple space
Object (model element)
relationship between model elements
execution thread transforming objects

Figure 1: Concurrent and distributed model transformation approach at a glance

2.2.3 *Summary*

A global view of the approach is presented in Figure 1. We can see how the input and output tuple spaces are partitioned and distributed among several machines (six for each space in the illustrated example). The input and the output models are stored in such partitions. Several threads access the input model and create the output model at the same time. Please note that the threads shown in the figure may either belong to the machines that store the tuple spaces or not, because any machine can access these tuple spaces. By having the models stored and the transformation executed in several machines, this approach presents a high degree of scalability for several reasons. First, large models are distributed over several machines, so that the memory in these machines is not overloaded. Second, any number of machines can access the tuple space, even if the space is not physically in these machines, so that the core(s) of each machine contribute to the transformation process. In fact, the more cores available, the more threads can be defined without the risk of context switching, so that the transformation is executed faster.

Let us point out than in our initial implementation, we have made experiments only with one machine. However, this machine has 16 cores and all of them participate in the transformation's execution, as we describe in more detail in Subsection 3.3.

3. CLASS2RELATIONAL CASE STUDY

The *Class2Relational* example[1] transforms simplified UML class diagrams to simplified relational database schemas. In this section, we show for this example (*i*) how the metamodels and models are represented, (*ii*) how the transformation rules are expressed, and (*iii*) we present some performance measurements for this example to reason about the scalability of the presented approach.

[1] It is available at: `http://www.eclipse.org/atl/atlTransformations`

3.1 Metamodel/Model Representation

The Ecore representation for the source and target metamodels is presented in Figures 2 and 3, respectively.

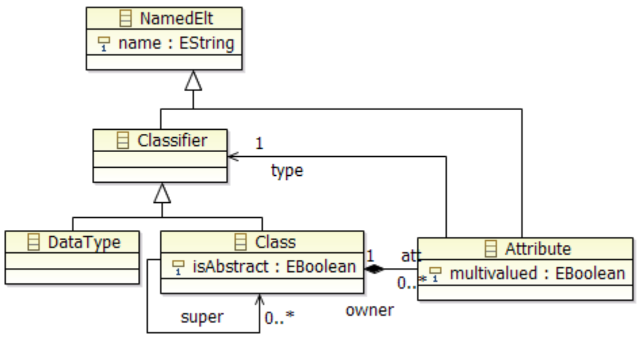

Figure 2: Class metamodel

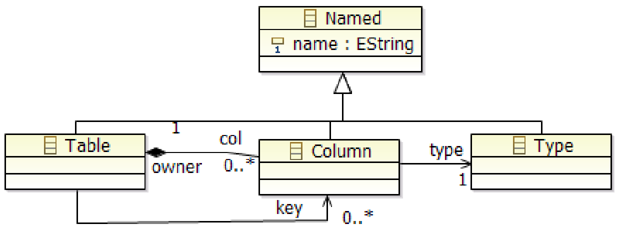

Figure 3: Relational metamodel

As we explained before, our representation of the metamodels is done in Java. We need a Java class for each class in the Ecore representation. Attributes of the Ecore classes are converted to fields in our representation and we represent relationships as explained in Section 2.2. An excerpt of the code for the `Attribute` class (cf. Figure 2) is depicted in Listing 1.

Listing 1: `Attribute class`

```
package classMM;
public class Attribute extends NamedElt {

String id;
Boolean multivalued;
String typeID; Boolean typeIsComposed = false;
String ownerID; Boolean ownerIsComposed = false;
Boolean transformed;

public Attribute(){ }

public Attribute (String id, String name, Boolean
    multiValued, String owner, String idType){
  super(name);
  this.id = id;
  this.multivalued = multiValued;
  this.ownerID = owner;
  this.typeID = idType;
  transformed = false;
}

public String getId() {
```

```
22   return id;
23 }
24
25 public void setId(String id) {
26   this.id = id;
27 }
28 ...
```

The `getter` and `setter` methods for the remaining attributes are not shown. There is a constructor with no attributes (requirement of XAP), and there is another one whose arguments are the attributes and relationships of the class.

3.2 Transformation Rule Representation

For exemplifying how transformation rules are implemented, we present one of the rules from the *Class2Relational* ATL transformation before we discuss how exactly this rule is realized with Linda. It is the `ClassAttribute2Column` rule, that takes non-multivalued `Attribute` instances (which, in turn, have as type an instance of `Class` assigned) as input and transforms them into `Column` instances. The column's name must be the attribute's name concatenated with "Id", and its `type` reference must point to a `Type` instance with name "Integer". If no object of type `Type` with name "Integer" exists, then one has to be created.

This rule is implemented in ATL as shown in Listing 2. The helper named `objectIDType` retrieves a `Type` instance with name "Integer".

Listing 2: `ClassAttribute2Column` rule in ATL

```
rule ClassAttribute2Column {
  from
    a : Class!Attribute (
      a.type.oclIsKindOf(Class!Class) and
      not a.multivalued
    )
  to
    out : Relational!Column (
      name <- a.name + 'Id',
      type <- thisModule.objectIdType
  )
}
```

In our Java implementation, a rule in ATL corresponds to a class in Java. Furthermore, the matches and filters we define are the same as those used by ATL. An exceprt of the code of the class implementing the `ClassAttribute2Column` is illustrated in Listing 3.

Listing 3: `ClassAttribute2Column` rule of the Linda-based solution

```
1  Attribute[] atts=retrieveElements(numElements);
2  if (atts.length > 0){
3    Column[] cols=new Column[attributes.length];
4    for (int i = 0; i < atts.length; i++) {
5     cols[i]=new Column(CommonMeth.genOutId(atts[i],1),
6          atts[i].getName() + "Id",
7          CommonMeth.objectIDType(gigaSpaceTrg));
8    }
9    gigaSpaceTrg.writeMultiple(cols);
10 }
```

The method `retrieveElements` (Listing 3, line 1) retrieves a block of objects (the number of objects retrieved is given by `numElements`) that match the rule's precondition. This is, it removes the elements from the input tuple space and place them locally in the `atts` array. The code of the `retrieveElements` method is shown in Listing 4. Please note that this method is located within the class representing the `ClassAttribute2Column` rule.

Listing 4: `retrieveElements` method

```
private Attribute[] retrieveElements(n) {
  Attribute[] atts=gigaSpaceSrc.takeMultiple(
    new SQLQuery<Attribute>(Attribute.class,
      "typeID like 'inMM.Class_\%' and multivalued=
           false and transformed=false"), n);
  if (atts.length>0){
    for (Attribute a : atts){
      a.setTransformed(true);
    }
    gigaSpaceSrc.writeMultiple(atts);
  }
  return atts;
}
```

XAP offers the possibility of querying the tuple spaces by means of SQL-like syntax. The method `retrieveElements` uses this mechanism in order to take the elements that fulfil the specified constraints. In this case, it returns the elements of type `Attribute` whose `type` reference is an instance of `Class`, which are not multi-valued and have not been transformed yet. The more complex issue here is how to know which class the object referenced by `type` belongs to. According to our representation, we have an attribute named `typeID`, that keeps the identifier of the referenced object (cf. line 6 in Listing 1). This is why object identifiers are strings with the signature: `<package_name>.<class_name>_Integer`, so that the class to what they belong is known and the problem of setting the restriction is reduced to a string comparison with the SQL operation `LIKE`. Finally, this method marks the elements as transformed and put them back in the source tuple space again, as explained in Section 2.2.

Going back to the code implementing the `ClassAttribute2Column` rule (i.e., Listing 3), if there are attributes to be transformed (line 2), then an array is created (line 3) with the purpose of storing the columns to be created for writing them afterwards in the target tuple space all at once. Then, for each attribute retrieved (line 4), a new column is created (lines 5, 6, and 7). Fresh identifiers for the columns are generated with the `genOutId` method, placed in a class accessible by all Java classes implementing rules called `CommonMeth`. This method is called in the first parameter of the columns' constructor. The code for `genOutId` method is shown in Listing 5.

Listing 5: `genOutId` method

```
1 public static String genOutId(Attribute attr, int
     numObj) {
2   String attrNumber=attr.getId().substring(
3         attr.getId().indexOf("_")+1);
4   if (isTypeOf(classOf(attr), DataType.class) &&
5       !attr.getMultivalued()){
6     return "outMM.Column_" + attrNumber;
7   } else if (isTypeOf(classOf(attr), Class.class ...
8 }
```

It receives as parameter the attribute instance and returns the identifier of the element in which it will be transformed. Since a rule can create more than one element, the second parameter is necessary in order to know the concrete element for what we are requesting the identifier. Line 1 shows that the first step is to extract the number from the identifier, removing all the other parts like the package name and the class name. Later, it checks the constraints that the input objects in the rule satisfy. For example, our rule transforms attributes whose `type` feature is `DataType` and are not multivalued, another rule can transform attributes whose `type` feature is `Class` and are multivalued. All these constraints need to be taken into account because the object(s) created depend on them, and thus, their identifiers.

When creating a column (lines 5-7 in Listing 3), the second parameter of the constructor sets the column's `name` and the third one makes use of the static method `objectIDType()`, also placed in class `CommonMeth`. This method looks in the target tuple space for a `Type` instance with name "Integer". If it does not exist, it is created; otherwise it is retrieved. The method returns the identifier of the retrieved (or created) object. Its Java code is displayed in Listing 6.

Listing 6: `objectIDType` method

```
1  public static String objectIDType(GigaSpace
       gigaSpaceTrg) {
2    Type t = new Type();
3    t.setName("Integer");
4    Type t2 = gigaSpaceTrg.read(t);
5    if (t2==null){
6      t.setId("outMM.Type_Integer");
7      gigaSpaceTrg.write(t);
8      return "outMM.Type_Integer";
9    } else {
10     return t2.getId(); }
11 }
```

We can resort on that in order to know if an object with name "Integer" exists, we only need to look for it in the target tuple space (line 4 in Listing 6) with the `read` operation that XAP provides. If it does not exist, we create it and set its features (lines 6 and 7) using the `write` operation. Finally, the identifier is returned in line 8 (if it was created) or line 10 (if it already existed).

The rule execution ends up with the last step (Listing 3, line 9), when all the columns created are stored in the target tuple space. They are all stored at once, with the `writeMultiple` method.

For the complete source code of this example as well as for a detailed description of this example, we kindly refer the interested reader to [1].

3.3 Performance Evaluation

In this subsection, we discuss the performance of our transformation approach by reporting results of an experiment. By following the guidelines by Roneson and Hörst [23], we used purposive synthetic scenarios for the aforementioned Class2Relational transformation.

3.3.1 Setup

The main goal of the presented approach is to provide good scalability, i.e., to load, represent, and transform very large models. Thus, to validate our approach, we aim to answer the following three research questions (RQ) based on the Class2Relational example:

- RQ1: What is the absolute transformation time for a representative set of large input models?
- RQ2: What are the input model size boundaries for our approach?
- RQ3: What is the relative improvement with respect to existing state-of-the-art approaches?

For answering RQ1 and RQ2, we have built several considerably large Class models to see how long the Class2Relational transformation execution runs take. For answering RQ3, we have done a comparison with ATL since it is one of the most used transformation language in academia as well as in industry. It is important to point out here that we have implemented our prototype in an equivalent way as ATL. This is, we have a Java class for each ATL rule. Furthermore, rule matching, filtering, and model element generation is performed in the same way as in ATL [26].

To compare the execution times, we have executed both transformations, i.e, the Linda-based version and the ATL version, on a machine running 64-bit Linux with 11.7 Gb of RAM memory and 16 cores of 2.67GHz. For executing the Linda-based version, we launched 16 execution threads, where each thread is assigned to one unique core and deals with a rule and a certain number of objects to transform. Please notice that in these initial experiments, we have not distributed the tuple spaces to different machines, this is, both the source and target tuple spaces are created in the same machine. However, all the 16 cores are used to concurrently execute the Class2Relational transformation. For executing the ATL version, we employed the EMF-specific virtual machine of ATL in its version 3.3.1.

An important remark is the number of threads assigned to each rule (i.e., Java class). We mentioned before that each Java class corresponds to an ATL rule, and that each thread transforms objects of a specific type (in fact, they transform the objects filtered by each rule). We have 16 cores, so we have created 16 execution threads to avoid context switching. In our concrete implementation, we have assigned more threads to those rules that are *heavier*. The more times a rule has to navigate the input model, the heavier it is. In our example, this navigation is required during the creation of objects (when an object needs data from other objects in order to be created). So, for example, in our example we have assigned 7 threads to the heaviest rule, `Class2Table`, and 3 to other rules such as `SingleValuedDataTypeAttribute2Column`, and 1 to the `DataType2Type` rule. When a thread is released, it is dedicate to help with the heaviest rule. We have come up with this configuration of threads after running several experiments for this specific case study using different configuration and conclude which one is the most effective. As part of our future work, we would like to automatize this choice.

3.3.2 Results

For collecting the measures to answer the stated research questions, we are using the following metrics:

- CPU Time: Execution time in seconds, without the time for loading the input model and storing the output model.
- Model size: Number of objects instantiated from metaclasses contained in the models.
- Speedup: Ratio between the execution time of the Linda-based transformation and the ATL transformation.

We decided to measure the CPU time only for the transformation execution, because so far, our models do not have a concrete and persistent representation. Thus, we make use

#Objects	ATL(sec)	Concurrent(sec)	Speedup
27,500	14.984	7.305	2.051
55,000	56.686	21.053	2.693
82,500	95.175	49.365	1.928
110,000	275.258	91.838	2.997
137,500	319.227	142.679	2.237
165,000	495.881	195.698	2.534
192,500	681.375	239.270	2.848
220,000	933.449	278.274	3.354
247,500	1,005.769	394.522	2.549
275,000	Exception	492.536	-

Table 1: Results of the performance evaluation (up to 275,000 elements).

of a Java program to build automatically the input model (create the objects) in memory and store them in the source tuple space at runtime.

We explored that the first time an ATL transformation is executed, it lasts for more than for succeeding runs. This is because the ATL virtual machine is *cold* in the beginning, and it needs to be *warmed up*. Since we want our approach to be applied on big models and model transformations, we consider that we should take into account the time spent by ATL in the first execution. For this reason, the times shown for the ATL execution have been taken from the first run, where the virtual machine has not been *warmed up* before, i.e., a fresh instance of the virtual machine is used for each run.

The second and third columns of Table 1 summarize the execution times for ATL and our approach, respectively.

3.3.3 Discussion of the results

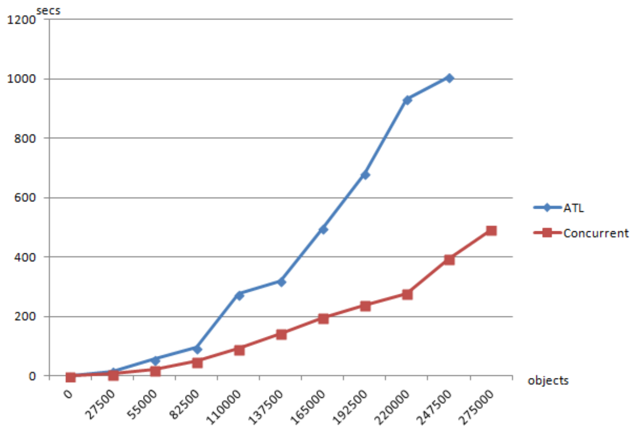

Figure 4: ATL and Linda-based transformation performance in comparison.

Figure 4 shows the execution times for both ATL and our approach for input models composed of up to $275,000$ objects. The last value for ATL is not displayed as ATL is not able to transform such a big model on our machine. In fact, it throws the error *GC overhead limit exceeded*. The problem that caused this exception is that almost all the time is spent in garbage collection and the ATL program is making little or no progress because the Java heap is too small for such a load.

#Objects	Time (hours)
550,000	0.545
1,100,000	2.407
1,650,000	5.269
2,200,000	9.049

Table 2: Results of the performance evaluation (up to 2,200,000 elements).

In both cases, the times have an asymptotic quadratic growth and tend to fit the polynomial function with a correlation coefficient of 0.99 and whose formulas are:

$$y_{ATL} = 13.6122 + 1.748 \times 10^{-8} x^2$$

$$y_{concurrent} = 8.5956 + 6.2759 \times 10^{-9} x^2$$

As the two functions are of degree two, an interesting value which allows us to estimate how many times the concurrent approach is faster than ATL, namely *speedup*, can be computed with the following formula:

$$speedup_i = \frac{y_{ATL_i}}{y_{concurrent_i}}$$

For our executions, the *speedup* is computed in the fourth column of Table 1 and, as average, this value is 2.57, what means that for this concrete example the concurrent approach is more or less two and a half times faster.

A representative measure of the performance is response time. Table 2 and Figure 5 give us an idea of how long the transformation with our approach takes for large input models. For example, the largest model we run counts on $2,200,000$ objects and takes about 9 hours to get the result.

A restriction in our approach is that our implementation uses an in-memory data grid (which stores the data in RAM memory) so another important point to consider is the amount of memory the models need in order to be stored and transformed. The last model transformed in Figure 5 is the biggest model that our machine (11.7 Gb) is able to transform due to the memory limitations.

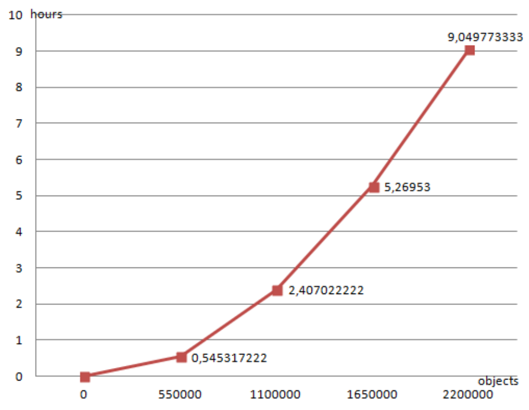

Figure 5: Results of Linda-based approach for very large models.

Nevertheless, since Linda's tuple spaces can be distributed over several machines, this would solve the problem of the memory limitation as long as we count on enough machines where to distribute the spaces.

With the results presented in this subsection we have answered the research questions presented earlier. The runtime performance of our implementation seems appropriate (RQ1). We also obtain a good scalability for input models, since we can transform models with up to 2,200,000 elements (RQ2). Finally, the performance and scalability both in response time and size of input models are better than ATL's for the given scenario (RQ3). Recall that the results obtained are for the Class2Relational case study with our specific and non-generalized implementation. As future work we want to consider more case studies and to develop a more generic implementation.

3.3.4 Threats to Validity

Internal validity. The threats to internal validity are the factors which might affect the results in the context of our experiments. As we have stressed in previous subsections, our experiments have been carried out for the concrete example of the Class2Relational case study. It is also worth mentioning that the number of elements filtered by each rule is proportional in the different input models we have used. This means that, e.g., more elements are transformed by rule `SingleValuedDataTypeAttribute2Column` than by rule `ClassAttribute2Column` because there are always more non-multivalued attributes with `type` an instance of a `DataType` than with `type` an instance of a `Class`. For this reason, more threads have been assigned to the former rule than to the latter. If we had not had any information about the quantity of each type of elements in the input models, the distribution of the threads would have been different and we would have obtained different performance measures.

External validity. Such threats may hinder that the results of the experiment are generalizable. The main threat to external validity is the fact of having experimented only with one case study. The current implementation of our approach has been conceived for a specific case study. Also, the ATL implementation of this case study has served us as a model for our implementation. If we had used a different ATL realization (e.g., using the imperative language features of ATL) or a different transformation language as starting point for this case study, we would have probably come up with a different implementation which could have been better or worse in terms of performance. Another threat of external validity is the knowledge about the number of cores available beforehand. In our experiments, we knew we counted on 16 cores, so we defined 16 threads and assigned them to the different rules. But, what if we do not know the number of cores beforehand? We have to take this into account for future work.

4. RELATED WORK

With respect to the contribution of this paper, we first elaborate on widely related approaches for storing and retrieving very large models, and second, we discuss closely related work considering the performance of model transformations.

Storing/loading very large models. The scalability problems of loading large models represented by XMI documents into memory has been already recognized several years ago. One of the first solutions for EMF models is the Connected Data Objects (CDO)[2] model repository which allows to store models in all kinds of database backends such as traditional relational databases or emerging NoSQL databases. CDO supports the ability to store and access large-sized models due to the transparent loading single objects on demand and caching them. If objects are no longer referenced, they are automatically garbage collected. There are also several emerging projects that are considering to store very large EMF models, like MongoEMF[3] and Morsa [8]. Both approaches are built on top of MongoDB, which is used as storage technology. In [5], Clasen et al. elaborate on strategies for storing models in a distributed manner by horizontal or vertical partitioning into the Cloud.

In our approach, we are reusing the storage of Gigaspaces Technologies by transforming models and their associated metamodels to a tuple representation to inherit the good scalability from the underlying technology. Although the stored tuples may be easily distributable, we have not considered this option in our experiments. Furthermore, by exploiting Cloud Computing possibilities, we may also deploy our approach on an Infrastructure-as-a-Service provider to have on demand scale-in and scale-out support as mentioned in [5]. However, a deeper comparison with existing approaches for storing large models is left as subject for future work, because the focus of this paper was on evaluating the transformation performance.

Transforming very large models. Several lines of work consider the transformation of large models. In this paper, we have been focusing on out-place model transformations running in batch mode. In particular, we read the complete input model and produce the output model from scratch by applying all matching transformation rules. However, to deal with large models, orthogonal techniques may be applied. Especially, two scenarios have been discussed in the past that benefit from alternative execution strategies. First, if an output model already exists from a previous transformation run for a given input model, only the changes in the input model are propagated to the output model. Second, if only a part of the output model is needed by a consumer, only this part is produced while other elements are produced just-in-time. For the former scenario, *incremental* transformations [17, 22] have been introduced, while for the latter *lazy* transformations [25] have been proposed. In this paper, we proposed a fundamental approach for parallelizing model transformation executions that may be also combinable with incremental and lazy transformations.

Another interesting line of research for executing transformations in parallel is the work on critical pair analysis [13] in the field of graph transformations. This work has been originally targeted to transformation formalisms that do have some freedom for choosing in which order to apply the rules. Rules that are not in an explicit ordering are considered to be executed in parallel if no conflict, e.g., add/forbid or delete/use conflicts, is statically computed. However, the execution engines follow a pseudo-parallel execution by going back to a sequential application of the rules. But the general notion of critical pairs may be also a valid input for distributing transformation rules. In particular, having non-

[2]`http://projects.eclipse.org/projects/modeling.emf.cdo`

[3]`http://code.google.com/a/eclipselabs.org/p/mongo-emf`

conflicting transformation rules allows for distributing them easier without having negative side-effects.

The performance of model transformations is now considered as an integral research challenge in MDE. For instance, Amstel et al. [27] considered the runtime performance of transformations written in ATL and in QVT. In [30], several implementation variants using ATL, e.g., using either imperative constructs or declarative constructs, of the same transformation scenario have been considered and their different runtime performance has been compared. However, these works only consider the traditional execution engines following a sequential rule application approach. The only work we are aware of dealing with the parallel execution of transformations is [5] where Clasen et al. [5] outlined several research challenges when transforming models in the cloud. In particular, they discussed how to distribute transformations and elaborated on the possibility to use the Map/Reduce paradigm for implementing model transformations.

5. CONCLUSIONS

This paper presents an emerging approach based on Linda for executing model transformations concurrently. Due to the distributed nature of Linda, this approach can be also applied over distributed systems where a model is transformed by several machines simultaneously, increasing significantly the performance of the transformation process.

We have presented a case study where we compare the execution times of a transformation implemented with our approach with the execution times of an equivalent ATL-based transformation. Our approach has proved to be faster for this particular case and also the scalability with respect to the input model's size seems to be fairly good. Nevertheless, this is only the beginning and there are several lines that we still have to explore in future work.

So far, we have defined a Java class for each ATL rule in the transformation. Then, a specific number of threads is assigned to each rule, because we know the number of cores we count on beforehand (16 in our specific setting). However, what if we do not know the number of cores beforehand? For this reason, we would like to make every thread execute the same piece of code. This way, if there are 3 threads working instead of 2, the target model is built faster, but we do not need to modify the implementation nor deal with the types of elements transformed by each thread. For achieving this behavior, we want every thread to execute the whole transformation over a subset of the input model. Then, the more cores/machines participating in the transformation, the more threads are launched for the same piece of code.

Also, and as a more ambitious future line of research, we would like to create our own concurrent model transformation language. This is, right now we are using Java code to implement the transformations, but it would be ideal to count on a language built on top of such implementation. Another possibility is to define a semantic mapping between sequential transformation languages, such as ATL or QVT, and our Linda-based representation, so that transformations written in the former could be executed concurrently by using the later. Here we plan to investigate on a higher-order transformation to produce parallelizable code from high-level transformation languages.

6. ACKNOWLEDGMENTS

This work is partially funded by Research Projects TIN2011-23795 and TIN2011-15497-E.

7. REFERENCES

[1] Atenea. Class2Relational example implemented in Java based on Linda, 2013. `http://atenea.lcc.uma.es/Descargas/MTLL/Class2Relational.zip`.

[2] J. Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[3] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[4] F. Budinsky, E. Merks, and D. Steinberg. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley, 2006.

[5] C. Clasen, M. Didonet Del Fabro, and M. Tisi. Transforming Very Large Models in the Cloud: a Research Roadmap. In *Proceedings of the 1st International Workshop on Model-Driven Engineering on and for the Cloud (co-located with ECMFA)*, 2012.

[6] J. S. Cuadrado. Towards a Family of Model Transformation Languages. In Z. Hu and J. de Lara, editors, *Proceedings of the 5th International Conference on the Theory and Practice of Model Transformations (ICMT)*, volume 7307 of *Lecture Notes in Computer Science*, pages 176–191. Springer, 2012.

[7] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, July 2006.

[8] J. Espinazo-Pagán, J. S. Cuadrado, and J. G. Molina. Morsa: A Scalable Approach for Persisting and Accessing Large Models. In J. Whittle, T. Clark, and T. Kühne, editors, *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, volume 6981 of *Lecture Notes in Computer Science*, pages 77–92. Springer, 2011.

[9] D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[10] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96–107, 1992.

[11] GigaSpaces Technologies Ltd. GigaSpaces, 2013. `http://www.gigaspaces.com/datagrid`.

[12] T. Goldschmidt and G. Wachsmuth. Refinement Transformation Support for QVT Relational Transformations. In *Proceedings of the 3rd Workshop on Model Driven Software Engineering (MDSE)*, 2008.

[13] R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In *Proceedings of the First International Conference on Graph Transformation (ICGT)*, volume 2505 of *Lecture Notes in Computer Science*, pages 161–176. Springer, 2002.

[14] E. Jakumeit, S. Buchwald, and M. Kroll. GrGen.NET - The expressive, convenient and fast graph rewrite system. *STTT*, 12(3-4):263–271, 2010.

[15] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev.

ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

[16] F. Jouault and J.-S. Sottet. An AmmA/ATL Solution for the GraBaTs 2009 Reverse Engineering Case Study. In *Proceedings of the 5th International Workshop on Graph-Based Tools - Grabats 2009 (co-located with TOOLS)*, 2009.

[17] F. Jouault and M. Tisi. Towards Incremental Execution of ATL Transformations. In L. Tratt and M. Gogolla, editors, *Proceedings of the 3rd International Conference on Theory and Practice of Model Transformations (ICMT'10)*, volume 6142 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2010.

[18] K. Lano and S. Kolahdouz-Rahimi. The UML-RSDS manual, 2012. `http://www.dcs.kcl.ac.uk/staff/kcl/uml2web`.

[19] T. Mens and P. V. Gorp. A Taxonomy of Model Transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006.

[20] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In L. C. Briand and C. Williams, editors, *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005.

[21] OMG. *MOF QVT Final Adopted Specification*. Object Management Group, 2005. OMG doc. ptc/05-11-01.

[22] A. Razavi and K. Kontogiannis. Partial evaluation of model transformations. In M. Glinz, G. C. Murphy, and M. Pezzè, editors, *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 562–572. IEEE, 2012.

[23] P. Runeson and M. Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[24] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[25] M. Tisi, S. M. Perez, F. Jouault, and J. Cabot. Lazy execution of model-to-model transformations. In J. Whittle, T. Clark, and T. Kühne, editors, *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS'11)*, volume 6981 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2011.

[26] J. Troya and A. Vallecillo. A Rewriting Logic Semantics for ATL. *Journal of Object Technology*, 10:5:1–29, 2011.

[27] M. van Amstel, S. Bosems, I. Kurtev, and L. F. Pires. Performance in Model Transformations: Experiments with ATL and QVT. In J. Cabot and E. Visser, editors, *ICMT*, volume 6707 of *Lecture Notes in Computer Science*, pages 198–212. Springer, 2011.

[28] G. Wells. New and improved: Linda in Java. *Science of Computer Programming*, 59(1-2):82–96, 2006.

[29] G. Wells, A. Chalmers, and P. G. Clayton. Linda implementations in Java for concurrent systems. *Concurrency - Practice and Experience*, 16(10):1005–1022, 2004.

[30] M. Wimmer, S. Martínez, F. Jouault, and J. Cabot. A Catalogue of Refactorings for Model-to-Model Transformations. *Journal of Object Technology*, 11(2):2:1–40, 2012.

[31] M. Wischenbart, S. Mitsch, E. Kapsammer, A. Kusel, B. Pröll, W. Retschitzegger, W. Schwinger, J. Schönböck, M. Wimmer, and S. Lechner. User profile integration made easy: model-driven extraction and transformation of social network schemas. In *Companion Proceedings of the 21st World Wide Web Conference (WWW)*, pages 939–948. ACM, 2012.