Concurrent Model Transformations with Linda

Loli Burgueño, Javier Troya, and Antonio Vallecillo

GISUM/Atenea Research Group. Universidad de Málaga (Spain)
{loli,javiertc,av}@lcc.uma.es

Abstract. Nowadays, model transformations languages and engines use a sequential execution model. This is, only one execution thread deals with the whole transformation. However, model transformations dealing with very large models, such as those used in biology or aerospace applications, require concurrent solutions in order to speed up their performance. In this ongoing work we explore the use of Linda for implementing a set of basic mechanisms to enable concurrent model transformations, and present our initial results.

Key words: Model transformation, concurrency, Linda

1 Introduction

Model transformations (MTs) are at the heart of model-driven engineering (MDE), and provide mechanisms for manipulating and transforming models. MTs provide mechanisms to specify how output models are produced from input models. They can be classified according to different characteristics [1]: abstraction level of input and output models, type of language (declarative, imperative and hybrid), directionality, type of target model, etc.

As far as we are concerned, the engine of the transformation languages available in literature offer a sequential and non-distributed execution model. Only one execution thread deals with the whole transformation. Furthermore, the transformation can only be executed locally in a machine. This solution is inadequate for model transformations that deal with very large models, such as those used in biology or aerospace applications, because their performance is rather low. If transformation engines made use of concurrent execution models, different parts of a very big model, which are independent from each other, could be transformed at the same time by different execution threads. Besides, these independent parts could be distributed in different machines, all sharing a common space for building the output model. This would speed up the transformation process and would prevent from loading big models at once, which is normally very inefficient and a well-known scalability problem of current approaches [2].

Linda [3] is a model of coordination and communication among several parallel processes operating upon objects stored in and retrieved from shared, virtual and associative memory. By storing (input and output) models in an associative memory, we can apply a concurrent and distributed approach for transforming models. In this ongoing work we explore the use of a Linda-based language [4] for implementing a set of basic mechanisms to enable concurrent model transformations.

After this introduction, Section 2 presents our proposed initial approach. Then, in Section 3 we apply it on the *Families2Persons* case study, and present some performance results. Finally, Section 4 concludes the paper and outlines how we would like to continue this work.

2 Initial Approach

For implementing our approach, we have used the Java implementation of an in-memory data grid offered by Gigaspaces Technologies [5]. It is called *XAP Elastic Caching Edition* (we will refer to it from here on as XAP) and supports the basic Linda operations, such as *read* and *write*, as well as a wide range of new features, like fast data access, performance and scalability. Linda, and consequently XAP, allows several machines to work simultaneously over a tuple space that can be distributed in a user-transparent manner. Since Gigaspaces 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 on extraction.

The first things 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 Java classes. XAP requires these classes to implement the Java *Serializable* interface. Also, *getter* and *setter* methods have to be defined in order to have access to attributes afterwards. Our models, consequently, are composed of instances (objects) of these classes. Transformation languages such as ATL [6] or QVT [7] require metamodels to conform to Ecore [8], which are stored in *.ecore* files. Models conform to these metamodels and are typically stored in XML Metadata Interchange (XMI) files. Our java representation can be directly obtained from such models and metamodels by means of straightforward model-to-text transformations.

With XAP, we can define as many distributed tuple spaces as needed. In our approach, we have two of them, one for storing the input model and the other one for the output model. The global view of the approach is presented in Figure 1. Different execution threads, which may come from different machines, can be reading the input model from the first tuple spaces at the same time. Likewise, several threads can be creating the output model together. By having the model transformation implemented in several machines, we can make the most of the processing capacity of all of them because they can be creating different parts of the output model at the same time.

The model transformation is written in Java. In the implementation, we can specify how many execution threads will be concurrently dealing with the transformation. In this way, each thread retrieves a set of elements (objects) from the tuple space containing the input model and apply the transformation over them to create new objects and place them in the tuple space containing the output model. Consequently, the performance depends on the processing power. The more cores we have, the more threads can be working at the same time and the higher the performance will be. We also take into account the possible context switching among threads. This is, the fact that the number of threads overtakes the number of cores is counter-productive, because threads have to change from one core to another, and this affects the performance.

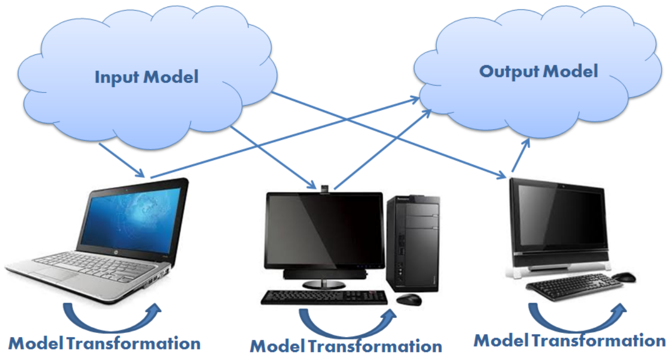

Fig. 1. Linda-based transformation schema

3 Case Study

The *Families2Persons* [9] example is rather small but sufficient for having an overview of the basics of our approach. Figure 2 shows the Ecore representation for the metamodels.

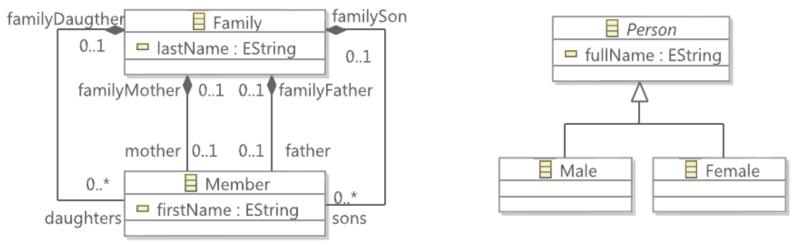

Fig. 2. Family and Person Ecore metamodels

The transformation aims to transform each element of type Member to an element of kind Person. In particular, to Female type if the Member is mother or daughter, and to Male if it is father or son. The code executed by each thread is as follows:

```
Family[] families = gigaSpaceSrc.takeMultiple(new Family(), numFamilies);

Person[] people = new Person[numMembers]; int k = 0;

for (int i = 0; i<families.length; i++){
    if (families[i].getMother() != null){
        people[k] = new Female(families[i].getMother().getName() +"␣"+ families[i
            ].getLastName()); k++;
    }
    if (families[i].getFather() != null){
        people[k] = new Male(families[i].getFather().getName() +"␣"+ families[i].
            getLastName()); k++;
    }
    for (Member m : families[i].getDaughters()){
```

```
13          people[k] = new Female(m.getName()+"␣"+families[i].getLastName()); k++;
14      }
15      for (Member m : families[i].getSons()){
16          people[k] = new Male(m.getName()+"␣"+families[i].getLastName()); k++;
17      }
18  }
19  gigaSpaceTrg.writeMultiple(people);
```

As we can see in line 1, we read a block of families from the input tuple space. Then we iterate over the retrieved families in order to transform (lines $5-18$) and finally store (line 19) the new objects in the data element people defined in line 3. Reading and writing from/in the tuple spaces can be a time-consuming task if they are realized over and over again by the same thread. For this reason, XAP provides the readMultiple and writeMultiple operations, where a set of elements are read and written at the same time. We have made use of these operations to improve the performance. The idea is that, if we have for example 1000 elements in the input tuple space and 10 threads, then each thread deals with the retrieval, transformation and storage of 100 elements (variable numFamilies in line 1).

The same transformation in ATL is as follows [9]:

```
module Families2Persons;
create OUT: Persons from IN: Families;

helper context Families!Member def: isFemale(): Boolean = ...
helper context Families!Member def: familyName: String = ...

rule Member2Male {
  from s: Families!Member (not s.isFemale())
  to t: Persons!Male (fullName <- s.firstName + ' ' + s.familyName )
}

rule Member2Female {
  from s: Families!Member (s.isFemale())
  to t: Persons!Female (fullName <- s.firstName + ' ' + s.familyName)
}
```

To compare the execution times between the ATL transformation and our Linda-based transformation, we have created several considerably big models with a variable number of families composed by 10 members each one. We executed both programs on a machine running Linux with 16 cores, launching that number of execution threads, where each thread deals with the same number of objects. Table 1 presents the results.

Table 1. Evaluation results

Number of families	ATL time	Linda time
1,000	0.246 s	0.222 s
10,000	4.112 s	2.162 s
100,000	Exception	9.06 s

We can appreciate that for small models which have around 10,000 members, the execution times are quite similar. For 100,000 members the execution time has been halved. Finally, when we tried to transform 1,000,000 members with the ATL transformation, we obtained 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.

4 Conclusions and Future Work

This paper presents an emergent 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 our approach with the execution times in ATL. Our approach has proved to be faster for this particular example, an does not present the scalability problems that appear in ATL. Nevertheless, this is only the beginning and there are several lines that we still have to explore.

In the first place, we plan to face the task of applying our approach on more complex transformations. For example, we want to implement transformations where internal traceability links would be needed, and transformations where several output models are created from one input model and viceversa. We also want to study how complex OCL expressions and constraints in transformations written in languages such as ATL or QVT would be expressed in our approach.

In the second place, we plan to provide a mechanism which transforms persistent metamodels and models from their original format into Java classes and Java instances. In the same way, after the transformation, the Java instances belonging to the output model will be stored in a file with the corresponding format.

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 some sequential transformation language, such as ATL, and our Linda-based representation, so that transformations written in the former could be executed concurrently.

Acknowledgements. This work is partially funded by Research Projects TIN2011-23795 and TIN2011-15497-E and by Universidad de Málaga (Campus de Excelencia Internacional Andalucía Tech).

References

1. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3) (2006) 621–645
2. Jouault, F., Sottet, J.S.: An AmmA/ATL Solution for the GraBaTs 2009 Reverse Engineering Case Study. In: Fifth International Workshop on Graph-Based Tools - Grabats 2009 (co-located with TOOLS 2009), Zurich, Suisse (2009)

3. Wells, G.: Coordination languages: Back to the future with linda. In: Proceedings of WCAT05. (2005) 87–98
4. Wells, G.C., Chalmers, A.G., Clayton, P.G.: Linda implementations in java for concurrent systems. Concurrency and Computation: Practice and Experience **16** (2003)
5. GigaSpaces Technologies Ltd. Gigaspaces: (2013) `http://www.gigaspaces.com/datagrid`.
6. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming **72**(1-2) (2008) 31–39
7. OMG: MOF QVT Final Adopted Specification. Object Management Group. (2005) OMG doc. ptc/05-11-01.
8. Budinsky, F., Merks, E., Steinberg, D.: EMF: Eclipse Modeling Framework (2nd Edition). Addison-Wesley Longman, Amsterdam (2006)
9. Eclipse: (2012) `http://wiki.eclipse.org/ATL/Tutorials\_-\_Create\_a\_simple\_ATL\_transformation`.