Parallel In-place Model Transformations with LinTra

Loli Burgueño[1], Javier Troya[2], Manuel Wimmer[2], and Antonio Vallecillo[1]

[1] Universidad de Málaga, Atenea Research Group, Málaga, Spain
{loli,av}@lcc.uma.es
[2] Vienna University of Technology, Business Informatics Group, Vienna, Austria
{troya,wimmer}@big.tuwien.ac.at

Abstract. As software systems have grown large and complex in the last few years, the problems with which Model-Driven Development has to cope have increased at the same pace. In particular, the need to improve the performance and scalability of model transformations has become a critical issue. In previous work we introduced LinTra, a model transformation platform for the parallel execution of out-place model transformations. Nevertheless, in-place model transformations are required in several contexts and domains as well. In this paper we discuss the fundamentals of in-place model transformations in the light of their parallel execution and provide LinTra with an in-place execution mode.

Keywords: In-place Model Transformations, Performance, Scalability, Linda

1 Introduction

Model transformations (MT) are gaining acceptance as model-driven techniques are becoming commonplace [1]. While models capture the views on systems for particular purposes and at given levels of abstraction, model transformations are in charge of the manipulation, analysis, synthesis, and refinement of the models [2].

So far the community has mainly focused on the correct implementation of a model transformation, according to its specification [3–8], although there is an emergent need to consider other (non-functional) aspects such as performance, scalability, usability, maintainability and so forth [9]. In particular, the study of the performance of model transformations is gaining interest as very large models living on the cloud have to be transformed as well. Consequently, new approaches to parallelize model transformations are starting to appear [10–13]. Following this path, in a previous work we introduced LinTra [11, 14], a model transformation engine, together with its implementation in Java called jLinTra. LinTra encapsulates all the concurrent mechanisms needed for the parallel execution of model transformations, and in particular users do not need to worry about threads and their synchronization. The engine is based on Linda [15], a mature coordination language for parallel processes.

So far, LinTra only permitted out-place model transformations. In this kind of transformations, input and output models often conform to different metamodels and output models are created from scratch. However, there are many situations in which we need to evolve models, instead of creating them anew. For instance, when migrating and modernizing software using model-driven engineering (MDE) approaches [16, 17], (i)

software is reverse-engineered to obtain a model representation of the system, (ii) modernization patterns are applied on the model level, and (iii) the modernized model is translated back into code. Modernization at model level is typically achieved using in-place model transformations, where the initial model is evolved until the final target model is obtained. Models which are reverse-engineered from large systems may be huge, thus high-performing in-place transformation engines are needed. For this reason, in this paper we extend our LinTra language with an in-place semantics.

This paper is structured as follows. Section 2 shortly introduces LinTra and our reference non-recursive in-place semantics. Section 3 shows how LinTra realizes its in-place semantics, while Section 4 illustrates the benefits of parallel in-place transformations. Finally, Section 5 discusses related work before we conclude the paper in Section 6 with an outlook on future work.

2 Background

Previous Work on LinTra. LinTra [11, 14] is a parallel model transformation engine that encapsulates all the concurrent mechanisms, so that users do not need to worry about threads and their synchronization. It uses the Blackboard paradigm [18] to store the input and output models, as well as the required data to keep track of the MT execution that coordinates the agents that are involved in the transformation process. One of the keys of LinTra is the model and metamodel representation, where we assume that entities in the model are independent from each other. Thus, every entity is assigned an identifier, which is used to reference objects and to represent relationships between them. Relationships between entities are represented by storing in the source entity the identifier(s) of its target entity(ies).

In out-place model transformations, there might be dependencies between rules because the element(s) created in a rule are needed to initialize some properties of the elements created by other rules. In LinTra, traceability is implemented implicitly using a bidirectional function that receives as a parameter the entity identifier of the input model and returns the identifier of the output entity, regardless whether the output entities have already been created or not. This means that LinTra does not store information about the traces explicitly; thus, the performance is not affected by the access to memory and the search for trace information.

In order to carry out the transformation process in parallel, LinTra uses the Master-Slave design pattern [18]. The master's job is to launch slaves and coordinate their work. Each slave is in charge of applying the transformation to submodels of the input model (i.e., partitions) as if each partition is a complete and independent model. Since in LinTra's out-place mode the complete input model is always available, in case the slaves have data dependencies with elements that are not in the submodels they were assigned, they only have to query the Blackboard to retrieve them.

Non-recursive In-place Transformations. In-place transformations specify how the input model evolves to obtain the output one, i.e., how the input model has to change. There are two kinds of in-place model transformation strategies, non-recursive and recursive, depending on whether recursive matching takes place or not. By *recursive matching* we understand that the matches of rules are not solely computed based

on the initial input model but on the current model state which probably has been modified by previous application of rules. This is the typical strategy followed in graph or rewriting systems, where a set of rules modifies the state of a configuration of objects (representing the model) one-by-one. Thus, after the application of each rule, the state of the system is changed, and subsequent rules will be applied on the system on this new state. Therefore, the transformation navigates the target model, which is continuously updated by every executed rule.

Regarding *non-recursive matching*, it shares some characteristics with out-place transformations. In this strategy, there is one input model which is used to directly compute the output model without considering intermediate steps. This is the reason why we chose to follow a non-recursive approach for the LinTra in-place mode. Our decision was also inspired by the ATL *refining mode* [19, 20], used to implement in-place transformations. ATL supports both out-place and in-place modes. In both execution modes, source models are read-only and target models are write-only. This is an important detail that significantly affects the way in which ATL works in refining mode. Indeed, ATL in-place mode does not execute transformations as these are executed in graph or rewriting systems, as explained in detail in [21]. Thus, we follow as well non-recursive matching in LinTra where rules always read (i.e., navigate) the state of the source model, which remains unchanged during all the transformation execution.

3 In-place Model Transformations with LinTra

LinTra follows a non-recursive approach for executing in-place transformations, as the ATL refining mode does. In this section we discuss some semantic issues that might occur in rule-based in-place model transformations in general as they are indeed highly relevant for the parallel execution of in-place transformations.

3.1 Atomic Transformation Actions

When executing a non-recursive in-place transformation, the first decision concerns the elements for which the transformation does not specify what to do. We could either decide to exclude them from the target model or to include them as they are. In jLinTra we decided for the second option, which implies that if we want to exclude objects in the target model, the transformation will have to explicitly remove them. Thus, after the input model is loaded, and once the transformation phase starts, an initialization phase is needed where the identity transformation is applied so that the target area contains a copy of the input model.

After the model is copied, in the following we explain the three operations that may be applied to it: deletion of elements, creation of new elements, and modification of existing elements.

Elements Deletion. When an element is deleted, the outgoing relationships from such element to others are deleted too, since such information is stored as attributes in the deleted element. However, the situation is different when the deleted element has incoming relationships. In such case, the information about relationships to the deleted element is stored in the attributes of other elements. In this case, we can distinguish

two different semantics. Either all the incoming relationships are deleted, for which the engine needs to traverse the whole model searching for relationships pointing to the deleted element, or they are not deleted, causing dangling references and, consequently, an inconsistent model. In the former option, we need to keep track of all the deleted elements, so that the traversal is realized only once as the last step of the transformation. The latter option is useful in order to make the user aware that he/she is removing an element by mistake. LinTra permits both behaviors, since it is aimed at offering a flexible implementation.

Elements Creation. If the developer wants to create a new element, he/she has to create the instance and set its attributes and relationships. In case of bidirectional relationships, there are two alternatives: *(i)* the opposite reference is created automatically, or *(ii)* the creation of the opposite relationship must be explicitly specified by the developer. We permit both behaviors.

Elements Updates. Updating an attribute or an ongoing unidirectional relationship of an element is trivial, since the transformation only has to change the corresponding attribute of the updated element. However, there are again two choices when updating a relationship which is bidirectional, since the previous target element of the relationship would still have a relationship to the updated element unless something is done. Thus, *(i)* the relationship from the previous target element should be automatically removed and a new relationship from the new pointed element to the updated element should be automatically created, or *(ii)* the developer has to specify explicitly in the transformation that the corresponding relationships are removed and created respectively. Again, we permit both alternative behaviors.

3.2 Confluence conflicts

Confluence conflicts typically occur when two rules are applied to the same part of the model and they treat it differently [22]. Thus, the resulting model may vary depending on the order in which those rules are applied. The application of a rule can conflict with the application of another rule in four different ways. Let us explain them for the ATL refining mode which acts as blueprint for the LinTra in-place transformation strategy. For the explanations, let us imagine a transformation for reverse engineering Java code.

Update/Update. Imagine that a rule sets the public variables to private and capitalizes the name of the ones that are private. This case is not a problem for the confluence of non-recursive in-place transformations since only the source model provided by the user is read—the changes done by the rule that changes the visibility are not visible to the rule that capitalizes the names of the variables. On the contrary, if a rule sets the visibility of the variables to private and another rule sets them to public, the transformation may not be confluent. A possible way to prevent this situation is to force the precondition of the rules to be exclusive, which leads to non-overlapping matches. This was the solution adopted by ATL concerning the declarative part. Nevertheless, it is easy to fool ATL by using the imperative part, which is executed after the declarative part of the rule.

Delete/Update. Suppose that a rule sets the visibility of the variables to private and another rule removes all the variables. The situation is similar to the second case we presented for the conflict *Update/Update*. The two rules are a conflicting pair, thus

the language should prevent this situation from happening or should establish the behaviour of the transformation. Again, it is possible to produce this case in ATL by using the imperative part to set the visibility and writing a declarative rule that removes the variables. Both rules are executed so that the visibility is changed and the variables are removed. As a result, the variables are not present in the resulting model. Apparently, the objects are removed in a later execution phase, after having done all the updates and creations specified in the declarative and imperative parts.

Produce/Forbid. Imagine that a rule adds a variable to a class and another rule removes all the empty classes (classes with no variables) from the model. The first rule is producing an additional structure that is forbidden by the precondition of the second rule. Once again, the order in which the rules are executed influences the result. This time, if we try to implement this transformation with ATL using the imperative part of a rule to add the variables and a declarative rule to remove the empty classes, both rules are applied but the transformation does not fulfil the purpose for which it was written (since only the source model is read). As a result, the classes are removed but the newly created variables remain in the model without any container.

Delete/Use. This conflict appears when a rule deletes elements that produce a match with another rule. Thus, it is the opposite case to *Produce/Forbid*. Depending on the order in which the rules are executed, the transformation is able to execute a higher or lower number of rules.

We have illustrated the conflicts that may appear between rules and how ATL tries to solve them using non-overlapping matches, how they can be avoided or produced, and which is the final result of the execution. Enforcing to have non-overlapping rules is not the only solution; another possibility is to statically detect the conflicting rules using the critical pair analysis approach [23], and subsequently, to deal with the conflicts making use of layers which is also implicitly done in ATL by using different phases in the transformation execution.

As jLinTra is realized as an internal transformation language embedded in Java, we have opted for not imposing any restriction. Thus, our solution is completely flexible with respect to rule executions. The idea is that high-level model transformation languages (such as ATL [24], ETL [25], or QVTO [26]) are compiled to an extended set of the primitives [27] that automatically compile to jLinTra. In these primitives the rule scheduling is already given (the layers and the rules that belong to each layer). Therefore, in case that the critical pair analysis is needed, it can be done statically during the compilation process from the high-level model transformation language to the LinTra primitives.

4 Evaluation

To evaluate our approach we performed an experimental study concerning a transformation which, in reverse engineered Java applications, removes all the comments, changes the attributes from public to private and creates the getters and setters.

4.1 Research Questions

The study was performed to quantitatively assess the quality of our approach by measuring the runtime performance of the transformations. We aimed to answer the following research questions (RQs):

1. *RQ1—Parallel vs. sequential in-place transformations*: Is the parallel execution of in-place transformations faster in terms of execution times compared to using the state-of-the-art sequential execution engines? And if there is a positive impact, what is the speedup with respect to the used number of cores for the parallel transformation executions?
2. *RQ2—Parallel in-place vs. parallel out-place transformations*: Is the parallel execution of in-place transformations faster in terms of execution time compared to using their equivalent out-place transformations?

4.2 Experiment Setup

To evaluate our approach, we use an experiment in which Java models are obtained from Java code using MoDISCO [17]. The Java metamodel has a total of 125 classes from which 15 are abstract, 166 relationships among them and 5 enumeration types. As source model we have selected the complete Eclipse project, containing $4,357,774$ entities. In order to assess how the transformation scales up with this kind of input, we generated 11 smaller sample source models (with subsets of the Eclipse project) ranging from $100,000$ elements to the complete model.

We apply an extended version of the Public2Private transformation – the original one is available in the ATL Zoo[3] – that changes the visibility of every public variable to private and creates the corresponding getter and setter methods. In addition, the transformation also removes all the comments contained in the code. All artifacts can be downloaded from our website[4].

Let us show the effects of this transformation with an example. Listing 1.1 shows the Java code that declares a class called *MyClass*, a public attribute *name* and the class's constructor. The code contains some comments too. After applying the transformation, the Java code that the model represents should look like the fragment presented in Listing 1.2.

Listing 1.1. Code to be refactored

```
1 public class MyClass {
2     //Declaration of variable called name
3     public String name; /* This variable contains the name */
4     public MyClass() { /* Description @param ... */  ... }
5 }
```

Listing 1.2. Refactored code

```
1 public class MyClass {
2     private String name;
3     public String getName() { return name; }
4     public void setName(String name) { this.name = name; }
5     public MyClass(){ ... }
6 }
```

[3] http://www.eclipse.org/atl/atlTransformations/

[4] http://atenea.lcc.uma.es/index.php/Main_Page/Resources/LinTra

An excerpt of the code corresponding to the rules in jLinTra is shown in Listing 1.3. As stated in Section 2, every slave is in charge of transforming a chunk of the model. For efficiency reasons the changes are made permanent once the whole chunk has been transformed. In order to keep the temporary changes the structures `deletedElems`, `modifiedElems` and `createdElems` (lines 2, 8 and 9) are needed.

Listing 1.3. jLinTra transformation

```
1  if (ie instanceof Comment){
2      deletedElems.add(ie); //Delete Comment
3  } else if (ie instanceof FieldDeclaration){
4      String modId = ((FieldDeclaration) ie).getModifier();
5      Modifier mod = (Modifier) srcArea.read(modId);
6      String visibility = mod.getVisibility();
7      if (visibility.equals(PUBLIC)){
8          mod.setVisibility(PRIVATE); modifiedElems.add(mod); // Modify visibility
9          ... createdElems.add(...); // Create getters and setters }
10 }
```

We have run all our experiments on a machine whose operating system is Ubuntu 12.04 64 bits with 11.7 Gb of RAM and 2 processors with 4 hyperthreaded cores (8 threads) of 2.67GHz each. We discuss the results obtained for the different transformations after executing each one 10 times for every input model and having discarded the first 5 executions as the VM has a warm-up phase where the results might not be optimal. The Eclipse version is Luna. The Java version is 8, where the JVM memory has been increased with the parameter *-Xmx11000m* in order to be able to allocate larger models in memory.

4.3 Performance Experiments

The in-place transformation described before has been implemented and executed in jLinTra and in ATL, for which we have used the EMFTVM[5]. We have also developed an out-place transformation version in jLinTra in order to compare its performance with the proposed in-place version. Table 1 shows in its left-most column the number of entities of the source models of the transformation. The second, third, and fourth columns correspond to the execution times (in seconds) obtained for ATL and jLinTra (using the in-place and out-place modes), respectively. Note that we have only taken into account the time of the execution of the transformation, meaning that we do not consider the time used for loading the models into memory, nor the time used to serialize them to the disk. The fifth column presents the speedup of jLinTra with respect to ATL. We can see that the speedup is not constant: it grows with the size of the model, reaching a value of 955.23 for the complete model, meaning that value that jLinTra is 955.23 times faster than ATL for this concrete case. Finally, column six shows the speedup of the in-place and out-place modes of LinTra, where we can see that the in-place model transformation is on average 1.81 times faster than its out-place version.

We already mentioned in Section 3 that an initialization phase where the input model is copied to the target area is needed. However, if we moved that process to the loading phase so that both the source and target areas were loaded at the same time, we would only pay a minimum price (an overhead of 5% in the loading phase) and the

[5] `https://wiki.eclipse.org/ATL/EMFTVM`

	ATL	LinTra		*Speedups*	
No. elements	**EMFTVM**	**In-place (LI)**	**Out-place (LO)**	*LI–EMFTVM*	*LI–LO*
0.1$\times 10^6$	2.40	0.11	0.19	21.23	1.72
0.2$\times 10^6$	12.04	0.29	0.36	41.88	1.25
0.5$\times 10^6$	65.06	0.73	0.98	89.06	1.34
1.0$\times 10^6$	371.41	1.29	2.38	287.34	1.84
1.5$\times 10^6$	1042.41	2.06	2.61	506.71	1.27
2.0$\times 10^6$	2030.82	2.99	5.63	678.16	1.88
2.5$\times 10^6$	2952.46	3.92	9.64	754.14	2.46
3.0$\times 10^6$	4156.69	5.13	8.82	809.92	1.72
3.5$\times 10^6$	5527.96	6.26	13.77	883.37	2.20
4.0$\times 10^6$	6737.97	7.57	15.20	890.70	2.01
Complete	7238.70	7.58	17.18	955.23	2.27

Table 1. Execution results and speedups.

performance in the transformation phase would be improved reaching a speedup of 3.89 w.r.t. the out-place mode and speedup of $1,195$ w.r.t. ATL.

Regarding the gain of in-place MTs in LinTra w.r.t. the number of cores involved in the transformation, the speedups of using only one core w.r.t. using four, eight, twelve and sixteen are 1.19, 1.62, 1.97, 3.24, respectively.

We also planned to execute and compare this transformation with the original ATL virtual machine. However, although it supports the refining mode it does not support the imperative block, which is applied in the particular transformation used in this study.

Regarding the out-place transformation developed in LinTra, it explicitly specifies that all elements that are not modified must be copied, together with their properties. The out-place transformation counts on $3,302$ lines of Java code (we generated the code for the identity transformation using Xtend [6] and adapted the corresponding code to fit the needs of the Public2Private transformation), while the in-place transformation has only 194 lines.

For answering the two RQs stated above, we can first conclude that the parallel execution of in-place transformations reduces the execution time compared to using sequential execution and that the execution time can be further improved by adding more cores. Second, for typical in-place transformation problems, in-place transformation implementations are more efficiently executed than their equivalent out-place transformations using parallelization in both executions.

4.4 Threats to Validity

In this section, we elaborate on several factors that may jeopardize the validity of our results.

Internal validity – Are there factors which might affect the results of this experiment? The performance measures we have obtained directly relate to the experiment we have used for the evaluation. Therefore, if we had used different experiments other than the Public2Private transformation then the speedups between the executions of the different implementations would have probably been different. Besides, we have generated 11 smaller sample source models. Should we have generated different models,

[6] `https://eclipse.org/xtend/`

the results in Table 1 would also be different. As another threat, we have decided to use the executions after the 5th one in order to avoid the possible influence of the VM warming-up phase. However, if after the 5th execution the VM has not finished warming up, our results are then influenced. Finally, we are quite confident that we have correctly written the equivalent transformation in ATL due to our expertise with such language. Nevertheless, there may exist tiny differences which may have an influence on the execution times.

External validity – To what extent is it possible to generalize the findings? As a proof of concept of our approach, we have compared the execution times of our approach with the ATL implementation executed with the EMFTVM engine. We have chosen ATL for the comparison study because we have enriched LinTra with the same in-place semantics that ATL has. Therefore, since our study only compares LinTra and ATL, our results cannot be generalized for all non-recursive engines.

5 Related Work

In this paper, we focus on in-place model transformations running in batch mode. However, to deal with large models, orthogonal techniques may be applied as well. Especially, two scenarios have been discussed in the past in the context of speeding-up model transformation executions. 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 [28–30] have been introduced, while for the latter *lazy* transformations [31] have been proposed.

An important line of research for executing transformations in parallel is the work on critical pair analysis [22] from the field of graph transformations as discussed in Section 3. 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 is statically computed. However, most existing execution engines follow a pseudo-parallel execution of the rules, but there are already some approaches emerging which consider the execution of graph transformations in a recursive way on top of multi-core platforms [12, 13, 32]. A closer comparison concerning the commonalities and differences of recursive and non-recursive in-place semantics concerning parallelism is considered as subject for future work.

The performance of model transformations is considered as an integral research challenge in MDE [33]. For instance, Amstel et al. [9] considered the runtime performance of transformations written in ATL and in QVT. In [34], 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. One line of work we are aware of dealing with the parallel execution of ATL transformations is [35], where Clasen et al. outlined several research challenges when transforming models in

the cloud. A follow-up work is presented in Tisi et al. [36] where a parallel transformation engine for ATL is presented. However, they only consider out-place transformations while we tackled the parallel execution of in-place transformations.

6 Conclusion and Future Work

We have presented an extension for LinTra that allows the parallel execution of in-place model transformations. We have shown with experiments that the performance is improved w.r.t. other in-place MT engines and that in cases where in-place transformations can be achieved also by means of out-place transformations, the in-place transformations provide better performance and usability.

There are several lines of future work. First, we plan to provide a new in-place execution mode that supports recursive matchings. Second, we plan to extend our set of primitives so that in-place transformations written in any MT language may be compiled to and executed with LinTra.

Acknowledgments This work is funded by the Spanish Research Projects TIN2011-23795 and TIN2014-52034-R, by Universidad de Malaga (Campus de Excelencia Internacional Andalucia Tech), by the European Commission under ICT Policy Support Programme (grant no. 317859), and by the Christian Doppler Forschungsgesellschaft and the BMWFW, Austria.

References

1. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers (2012)
2. Lucio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G., Syriani, E., Wimmer, M.: Model transformation intents and their properties. SoSyM (2014) 1–38
3. Gogolla, M., Vallecillo, A.: *Tract*able Model Transformation Testing. In: Proc. of ECMFA. (2011) 221–235
4. Amrani, M., Lúcio, L., Selim, G., Combemale, B., Dingel, J., Vangheluwe, H., Traon, Y.L., Cordy, J.R.: A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In: Proc. of VOLT Workshop @ ICST. (2012) 921–928
5. Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., Hamann, L.: Formal Specification and Testing of Model Transformations. In: Proc. of SFM. (2012) 399–437
6. Sánchez Cuadrado, J., Guerra, E., de Lara, J.: Uncovering errors in ATL model transformations using static analysis and constraint solving. In: Proc. of ISSRE. (2014) 34–44
7. Wimmer, M., Burgueño, L.: Testing M2T/T2M Transformations. In: Proc. of MODELS. (2013) 203–219
8. Burgueno, L., Troya, J., Wimmer, M., Vallecillo, A.: Static Fault Localization in Model Transformations. IEEE Transactions on Software Engineering **41**(5) (2015) 490–506
9. van Amstel, M., Bosems, S., Kurtev, I., Pires, L.F.: Performance in Model Transformations: Experiments with ATL and QVT. In: Proc. of ICMT. (2011) 198–212
10. Tisi, M., Perez, S.M., Choura, H.: Parallel Execution of ATL Transformation Rules. In: Proc. of MODELS. (2013) 656–672
11. Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A.: On the Concurrent Execution of Model Transformations with Linda. In: Proc. of BigMDE Workshop @ STAF. (2013) 3:1–3:10

12. Imre, G., Mezei, G.: Parallel Graph Transformations on Multicore Systems. In: Proc. of MSEPT. (2012) 86–89
13. Krause, C., Tichy, M., Giese, H.: Implementing Graph Transformations in the Bulk Synchronous Parallel Model. In: Proc. of FASE. (2014) 325–339
14. Burgueño, L.: Concurrent Model Transformations based on Linda. In: Doctoral Symposium @ MODELS. (2013) 9–16
15. Gelernter, D., Carriero, N.: Coordination Languages and Their Significance. Commun. ACM **35**(2) (1992) 97–107
16. Bergmayr, A., Brunelière, H., Canovas Izquierdo, J., Gorronogoitia, J., Kousiouris, G., Kyriazis, D., Langer, P., Menychtas, A., Orue-Echevarria, L., Pezuela, C., Wimmer, M.: Migrating Legacy Software to the Cloud with ARTIST. In: Proc. of CSMR. (2013) 465–468
17. Brunelière, H., Cabot, J., Dupe, G., Madiot, F.: MoDisco: A model driven reverse engineering framework. Information and Software Technology **56**(8) (2014) 1012–1032
18. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. Wiley (1996)
19. Tisi, M., Martínez, S., Jouault, F., Cabot, J.: Refining Models with Rule-based Model Transformations. Research Report RR-7582 (2011)
20. Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a General Composition Semantics for Rule-Based Model Transformation. In: Proc. of MODELS. (2011) 623–637
21. Troya, J., Vallecillo, A.: A Rewriting Logic Semantics for ATL. JOT **10** (2011) 5:1–29
22. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of Typed Attributed Graph Transformation Systems. In: Proc. of ICGT. (2002) 161–176
23. Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. Electr. Notes Theor. Comput. Sci. **127**(3) (2005) 113–128
24. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming **72**(1-2) (2008) 31–39
25. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Transformation Language. In: Proc. of ICMT. (2008) 46–60
26. OMG: MOF QVT Final Adopted Specification. Object Management Group. (2005)
27. Burgueño, L., Syriani, E., Wimmer, M., Gray, J., Vallecillo, A.: LinTraP: Primitive Operators for the Execution of Model Transformations with LinTra. In: Proc. of BigMDE Workshop @ STAF. (2014) 23–30
28. Jouault, F., Tisi, M.: Towards Incremental Execution of ATL Transformations. In: Proc. of ICMT. (2010) 123–137
29. Razavi, A., Kontogiannis, K.: Partial evaluation of model transformations. In: Proc. of ICSE. (2012) 562–572
30. Szárnyas, G., Izsó, B., Ráth, I., Harmath, D., Bergmann, G., Varró, D.: IncQuery-D: A Distributed Incremental Model Query Framework in the Cloud. In: Proc. of MODELS. (2014) 653–669
31. Tisi, M., Perez, S.M., Jouault, F., Cabot, J.: Lazy Execution of Model-to-Model Transformations. In: Proc. of MODELS. (2011) 32–46
32. Bergmann, G., Ráth, I., Varró, D.: Parallelization of Graph Transformation Based on Incremental Pattern Matching. ECEASST **18** (2009) 1–15
33. Kolovos, D.S., Rose, L.M., Matragkas, N., Paige, R.F., Guerra, E., Cuadrado, J.S., De Lara, J., Ráth, I., Varró, D., Tisi, M., Cabot, J.: A Research Roadmap Towards Achieving Scalability in Model Driven Engineering. In: Proc. of BigMDE Workshop @ STAF. (2013) 2:1–2:10
34. Wimmer, M., Martínez, S., Jouault, F., Cabot, J.: A Catalogue of Refactorings for Model-to-Model Transformations. JOT **11**(2) (2012) 2:1–40
35. Clasen, C., Didonet Del Fabro, M., Tisi, M.: Transforming Very Large Models in the Cloud: a Research Roadmap. In: Proc. of CloudMDE Workshop @ ECMFA. (2012)

36. Tisi, M., Perez, S.M., Choura, H.: Parallel Execution of ATL Transformation Rules. In: MODELS. (2013) 656–672