Primitive Operators for the Concurrent Execution of Model Transformations Based on LinTra

Loli Burgueño[1], Eugene Syriani[2], Manuel Wimmer[3], Jeff Gray[2], and Antonio Vallecillo[1]

[1] Universidad de Málaga, GISUM/Atenea Research Group, Málaga, Spain
{loli,av}@lcc.uma.es
[2] University of Alabama, Department of Computer Science, Tuscaloosa AL, U.S.A.
{esyriani,gray}@cs.ua.edu
[3] Vienna University of Technology, Business Informatics Group, Vienna, Austria
wimmer@big.tuwien.ac.at

Abstract. Performance and scalability of model transformations are becoming prominent topics in Model-Driven Engineering. In previous work, we introduced LinTra, a platform for executing out-place model transformations in parallel. LinTra is based on the Linda coordination language for archiving concurrency and distribution and is intended to be used as a middleware where high-level model transformation languages (such as ATL and QVT) are compiled. To define modularly the compilation, this paper presents a minimal, yet sufficient, collection of primitive operators that can be composed to (re-)construct any out-place, unidirectional model transformation language (MTL). These primitives enable any MTL to be executed in parallel in a transparent way, without altering the original transformation.

Keywords: Model Transformation, Primitives, LinTra

1 Introduction

Model-Driven Engineering is a relatively new paradigm that has grown in popularity in the last decade. Although there are a wide variety of approaches and languages with different characteristics and oriented to different types of model transformations (MT), most of the model transformation engines are based on sequential and local execution strategies. Thus, they have limited capabilities to transform very large models and even less capability to do it in a reasonable amount of time.

In previous works [1,2], we investigated concurrency and distribution for out-place transformations to increase their performance and scalability. Our approach, LinTra, is based on Linda [5], a mature coordination language for parallel processes. Linda supports reading and writing data in parallel into distributed tuple spaces. A tuple space follows the Blackboard architecture [3], which makes the data distributed among different machines transparent to the user.

To execute transformations on the LinTra architecture, LinTra specifies how to represent models and metamodels, how the trace links are encoded for efficient retrieval, which agents are involved in the execution of the MT and their role, and how the transformation is distributed over the set of machines composing the cluster where the MT is executed. The implementation of several case studies using the Java implementation of LinTra (jLinTra) is available on our website [4], together with the performance comparison with several well-known model transformation languages (MTLs) such as ATL [6], QVT-O [8] and RubyTL [4].

The goal of this paper is to introduce a collection of minimal, yet sufficient, primitive operators that can be composed to (re-)construct any out-place and unidirectional MTL. These primitive operators encapsulate the LinTra implementation code in order to hide the underlying architecture that makes the parallel and distributed execution possible. This collection of primitives serves as an abstraction of the implementation details of the general-purpose language in which LinTra is implemented.

Section 3 introduces the collection of primitives. Section 4 illustrates examples of primitive combinations in order to write MTs. Finally, Section 5 presents our conclusions, the work that motivates our approach and an outlook on future work.

2 Background on LinTra

LinTra uses the Blackboard paradigm [3] 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 process.

One of the keys of our approach is the model and metamodel representation. In this representation, we assume that every entity in the model is independent from another. Each entity is assigned an identifier. Relationships between entities are represented by storing in the source entity the identifier of its target entity.

Traceability is frequently needed when executing an out-place model transformation because the creation of an element might require information about some other elements previously transformed, or even information about elements that will be transformed in the future. This means that there might be dependencies that can affect the execution performance, *e.g.*, when one element needs access to an element that has not been created yet. In LinTra, traceability is implemented implicitly using a bidirectional function that receives as parameter the entity identifier (or all the entity identifiers in the case that the match comprises more than one entity) of the input model and returns the identifier of the output entity(ies), regardless whether the output entities have already been created or not.

Together with the Blackboard, LinTra uses the Master-Slave design pattern [3] to run MTs. The master's job is to launch slaves and coordinate their work. Slaves are in charge of applying the transformation in parallel to submodels of the input model (partition) as if each partition is a complete and independent model. Since LinTra only deals with out-place transformations, the complete input model is always available. Thus, if the slaves have data dependencies with elements that are not in the submodels they were assigned, they only have to query the Blackboard to get them.

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

3 Collection of Primitives

3.1 Primitives

Primitive for the Concurrent Platform. LinTra requires the user to specify how the input model is partitioned. The `PartitionCreator` primitive receives the input model, an OCL expression, *OE*, and the maximum number of model entities, *S*, that each partition will contain. The `PartitionCreator` queries the input model using *OE* and partitions the resulting submodel into partitions of size *S*. The combination of `PartitionCreators` with different OCL expressions may lead to overlapping partitions; thus, the LinTra engine checks internally that the intersection of all the partitions is the empty set and the union is the whole model.

Primitives for the Model Transformation Language. The minimum set of primitive constructs needed to define out-place model transformations are: `Composer`, `Tracer`, `EntityCreator`, `CondChecker` and `Finder`.

`Composer` is a primitive that allows the grouping of a combination of primitives and assign the combination a name. Its syntax is *Composer <composerName>* { *<combination of primitives>* } and it is mainly used by the `Tracer`.

The `Tracer` provides access to the trace model needed by out-place MT engines for linking the entities in the output model. Given an input entity or set of entities that match the pre-condition of a rule, the traces give access to the entities that were created in the post-condition, and vice versa. In this case, to identify which primitive belongs to which rule, we propose to encapsulate them in a `Composer` so that the `Tracer` receives as parameter the name of the Composer and the set of entities from the pre or post-condition and give the reference to the other entities. Its signature is *Tracer(composer : Composer, e : Entity) : Collection(Entity)* and *Tracer(composer : Composer, e : Collection(Entity)) : Collection(Entity)*

`EntityCreator` creates an entity given its type and its features (attributes and bindings) and writes it in the Blackboard. The primitive receives as parameter the entity type and a dictionary which stores the values of every feature. Its syntax is *EntityCreator(type : Factory, features : Dictionary<feature, value>).*

`CondChecker` allows the querying of the Blackboard with an OCL expression that evaluates to a boolean value. It receives as input the OCL expression, queries the Blackboard and returns the result. Its signature is *CondChecker(expr : OCLExpression) : Boolean*.

`Finder` allows the retrieval of elements from the Blackboard that satisfy a constraint. It receives as parameter an OCL expression and returns the set of entities (submodel) that fulfils the OCL expression. Its signature is *Finder(expr : OCLExpression) : Collection(Entity).*

3.2 Integrating the primitives with the LinTra engine

When executing a transformation with LinTra there are several steps. Some of the steps are done automatically by the engine and others require that the user gives certain guidelines on how to proceed by means of the primitives. Two different phases can be distinguished: the setup and the MT itself.

The semantics of some MTs might require that a certain set of rules are applied to the whole input model before applying or after having applied some others. This is the case, for example, of top rules in QVT-R [8], and entrypoint and endpoint rules in ATL [6]. In order to be able to express this behaviour, in the setup phase, the rule schedule must be extracted from the transformation given by the user and a collection of collection of rules (collection of rule layers) must be created. All the rules belonging to the same layer can be executed in parallel, but all rules in one layer must have terminated before rules in a subsequent layer can begin.

Furthermore, during the setup, the transformation written in a high-level MTL is compiled to the MTL primitives, and the input model is parsed to the tuple space representation and stored into the Blackboard. Then, the `PartitionCreator` provided by the user is executed and the model partitions are created. Finally, the tasks to be executed by the slaves are created and stored in order in the Blackboard. A task is a pair consisting of a rule layer and a model partition. The tasks are produced by computing all the possible combinations between the partitions and the rule layers.

Once the setup phase is finished, the LinTra MT starts using the Master-Slave design pattern. The master creates slaves that execute the tasks that share the same rule layer and waits for all the tasks to be finished before starting to execute the ones that involve the following layer. Every slave executes the assigned task sequentially and all the slaves work in parallel. The master behaviour after launching the slaves is given by the pseudo-code presented in Listing 1.1.

Listing 1.1. Master.

```
rule_layer := 1
while (∃ task ∈ Blackboard.Tasks){
    createSlaves()
    while (∃ slave : slave.available and
           ∃ task ∈ Blackboard.Tasks : task.ruleLayer = rule_layer) {
        assign(slave, task) }
    join() -- wait for all the slaves to finish
    rule_layer := rule_layer+1 }
```

When a slave receives a task, it transforms the submodel given by its partition with the rules given by its rule layer. These rules are a collection of MT primitives. The code executed by the slaves is shown in Listing 1.2. An overview of how the system works can be seen in Figure 1.

Listing 1.2. Slave.

```
for each e ∈ task.partition { task.ruleLayer.transforms(e) }
```

4 Examples

Consider the metamodels for the Class-to-Relational case study described on our website [5] and a very simple transformation that from every Attribute creates a Column with the same name. From every non-abstract class creates a Table that is associated with the Columns that were created from its Attributes, where the name is the same as the Class name. Finally, and after having applied the previous rules, from every possible

[5] http://atenea.lcc.uma.es/index.php/Main_Page/Resources/MTBenchmark/Class2Relational

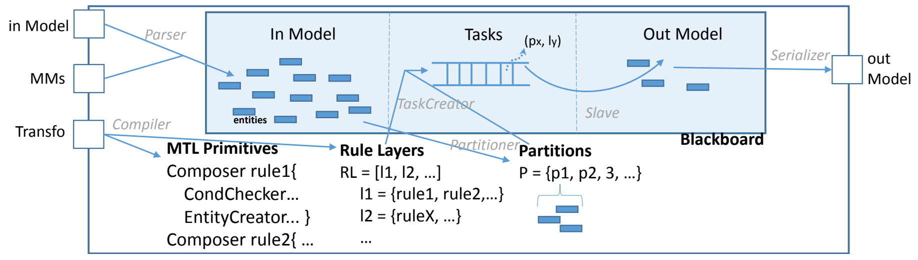

Fig. 1. Overview on the MT execution.

combination between Methods and Parameters, a Column must be created which name is the concatenation of the names of the elements from which it is created.

Let us assume that the user does not specify how the entities are assigned to the different partitions and the partition size is 100. The partition creator is invoked as *PartitionCreator(inModel, Entity.allInstances, 100)*. Let us suppose that it returns three partitions, $P = \{p1, p2, p3\}$. From the MT, the rule schedule is extracted and the rule layers are created. Given the MT definition, two different rule layers are created: $RL = [l1, l2]$ where *l1* contains the first two rules (class2Table and Att2Column) and *l2* the rule that is executed after the other two ruled have finished (MethParam2Column). Given the partitions and the layers, the tasks to be executed are $T = [T1, T2]$, where $T1 = \{(p1, l1), (p2, l1), (p3, l1)\}$ and $T2 = \{(p1, l2), (p2, l2), (p3, l2)\}$. We make the distinction between *T1* and *T2* to clarify that all tasks in *T1* are relative to *l1* and all tasks in *l2* to *T2*; thus, until all tasks from *T1* have been executed, tasks from *T2* cannot start. The compilation process from the high-level MT to the primitives produces the code shown in Listing 1.3

Listing 1.3. MTL primitives.

```
 1 Composer ruleClass2Table { -- RuleLayer 1
 2   if (CondChecker(e.oclIsTypeOf(Class) and not e.isAbstract))
 3     EntityCreator(Table, {[name, e.name], [col, Tracer(ruleAtt2Column, e.att)]}) }
 4 Composer ruleAtt2Column {
 5   if (CondChecker(e.oclIsTypeOf(Attribute))
 6     EntityCreator(Column, {[name, e.name]}) }
 7 Composer ruleMethParam2Column { -- RuleLayer 2
 8     if (CondChecker(e.oclIsTypeOf(Parameter))
 9         meths := Finder(Method.allInstances)
10         for each (m : Method in meths)
11             EntityCreator(Column, {[name, e.name + '-' + m.name]}) }
```

The code for the first rule layer goes from line 1 to line 7. Lines 2, 5 and 9 define the `Composers`, one for each rule. Lines 3, 6 and 10 show the `CondCheckers` which impose the pre-conditions that the entities, e, have to fulfil for each rule. In line 7 the Columns are created using the `EntityCreator` in the same way that in line 4 the Tables are created, with the only difference is that in order to create the Tables, the `Tracer` is needed to create the links to its Columns. When the `CondChecker` in line 10 is fulfilled, the `Finder` in line 11 retrieves all the Methods in the Blackboard, and for each one, a Column is created in line 13.

5 Conclusion and Other Work

In this paper, we have presented a collection of primitives which will be combined for running concurrent and distributed out-place model transformations using LinTra. As this is our initial attempt to come up with the collection of primitives, in the next, we plan to study deeply existing MTLs and extend our set of primitive to encompass, some functionality that we are missing if any.

In terms of related work, several lines of work consider the transformation of large models and their performance and scalability problems [7]. With LinTra [1,2], and its current implementation written in Java, we provide a framework to execute parallel and distributed model transformations that requires all MTs to be executed in Java. With the goal of designing a Domain-Specific Language (DSL), we inspire our work on T-Core [9]. Specifically, on its collection of primitive operators that allows to write in-place MTs in an intermediate level of abstraction which is between the high-level MTLs and the low-level code used by the engines.

After we discover the complete set of primitive operators, there are some other lines of work we would like to explore. First, we will implement the primitives and encapsulate the LinTra code into them. To achieve that, we will explore how to formulate, in the most efficient way, the OCL constraints using the methods available in LinTra to query the Blackboard. Second, we plan to create compilers from the most common MTLs the primitives, so that distributed models can be transformed in parallel reusing MTs written in those languages by means of executing them in the LinTra engine. Third, we want to investigate some annotations for the high-level MTL, so that the user can provide the engine details such as how the parallelization must be done, how the input model should be partitioned, etc. to improve the performance of the transformation. Finally, we plan to investigate the possibility of creating a new and more specific high-level MTL.

Acknowledgments This work is funded by Research Project TIN2011-23795 and by Universidad de Málaga (Campus de Excelencía Internacional Andalucía Tech).

References

1. Burgueño, L.: Concurrent Model Transformations based on Linda. In: Doctoral Symposium @ MODELS. (2013)
2. Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A.: On the Concurrent Execution of Model Transformations with Linda. In: BigMDE Workshop @ STAF. (2013)
3. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. Wiley, Chichester, UK (1996)
4. Cuadrado, J.S., Molina, J.G., Tortosa, M.M.: RubyTL: A Practical, Extensible Transformation Language. In: ECMFA. (2006)
5. Gelernter, D., Carriero, N.: Coordination languages and their significance. Commun. ACM **35**(2) (1992) 96–107
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. 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: BigMDE Workshop @ STAF. (2013)
8. OMG: MOF QVT Final Adopted Specification. Object Management Group. (2005)

9. Syriani, E., Vangheluwe, H., LaShomb, B.: T-core: A framework for custom-built model transformation engines. Software & Systems Modeling (2013) 1–29