LinTraP: Primitive Operators for the Execution of Model Transformations with LinTra

Loli Burgueño
Universidad de Málaga
Malaga, Spain
loli@lcc.uma.es

Eugene Syriani
University of Alabama
Tuscaloosa AL, USA
esyriani@cs.ua.edu

Manuel Wimmer
Vienna University of Technology
Vienna, Austria
wimmer@big.tuwien.ac.at

Jeff Gray
University of Alabama
Tuscaloosa AL, USA
gray@cs.ua.edu

Antonio Vallecillo
Universidad de Málaga
Malaga, Spain
av@lcc.uma.es

ABSTRACT

The problems addressed by Model-Driven Engineering (MDE) approaches are increasingly complex, hence performance and scalability of model transformations are gaining importance. In previous work, we introduced LinTra, which is a platform for executing out-place model transformations in parallel. The parallel execution of LinTra is based on the Linda coordination language, where high-level model transformation languages (MTLs) are compiled to LinTra and eventually executed through Linda. In order to define the compilation modularly, this paper presents a minimal, yet sufficient, collection of primitive operators that can be composed to (re-)construct any out-place, unidirectional MTL.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.1.3 [**Programming Techniques**]: Concurrent Programming; C.4 [**Computer Systems Organization**]: Performance of systems

Keywords

Model Transformation, Linda, LinTra

1. INTRODUCTION

Model-Driven Engineering [2] is a relatively new paradigm that has grown in popularity in the last decade. Although there is a wide variety of approaches and languages with different characteristics and oriented to different types of model transformations (MT), most model transformation engines are based on sequential and local execution strategies. Thus, they have limited capabilities to transform very large models (with thousands or millions of elements), and provide even less capabilities to perform the transformation in a reasonable amount of time.

BigMDE '14, July 24, 2014 York, United Kingdom

In previous works [3, 4], we investigated concurrency and distribution for out-place transformations to increase their performance and scalability. Our approach, LinTra, is based on Linda [8], a mature coordination language for parallel processes that supports reading and writing data in parallel into distributed tuple spaces. A tuple space follows the Blackboard architecture [5], 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 between the elements in the input model and the elements created from them are encoded for efficient retrieval, which agents are involved in the execution of the MTs and their role, how the MTs are executed in parallel, and how the models are distributed over the set of machines composing the cluster where each MT is executed.

The implementation of several case studies using the Java implementation of LinTra (jLinTra) is available on our website[1], together with the performance comparison with several well-known model transformation languages (MTLs) such as ATL [11], QVT-O [14] and RubyTL [7].

In order to hide the underlying LinTra architecture and in order to ease the compilation from any existing out-place MTL to the LinTra engine, in this paper we propose 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 that makes the parallel and distributed execution possible, serving as an abstraction of the implementation details of the general-purpose language in which LinTra is implemented.

The rest of the paper is structured as follows. Section 2 introduces the collection of primitives. Section 3 illustrates examples of primitive combinations in order to write MTs. Section 4 discusses the related work to our approach. Finally, Section 5 presents our conclusions and an outlook on future work.

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

2. COLLECTION OF PRIMITIVES

This section shortly introduces LinTra, presents the set of primitive operators, and describes the mapping of the primitive operators to LinTra.

2.1 Background on LinTra

LinTra uses the Blackboard paradigm [5] 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 that is used for representing relationships between entities and by the trace model. Relationships between entities are represented by storing in the source entity the identifier(s) of its target entity(ies).

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 a 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. 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.

Together with the Blackboard, LinTra uses the Master-Slave design pattern [5] to execute 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 (i.e., partitions) 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 retrieve them.

2.2 Primitives

Two different kinds of primitives can be distinguished in LinTra: the primitive constructs to encapsulate the concurrent execution platform and the primitive constructs needed by the MTL.

Primitives for the Concurrent Platform. Despite the fact that due the representation of models in LinTra, all model elements are independent from each other, LinTra requires the user to specify the size of every partition, i.e., how many elements belong to each one. Furthermore, although there is no need of specifying how the elements are partitioned or which elements belong to the same partition, LinTra offers that possibility.

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 empty and the union is the whole model. The purpose of OE is to give the user the possibility to optimize the MT execution.

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

`Composer` is a primitive that allows the grouping of a combination of primitives and assigns 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 a parameter the name of the Composer and the set of entities from the pre or post-condition and gives 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)*. The *Collection* corresponds to the four collection types in OCL: Set, OrderedSet, Bag, and Sequence. Furthermore, in a Composer, more than one element might be created; thus, in the `Tracer`, the concrete `Creator` might need to be specified given its name, being its syntax *Tracer(composer : Composer, e : Collection(Entity), creatorName : String) : Collection(Entity).*

`Creator` creates an entity given its data 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. The dictionary is a collection of *key-value* pairs where the first element is the name of the feature and the second its value. The type of the values received by the dictionary are of two kinds: OCL primitive data types, which correspond to the basic data types of the language (string, boolean and numbers in their different formats), and the types defined by all the classes given by the output metamodel. Furthermore, the values can be an OCL collection of the previous types. Its syntax is *Creator(type : Factory, features : Dictionary<feature : String, value: OCLDataType | Entity>).* Moreover, the Creator might have an optional parameter of type *String* specifying its name, *Creator(type : Factory, features : Dictionary<feature : String, value: [OCLDataType | Entity]>, name : String).* This is needed in case that it is referenced by a *Tracer.*

`CondChecker` allows the querying of the input model in 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 a parameter an OCL expression and returns the set of entities (submodel) that fulfils the OCL expression. Its signature is *Finder(expr : OCLExpression) : Collection(Entity).*

`Declarer` allows to create a global variable that can be accessed by its name from any other primitive and that is accessed by all the Slaves involved in the transformation process. Its syntax is *Declarer(type : [OCLDataType | Entity], name : String).* The value of the variable is set by `Assigner`.

`Assigner` sets the value of a variable defined by `Declarer`. `Assigner` receives as a parameter the name of the variable and its value. Its syntax is *Assigner(varName : String, value : [OCLDataType | Entity | Creator]).* In the case that the second parameter is a `Creator`, the element is stored in the Blackboard and the variable points to it. In case the variable is stored in the Blackboard, every time it is updated, the corresponding value in the Blackboard is overwritten. If the second parameter is an OCL primitive data type or an entity, the variable is stored in memory and accessed while the MT is executed but it is not a persistent value in the Blackboard.

Figure 1 shows a class diagram with all the primitives and their relationships.

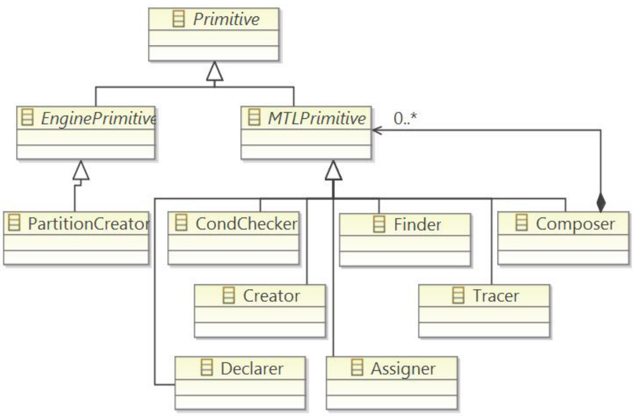

Figure 1: Primitives Class Diagram.

2.3 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 execution 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 [14], and entrypoint and endpoint rules in ATL [11]. 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 rules (or 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.

After the setup phase is finished, the LinTra MT engine 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.

Listing 1: Master.

```
params :: Integer : nSlaves
index := 1
slavePool := createSlaves(nSlaves)
task := Blackboard.Tasks.dequeue()
while ( task != null ){
  while (task != null
    and task.ruleLayer.index = index)
    slave := slavePool.getIdleSlave() -- blocking
    slave.execute(task)
    task := Blackboard.Tasks.dequeue()
  }
  join() -- wait for all the slaves to finish
         -- before starting to transform the
         -- tasks involving the next ruleLayer
  index := index + 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 2. An overview of how the system works can be seen in the activity diagram presented in Figure 2.

Listing 2: Slave - execute method

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

The sequential execution of a MT is a concrete scenario in LinTra. There are several ways to achieve it. The MT is executed sequentially either by not partitioning the input model (therefore, only one task is created and executed sequentially by a single slave) or by launching only one slave

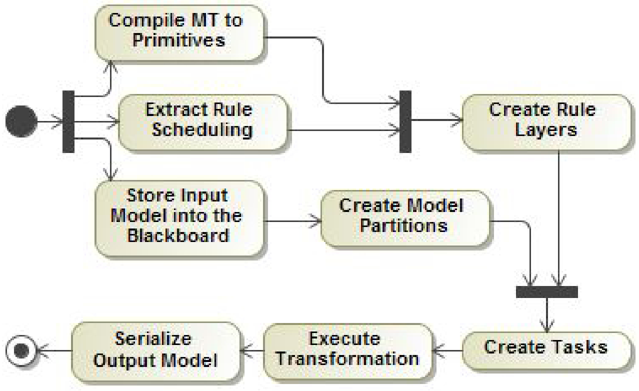

Figure 2: Activity Diagram of the Transformation Process.

that transforms all the tasks.

A class diagram showing all the elements involved in LinTra and how they are related to each other can be found in Figure 3. It contains the Master and Slave where every slave executes a Transformation which is a collection of MT Primitives that accesses to a Blackboard which is composed by Areas that contain both Tasks - formed by a Rule Layer and Partitions - and and the Entities that belong to a certain Model. *MTLPrimitive* in this diagram corresponds with the root class in the diagram presented in Figure 1.

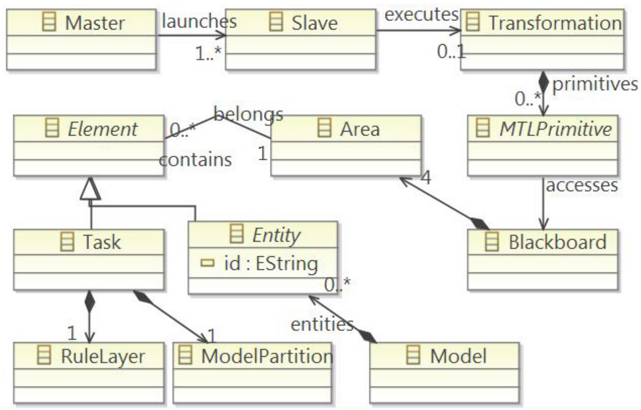

Figure 3: LinTra Class Diagram Metamodel.

3. EXAMPLES

This section demonstrates how the introduced primitives are used for concrete transformation examples.

3.1 Activity Diagram to Petri Net

This case study is a simplification of the transformation from UML Activity Diagrams to Petri Nets described in [15]. The metamodels are represented in Figures 5 and 6 and, for simplicity, only contain the elements needed by our simplified transformation.

The MT simplification consists of an unaltered subset of the original MT which focuses on transforming only several elements belonging to the input model instead of the whole model. Every *Initial Node* is transformed to a processing *Place* with one token, an *Arc* pointing to a *Transition* and other *Transition*. Every *Final Node* is transformed to a *Transition*, an *Arc* pointing to a *Place* and such *Place*. From every *Action Node*, an entry *Transition*, an *Arc* pointing to a *Place*, such *Place*, an *Arc* from it to another *Transition*, and such *Transition* are created. Every *Signal* is transformed in the same way as a *Final Node* and every *Accept Signal* as an *Initial Node* but with no token. *Activity Edges* between any kind of nodes are transformed as an *Arc* pointing to a *Place*, the *Place* and another *Arc* coming from it. Every pair *Signal-Accept Signal* with the same value for their feature *signalId* are transformed in the same way as *Activity Edges*. For a better understandability, the previous transformation rules are represented graphically in Figure 4. Finally, only one entity of *PetriNet* is created in the output model whose name is the String “PNet” concatenated with the number of arcs, the number of places and the number of transactions in the output model after the whole transformation process. All places, arcs and transitions must be linked to that *PetriNet* entity.

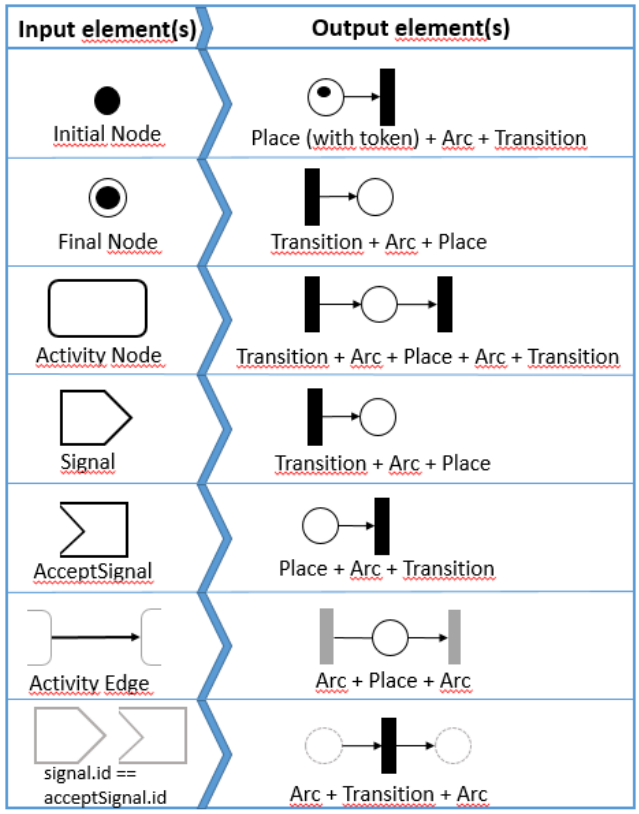

Figure 4: Activity Diagram to Petri Net Transformation.

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, three different rule lay-

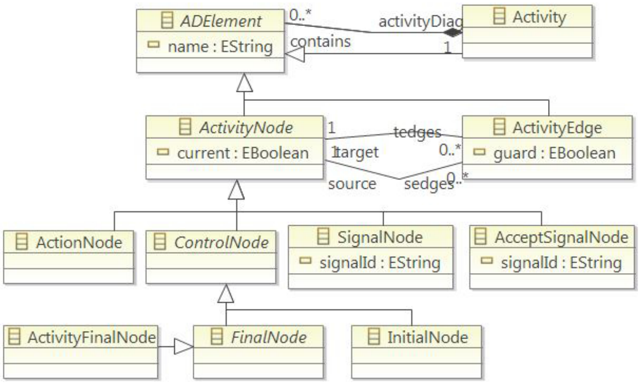

Figure 5: Activity Diagram Metamodel.

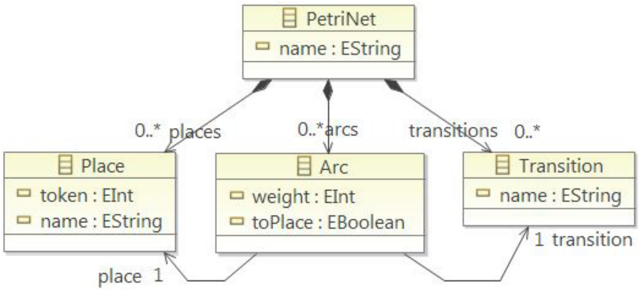

Figure 6: Petri Net Metamodel.

ers are created: $RL = [l1, l2, l3]$ where *l1* contains the first composer where a global variable for the unique Petri Net that will be referenced by the rest of entities is created, in *l2* all the elements are created and in *l3*, the name of the Petri Net is changed. Given the partitions and the layers, the tasks to be executed are $T = [T1, T2, T3]$, where $T1 = \{(p1, l1), (p2, l1), (p3, l1)\}$, $T2 = \{(p1, l2), (p2, l2), (p3, l2)\}$ and $T3 = \{(p1, l3), (p2, l3), (p3, l3)\}$. We make the distinction between *T1*, *T2* and *T3* to clarify that all tasks in *T1* are relative to *l1*, all tasks in *T2* are relative to *l2* and all tasks in *T3* are relative to *l3*; thus, until all tasks from *T1* have been executed, tasks from *T2* cannot start and until all tasks in *T2* have been executed, tasks from *T3* cannot start.

The compilation process from the high-level MT to the primitives produces the code shown in Listings 3, 4, and 5.

Listing 3: MTL Primitives for the first rule layer (l1).

```
1 Composer First {
2   Declarer (PetriNet, pNet)
3   Assigner(pNet,
4     Creator(PetriNet, {[name, 'PNet']}))
5 }
```

As the case study requires that only one *PetriNet* instance is created and the rest of the elements in the output model reference it, there is a need for a global variable that must be available before the rest of the rules are applied. Listing 3 declares in line 1 a composer which, encapsulates the declaration of a variable called **pNet** (line 2) and the creation of the *PetriNet* entity (lines 3 and 4). Note that, as the entity created is a persistent entity which is part of the output model (instead of a temporary variable), the second parameter of the assigner is a creator, which means that the value is stored in the Blackboard and the variable is a pointer to it.

Listing 4 shows part of the primitives that compose the second rule layer. In particular, this listing shows the collection of primitives to transform *ActionNodes* and *SignalNodes* and to match the output entities created from *SignalNodes* and *AcceptSignalNodes*.

Lines 2, 21 and 33 show the condition checkers which impose the pre-conditions that the entity, e, given by a task, has to fulfil to be transformed by the set of primitives inside the `if` the condition checker. For instance, given e, if the condition checker in line 2 is fulfilled, it means that e is of type *SignalNode* and from it, the entities specified by the creators in lines 3, 5, 10, 12 and 17 will be created. For example, in the creator in line 5, an `Arc` is created where *transition* points to the entity created by the creator called *t1*, *place* points to the entity created by creator `p`, *toPlace* is set to `true` and `net` points to the element given by the global variable `pNet`. The name of the creators is optional, and in this example, it is only given when it is needed by a tracer. For example, the tracer in line 6 gives the reference to the entity created from e in *ActNode* by a creator called *t1*.

A tracer can give the reference to an entity that has been created either in the same composer or in a different composer. It can also point either to a composer located in the same rule layer or in a different rule layer. An example of the first case is the tracer in line 39, which points to a creator in the composer *Signal*.

The last composer encompasses the entities created by every pair *Signal-Accept Signal* with the same *signalId*. This is a particular case where from every entity, e, received in the task and fulfilling the condition checker in line 33 (i.e. whose type is *SignalNode*), it is needed to find in the Blackboard all the elements of type *AcceptSignalNode* with the same signal identifier as e. This is achieved by using the Finder primitive in line 34.

Listing 4: MTL Primitives for the second rule layer (l2).

```
1 Composer ActNode {
2   if (CondChecker(e.oclIsTypeOf(ActionNode)))
3     Creator(Transition,
4       {[name, e.name], [net, pNet]}, 't1')
5     Creator(Arc,
6       {[transition,Tracer(ActNode,e, 't1')],
7       [place,Tracer(ActNode,e,'p')],
8       [toPlace, true],
9       [net, pNet]})
10    Creator(Place,
11      {[name,e.name],[net,pNet],[token,0]}, 'p')
12    Creator(Arc,
13      {[transition,Tracer(ActNode, e, 't2')],
14      [place,Tracer (ActNode, e, 'p')],
15      [toPlace, false],
16      [net, pNet]})
17    Creator(Transition,
18      {[name, e.name], [net, pNet]}, 't2')
19 }
```

```
Composer Signal {
  if (CondChecker(e.oclIsTypeOf(SignalNode)))
    Creator(Transition,
      {[name, e.name], [net, pNet]}, 't')
    Creator(Arc,
      {[transition,Tracer(Signal,e,'t')],
      [place, Tracer(Signal, e, 'p')],
      [toPlace, true],
      [net, pNet]})
    Creator(Place,
      {[name, e.name],[net, pNet]}, 'p')
}
Composer MatchSignals {
  if(CondChecker(e.oclIsTypeOf(SignalNode)))
    for(a in Finder(AcceptSignalNode.allInstances
      ->select(as|e.activityDiag = as.activityDiag
      and e.signalId = a.signalId)
      Creator (Arc,
        {[place,
          Tracer(Signal, e, 'p')],
        [transition,
          Tracer(MatchSignals, {e, a}, 't')],
        [toPlace, false ],
        [net, pNet]})
      Creator (Transition,
        {[name, e.name+'-'+a.name],
        [net, pNet]}, 't')
      Creator (Arc,
        {[place,
          Tracer(AcceptSignal, e, 'p')],
        [transition,
          Tracer(MatchSignals, {e, a}, 't')],
        [toPlace, true ],
        [net, pNet]})
}
...
```

Finally, once all the output entities have been created, the third rule layer, where the name of the only *PetriNet* is updated, can be executed. Listing 5 shows how it is done using an `Assigner` and a `Creator` inside of it that overwrites the value of the `pNet`.

Listing 5: MTL Primitives for the third rule layer (l3).

```
Composer Last {
  Assigner (pNet,
    Creator(PetriNet,
      {[name, pNet.name+(pNet.arcs.size()
        +pNet.places.size()
        +pNet.transitions.size())]}))
}
```

The complete case study can be downloaded from our website[2]. Note that, although the case study in [15] is an out-place MT, i.e. the input and output metamodels are different and the input model is not modified, the authors have used an in-place MTL; thus, although the semantics of the MT is the same, our solution is different to theirs.

3.2 Filtering Families

In this subsection, we introduce a second case study where the input and output metamodel are the *Family* metamodel shown in Figure 7. The MT consists of filtering the input model so that the output metamodel is a subset of the input model that contains only the families which have exactly two daughters, two sons and their family members. This means that the members belonging to families with more than two daughters and two sons are not in the output model.

[2] `http://atenea.lcc.uma.es/index.php?title=Main_Page/Resources/Linda/ActivityDiag2PetriNet`

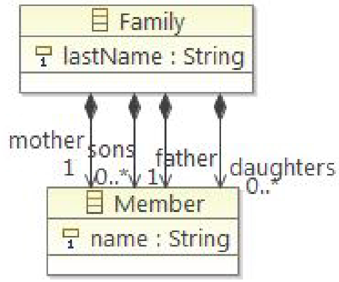

Figure 7: Family Metamodel.

For example, this behaviour is done in ATL using a particular kind of rule called a lazy rule. Lazy rule are not completely declarative, but they must be invoked explicitly. In this way, the transformation for this example has a main rule that checks if a family fulfilled the requirements and in that case, a lazy rule that transforms its members is called. Although in most of the cases there is a direct relation between rules in the high-level MTL and composers, this case is an exception. With our collection of primitives, this is done by means of a unique Composer.

Listing 6 shows the MTL primitives for this case study. An entity, e, fulfils the condition in line 2, in line 5 a *Family* is created. Then, the condition checkers in lines 11 and 15 and creators in lines 12 and 16 transform every mother and father of that family. All sons and daughters are transformed in lines 20 and 24. Tracers in lines 6 and 7 reference creators that can be invoked or not because they are inside `if`s, in the case that no entity is created, the reference points to null. Tracers in lines 8 and 9 point to entities created inside a `for`, those tracers return the pointers to all the elements created in that creator. The complete case study can be found on our website[3].

Listing 6: MTL primitives for the Filtering Families case study.

```
1  Composer R {
2    if(CondChecker(e.oclIsTypeOf(Family)
3      and e.daughters.size()=2
4      and e.sons.size()=2 ))
5      Creator(Family, {[lastName, e.lastName],
6        [father, Tracer(R, e, 'f')],
7        [mother, Tracer(R, e, 'm')],
8        [daughters, Tracer(R, e, 'ds')],
9        [sons, Tracer(R, e, 'ss')]},
10       'fam')
11     if(CondChecker(not e.father.isOclUndefined()))
12       Creator(Member,{[name,e.father.name],
13         [familyFather, Tracer(R, e, 'fam')]},
14         'f')
15     if(CondChecker(not e.mother.isOclUndefined()))
16       Creator(Member,{[name,e.mother.name],
17         [familyMother, Tracer(R, e, 'fam')]},
18         'm')
19     for(daughter in e.daughters)
20       Creator(Member,{[name,daughter.name],
21         [familyDaughter, Tracer(R, e, 'fam')]},
22         'ds')
23     for(son in e.sons)
24       Creator(Member,{[name,son.name],
25         [familySon, Tracer(R, e, 'fam')]},
26         'ss')
27 }
```

[3] `http://atenea.lcc.uma.es/index.php?title=Main_Page/Resources/Linda/FilteringFamilies`

4. RELATED WORK

With respect to the contribution of this paper, we first elaborate on related work considering the performance of model transformations in general and concerning parallel execution in particular and second we discuss how the work on primitives for model transformations is extended by this work.

The performance of model transformations is now considered as an integral research challenge in MDE [12]. For instance, Amstel et al. [18] considered the runtime performance of transformations written in ATL and in QVT. In [19], 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 [6] where Clasen et al. 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. A follow-up work on this is presented in Tisi et al. [17] where a parallel transformation engine for ATL is presented.

There is also some work in the field of graph transformations where multi-core platforms are used for the parallel execution of model transformation rules [1, 9] especially for the matching phase of the left-hand side of graph transformation rules. A recent work exploiting the Bulk Synchronous Parallel model for executing graph transformations based on the Henshin transformation tool is presented in [13]. Finally, model queries are executed for large models in a distributed manner in an extension of EMF Inc-Query by combining incremental graph search techniques and cloud computing [10].

With LinTra [3, 4], and its current implementation written in Java, jLinTra[4], 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 based our work on T-Core [16], with specific focus on T-Core's 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.

The main difference between T-Core and LinTraP is that T-Core focuses on in-place MT while LinTra focuses on out-place MT. This means that the nature of the problems to address is different and also the way in which the MTs are written. For instance, while in T-Core there exists the primitive Rewriter that update the input model, in LinTra there exists the primitive Creator that creates entities in the output model.

5. CONCLUSION AND FUTURE 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.

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

After having analyzed different high-level MTLs and the LinTra characteristics and having discovered the complete set of primitive operators, there are several other lines of work we would like to explore. First, we will implement the primitives and encapsulate the LinTra code written in Java (jLinTra) 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 languages such as ATL or QVT-O to 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 for parallel transformations.

Acknowledgment

This work has been supported by Spanish Project TIN2011-23795 and by Universidad de Málaga (Campus de Excelencía Internacional Andalucía Tech).

6. REFERENCES

[1] G. Bergmann, I. Ráth, and D. Varró. Parallelization of graph transformation based on incremental pattern matching. *ECEASST*, 18, 2009.

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

[3] L. Burgueño. Concurrent Model Transformations based on Linda. In *Proceedings of Doctoral Symposium @ MODELS*, pages 9–16, 2013.

[4] L. Burgueño, J. Troya, M. Wimmer, and A. Vallecillo. On the Concurrent Execution of Model Transformations with Linda. In *BigMDE Workshop @ STAF*, 2013.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[6] C. Clasen, M. Didonet Del Fabro, and M. Tisi. Transforming Very Large Models in the Cloud: a Research Roadmap. In *Proceedings of CloudMDE Workshop @ ECMFA*, 2012.

[7] J. S. Cuadrado, J. G. Molina, and M. M. Tortosa. RubyTL: A Practical, Extensible Transformation Language. In *Proceedings of ECMFA*, pages 158–172, 2006.

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

[9] G. Imre and G. Mezei. Parallel Graph Transformations on Multicore Systems. In *Proceedings of MSEPT*, pages 86–89, 2012.

[10] B. Izsó, G. Szárnyas, I. Ráth, and D. Varró. IncQuery-D: Incremental Graph Search in the Cloud. In *Proceedings of BigMDE Workshop @ STAF*, pages 4:1–4:4, 2013.

[11] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

[12] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot. A Research Roadmap Towards Achieving Scalability in Model Driven Engineering. In *Proceedings of BigMDE Workshop @ STAF*, 2013.

[13] C. Krause, M. Tichy, and H. Giese. Implementing Graph Transformations in the Bulk Synchronous Parallel Model. In *Proceedings of FASE*, pages 325–339, 2014.

[14] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Object Management Group, 2011.

[15] E. Syriani and H. Ergin. Operational semantics of UML activity diagram: An application in project management. In *Proceedings of MoDRE Workshop @ RE*, pages 1–8, 2012.

[16] E. Syriani, H. Vangheluwe, and B. LaShomb. T-core: a framework for custom-built model transformation engines. *Software & Systems Modeling*, pages 1–29, 2013.

[17] M. Tisi, S. M. Perez, and H. Choura. Parallel Execution of ATL Transformation Rules. In *Proceedings of MoDELS*, pages 656–672, 2013.

[18] M. van Amstel, S. Bosems, I. Kurtev, and L. F. Pires. Performance in Model Transformations: Experiments with ATL and QVT. In *Proceedings of ICMT*, pages 198–212, 2011.

[19] 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.