Solving the Movie Database Case: A solution using the Maude-based e-Motions tool.

Antonio Moreno-Delgado and Francisco Durán

University of Málaga
{amoreno,duran}@lcc.uma.es

Abstract.

1 Introduction

TO-DO: two sentences introducing Maude

Maude may be seen as a general framework where to develop model transformations. In this way, some work has been done [**?**]. Since a term is very general, one could specify graphs or models as terms. Thus, a Maude module can define a *in-place* model transformation, where rewriting rules define transitions between two states or models.

e-Motions [**?**] is a Domain-Specific Modeling Language (DSML) and a very general tool that supports the specification and simulation of any real-time DSML. Artifacts developed in e-Motions are translated to Maude in a transparent way. The e-Motions simulation is achieved using the Maude engine. Therefore, e-Motions can be seen as a framework where graphically code in Maude.

1.1 e-Motions

For the sake of comprehension of the rest of the paper, in this section we briefly present e-Motions. The definition of a Domain-Specific Language (DSL)[(i)] comprises three tasks: (i) the definition of the abstract syntax, (ii) the definition of the concrete syntax and (iii) the specification of the behavior.

[(i)] hablamos de DSLs en general, no?

The abstract syntax is defined by means of a Ecore metamodel, in which all the language concepts and the relations between them are specified. The concrete syntax is achieved defining the so-called Graphical Concrete Syntax (GCS). A GCS is a model (conforms the GCS metamodel) where an image is attached to each concept defined in the abstract syntax.

The behavior of a DSL is specified using visual graph-transformation rules. An e-Motions rule consists of a—possibly conditional—Left-Hand Side (LHS), a Right-Hand Side (RHS) and zero or more Negative Application Conditions (NACs). The LHS defines a (sub)-graph matching, optionally conditional. The RHS specifies a (sub)-graph replacement, which if the rule is applied, every object in the LHS that is not in the RHS is deleted, new objects in the RHS that are not in the LHS are created, and those objects whose attributes (or links)

are changed are updated. NACs specify conditions or (sub)-graph such that if there is a matching, the rule cannot be fired.

Fig. 1 shows an example of an e-Motions rule.[1] The objects in both the RHS and LHS are represented by their images defined in the GCS model. Rule `Assemble`'s LHS defines the precondition of the rule. It models a assemble machine who needs both a head and a handle in its connected conveyor. If the `NAC1`, stating that the current matched `Assemble` has not unfinished other rule, is not satisfied, the rule can be applied. The rule is applied as follows. All objects in the LHS which they do not appear in the RHS are deleted, i.e. `he` and `ha` objects. Those objects in the RHS which do not appear in the LHS are created, setting their attributes properly, i.e. the `ham` object with its three attributes. The rest of objects remain changeless. Moreover, as e-Motions is a framework where to define real-time systems, each rule is applied in a established time, i.e. `[prodTime,prodTime]` in the `Assemble` rule. A rule may contain zero or more local or auxiliary variables. All attribute or variable assignments and conditions are coded in Object-Constraint Language (OCL) [**?**].

The abstract and concrete syntax, and the behavior of a DSL are models and the e-Motions tool has been developed following MDE principles. The Maude code corresponding to a system defined in e-Motions is generated by an ATL/TCS transformation [**?**].

2 Solution

(ii) esto se diría así?

Since e-Motions supports in a very intuitive, user-close(ii) and visual way, we show how the IMDb problem [2] can be solved using it. Each task is solved using the definition of a DSL, which shares in general the abstract and concrete syntax. The rewrite rules defining the behavior depends on the solution given. All rewrite rules are *instantaneous*, since the case to be solved do not have time requirements.

(iii) se dice así?

However, e-Motions is very general: it supports OCL expressions and time requirements. This makes that the Maude code generated by e-Motions tool is not as time performance(iii) as desired. Together with the e-Motions solution we present a optimized Maude solution. This approach is very similar to real software engineering problems, in which a initial prototype is specified and once the developers are confident with the design, they develop the final implementation.

Finally, although e-Motions is presented as a Eclipse tool, we run the simulations with the code generated by it directly in a Maude instance, since the transformation from a e-Motions-based system into Maude code takes time.

2.1 Task 1

Task 1 comprises the generation of synthetic models (conforming the movie database metamodel [2]) from an input parameter $N \geq 0$. In the following we present a e-Motions based solution and a Maude solution.

[1] A bunch of examples and installation help are available at `http://atenea.lcc.uma.es/e-Motions`.

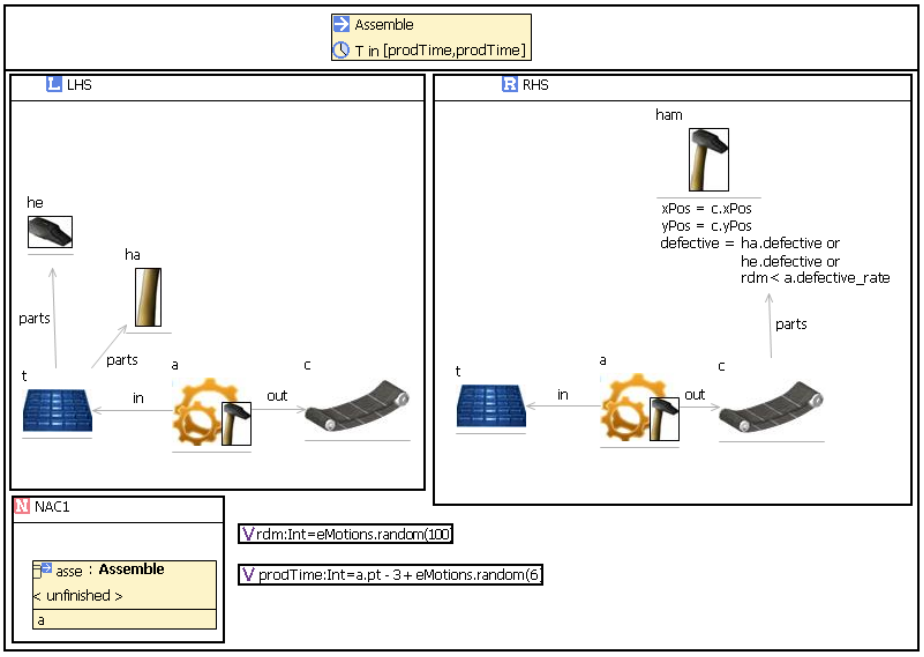


Fig. 1: e-Motions `Assemble` rule.

e-Motions-based solution Following an e-Motions based approach, we define the abstract and concrete syntax and the behavior of our so-called *Task 1 DSL*, which takes a model with a parameter N and generate as output a model containing synthetic data to be used as test case.

As it has been introduced in Section 1.1, the abstract syntax of a DSL is given by means of a Ecore metamodel, which is provided in [1] and, in the following, we call it *Movies MM*. However, the *parameter N* concept has to be modeled in some way, since in e-Motions the state[(iv)] is just a model. Hence, a new concept call `Parameter` with two Integer attributes `nP` and `nN` (positive and negative graphs respectively) has been added to Movies MM. This results in a so-called *Movies* MM*.[(v)]

For the concrete syntax, Fig. 2 shows how an image has been attached to each concept modeled in the Movies* MM. The behavior of this *Task 1 DSL* is given by means of two in-place rules: `createPositive` and `createNegative`. Figure 3a shows the `createPositive` rule, which takes an object `p` of type *Parameter* with *nP* attribute is greater or equal than 0 and, after the rule application, synthetic data conforming to the Henshin rules [2] are created. Fig. 3b shows the `createNegative` rule, which is analogously defined.

Once the syntax and the behavior of the system has been coded, the user may specify a model, which conforms to *Movies* MM*, containing an object

(iv) con este state me refiero al estado del sistema, de una ejecución

(v) podríamos referenciar a los trabajos donde esto se hace de forma modular

`Parameter` with its two attributes `nP` and `nN` properly set. This model is used as initial model of the execution.

Maude version This proposal of Task 1 consists of a object-based Maude specification, which is composed by two main modules: the `MOVIES@MM` module defining the classes structure and the `TASK1` module defining the solution. The solution is coded using again two rules: `createPositive` and `createNegative`. One could realized that the Maude version is very much like the e-Motions version. In fact, the former is almost the textual version of the latter. Listing 1.1 shows the `createPositive` Maude rule that takes the `createPositive(N:Nat)` message and a `freshOid` auxiliary message—used to create new object identifiers—and returns such a object configuration conforming the Henshin specification [2].

The messages `createPositive` and `createNegative` are generated in zero-rewrite steps with the equation showed in Listing 1.2.

Listing 1.1: `createPositive` Maude rule.

```
rl [createPositive] :
  createPositive(N)
  freshOid(N')
=>
  < N'     : Movie | rating : (10.0 * float(N)) >
  < N' + 1 : Movie | rating : (10.0 * float(N) + 1.0) >
  < N' + 2 : Movie | rating : (10.0 * float(N) + 2.0) >
  < N' + 3 : Movie | rating : (10.0 * float(N) + 3.0) >
  < N' + 4 : Movie | rating : (10.0 * float(N) + 4.0) >

  < N' + 5 : Actor | name : ("a" + string((10 * N),     10)),
                     movies : (N', N' + 1, N' + 2, N' + 3)     >
  < N' + 6 : Actor | name : ("a" + string((10 * N + 1), 10)),
                     movies : (N', N' + 1, N' + 2)             >
  < N' + 7 : Actor | name : ("a" + string((10 * N + 2), 10)),
                     movies : (N' + 1, N' + 2, N' + 3)         >
  < N' + 8 : Actress | name : ("a" + string((10 * N + 3), 10)),
                     movies : (N' + 1, N' + 2, N' + 3, N' + 4) >
  < N' + 9 : Actress | name : ("a" + string((10 * N + 4), 10)),
                     movies : (N' + 1, N' + 2, N' + 3, N' + 4) >
  freshOid(N' + 10) .
```

Listing 1.2: Equation `createExample(N:Nat)`.

```
eq createExample(0) = none .
eq createExample(s(N)) = createPositive(N)
                         createNegative(N)
                         createExample(N) .
```

TO-DO: Correctness?

TO-DO: Time tables once we have installed maude in the Ubuntu image

(a) Actor. (b) Actress. (c) Movie. (d) Couple. (e) Parameter.

Fig. 2: Concrete syntax for *Movies* MM*.

2.2 Task 2

Task 2 consists of extract all couples from a given model, either from Task 1 or IMBd database [**?**]. Two persons are couple whether they played together in at least three movies [2]. Once again we present the e-Motions and the Maude solution.

e-Motions-based solution Fig. 4 shows the `createCouple` rule which implements the whole task. `Person` objects are shown using square shapes because `Person` is an abstract class and it does not have image attached. The `createCouple` rule consists of a LHS with a OCL condition, which states *"LHS holds iff there are two persons* `per1` *and* `per2`*, such that the number of movies in the intersection between* `per1`*'s movies and* `per2`*'s movies is greater or equal than 3"*. Moreover, the `coupleHasNotBeenCreated` NAC avoids the application of the rule if the couple already exists.

However, although this solution works, one can notice that the number of matchings in the LHS of the rule is combinatorial. The fact that there are such a large number of matchings makes this solution too inefficient. **TO-DO**: add numbers supporting this.

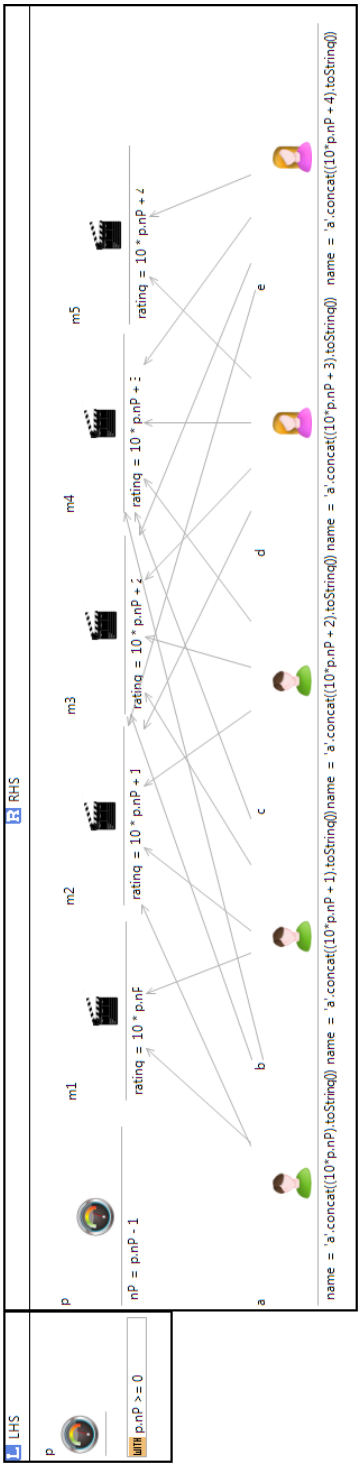
We have implemented another solution in which we limit the number of matchings using the next algorithm:

1. We split `Person`s and `Movie`s into separate configurations.
2. We fix a `Person`.
3. Given a `Person`, we look for all couples.
4. Whether the current `Person` set has been gone over all the other persons, we set the next person, and the current person is move to the resulting collection.

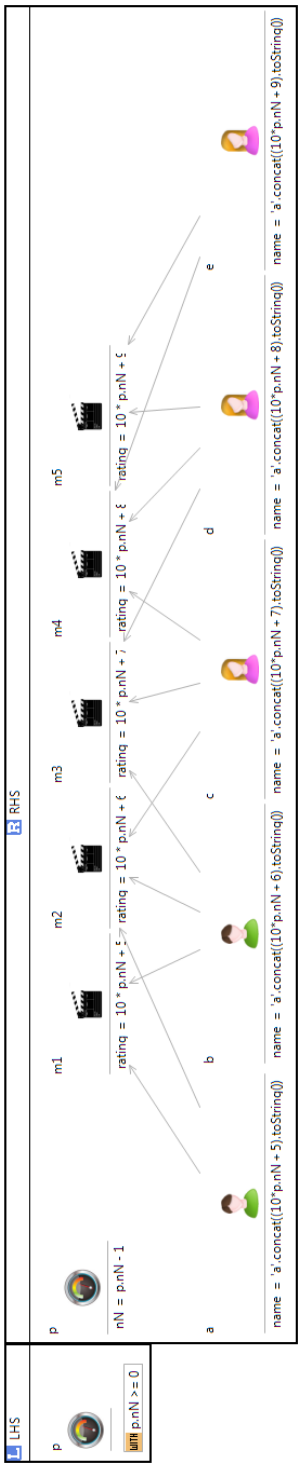
Following this approach the number of persons to match as possible couple decrease. A new concept so-called `Collection` is added to the metamodel with its concrete syntax to implement the algorithm above mentioned. Fig. 5 shows one of the rules specified for this solution,[2] namely those that creates new couples. The box is the concrete syntax for the `Collection` concept.

TO-DO: add numbers

[2] The rest of the rules are available at `https://github.com/antmordel/ttc14emotions`.



(a) The `createPositive` rule.



(b) The `negativePositive` rule.

Fig. 3: Task 1 rules.

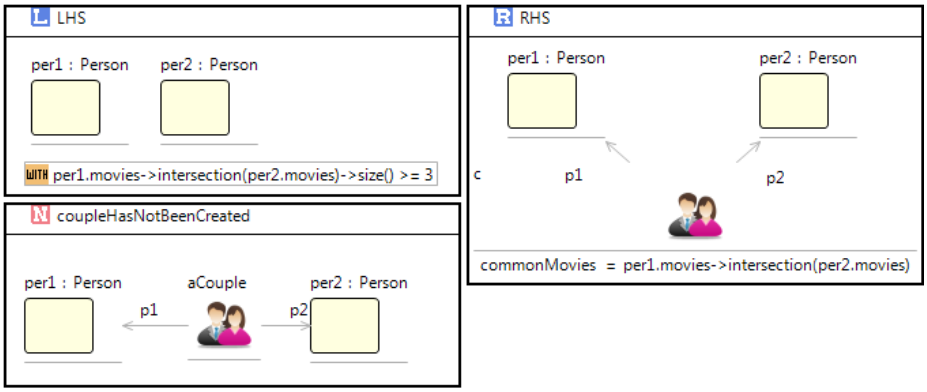


Fig. 4: `createCouple` rule.

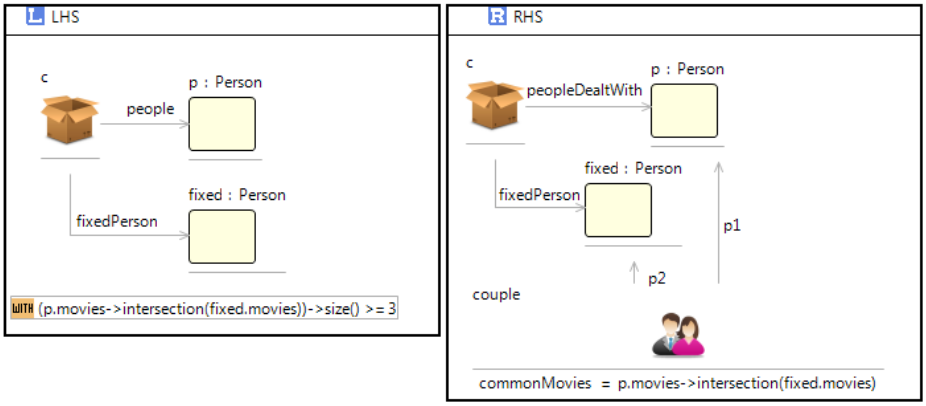


Fig. 5: `doingCouples` rule.

Maude-based solution

References

1. Horn, T.: IMDB2EMF, `https://github.com/tsdh/imdb2emf`
2. Horn, T., Krause, C., Ticky, M.: The TTC 2014 Movie Database Case, available at TTC14 web site.