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

Maude [?,?] is an executable formal specification language based on rewriting logic, which counts with a rich set of validation and verification tools [?,?], increasingly used as support to the development of UML, MDA, and OCL tools (see, e.g., [?,?,?]). Furthermore, Maude has demonstrated to be a good environment for rapid prototyping, and also for application development (see surveys [?,?]).

Maude may be seen as a general framework where to develop model transformations. Thus, Meseguer and Boronat use it to implement their model transformation framework MOMENT2; Durán, Vallecillo and others have used it to develop e-Motions [3], a tool that supports the definition and simulation of real-time Domain-Specific Modeling Languages (DSMLs); and similar approaches have later been used to give semantics to ATL [4] and other transformation languages.

The e-Motions tool is a DSML and graphical framework developed for Eclipse that supports the specification, simulation, and formal analysis of real-time systems. It provides a way to graphically specify the dynamic behavior of DSMLs using their concrete syntax, making this task quite intuitive. Furthermore, e-Motions behavioral specifications are models too, so that they can be fully integrated in MDE processes.

In e-Motions, MOF metamodels are formalized in rewriting logic, providing a representation of the structural aspects of any modeling language with a MOF metamodel. Then, given a description of the behavior of such modeling language as in-place transformation rules, e-Motions may be used to define both the syntax and the operational semantics of DSMLs. Artifacts developed in e-Motions are automatically translated into Maude.

As we will see in the following sections, e-Motions provides a very rich set of features, that enables the formal and precise definition of real-time DSMLs as models in a graphical and intuitive way. It makes use of an extension of in-place model transformation with a model of timed behavior and a mechanism to state action properties. The extension is defined in such a way that it avoids artificially modifying the DSML's metamodel to include time and action properties~~,~~. Moreover, it supports attribute computations and ordered collections, which are specified by means of OCL expressions, thanks to mOdCL [?]. All these features makes the language very expressive, but directly impact on performance. To gain an idea of this impact, we provide below solutions to the proposed problems both in e-Motions and directly in Maude and compare them.

The e-Motions system documentation and several examples are available at `http://atenea.lcc.uma.es/e-Motions`. The Maude web site is at `http://maude.cs.uiuc.edu`.

1.1 e-Motions

The definition of a Domain-Specific Language (DSL) typically comprises three tasks: (i) the definition of its abstract syntax, (ii) the definition of its concrete syntax and (iii) the specification of its behavior.

In e-Motions the abstract syntax is defined by means of an Ecore metamodel, in which all the language concepts and the relations between them are specified. The concrete syntax is provided by 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.

In e-Motions 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)-graphs such that if there is a matching, the rule cannot be fired.

Figure 1 shows an example of an e-Motions rule. The objects in both the RHS and LHS are represented by their associated images, as defined in the GCS model. Rule `Assemble`'s LHS defines the precondition of the rule. It models an assemble machine who needs both a head and a handle in its connected conveyor. If `NAC1`, stating that the current matched `Assemble` object is not involved in other `Assemble` action, is not satisfied, the rule can be applied. The rule is applied as follows. All objects in its LHS which do not appear in its RHS are deleted, i.e., objects `he` and `ha`. Those objects in its RHS which do not appear in its LHS are created, properly setting their attributes, i.e., the `ham` object with its three attributes. The rest of the 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 with execution time `[0,0]` is considered instantaneous. A rule may contain zero or more local or auxiliary variables. All attribute or variable assignments and conditions are expressed using 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 [?].

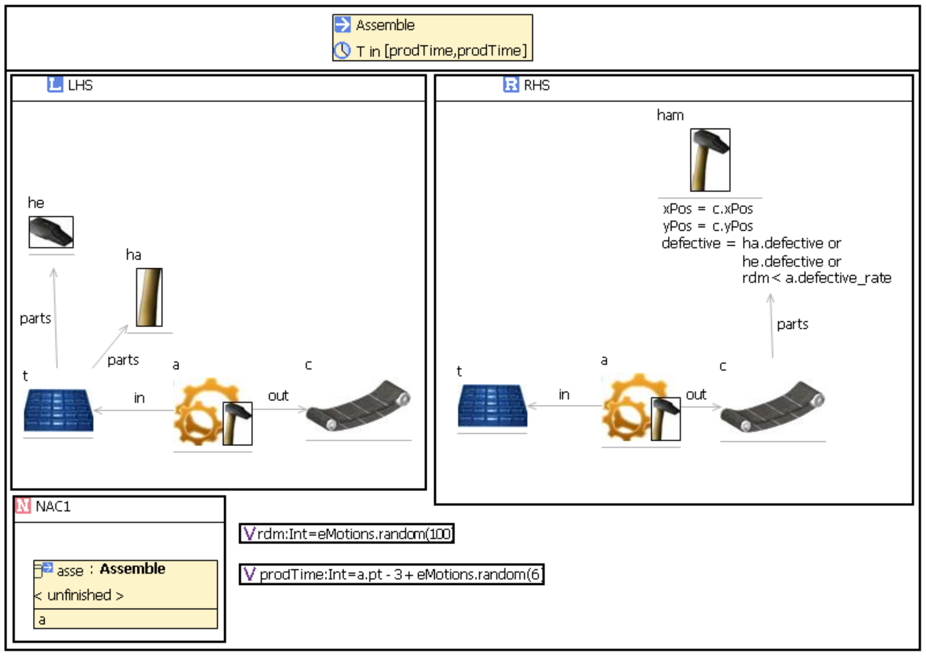

Assemble
T in [prodTime,prodTime]
LHS
he
ha
parts
parts
t
in
a
out
c
RHS
ham
xPos = c.xPos
yPos = c.yPos
defective = ha.defective or he.defective or rdm < a.defective_rate
parts
t
in
a
out
c
NAC1
asse : Assemble
< unfinished >
a
rdm:Int=eMotions.random(100)
prodTime:Int=a.pt - 3 + eMotions.random(6)

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

1.2 Rewriting Logic and Maude

Rewriting logic (RL) [?] is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. In RL, the state space of a distributed system is specified as an algebraic data type in terms of an equational specification (Σ, E), where Σ is a signature of sorts (types) and operations, and E is a set of equational axioms. The dynamics of a system in RL is then specified by rewrite *rules* of the form $t \to t'$, where t and t' are Σ-terms. This rewriting happens modulo the equations E, describing in fact local transitions $[t]_E \to [t']_E$. These rules describe the local, concurrent transitions possible in the system, i.e. when a part of the system state fits the pattern t (modulo the equations E) then it can change to a new local state fitting pattern t'. Notice the potential of this type of rewriting, and the very high-level of abstraction at which systems may be specified, to perform, e.g., rewriting modulo associativity or associativity-commutativity.

Maude [?,?] is a wide spectrum programming language directly based on RL. Thus, Maude integrates an equational style of functional programming with RL computation. Maude also supports the modeling of object-based systems by providing sorts representing the essential concepts of object (`Object`), message (`Msg`), and configuration (`Configuration`). A configuration is a multiset of objects and messages (with the empty-syntax, associative-commutative, union operator `__`) that represents a possible system state.

Although the user is free to define any syntax for objects and messages, several additional sorts and operators are introduced as a common notation. Maude provides sorts `Oid` for object identifiers, `Cid` for class identifiers, `Attribute` for attributes of objects, and `AttributeSet` for multisets of attributes (with `_,_` as union operator). Given a class C with attributes a_i of types S_i, the objects of this class are then record-like structures of the form

$$\texttt{<}\ O\ \texttt{:}\ C\ \texttt{|}\ a_1\texttt{:}v_1\texttt{,}\ \ldots\texttt{,}\ a_n\texttt{:}v_n\ \texttt{>}$$

where O is the identifier of the object, and v_i are the current values of its attributes (with appropriate types). See [?] for additional details on how object-oriented systems are represented in Maude, including explanations on how to represent inheritance, syntax for object-oriented modules, different forms of object communication, etc.

The following Maude definitions specify a class `Account` of bank accounts, with messages `withdraw` and `transfer` to operate with such bank accounts. The `Account` class is defined with a single attribute `balance`, of sort `Int`, representing the balance of an account. The `withdraw` message has two parameters, namely the addressee of the message and the amount of money to withdraw from the account. The `transfer` message will make the amount of money specified as its third argument to be transferred from the account given as first argument to the one given as second argument.

```
sort Account .
subsort Account < Cid .
op Account : -> Account .
op balance :_ : Int -> Attribute .
op withdraw : Oid Int -> Msg .
op transfer : Oid Oid Int -> Msg .
```

Rules `debit` and `transfer` below represent local transitions of the system that specify the behavior of bank accounts upon the reception of such messages. E.g., if an `Account` object receives a `withdraw` message and the amount of money to withdraw is smaller or equal than the balance of the account receiving the message, then the message is ‘consumed’ and the balance of the account is decremented is such an amount. Notice the synchronization of `Account` objects in the `transfer` rule.

```
vars A B : Oid .
vars BalA BalB M : Int .
```

```
crl [debit] :
  < A : Account | balance : BalA >
  withdraw(A, M)
  => < A : Account | balance : BalA - M >
  if BalA >= M .
crl [tranfer] :
  < A : Account | balance : BalA >
  < B : Account | balance : BalB >
  withdraw(A, M)
  => < A : Account | balance : BalA - M >
     < B : Account | balance : BalB + M >
  if BalA >= M .
```

Notice that, since the `__` operator is declared associative, commutative, and with identity element, we do not need to worry about the order in which objects and messages appear in the rules. And since rules describe local transitions, we do not need to worry about the rest of the objects and messages in the configuration either.

Well-formedness of objects may be automatically checked by Maude's typing system. For example, we can add declarations constraining `Account` objects:

```
sort AccountObject .
subsort AccountObject < Object .

var O : Oid .
var Bal : Int .

mb < O : Account | balance : Bal > : AccountObject .
```

Notice that with these declarations, an object `< O : Account | >` is a valid term of sort `Object`, but since the membership cannot be applied on it, it is not of type `AccountObject`.

2 Solution

We are presenting two solutions for the different tasks, one graphical solution using e-Motions, and another one using directly Maude. Each task is solved by defining respective DSLs, which share their abstract and concrete syntaxes. The abstract syntax used is the one provided at [1] — we will see below that some of the tasks have required extensions of this common syntax. The main differences between the DSLs defined for the different tasks is in their concrete behaviors describing what need to be done in each case, that is, the rewrite rules defining the behavior depends on the concrete task and its solution.

The e-Motions description of the different tasks is then transformed into a Maude specification and executed in Maude. We show how the formal tools available in Maude allow us to check the transformations carried out. Specifically, we will illustrate the use of Maude's reachability analysis capabilities to check that no undesired situation is reached along the execution.

Although the expressiveness of e-Motions is very welcome in complex problems, thanks to its capabilities to express problems visually, very intuitively and in a language very close to the problem domain, the overhead to be paid in cases like the ones at hand is too high. Specifically, the generality provided by its support for OCL expressions and time requirements, makes that the Maude code generated by the e-Motions tool is not as time performant as we would like. However, the general purpose rewrite-modulo engine at the core of Maude may also be used as a transformation language. Thus, together with the e-Motions solution we present a optimized Maude solution for each task.

As we will see below, the Maude version of the transformation closely follows the transformations provided in e-Motions, were all rewrite rules are *instantaneous* and expressions are solved directly by Maude built-in types instead of by the OCL interpreter [**?**]. Indeed, for problems as simple as the ones at hand, we will see that the representation distance between Maude and e-Motions to the problem domain would be almost as small, making both solutions very appropriate. Although a more in deep analysis of the problem at hand would most probably have allowed us to even improve the numbers obtained, we have preferred to keep the specification clear and intuitive.

2.1 Task 1

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

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*. Taking a parameter N as input model, *Task 1 DSL* generates a model containing synthetic data.

As it has been introduced in Section 1.1, the abstract syntax of a DSL is given in e-Motions by means of an Ecore metamodel. Since we model the solution of the task as a model that evolves until reaching its final solution, we take as metamodel the one provided beforehand in [1], which we call *Movies MM*, extended with a `Parameter` concept. This results in a so-called *Movies* MM.* The class `Parameter` has two integer attributes, namely `nP` and `nN`, which represent positive graphs and negative graphs, respectively, for the generation following Henshin graphs [**?**].

For the concrete syntax, Figure 2 shows how an image has been attached to each concept modeled in the Movies* MM.

The behavior of this *Task 1 DSL* is then given by means of two in-place transformation rules: `createPositive` and `createNegative`. Figure 3a shows the `createPositive` rule, which takes an object `p` of type *Parameter*, with *nP* attribute greater or equal than 0, and produces synthetic data conforming to the Henshing rules [**?**]. Figure 3b shows the `createNegative` rule, which is analogously defined.

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

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

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

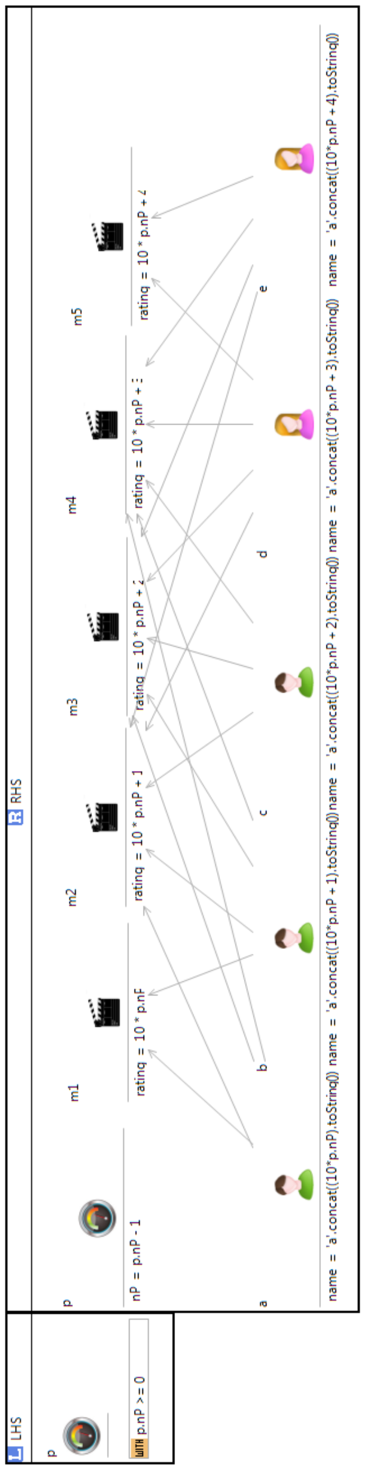
Please, note that this solution is really close to the problem specification in [2]. Figure 3 and [2, Figure 2], specifying the data generation, are almost the same. This demonstrates how close the solution by e-Motions is to the problem domain, and how convenient its graphical facilities are.

Maude version. Our Maude-based solution for Task 1 consists of an object-based Maude specification, consisting of two modules: the `MOVIES@MM` module defining the classes structure, and the `TASK1` module defining the rewrite rules to calculate the solution. As in the e-Motions solution, we have two rewrite rules: `createPositive` and `createNegative`. Listing 1.1 shows the `createPositive` Maude rule, that takes the `createPositive(s(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]. A similar rule generates the negative cases. Please notice that the Maude version is very much like the e-Motions version. In fact, the former is almost the textual version of the latter.

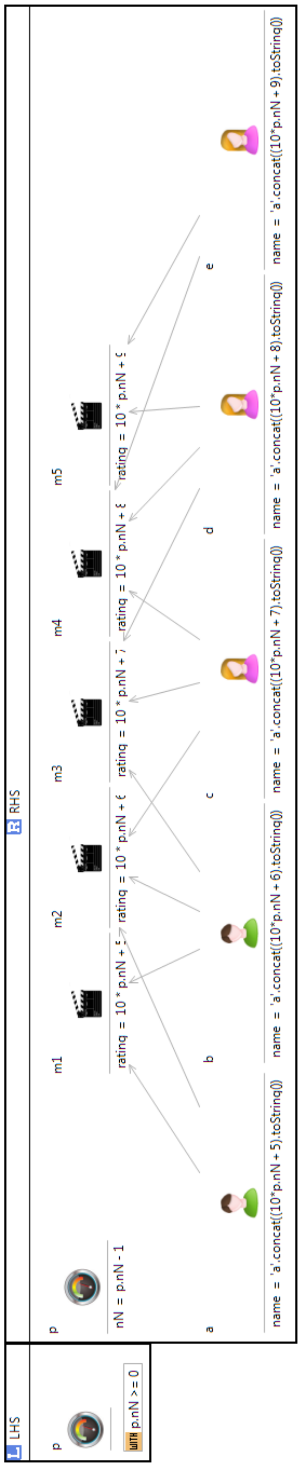
Listing 1.1: `createPositive` Maude rule.

```
rl [createPositive] :
  createPositive(s(N))
  freshOid(N')
=>
  createPositive(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)),
```


LHS
p
WITH p.nP >= 0
RHS
p
nP = p.nP - 1
m1
rating = 10 * p.nP
m2
rating = 10 * p.nP + 1
m3
rating = 10 * p.nP + 2
m4
rating = 10 * p.nP + 3
m5
rating = 10 * p.nP + 4
a
name = 'a'.concat((10*p.nP).toString())
b
name = 'a'.concat((10*p.nP + 1).toString())
c
name = 'a'.concat((10*p.nP + 2).toString())
d
name = 'a'.concat((10*p.nP + 3).toString())
e
name = 'a'.concat((10*p.nP + 4).toString())

(a) The `createPositive` rule.


LHS
p
WITH p.nN >= 0
RHS
p
nN = p.nN - 1
m1
rating = 10 * p.nN + 5
m2
rating = 10 * p.nN + 6
m3
rating = 10 * p.nN + 7
m4
rating = 10 * p.nN + 8
m5
rating = 10 * p.nN + 9
a
name = 'a'.concat((10*p.nN + 5).toString())
b
name = 'a'.concat((10*p.nN + 6).toString())
c
name = 'a'.concat((10*p.nN + 7).toString())
d
name = 'a'.concat((10*p.nN + 8).toString())
e
name = 'a'.concat((10*p.nN + 9).toString())

(b) The `negativePositive` rule.

Fig. 3: Task 1 rules.

	e-Motions		Maude	
N	Time (s)	# Rewrites	Time (s)	# Rewrites
1			0.0	68
2	0.0	4,910		
10	0.0	24,334		
20	0.0	48,614		
100	0.6	242,854		
1000	55.7	2,428,054	1.9	67,001
2000	395.0	4,856,054	12.7	134,001
3000			33.9	201,001
4000			64.1	268,001
5000			104.4	335,001
6000			109.6	402,001
7000			144.5	469,001

Table 1: Times for the e-Motions and Maude solutions to Task 1

```
                  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) .
```

Execution performance for both solutions. Table 1 shows the number of rewrites and execution times for both solutions. As explained above, the the execution times for the Maude specification obtained from the e-Motions definition grows very quickly. Notice that, although the number of rewrites grows linearly with respect to N, the time is exponential due to the infrastructure to deal with all the extra features in e-Motions. However, notice how the number of rewrites for the Maude solution grows linearly as well, but in this case the execution times grows more slowly, being able to handle problems of much bigger sizes.

On the correctness of the transformation Maude provides a whole formal environment where we can perform proofs of correctness of our solution. In addition to tools to verify the termination, confluence, etc. of our rewrite systems, Maude provides a reachability analysis tool and a model checker, which are particularly attractive for performing checks on the correctness of systems. Let us consider here the Maude `search` command, which allows us to explore the whole reachable state space, or up to a given depth, looking for states satisfying a given condition.

For instance, following the results given in [2], for some N, the above rules `createPositive` and `createNegative` create $20N$ objects, specifically, $10N$ movies, $5N$ actresses, and $5N$ actors. We could check that the solution found satisfies this condition, but we can do something more interesting by checking that no final reachable state fails to satisfy it. Of course, if the rewrite system is confluent and terminating the solution would be unique.

Given the `numOfMovies` operation, which takes an object configuration as input and returns the number of movies in it, we may look for those final states in which the number of movies will be different than $10N$ (being N the parameter of the operation `createExample`):

```
search createPositive(8) createNegative(8) freshOid(0)
  =>! C:Configuration
  such that numOfMovies(C:Configuration) =/= 10 * 8 .
```

The arrow `=>!` means that it search for final states, that is, states that cannot be further rewritten, starting from the given initial configuration, that satisfy the given condition. Maude returns no solution for the above code, that means all final states reached have exactly 10 movies:

```
No solution.
```

2.2 Task 2

Task 2 consists in finding all 'couples' from a given model, given that two persons are a 'couple' if they played together in at least three movies [2]. Couples are to be obtained using either from the model obtained in Task 1 or from the IMBd database [1]. Once again, we present solutions using e-Motions and Maude.

e-Motions-based solution The solution for this task is implemented with one single rule, `createCouple`, shown in Fig. 4. `Person` objects are shown using square shapes because `Person` is an abstract class and it does not have attached image. The `createCouple` rule models the creation of a couple by taking two persons and generating a couple with them. The rule has two conditions: a positive condition stating that *"the number of movies in the intersection between the movies of `per1` and `per2` is greater or equal than 3"*; and a negative condition, the `coupleHasNotBeenCreated` NAC, requiring that the couple does not exists yet.

Although very intuitive and simple, this solution is computationally very expensive. Notice that the number of matchings in the LHS of the rule is combinatorial, leaving all the task to the evaluation of the conditions to accept or discard the couples.

We have implemented another solution in which we limit the number of matchings using a very simple algorithm: For each person, we iterate on the rest of persons looking for couples. With this algorithm, the number of persons to match as candidate couple decreases significantly. To model this solution,

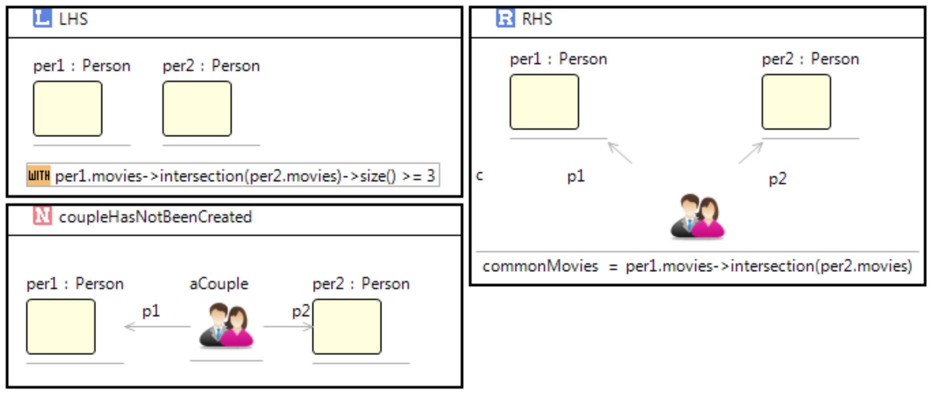


Fig. 4: `createCouple` rule.

N	Time (s)	# Rewrites
2	0.7	524,781
10	46.7	19,453,091
20	660.1	161,741,321

Table 2: e-Motions times for Task 2 First Version.

we extend the metamodel and its concrete syntax with a so-called `Collection` concept. This class has three attributes, namely `people`, with the set of persons to be handled, `fixedPerson`, to iterate on each person, and `peopleDealtWith`, to keep the persons already considered to create a couple with the current 'fixed person'. Figures 5-9 show the rules specifying this solution:

– Rule `initialRule` in Figure 5 initializes the collection object assigning the set of all actors and actresses to its `people` attribute (the other attributes get default values). This rule is only fired if there is no collection object in the model.
– Rule `fixPerson` in Figure 6 takes a person from the `people` set and takes it as `fixedPerson`. Notice that `p` is removed from the `people` set.
– Rules `doingCouples-AreCouple` and `doingCouples-AreNotCouple` in Figures 7 and 8 take a person from the set `people` and the `fixedPerson` and make a couple if the number of movies they share is greater or equal than three. In both cases the person considered is passed to the `peopleDealtWith` set.

When all persons has been considered as couple of the current `fixedPerson`, the `nextPerson` rule, takes the `peopleDealtWith` as new `people` set.

Maude-based solution. Again, we specify directly in Maude both solutions. As for Task 1, the solutions match very closely their e-Motions counterparts.

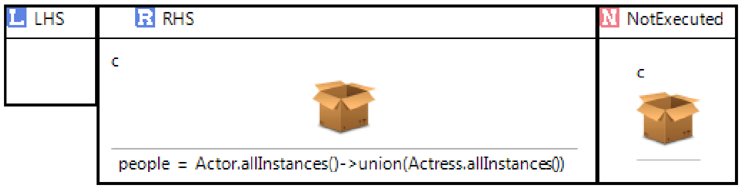

LHS
RHS
NotExecuted
c
c
people = Actor.allInstances()->union(Actress.allInstances())

Fig. 5: `initialRule` rule.

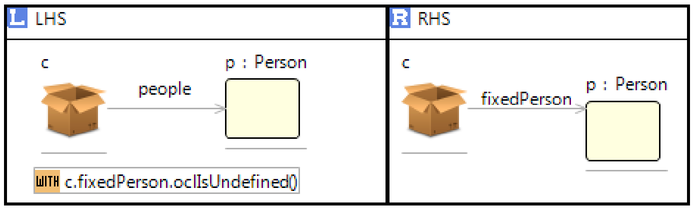


Fig. 6: `fixPerson` rule.

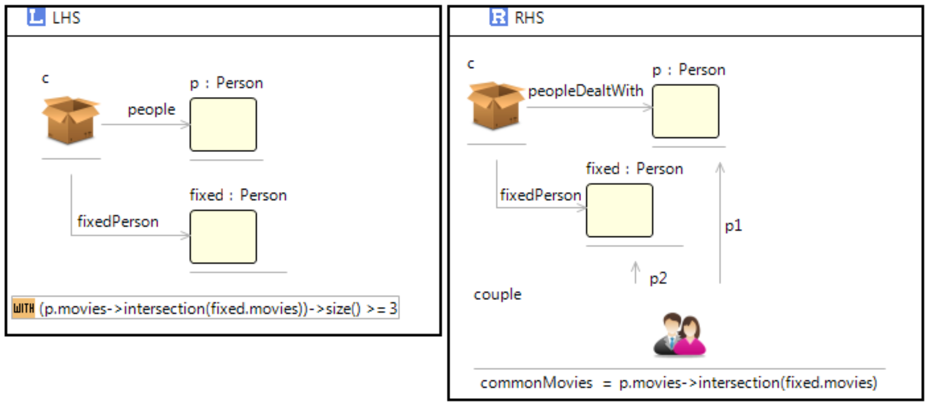


Fig. 7: `doingCouples-AreCouple` rule.

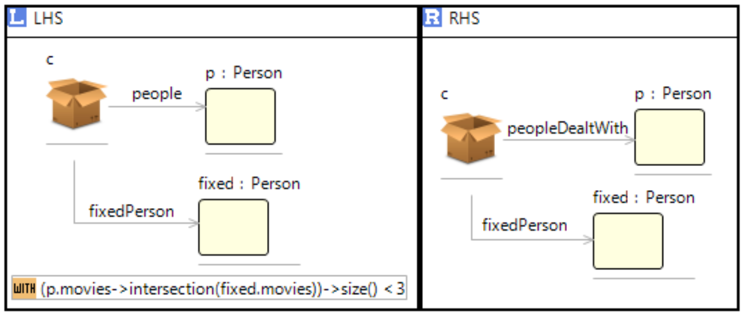


Fig. 8: `doingCouples-AreNotCouple` rule.

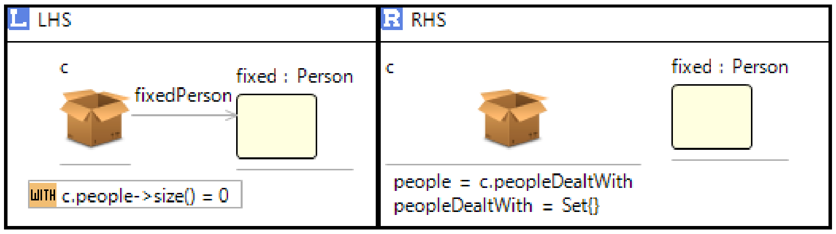


Fig. 9: `nextPerson` rule.

The Maude solution for the first alternative solution to Task 1 is shown in Listing 1.2. The rules takes two persons and creates a new couple if they share three movies and such couple has not been previously created. Some numbers for its execution are shown in Table 3.

Listing 1.2: `createCouples` Maude rule.

```
crl [findCouples] :
 { freshOid(N) findCouples
   < O1 : V1:Person | movies : MS1, Atts1 >
   < O2 : V2:Person | movies : MS2, Atts2 >
   Conf }
=>
 { freshOid(s(N)) findCouples
   < O1 : V1:Person | movies : MS1, Atts1 >
   < O2 : V2:Person | movies : MS2, Atts2 >
   < N : Couple |
           commonMovies : (intersection((MS1), (MS2))),
           p1 : O1, p2 : O2 >
   Conf }
if | intersection((MS1), (MS2)) | >= 3
/\ not coupleInConf(C, Conf) .
```

N	Time (s)	# Rewrites
1	0.0	8,680
5	0.5	1,343,000
10	5.0	11,020,000
20	66.3	89,276,000
30	314.0	302,568,000

Table 3: Maude times for Task 2 First Version.

As for e-Motions, the second solutions consists of several rules. In this case, our collection object is model by an operator {_}{_}{_}{_}{_}{_} representing an object with six attributes as its e-Motions counterpart: the first argument takes the starting argument, the second the people set, the movies set, the dealt people, the fixed person on which to iterate, and the result configuration. We show in Listing 1.3 the code for the `doingPairs` rule. In the case of Maude, we only need one rule to consider the positive and negative cases.

Listing 1.3: `doingCouples` Maude rule.

```
rl [doingPairs] :
  < { none }
    { < O1 : V1@Person | movies : MS1, Atts1 >  C1 }
    { C2 }
    { C3 }
    { < O2 : V2@Person | movies : MS2, Atts2 > }
    { freshOid(New) C4 } >
=>
  < { none }
    { C1 }
    { C2 }
    { < O1 : V1@Person | movies : MS1, Atts1 > C3 }
    { < O2 : V2@Person | movies : MS2, Atts2 > }
    { if | intersection(MS1, MS2) | >= 3
      then < New : Couple | p1 : O1, p2 : O2,
                   commonMovies : intersection(MS1, MS2),
                   avgRating : 0.0 >
           freshOid(s New)
      else freshOid(New)
      fi
      C4 } > .
```

2.3 Task 3

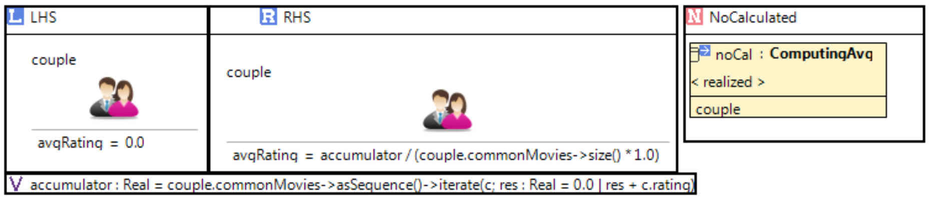

LHS
couple
avgRating = 0.0
RHS
couple
avgRating = accumulator / (couple.commonMovies->size() * 1.0)
NoCalculated
noCal : ComputingAvg
< realized >
couple
accumulator : Real = couple.commonMovies->asSequence()->iterate(c; res : Real = 0.0 | res + c.rating)

Fig. 10: `computingAvgRating` rule.

e-Motions-based solution.

N	Time (s)	# Rewrites
2	0.0	4527
10	2.1	891432

Table 4: e-Motions times for Task 23.

Listing 1.4: Maude rule for Task 3 solution.

Maude-based solution.

```
crl [avgRating] :
  { < M : Couple | commonMovies : MovieSet , avgRating : 0.0, Atts1 >
    couplesCalculated((Couples)) C
  }
=>
  { < M : Couple | commonMovies : MovieSet ,
       avgRating : (sumAllRatings(MovieSet , C) / float(| MovieSet |)),
       Atts1 >
       couplesCalculated((M, Couples)) C
  }
if not(M in Couples) .
```

N	Time (s)	# Rewrites
100	1.5	21800
200	6.3	43600
300	14.9	65400
400	29.7	87200

Table 5: Maude times for Task 23.

3 Conclusions

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.
3. Rivera, J.E., Durán, F., Vallecillo, A.: On the behavioral semantics of real-time domain specific visual languages. In: WRLA. pp. 174–190 (2010)
4. Troya, J., Vallecillo, A.: Towards a Rewriting Logic Semantics for ATL. In: ICMT. pp. 230–244 (2010)