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 [?], 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 [?] 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, 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 **??** 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 [**?**].

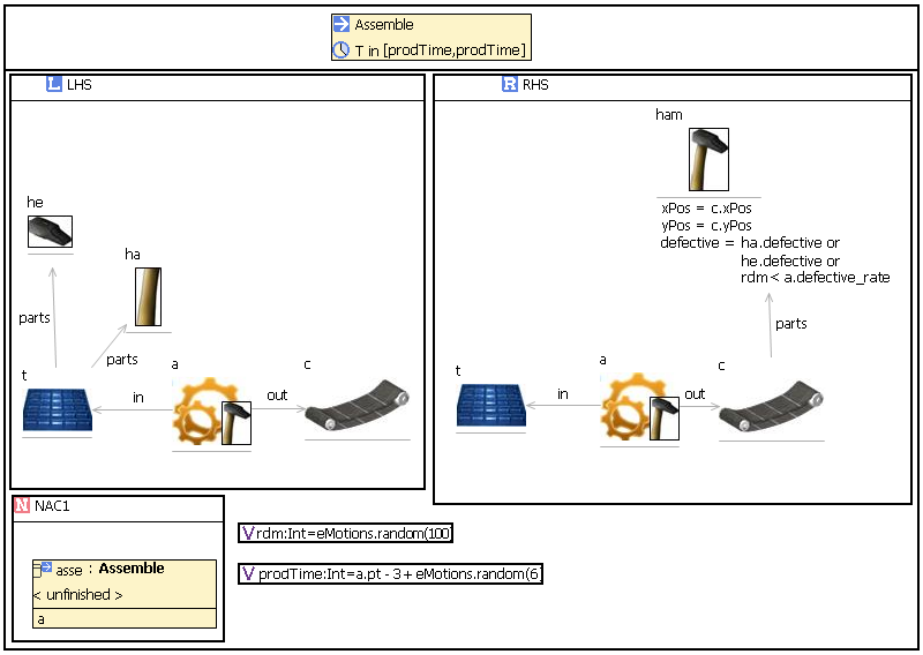


Fig. 1: e-Motions `Assemble` rule.

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

~~Since e-Motions supports in a very intuitive, user-close and visual way, we show how the IMDb problem [?] 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.~~

We show how the IMDb problem can be solved using e-Motions tool. Each task is figured out using the definition of a DSL, which shares, in general, the abstract and concrete syntax. In fact, the abstract syntax is given in advance in [**?**]. The rewrite rules defining the behavior depends on the concrete task and its solution. All rewrite rules are *instantaneous*, since the case to be solved do not have time requirements.

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 performant as desired. Together with the e-Motions solution we present a optimized Maude solution for each task.

TO-DO: terminar esta sección:

e-Motions-based solution which are visual, very intuitive and very close to the problem domain are shown against Maude-based solution. Although Maude-based solutions are also very expressive, the fact that

Both e-Motions-based and Maude-based solutions are very expressive and very close to the problem domain

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.

Task 1 comprises the generation of synthetic models (conforming the movie database metamodel [**?**]) from an input parameter $N \geq 0$. In the following we present an e-Motions based solution and 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 **??**, the abstract syntax of a DSL is given in e-Motions by means of a Ecore metamodel. In fact, this metamodel is provided beforehand in [**?**]. We call this metamodel *Movies MM*. However, the *parameter N* has to be modeled in some way, since in e-Motions the state is just a model. Hence, a new concept call `Parameter` has been added to Movies MM. This results in a so-called *Movies* MM*. The class `Parameter` has two integer attributes `nP` and `nN`, positive graphs and negative graphs, due to data generation following Henshing graphs [**?**] is divided into positive and negative cases.

For the concrete syntax, Fig. **??** 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 **??** shows the `createPositive` rule, which takes an object `p` of type *Parameter* with *nP* attribute greater or equal than 0 and, after the rule application, synthetic data conforming to the Henshin rules [**?**] are created. Fig. **??** 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 `Parameter` with its two attributes `nP` and `nN` properly set. This model is used as initial model of the execution.

This solution is really close to the problem specification [**?**]. Both Fig. **??** and Fig. 2 in [**?**] specifying the data generation are almost the same. This solution demonstrates how close is the solution to the problem domain, and how user-friendly is e-Motions.

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 **??** 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 [**?**].

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

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: Tiempos

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 [**?**]. Once again we present the e-Motions and the Maude solution.

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

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

e-Motions-based solution Fig. **??** 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. **??** shows one of the rules specified for this solution,[1] namely those that creates new couples. The box is the concrete syntax for the `Collection` concept.

TO-DO: add numbers

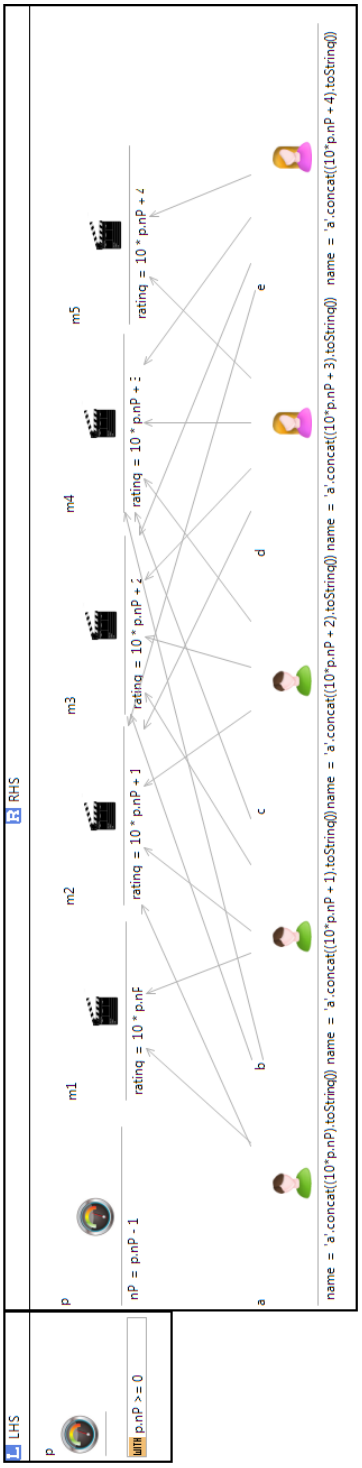
Maude-based solution

3 Conclusions

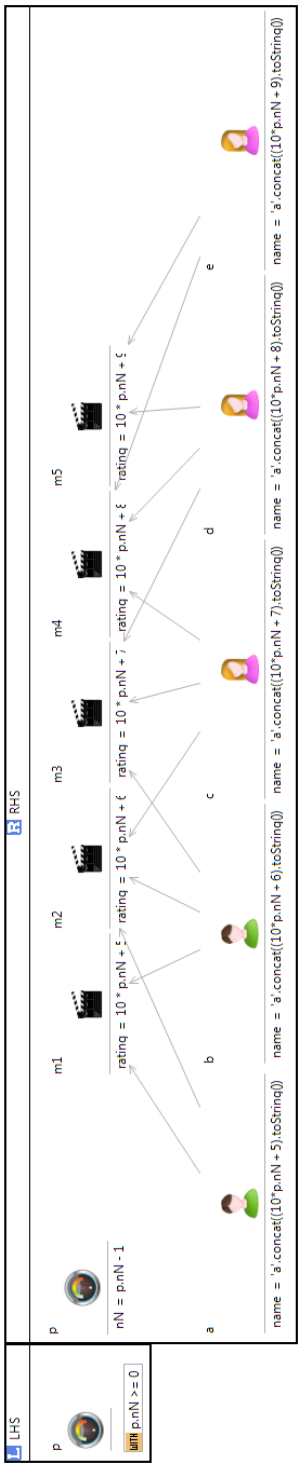
References

1. Horn, T.: IMDB2EMF, `https://github.com/tsdh/imdb2emf`

[1] 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.

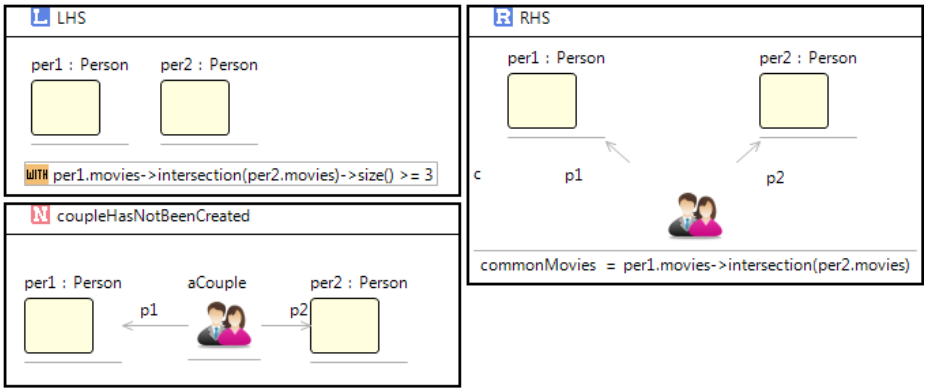


Fig. 4: `createCouple` rule.

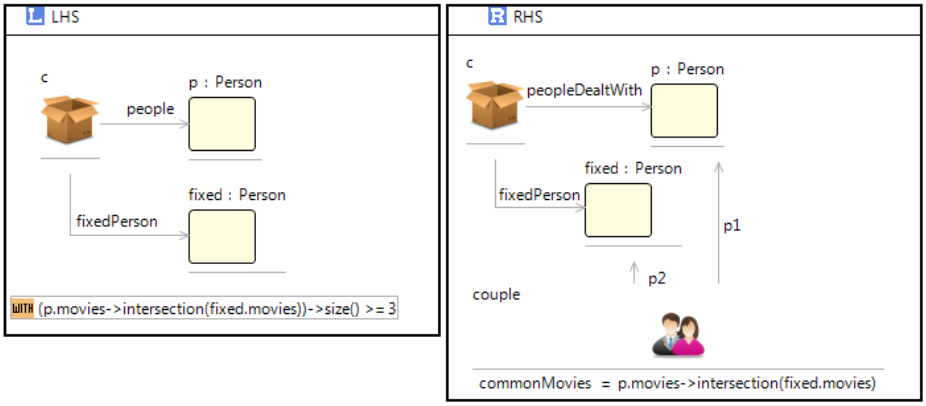


Fig. 5: `doingCouples` rule.

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)