e-Motions: A graphical approach for modeling time-dependent behavior of domain specific languages

Article · January 2008

CITATIONS
13

READS
169

3 authors, including:

Francisco Durán
University of Malaga
178 PUBLICATIONS **3,994** CITATIONS
SEE PROFILE

Antonio Vallecillo
University of Malaga
253 PUBLICATIONS **4,112** CITATIONS
SEE PROFILE

e-Motions: A Graphical Approach for Modeling Time-Dependent Behavior of Domain Specific Languages

José E. Rivera, Francisco Durán and Antonio Vallecillo
University of Málaga, Spain
{rivera, duran, av}@lcc.uma.es

Abstract

Domain specific languages (DSLs) play a cornerstone role in Model-Driven Software Development for representing models and metamodels. DSLs are usually defined only in terms of their abstract and concrete syntaxes, something that may hamper the development of formal analysis and simulation tools. In-place model transformations provide an intuitive way to complement metamodels with behavioral specifications. In this paper we extend in-place rules with a quantitative model of time and with mechanisms that allow designers to state action properties, facilitating the design of real-time complex systems. This approach avoids making unnatural changes to the DSL metamodels to represent behavioral and time aspects. The resulting specifications can then be translated into different semantic domains, such as Real-Time Maude, making them amenable to simulation and formal analysis. Finally, we present the graphical modeling tool we have built for visually specifying these timed specifications.

1. Introduction

Domain specific languages (DSLs) play a cornerstone role in Model-Driven Engineering (MDE) for representing models and metamodels. DSLs are normally defined in terms of their abstract and concrete syntaxes. The abstract syntax is defined by a metamodel, which describes the concepts of the language, the relationships between them, and the structuring rules that constrain the combination of model elements according to the domain rules. The concrete syntax specifies how the domain concepts included in the metamodel are represented, and is usually defined as a mapping between the metamodel and a textual or graphical notation. This metamodeling approach enables the rapid and effective development of languages and their associated tools (e.g., graphical or textual model editors).

Explicit and formal specification of model semantics has not received much attention from the MDE community until recently, despite the fact that this issue is particularly important in safety-critical real-time and embedded system domains, where semantic ambiguities may produce conflicting results across different tools. Furthermore, the lack of explicit behavioral semantics strongly hampers the development of simulation and formal analysis tools.

One way of specifying the dynamic behavior of a DSL is by describing the evolution of the modeled artifacts along some time model. In MDE, this can be naturally done using model transformations supporting in-place update [3]. The behavior of the DSL is then specified in terms of the permitted actions, which are in turn modeled by the model transformation rules. However, only a few of the current approaches deal with time-dependent behavior (see Section 6). Timeouts, timing constraints and delays are essential concepts in these domains, and therefore they should be explicitly modeled. Besides, current approaches do not allow users to model action-based properties, making them inexpressible and forcing unnatural changes to the system specification [7].

In this paper we extend standard in-place rules so that time and action statements can be included in the behavioral specifications of a DSL. We provide a graphical framework aimed at defining behavioral specification models, which can be fully integrated in MDE processes. Its precise behavioral semantics are given by mappings to different semantic domains. In particular, we show how a mapping between these specifications and Real-Time Maude [9, 2] can be defined, making them amenable to simulation and different kinds of formal analyses [2]. This paper extends our initial proposal presented in [12] to include variables, ongoing actions, periodicity and rule execution modes (eager and lazy), which are essential aspects to capture some critical properties of real-time systems.

After this introduction, Section 2 shows how in-place transformation rules are extended to model time-dependent behavior and action-based properties. Then, Section 3 introduces a motivating example modeled with the graphical modeling tool we have built to support these specifications, which is presented in Section 4. Section 5 gives an overview

of a semantic mapping from timed behavioral specifications to Real-Time Maude. Finally, Section 6 compares our work with other related proposals, and Section 7 draws some conclusions and outlines some future research activities.

2. Extending In-place Transformations Rules

There are several approaches that propose in-place model transformations to deal with the behavior of a DSL, from textual to graphical (see [10] for a comprehensive survey). This approach provides a very intuitive and natural way to specify behavioral semantics, close to the language of the domain expert and the right level of abstraction [4].

These transformations are composed of a set of rules. Each rule represents a possible *action* of the system. Rules are of the form $l : [\text{NAC}] \times \text{LHS} \rightarrow \text{RHS}$, where l is the rule label (name); LHS (Left Hand-Side) and RHS (Right Hand-Side) are model patterns that represent certain states of the system, and NAC is a set of optional model patterns that represent Negative Application Conditions that forbid applying the rule if one of these patterns is found in the model. The LHS and NAC patterns express preconditions for the rule to be applied, whereas the RHS contains the rule post-conditions. LHS and NAC patterns may include conditions. Thus, a rule can be applied (i.e., *triggered*) if an occurrence (or match) of the LHS is found in the source model, and none of the NAC patterns is found. Generally, if several matches are found, one of them is non-deterministically selected and applied, producing a target model where the match is substituted by the RHS (the rule *realization*). The model transformation proceeds by applying the rules in a non-deterministic order, until none is applicable (although this behavior can be usually modified by some execution control mechanisms).

Most existing in-place transformation approaches do not allow modeling the notion of time in a quantitative way, or allow it by adding some kind of clocks to the DSL metamodel. This latter approach forces designers to modify the DSL metamodel to include time-dependent aspects, and does not forbid the design of rules that may lead to time-inconsistent sequences of states (e.g., decreasing the time), since clocks are handled as common objects (see Section 6).

Figure 1 shows the *Behavior Metamodel*, which describes the main concepts of our approach to model time-dependent behavior. The novelty in this metamodel is the addition of time-related attributes to rules (to represent duration, periodicity, etc.), and the inclusion of the ActionExec metaclass, whose instances represent action executions. MetamodelGD and ClassGD metaclasses are used for defining the graphical concrete syntax os the DSL (see Section 4). Other concepts, such as the single and double pushout formalizations of the transformations, and the non-injectiveness of the rules, are handled in the same way as in common graph transformation approaches [10], although adapted to the tree-structure of Eclipse models [1]. The following paragraphs describe the main concepts of our approach with more detail. In this paper we have considered only discrete time, although dense time [9] is also supported.

Atomic actions. One natural way to model time-dependent behavior quantitatively consists of extending the rules with the time they consume, i.e., by assigning to each action the time it takes. Thus, we define *atomic rules* as in-place transformation rules of the form $l : [\text{NAC}] \times \text{LHS} \xrightarrow{t} \text{RHS}$, where t expresses the duration of the action modeled by the rule, in some time granularity.

Atomic rules can be triggered whenever an occurrence (or match) of the LHS is found in the source model, and none of the NAC patterns is found. Then, the action specified by the rule is scheduled to be applied after t time units. At that moment of time, the rule is applied by performing the attribute computations and substituting the match by its RHS.

Ongoing actions. We also count on rules to model actions that are continuously progressing. Think for instance of an action that models the consumption of a phone battery, whose level decreases continuously with time. An ongoing action progresses with time while the rule preconditions (LHS and not NACs) hold, until the time limit of the action is reached (given by attribute maxDuration of this kind of rules, see Figure 1). Ongoing rules are executed until any scheduled event happens; at that time, the values of the attributes in the RHS of the rule are computed, the state of the system updated, and the rule is triggered again if its preconditions still hold. Note that rule preconditions act as invariants for this kind of actions.

Of course, we could try to model ongoing actions using atomic actions, by somehow "discretizing" them using the minimum time elapse — but we would be losing some interesting expressive properties, and probably become quite inefficient. Analogously, atomic actions could also be modeled using ongoing actions: one instantaneous rule that models the beginning of the action, another rule that decreases an added timer to represent the duration of the rule, and a second instantaneous rule that substitutes the match by the RHS. Note that in this case, the DSL metamodel needs to be modified to include the corresponding timer.

Action executions. In standard in-place transformation approaches, model patterns (LHS, RHS and NACs) are only defined in terms of system states. This is a strong limitation in those situations in which we need to refer to the actions currently executing, or to those that have been executed in

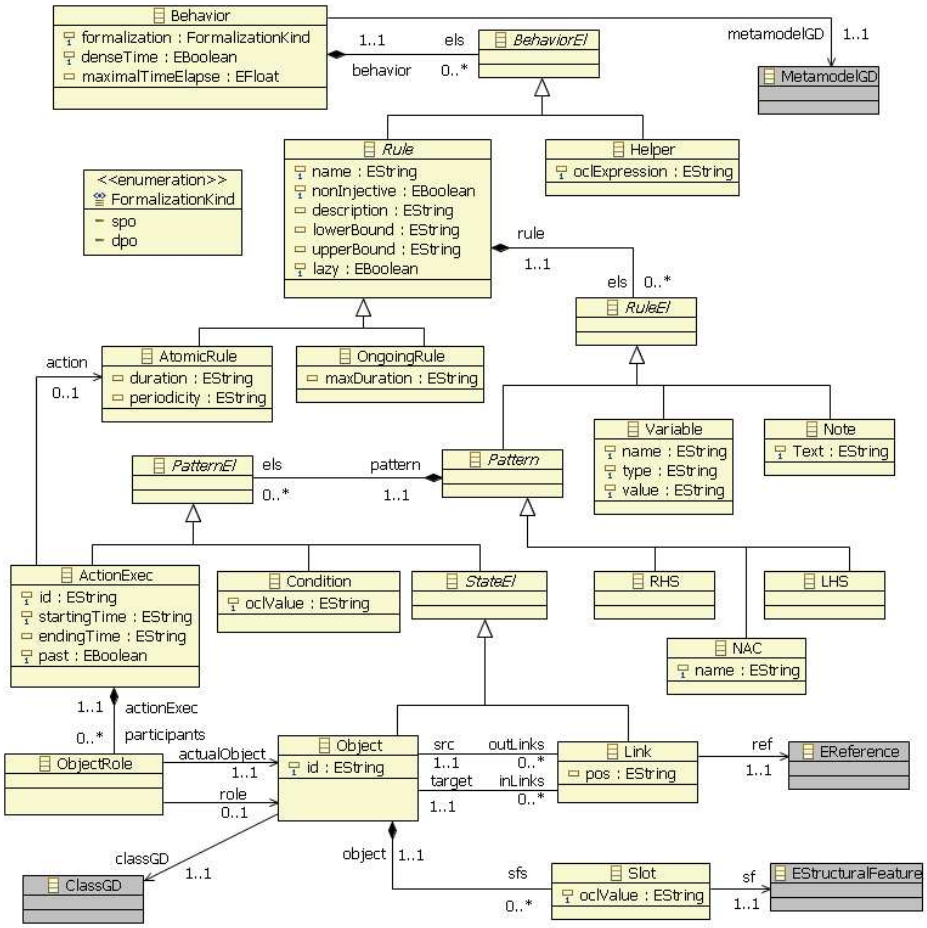

Figure 1. The Behavior metamodel.

the past. For example, we can be interested in knowing if an object is currently performing a given action, in order not to allow it to perform another. In general, the inability of being able to model and deal with action occurrences hinders the specification of many useful action properties, unless some unnatural changes are introduced in the system model — such as extending the system state with information about the actions currently happening (cf. [7]).

In order to be able to model both state-based and action-based properties, we propose extending model patterns with **action executions** that model action occurrences. These action executions represent atomic rule executions that are currently happening or that were previously performed. Action executions specify the type of the action (i.e., the name of the atomic rule), the identifier of the action execution, its starting and ending time, and the set of objects involved in the action. These objects are specified by *object mappings*, which are sets of pairs ($o \rightarrow r$). Each pair identifies the object that participates in the action (o) and one of the roles it plays in the rule (r). For instance, the *Talk* rule in Section 3 (see Figure 7) defines four roles: two phones ($p1$ and $p2$), one connection (c), and the global clock (cl).

In an action execution specification it is also possible to leave empty (i.e., unspecified) the name of the rule and the role of an object. This provides a very useful mechanism when we want to check if an object is participating in *any* action, or if an object is playing *any* role in a given action.

Please note that in our proposal elements can perform, or be engaged in, several actions at a time because in-place rules can be applied always that an occurrence of the LHS pattern and none of the NAC patterns are found in the model. This is very useful to model realistic situations in a natural way, e.g., a phone can be engaged in a conference call and also moving. The use of action executions in the LHS or NAC patterns of the rules provides a powerful mechanism to prevent this kind of behavior (in case we want some kind of elements to realize only one action at a time) or, conversely, to enforce it when required (if we want one element to perform one action while it is already performing another one). Examples are presented in Section 3.

Finally, another important aspect of incorporating action executions as first-class citizens of our rules is that we can reason about them, being able for instance to search for undesirable action occurrences, use model-checking techniques to look for invalid order of actions executions, etc.

Rule execution. We distinguish different possibilities: **eager** rules execute as soon as possible; **lazy** rules start executing in a non-deterministic moment in time, but always within their allowed interval (such an interval is defined by rules' lower and upper bounds); and **scheduled** actions which are modeled by eager rules whose lower and upper bounds coincide with the moment in time at which the rules are scheduled.

Global time elapse. We provide a special kind of object, named Clock, that represents the current global time elapse. A unique and read-only Clock instance is provided by the system (i.e., it does not need to be created by the user) to model time elapse through the underlying platform. This allows designers to use the Clock in their timed rules to get the current time (using its attribute time) to model, e.g., time stamps. Provided that the clock behavior cannot be modified, users cannot drive the system to time-inconsistent sequences of states (even unwillingly).

Periodicity. Another essential aspect for modeling time-dependent behavior is periodicity. Atomic rules admit a parameter that specifies an amount of time after which the action is periodically triggered (if the rule preconditions hold, of course). Eager rules are tried to be triggered at the beginning of the period, while lazy rules can be executed during the whole period (but only once per period, and whenever its duration does not exceed it).

Helpers and variables. Helpers (expressed as OCL operations) are library functions which are useful for specifying some of the rule parameters, or object attributes. They can be used throughout the behavioral specifications. Function random(b : Int) : Int, which is provided by the system and generates a pseudo-random number between zero and the given bound b, is an example of such helpers.

There are situations in which variables are also needed in rules. Think for instance of a rule whose duration is computed using the random helper and that we need to use such a value in several places in the rule. The use of a variable that is computed only once during the rule execution solves the problem that we would face if we had to compute this value several times (e.g., random returns a different value each time it is invoked). The context of a user-defined variable is the rule in which it is defined, and its value is computed when the rule is triggered.

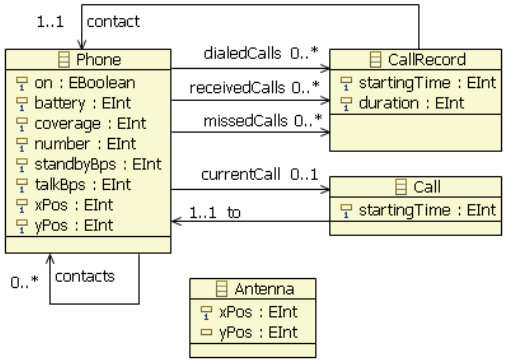

Figure 2. Mobile Phone Network Metamodel.

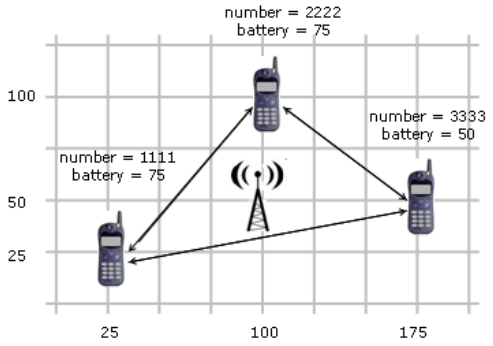

Figure 3. A mobile phone network example.

3. A Mobile Phone Network Example

For illustration purposes, let us introduce a modeling language for mobile phone networks (MPNs), which will serve as the motivating example to show the capabilities of our approach. The MPN metamodel is shown in Figure 2. A MPN is composed of cell phones and antennas. Antennas provide coverage to cell phones, depending on their relative distance. A cell phone is identified by its number, and can perform calls to other phones of its contact list. Calls are registered. Phone attributes standbyBps and talkBps represent the battery consumption per second while being in standby or talking, respectively.

Figure 3 shows a MPN example using a visual concrete syntax. The model consists of three cell phones and one antenna. The position of each element is dictated from its position in the grid. All phones are initially off, and their contacts are represented by arrows between them.

Atomic actions. Figure 4 shows timed rules *SwitchOn* and *BatteryOff*. When a phone is off, it can be switched on if it has enough battery for it. This action takes ten seconds. Whenever a phone is on and it has no battery, it is

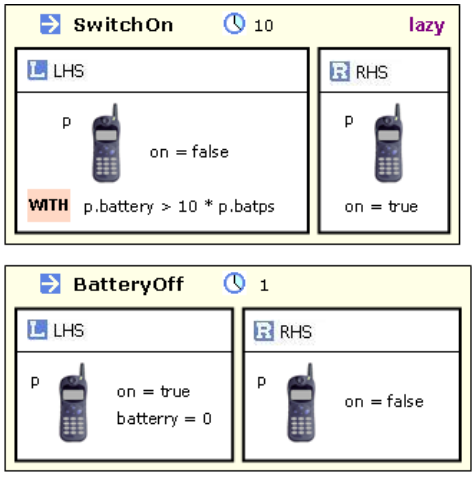

Figure 4. The *SwitchOn* and *BatteryOff* rules.

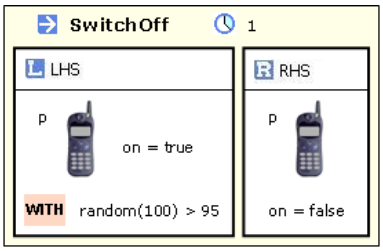

Figure 5. The *SwitchOff* atomic rule.

switched off as modeled by rule *BatteryOff*, whose duration is one second. Both rules are atomic, i.e., once the precondition is satisfied, the corresponding match is substituted by the RHS pattern after t units of time. Action *SwitchOn* is modeled by a lazy rule, which means that is not forced to be applied whenever its LHS pattern is found in the model: we allow phones to be switched on at a non-deterministic moment in time. Of course, phones can also be switched off voluntarily. The *SwitchOff* rule models this situation (Figure 5). Since this is not a frequent action, we give the rule a low probability of execution using operation random.

Action executions. Figures 6 and 7 show three atomic rules that model the behavior of phone calls. Lazy rule *MakeCall* describes the initiation of a call from a cell phone to one of its contacts. For this purpose, both phones must be on and have coverage. The four NAC patterns forbid the execution of the rule whenever one of the phones is participating in another (incoming or outgoing) call.

Once the call is initiated, it can be picked up to start a talk (*Talk* rule) or just ignored (*MissedCall* rule). If the call is picked up, a conversation will take place for talkTime time units. The value of talkTime is defined as a pseudo-random value (random(100)), and will be computed, as every variable value, when the rule is triggered. At the end of the talk, the call is registered in both phones (as a dialed call in phone p1 and as a received call in phone p2) including the duration of the call (talkTime) and its starting time. In our case, we have considered that the starting time of a received call is the moment at which the call is picked up. Note that the clock time in the RHS pattern of the rule will refer to the moment of the finalization of the rule, since attribute computations are performed at that time.

If the call is ignored (*MissedCall* rule), it will be registered as a dialed call in the p1 phone and as a missed call in the p2 phone. The *Talk* and *MissedCall* rules are lazy, and therefore they can be executed at a non-deterministic moment in time. However, only one of them will be executed over a call: if the *MissedCall* rule is started before the *Talk* rule, the call is deleted from the model and therefore no more rules can be applied over it. But, if the *Talk* rule is started before the *MissedCall* rule, the latter cannot be applied afterwards, as specified in its NAC pattern by using an action execution (on which we explicitly specify that the same call c cannot be participating in the action *Talk* with the c role).

Periodicity. Figure 8 shows the *Coverage* rule, which specifies the way in which coverage changes. Coverage is updated every ten seconds (see the loop icon at the header of the *Coverage* rule). Each cell phone is covered by the closest antenna: as specified in its NAC pattern, the rule cannot be applied if there exists another antenna closer to the phone. To compute the distance between the two objects, we have the following helper:

```
context Antenna::distance(p : Phone): Integer
  body: (self.xPos  p.xPos).abs() + (self.yPos  p.yPos).abs()
```

Helper invocations in LHS and NAC patterns are computed at the triggering of actions, while helper invocations in RHS patterns are computed in their finalization. Thus, note that the distance between the antenna and the phone may vary on these two different moments of time. If the same value was needed, a *variable* could be used.

Exceptions in actions. So far, we have modeled a simplified behavior of mobile phone network, avoiding exceptional cases. How does a phone behave when it runs out of battery in the middle of a conversation? What happens when the phone gets out of coverage? In our approach, atomic actions are triggered if their preconditions (LHS and not NACs) are met, and their effects take place once they finish (after their corresponding duration). Nothing is assumed about what happens while the action is being exe-

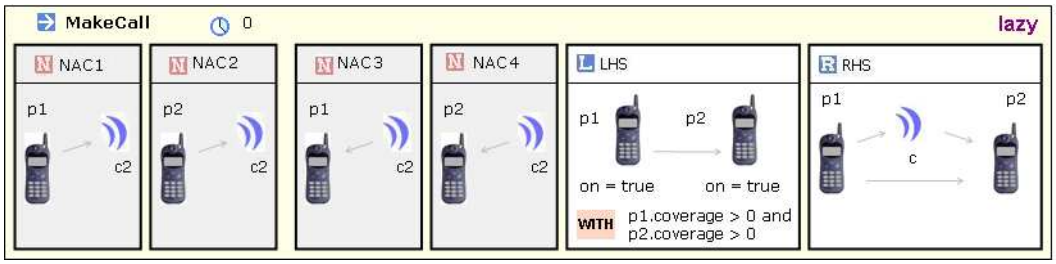

Figure 6. The *MakeCall* atomic rule.

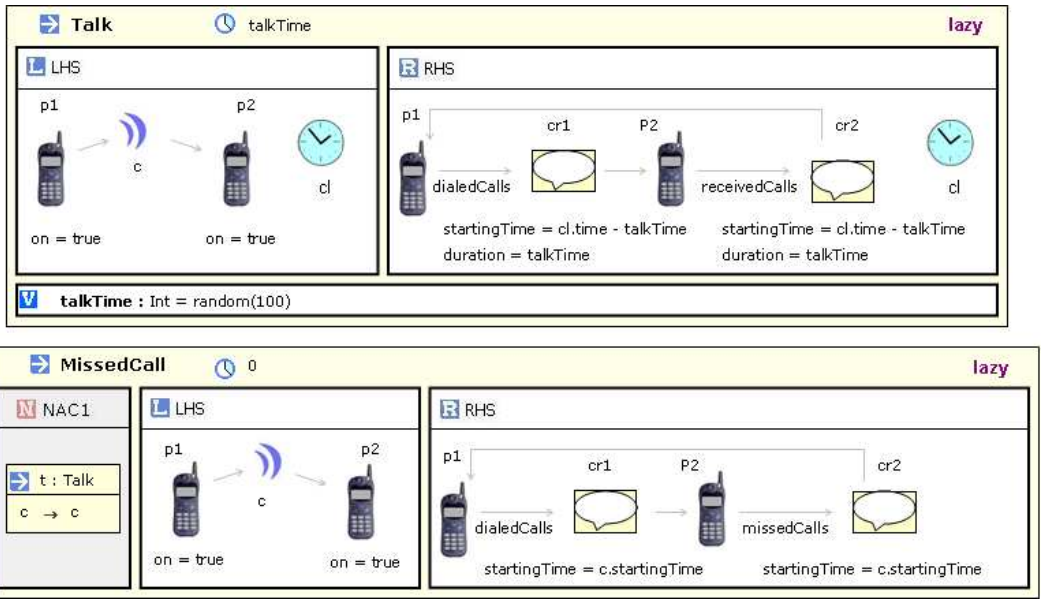

Figure 7. The *Talk* and *MissedCall* atomic rules.

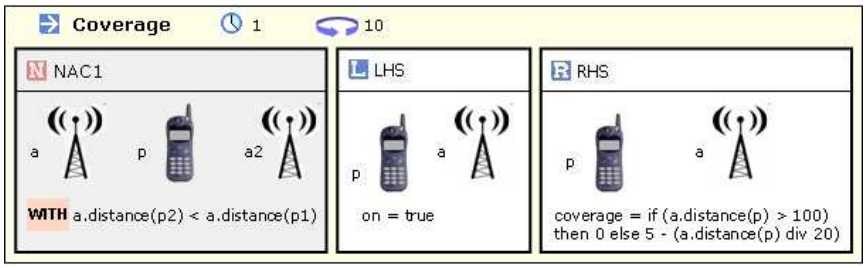

Figure 8. The *Coverage* atomic rule.

cuted. However, there are situations in which we want to make sure that something happens (or does not happen) during the action execution. In these cases, action executions can also be used for interrupting atomic actions. We can add new rules that model actions cancelations by deleting their corresponding action executions, i.e., by including them in the rule's LHS pattern but not in the RHS pattern. Their corresponding effects will be then defined in the rule's RHS pattern.

Consider, for instance, the *OffInTalk* and *OffInCall* atomic rules in Figure 9, which model the behavior of a phone when it is switched off (either voluntarily or if it runs

out of battery) in the middle of a call. The *OffInTalk* rule is applied whenever two phones are having a talk and at least one of the phones is found to be off. In this case, the talk action is canceled and the call registered in both phones. Rule *OffInCall* models a similar behavior, but in the case that the call has not been picked up yet. If applied, the call is canceled (i.e., deleted from the model) and the corresponding missed call registered. In both rules, at least one phone registers the call once it is off.[1]

CallerOutOfCoverage and *CalleeOutOfCoverage* actions in Figure 10 model the interruption of a talk whenever one of the phones in a call is out of coverage for five seconds. If such a case, the talk is finalized and registered in both phones. In both rules, we want them to be applied if and only if the phone is out of coverage during the whole rule execution (five seconds). However, the LHS pattern acts only as the rule precondition. In order to enforce it to be the action invariant, too, we can simply add new instantaneous rules that interrupt the action (by deleting its corresponding action execution) if the invariant is violated. For instance, the *CallerCoverageRecovering* and *CalleeCoverageRecovering* rules (Figure 11) interrupt the execution of rules *CallerOutOfCoverage* and *CalleeOutOfCoverage*, respectively, if phone p1 (or p2) recovers its coverage. Please note that *CallerOutOfCoverage* and *CalleeOutOfCoverage* will also be canceled whenever the talk is finalized, since atomic actions are canceled whenever one of their LHS instantiation objects or action execution elements are deleted before they finalize.

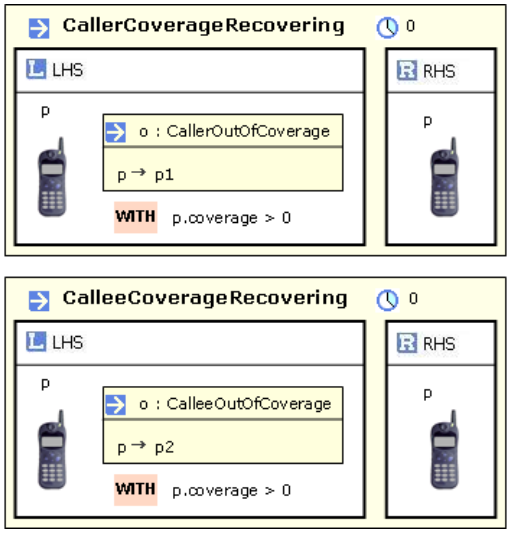

Figure 11. The *CallerCoverageRecovering* and *CalleeCoverageRecovering* atomic rules.

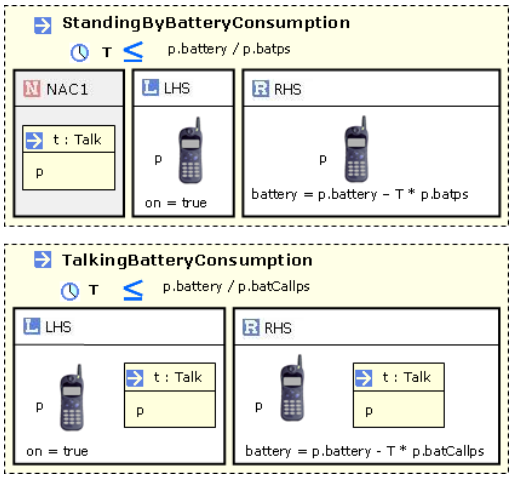

Figure 12. The *StandingByBatteryConsumption* and *TalkingBatteryConsumption* ongoing rules.

Ongoing actions. So far, we have used atomic rules to model the behavior of our DSL. Let us now show how ongoing rules are very suited for handling continuous actions. This kind of rules will be very convenient for modeling system properties that must be updated at every moment. Figure 12 shows two ongoing rules that model battery consumption. Rule *TalkingBatteryConsumption* specifies the battery consumption while the phone is talking. In this case, the battery power is decreased talkBps units per second. A similar rule updates the amount of battery when the phone is in standby (*StandingByBatteryConsumption* rule).

Note that, according to the current metamodel, we cannot distinguish between calls that have already been picked up or not by using only system states. Thus, the *TalkingBatteryConsumption* and *StandingByBatteryConsumption* rules make use of action execution elements to identify the desired situation: we explicitly specify that the phone p must be participating (or is not participating, respectively) in a Talk rule execution. Furthermore, since the role of phone p in rule Talk (caller or callee) is not specified, we do not need to distinguish between incoming and outgoing call cases.

By using ongoing rules, we have modeled battery consumption separately from the other behavior rules, avoiding concerns tangling. Furthermore, ongoing rules are executed for the the maximum time possible before another event occurs, and previously to atomic rules, so the battery power is always updated whenever another rule tries to access it. To explicitly identify the state in which a phone's battery

[1] We could force the phone to register a call before it occurs by restricting both rules to be applied whenever rules *SwitchOff* or *BatteryOff* are being executed (instead of applying it when the phone is already off), by identifying both cases with two more rules.

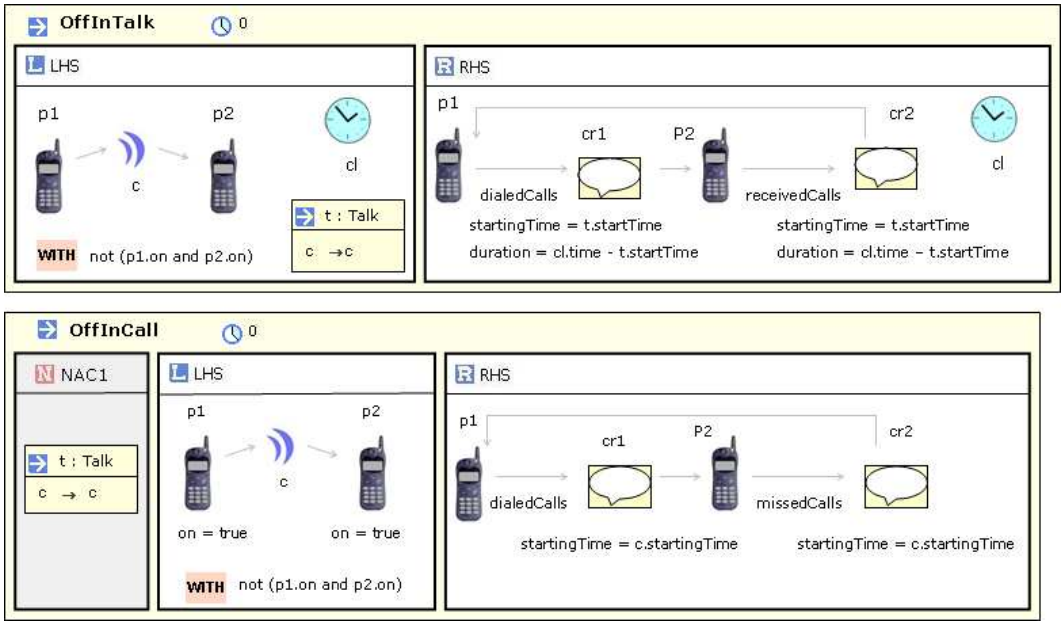

Figure 9. The *OffInTalk* and *OffInCall* atomic rules.

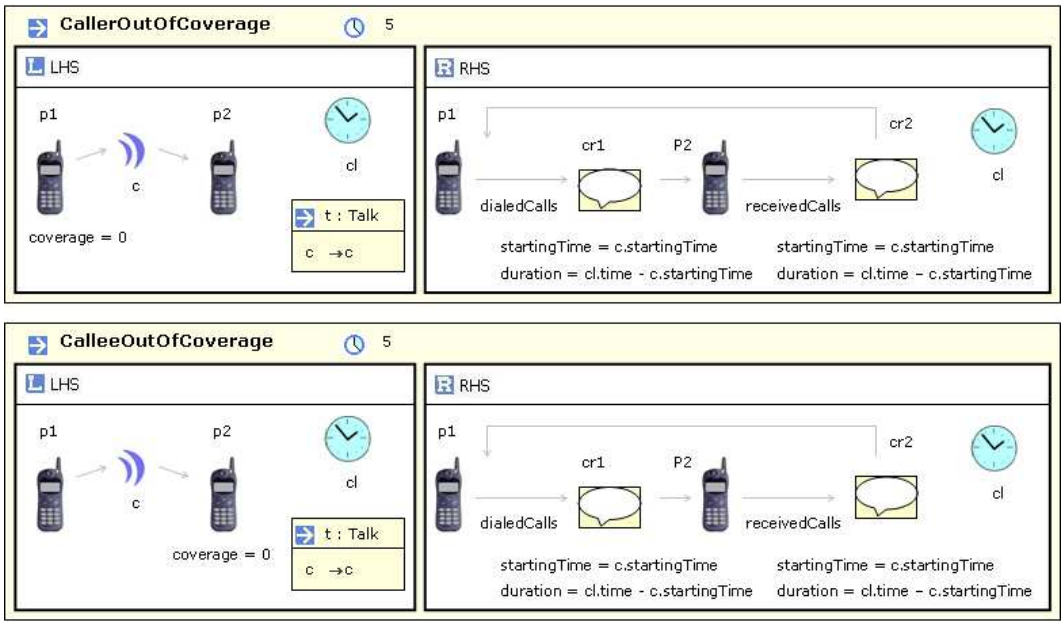

Figure 10. The *CallerOutOfCoverage* and *CalleeOutOfCoverage* atomic rules.

has run out (and not to decrease the battery power below zero), we limit the duration of both rules (look at the right of symbol $\leq$) not to exceed the battery power.

Finally, we can easily simulate phones moving by using another ongoing rule, shown in Figure 13. In this case, phones will cover a random number of meters per second (up to ten meters in each direction).

4 Tool support

We have developed a tool, named e-Motions, as an Eclipse plug-in that supports the graphical modeling dis-

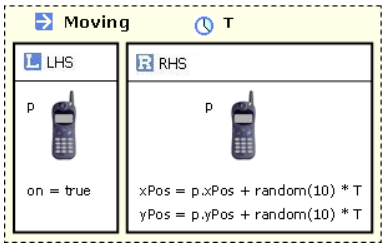

Figure 13. The *Moving* ongoing rule.

cussed in the previous sections. The tool comprises two editors: the *behavior editor*, which allows users to specify the set of rules; and the *rule editor*, used to define rule patterns (LHS, RHS and NAC) and their corresponding elements (objects, links, conditions and action executions).

The tool allows users to include and use the concrete syntax of the DSL for defining timed behavioral rules. The graphical syntax is defined through a GCS (Graphical Concrete Syntax) model, on which we basically associate an image to each metaclass (through *ClassGD* instances).Then, when an object is included in a pattern and its class is specified, the object is drawn using the corresponding *ClassGD* image if defined; otherwise, a default image (a rectangle) is depicted.

5. Semantic Mapping to Real-Time Maude

Once we have defined the dynamic behavior of a DSL using timed rules, the following step is to perform simulation and formal analysis over the specifications. Since these behavioral specifications are also models, we can automatically transform them into different semantic domains. In general, each semantic domain is more appropriate to represent and reason about certain properties, and to conduct certain kinds of analysis.

Analyzing these timed rules cannot be easily done using the common theoretical results and tools defined for graph transformations, since timed rules include some extensions. However, other semantic domains are better suited. In this paper, we propose an automatic mapping into rewriting logic. This semantic mapping provides a formal semantics of the proposal, and enables the formal analysis of the source models using the capabilities of Maude for executing the specifications, and for conducting reachability and model-checking analysis. In this way, the designer will have the advantage of a visual and intuitive specification, and an efficient and powerful mean for analysis with a formal basis.

5.1. Real-Time Maude

Real-Time Maude [9, 2] (hereinafter RTMaude) is a rewriting-logic-based specification/modeling language and high-performance formal analysis tool for real-time systems. Rewriting logic is a logic of change that can naturally deal with states and non-deterministic concurrent computations. A distributed system is axiomatized by an equational theory describing its set of states and a collection of rewrite rules. Rewrite rules are of the form $t \longrightarrow t'$, where t and t' are terms, and they specify the dynamics of a system in rewriting logic. Rewrite rules describe the local, concurrent transitions that are possible in the system, i.e., when a part of the system state fits the pattern t then it can change to a new local state fitting the pattern t'. The guards of conditional rules act as blocking preconditions, in the sense that a conditional rule can only be fired if the condition is satisfied. The syntax for conditional rules is crl [l] : t => t' if *Cond*, with l the rule label and *Cond* its condition. *Tick rewrite rules* are conditional rules of the form crl [l] : $\{t\}$ => $\{t'\}$ in time τ if *Cond*, where τ is a term of sort Time that denoted the *duration* of the rewrite, and that affects the global time elapse. In RTMaude, both discrete and dense time are implemented, and users can define their own time domain.

5.2. Encoding Timed Rules in RTMaude

In RTMaude, since tick rules affect the global time, time elapse is usually modeled by one single tick rule, and the system dynamic behavior by instantaneous transitions [9]. The tick rule models time elapse by using two functions: delta and mte (maximal time elapse). The delta function defines the effect of time elapse over every model element, and the mte function defines the maximum amount of time that can elapse before any action is performed. Then, time advances non-deterministically by any amount T, which must be equal or less than the maximum time elapse of the system.

Atomic Rules. Atomic rules are represented as two RTMaude instantaneous rules, one modelling its *triggering* and one modeling its actual *realization*.

- *The triggering rule.* When the rule precondition is satisfied, an AtomicActionExec object is created. AtomicActionExec objects represent atomic rules executions, each one acting as a countdown (attribute timer) to the finalization of the action. They gather all the information needed for its instantiation, such as the rule name, the identifiers of the participating elements, the start time and the variable definitions. Initially, the timer is set to the given duration of the rule. Additionally, another AtomicActionExec object with the same attribute

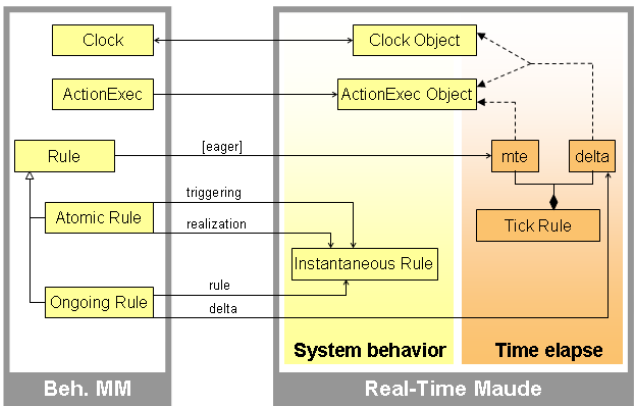

Figure 14. Mappings to Real-Time Maude.

values and an undefined ending time (attribute endingTime) is included in the rule as a NAC to avoid infinitive number of rule executions.

- *The realization rule.* Once a rule's timer is consumed (i.e., there is an AtomicActionExec object with timer=0), the corresponding action can be performed if none of the action's participants has been deleted. Then, the LHS is substituted by the RHS and the attribute values are computed. To keep track of the performed actions, the AtomicActionExec object is not deleted and its ending time set.

Ongoing rules. Each ongoing rule is represented as a RTMaude instantaneous rule plus an equation for the delta operator. Since ongoing rules express continuous actions, they are represented as instantaneous rules which will be attempted after every execution of the tick rule.

- *The instantaneous rule.* The instantaneous rule is encoded as a triggering rule of a atomic action. When the rule precondition is satisfied, an OngoingActionExec object is created, which identifies that the corresponding ongoing rule can be executed. It also gathers all the information needed for its instantiation, such as the rule name, the rule participants, the maximal duration (attribute maxDuration) and the time interval upper bound (attribute upperBound).

- *The delta equation.* In the following time elapse, the delta equation consumes the OngoingRuleExec object and substitutes the LHS instantiation by the RHS if applicable. Note that, at this time, the value of the time elapse T is known, and therefore attribute computations relative to T can be properly accomplished.

Time passage and rule execution modes. As previously mentioned, time elapse is modeled by using RTMaude delta and mte functions. Both functions need only to be defined over Clock, AtomicActionExec and OngoingActionExec objects, thus making DSL objects completely unaware of time.

The delta function decreases AtomicActionExec timers in T units, and increases the clock value in T units. It also consumes OngoingActionExec objects (applying the corresponding substitution, as explained before). The mte function is defined as the minimum of timer values of current AtomicActionExec objects, maxDuration values of current OngoingActionExec objects, and the difference between the current time elapse and upperBound values of current OngoingActionExec objects. Note that UpperBound values of AtomicActionExec objects are considered in atomic triggering rules: an atomic rule cannot be triggered if the current time plus its rule duration exceeds its upperBound. In this way, we force atomic rules to be finalized when their timer expires, and ongoing rules are applied (if possible) up to maxDuration units of time and always before their time interval upper bound.

Regarding the eager/lazy rule execution modes, we add an additional mte equation for every eager rule. This mte equation forbids the time to elapse (mte = 0) whenever the corresponding eager rule can be triggered and it has not been so.

Further Elements. The remaining elements are encoded as follows:

- the clock object is encoded as a RTMaude object belonging to the predefined class Clock;

- LHS and NAC conditions are encoded as RTMaude rule conditions;

- attribute computations (slots) are encoded as computations performed in the right-hand side of the rules;

- NAC patterns are encoded as invocations to predicates in the corresponding rule condition that checks wether an occurrence of the specified pattern is found in the model (and if so, forbidding the application of the rule); and

- action executions are encoded by means of AtomicActionExec objects that refer to the corresponding atomic rule

Conditions, attribute computations, helpers and several rule property values are expressed with OCL, which is fully supported by Maude [13].

5.3. Analysis and Simulation of Time-Dependent Behavioral Specifications

By transforming our specifications into RTMaude, they can be used for simulation and analysis. RTMaude offers tool support for interesting analysis possibilities, such as simulation, reachability analysis and model checking, which can be naturally used with the RTMaude specifications we produce (see, e.g., [8, 10] for examples of these kinds of analyses).

6 Related Work

There are several approaches that propose in-place model transformations to deal with the behavior of a DSL, from textual to graphical (see [10] for a comprehensive survey). However, none of these works includes a quantitative model of time. When time is needed, it is modeled by adding some kind of clocks to the DSL metamodel. These clocks are handled in the same way as common objects, which forces designers to modify the DSL metamodel to include time aspects. Furthermore, this does not constrain designers from unwillingly defining time-inconsistent sequences of states, as we previously mentioned. A similar approach is followed in [6], where graph transformation systems are provided with a model of time by representing logical clocks as distinguished node attributes. This work, based on time environment-relationship (TER) nets [5] (an approach to modeling time in high-level Petri nets), does not extend the base formalism but specializes it (as its predecessor), and enables the incorporation of the theoretical results of graph transformation. The verification of the system time-consistency is discussed by introducing several semantic choices and a *global monotonicity theorem*, which provides conditions for the existence of time ordered transformation sequences.

A recent work [15] proposes to complement graph grammar rules with the Discrete EVent system Specification (DEVS) formalism to model time-dependent behavior. Although this has the benefit of allowing modular designs, this approach requires specialized knowledge and expertise about the DEVS formalism, something that may hinder its usability by the average DSL designer. Furthermore they do not provide analysis capabilities: system evaluation is accomplished through simulation.

Maude has already been proposed as a formal notation and environment for specifying and effectively analyzing models and metamodels [11, 14]. Simulation, reachability and model-checking analysis are possible using the tools and techniques provided by Maude. In [10] we showed how Maude is also suitable as a semantic domain for standard in-place rules, formalizing graph transformations using equational and rewriting logic with Maude.

In our previous work [12] we also how some timed behavioral specifications can be supported, adding duration to in-place rules. In this paper we have extended our previous proposal in numerous ways (e.g., ongoing and scheduled rules, periodicity, helpers, variables, etc.), based on the experience we have gained from the specification of several real-time systems and applications. We have used RTMaude to model time elapse and to perform the same kind of analysis we were able to perform for time-unaware systems [14]. Our approach also supports attribute computation, ordered collections, and OCL expressions. In addition, it provides a quantitative time representation and a way to model action execution, opposite to standard approaches, in which rule patterns are simply composed of system states.

7 Conclusions and Future Work

In this paper we extend in-place rules with a quantitative model of time and with mechanisms that allow designers to state action properties, easing the design of real-time complex systems. This proposal permits decoupling time information from the structural aspects of DSLs (i.e., their metamodels). In addition, this paper also presents a graphical modeling tool aimed at visually specifying these timed rules, as well as a mapping to RTMaude, which will make the visual specifications amenable to simulation and other kinds of formal analyses.

We are currently working on further extending our graphical tool to support the input and output of RTMaude's analysis tools using the native visual notation of the DSL. This will make the use of RTMaude completely transparent to users.

References

[1] E. Biermann, K. Ehrig, C. Khler, G. Kuhns, G. Taentzer, and E. Weiss. Graphical definition of in-place transformations in the Eclipse Modeling Framework. In *MDE Languages and Systems*, number 4199 in LNCS. Springer, 2006.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude – A High-Performance Logical Framework*. Number 4350 in LNCS. Springer, Heidelberg, Germany, 2007.

[3] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[4] J. de Lara and H. Vangheluwe. Translating model simulators to analysis models. In *Proc. of FASE 2008*, number 4961 in LNCS, pages 77–92. Springer, 2008.

[5] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. A unified high-level petri net formalism for time-critical systems. *IEEE Trans. Softw. Eng.*, 17(2):160–172, 1991.

[6] S. Gyapay, R. Heckel, and D. Varró. Graph transformation with time: Causality and logical clocks. In *Proc. of 1st Int. Conference on Graph Transformation (ICGT'02)*, pages 120–134. Springer-Verlag, 2002.

[7] J. Meseguer. The temporal logic of rewriting: A gentle introduction. In *Concurrency, Graphs and Models*, pages 354–382, 2008.

[8] P. C. Ölveczky. *Real-Time Maude 2.3 Manual*, 2007. `http://www.ifi.uio.no/RealTimeMaude/`.

[9] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[10] J. E. Rivera, E. Guerra, J. de Lara, and A. Vallecillo. Analyzing rule-based behavioral semantics of visual modeling languages with Maude. In *Proc. of the International Conference on Software Language Engineering (SLE'08)*, LNCS. Springer, Oct. 2008.

[11] J. E. Rivera and A. Vallecillo. Adding behavioral semantics to models. In *Proc. of EDOC 2007*, pages 169–180. IEEE Computer Society, Oct. 2007.

[12] J. E. Rivera, C. Vicente-Chicote, and A. Vallecillo. Extending visual modeling languages with timed behavioral specifications. In *Proc. of IDEAS 2009*, Medellín, Colombia, Apr. 2009.

[13] M. Roldán and F. Durán. Representing UML models in mOdCL. Manuscript. Available at `http://maude.lcc.uma.es/mOdCL`., 2008.

[14] J. R. Romero, J. E. Rivera, F. Durán, and A. Vallecillo. Formal and tool support for model driven engineering with Maude. *Journal of Object Technology*, 6(9):187–207, June 2007.

[15] E. Syriani and H. Vangheluwe. Programmed graph rewriting with time for simulation-based design. In *Proc. of the International Conference on Model Transformation (ICMT 2008)*, number 5063 in LNCS, pages 91–106. Springer-Verlag, 2008.