A Graphical Approach for Modeling Time-Dependent Behavior of DSLs

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' abstract syntax are usually defined by a metamodel. 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. 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 may produce conflicting results across different tools. Furthermore, the lack of explicit behavioral semantics strongly hampers the development of simulation and formal analysis tools, which is particularly important in safety-critical real-time and embedded system domains.

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 3). 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 or forcing unnatural changes to the system specification [6].

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, called e-Motions [1], aimed at defining behavioral specification models, which can be fully integrated in MDE processes. Its precise behavioral semantics is given by mappings to different semantic domains. In particular, a mapping between these specifications and Real-Time Maude [7] can be defined [9], making them amenable to simulation and different kinds of formal analyses (see, e.g., [2]). This paper extends our initial proposal presented in [10] 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.

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 [8] 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 : [NAC] $\times$ LHS $\rightarrow$ RHS, where l is the rule's label (its name); LHS (Left Hand-Side) and RHS

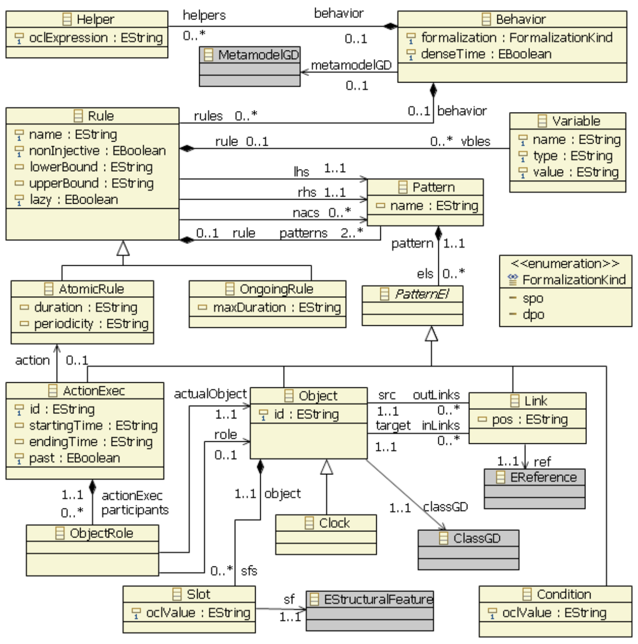

Figure 1. The Behavior metamodel.

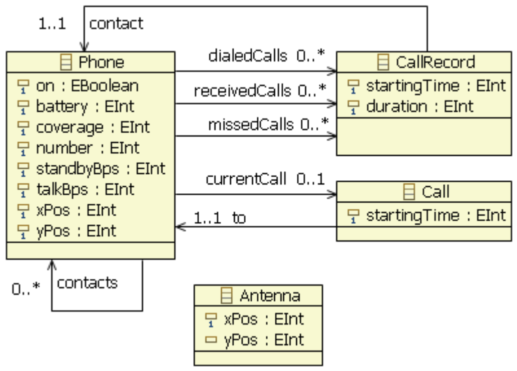

Figure 2. Mobile Phone Network Metamodel.

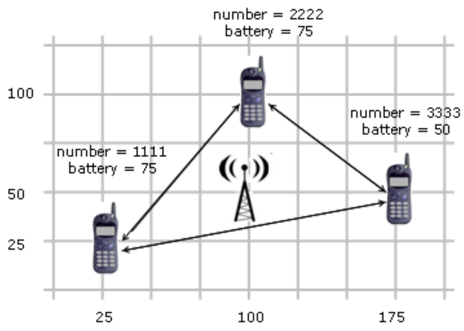

Figure 3. A mobile phone network example.

(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 the *precondition* for the rule to be applied, whereas the RHS represents its *postcondition*. 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 model, and none of the NAC patterns occurs. Generally, if several matches are found, one of them is non-deterministically selected and applied, producing a new 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 mechanism).

Fig. 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 [9]. 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 (see, e.g., [8]), although adapted to the tree-structure of Eclipse models. The following subsection describe the main concepts of our approach by introducing a modeling language for mobile phone networks (MPNs). In this paper we have considered only discrete time, although dense time is also supported.

2.1. A Mobile Phone Network Example

The MPN metamodel is shown in Fig. 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.

Fig. 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.

We show in what follows a possible specification of

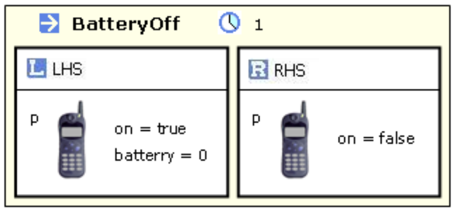

Figure 4. The *BatteryOff* rule.

MPNs behavior. Our main concern is to illustrate the different features of our proposal. Of course, different design decisions could have been taken. The complete specification of the MPN example can be found in [9].

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. Atomic rules can be triggered whenever an occurrence (or match) of the LHS, and none of the NAC patterns, is found in the model. 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 (if the objects are still there).

Fig. 4 shows the *BatteryOff* atomic rule. Whenever a phone is on and it has no battery, it is switched off. This action takes one second.

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 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, etc. Provided that the clock behavior cannot be modified, users cannot, e.g., drive the system to time-inconsistent sequences of states (even unwillingly).

Figure 5 shows two atomic rules that model the behavior of phone calls. The *MakeCall* rule 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 (see the condition specified in the WITH clause). The four NAC patterns forbid the execution of the rule whenever one of the phones is participating in another (incoming or outgoing) call. This action is modeled by a lazy rule, which means that is not forced to be applied whenever its LHS pattern occurs in the model: we allow phones to make calls at a non-deterministic moment in time.

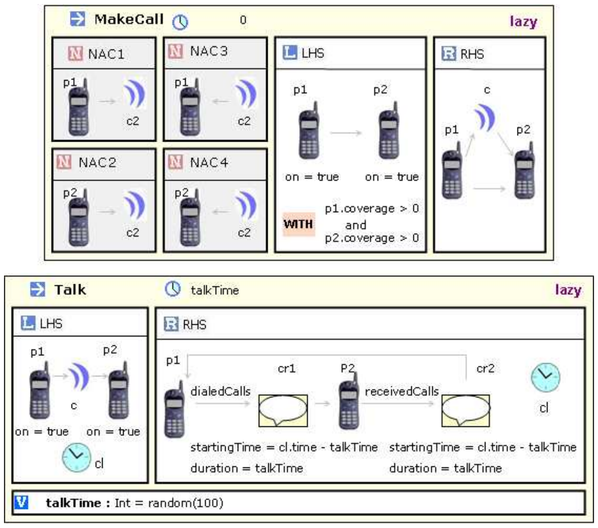

Figure 5. The *MakeCall* and *Talk* atomic rules.

Once the call is initiated, it can be picked up to start a talk (*Talk* rule) or just ignored (*MissedCall* rule [9], which is not specified here due to space reasons). 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. The context of a user-defined variable is the rule in which it is defined. Function random(b : Int) : Int is provided by the system, and generates a pseudo-random number between zero and the given bound b. 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 the use of a 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.

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's precondition holds, 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).

Fig. 6 shows the *Coverage* rule, which specifies the way in which coverage changes. Coverage is updated every ten seconds (see the loop icon in the header of the *Coverage* rule). Each cell phone is covered by the closest antenna: as

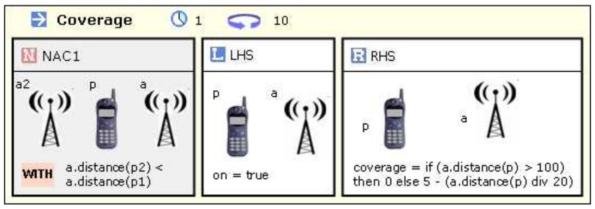
Coverage
1
10
NAC1
a2
p
a
WITH
a.distance(p2) < a.distance(p1)
LHS
p
a
on = true
RHS
p
a
coverage = if (a.distance(p) > 100) then 0 else 5 - (a.distance(p) div 20)

Figure 6. The *Coverage* atomic rule.

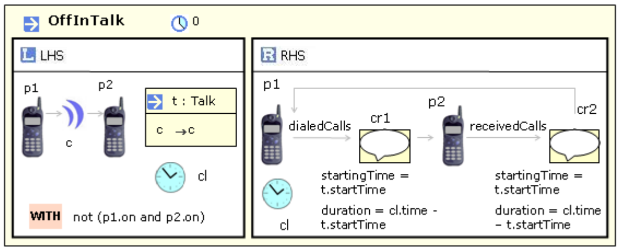
OffInTalk
0
LHS
p1
p2
c
t : Talk
c →c
cl
WITH
not (p1.on and p2.on)
RHS
p1
dialedCalls
cr1
p2
receivedCalls
cr2
startingTime = t.startTime
duration = cl.time - t.startTime
startingTime = t.startTime
duration = cl.time – t.startTime
cl

Figure 7. The *OffInTalk* atomic rule.

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 (OCL operation):
$contextAntenna :: distance(p : Phone) : Integer$
$body : (self.xPosp.xPos).abs() + (self.yPosp.yPos).abs()$

Action executions. In standard in-place transformation approaches, model patterns (LHSs, RHSs and NACs) are defined in terms of system states. This is a strong limitation in those situations in which we need to refer to actions currently executing, or to those that have been executed in the past. For example, we can be interested in knowing whether an object is currently performing a given action, in order to not allow it to perform another (see [9] for some examples), or in reasoning about the performed actions to be able to, for instance, search for undesirable action occurrences or invalid sequences of action executions. 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. [6]).

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 (by using the past attribute). Action executions specify the type of the action (i.e., the name of the atomic rule), its identifier, 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 → 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 *MakeCall* rule (see Fig. 5) defines three roles: two phones (p1 and p2) and one call (c). We can also leave unspecified the type of the rule and the role of an object to represent whatever rule instantiation or object role, respectively.

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

Consider, for instance, the *OffInTalk* atomic rule in Fig. 7, which models the behavior of a phone when it is switched off in the middle of a talk. The *OffInTalk* rule is applied whenever two phones are having a talk (we explicitly specify that the call c is participating in the *Talk* action with the c role) 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.

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. To model this action, we need a rule that must be recomputed each time a scheduled event happens to have the battery attribute updated at every moment.

Ongoing rules performs in this way. They do not have an a-priori duration time: they progress with time while the rule preconditions (LHS and not NACs) hold, or until the time limit of the action is reached (given by the maxDuration attribute, see Fig. 1). Fig. 8 shows the *StandByBatteryConsumption* ongoing rule. It models battery consumption when the phone is in standby (it is not talking). In this case, the battery power is decreased standbyBps units per second. Note the use of the action execution element instead of a simple Call object to differentiate received calls. In this way, we do not also need to distinguish between incoming and outgoing call cases, since the role of phone p in rule Talk (caller or callee) is not specified. To explicitly identify the state in which a phone's battery has run out (and not to decrease the battery power below zero), we limit the

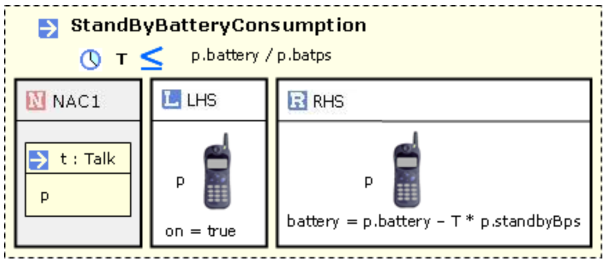
StandByBatteryConsumption
T ≤ p.battery / p.batps
NAC1
t : Talk
p
LHS
p
on = true
RHS
p
battery = p.battery - T * p.standbyBps

Figure 8. The *StandByBatteryConsumption* **rule.**

duration of both rules (look at the right of symbol $\leq$) not to exceed the battery power. Analogously, we can define a similar rule that updates the amount of battery while the phone is talking [9].

3 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 [8] for a comprehensive survey). However, none of these works includes a quantitative model of time. When time is needed, it is usually 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 [5], 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 (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 [11] 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.

4 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. We are currently working on further extensions of this tool to automate the interaction with Real-Time Maude and its analysis tools.

References

[1] The e-Motions tool. `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/E-motions`.

[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] 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.

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

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

[8] 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.

[9] J. E. Rivera, A. Vallecillo, and F. Durán. e-Motions: A graphical approach for modeling time-dependent behavior of domain specific languages. Manuscript. Available at `http://atenea.lcc.uma.es/`, 2008.

[10] 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.

[11] 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.