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. The abstract syntax of a DSL is usually defined by a metamodel, while in-place model transformation rules provide an intuitive way to complement metamodels with their behavioral specifications. We propose a modeling notation that extends in-place rules with a quantitative model of time, and with mechanisms that allow designers to specify action-based properties, thus facilitating the design of real-time complex systems. We present a graphical modeling tool that we have built for visually specifying these timed behavioral 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 by 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 [10], 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 [2]. The behavior of the DSL is then specified in terms of the permitted actions, which are in turn modeled by the 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 only allow users to model state-based properties, forcing designers to unnaturally extend their metamodels with the state of the actions that should be observed. The need for action-based specifications has been recently acknowledged, showing that this kind of specifications are more natural and expressive in many different situations [5].

In this paper we propose a modeling notation that extends in-place transformation rules with a quantitative model of time and with mechanisms that allow designers to specify action-based properties, thus facilitating the design of real-time complex systems. We present a graphical modeling tool, called e-Motions [**?**], that we have built for visually specifying these timed behavioral specifications.

2. Real-Time In-Place Transformations Rules

There are several approaches that propose in-place model transformation rules to deal with the behavior of DSLs, from textual to graphical (see [7] 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 [3].

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's label (its 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

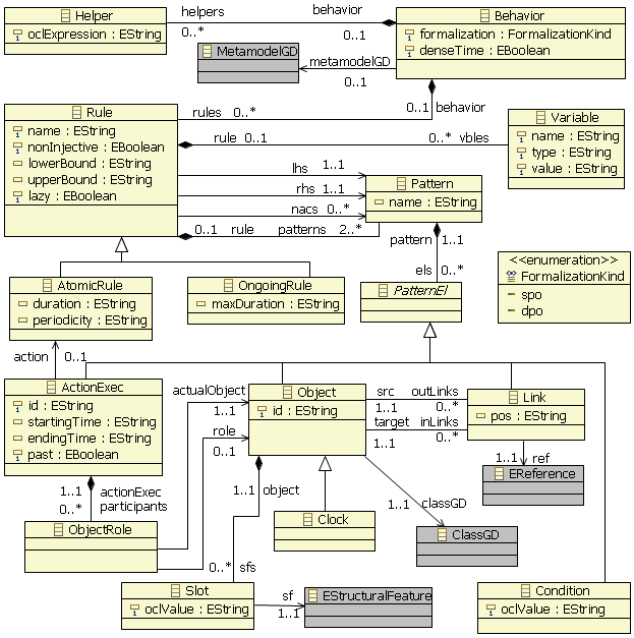

Figure 1. The Behavior metamodel.

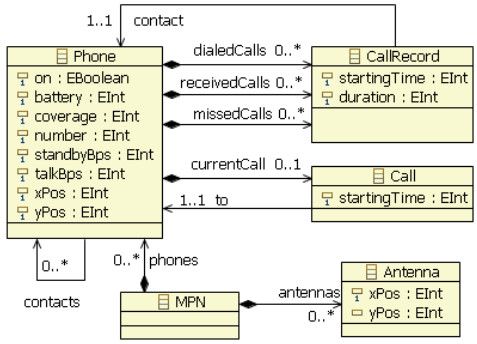

Figure 2. Mobile Phone Network metamodel.

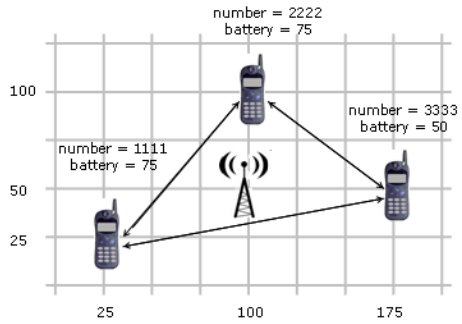

Figure 3. A mobile phone network example.

in the model. The LHS and NAC patterns of a rule express its *precondition*, whereas its RHS represents its *post-condition*. 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 corresponding RHS (the rule *realization*). The model transformation proceeds by applying the rules in a non-deterministic order, until none is applicable [7].

Metamodel for time-dependent behavior. 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 of the DSL [1]. 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 time attribute) to model, e.g., time stamps. Of course, users can define their own clocks to model other aspects of distributed systems (such as distributed clocks) and specify how they individually evolve. 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., [7]), although adapted to the tree-structure of Eclipse models.

A Mobile Phone Network Example. Let us introduce a modeling language for mobile phone networks (MPNs), which will serve us as the motivating example to illustrate the main features of our approach. In this paper, we will only deal with linear discrete time, although dense time is also supported.

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 by its position in the grid. All phones are initially off, and their

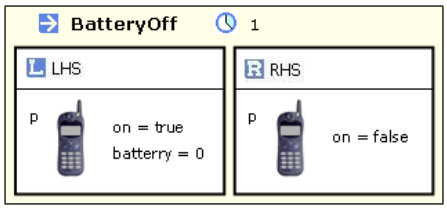

Figure 4. The *BatteryOff* rule.

contacts are represented by arrows between them.

A possible specification of the MPNs behavior follows. Our main aim is to illustrate the different features of our proposal. Of course, alternative design decisions could have been taken. The complete specification of the MPN example can be found in [1].

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 realized after t time units. At that time, the rule is applied by performing the attribute computations and substituting the objects in the match by its RHS. Note that the objects in the LHS may be involved in other actions during the time elapse, and therefore their states may have changed in the meanwhile. The only condition for the final application of the triggered rule is that the objects involved are still there.

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

Figure 5 shows two other 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 is a special kind of rule that is not forced to be immediately applied whenever their LHS pattern occurs in the model. Thus, we allow a phone to make calls at a non-deterministic moment in time.

Once a call is initiated, the user can pick up the phone to start a talk (*Talk* rule) or just ignore it (modeled by the *MissedCall* rule, which can be found in [1]). If the phone

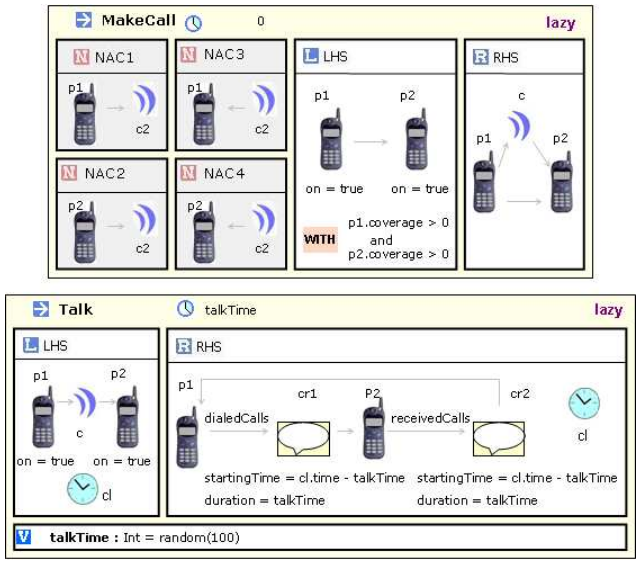

Figure 5. The *MakeCall* and *Talk* atomic rules.

is picked up, a conversation will take place for talkTime time units. The value of the talkTime variable (in the variables box at the bottom of the rule specification) is defined as a pseudo-random value (random(100)). The context of a variable is the rule in which it is defined. Rule *variables* provide the mechanism to define values that are used in several parts of the rule. Variables are computed when rules are triggered, and do not change their values until they are finally applied — unlike object attributes, whose values are computed either when the rule is triggered, in case of attributes appearing in LHS and NACs patterns, or when the rule is realized, if they appear in the RHS pattern of the rule. 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 this example, we consider that the starting time of a received call is the moment at which the phone is picked up. Note the use of the Clock's time attribute in the RHS pattern of the rule, which refers to the moment of the finalization of the rule, since attributes appearing in the rule's RHS are computed 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 still holds). Normal atomic rules are triggered at the beginning of the period, while lazy rules can be executed at any time within the period (only once per period).

Fig. 6 shows the *Coverage* rule, which specifies the way in which coverage changes. Coverage is updated every ten

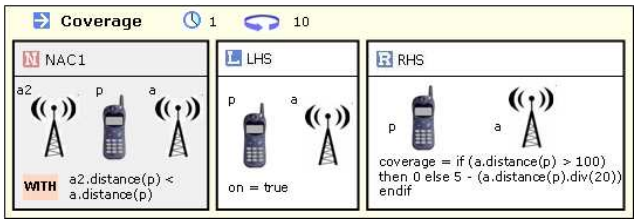

Figure 6. The *Coverage* atomic rule.

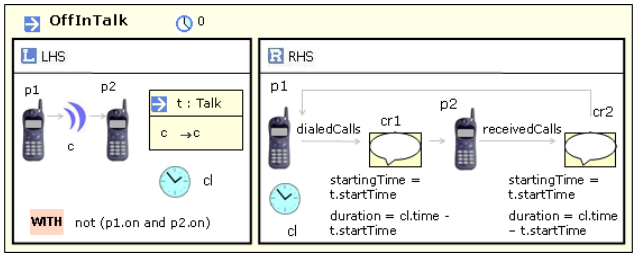

Figure 7. The *OffInTalk* atomic rule.

seconds (see the loop icon in the header of the *Coverage* rule). Each cell phone is covered by its closest antenna: as specified in its NAC pattern, the rule cannot be applied if there exists another antenna which is closer to the phone. The Manhattan distance between the two objects is computed by the following helper (OCL operation):

context Antenna :: $distance(p : Phone) : Integer$
body : $(xPos - p.xPos).abs() + (yPos - p.yPos).abs()$

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 engaged in a given action (e.g., in order not to allow it to perform another), or in reasoning about the performed actions so far (e.g., to be able to 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. [5]).

In order to be able to model both state-based and action-based properties, we have extended 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 (using its past attribute). Action executions specify the action type (i.e., the name of the atomic rule), its identifier, its starting and ending times, 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 *MakeCall* rule (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 *any* rule instantiation or object role, respectively.

Exceptions. 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 action 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 effects are defined by the 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 conversation. 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 along time.

Ongoing rules model this kind of behavior. They do not have any 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 (defined by the maxDuration attribute, see Fig. 1). Fig. 8 shows the *StandBy-BatteryConsumption* ongoing rule. It models battery consumption when the phone is in standby (i.e., it is not talking). In this case, the battery power is decreased in standbyBps units per second. Note the use of the action execution element instead of a simple Call object. In this way we do not need to differentiate between incoming and outgoing calls, thus covering both cases since the role of phone p in rule Talk (caller or callee) is not specified. To explicitly identify the state of a phone that has run out of battery (in order not to decrease the battery power below zero, for instance), we limit the duration of the rule (after the $\leq$ symbol) not

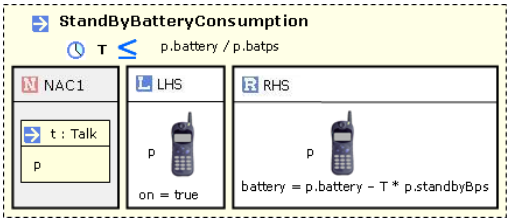

Figure 8. The *StandByBatteryConsumption* rule.

to exceed the battery power. Analogously, we can define a similar rule that updates the amount of battery while the phone is talking (see [1] for additional examples).

3 Related Work

There are several approaches that propose in-place model transformations to deal with the behavior of DSLs, from textual to graphical (see [7] 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's 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. This kind of approach is followed in [4], 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 achieved by introducing several semantic choices and a *global monotonicity theorem*, which provides conditions for the existence of time-ordered transformation sequences.

A recent work [9] 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.

In our previous work [8] we showed how some timed behavioral specifications can be supported, adding duration to in-place rules. Here, we have extended our previous proposal in numerous ways to include variables, ongoing actions, periodicity, and lazy execution modes.

4 Conclusions

In this paper we extend in-place rules with a quantitative model of time and with mechanisms that allow designers to specify action properties, easing the design of real-time complex systems. This proposal enables decoupling time information from the structural aspects of DSLs (i.e., their metamodels). Our proposal also supports a way to model action executions, attribute computations, ordered collections, and OCL expressions. We have also presented a graphical modeling tool (called e-Motions) aimed at visually specifying these timed rules. The precise semantics of the language, which will appear somewhere else, is defined by a mapping to Real-Time Maude [6, 10]. This mapping makes the specifications amenable to simulation and different kinds of formal analyses.

References

[1] J. E. Rivera, A. Vallecillo, and F. Durán. e-Motions: A graphical approach for modeling time-dependent behavior of domain specific languages. Available at `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/E-motions`, 2009.

[2] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proc. of W. on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

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

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

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

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

[7] 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 Intl. Conf. on Software Language Engineering (SLE'08)*, LNCS. Springer, 2008.

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

[9] E. Syriani and H. Vangheluwe. Programmed graph rewriting with time for simulation-based design. In *Proc. of the Intl. Conf. on Model Transformation (ICMT 2008)*, vol. 5063 of LNCS, pp. 91–106. Springer, 2008.

[10] J. E. Rivera, F. Durán, and A. Vallecillo. Programmed graph rewriting with time for simulation-based design. To appear in *Simulation: Transactions of the Society for Modeling and Simulation International*, 2009.