Extending Visual Modeling Languages with Timed Behavior Specifications.

Conference Paper · January 2009

Source: DBLP

CITATIONS
11

READS
176

3 authors, including:

Cristina Vicente-Chicote
Universidad de Extremadura
109 PUBLICATIONS **698** CITATIONS

SEE PROFILE

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

SEE PROFILE

Extending Visual Modeling Languages with Timed Behavioral Specifications

José Eduardo Rivera[1], Cristina Vicente-Chicote[2], and Antonio Vallecillo[1]

Universidad de Málaga (Spain)[1]
Universidad Politécnica de Cartagena (Spain)[2]
rivera@lcc.uma.es, cristina.vicente@upct.es, 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, although this hampers the development of formal analysis and simulation tools. In this paper we advocate the use of in-place model transformations to complement metamodels (the structural aspects of a DSL) with timed behavioral specifications. In particular, we propose an extension for in-place transformation rules to state action properties (not only model element properties), and to model time-dependent behavior. This approach avoids making unnatural changes to the DSL metamodels to represent behavioral and time aspects, and allows the resulting specifications to be translated into different semantic domains, such as Real-Time Maude, making them amenable to simulation and other kinds of formal analysis.

1 Introduction

Domain specific languages (DSLs) play a cornerstone role in Model-Driven Software Development (MDSD) 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 MDSD community until recently, despite the fact that this facilitates semantic mismatches between design models and analysis tools. While this problem exists virtually in every domain where DSLs find application, it is more apparent in domains in which behavior needs to be explicitly represented. 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 other formal analysis tools.

One way of specifying the dynamic behavior of a DSL is describing the evolution of the modeled artifacts along some time model. In MDSD, where models, metamodels and model transformations play a key role, model transformations supporting in-place update [1] seem to be the natural way. In these transformation languages, source and target models are always instances of the same metamodel (i.e., they are *endogenous* transformations), and rules are used both (1) to describe the preconditions of the actions to be triggered, and (2) to represent the effects of these actions in the model. If these transformations use the concrete syntax associated to a DSL, they allow designers to work using only domain specific concepts [2], raising the level of abstraction and making behavioral specifications more intuitive and natural both to specify and understand.

Only a few of the current approaches deal with time-dependent behavior in in-place model transformations (see Sec. 5). In critical domains, such as real-time and embedded systems, timeouts, timing constraints and delays are essential concepts. Therefore, these concepts should be explicitly modeled in their behavioral specification to enable a proper analysis and simulation. Furthermore, current approaches do not allow users to model action-based properties, making them inexpressible and forcing unnatural changes to the system specification [3].

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 Model Driven Engineering (MDE) processes and, e.g., transformed into different semantic domains. In particular, we show how a mapping between these specifications and Real-Time Maude (RTMaude) can be defined, making them amenable to simulation and other kinds of formal analyses, such as reachability or model-checking [4].

Paper organization. First, Section 2 shows how in-place transformation rules are extended to model time-dependent behavior and action-based properties. Then, Section 3 introduces the graphical modeling tool we have built, and how it can be used to model a production system example. Section 4 gives an overview of a semantic mapping from timed behavioral specifications to RTMaude. Finally, Section 5 compares our work with other related proposals, and Section 6 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 dynamic behavior of a DSL [5]. These transformations are composed of a set of rules of the form $l : [\text{NAC}] \times \text{LHS} \longrightarrow \text{RHS}$, 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 postconditions. Thus, a rule can be applied 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 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).

2.1 A Production System Example

For illustration purposes, let us introduce a modeling language for industrial production lines, which will serve as the motivating example to show the capabilities of our approach. It is inspired by a common example used in several works for describing DSLs (see, e.g. [6, 7]).

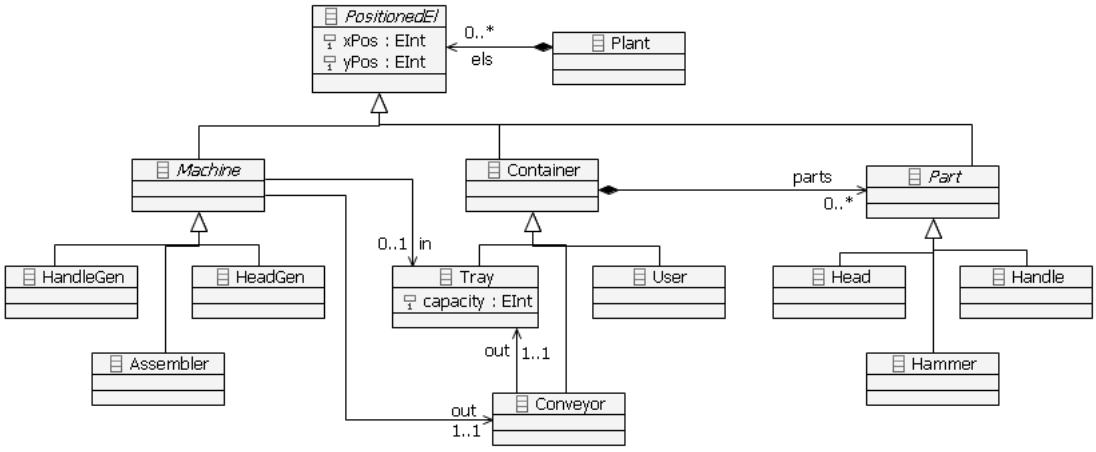

Fig. 1. Meta-model of a DSL for production systems.

The production system metamodel is shown in Fig. 1. A production system is made of plants that are composed of machines: generators and assemblers. Generators produce parts and assemblers consume them to create new ones. Machines take their inputs from trays and put their results in conveyors. Users collect fallen parts from the plant floor and assembled hammers from trays.

Fig. 2 shows a production model example using a visual concrete syntax. The model consists of one plant with two generators, connected to two different conveyors. These conveyors are connected to the same tray from which parts are assembled, deposited in a third conveyor, and finally stored in another tray. An operator collects hammers from the final tray and fallen parts from the plant

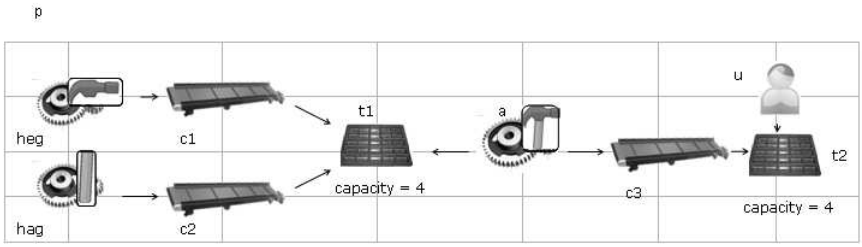

Fig. 2. `initModel` example production system.

floor. The position of each element is dictated from its position in the plant, depicted as a grid.

The way to specify the behavior of the system using in-place transformation rules is illustrated in Fig. 3, that shows one of the permitted actions: how conveyors shift pieces. When a part is placed on a conveyor (as described by the LHS pattern of the rule), it is shifted to the conveyor output tray (as shown in the RHS), and the part takes the tray's position.

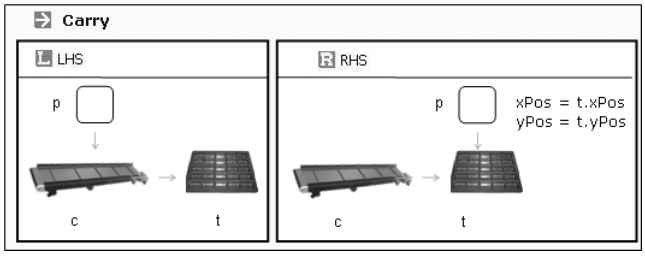

Fig. 3. The `Carry` rule.

2.2 Extending In-Place Transformation Rules with Time

Current in-place transformation techniques 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 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 Sec. 5).

One way to avoid this problem is to extend the rules with the time they consume, i.e., by assigning to each action the time it takes. In this way, a time elapse is associated to each action or piece of behavior (i.e., each in-place rule), without having to unnaturally modify the DSL metamodel to incorporate time information. Furthermore, if time is specified as an interval $[t, t']$ (representing

the minimum and the maximum amount of time needed to perform an action) instead of using a single value, we allow time delays to be easily included in the specifications. Thus, we define *timed rules* as in-place transformation rules of the form $l : [\text{NAC}] \times \text{LHS} \xrightarrow{[t,t']} \text{RHS}$, where $[t, t']$ expresses the duration interval of the action modeled by the rule, in some time granularity. Whenever the minimum and the maximum values are the same, we will simplify the timed rules representation as $l : [\text{NAC}] \times \text{LHS} \xrightarrow{[t]} \text{RHS}$.

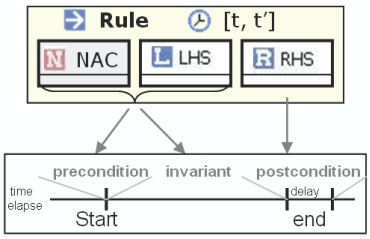

Fig. 4. Actions modeled using timed rules.

Timed rules admit different semantics. In its simplest form, the LHS and NAC patterns express preconditions for the rule to be triggered. If they occur, the action specified by the rule is scheduled to be applied after between t and t' time units. In that interval of time, the preconditions are again evaluated and if they still hold, the rule is applied by substituting the match by the RHS (which expresses the postcondition). In another semantic variation, the LHS and NAC patterns should hold not only at the beginning and at the end of the action, but also throughout its duration, acting as *invariants*. Further extensions to the rules are also possible, such as defining specific invariant patterns. In our current approach, the LHS and NAC pattern conditions act both as preconditions and invariants, i.e., timed rules are applied if the LHS and NAC patters hold during the whole duration of the action (see Fig. 4).

2.3 Accessing the Global Time Elapse

So far, the notion of time has been related to the duration of actions (rules). However, this simple model of timed actions may not be suitable for modeling other time constraints, such as time stamps or scheduled actions (actions that take place at some particular moment of time).

Thus, we have included in our specifications 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 (from its attribute `time`). Provided that the clock behavior cannot be modified, users cannot drive the system to time-inconsistent sequences of states (even unwillingly).

2.4 Extending Transformation Rules with 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 that are being executed, of to those that have been executed in the past. For example, we do not want a generator to produce a new piece until it has finished producing one, or we do not want an operator to pick up a part while he is moving. In general, this limitation hinders the specification of many useful action properties, unless some unnatural changes are introduced in the system model (cf. [3]).

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 rule executions that are currently happening or that were previously performed. Each action specification defines several *roles*, one for each of its participant objects (e.g., the `Carry` rule defines three roles: the part `p`, the conveyor `c` and the tray `t` — see Fig. 3). Thus, the objects that participate in these action occurrences (playing one of the action roles) need normally to be also specified; for example, we need to specify the particular operator who is moving, because the rest of the operators can be, of course, engaged in other actions. In our proposal, the set of objects involved in an action 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`).

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 conveyor may be shifting several pieces at a time. 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), as shown in the next section.

3 A Graphical Model Editor for Timed Specifications

In this section we present the graphical model editor implemented to support the specification of timed rules. For this purpose, we selected the Eclipse Graphical Modeling Framework (GMF [8]). GMF provides a generative component and runtime infrastructure for developing graphical model editors for DSLs. It automatically generates an Eclipse plugin with a DSL diagram editor from (1) the DSL metamodel (abstract syntax); (2) a graphical definition (concrete syntax); (3) a tooling definition (namely, the buttons that enable the creation of the model elements); and (4) a mapping model relating the three previous artifacts.

In our case, the domain model, named *Behavior Metamodel*, is shown in Fig. 5. The dynamic behavior of a DSL is specified by a set of rules. Each rule

expresses the preconditions (LHS and NAC patterns) and postcondition (RHS pattern) of an action to be performed (as in standard in-place transformation approaches). The novelty in this metamodel is the addition of the `maxTime` and `minTime` rule attributes (to represent the duration of the rule), and the inclusion of the `ActionExec` and `Clock` metaclasses. `ActionExec` instances represent action executions, while the unique and read-only instance of `Clock` represents the current global time elapse. 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 [5], although adapted to the tree-structure of Eclipse models [9].

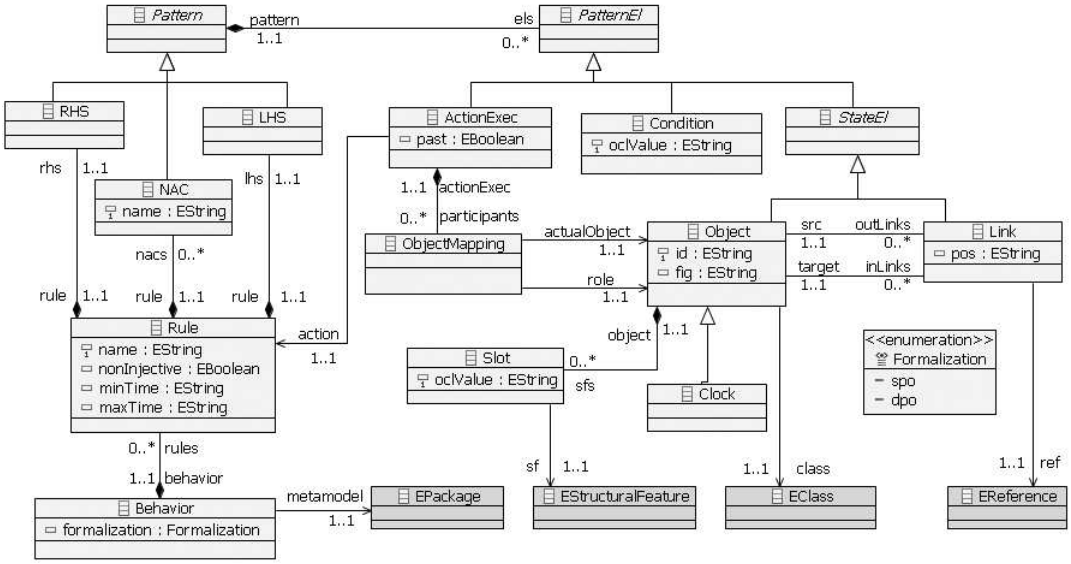

Fig. 5. The Behavior Metamodel.

Fig. 6 shows a snapshot of the graphical tool, which has been used to visually specify the `Carry` action. The tool comprises two editors: the *behavior editor*, that allows users to specify the set of rules; and the *rule editor*, that allows us 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 the behavioral rules. Therefore, when an object is included in a pattern and its class is defined, the object is drawn using the image associated to its class (taken from its corresponding DSL GMF specifications, if defined). Otherwise, a default image (a rectangle) is depicted.

The editor also enables the specification of timed rules (in this paper we have considered only discrete time, although dense time can also be handled).

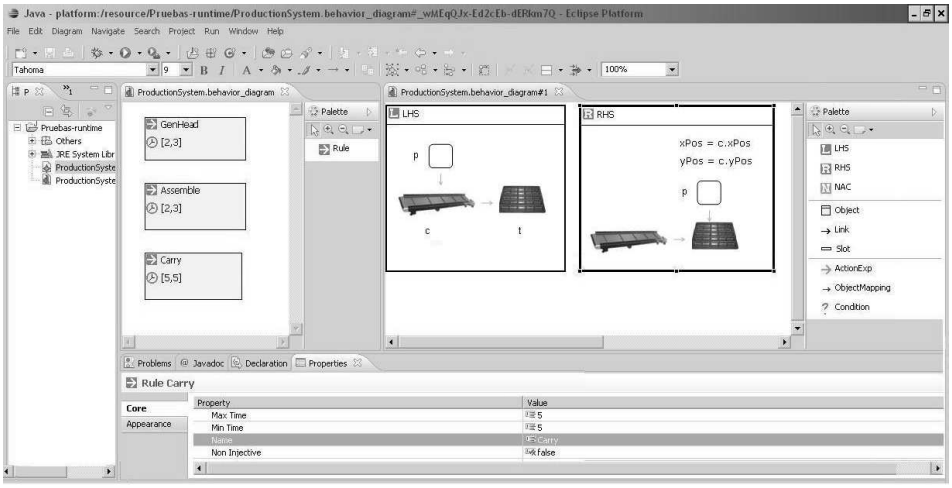

Fig. 6. Behavior and rule editors.

For instance, we can specify that the `Carry` rule takes exactly five time units. Note that the only difference with standard in-place rules is the specification of the time consumed by the rule (in this case five time units).

However, this small change has some significant semantic implications. What happens if a new part is placed on the conveyor when it is shifting another part? Since there is no restriction about this situation and an occurrence of the LHS pattern is found in the model, the conveyor will move both parts, i.e., conveyors can shift several pieces at a time, and each part will stay five time units over it.

Fig. 7 shows the behavior of handle generators. A handle generator takes between two and three time units to create a handle and to put it on its output conveyor (one way to interpret this time duration is that it produces a handle in two time units, with a possible delay of one time unit). Since generators can only create one part at a time, we have to restrict their behavior by including the corresponding action execution in the NAC. The action execution will forbid that the corresponding generator `hg` starts creating a new handle if it is already generating one. Please note that arrow $\rightarrow$ of action executions (placed on the left of the rule's name) represents that the action is being performed in that very moment, while arrow $\leftarrow$ represents that the action has already been performed.

More complex actions and time expressions can also be specified in this way. Fig. 8 shows further rules defined for the sample DSL. Among them, we can find the behavior of users when collecting hammers from trays (rule `Collect`) and when picking parts up from the plant floor (rule `PickUp`). In this case, users can only take (i.e., collect or pick up) one part at a time, and the time the user consumes depends on the (Manhattan) distance between the part and the user himself. Note the use of OCL expressions for specifying conditions, attribute computations and time expressions.

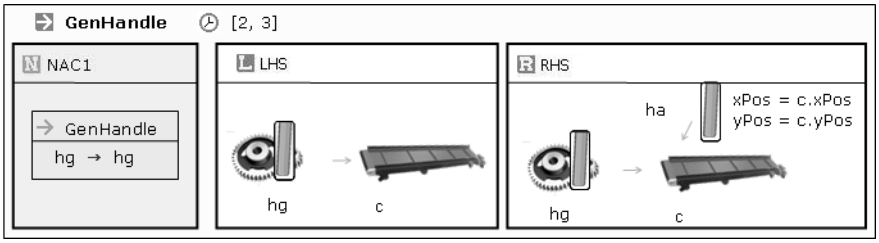

Fig. 7. `GenHandle` rule.

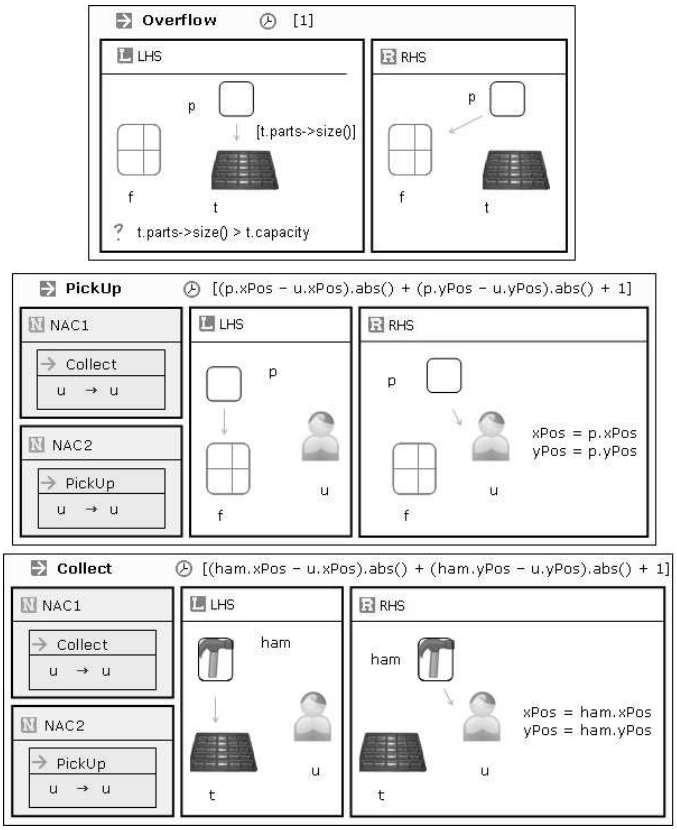

Fig. 8. Further production system rules.

4 Analyzing Timed Behavioral Specifications with 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 [10].

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, which allows us to perform different types of analysis with RTMaude. In this way, the designer will have the advantage of a visual and intuitive specification, and an efficient and powerful mean for analysis.

4.1 Real-Time Maude

Real-time Maude [11, 4] 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 $t \rightarrow t'$, 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 extensions of 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 whole system time elapse. In RTMaude, both discrete and dense time are implemented, and users can define their own time domain.

4.2 Encoding Timed Rules in Real-Time Maude

Maude has already been proposed as a formal notation and environment for specifying and effectively analyzing models and metamodels [12, 13]. In [5] we showed how Maude is suitable as a semantic domain for standard in-place rules. This paper shows how the timed behavioral specifications can also be supported.

In RTMaude, time elapse is usually modeled by one tick rule and the system dynamic behavior by instantaneous transitions [11]. Thus, timed rules can be naturally translated into RTMaude rewrite rules. In our proposal, a timed rule will be encoded as three RTMaude instantaneous rules, each one modeling one of the rule statements: the precondition, the postcondition and the invariant.

– *The precondition rule.* When the rule precondition is satisfied, i.e., when an occurrence of the LHS is found in the model that does not fulfill any of the NAC patterns ($LHS \wedge \neg NAC$), an `ActionExec` object is created.

`ActionExec` objects represent rule executions, each one acting as a countdown (attributes `minTimer` and `maxTimer`) to the finalization of the rule, and gathering all the information needed for its instantiation: the rule name (attribute `ruleLabel`) and the identifiers of the objects that participate in it (attribute `participants`). Initially, the limits of the interval, `minTimer` and `maxTimer`, are respectively set to the attributes `minTime` and `maxTime`, which represent the duration of the rule. Additionally, the same `ActionExec` object, but with a `minTimer` value equal or greater than zero (time is considered positive) is included in the rule as a NAC, in order to avoid indefinite executions.

- *The postcondition rule.* When the precondition ($LHS \wedge \neg NAC$) and the invariant (in the current approach the invariant coincides with the precondition) are satisfied during the action execution, and the minimum time defined for the rule is consumed (i.e., there is an `ActionExec` object that refers to the rule with `minTimer=0`), the action can be performed. Then, the LHS is substituted by the RHS and the attribute computations are done. To keep track of the performed actions, the `ActionExec` object is saved but with `minTimer=null`. Please note that, except for the presence of the `ActionExec` object, the postcondition rule corresponds to the standard encoding of in-place rules.
- *The invariant rule.* If the invariant breaks ($\neg LHS \vee NAC$) at any time during the action execution (i.e., there is an `ActionExec` object that refers to the rule whose `minTimer` is equal or greater than zero) the action is interrupted, i.e., its corresponding `ActionExec` object is destroyed.

Rule elements are encoded as follows: 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 operations placed in the 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); action executions are encoded by means of `ActionExec` objects that refer to the corresponding rule with provided object identifiers instantiation; and the clock object is encoded as a RTMaude object belonging to the predefined class `Clock`.

As mentioned above, the proposed timed rule encoding in RTMaude allows elements to simultaneously perform different actions when they instantiate different rules at the same time. Furthermore, it makes DSL objects to be completely unaware of time, and therefore time elapse needs only be defined over `ActionExec` and clock objects.

- *The tick rule.* The tick rule models time elapse by using two functions: `delta` and `mte` (maximal time elapse). Function `delta` defines the effect of time elapse over every model element, and function `mte` 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. Function `delta` operates

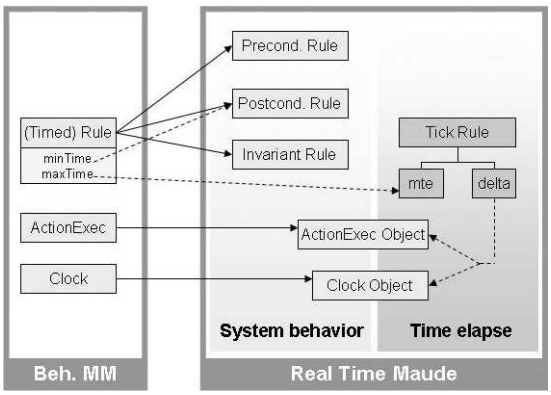

Fig. 9. Semantic Mappings to Real-Time Maude

only over `ActionExec` and clock objects: it decreases `ActionExec` timers (both `minTimer` and `maxTimer`) in `T` units, and increases the clock value in `T` units too. Function `mte` is defined as the minimum `maxTimer` value of current `ActionExec` objects. In this way, all rules will be executed before their `maxTime`, due to function `mte`, and after `minTime`, due to the corresponding precondition rule (see Fig. 9).

4.3 Analysis and Simulation with Real-Time Maude

Once the system specifications are encoded in RTMaude, we obtain a rewriting logic specification of the system. As these specifications are executable, they can be used to simulate and analyze the system. RTMaude offers tool support for interesting analysis possibilities such as model simulation, reachability analysis and model checking, which can be naturally used with the RTMaude specifications we produce (see, e.g., [5, 14] for examples of these kinds of analyses).

5 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 [5] for a comprehensive survey). Graphical notations are proving extremely helpful in software and systems development, since they are intuitive and more natural both to specify and to understand the behavior of complex systems.

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 [15], 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 [16] (an approach to modeling time in high-level Petri nets), does not extend the base formalism but specialize 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 [17] 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 a previous work [5], we have formalized graph transformations using equational and rewriting logic with Maude [13]. This allows performing simulation, reachability and model-checking analysis using the tools and techniques provided by Maude. In this paper we have extended in-place rules with time, and therefore we have moved to RTMaude to model time elapse and to perform the same kind of analysis. Our approach supports attribute computation, ordered collections, and OCL expressions. Besides, 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.

6 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 makes 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. We are also studying the different semantic possibilities that can be supported by timed rules, such as adding special patterns to define invariants. In this vein, we are weighing up the additional effort that means for the user to define a new pattern, against the improved level of expressiveness that it provides.

Acknowledgements. The authors would like to thank Francisco Durán and Peter C. Ölveczky for their comments on previous versions of the paper, and

also to the anonymous referees for their insightful comments and very constructive suggestions. This work has been supported by Spanish Research Projects TIN2008-00889-E, TIN2008-03107 and P07-TIC-03184.

References

1. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture. (2003)
2. de Lara, J., Vangheluwe, H.: Translating model simulators to analysis models. In: Proc. of FASE 2008. Number 4961 in LNCS, Springer (2008) 77–92
3. Meseguer, J.: The temporal logic of rewriting: A gentle introduction. In: Concurrency, Graphs and Models. (2008) 354–382
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude – A High-Performance Logical Framework. Number 4350 in LNCS. Springer, Heidelberg, Germany (2007)
5. Rivera, J.E., Guerra, E., de Lara, J., Vallecillo, A.: 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 (2008)
6. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Analysing graph transformation rules through OCL. In: Proc. of ICMT 2008. Number 5063 in LNCS, Springer (2008) 225–239
7. Vangheluwe, H., de Lara, J.: Automatic generation of model-to-model transformations from rule-based specifications of operational semantics. In: Proc. of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07). (2007)
8. Eclipse: Graphical Modeling Framework (2008) `http://www.eclipse.org/modeling/gmf/`.
9. Biermann, E., Ehrig, K., Khler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical definition of in-place transformations in the Eclipse Modeling Framework. In: MDE Languages and Systems. Number 4199 in LNCS, Springer (2006)
10. Vallecillo, A.: A journey through the secret life of models. In: Model Engineering of Complex Systems (MECS). Number 08331 in Dagstuhl Seminar Proceedings (2008) `http://drops.dagstuhl.de/opus/volltexte/2008/1601`.
11. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation **20**(1-2) (2007) 161–196
12. Rivera, J.E., Vallecillo, A.: Adding behavioral semantics to models. In: Proc. of EDOC 2007, IEEE Computer Society (2007) 169–180
13. Romero, J.R., Rivera, J.E., Durán, F., Vallecillo, A.: Formal and tool support for model driven engineering with Maude. Journal of Object Technology **6**(9) (2007) 187–207
14. Ölveczky, P.C.: Real-Time Maude 2.3 Manual. (2007) `http://www.ifi.uio.no/RealTimeMaude/`.
15. Gyapay, S., Heckel, R., Varró, D.: Graph transformation with time: Causality and logical clocks. In: Proc. of 1st Int. Conference on Graph Transformation (ICGT'02), Springer-Verlag (2002) 120–134
16. Ghezzi, C., Mandrioli, D., Morasca, S., Pezzè, M.: A unified high-level petri net formalism for time-critical systems. IEEE Trans. Softw. Eng. **17**(2) (1991) 160–172
17. Syriani, E., Vangheluwe, H.: Programmed graph rewriting with time for simulation-based design. In: Proc. of the International Conference on Model Transformation (ICMT 2008). Number 5063 in LNCS, Springer-Verlag (2008) 91–106