Simulating Domain Specific Visual Models by Observation

Javier Troya, José E. Rivera and Antonio Vallecillo
GISUM/Atenea Research Group. Universidad de Málaga, Spain
{javiertc,rivera,av}@lcc.uma.es

Keywords: Domain specific visual languages, behavioral semantics, observers, Maude

Abstract

Domain Specific Visual Languages (DSVLs) are essential elements in Model-Driven Engineering (MDE) for representing models and metamodels. In-place model transformations provide an intuitive way to complement metamodels with behavioral specifications. Besides, they can be extended with quantitative models of time and with mechanisms that facilitate the design of real-time complex systems. In this paper we present an approach to simulate and analyze the behavior of systems described by DSVLs using *observers*, which are objects that can monitor both the state of the rest of the objects of the system, and the execution of the system actions. Our proposal is supported by e-Motions, a graphical framework and tool for defining timed behavioral specifications of models. We also show how this approach enables the specification and simulation of other important features of systems, such as the automatic reconfiguration of the system when the value of some of the observed properties change.

1. INTRODUCTION

Domain specific visual languages (DSVLs) play a cornerstone role in Model-Driven Engineering (MDE) for representing models and metamodels. The benefits of using DSVLs is that they provide an intuitive notation, closer to the language of the domain expert, and at the right level of abstraction. The Software Engineering community's efforts have been progressively evolving from the specification of the structural aspects of a system to modeling its behavioral dynamics. Thus, several proposals already exist for modeling the structure and behavior of a system. Some of these proposals also come with supporting environments for animating or executing the specifications, based on the transformations of the models into other models that can be executed [5, 6, 7].

The normal way in which the DSVL models are simulated and analyzed (to obtain information about, e.g., mean-time between failures, end-to-end throughput, or idle-time of some of the system elements) is based on the idea of annotating the models with data related to the properties we want to simulate, and then translating the annotated models into models ready to be analyzed by the appropriate tools. Most of these proposals exist only for UML-based notations, with UML Profiles such as UML-QoSFT, UML-SPT or MARTE [8, 9, 10]. They allow annotating UML models with time and QoS information and a set of requirements on the behavior of the system (e.g., response time, jitter or throughput).

In this paper we present an alternative approach to specify the properties that need to be simulated, integrating new objects in the specifications that capture such properties. Our proposal is based on the *observation* of the execution of the system actions and of the state of its constituent objects. We use this approach to simulation and analysis in the case of DSVLs that specify behavior in terms of rules (which describe the evolution of the modeled artifacts along some time model), and illustrate the proposal with the running example of a Production Line system. We show how, given an initial specification of the system, the use of observer objects enables the analysis of some of the properties usually pursued by simulation, including mean and max cycle-time for the produced parts, busy and idle cycles for every machine in the system, mean-time between failures, etc. Analogously, further properties can be easily simulated using this approach.

We show as well how this approach enables the specification of other important features of systems, such as the automatic reconfiguration of the system when the value of some of the observed properties change.

2. MODELING BEHAVIOR

One way of specifying the dynamic behavior of a DSVL is by describing the evolution of the modeled artifacts along some time model. In MDE, this can be done using model transformations supporting in-place update [3]. The behavior of the DSVL is then specified in terms of the permitted actions, which are in turn modeled by the transformation rules.

There are several approaches that propose in-place model transformations to deal with the behavior of a DSVL, from textual to graphical (see [12] for a brief survey). This approach provides a very intuitive way to specify behavioral semantics, close to the language of the domain expert and the right level of abstraction [4]. In-place transformations are composed of a set of rules, each of which represents a possible *action* of the system. These rules are of the form $l : [\mathrm{NAC}]^* \times \mathrm{LHS} \rightarrow \mathrm{RHS}$, where l is the rule's label (its name); and LHS (left-hand side), RHS (right-hand side), and NAC (negative application conditions) are model patterns that represent certain (sub-)states of the system. The LHS and

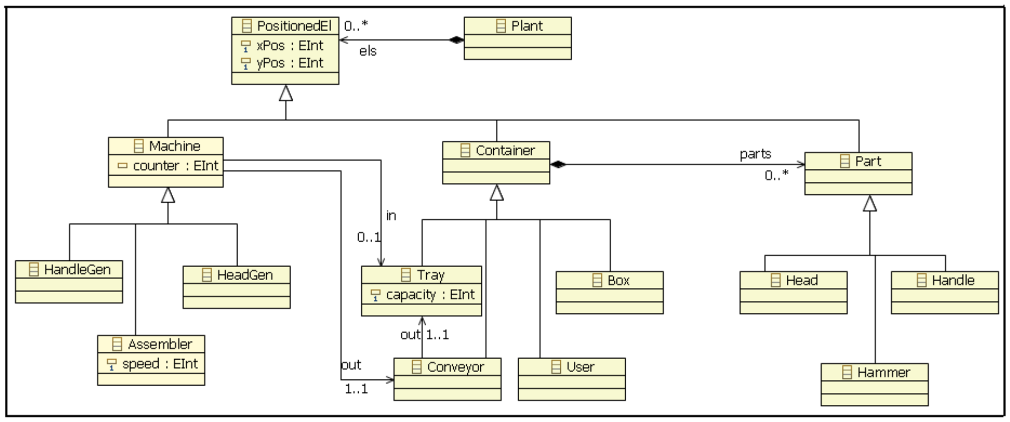

Figure 1. Production Line Metamodel.

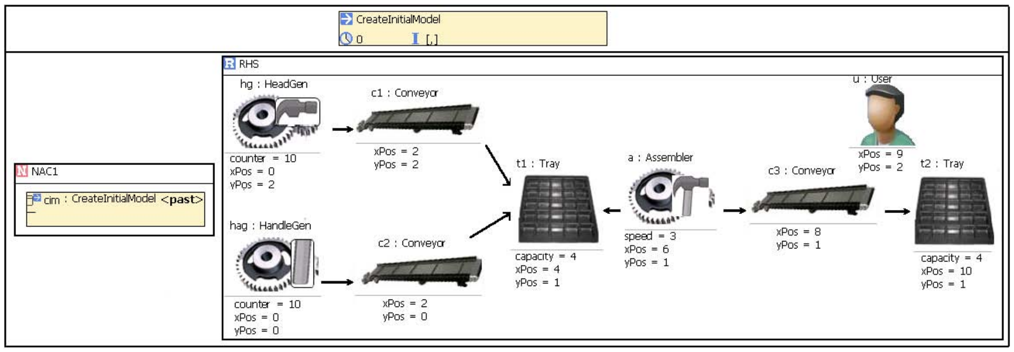

Figure 2. Creation of the initial configuration of the system.

NAC patterns express the precondition for the rule to be applied, whereas the RHS one represents its postcondition, i.e., the effect of the corresponding action. 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 its 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 appropriate instantiation of its RHS pattern (the rule's *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 [13].

Modeling Time-Dependent Aspects. In [11] we showed how time-related attributes can be added to rules to represent features like duration, periodicity, etc. We defined two types of rules to specify time-dependent behavior, namely, *atomic* and *ongoing* rules. The former ones represent atomic actions, with a specific duration. They can be canceled, but cannot be interrupted. Examples of this kind of actions include generating a piece, or collecting it from a tray. These rules can be periodic, i.e., they admit a parameter that specifies an amount of time after which the action is periodically triggered (if the rule's precondition holds, of course). *Ongoing* rules represent interruptible continuous actions which progress along time while the rule preconditions (LHS and not NACs) hold. This kind of rules is used, for instance, to represent how the state of some observers changes with time (Figs. 10 and 11). Both atomic and ongoing rules can be scheduled, or be given an execution interval.

We have also included the explicit representation of *action executions*, which describe actions currently executing. They 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. This provides a very useful mechanism when we want to check whether an object is participating in an action or not, or if an action has already been executed, for instance.

Finally, a special kind of object, named Clock, represents the current global time elapse. Designers are allowed to use it in their timed rules (using its attribute time) to look up the amount of time that the system has been working. Further clocks can be specified by users, according to the requirements of their systems (to model, for instance, systems with distributed clocks).

A running example. To illustrate how the behavior of a system can be modeled using our approach, we will show an example of a hammer production line. The metamodel for such system is depicted in Fig. 1. There are different kinds of machines (head generators, handle generators and assemblers), containers (trays, conveyors and users) and parts (head, handles, and hammers), and they all have a position. Generators produce parts and deposit them in trays; conveyors move parts from machines to trays; assemblers consume parts from trays to create hammers, which are deposited in conveyors, and finally collected by users. Trays contain parts up to their capacity. For creating an initial model of the system, we create a rule (Fig. 2) whose LHS pattern is empty (and not shown here) and where the initial model is created in the RHS pattern. For this rule to be applied, it must not have been executed before. This is modeled by an action execution, cim, in the NAC pattern NAC1 that indicates that the action CreateInitialModel was executed before (see the past attribute).

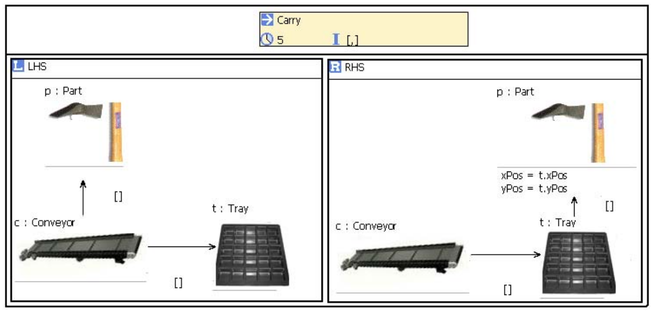

Figure 3. Carry Rule

As an example of the rules that describe the possible actions of the system, Fig. 3 shows the atomic Carry rule, which specifies how a Part is transported through a Conveyor to the Tray connected to the end of it. This action takes 5 time units.

3. NON-FUNCTIONAL PROPERTIES

Once we count on languages for specifying models and their behavior, the next step is to try to simulate the system, and/or to reason about it. There are three main issues associated to this simulation. Firstly, we need to be able to express the properties that we want to simulate (e.g., delays, mean-time between failure, or end-to-end throughput). Secondly, once we have specified such properties we need to have a simulation engine that executes the specifications. Finally, we have to introduce somehow this simulation engine in the behavior of our system. In this section we describe our proposal to model the simulation properties of the system as well as the execution engine.

3.1. Properties to simulate

Let us suppose then that we want to analyze the following five parameters of the Production Line system:

Throughput: number of pieces produced by the system per unit of time.

Mean time between failures (MTBF): mean time between pieces lost (parts can fall from conveyors when the destination tray is full).

Idle-time: Amount of time that a machine has not been working (i.e., idle or waiting) so far.

Mean Idle-time: Average idle time of all machines.

Cycle time (of a produced hammer): Amount of time between the first part of the hammer is generated, and the hammer is finally collected by the user.

Mean cycle time (MCT): Average cycle time of all produced hammers so far.

3.2. Observers

To calculate the value of these simulation parameters we propose the use of *observers*. An observer is an object whose purpose is to monitor the state of the system: the state of the objects, of the actions, or both. For this particular example we will use two kinds of observers, depending on whether they monitor the state of the complete system, or the state of individual objects. In the former case, observers are created with the system and remain during its whole life. In the latter case, observers are created with the objects they monitor, and destroyed with them.

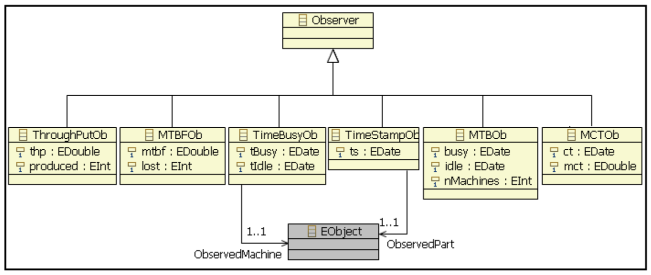

Figure 4. Observers Metamodel

Observers, as any other objects, have a state and a well-defined behavior. The attributes of the observers capture their state, and are used to store the variables that we want to monitor. We have defined an *Observers metamodel*, which is shown in Fig. 4. There are six different observers which inherit from a general Observer class. The ThroughPutOb observer stores in its attributes the throughput and the number of produced pieces. The MTBFOb observer stores the mean time between failures and the number of lost pieces. Then we

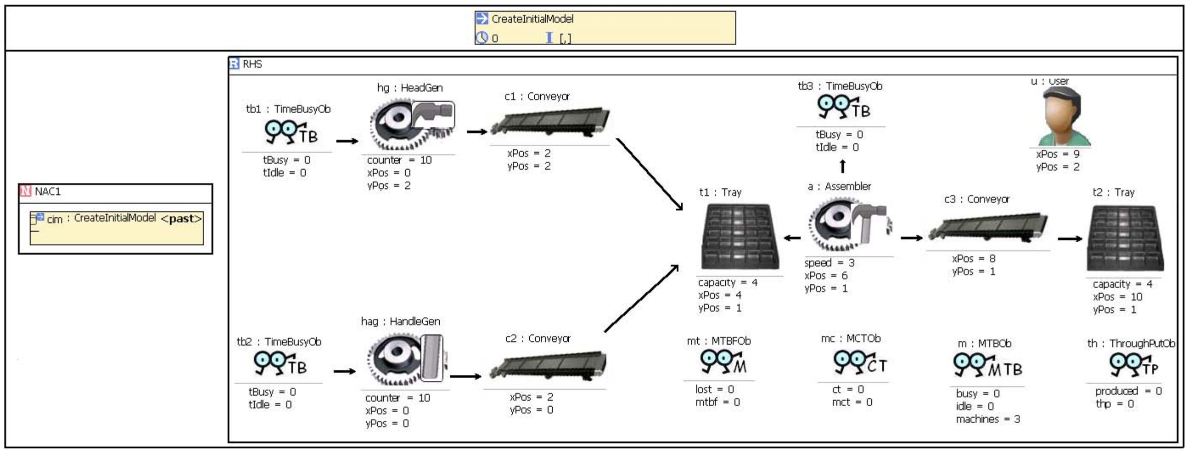

Figure 5. Creation of the initial configuration of the system with Observers

have the TimeBusyOb and TimeStampOb observers, which are associated to machines and parts, respectively. The former stores the time the associated machine is busy and idle. The latter stores the time its associated part appears in the system. One MTBOb observer stores the mean time that all the machines in the system have been working and waiting, and also stores the number of machines in the system. Finally, the MCTOb observer computes the mean cycle time of every produced piece, and also stores the addition of all the cycle times of every piece.

Note that TimeBusyOb and TimeStampOb observers have a reference to an EObject, which is the interface implemented by every model object in Eclipse. In this way, any of these two observers can be associated to any of the elements of another metamodel.

3.3. Observers' behavior

The idea for simulating the system with observers is to combine the two metamodels (Figs. 1 and 4) to be able to use the observers in our production line language. In fact, since *e-Motions* allows users to merge several metamodels in the definition of a DSVL behavior, we can define the *Observers metamodel* in a non-intrusive way, i.e., we do not need to modify the system metamodel to add observers in their rules. Furthermore, this approach also enables the reuse of observers across different DSVLs.

So firstly, we modify our CreateInitialRule rule, which creates the inital model, by adding observers to our initial model (Fig 5). There are four observers for the whole system (ThroughPutOb, MTBFOb, MTBOb and MCTOb), and three TimeBusyOb observers, each of them associated to a machine to control the time it is working.

Then, the behavior of the observers is specified using rules, too. Hence, not only the model of the system is enriched with observers, but also the rules are. Consequently, hereafter these rules will not only describe how the system behaves, but also how the observers monitor the system.

It has to be taken into account that, with our *e-Motions* tool [11], we are able to specify the duration of in-place rules with any OCL expression. In particular, statistical distributions can be used to specify rule durations.

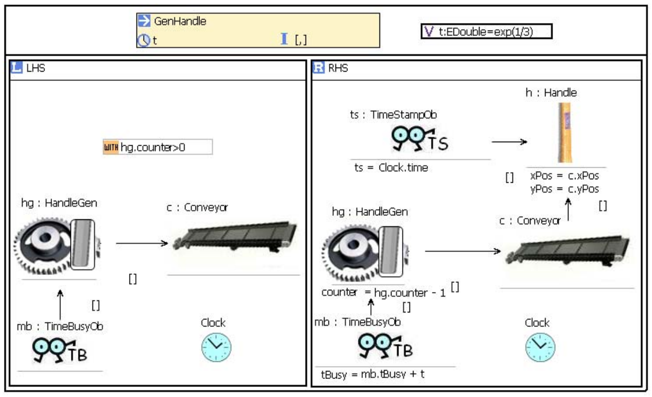

Figure 6. GenHandle Rule

For example, Fig. 6 shows the atomic rule GenHandle, which creates handles following an Exponential distribution with parameter $\lambda = 1/3$, i.e., the average time between two produced parts is 3 time units. ($Exp(\lambda) = -\ln(random(0,1))/\lambda$). For this rule to be applied, the LHS pattern indicates that the system has to have a HandleGen generator which has to be, in turn, connected to a Conveyor. Note that this rule is only applied if the attribute counter of the GenHandle is bigger than 0. In the RHS pattern a new Handle is generated, and it acquires the position of the Conveyor connected to the HandleGen machine. The counter value of the GenHandle is decreased in 1 unit, which models that a new handle has been created. A TimeStampOb observer has been created and it is associated to the Handle, storing in its

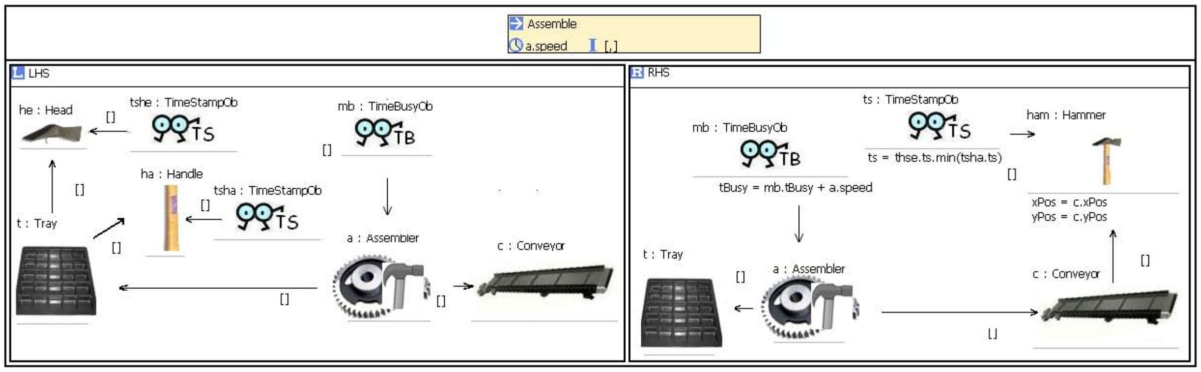

Figure 7. Assemble Rule

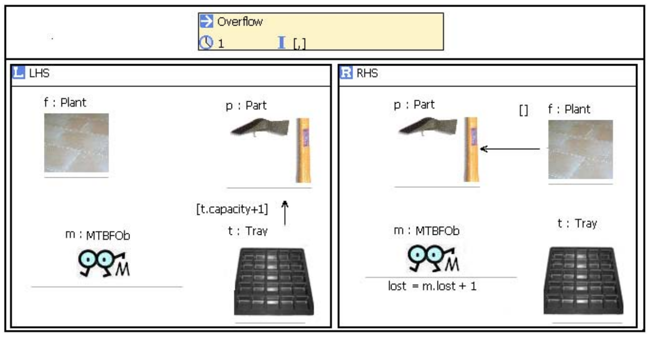

Figure 8. Overflow Rule

ts attribute the time at which the Handle has appeared in the system. Finally, the tBusy attribute of the TimeBusyOb is increased in $t = Exp(1/3)$ times unit, as the HandleGen spends this time in producing a new Handle. Analogously, a GenHead rule (not shown here for space reasons) creates a new Head according to an exponential distribution with parameter $\lambda = 1/5$ (we suppose that it takes more time to generate a head).

The Assemble rule is shown in Fig. 7. This rule models the behavior of generating a Hammer from one Head and one Handle. We can see in the LHS pattern that the Tray which is connected to the Assembler has to contain a Head and a Handle. In the RHS pattern we can see how the Head, the Handle and their associated observers have all disappeared and a new Hammer has been created in the position of the Conveyor connected to the assembler. It has a TimeStampOb observer associated whose ts attribute is the lower value between the time stamp of the Head and the Handle. The time spent by this rule depends on the speed attribute of the Assembler. So we can see how the TimeBusyOb observer associated to the Assembler has increased its tBusy value according to the Assembler's speed, since it has been working that time. (For illustration purposes this rule takes a fixed amount of time, instead of following a statistical distribution.)

The Overflow rule (Fig. 8) models the behavior of the system when a part is lost. The LHS pattern indicates that this rule is triggered when a part is placed in a Tray that contains more parts than its capacity allows. The MTBFOb observer is present because it will be modified in the RHS pattern. In this pattern the number of lost pieces of the observer is increased and the Part is not in the Tray anymore. Nevertheless, the Part does not disappear from the system, so it gets associated to the Plant floor.

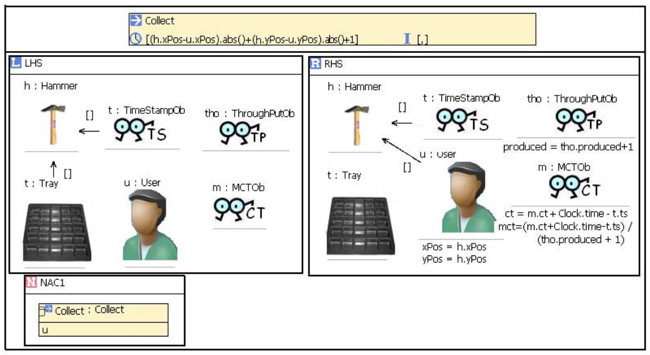

Figure 9. Collect Rule

Fig. 9 shows the Collect rule. This rule models the behavior of the system when the User finally takes an assembled Hammer. The User acquires the position of the Hammer and gets associated to it. We see as well the presence of three observers. The produced attribute of the ThroughPutOb observer is increased in one unit, since a new Hammer has been successfully produced and collected by the User. The mean cycle time of every produced Hammer has to be updated every time a new one is produced. This is why the two attributes of the MCTOb observer are updated by this rule. The first one, ct, is increased with the cycle time of the new Hammer, which is $Clock.time - ts.ts$ (here, $ts.ts$ is the time stamp of the Hammer). Then, the mct attribute divides the ct value by the total number of produced Hammers. Note that every attribute that appears in the RHS pattern refers to its LHS pattern value. The time this rule spends is the Manhattan distance between the Hammer and the User. There is also a NAC in this rule, which forbids users collect more than one hammer at a time. This is expressed by an action execution that represents the action (Collect) being executed by the same user (u).

In two previous rules (Fig. 6 and Fig. 7) we showed how TimeBusyOb observers update their tBusy attributes. Now we show with an ongoing rule how the tIdle attribute of the TimeBusyOb observer associated to the Assembler is updated. This rule is shown in Fig. 10. It is executed as long as the Assembler is not assembling a Hammer, as it is specified in the NAC1. In the LHS we see the Assembler and its associated observer. In the RHS pattern, the tIdle attribute is updated by adding its previous tIdle value to the amount of time that the Assembler is not working (which is the duration of the rule). Since this is an ongoing rule and the LHS pattern is always present, it is executing whenever the Assembler is not working, so the amount of time this rule is executing is the amount of time that no Hammer is being assembled by this machine.

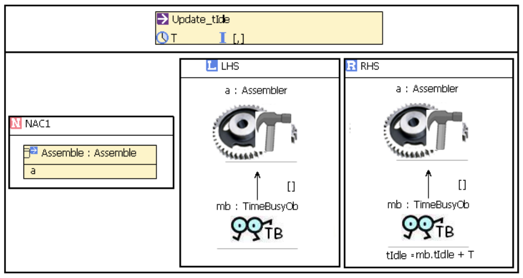

Figure 10. UpdatetIdle Rule

Another ongoing rule is used to keep the state of the ThroughPutOb, MTBFOb and MTBOb observers updated (Fig. 11). The Clock is present in the rule because we need to know the amount of time that the system has been working. Thus, the mtbf attribute of the MTBFOb observer is obtained by dividing this value by the number of lost parts, which is also stored by this observer. The thp attribute of the ThroughPutOb observer is calculated by diving the number of produced hammers, stored by this observer, by the amount of time the system has been working. Regarding the MTBOb, it keeps the mean time that all the machines of the system have been working and waiting. To obtain those values, we use two OCL expressions which collect all the instances of TimeBusyOb observers (which are associated to machines) present in the system and calculate the addition of their tBusy and tIdle values, respectively. Then, the obtained values are divided by the number of machines of the system. This way we keep the state of these three observers of the whole system updated all the time (note that the LHS pattern is always present in the model).

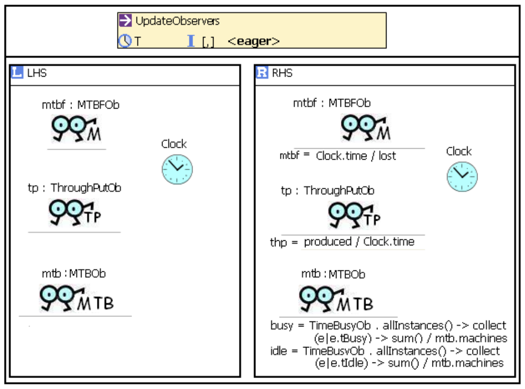
UpdateObservers
T
[,]
<eager>
LHS
mtbf : MTBFOb
M
Clock
tp : ThroughPutOb
TP
mtb : MTBOb
MTB
RHS
mtbf : MTBFOb
M
mtbf = Clock.time / lost
Clock
tp : ThroughPutOb
TP
thp = produced / Clock.time
mtb : MTBOb
MTB
busy = TimeBusyOb . allInstances() -> collect (e|e.tBusy) -> sum() / mtb.machines
idle = TimeBusyOb . allInstances() -> collect (e|e.tIdle) -> sum() / mtb.machines

Figure 11. Update Observers Rule

4. ANALYSIS OF THE SYSTEM

The kind of rules described in the previous section allow users to model the behavior of their systems. We have developed an Eclipse environment called *e-Motions* [11] that supports the specification of such kind of models and rules.

Once the specifications are written, our environment supports their translation (using ATL model transformations) into the corresponding formal specifications in Maude [2]. The goal of such formal specifications is to provide formal semantics to the visual specifications of the system. In addition, since the Maude rewriting logic specifications are executable, they can be used as a prototype of the system, which allows us to simulate and analyze it. Maude offers tool support for interesting possibilities such as model simulation, reachability analysis and model checking of this kind of specifications [13].

Simulation. Maude also offers different possibilities for realizing the simulation, including step-by-step execution, several execution strategies, etc. In particular, Maude provides two different rewrite commands, namely rewrite and frewrite, which implement two different execution strategies, a top-down rule-fair strategy, and a depth-first position-fair strategy, respectively [2]. The result of the process is the final configuration of objects reached after the rewriting steps, which is nothing but a model. As an example, Fig. 12 shows the resulting model after simulating the system for 16 time units. It contains the current state of the objects in the system.

Reachability analysis. Executing the system using the rewrite and frewrite commands means exploring just one possible behavior of the system. However, a rewrite system do not need to be Church-Rosser and terminating,[1] and there

[1]For membership equational logic specifications, being Church-Rosser and terminating means not only confluence—a unique normal form will be reached—but also a sort decreasingness property, namely that the normal form will have the least possible sort among those of all other equivalent

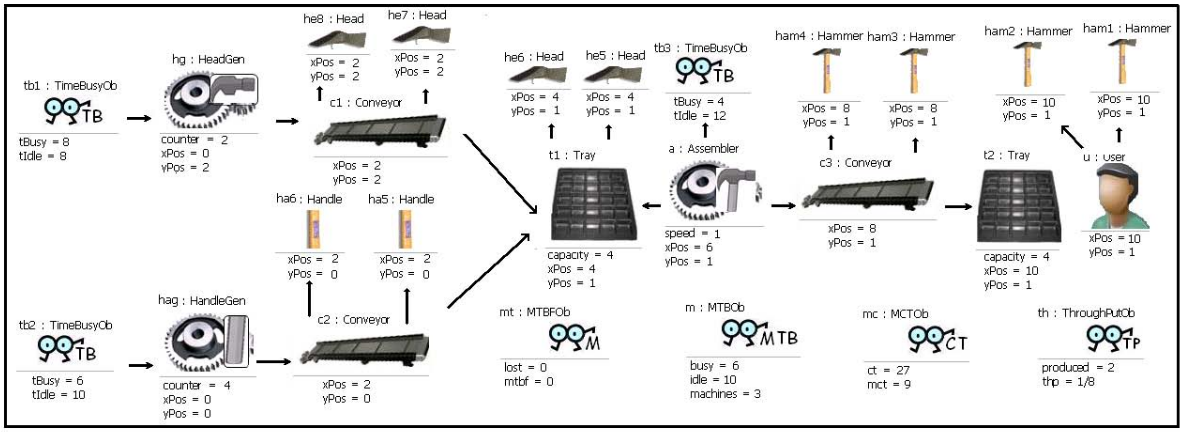

Figure 12. Resulting model of the simulation after 16 time units.

might be many different execution paths. Although these commands are enough in many practical situations where an execution path is sufficient for testing executability, the user might be interested in exploring all possible execution paths from the starting model, a subset of these, or a specific one.

Maude search command allows us to explore (following a breadthfirst strategy up to a specified bound) the reachable state space in different ways. For example, we could look for certain states of special interest, such as deadlock states—i.e., states in which no further rewrite may take place. Maude allows us to specify a model of the system and search for final configurations that contain it. Interestingly, even with this small Production example we could find some deadlock states. One of these final models has tray t1 full of items of the same type, not allowing the assembler machine to proceed. It happens when a generator works much faster than the other one. When it happens, both generators keep generating pieces until the have generated 10, and all the pieces produced after the overload will overflow.Other possibilities would include searching for any state (given by a model) in the execution tree, let it be final or not.

Currently the formal analysis of the specifications needs to be done in Maude, although we are working on the integration of parts of the Maude toolkit within the e-Motions environment. This would allow system modelers to be able to conduct different kinds of analysis to the system without leaving the e-Motions tool, and being unaware of the formal representation of their specifications in Maude.

Self-adaptive systems. Apart from computing the values of the properties that we want to simulate and analyze in the system, observers can be very useful for defining alternative behaviors of the system, depending on specific thresholds levels. For instance, the system can self-adapt under certain conditions, since we are able to search, as discussed above, for states of the system in which some attributes of the observers take certain values, or go above or below some limits.

terms.

As an example, consider the speed of assemblers, which depends on its speed attribute (Fig. 7). It would be easy to define a rule that toggles between two speeds when needed by changing the value of the attribute. This way Fig. 13 shows a rule, Self-adapt, which changes the assembler's speed when the mean cycle time of the pieces of the system is lower than a threshold value, 10 in this case. After this rule is executed, the Assembler only spends one time unit in assembling hammers. The rule that reverts the speed to normal is analogous to that rule. It is not shown here for space reasons.

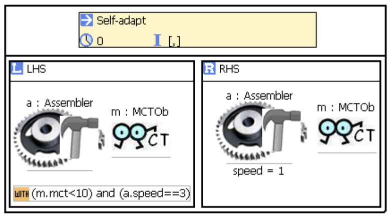

Figure 13. Self-Adapt Rule

5. RELATED WORK

As mentioned in the introduction, the usual approach to specifying the properties of the system that we want to analyze through simulation consists of enriching the system models with annotations. Although this might (partially) work for UML models, the situation is different when the models of the system are specified using ad-hoc domain specific visual languages. In these cases the annotations are normally done using languages which are completely alien to the system designers, because such languages are normally influenced by the analysis tools that need to be used, and written in these tools' languages. In addition, when several properties want to be analyzed, the annotated models become cluttered with a plethora of different annotations and marks (see, e.g., many of the diagrams shown in the MARTE specification).

Alternatively, other analysis tools (such as ARENA [1]) allow users to specify the models to simulate using visual notations, but just within the tools' environment. In other words, these tools cannot take as input models those produced by different editors, nor to export their models so that they can be analyzed by other tools. In our approach we separate the visual specification of the system from the tool(s) that will be finally used to simulate or analyze them.

Observers are not a new concept. They have been defined in different proposals for monitoring the execution of systems and to reason about some of their properties. In fact, the OMG defines different kinds of observers in the MARTE specification [10]. Among them, TimedObservers are conceptual entities that define requirements and predictions for measures defined on an interval between a pair of user-defined observed events. They must be extended to define the measure that they collect (e.g., latency or jitter), and aim at providing a powerful mechanism to annotate and compare timing constraints over UML models against timing predictions provided by analysis tools. In this sense they are similar to our observers. The advantage of incorporating them into DSVLs using our approach is that we can also reason about their behavior, and not only use them to describe requirements and constraints on models. In addition, we can use our observers to dynamically change the system behavior, in contrast with the more "static" nature of MARTE observers.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed the use of special objects (observers) that can be added to the graphical specification of a system for describing and monitoring some of its non-functional properties, and shown its application to simulation. and when The example shown in this paper is rather simple, but it shows the essence of what we can achieve with the integration of observers.

Observers extend the global state of the system with the variables that the designer wants to analyze when running the simulations. Another advantage of our proposal is that it can serve to monitor not only the states of the objects of the system, but also their actions. The fact that action executions are first-class citizens of the e-Motions visual language enables their monitorization by our observers.

In a previous work [14] we proposed the use of observers for specifying and monitoring QoS properties. In this work we have shown how it is possible to use them to monitor the system parameters that we are interested in when simulating the specifications.

As part of our future work we want to extend e-Motions with direct connection to Maude's reachability analysis tools, so that the search facilities previously described can be made available to users from the e-Motions environment. Another line of future work is to study in more detail the expressiveness of this approach, investigating which kinds of properties can be analyzed and simulated using observers, and which ones cannot.

Acknowledgements. This work has been supported by Spanish Research Projects TIN2008-03107 and P07-TIC-03184. The authors would like to thank the referees for their helpful comments and suggestions.

REFERENCES

[1] Arena simulation software. Rockwell automation, 2010. `http://www.arenasimulation.com/`.

[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 MDA*, 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. Efroni, D. Harel, and I. R. Cohen. Reactive animation: Realistic modeling of complex dynamic systems. *Computer*, 38(1):38–47, 2005.

[6] C. Ermel and H. Ehrig. Behavior-preserving simulation-to-animation model and rule transformations. *ENTCS*, 213(1):55–74, 2008.

[7] C. Ermel, K. Holscher, S. Kuske, and P. Ziemann. Animated simulation of integrated uml behavioral models based on graph transformation. In *Proc. of VL/HCC '05*, pages 125–133, Washington, DC, USA, 2005. IEEE Computer Society.

[8] OMG. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*. OMG, Needham (MA), USA, Sept. 2004. ptc/04-09-01.

[9] OMG. *UML Profile for Schedulability, Performance, and Time Specification*. OMG, Needham (MA), USA, Jan. 2005.

[10] OMG. *A UML Profile for MARTE: Modeling and Analyzing Real-Time and Embedded Systems*. OMG, Needham (MA), USA, June 2008.

[11] J. E. Rivera, F. Durán, and A. Vallecillo. A graphical approach for modeling time-dependent behavior of DSLs. In *Proc. of VL/HCC'09*, Corvallis, Oregon (US), Sept. 2009.

[12] 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 SLE'08*, number 5452 in LNCS, pages 54–73, Tolouse, France, 2008. Springer.

[13] J. E. Rivera, A. Vallecillo, and F. Durán. Formal specification and analysis of domain specific languages using Maude. *Simulation: Transactions of the Society for Modeling and Simulation International*, 85(11/12):778–792, 2009.

[14] J. Troya, J. E. Rivera, and A. Vallecillo. On the specification of non-functional properties of systems by observation. In *Proc. of the 2nd International Workshop of Non-functional System Properties in Domain Specific Modeling Languages (NFPinDSML'09)*, Denver, Colorado (US), Oct. 2009.