Model-Driven Performance Analysis of Rule-Based Domain Specific Visual Models

Javier Troya[a], Antonio Vallecillo[a], Francisco Durán[a], Steffen Zschaler[b]

[a] *Dept. Lenguajes y Ciencias de la Computación, Universidad de Málaga, Spain*
[b] *Dept. of Informatics, King's College London, UK*

Abstract

Context. Domain-specific Visual Languages (DSVLs) play a crucial role in Model-Driven Engineering (MDE). Most DSVLs already allow the specification of the structure and behavior of systems. However, there is also an increasing need to model, simulate and reason about their non-functional properties. In particular, QoS usage and management constraints (performance, reliability, etc.) are essential characteristics of any non-trivial system.
Objective. Very few DSVLs currently offer support for modeling these kinds of properties. And those which do, tend to require skilled knowledge of specialized notations, which clashes with the intuitive nature of DSVLs. In this paper we present an alternative approach to specify QoS properties in a high-level and platform-independent manner.
Method. We propose 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.
Results. Observers allow extending the global state of the system with the variables that the designer wants to analyze, being able to capture the performance properties of interest. A performance evaluation tool has also been developed as a proof of concept for the proposal.
Conclusion. The results show how non-functional properties can be specified in DSVLs using observers, and how the performance of systems specified in this way can be evaluated in a flexible and effective way.

Keywords: Model-driven Engineering, Domain Specific Visual Languages,

Email addresses: `javiertc@lcc.uma.es` (Javier Troya), `av@lcc.uma.es` (Antonio Vallecillo), `duran@lcc.uma.es` (Francisco Durán), `steffen.zschaler@kcl.ac.uk` (Steffen Zschaler)

Performance Analysis, Simulation

1. Introduction

Domain specific visual languages (DSVLs) play a crucial role in Model-Driven Engineering (MDE) for representing models and metamodels. The benefits of using DSVLs is that they provide intuitive notations, closer to the language of the domain expert and at the right level of abstraction. In other words, they provide languages that help to model systems in a way which is closer to the problem domain. The MDE community's efforts have been progressively evolving from the specification of the structural aspects of systems to the development of languages that allow for modeling their behavioral dynamics. Thus, several proposals already exist for modeling the structure and behavior of systems. 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 [1, 2, 3, 4, 5].

The correct and complete specification of a system includes, however, other aspects beyond structure and basic behavior. In particular, the specification and analysis of its non-functional properties, such as QoS usage and management constraints (performance, reliability, etc.), is critical in many important distributed application domains including embedded systems, multimedia applications and e-commerce services and applications.

In order to fill this gap, in the last few years researchers have faced the challenge of defining quantitative models for non-functional specification and validation from software artifacts [6, 7]. Several methodologies have been introduced, all sharing the idea of annotating software models with data related to non functional aspects, and then translating the annotated model into a model ready to be validated [8, 9, 10]. Most of these proposals for annotating models with QoS information exist for UML-based notations, with UML Profiles such as UML-QoSFT, UML-SPT or MARTE [11, 12, 13]. Several tools already exist for analyzing and simulating such models and for carrying out performance analyses.

Although these profiles provide solutions for UML models, the situation is different when domain specific visual languages are used to specify a system. In the first place, the QoS annotations are normally written using languages which are completely alien to system designers, because such languages are typically influenced by the analysis methods that need to be used, and written in the languages of these methods. Besides, their level of abstraction is normally lower than the DSVL notations used to specify the system, and tend to be closer to the solution domain than to the problem domain (in contrast with the problem-domain orientation of DSVLs). Furthermore, when several properties need 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 [13]). Another problem is that current proposals for the specification of these kinds of properties tend to require skilled knowledge of specialized languages and notations, which clashes with the intuitive nature of end-user DSVLs and hinders its smooth combination with them. Finally, most of these proposals specify QoS characteristics and constraints using a *prescriptive* approach, i.e., they annotate the models with a set of requirements on the behavior of the system (response time, throughput, etc.) and with constraints on some model elements, but are not very expressive for describing how such values are dynamically computed or evolve over time.

In this paper we present an alternative approach to specify the properties that need to be analyzed, integrating new objects in the specifications that allow them to be captured. 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 targeted by performance engineering, including throughput, mean and max cycle-time for produced items, busy and idle cycles for the machines in the system, mean-time between failures, etc. One of the benefits of observers is that they can be specified in the same language that the domain expert is using for describing the system. Finally, we show how this approach also 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.

This work extends our previous work with *e-Motions*, a notation and a tool for the specification of real-time systems [14, 15, 16], which has served as a proof-of-concept for the proposal. It also extends our initial approach presented in [17, 18] with a complete treatment of the properties being studied and the extensions we have implemented in our tool (Section 5), a methodology for the specification of non-functional properties in rule-based DSVLs (Section 6), and a modular approach for the specification of observers (Section 4).

Following this introduction, Section 2 briefly describes a proposal for modeling the functional aspects of systems and presents the running example that will be used throughout the paper to illustrate our approach. Section 3 introduces the main concepts of our proposal and how they can be used to specify the system parameters that we want to analyze. Section 4 presents some ideas to enable the modular specification of non-functional properties of systems by decoupling the definition of the observers from the system specifications, and the use of aspect-oriented techniques to realize the bindings. Then, Section 5 shows how the specifications produced can be used to analyze the performance of the system and the tool support available for that. Once the main ideas of our proposal have been illustrated using a running example, Section 6 describes the general methodology that we propose for the performance analysis of systems specified with rule-based domain-specific languages such as *e-Motions*. It also explains the current tool support and how the simulations are conducted to obtain the analysis results. Finally, Section 7 compares our work with other related proposals, and Section 8 draws some conclusions and outlines some future research lines.

2. Modeling Real-Time DSVLs

One way of specifying the dynamic behavior of the models expressed with 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 updates [19]. The behavior of the system is then specified in terms of the permitted actions, which are in turn modeled by the model transformation rules.

There are several approaches that propose in-place model transformations for specifying the behavior of a DSVL, from textual to graphical (see [20] for a brief survey). This approach provides a very intuitive way to specify behavioral semantics, close to the language of the domain expert and at the right level of abstraction [1, 2].

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 $l : [\text{NAC}]^* \times \text{LHS} \to \text{RHS}$, where l is the rule's label (its name), and LHS (left-hand side), RHS (right-hand side) and NAC (negative application condition) are model patterns that represent certain (sub-)states of the system. The LHS and NAC patterns express the preconditions for the rule to be triggered, whereas the RHS 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. For simulation, 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 usually be modified by an execution control mechanism, e.g., strategies [21].

In [14], *e-Motions* was presented, a DSVL and a graphical tool developed for Eclipse that extends in-place model transformations with a model of time and mechanisms to state action properties, designed for the specification of real-time systems' behavior. It was showed how time-related attributes can be added to in-place rules to represent features like duration, periodicity, etc. In *e-Motions* there are two types of rules to specify time-dependent behavior, namely, *atomic* and *ongoing* rules. Atomic rules represent atomic actions with a specific duration, which is specified by an interval of time with any OCL expression. Examples of these kinds of actions include the generation of a part, or its collection from a tray. These rules can be periodic, i.e., they admit a parameter that specifies the amount of time after which the action is periodically triggered (if the rule's precondition holds, of course). *Ongoing* rules represent actions that progress continuously over time while the rule's preconditions (LHS and not NACs) hold. This kind of rule is used, for instance, to represent how the state of some observers changes over time (see, e.g., Figs. 11 and 12). Both atomic and ongoing rules can be scheduled, or be given an execution interval.

There are two main approaches to the specification of concurrent systems and their properties: state-based and action-based [22, 23]. In the former case the system is characterized by states and state changes, while in the latter case it is characterized by the events (actions) that can be performed to move from one state to another. Although in theory the expressiveness of both approaches is similar, in practice the use of one or the other can force the use of unnatural or artificial expressions of some properties (for example, reasoning about some particular actions in a pure state-based approach

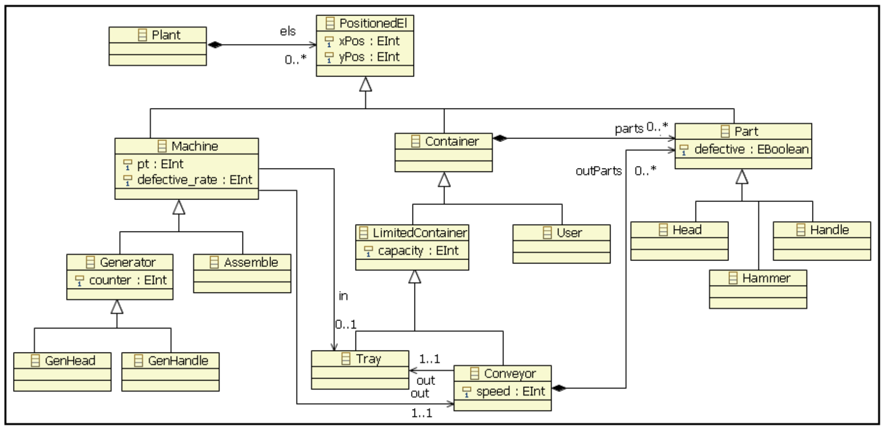


Figure 1: Production Line Metamodel.

normally forces the artificial extension of the system elements specifications with additional attributes to capture how their states evolve as the action progresses).

In order to be able to model both state-based and action-based properties, *e-Motions* also implements a reflective mechanism that allows extending model patterns with *action executions* to specify action occurrences. These action executions specify the type of the action (i.e., the name of the atomic rule), its status (e.g., if the action is unfinished or realized) and its identifier. They may also specify its starting, ending and execution time and the set of participants involved in it. 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.

Finally, a special kind of object, named Clock, represents the current global time elapse. Designers can use it in their rules (using its attribute time) to know 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 several 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. As any other DSVL, our production line system is defined in terms of three main elements: abstract syntax, concrete syntax and semantics. The abstract syntax defines the domain concepts that the language is able to represent, and is defined by a metamodel. The concrete syntax

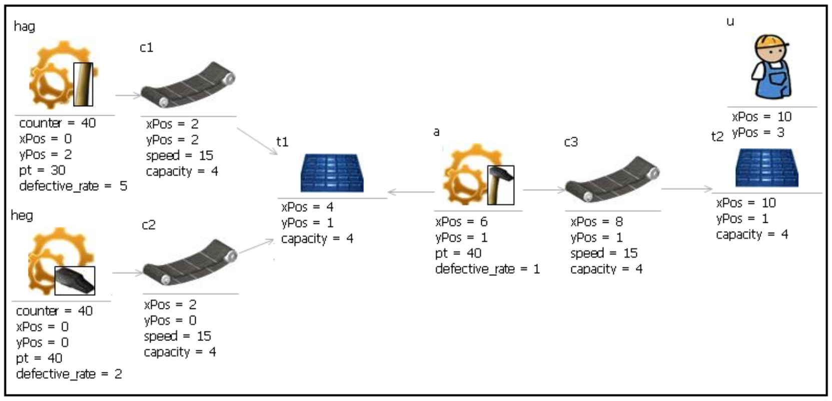


Figure 2: A model of the system, which will be used as initial configuration.

defines the notation of the language, and in our example it is defined by assigning an icon to each concept in the metamodel. The semantics describe the meanings of the models represented in the language, and in case of models of dynamic systems (such as ours) the semantics of a model describe the effects of executing that model. In our case, semantics are specified by a set of behavioral rules.

The metamodel for the system is depicted in Fig. 1. There are different kinds of machines (head generators, handle generators and assemblers), containers (users and containers with a limited capacity, such as trays and conveyors) and parts (head, handles and hammers). They all have a position in the plant, indicated by a set of coordinates. Generators will produce as many parts as their counter indicates and deposit them on conveyors; conveyors move parts from machines to trays at a given speed; and assemblers consume parts from trays to create hammers, which are deposited on conveyors and finally collected by operators. Parts can be either defective or not. Machines work according to a production time (pt) that dictates the average number of time units that they take to produce a part. They also have an attribute (defective_rate) that determines the percentage of defective parts they produce, which depends on their production time (normally, the faster they work the more defective parts they produce). Trays and conveyors can contain parts up to their capacity.

Fig. 2 shows an example of a model of the system, depicted with *e-Motions*, with a set of objects and values for their attributes, which will be used as initial configuration. In that Figure, we can see the concrete syntax

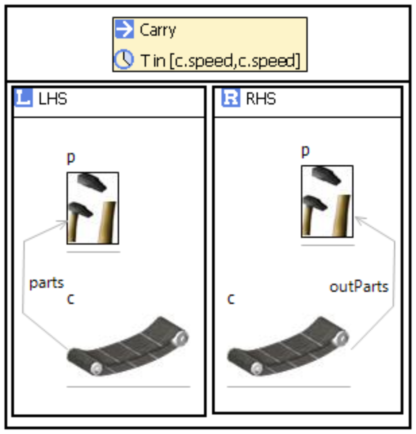


Figure 3: Carry Rule.

given to the concepts of the production line metamodel.

The behavior of the system is then expressed in terms of a set of rules, each one representing a possible action. In the case of our Production Line System, its behavior can be described by 6 rules: one for generating handles (GenHandle); one for generating heads (GenHead); one for describing how conveyors move parts (Carry); one for assembling heads and handles into hammers (Assemble); one for depositing parts in trays once they have been transported by conveyors (Transfer), and one final rule to describe how the operator collects assembled hammers (Collect).

For example, Fig. 3 shows the atomic Carry rule, which specifies how a Part is transported through a Conveyor, moving the part from the beginning of the conveyor to its end. In the LHS pattern of the rule, the Part is related to the Conveyor with the parts reference, indicating that the part is placed at the beginning of it. In the RHS pattern, the relation between both objects is outParts, which indicates that the part is placed at the end of the conveyor and is ready to be transferred to the Tray connected to it. The time this rule spends, which simulates the time needed to move a part through a conveyor, is the corresponding speed of the conveyor (15 time units in our example).

Further examples that illustrate our proposal can be found in [24].

3. Monitoring the System with Observers

Once we can count on languages for specifying models and their behavior, the next step is to analyze its non-functional properties. For that we need to be able to express the properties that we want to analyze (e.g., delays, mean-time between failure, or end-to-end throughput), and then we need to have a simulation engine that executes the specifications.

3.1. Specifying the Properties to be Analyzed

Although the number of non-functional properties that can be defined for a system can be large, in this paper we will concentrate on some representative QoS properties [25]. Let us suppose then that we want to analyze the following performance parameters of the Production Line System:

- **Throughput**. Number of non-defective hammers collected by the operator per unit of time.
- **Mean time between failures (MTBF)**. It is the mean time between producing defective hammers.
- **Idle-time**. Amount of time that a machine has not been working (idle, waiting for parts or waiting for the output conveyor to be free).
- **Mean Idle-time**. Average idle time of all machines.
- **Cycle time**. Time required to produce a hammer: from the production of its parts until its final collection.
- **Mean cycle time (MCT)**. Mean cycle time of all produced hammers.
- **Delay**. This property indicates the difference between the *theoretical* optimal cycle time of a hammer and its actual cycle time.

Let us clarify here that all collected hammers will be taken into account for computing the system mean cycle time and delay, while only non-defective hammers will be considered for computing the system throughput.

3.2. Defining the Observers' Structure

To specify and calculate the value of system properties, the use of *observers* was proposed in [18, 17]. An observer is an object whose purpose is to monitor the execution of the system: the state of the objects, of the actions, or both. 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

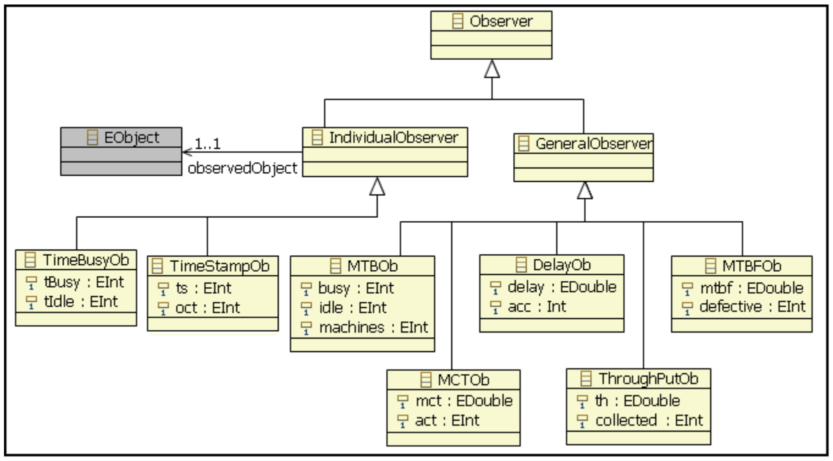


Figure 4: Observers Metamodel.

there throughout its entire life. In the latter case, observers are created and destroyed with the objects they monitor.

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. It defines two kinds of observers, IndividualObservers and GeneralObservers, both inheriting from a general Observer class. Observers of individual objects have a reference to an EObject class, which is the interface implemented by every model object in the *Eclipse Modeling Framework* (EMF [26]). In this way, these observers can be associated to any element of any model. We count on two specific observers for individual objects and five for the whole system. Let us explain the objectives of each of them for our example:

- **TimeBusyOb**. It monitors the time a machine is working and the time it is idle.

- **TimeStampOb**. It monitors individual parts. Its ts attribute stores the time at which the generator started producing the part. Attribute oct keeps the *optimal cycle time* of the part. It is updated as the part moves forward in the system. It is used to compute the delay.

- **MTBOb**. This observer monitors the average working time of all the machines in the plant. It uses the number of machines (attribute machines) as well as the attributes of the TimeBusyOb observers of all machines.

- **MCTOb**. This observer keeps the mean cycle time of every collected hammer in its mct attribute.

- **DelayOb**. The sum of the delays of all collected hammers is stored in its acc attribute, while the hammers' average delay is stored in its delay attribute.

- **ThroughPutOb**. It stores the throughput of the system in its th attribute. The number of collected hammers is stored in its collected attribute.

- **MTBFOb**. It records the mean time between failures in the system and the number of defective hammers.

Note how the use of observer objects provides a modular approach to extend the state of the system with the attributes that capture the properties we want to analyze. However, to conduct the performance analyses we also need to specify the behavior of the observers while the system executes.

3.3. Defining the Observers' Behavior

The idea of analyzing the system with observers is to combine the original metamodel (Fig. 1) with the Observers' metamodel (Fig. 4) to be able to use the observers in our DSVL for specifying production line systems. *e-Motions* allows users to merge several metamodels in the definition of a DSVL behavior.

The behavior of the observers is also specified using rules. In fact, their behavior is described by enriching the system rules with information about the observers—so that now the rules not only describe how the system behaves but also how the observers monitor the system and update their state. In what follows we show how the behavior of the observers is specified by adding them to the system rules.

Firstly, we modify the model with the initial configuration of objects by adding observers to it (Fig. 5). We have defined five observers for the whole system (DelayOb, ThroughPutOb, MTBFOb, MTBOb and MTCOb), and three individual TimeBusyOb observers, each one associated to a machine to record the time it is working.

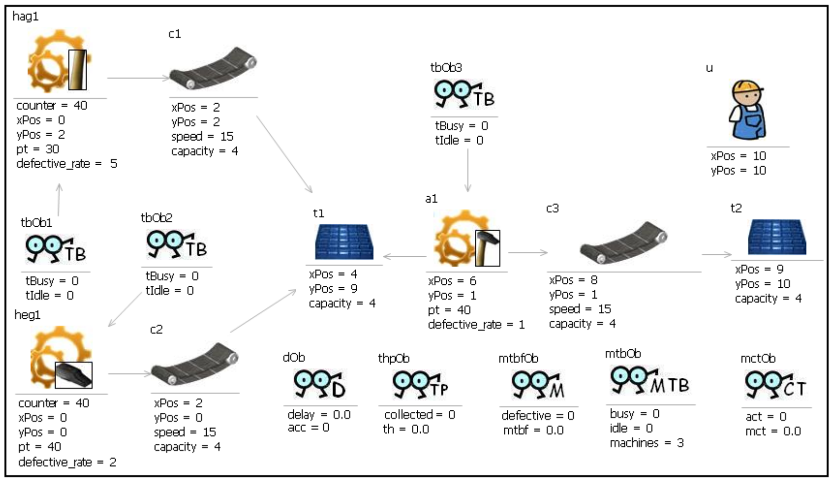


Figure 5: The initial configuration of the system with Observers.

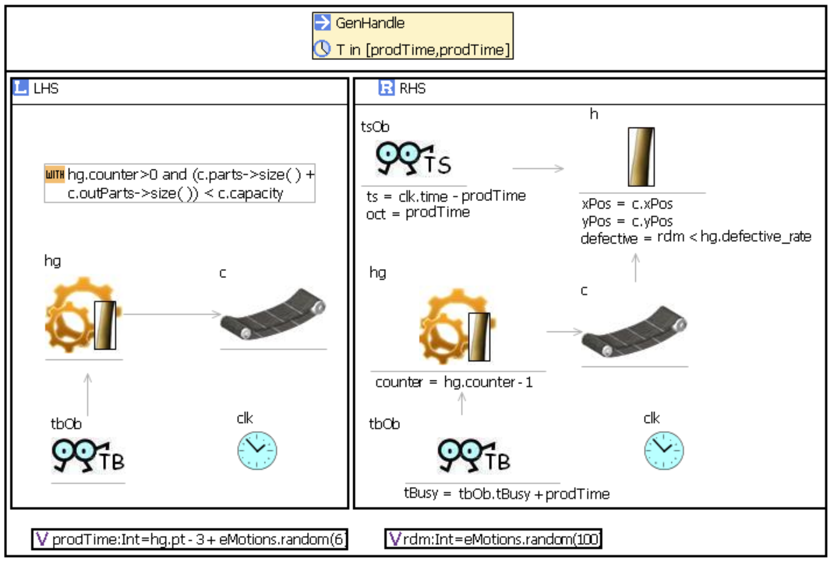

GenHandle
T in [prodTime,prodTime]
LHS
RHS
WITH hg.counter>0 and (c.parts->size() + c.outParts->size()) < c.capacity
hg
c
tbOb
clk
TB
tsOb
TS
ts = clk.time - prodTime
oct = prodTime
h
xPos = c.xPos
yPos = c.yPos
defective = rdm < hg.defective_rate
hg
c
counter = hg.counter - 1
tbOb
TB
tBusy = tbOb.tBusy + prodTime
clk
V prodTime:Int=hg.pt - 3 + eMotions.random(6
V rdm:Int=eMotions.random(100

Figure 6: GenHandle Rule.

Then, we need to specify the behavioral rules. The first two define how and when heads and handles are generated. Fig. 6 shows the GenHandle atomic rule that generates a new handle every time it is launched. The time it spends in the generation of a handle depends on its production time attribute (pt). Instead of a fixed time, we use a random value from the

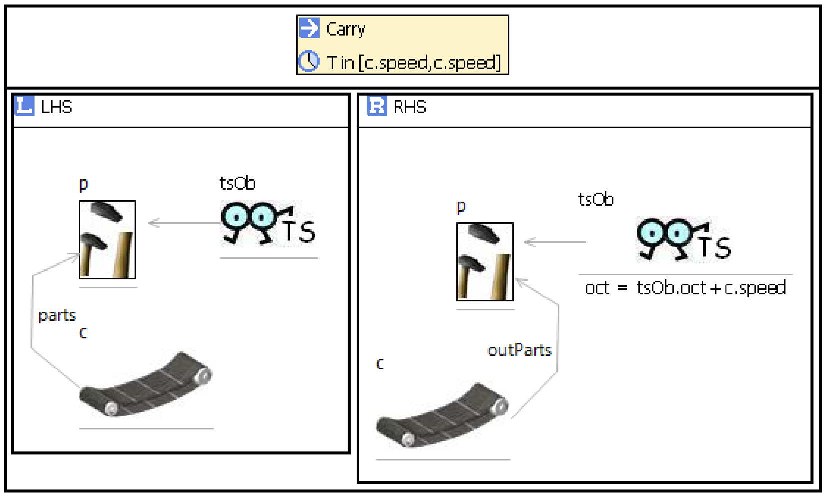
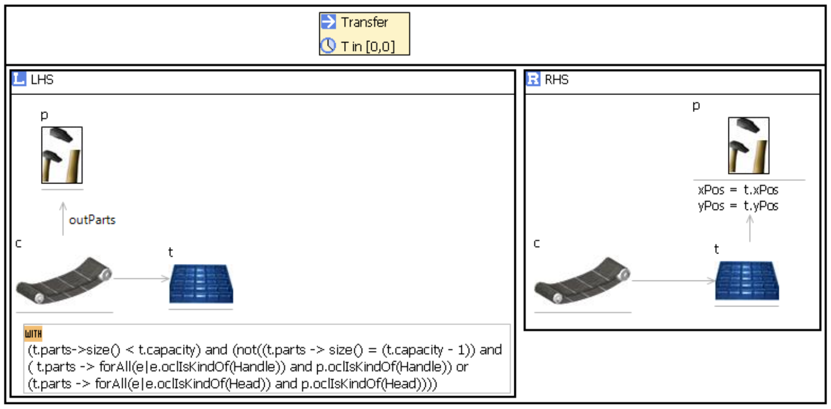


Figure 7: Carry Rule with Observer.

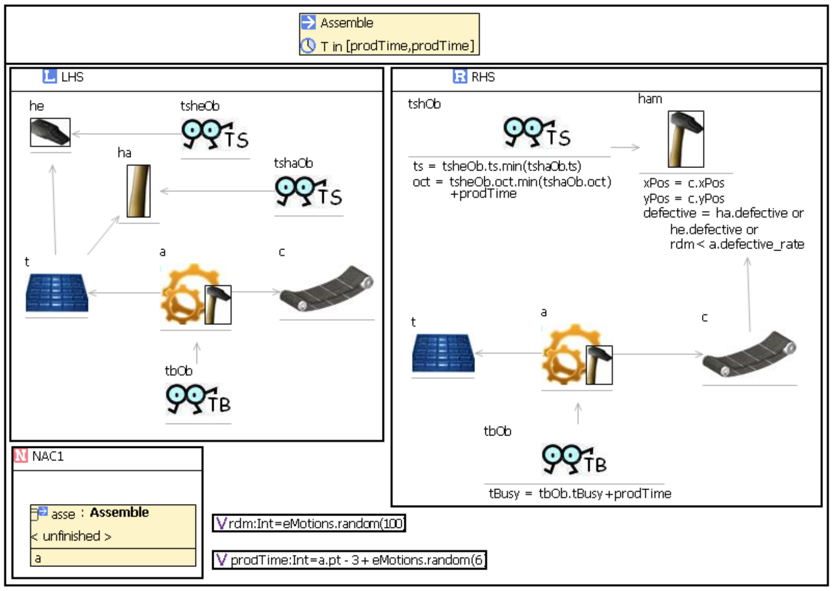
interval $[hg.pt - 3, hg.pt + 3]$ to specify its duration. It uses the $random(n)$ function available in *e-Motions* that returns an integer value between 0 and n. For this rule to be applied, the LHS pattern indicates that the system has to have a HandleGen generator which has to be connected to a Conveyor. The LHS pattern also has a condition (see the WITH clause), so the rule can only be applied if the attribute counter of the GenHandle object is greater than 0 and the Conveyor has room for the generated part. In the RHS pattern a new Handle is generated and it acquires the position of the Conveyor connected to the HandleGen machine. To decide whether the new Handle generated is a defective part or not, we check if a random variable between 0 and 100 (rdm) is smaller than the defective rate of the generator. The counter value of the GenHandle is decreased in 1 unit, which represents the fact that a new handle has been produced. A TimeStampOb observer is created and it is associated with the Handle, storing in its ts attribute the time at which the HandleGen started to generate the Handle—we use the Clock instance to get the time the system has been working. Note the use of OCL to compute the values of the objects' attributes. Analogously, the GenHead rule (not shown here) models the generation of heads.

As soon as a new part is placed on a Conveyor, it is transported from its beginning to its end. We modify the Carry rule shown in Fig. 3, which models this behavior, to introduce the corresponding observers in our system. The resulting rule is shown in Fig. 7. The observer associated to the Part simply



```
(t.parts->size() < t.capacity) and (not((t.parts -> size() = (t.capacity - 1)) and
( t.parts -> forAll(e|e.oclIsKindOf(Handle)) and p.oclIsKindOf(Handle)) or
(t.parts -> forAll(e|e.oclIsKindOf(Head)) and p.oclIsKindOf(Head))))
```

Figure 8: Transfer Rule.



```
ts = tsheOb.ts.min(tshaOb.ts)
oct = tsheOb.oct.min(tshaOb.oct)
      +prodTime

xPos = c.xPos
yPos = c.yPos
defective = ha.defective or
      he.defective or
      rdm < a.defective_rate

tBusy = tbOb.tBusy+prodTime

rdm:Int=eMotions.random(100)
prodTime:Int=a.pt - 3 + eMotions.random(6)
```

Figure 9: Assemble Rule.

updates its *optimal cycle time* by adding the time the Conveyor takes when transporting the Part.

Once there is a part on a Conveyor ready to be transferred to the Tray connected to the end of it, the instantaneous rule Transfer (Fig. 8) deals with it. The LHS pattern specifies that this Part must be placed in the outParts part of the Conveyor. The rule condition forbids the triggering of the rule

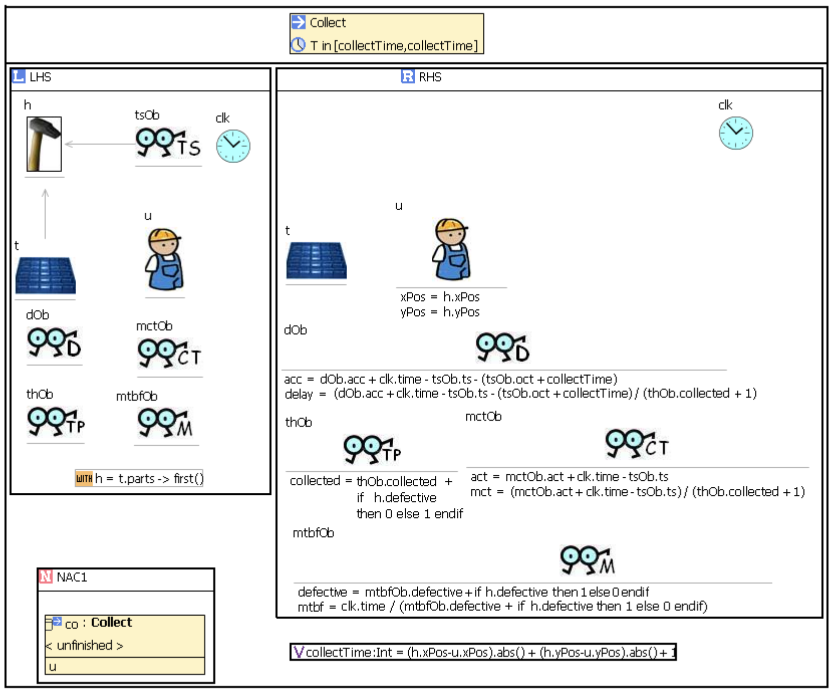

Collect
T in [collectTime,collectTime]
LHS
h
tsOb
clk
t
u
dOb
mctOb
thOb
mtbfOb
WITH h = t.parts -> first()
RHS
clk
u
t
xPos = h.xPos
yPos = h.yPos
dOb
acc = dOb.acc + clk.time - tsOb.ts - (tsOb.oct + collectTime)
delay = (dOb.acc + clk.time - tsOb.ts - (tsOb.oct + collectTime) / (thOb.collected + 1)
thOb
mctOb
collected = thOb.collected + if h.defective then 0 else 1 endif
act = mctOb.act + clk.time - tsOb.ts
mct = (mctOb.act + clk.time - tsOb.ts) / (thOb.collected + 1)
mtbfOb
defective = mtbfOb.defective + if h.defective then 1 else 0 endif
mtbf = clk.time / (mtbfOb.defective + if h.defective then 1 else 0 endif)
NAC1
co : Collect
< unfinished >
u
collectTime:Int = (h.xPos-u.xPos).abs() + (h.yPos-u.yPos).abs() + 1

Figure 10: Collect Rule.

when the Tray is full of parts, or when there is only one part needed to reach the capacity of the Tray and the parts on it are of the same type as part p. This last condition, together with the restriction of generators to produce a part when their out conveyor is full, prevents the system from deadlocking: if all the parts in the Assembler's input Tray were of the same type, the Assembler could not assemble anymore and the production line would stop.

The Assemble rule is shown in Fig. 9. This rule models the behavior of generating a Hammer using 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. The NAC indicates that this rule cannot be triggered if the Assembler is already participating in an action of type Assemble, i.e., if it is already assembling a Hammer—maybe with different parts. In the RHS pattern we see how the Head, the Handle and their associated observers have been removed and a new Hammer has been created in the position of the Conveyor connected to the assembler. The Hammer has an associated TimeStampOb observer whose ts attribute is the lowest value between the timestamps of the Head and the Handle. Its oct (optimal cycle time) attribute is the lowest

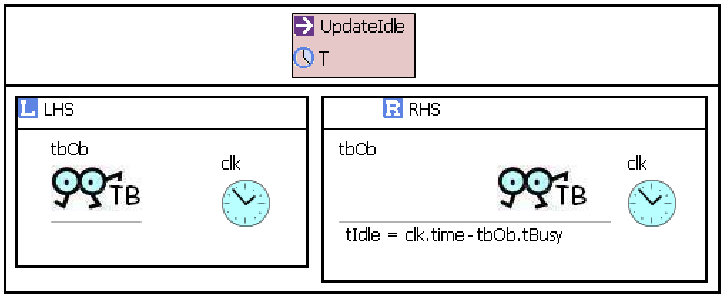


Figure 11: UpdateIdle Rule.

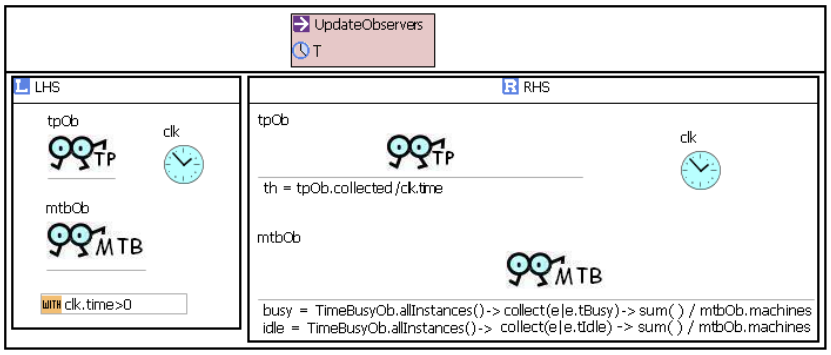

UpdateObservers
T
LHS
tpOb
TP
clk
mtbOb
MTB
clk.time>0
RHS
tpOb
TP
th = tpOb.collected /clk.time
clk
mtbOb
MTB
busy = TimeBusyOb.allInstances()-> collect(e|e.tBusy)-> sum( ) / mtbOb.machines
idle = TimeBusyOb.allInstances()-> collect(e|e.tIdle) -> sum( ) / mtbOb.machines

Figure 12: UpdateObservers Rule.

oct value of the Head and the Handle plus the time the Assembler has spent in assembling the Hammer, which is specified by the variable prodTime.

Fig. 10 shows the Collect rule. This rule models the behavior of the system when the User finally collects an assembled Hammer. The User acquires the position of the Tray where the Hammer is. The Hammer and its associated observer disappear from the system, modeling that the User has collected the Hammer, since they are no longer needed. The time this rule takes is defined by the Manhattan distance between the Hammer and the User plus one, which is stored in the variable collectTime. It models the time the User spends in getting to the Tray where the Hammer is. There is also a NAC in this rule, which forbids users to collect more than one hammer at a time. We also see the presence of four general observers, whose states are updated when the rule is executed.

We showed in rules in Figs. 6 and 9 how TimeBusyOb observers update their tBusy attributes. Their tIdle attributes are kept up to date with an ongoing rule. This rule is shown in Fig. 11. It simply calculates the time the machine associated with this observer has been idle by subtracting the time the machine has been working from the current time elapse of the system.

Finally, another ongoing rule is used to keep the state of the ThroughPutOb and MTBOb observers updated (Fig. 12).

4. A Modular Approach for the Specification of Observers

Observers allow a flexible way to monitor the system and to obtain information about its behavior. However, the addition of observers may require to change the existing behavioral models to a large extent, making system models potentially complex and difficult to maintain. One way to cope with this problem is by using aspect-oriented techniques [27], whereby a modular specification of the behavior of observers is provided, and then *woven* (in the aspect-oriented sense) into the system behavioral rules. Thus, we will have a model with a set of observers rules and another with a set of system rules and they will be merged (*woven*) together using weaving mechanisms as explained in Sections 4.3 and 4.4.

The goal is to define a library with different observers' behavior, which can then be reused by concrete systems to incorporate observers within their behavior specifications. This provides a modular approach to the specification and monitoring of individual properties, for which observers can be independently defined and reused across system specifications.

Our approach is based on standard software measurement approaches, which define *base* and *derived* measures [28]. The former ones allow measuring individual object attributes, while the latter build on the values of base measures to define aggregated metrics. Similarly, we propose *individual* and *general* observers. As mentioned in Section 3, individual observers are attached to individual objects to monitor their state and/or behavior. General observers will monitor individual observers, as well as the rest of the objects in the system, to build derived measures for non-functional properties such as throughput or mean time between failures, for instance.

Although of different nature, the behavior of individual and general observers share a similar and regular pattern, that corresponds to their lifecycle: creation, monitoring and termination. Hence their behavior can be specified using a similar approach, in which rules are defined for each of these phases. Such a behavior, once specified, can be woven to the functional behavior of a system to produce the complete system specifications.

In the remainder of this section we present the behavior for both individual and general observers in terms of behavioral rules, and present our approach on how to *weave* these behavioral rules with the functional rules of our system (without observers) so that observers are incorporated to the latter. We show this approach graphically and also show some AMW and ATL essentials in order to implement it. Finally, we present some room for improvement regarding this modular approach.

4.1. The Behavior of Individual Observers

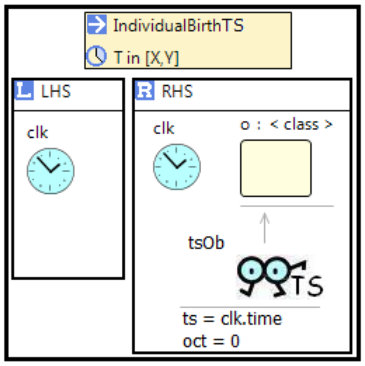
Individual observers are "born" with the objects they monitor, store information as the associated objects perform relevant actions and, eventually, "die" when the associated objects terminate. Thus, the behavior of individual observers can be usually described by three rules, one for each phase.

For illustration purposes, let us specify here the behavior of the TimeStampOb observer. Rule IndividualBirthTS (Figure 13(a)) describes how the individual observer is created with the monitored object (expressed by a generic object represented as an empty box). Usually, the value of the ts attribute is the current time elapse, although the modeler can specify another value when weaving the rule. Attribute oct is set to 0.

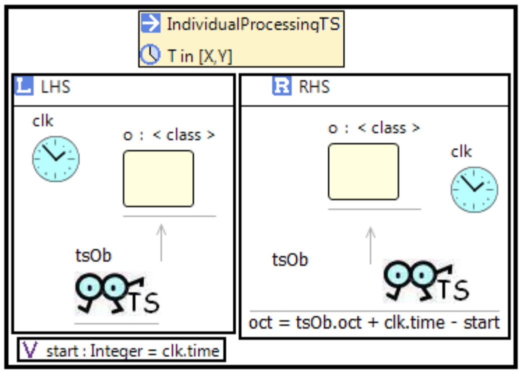
Rule IndividualProcessingTS (Figure 13(b)) specifies how a TimeStampOb observer updates its state variables. In this case, it has to update its *optimal cycle time* whenever it participates in a rule, i.e., when the object associated to it realizes some action in the rule (like processing some object or being processed). Note that the start variable stores the moment in time at which the rule starts being processed, while the direct use of the elapsed time (clk.time) in the RHS of a rule refers to the time at which the rule finishes. Thus, the oct attribute is updated with the addition of the time spent by the rule in its execution.

Finally, rule IndividualEndTS (Figure 13(c)) shows how the observer disappears from the system when the monitored object is dismissed.

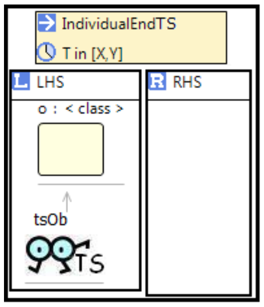
Other individual observers follow very similar patterns. In fact, one possibility is to define the behavior of individual objects at the highest level in the inheritance hierarchy, in case they all share the same behavior. In this way, the IndividualEndTS rule, for example, would be the same for many other individual observers. This is not always possible, however, because some individual observers may have particular pre-conditions for recording events, or perform specific actions when recording them.



(a) IndividualBirthTS



(b) IndividualProcessingTS



(c) IndividualEndTS

Figure 13: Behavioral rules for TimeStampOb observers.

4.2. General Observers

The behavior of general observers is normally determined by four kinds of rules. Let us show them here in a generic way, and then we will illustrate them in the particular case of the ThroughPutOb global observer. The first rule (GeneralBirth, shown in Figure 14) specifies the creation of a global observer. Since they are created at start time, this rule is normally woven to the initial rule of the system.

Two rules specify how global observers update their state variables, depending on whether they do it when an object disappears from the system, or when the object participates in a rule. Thus, generic rule RecordLeave (Fig. 15(a)) shows how a counter attribute is updated when an object leaves the system. Similarly, rule RecordEvent (Fig. 15(b)) models the behavior of a global observer that records that an object has participated in a rule. Note that these are generic rules, and that we will have similar rules (sometimes

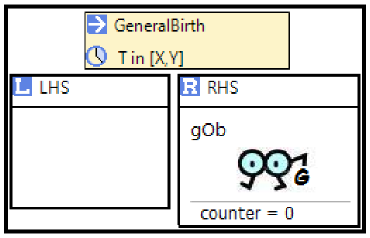
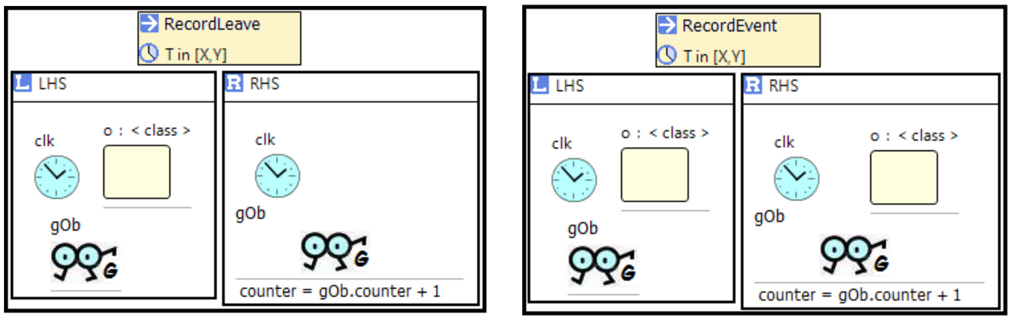


Figure 14: GeneralBirth Generic Rule.



(a) RecordLeave Generic Rule (b) RecordEvent Generic Rule

Figure 15: Generic rules for event recording.

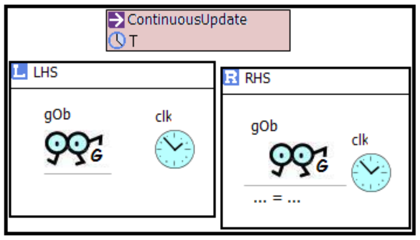
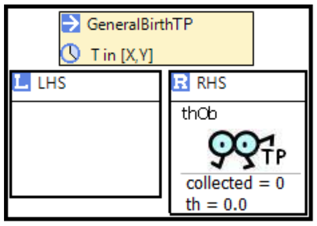


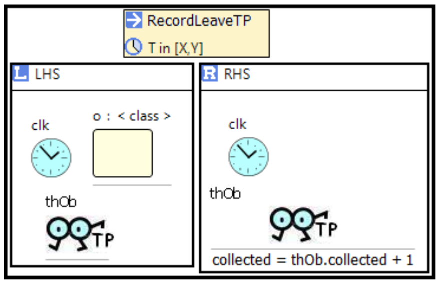
Figure 16: ContinuousUpdate Generic Rule.

not both RecordLeave and RecordEvent, but just one of them) for each general observer in our observers' library. We shall later see the rules for observer ThroughPutOb. Note as well the presence of the clock object. We include it in case the modeler specifies different values for the observer's attributes that need the current time elapsed.

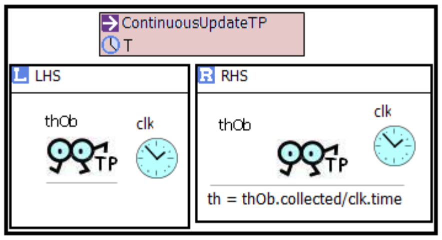
Some attributes of some general observers need to be updated as time moves forward. Such a behavior is modeled with ongoing rules, whose generic form is shown in Fig. 16. When the concrete system's behavior is specified,



(a) GeneralBirthTP Rule



(b) RecordLeaveTP Rule



(c) ContinuousUpdateTP Rule

Figure 17: Behavioral rules for the ThroughPutOb observer.

more than one of these rules can be put together, so that more than one general observer may be updated as time progresses.

Thus, to specify the behavior of the ThroughPutOb global observer, we just need three rules. The first one creates the object, and is similar to the GeneralBirth rule shown in Fig. 14, the only difference being that the initialized attributes are th and collected. The second rule is in charge of updating the collected attribute every time a final element is consumed. Thus, it follows the RecordLeave rule pattern. Finally, we need an ongoing rule so that the throughput value is continuously updated. Figure 17 shows these three rules for the ThroughPutOb observer.

4.3. The Weaving of Rules

Once defined, the independent observers' behavior can then be woven with specific system specifications. By "weaving" them we mean that the observer objects included in the observers rules will appear, as result of the weave, in the system rules, by merging the *generic* objects present in the observer rules with concrete objects in the system rules. For that we use a weaving model that uses the AtlanMod Model Weaver (AMW[1]) [29] to define the correspondences between the generic objects in the observers rules and the concrete objects that appear in the system behavioral rules.

It is also possible to define a correspondence between one observer rule and one system rule. In this case, all the elements from the observer rule will be copied to the system rule.

When defining the weaving links between the observers and the system rules it is possible to add expressions that overwrite how the values of the attributes are calculated in the observer's rule. In this way we allow some kind of rule *parametrization*.

Finally, it is possible (and very common) to establish correspondences between several observers rules and one system rule, when we want to add more than one kind of observer to a rule. In this case, only one final rule is produced with the results of all weaves. When the attributes of the observers to be added in a rule does not depend on each other, i.e., they do not use the values of other observers attributes in their calculations, it does not matter the order in which the weaves are applied. However, when this is not the case, those observers whose attributes are needed in the calculations of other observers should be woven first, as shown for example in the DelayOb observer in [30]. In the same way, it is also possible to establish correspondences between one observer rule and several system rules, when we want to add the same observer in different rules.

To illustrate this approach, let us apply it to our case study. In this section we show our approach graphically, while in Section 4.4 we show how model weaving works in AMW and ATL.

Starting from the behavioral rules described in Section 3.3, taking out the observers, and assuming that we have defined the rules for the observers as explained in Sections 4.1 and 4.2, below we present the weaving for the ThroughPutOb and TimeStampOb observers in order to include them in the system's behavioral rules.

Figure 18 shows the weaving links between the rules specifying the behavior of the ThroughPutOb observer and the system rules where we want the observer to be woven. For illustration purposes, weaving links are graphically depicted in the figure as thick lines, although in practice the textual Eclipse AMW weaving editor is used to specify these links. The duration of

[1]The address of the official website of AMW is http://wiki.eclipse.org/AMW

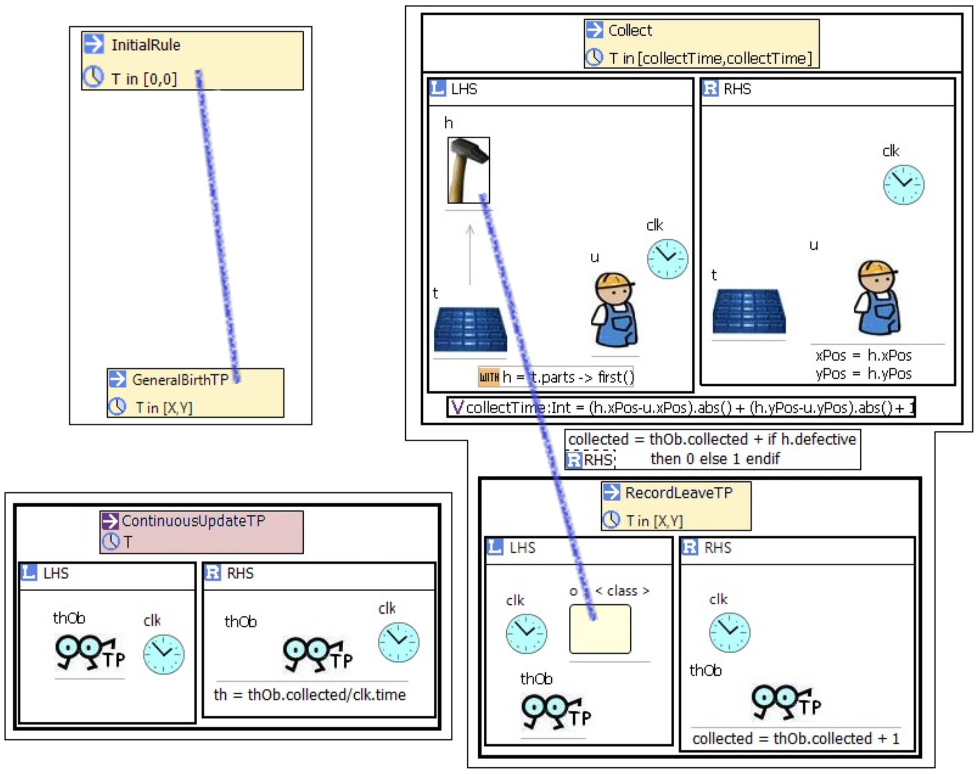

InitialRule
T in [0,0]
GeneralBirthTP
T in [X,Y]
Collect
T in [collectTime,collectTime]
LHS
RHS
h
t
u
clk
WITH h = t.parts -> first()
xPos = h.xPos
yPos = h.yPos
collectTime:Int = (h.xPos-u.xPos).abs() + (h.yPos-u.yPos).abs() + 1
collected = thOb.collected + if h.defective then 0 else 1 endif
RHS
ContinuousUpdateTP
T
thOb
th = thOb.collected/clk.time
RecordLeaveTP
o : < class >
collected = thOb.collected + 1

Figure 18: Weaving the ThroughPutOb observer.

the resulting rules is that of the system rules. Two weaving links are defined in order to include the ThroughPutOb observer in the system. The first one is defined between the InitialRule and GeneralBirthTP rules, where InitialRule is a system rule that has the initial configuration of the system without observers (Figure 2) in its RHS and GeneralBirthTP is an observer rule (Figure 17(a)). We only show the headings of both rules for making the figure simpler. The aim of this binding is to include the observer in the initial configuration of the system (as it appears in the initial model shown in Figure 5). The second weaving link is defined between the Hammer object in the Collect rule of the system and the generic object in the RecordLeaveTP rule (the NAC of the former rule is not shown for readability). The effect of this binding is to include the ThroughPutOb observer in both the LHS and RHS of the Collect rule. Note that the expression used to calculate the value of the collected attribute in the RHS of the observer rule has been overwritten by a new expression

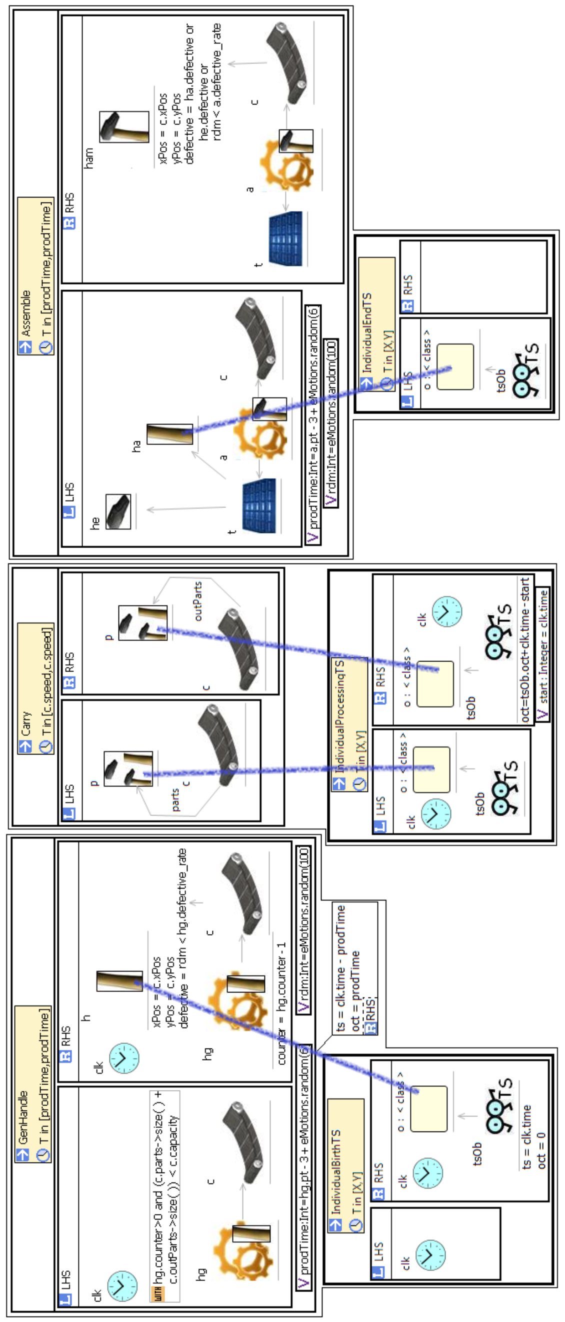

GenHandle
T in [prodTime,prodTime]
LHS
clk
hg.counter>0 and (c.parts->size() + c.outParts->size()) < c.capacity
hg
c
RHS
h
xPos = c.xPos
yPos = c.yPos
defective = rdm < hg.defective_rate
counter = hg.counter - 1
prodTime:Int=hg.pt - 3 + eMotions.random(6)
rdm:Int=eMotions.random(100)
IndividualBirthTS
T in [X,Y]
o : < class >
tsOb
ts = clk.time
oct = 0
ts = clk.time - prodTime
oct = prodTime
Carry
T in [c.speed,c.speed]
p
parts
outParts
IndividualProcessingTS
oct=tsOb.oct+clk.time - start
start : Integer = clk.time
Assemble
T in [prodTime,prodTime]
he
ha
a
t
ham
defective = ha.defective or he.defective or rdm < a.defective_rate
prodTime:Int=a.pt - 3 + eMotions.random(6)
IndividualEndTS

Figure 19: Weaving the TimeStampOb observer associated to handles.

(this is indicated in the figure inside the box between the two woven rules). Regarding the clock object, it has a special treatment. Whenever it appears in one of the two rules being woven, it appears in the resulting rule. If there is a clock in both the system and observer rules, only one will be present in the final rule. Finally, the ContinuousUpdateTP rule is added to the system rules (it becomes a new rule in the system).

Figure 19 shows the weaving links for the TimeStampOb observer associated to handles. The observer rules (in the lower part of the figure–note that the figure is rotated) specify its life-cycle. The generic object in the IndividualBirthTS rule to which the TimeStampOb observer is associated is linked to the Handle in the GenHandle rule. Such a link inserts the observer associated to the handle in the GenHandle rule. The initial values for the attributes of the observer for this system are specified in the weaving link (graphically shown in the box attached to the line that represents the binding). In the middle part of the figure we can see how the generic objects in the IndividualProcessingTS rule are linked to the Part objects in the Carry rule (remember that the Part class is a generalization of the Handle one, see Figure 1). The weaving link will make the TimeStampOb be inserted in the Carry rule and associated to the Part object. Finally, the observer has to disappear from the system when its associated Handle object disappears. For that purpose, we link the generic object in the IndividualEndTS rule with the Handle object in the Assemble rule, since the latter disappears in that rule.

The weavings for including the rest of the observers in the system can be found in [30].

4.4. A Prototype Implementation Based on AMW and ATL

The AMW is a framework for establishing relationships (i.e., links) between models. The links are stored in a model, called weaving model. It is created conforming to the weaving metamodel shown in Figure 20.

WElement is the base element from which all other elements inherit. WModel represents the root element that contains all model elements. WLink denotes the link type and has a reference end to associate it with a set of link endpoints (WLinkEnd). Each WLinkEnd references one WElementRef. The attribute ref contains the identifier of the linked elements. WModelRef is similar to WElementRef, but it contains references to the models.

This metamodel is to be extended for each concrete situation according to our semantics for *Behavior* in *e-Motions* (see [14] for a detailed explanation),

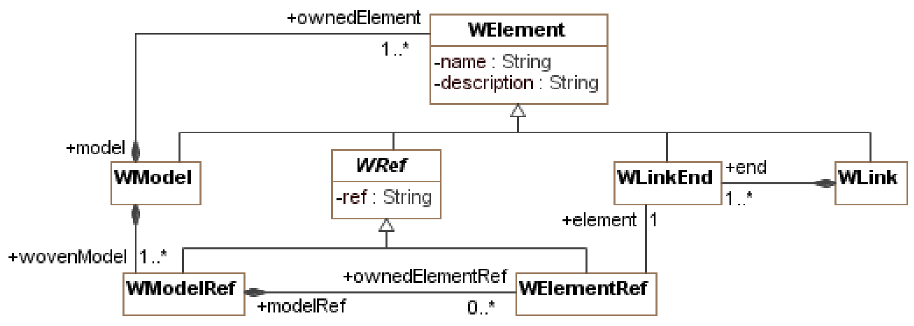


Figure 20: AMW Core Weaving Metamodel (from [29]).

specified as a set of in-place rules as explained in Section 2. A possible extension for the core weaving metamodel for our example is the following:

- The models to be woven are defined as extensions of the WModel class. We want to weave two models, those with the system and observers rules:

```
class MergeModel extends WModel{
  reference systRulesModel container : WModelRef;
  reference obsRulesModel container : WModelRef;
}
```

- The different kinds of links are extensions of WLink. There are two kinds of links in our case: those that define a matching between rules and those defining matchings among objects. In the latter links, it can be specified the new slot value that supersedes the slot value in the observer of the observer's rule:

```
class MergeObjLink extends WLink{
  reference obsObj container : ObsObject;
  reference systOb container : WLinkEnd;
}
class MergeRuleLink extends WLink{
  reference obsRule container : WLinkEnd;
  reference systRule container : WLinkEnd;
}
class ObsObject extends WLinkEnd{
  attribute slotsR [0-*] : String;
  attribute slotsL [0-*] : String;
}
```

- The elements that can be woven are defined as extensions of WElementRef. We can weave objects and rules:

```
    class Object extends WElementRef{}
    class Rule extends WElementRef{}
```

The models conforming to this extension of the core weaving metamodel will have two kinds of links, those specifying matches between rules and those specifying matches among objets. An example of the matching between rules InitialRule and GeneralBirthTP shown in Figure 18 is the following:

```
<ownedElement xsi:type="MergeRuleLink" xmi:id="MergeRuleLink1"
  name="RuleLink1">
    <end xsi:type="obsRule" xmi:id="obsRule1" name="GeneralBirthTP"
      element="ObsRuleTPRefXMI1"/>
    <end xsi:type="systRule" xmi:id="systRule1" name="InitialRule"
      element="SystRuleRefXMI2"/>
</ownedElement>
```

In contains the names of the rules to be merged as well as references to them.

Finally, the weaving operation is interpreted by the ATL [31] engine. An ATL transformation matches each type of link and executes the appropriate weaving operation. Thus, the ATL transformation takes the weaving model, the system rules model and the observers rules model as input and produces a new woven model composed by behavioral rules. An excerpt of the rule dealing with the weaves among rules is the following:

```
rule MergeRules {
  from
    link : AMW!MergeRuleLink
  to
    rul : Beh!Rule (
      name <- link.end ->last().name,
      soft <-link.end ->last().soft,
      lowerBound <- link.end ->last().lowerBound,
      upperBound <- link.end ->last().upperBoud,
      maxDuration <- link.end ->last().maxDuration,
      vbles <- link.end -> collect(vbles),
      rhs <- r,
      lhs <- l
    ),
    r <- Beh!Pattern (
      els <- link.end -> collect (rhs : RHS | rhs.els)
    ),
    l <- Beh!Pattern (
      els <- link.end -> collect (lhs : LHS | lhs.els)
    )
}
```

The MergeRules rule receives as input a link among rules that has been previously identified by the modeler. From that link, we can access its end-points, which are a system rule and an observer rule. This ATL rule creates a behavioral rule which is the result of putting the elements from both (the system and the observer) rules into one. The value of the attributes name, soft, lowerBound, upperBound and maxDuration (specified in the *Behavior* metamodel [14]) are those of the system rule, which is the second (and last) element in the end reference of the MergeRuleLink. For the objects and variables to be present in the resulting rule, the OCL collect operation is used in order to gather the elements from both rules together.

4.5. Room for Improvement

We have shown how the modular and independent specification of observers and their later weaving with the system specification can be achieved, thus facilitating the specification of observers for monitoring system properties. The modular definition of observers allows the reuse of observers across systems, and provides support for defining the semantics of observers once and then reuse them, instead of having to specify the behavior of observers once and again. Modularity also improves the maintainability of the specifications, and achieves simplicity by separating independent concerns.

Based on our initial experiments we have seen that most of the usual QoS properties can be easily and naturally expressed following the modular approach. However, some further research, and improvements in the current approach tool implementation, are required to make it really useful. For example, if we wish to use observers to define systems with self-adaptive behavior (cf. Section 5.1), the rules for self-adaptation should make use of the woven model of the system. This woven model might require some modifications to incorporate the adaptation rules based on the observers states. Similar situations may occur in other cases, and thus it would be very useful to provide facilities for rule modification on the woven system without actually manipulating the resulting specification. Facilities for rule replacement or redefinition, inside the model management world, would be useful here too.

Some particular observers may also imply rather complex bindings, since their attributes may depend on the kind of objects present on the system rules, or may even require the introduction of new system objects in the rules because they were not initially present. Further capabilities could facilitate this task.

Finally, there is the issue of the semantics of the resulting system specification after the weaving process is performed. Notice that by introducing observers we may be modifying the behavior of our system, both when doing it by hand and when using the modular approach. A deeper understanding of the meaning and semantics of the weaving operation in the modular case is required. In this respect, this proposal has just served as a proof-of-concept that such an approach is feasible, but still a proper study on the foundations of such approach, as well as of its applicability and correctness, is required. We claim that, with the appropriate formalization and requirements, the preservation of the semantics of the system specification by the merging could be proven. Some results on this direction are expected to be available soon.

5. Performance analysis

The rules described in the previous sections allow users to specify the behavior of their systems and of their monitoring observers. Once these specifications are completed, we show in this section how we can analyze them, together with the tool available for conducting such analyses.

In *e-Motions*, the semantics of real-time specifications is defined by means of transformations to another domain with well-defined semantics, namely Real-Time Maude [32]. The *e-Motions* environment not only provides an editor for writing the visual specifications, but also implements their automatic transformation (using ATL [31]) into the corresponding formal specifications in Maude—in a way transparent to the user.

One of the benefits of this approach is that it enables the use of Maude's facilities and tools available for executing and analyzing the system specifications once they are expressed in Maude. The work in [15, 16] present some examples of analyses that can be performed on rule-based DSL specifications using Maude. Furthermore, Maude rewriting logic specifications are executable, and therefore they can be used as a prototype of the system on which to carry on different types of checks and perform simulations.

In Maude, the result of a simulation is the final configuration of objects reached after completing the rewriting steps, which is nothing but a model. The resulting model can then be transformed back into its corresponding EMF notation, allowing the end-user to manipulate it from the Eclipse platform. The semantic mapping as well as the transformation process back and forth between the e-Motions and Real-Time Maude specifications are described in detail in [16]. These transformations are completely transparent to the *e-Motions* user. In this way the user feels like working only within the *e-Motions* visual environment, without the need to understand any other formalisms and being completely unaware of the Maude rewriting engine performing the simulation.

Another very important advantage with our approach is that observers are also objects of the system, and therefore the values of their attributes can be effectively used to know how the system behaved after the simulation is carried out. For example, if we start the simulation from the initial configuration of the system shown in Fig. 5, the values obtained by the observers after running a simulation are shown in Table 1 (recall that the counter attributes of the generator machines were set to 40). Let us remind the reader that a given production time of 40 time units for a machine indicates that it will take an amount of time within the range [37,43]. Similarly, a production time of 30 means that the real production time taken by that machine to produce a part will be within the range [27,33]. In the following tables we will consider that units of time correspond to seconds, to show the simulation results in a clearer way—e.g., we will write 9'05" (meaning 9 minutes and 5 seconds) instead of writing 545 units of time.

Table 1 shows the performance parameters achieved by the system, whose simulation indicated that the production time was 27'16" (this is the time the system took to produce all the parts). Starting with the ThroughputOb observer, it indicates that 38 non-defective hammers have been collected. The throughput value is 1.39 (expressed in parts/minute), which means that 1.39 non-defective hammers are collected on average every minute. Two defective hammers have been produced, and the mean time between their collection was 9'05". Depending on the price of materials, this could be of greater or lesser concern. The mean time that a hammer is in the system is 4'05". Let us clarify that this is the time elapse from when its parts start being generated to the moment it is collected. The average delay of all hammers produced is 2'26", which means that they are in the system for around 2'26" more than their optimal (theoretical) production time. Finally, let us have a look at the time the machines have been working. The head generator and assembler have been working almost the whole time (95.4%). However, the handle generator is idle one third of the time. As expected, this is because its production time is faster than the other machines, so it produces the parts faster and has to wait for the other machines before being able to carry on.

We also need to study the reasons for the delay. In the first place, handles are generated faster than heads. This may cause an overload of handles in tray `t1` (see Fig. 5) and also in conveyor `c1`, which makes the handle generator to stop generating handles (see Sec. 3.3). This overload means that parts

ThroughPut	MTBF	MCT	Delay	HandleGen	HeadGen	Assembler
th:1.39	mtbf:9'05"	mct:4'05"	delay:2'26"	tBusy:71%	tBusy:95.4%	tBusy:95.4%
collected:38	defective:2	act:163'15"	acc:97'10"	tIdle:29%	tIdle:4.6%	tIdle:4.6%

Table 1: Observers results after a simulation with one Assembler (pt = 40 and defective_rate = 1), one GenHandle (pt = 30 and defective_rate = 5) and one GenHead (pt = 40 and defective_rate = 2). System production time: 27'16".

ThroughPut	MTBF	MCT	Delay	HandleGen	HeadGen	Assembler
th:1.41	mtbf:8'59"	mct:3'42"	delay:2'04"	tBusy:72.4%	tBusy:73.6%	tBusy:96.3%
collected:38	defective:2	act:148'3"	acc:82'28"	tIdle:27.6%	tIdle:26.4%	tIdle:3.7%

Table 2: Observers results after a simulation with one Assembler (pt = 40 and defective_rate = 1), one GenHandle (pt = 30 and defective_rate = 5) and one GenHead (pt = 30 and defective_rate = 5). System production time: 26'57".

stay in the system longer, increasing their cycle time and, consequently, their delay. To try to solve this problem, let us see what happens if the production time of the heads generator is set to 30" (and the defective rate to 5%) so that now the handles do not have to wait for the heads to be assembled because the production time of both machines is similar. The results of this simulation are shown in Table 2.

These results are similar to those in Table 1. Once again, 38 non-defective hammers and two defective hammers have been collected. The mean cycle time and delay of collected hammers have been slightly reduced, but they are still significant. The explanation for this is simple. Despite handle and head generators having similar production times, now parts which are ready to be assembled on tray `t1` have to wait for the assembler to be available, since its production time is greater than the production time of the generators. This is also the reason why the busy time of generators is smaller than the assemblers', which is not desirable. A solution to the problem of these big mean cycle times, delays and idle times of machines seems to be making the production time of all machines equal. As we have been simulating with two different production times (and, consequently, defective rates), let us do a simulation with all times set to 30" (and a defective rate of 5% for generators and 3% for assemblers, see Table 3), and another with 40" (and a defective rate of 2% for generators and 1% for assemblers, see Table 4).

Now we can see improvements in these two simulations regarding the mean cycle time, delay and idle time of the machines; the mean cycle time

ThroughPut	MTBF	MCT	Delay	HandleGen	HeadGen	Assembler
th:1.8	mtbf:7'35"	mct:1'37"	delay:8"	tBusy:95%	tBusy:94.8%	tBusy:93.3%
collected:38	defective:2	act:64'35"	acc:5'30"	tIdle:5%	tIdle:5.2%	tIdle:6.7%

Table 3: Observers results after a simulation with one Assembler (pt = 30 and defective_rate = 3), one GenHandle (pt = 30 and defective_rate = 5) and one GenHead (pt = 30 and defective_rate = 5). System production time: 20'48".

ThroughPut	MTBF	MCT	Delay	HandleGen	HeadGen	Assembler
th:1.4	mtbf:13'56"	mct:2'03"	delay:13"	tBusy:94%	tBusy:93.7%	tBusy:94.9%
collected:39	defective:1	act:81'53"	acc:9'01"	tIdle:6%	tIdle:6.3%	tIdle:5.1%

Table 4: Observers results after a simulation with one Assembler (pt = 40 and defective_rate = 1), one GenHandle (pt = 40 and defective_rate = 2) and one GenHead (pt = 40 and defective_rate = 2). System production time: 27'52".

of collected hammers has been substantially decreased and the delay is now quite small. The reason is that all machines work now at the same speed. The mean cycle time is bigger when the production time of machines is 40" because parts are within the system for a longer time until the user collects them. The busy and idle times in both situations are very similar (and acceptable), since the machines are working for almost the whole time. As expected, the simulation when the production times are 30" finishes earlier. In fact, it now takes seven minutes less: the system produces all parts in 20'48". Besides, the throughput of the former, 1.8, is better than the one of the latter, 1.4. However, in the former two defective hammers have been collected, while in the latter only one hammer was defective. Taking into consideration all these factors, we, acting as managers of the plant, should have to decide which configuration is better for our plant. For instance, if the costs of the materials were very expensive, we may prefer the second configuration, where the process takes 7' longer but we only get one defective hammer. However, in this case, since we are dealing with hammers, we would probably prefer to get two defective hammers and save 7 minutes. In any case, we now count on precise performance figures that allow us to take decisions and to assess their associated costs and impact (in terms of time and money).

5.1. Adding Rules for Self-Adaptation

Apart from computing the values of the properties that we want to analyze in the system, observers can also be very useful for defining alternative

ThroughPut	MTBF	MCT	Delay	HandleGen	HeadGen	Assembler
th:1.7	mtbf:11'30"	mct:1'50.3"	delay:15"	tBusy:95%	tBusy:94.9%	tBusy:93.5%
collected:39	defective:1	act:73'33"	acc:10'10"	tIdle:5%	tIdle:5.1%	tIdle:6.5%

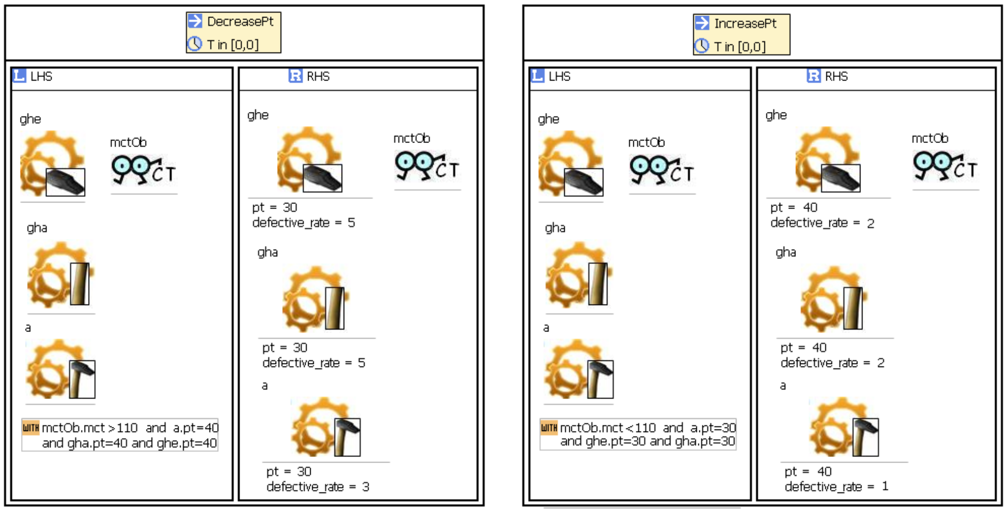
Table 5: Observers results of simulation with the system being self-adapting and pursuing a mct of 1'50". System production time: 22'59".

behaviors of the system, depending on specific threshold levels. For instance, the system can self-adapt under certain conditions, since we are able to search for states of the system in which some attributes of the observers take certain values, or go above or below some limits.

As an example, let us consider the mean cycle time value given by the MCTOb observer. This value computes, each time a new hammer is collected, the mean cycle time of all hammers produced so far, and it directly depends on the production time of the machines (as we saw in the previous section). Let us suppose that the production time of the machines can be changed during execution time. In fact, it is very common in real world systems, where the working speed of different machines can be adjusted according to certain parameters while the system operates.

Let us consider the two configurations that gave us better results in the previous simulations (those where all the machines had the same production time). The simulation of the configuration on which every machine has a production time of 40" (resp. 30") resulted in a final mean cycle time of 2'03" (resp. 1'37"). Now, suppose that we want to keep the value of the mean cycle time close to a given optimal value for us (let us say 1'50") by appropriately swapping the two configurations. Taking this into account, we include two new rules in the system to self-adapt its configuration (Fig. 21). The first rule, DecreasePt, will be triggered whenever the current mean cycle time of the system goes above 1'50" and the current production time of the three machines is 40". This rule will change the production times of every machine to 30" (and consequently their defective rate) with the aim of raising up the mean cycle time. Rule IncreasePt will then be triggered whenever the mean cycle time goes below 1'50" and the production time of the three machines is 30". It will change the production times of every machine to 40" (and accordingly their defective rate) to decrease the mean cycle time.

For checking the efficiency of this approach, we have carried out a simulation where the initial production time of every machine is 40". The result of



(a) DecreasePt Rule (b) IncreasePt Rule

Figure 21: The rules that describe how the system configuration self-adapts.

this simulation is presented in Table 5, which shows how the mean cycle time is close to 1’50”, as desired. In fact, we see that this is the most appropriate configuration: we obtain a throughput very close to the highest before (1.8) but with only one defective hammer.

5.2. Storing Attributes in Traces and Exporting Analysis Results

So far we have seen how it is possible to simulate the system and obtain performance information on its behavior. But there are also occasions in which we are not only interested in the performance indicators at the end of the system simulation, but also during its execution. For instance, we are now able to know the average cycle time for the produced parts, but it might be useful to see a graphical representation of how said value is changing while the system is operating. In addition, it would be important to have the resulting models designed in a way which can be analyzed and displayed by different tools (math programs, spreadsheets, etc.), outside the EMF environment.

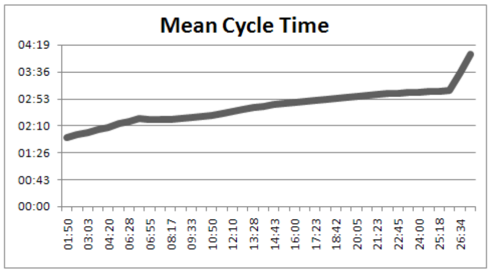
This section shows how to achieve this and the tool support that *e-Motions* provides. In the first place, if we want to keep track of how the values of the observed properties change during the system execution it is just a matter of changing the observers’ attributes to be sequences of values. And then the behavior of the observers needs to be slightly modified in the rules to append every computed value to these sequences (instead of gathering only the final value). This can be useful when we want to analyze the values of some property along time. It could also be useful if the user wants to use the values of the traces within the rules. For example, the Batch Means function described later in Section 6.2 uses these values.

Regarding the use of the resulting models by other tools, *e-Motions* implements a trivial model-to-text transformation that enables the creation of a comma-separated values (*csv*) file from an Ecore model. Such a *csv* file contains the information of every object in the model, together with the values of all its attributes. Objects are named by their identifiers, and attributes are expressed as a list of name-value pairs. That file can be directly imported by different applications for performing different kinds of analysis. For example, it can be fed to an spreadsheet application that the domain expert can use to analyze the data, display charts, etc. In this way, the domain expert will be able to easily display charts with the result of a simulation (which is in fact a model) to graphically represent the values of the parameters monitored by the observers throughout the whole simulation.

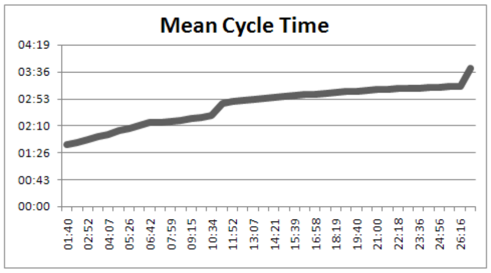
Fig. 22 shows, for instance, two charts that display the mean cycle time of the simulations whose final results were shown in Table 1 and Table 2, respectively. We can see in both of them how, in general, the mean cycle time of parts grows as time goes by. This is because tray `t1` gets overloaded at some point in the simulation, and when this occurs the generated parts at that moment increase the mean cycle time. However, in some time intervals the mean cycle time does not vary or even slightly decreases. This is caused by the parts that are produced when the generators re-start their work after stopping to avoid a deadlock (as discussed in Sec. 3.3), since these new parts will have a smaller mean cycle time than the average.

Fig. 23 shows the busy time of the handle generator in the simulation whose final results were shown in Table 1. In this simulation, the production time of the handle generator was smaller than the other machines. Therefore, handles were generated faster than heads and the handle generator had to eventually stop generating handles to avoid the overload of parts in tray `t1` and the overload of heads in conveyor `c2`. In particular, we see that the generator stops for the first time around minute 8 (moment at which the overload happened) and after this moment it continuously restarts and stops the generation of parts as needed, working as expected.

This analysis also indicates further potential points for improvement in



(a) Production times 40-30-40



(b) Production times 40-30-30

Figure 22: Mean Cycle Times.

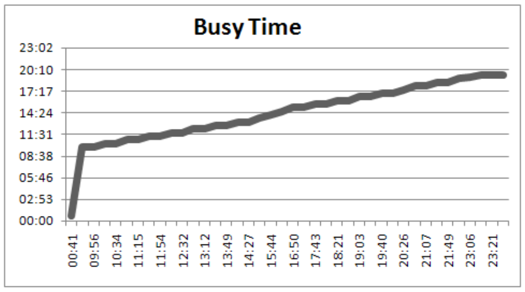


Figure 23: Handles generator busy time 40-30-40.

the system, such as replacing the current conveyors with faster ones (although probably more expensive—hence another trade-off to consider). Finding the right balance between costs, performance and benefits is not easy, although the kinds of analysis presented here can help solve this problem at the earliest phases of the system's design, with precise and objective figures, and also using notations and mechanisms which are quite close to the domain experts.

6. General methodology

After having described our proposal with a running example, this section presents a general methodology for the performance analysis of those systems which are specified using rule-based domain-specific languages—such as *e-Motions*. In addition, it explains the current tool support and how the simulations can be conducted with *e-Motions* to obtain the performance analysis results.

6.1. Methodology

In order to carry out the specification of a system and to obtain performance metrics by simulating the specifications with *e-Motions*, the following steps are required:

1. All the relevant elements of the system have to be specified. In an MDE setting, this is done by defining the abstract syntax of the system, which is specified by means of its metamodel. It describes the concepts of the system and the relationships between them. The concepts of the system are defined as classes in the metamodel. Element features are specified by means of class attributes.
2. Once we know the elements that may be present in the system, their relationships and features, the system behavior needs to be defined. In our approach this is specified by means of a set of rules, each of which represents a possible action in the system. These rules should be enough to cover all the aspects of interest for the system. The form of the rules is described in Section 2.
3. We then need to specify the initial configuration of the system, which is nothing but a model conforming to the system metamodel.

As a result of these steps we obtain the specification of the basic structure and behavior of the system, in a way that can be simulated. This simulation will execute the rules starting from an initial configuration of the system, which will be changing as the simulation evolves and the rules are applied. The result of the simulation will be a model (representing the final state of the system after the simulation) which conforms to the system metamodel.

If we also want to measure the performance of the system and conduct an analysis over a set of non-functional properties, the following additional steps are required:

4. We need to identify the non-functional properties that we want to analyze in the system, such as throughput, MTBF, delay, cycle time, etc. Section 3.1 describes some of these properties and how they can be defined.
5. Once we know the non-functional properties to analyze, we have to define the *Observers* that will be in charge of monitoring the state of the system and its elements during the simulation. For this, a metamodel for the observers has to be defined. There can be observers monitoring

properties of the global system and observers monitoring individual elements' features. Special care has to be taken when defining the attributes of the observers. Data structures must be properly defined depending on what we want to monitor. For example, for a specific feature of a specific observer we may be interested in storing just the last value, all the values taken during the simulation, or an aggregated of these (e.g., the average). The metamodel for the observers used in our running example is shown in Section 3.2.

6. When we know and have defined the observers for our system, we need to specify their behavior, and how their values are computed as the system executes. For this we have discussed two options: to modify the system behavioral rules defined in step 2 with the behavior of the observers; or to establish correspondences between the observers' behavioral rules and the system rules in case we already have the specification of observers' behavior. Examples of how rules are enriched with observers are discussed and shown in Section 3.3, and examples of weaving behavioral rules are shown in Section 4.3.
7. In the specification of self-adaptive systems, rules that make some changes in the configuration of the system depending on the observers' attributes may also be needed. An example was shown in Fig. 21, where the production time and defective rate of machines changed depending on the current value of the mean cycle time. Further changes would also be possible, such as the inclusion of new part generators, new assemblers, etc.

Once the complete behavior of the system and of the observer objects have been defined, we are in the position to run the simulations, visualize and analyze the results, and make those changes to the system that are required to improve its performance and behavior according to our requirements. These steps are described in detail in the following sections.

6.2. Conducting the Simulations

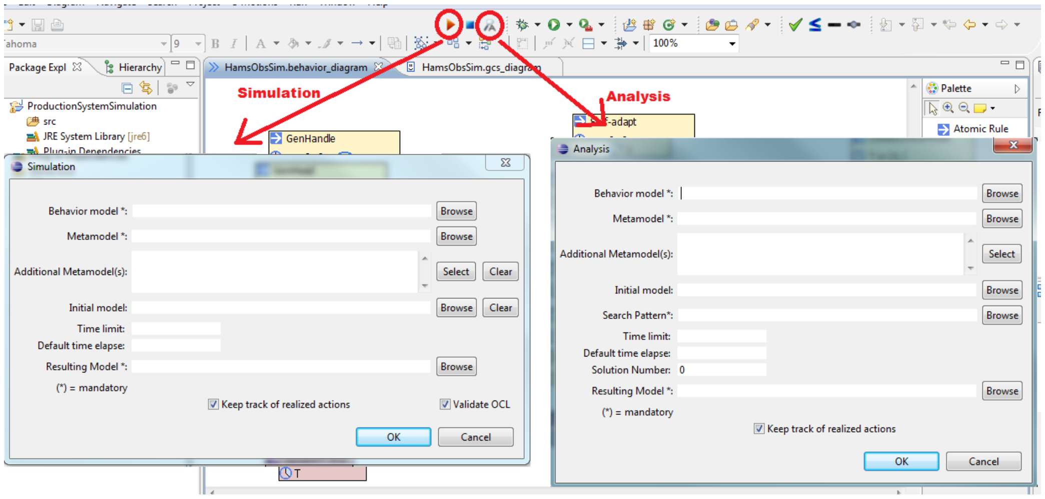
Normally the behavior of a non-trivial system is stochastic, and therefore it needs to be defined using probability distributions that specify the rates at which external events occur (e.g. arrival rates), the duration of the rules, the probability of failures, etc. The treatment of these stochastic events and the use of random data implies that two different simulations of the same system may produce different performance results, since the distributions provide different values. In order to deal with this, the user should launch several simulations for the same input model, and should aggregate the results appropriately.

There are several issues to be considered here: the length of the simulations, the number of simulations that have to be carried out to get meaningful results, and the aggregation function to use for combining the results.

For some systems, it is useful to set a time limit for the simulations, since the user may want to see the state of the system after a given number of time steps. In other cases, we are interested in simulating the system until a stable value is found (if it exists). There are different methods described in the literature to determine when the steady state of a simulation is reached, namely long runs, proper initialization, truncation, initial data deletion, moving average of independent replications, or batch means [33].

In our case we have implemented both the *long runs* and the *batch means* method (with slight modifications). The long runs method is useful when simulations do not last much, and many of them can be executed rapidly. In this case, the observers do not need to keep track of all intermediate states, just the final results. The original batch means method requires running a long simulation and later divide it up into several parts of equal duration, which are called batches. The mean of the observations within each batch is called the batch mean. The method requires studying the variance of these batch means as a function of the batch size. What we do, instead of running a long simulation and then dividing it, is to apply the method at certain points during the simulation as it moves forward. For that we store the values of the performance parameters in traces to be able to apply this method over them. In our Production Line system, we have applied the batch means method over the traces whenever 20 new hammers are assembled. We consider that a simulation reaches its steady state when the variance of the batch means of a trace goes below a threshold value (10^{-x}). Such value is different depending on the system and the precision that we want to get. Our results were obtained for 10^{-6}.

Finally, the results of a set of simulations need to be aggregated in order to provide one meaningful final result. Traditionally, the final value is obtained by calculating the average of the resulting values, and this is the way we currently compute our final result. However, the average does not always produce the most meaningful results. As future work we plan to analyze the results given by the different simulations to return not the mean, but the probability distribution that the results from the different executions follow.



Simulation

Behavior model *: Browse
Metamodel *: Browse
Additional Metamodel(s): Select Clear
Initial model: Browse Clear
Time limit:
Default time elapse:
Resulting Model *: Browse
(*) = mandatory
[✓] Keep track of realized actions [✓] Validate OCL
OK Cancel

Analysis

Behavior model *: Browse
Metamodel *: Browse
Additional Metamodel(s): Select
Initial model: Browse
Search Pattern*: Browse
Time limit:
Default time elapse:
Solution Number: 0
Resulting Model *: Browse
(*) = mandatory
[✓] Keep track of realized actions
OK Cancel

Figure 24: Simulating and Analyzing in *e-Motions*.

6.3. Tool Support

Our tool, *e-Motions* [24], has three main functionalities: model edition, model simulation and model analysis. The first two are directly related with the work here and the third one was developed for further kinds of analysis.

e-Motions provides model editors to graphically define the abstract and concrete syntaxes of DSVLs, and their behavioral rules. An abstract syntax is defined by means of a metamodel. The concrete syntax specifies how the domain concepts included in the metamodel are represented, and it is defined as a mapping between the metamodel and a graphical notation. This graphical notation is used to define initial models and also for specifying behavioral rules.

Once the metamodel and the behavioral rules (either with observers or not) are defined, the user can simulate the specifications (see Fig. 24). When launching a simulation, the user indicates: the file with the metamodel; the file with the behavioral rules; the time limit for the simulation (if not specified, the simulation runs until no further rule can be triggered); the default time elapse (1, unless otherwise specified); the initial model, and the file that will store the resulting model returned by the simulation. Before carrying out the simulation, *e-Motions* applies an integrated OCL validator that checks if the OCL expressions used in the rules are correct. In case of errors, these are displayed and the user warned. Otherwise, three ATL transformations are automatically applied to transform the specifications into Maude code.

One of them transforms the metamodel to its Maude specification [15, 34], the second one transforms the behavioral rules into Maude rules [20], and a third one is applied if there is an input file with the initial model [15].

Immediately after the transformations are applied, the simulation is launched. All this happens transparently to the user, who does not need to be aware of the existence of the Maude back-end tool, neither of the transformations happening in the background. Indeed, no Maude code is shown in the *e-Motions* tool. The simulation is executed in background, so the tool is not blocked and users can continue with their activities. A status bar is shown as long as the simulation is running. When it finishes, a notification message is shown and the file with the resulting model is created. This is a model conforming to the metamodel of the system. The user can also transform the data in the resulting model into a *csv* file by simply right-clicking on the file and selecting the transformation.

As mentioned above, random functions and probability distributions can be used in the behavioral rules. The most common distributions are available in *e-Motions*: Constant, Uniform, Exponential, Normal, Gamma, Weibull, Erlang, F, Chi-square, Geometric, Lognormal, Pareto, Pascal and Poisson. They are further explained in [24].

e-Motions also offers some analysis options, for example to perform *reachability* analysis. The tool makes use of the Maude search command, which allows users to explore, following a breadth-first strategy, the reachable state space of the system in different ways. Thus, it is possible for example to look for deadlock states without needing to simulate the whole system. In our example, a possible deadlock state is that tray t1 is full of parts of the same type, preventing the assembler machine from continuing. Once again, the reachability analysis is conducted from the *e-Motions* tool (see Fig. 24), which, again, hides the whole interaction with Maude from/to the user. When performing reachability analysis, the user has to specify the behavior model, the metamodel, the initial model, the search pattern (for example looking for deadlocks), the maximum number of solutions to return (since it may return hundreds of solutions) and where to store the resulting models.

Finally, it is important to remark that although our proposal has been implemented on top of *e-Motions*, it is applicable to any other rule-based domain-specific language that is able to describe the behavior of the system in terms of in-place rules, and that admits the specification of time-related properties. Examples of these languages are MOMENT2 [35] or the approach presented in [36], which proposes an alternative way to incorporate time to graph transformations.

6.4. Pros and Cons

This section discusses some of the main advantages and limitations of our proposal concerning performance analysis of systems.

As major advantages, our approach permits obtaining the performance analysis of end-user defined DSVLs, and is supported by a prototype tool. As long as the user is capable of describing the structure of the system in terms of a metamodel and can give behavioral semantics to it by means of in-place rules, the system can be simulated and analyzed. Furthermore, many different kinds of analysis can be performed. In fact, users can define as many types of observers as needed in order to analyze the properties of interest, which provides a high degree of flexibility. The models resulting from the simulations can be transformed into a format compatible with spreadsheet applications, so charts and tables with performance data can be easily obtained. Another advantage of our approach is that, in spite of using Maude for simulation, the user does not have to deal with any Maude code, since it is generated and used in a transparent way.

The possibility of using probability distributions in the behavioral rules enables the acquisition of performance metrics for stochastic systems, such as queuing networks, Petri nets, stochastic neural networks and genetic algorithms.

But our approach also presents some limitations. Firstly, learning to specify systems in terms of metamodels and behavioral rules is not obvious, and the learning curve is not negligible. Secondly, we also have to consider the time required to master the tool. Finally, simulations are always expensive (time-wise). The efficiency of our current implementation can be improved in some ways. Although simulating small systems with *e-Motions* is efficient and rather fast (in the order of a few seconds), as the complexity of systems grows (in terms of number of model elements) the simulations become slow. The use of DSVLs normally implies that models are not very large, because of the high-abstraction level at which the system is specified. However, there are cases of models in which the high number of elements (e.g., greater than 200 at this time) represents a heavy burden for *e-Motions*. To improve the speed of the simulations we are studying alternative representations of the models and behavioral rules in Maude, as well as the parallel distribution of simulations among servers. In this way, a simulation where probability distributions take part are distributed over several machines, each one dealing

# Model Elements	With no Observers	Global Observers	Individual Observers
30	0:00:01	0:00:01	0:00:12
100	0:00:06	0:00:18	0:02:00
120	0:00:12	0:00:34	0:04:26
150	0:00:35	0:01:07	0:08:51
200	0:01:12	0:01:56	0:17:43
300	0:02:28	0:03:32	0:59:42
400	0:03:45	0:05:04	1:59:24
500	0:05:51	0:07:38	4:56:07
600	0:07:31	0:10:30	8:36:08
750	0:11:16	0:17:15	14:51:18
900	0:15:33	0:32:12	31:05:04
1000	0:19:55	0:36:45	127:26:48

Table 6: Simulation times for the Production Line system.

with one system simulation. When the simulations are finished, the resulting models are gathered and the corresponding results are returned.

We have also evaluated the overhead of introducing the observers in the system specifications. They are objects, too, and hence they may mean multiplying by two or three the number of objects in the model if individual observers are used for all system elements. For small systems this is an acceptable increase: for systems with less than 100 elements the simulation time goes from a few seconds to a few minutes. However, the exponential nature of the Maude simulations and our current implementation of the tool impose a limit on the number of elements if simulations are to be finished in a reasonable time—see Table 6.

7. Related Work

As mentioned in the introduction, the usual approach to specifying the properties of the system that users want to analyze through simulation consists of enriching the system elements with new states and several kinds of annotations. Although this might (partially) work for UML models, the situation is different when the models of the system are specified using domain specific visual languages, for which no clear solution currently exists. Most of the existing proposals for specifying QoS and other non-functional properties require skilled knowledge of specialized languages, which is precisely what DSVLs try to avoid with their notations closer to the end-users and to the problem domain.

Other analysis tools (such as ARENA [37]) allow users to specify the models to be simulated using visual notations, but just within the tools environments and using their proprietary notations. In other words, these tools cannot easily take as input models produced by different editors, nor they can easily export their models so that they can be analyzed by other tools. In addition, users cannot arbitrarily use these tools with their in-house developed visual languages, nor define in a flexible way the properties to be monitored and analyzed (let them be properties of the whole system or of any of its elements). In our proposal we separate the visual specification of the system from the tools that will be finally used to simulate or analyze them. Furthermore, the fact that we can use user-defined monitoring objects that are added to the system specifications for capturing the properties to be analyzed allows a high degree of flexibility, since it is the end-user who defines the observer objects as part of his system design.

Our observers were originally introduced in [18] for specifying QoS properties, and in [17] it was discussed how they could be used during simulations. In this paper we have shown how they can be effectively used for conducting performance analysis. We have also introduced the use of probability distributions, which gives us the possibility of analyzing stochastic systems, and the use of the batch means method for deciding when simulations become stable. More importantly, a generic methodology that supports our approach has been presented. One of the benefits of our proposal is that it can be easily applied to other proposals that also advocate the use of in-place rules for specifying the behavior of real-time systems [38, 35, 39, 40], and specially those that propose visual notations for doing so.

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 [13]. 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 to provide 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.

Regarding the model of time used, Boronat and Olveczky also present in [35] the use of in-place model transformations to complement metamodels with timed behavioral specifications. However, the way in which they model time is different. They add explicit constructs for defining time behavior and include time constructs in the system state whose semantics is encoded in MOMENT2. We, instead, do not add any explicit constructs for defining time behavior, but our transformation rules have time intervals denoting the duration interval of each local action. Moreover, any OCL expression and variable used within the rule can be used for denoting the duration of the rule, which permits us to model stochastic systems if we use probability distributions for the durations. In [36], de Lara et al. also use metamodels to describe the abstract syntax of systems. Then, they use DPO Graph Transformation rules [41] to specify its behavior. To introduce an implicit notion of simulation time, they extend their grammars with scheduling functions, associating edges with relative time values and with probability density functions. In that way, they can model either specific times as well as discrete and continuous distributions, such as the uniform, normal and exponential negative. They also propose how to measure some performance metrics, although they do not propose a complete solution. Furthermore, we can use random values and any probability distribution in our rules, and not only for time durations but also for any other values within the rules.

Stochastic Graph Transformation systems were defined by Reiko Heckel in [42]. They are an extension of graph transformations where the duration of the rules can follow probabilistic distributions. They have been used to study the performance and reliability of different kinds of systems [43]. At the beginning, this approach was based on the use of analytical methods, which generally require some simplifications on the kinds of probability distributions they admit, and have also limitations to cope with the huge state spaces of current software systems. To overcome these limitations, more recent work in this field proposes the use of stochastic graph transformation simulations, which allows larger models and more general distributions. For example, in [44] Heckel and Torrini add a model of global time and apply stochastic simulations on mobile systems, and in [45] they use stochastic simulation in order to verify software performance in large-scale systems.

In [40], de Lara and Vangheluwe presented an interesting approach to automatically generate model-to-model transformations from DSVLs into semantic domains with a explicit notion of transition. From the initial model of the system and the behavioral rules, they generate a transformation which is expressed in the form of operational triple graph grammar rules. More precisely, they transform a production system described with a metamodel, an initial model and a set of behavioral rules into a Timed Petri Net (TPN). They allow the specification of the duration of rules as an interval between a lower and an upper bound. Such rule durations are then translated into durations in transitions in the TPNs. An important advantage of this approach is that after translating a system to its representation in Petri nets, specific Petri net techniques can be used to analyze said system. In spite of the usefulness of this new technique, it imposes some strict requirements on the behavioral rules in order to successfully transform them into Petri nets. Furthermore, it is not possible to use any probability distribution on the rules duration. On the other hand, the use of Petri nets allows some interesting kinds of analysis on the system, and this is why this proposal can be considered as complementary to ours.

An alternative approach to relying on observers, as we do in our approach, is to rely on traces that can be generated from the transformations. The analysis of system execution traces to validate QoS properties has proved to be very effective in the case of component-based systems, network protocols and distributed middlewares [46, 47, 48], and allows different kinds of very powerful analysis, as described in [49, 50]. But in fact, our observers can be used to generate selected sets of traces, as we have shown in Section 5.2, and then use these traces (stored in the observers' attributes) to analyze the system. The advantages in our approach is that the user, when defining the observers, can select the kind of execution traces he is interested in—instead of having the transformations generate all system traces, which can result in too much information.

Finally, [7] introduces a generic specification language for non-functional properties of component-based systems that has a formal foundation and semantics, using temporal logic, in which non-functional properties are specified as constraints over measurements. Although this approach is different in nature to the one presented here, it can probably be used for providing some interesting semantic foundations that can help reasoning about the modular addition of observers to the system specifications.

8. 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 performance analysis of systems specified by user-defined DSVLs—in particular those whose behavior is specified in terms of in-place rules. Observers allow extending the global state of the system with the variables that the designer wants to analyze when running the simulations, being able to capture the performance properties of interest. Furthermore, the fact that action executions are first-class citizens in the *e-Motions* specifications [14] can enable observers to monitor not only the states of the objects of the system but also their actions.

We have also presented the *e-Motions* prototype tool for specifying the systems and their observers using DSVLs, and the facilities it provides for conducting different analyses of these systems (within *e-Motions* or by feeding its output models into other tools by means of *csv* files).

We have described a general methodology for the performance analysis of systems using our approach, after introducing it with a running example, and discussed the current tool support—in particular how simulations can be conducted and the kinds of analysis that are possible with our proposal. Further examples and case studies can be found in [24]. We have also presented an approach for applying a modular specification of the behavior of observers based on weaving mechanisms for rules. We propose the decoupling in the definition of the functional and non-functional behavior of a system by separating the functional rules and the inclusion of observers within them.

Other than that, we also plan to study in more detail the expressiveness of our approach, investigating which kinds of properties can be analyzed using observers and which ones cannot. We are also working on improving the internal representation of our models and of the observers in Maude, to make simulations much faster. We would also like to explore the use of other existing tools to perform different kinds of analysis based on the output models of our simulations (e.g., predictions). Some graphical aid to specify the weaving models between the system and the observers rules is also planned as a future activity, in order to provide a more user-friendly interface than the current AMW Eclipse-based editor. In addition, weaving is currently accomplished outside the *e-Motions* environment but would need to be fully integrated within the tool.

Finally, we would like to explore the possible connection between the final implementation of the systems and their specifications in *e-Motions* in order to study the conformance of the system execution with its specified behavior. In this way we could easily detect deviations from the expected behavior and degradations in the system performance with regards to its high-level visual specifications.

Acknowledgements. The authors would also like to thank the anonymous referees for their insightful comments and very constructive suggestions. This work has been supported by Spanish Research Projects TIN2008-031087 and TIN2011-23795.

References

[1] J. de Lara, H. Vangheluwe, Defining visual notations and their manipulation through meta-modelling and graph transformation, Journal of Visual Languages and Computing 15 (3–4) (2006) 309–330.

[2] J. de Lara, H. Vangheluwe, Translating model simulators to analysis models, in: Proc. of FASE 2008, no. 4961 in LNCS, Springer, 2008, pp. 77–92.

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

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

[5] C. Ermel, K. Holscher, S. Kuske, P. Ziemann, Animated simulation of integrated UML behavioral models based on graph transformation, in: Proc. of VL/HCC'05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 125–133.

[6] S. Balsamo, A. D. Marco, P. Inverardi, M. Simeoni, Model-based performance prediction in software development: A survey, IEEE Trans. on Software Engineering 30 (5) (2004) 295–310.

[7] S. Zschaler, Formal specification of non-functional properties of component-based software systems, Software and System Modeling 9 (2) (2010) 161–201.

[8] V. Cortellessa, A. D. Marco, P. Inverardi, Integrating performance and reliability analysis in a non-functional MDA framework, in: Proc. of FASE 2007, no. 4422 in LNCS, Springer, 2007, pp. 57–71.

[9] M. Fritzsche, H. Bruneliere, B. Vanhooff, Y. Berbers, F. Jouault, W. Gilani, Applying megamodelling to model driven performance engineering, in: Proc. of ECBS'09, IEEE Computer Society, 2009, pp. 244–253.

[10] M. Fritzsche, J. Johannes, U. Aßmann, S. Mitschke, W. Gilani, I. Spence, J. Brown, P. Kilpatrick, Systematic usage of embedded modelling languages in automated model transformation chains, in: Proc. of SLE'09, no. 5452 in LNCS, Springer, 2009, pp. 134–150.

[11] OMG, UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, OMG, ptc/04-09-01 (Sep. 2004).

[12] OMG, UML Profile for Schedulability, Performance, and Time Specification, OMG (Jan. 2005).

[13] OMG, A UML Profile for MARTE: Modeling and Analyzing Real-Time and Embedded Systems, OMG (Jun. 2008).

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

[15] J. E. Rivera, A. Vallecillo, 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) (2009) 778–792.

[16] J. E. Rivera, F. Durán, A. Vallecillo, On the behavioral semantics of real-time domain specific visual languages, in: Proc. of WRLA'10, no. 6381 in LNCS, Springer, 2010.

[17] J. Troya, J. E. Rivera, A. Vallecillo, Simulating domain specific visual models by observation, in: Proc. of the Symposium on Theory of Modeling and Simulation (DEVS'10), Orlando, FL (US), 2010.

[18] J. Troya, J. E. Rivera, A. Vallecillo, On the specification of non-functional properties of systems by observation, in: Proc. of NF-PinDSML'09, Vol. 553, CEUR Workshops, Denver, Colorado, 2009.

[19] K. Czarnecki, S. Helsen, Classification of model transformation approaches, in: OOPSLA'03 Workshop on Generative Techniques in the Context of MDA, 2003.

[20] J. E. Rivera, E. Guerra, J. de Lara, A. Vallecillo, Analyzing rule-based behavioral semantics of visual modeling languages with Maude, in: Proc. of SLE'08, no. 5452 in LNCS, Springer, Tolouse, France, 2008, pp. 54–73.

[21] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude – A High-Performance Logical Framework, no. 4350 in LNCS, Springer, 2007.

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

[23] M. H. Ter Beek, A. Fantechi, S. Gnesi, F. Mazzanti, An action/state-based model-checking approach for the analysis of communication protocols for service-oriented applications, in: Proc. of FMICS'07, Springer, 2008, pp. 133–148.

[24] Atenea, e-Motions, `http://atenea.lcc.uma.es/E-motions` (2011).

[25] ISO/IEC FCD 10746-5, Information Technology – Open Distributed Processing – Quality of Service (1999).

[26] Eclipse, The Eclipse Modeling Framework (EMF), `http://www.eclipse.org/modeling/emf/`.

[27] S. Clarke, E. Baniassad, Aspect-Oriented Analysis and Design, Addison-Wesley Professional, 2005.

[28] F. García, M. F. Bertoa, C. Calero, A. Vallecillo, F. Ruíz, M. Piattini, M. Genero, Towards a consistent terminology for software measurement, Information and Software Technology 48 (8) (2006) 631–644.

[29] M. D. D. Fabro, P. Valduriez, Semi-automatic model integration using matching transformations and weaving models, in: The 22nd Annual ACM SAC, MT 2007–Model Transformation Track, Seoul, Korea, 2007, pp. 963–970.

[30] Atenea, Modular Approach for the Observers' Specification in the PLS case study, `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/E-motions/PLSObModExample` (2012).

[31] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: a model transformation tool, Science of Computer Programming 72 (1–2) (2008) 31–39.

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

[33] R. Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling, Wiley, 1991.

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

[35] A. Boronat, P. C. Ölveczky, Formal real-time model transformations in MOMENT2, in: Proc. of FASE 2010, no. 6013 in LNCS, Springer, 2010, pp. 29–43.

[36] J. de Lara, E. Guerra, A. Boronat, R. Heckel, P. Torrini, Graph transformation for domain-specific discrete event time simulation, in: Graph Transformations, no. 6372 in LNCS, Springer, 2010, pp. 266–281.

[37] Rockwell Automation, Arena Simulation Software, `http://www.arenasimulation.com/` (2011).

[38] S. Gyapay, R. Heckel, D. Varró, Graph transformation with time: Causality and logical clocks, in: Proc. of ICGT 2002, Springer, 2002, pp. 120–134.

[39] S. Burmester, H. Giese, M. Hirsch, D. Schilling, M. Tichy, The Fujaba real-time tool suite: model-driven development of safety-critical, real-time systems, in: Proc. of ICSE'05, ACM, 2005, pp. 670–671.

[40] J. de Lara, H. Vangheluwe, Automating the transformation-based analysis of visual languages, Formal Aspects of Computing 22 (3-4) (2010) 297–326.

[41] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, Monographs in TCS, Springer, 2006.

[42] R. Heckel, G. Lajios, S. Menge, Stochastic graph transformation systems, Fundam. Inform. 74 (1) (2006) 63–84.

[43] A. Khan, R. Heckel, Model-based stochastic simulation of super peer promotion in P2P VoIP using graph transformation, in: Proc. of DC-NET/OPTICS'11, 2011, pp. 32–42.

[44] R. Heckel, P. Torrini, Stochastic Modelling and Simulation of Mobile Systems, in: Graph Transformations and Model-Driven Engineering, Vol. 5765 of LNCS, Springer, 2010, pp. 87–101.

[45] P. Torrini, R. Heckel, I. Rth, Stochastic Simulation of Graph Transformation Systems, in: D. Rosenblum, G. Taentzer (Eds.), Proc. of FASE'10, Vol. 6013 of LNCS, Springer, 2010, pp. 154–157.

[46] J. Drummond, V. Berzins, Luqi, W. Kemple, A. Mikhail, N. Chaki, Quality of service behavioral model from event trace analysis, in: Proc. of the 7th International Command and Control Research and Technology Symposium, Quebec, Canada, 2002.

[47] D. Mania, J. Murphy, J. Mcmanis, Developing performance models from nonintrusive monitoring traces, in: Proc. of Proceeding of Information Technology and Telecommunications (IT&T), The MIT Press, 2002.

[48] J. M. Slaby, S. Baker, J. Hill, D. C. Schmidt, Applying system execution modeling tools to evaluate enterprise distributed real-time and embedded system QoS, in: Proc. of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications, IEEE Computer Society, Sydney, Australia, 2006, pp. 350–362.

[49] M. Hauswirth, A. Diwan, P. F. Sweeney, M. Mozer, Automating vertical profiling, in: Proc. of OOPSLA'05, San Diego, California, 2005.

[50] J. H. Hill, Data mining execution traces to validate distributed system quality-of-service properties, in: In: Knowledge Discovery Practices and Emerging Applications of Data Mining: Trends and New Domains, IGI Global, 2011, pp. 174–197.