On the Modular Specification of NFPs: A Case Study

Antonio Moreno-Delgado, Javier Troya, Francisco Durán, and Antonio Vallecillo

GISUM/Atenea Research Group. Universidad de Málaga (Spain)
{amoreno, javiertc, duran, av}@lcc.uma.es

Abstract. The modular specification of non-functional properties of systems is a current challenge of Software Engineering, for which no clear solution exists. However, in the case of Domain-Specific Languages some successful proposals are starting to emerge, combining model-driven techniques with aspect-weaving mechanisms. In this paper we show one of these approaches in practice, and present the implementation we have developed to fully support it. We apply our approach for the specification and monitoring of non-functional properties using observers to a case study, illustrating how generic observers defining non-functional properties can be defined in an independent manner. Then, correspondences between these observers and the domain-specific model of the system can be established, and then weaved into a unified system specification using an ATL model transformation. Such a unified specification can also be analyzed in a natural way to obtain the required non-functional properties of the system.

Key words: non-functional properties, domain-specific languages, model transformations, weaving mechanisms

1 Introduction

Models and specifications of systems have been around the software industry from its very beginning, but Model-Driven Engineering (MDE) has come to articulate these models so that the development of information systems can be automated. MDE has raised the level of abstraction at which systems are developed in practice, yielding the focus in the development of software to models. Today, models are being used not only to specify systems, but also to simulate, analyze, modify and generate code of such systems. The popularity of MDE is indeed continuously growing, mainly because of the maturity of model transformation technologies.

MDE is at present being applied in many different scenarios, however it still needs appropriate mechanisms to handle the development of complex systems. Although some advances have happened in recent years, more powerful mechanisms for modularity and reusability of models, metamodels and model transformations are still to come if the technological solutions are to be applied on real developments.

Together with model transformations, the other key component making MDE so appealing is the use of Domain-Specific Modeling Languages (DSMLs). The use of languages based on concepts of the problem domain allow domain-experts, independently of their programming skills, to construct or participate in constructing substantial parts of new systems. Moreover, the implicit knowledge associated to specific domains may make it possible to provide much more extensive code generation, allowing, e.g., the production of executable systems from relatively simple DSML-based models [11].

Using DSMLs, however, comes at the cost of having to develop a new language for every new domain/application. The definition of a DSML involves at least three aspects: abstract syntax, concrete syntax, and semantics. In MDE, the abstract syntax of a DSML is usually defined by means of a metamodel, which basically is a model that describes the concepts of the language, the relationships between them, and the structuring rules that constrain the model elements and their combinations to respect the domain rules. The concrete syntax is usually defined as a mapping between the metamodel concepts and a textual or graphical notation. However, the way in which to define the semantics of DSMLs is far from finding a consensus. Defining DSMLs by means of their structural aspects—abstract and concrete syntaxes—allows the rapid development languages and some of their associated tools, such as editors or browsers, but, to take full advantage of MDE, and to allow further activities such as simulation or behavioral analysis, the specification of such behavioral semantics of DSMLs is also required.

There currently are different proposals to represent DSML operational semantics, using UML behavioral models [10], Abstract State Machines [3], in-place model transformations [16], etc. In the context of MDE it seems natural to describe them by means of models, so that they may be integrated in the MDE process and tools. Moreover, since DSMLs and model transformations are the key artifacts in MDE, in-place model transformations seem a natural way to specifying the behavior of DSMLs.

The use of in-place model transformations, either textual or graphical, provides a very intuitive way to specify behavioral semantics, close to the language of the domain expert and at an appropriate level of abstraction (see [6, 16]). In-place transformations are defined by a set of rules, each of which represents a possible *action* of the system. These rules are of the form $[\text{NAC}]^* \times \text{LHS} \rightarrow \text{RHS}$, where 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 conditions for the rule to be applied, whereas the RHS represents the effect of the corresponding action. The transformation of the model proceeds by applying the rules on sub-models of it in a non-deterministic order, until no further transformation rule is applicable.

The definition of DSMLs by means of metamodels and in place-model transformations has been shown to be an effective mechanism. However, to make the development of DSMLs efficient, we must be able to construct them from reusable building blocks, given appropriate mechanisms to reuse and compose partial languages for new domains. So far, the contributions in this direction are scarce though. Although there are some proposals for the modularity and reusability of models and metamodels (see, e.g., [7, 4]), there is very little for behavior definitions. The MetaDepth system [5] is perhaps the currently most interesting proposal, where concepts, templates and mixing layers are proposed for generic metamodeling, and where behavior defined for concepts and templates, independently of metamodels, using EOL [13] or Java, can be reused. For DSMLs whose dynamic behavior is defined with in-place model transformations, there is no current support for modularity.

$$\begin{tikzcd}
& \boxed{M_C = MM_C \oplus Rls_C} \arrow[dl, Rightarrow, "f"'] \arrow[dr, Rightarrow, "g"] & \\
\boxed{M_A = MM_A \oplus Rls_A} \arrow[dr, dotted] & & \boxed{M_B = MM_B \oplus Rls_B} \arrow[dl, dotted] \\
& \boxed{M_A \otimes_{M_C} M_B = (MM_A \otimes_{MM_C} MM_B) \oplus (Rls_A \otimes_{MM_C \oplus Rls_C} Rls_B)} &
\end{tikzcd}$$

Fig. 1. Amalgamation in the category of DSMLs and DSML morphisms.

The semantics of DSMLs whose dynamics is described using in-place transformations is typically given by graph grammars [17]. Modularity and reusability of graph transformation systems have been largely studied, with well established results on modularity, encapsulation, composition, etc. Based on these ideas, we developed in [20, 9] a novel mechanism for the composition of DSMLs thus defined based on graph pushouts and rule amalgamations. In [8], we proved that behavior was protected under certain circumstances. Based on these ideas, we present in this paper a novel implementation of this composition operation of DSMLs, and illustrate its use on a case study describing the check-in system of an airport.

Although our implementation is based on the *e-Motions* language and system [14, 15], both our approach and formal framework are applicable to any DSL specification whose semantics is based on in-place model transformations. Moreover, while our work is clearly motivated from the need of modularizing the specification of non-functional properties (NFPs), the formal framework covers arbitrary conservative extensions of such DSMLs. The *e-Motions* system allows the definition of visual DSMLs and their semantics through in-place model-transformation rules, providing support for their analysis through simulation, reachability analysis, and model checking.

[19] builds on the ideas of the *e-Motions* framework to keep track of specific NFPs by adding auxiliary objects to DSMLs. However, that approach requires the NFP specification and analysis component to be redefined from scratch for every new DSML. [20, 9] build on that work, but with the aim of introducing modularization to the definition of NFPs so that different NFPs could be independently specified and later added. Here, we apply ideas from [8] to support the modular definition of any kind of DSML.

Given DSMLs A, B, and C, defined by respective metamodels (MM_i) and behaviors (transformation rules Rls_i), $M_i = MM_i \oplus Rls_i$, for $i = A, B, C$, the weaving of DSMLs A and B along C will correspond to their amalgamation along C. Although we will illustrate it with examples along the paper and will describe its implementation in Sect. 5, let us just say that given DSML morphisms $f : C \to A$ and $g : C \to B$, which basically map each element in its source DSML to its corresponding target element in the target DSML, their amalgamation is constructed by constructing the pushout of the corresponding metamodel morphisms (in the appropriate category) and the amalgamation of their rule morphisms one by one. Fig. 1 shows the amalgamation of DSMLs A and B along C. The formal details for this construction can be found in [8].

Given the situation in Fig. 1, it is of particular interest the case in which one of the DSML morphisms, let us say f, is an inclusion, since then the DSML M_C may be seen as a parameter of M_A. In this situation, it has been proved in [8] that if g protects—i.e. both preserves and reflects—behavior, and g reflects behavior, then the induced morphism between M_B and $M_A \otimes_{M_C} M_B$ is an inclusion, and more importantly, it also protects behavior. This result is very important, since we have that once we have defined our system model, an airport check-in system in our case study, we can later merge the DSMLs corresponding to the NFPs we are interested in with it, with the guarantee that its behavior will remain as originally defined.

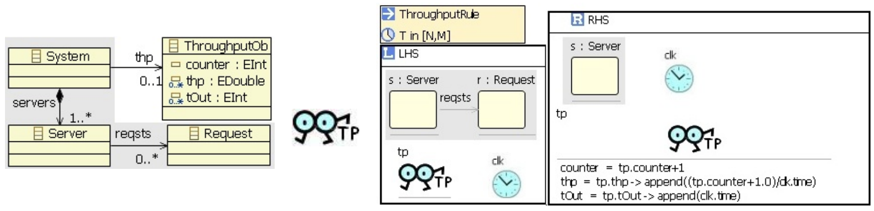
The rest of the paper is structured as follows. Sect. 2 shows how we can use the modularization described here to define independent languages for NFP observers. Specifically, we present DSMLs modeling throughput, response time and cycle time. Sect. 3 presents our case study. Sect. 4 describes how we apply our approach on it. We show how to extend the check-in system DSML presented in Sect. 3 with the performance analysis observer DSMLs, thus obtaining a DSML modeling the check-in system with observers for the analysis of its performance. We show the results of different observations on a simulation of the system. Then, Sect. 5 outlines the implementation. The paper finishes with some conclusions and future work in Sect. 6.

2 Independent Definition of Observers

In communication networks, *throughput* is defined as the average rate of message delivery over a communication channel. However, the notion has also been used in other disciplines, acquiring a more general meaning. We can define throughput as the average rate of work *items* flowing through a system. Thus, the same generic notion allows us to measure the number of information packages being delivered through a network, the number of cars being manufactured in a production line, or the number of passengers checking-in in an airport desk.

Given this more general definition, and given the description of a system, to measure throughput we basically need to be able to count the number of items delivered or produced, and calculate its quotient with time. We define a ThroughputOb observer class with attributes counter and thp keeping these values. The metamodel for the DSML of ThroughputOb observers may be the one depicted in Fig. 2(a). ThroughputOb observer objects will basically count instances of some *generic* class Request, which, as we will see in Sect. 4, will later be instantiated to passengers, as could be instantiated to data packages or to cars. These ThroughputOb objects will be associated to specific systems, so that we may, e.g., measure the throughput of each of the connections in a network, each of the production lines in a factory, or each of the check-in desks in an airport, as we will see in Sect. 4.

In *e-Motions*, the concrete syntax is provided by a GCS (Graphical Concrete Syntax) model. Given the concrete syntax to be used, depicted in Fig. 2(b), the behavior of ThroughputOb objects is defined by transformation rules. Specifically, we describe its behavior by the single rule in Fig. 2(c). This rule represents the way in which the values of its counter and thp attributes are to be updated. This rule is intended to be interpreted as generic: when a request disappears, the ThroughputOb observer gets up-



(a) Abstract Syntax (b) C.S. (c) Behavior

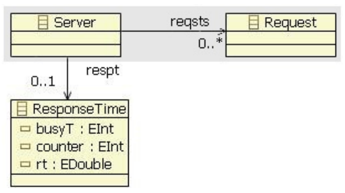
Fig. 2. Throughput observer DSML definition.

dated. Given a model in which there is a Server object s with a pending Request object r, plus a ThroughputOb object tp, and the system clock clk—its LHS—it can evolve to a system in which the request vanishes, and the observer object gets updated—its RHS. A few further comments are due to fully understand this rule: (a) the clk object is an instance of the predefined Clock class, which represents the pass of time in the system, and whose time attribute keeps the time of the system since its start-up (see [14] for a detailed presentation of the modeling of time in *e-Motions*); (b) since classes System, Server and Request are generic, no concrete syntax is provided for them, and therefore instances of them are represented as boxes; and (c) the rule heading (box on the rule) shows its identifier, ThroughputRule, and the time T that action takes, in this case some non-deterministic value in the range [N, M], for some values N and M that will become later instantiated (see Sect. 4). Note that this definition of throughput is rather naive, since the thp attribute only changes when requests are consumed. We have used this definition here to simplify presentation. See [20] for a more accurate definition of throughput. Note also that this DSML is not usable by itself, it is a generic DSML, and will be instantiated before used. Please notice that the parameter part of generic DSMLs is depicted shadowed in Fig. 2 and in all other generic DSML definitions.

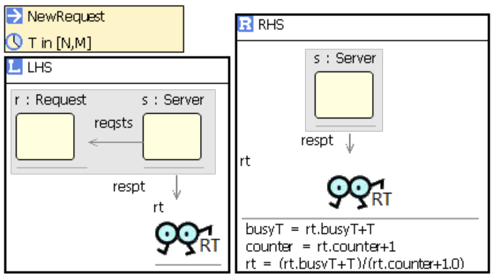
Other performance analysis measures, as response time, resource consumption, etc. may be defined similarly. This is indeed what is proposed in [18]. For instance, *response time* may be defined as the amount of time needed for some request to be processed. In the case of a production line it may be interpreted as the time a machine takes to produce some component, or the time a router takes to deliver a package through a specific connection for a network system. A DSML defining response time observers is shown in Fig. 3, where the busyT attribute of a ResponseTime object represents the time taken by a specific Server object to process a request, and with the RespTime transformation rule in Fig. 3(b) modeling the request-processing atomic action.

The processing of a request is not always an atomic action though. We refer to the time taken by non-atomic requests as *cycle time*, and define CycleTimeOb observers to measure them. For us the difference between response time and cycle time is then just a matter of granularity.

To measure cycle time we keep the time at which a request shows up, to be able to calculate the elapsed time when it is completed. If we are to calculate the mean cycle time, we need to update such value by keeping the number of processed requests. Fig. 4(a) shows the metamodel of the Mean Cycle Time observer DSML. As for the

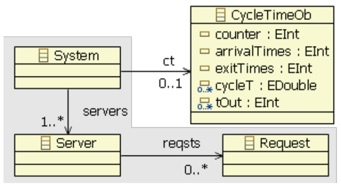


(a) Abstract Syntax

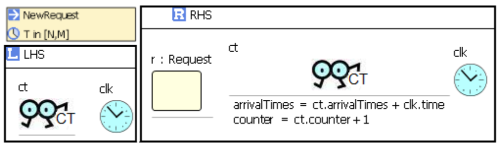


(b) Behavior: New request

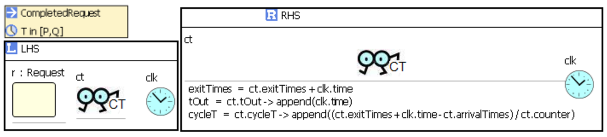
Fig. 3. Response Time observer DSML definition.



(a) Abstract Syntax



(b) Behavior: New request



CompletedRequest
T in [P,Q]
LHS
r : Request
ct
CT
clk
RHS
ct
CT
clk
exitTimes = ct.exitTimes + clk.time
tOut = ct.tOut -> append(clk.time)
cycleT = ct.cycleT -> append((ct.exitTimes + clk.time - ct.arrivalTimes) / ct.counter)

(c) Behavior: Completed request

Fig. 4. Mean Cycle Time observer DSML definition.

throughput observer in Fig. 2, a CycleTimeOb will be associated to a generic System object, and will keep information on the number of attended requests (counter), the sum of their arrival and exit times (arrivalTimes and exitTimes), and the calculated mean cycle time (cycleT). The behavior of this DSML is defined by two transformation rules: NewRequest, which models the inception of a request, depicted in Fig. 4(b), and CompletedRequest, which models the ending of a request, depicted in Fig. 4(c). tOut and cycleT record the sequences of times at which requests are completed and cycle times for each of the requests, respectively.

3 An Airport Check-in System

As case study for our approach, we present the model of the check-in system of an airport. It is basically a queue system, where we model the arrival, waiting and check-in of passengers for a flight. A previous version of this case was presented in [2].

The check-in process is open for 2 hours, and the passengers arrival follows a Gaussian distribution (bell-shaped curve), where the major number of passengers arrives one hour before the closing. We assume there is a fixed number of check-in desks, 5, and we follow the Anglo-Saxon queue model. Thus, there is a single queue and a dispatcher. When a passenger arrives at the airport, the dispatcher forwards him/her to an available check-in desk. If there is none available, the passenger waits in the dispatcher's queue. Furthermore, in order to keep the cost of the system low, the number of open desks will be minimal at all times. One single desk is open at the beginning, being the others opened or closed depending on the number of passengers in the dispatcher's queue. We assume that the time spent in opening/closing a desk is fixed.

To define the check-in system DSML, we define its abstract and concrete syntaxes, and a semantics for it. The metamodel of our system is shown in Fig. 5. An Airport is composed of Dispatchers and CheckInDesks. The former have associated one or more instances of the latter. CheckInDesks may be open or close, according to their open attribute. They have another attribute, serviceTime, that gives the average time in doing the check-in of a passenger. Attributes thresholdMin and thresholdMax in a Dispatcher are used to decide when queues are to be opened/closed. When the length of a Dispatcher's queue goes above the thresholdMax, a new desk is opened, provided there is at least a closed desk. When the length of the queue goes below the thresholdMin, an open desk is closed, provided there are more than one open desks.

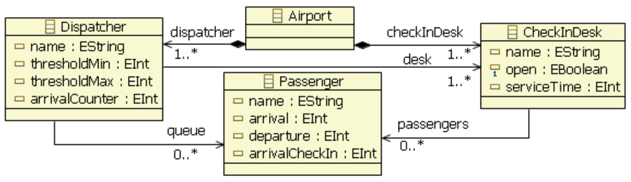
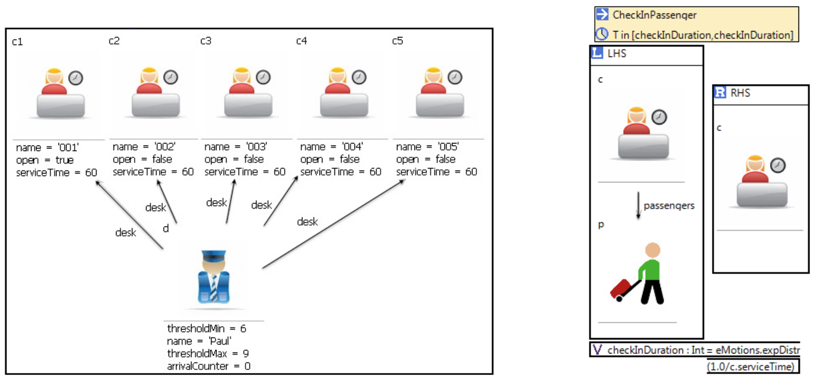


Fig. 5. Airport Metamodel

We can see the concrete syntax chosen for our case study in the initial model of the system shown in Fig. 6(a). Then, we specify its semantics by means of a set of in-place transformation rules. We have transformation rules for each of the actions that may happen in the system. Thus, we have rules modeling the arrival of a new passenger, the assignment of a check-in desk, the opening/closing of a check-in desk, etc. Rule CheckInPassenger, that models the check-in of a passenger, is shown in Fig. 6(b). The duration of the rule follows an exponential distribution with the desk's serviceTime as mean. The whole set of actions is available at [1].

4 Merge for Performance Analysis

In this section we apply the set of generic observers defined in Sect. 2 to the case study presented in Sect. 3. It consists of weaving the metamodels of the observers, one by one, with the metamodel of our system (Fig. 5), and the generic rules in the observers with specific rules in our case study. We first present the weaves of the metamodels



(a) Initial model (b) CheckInPassenger

Fig. 6. Initial rule and sample rule for our case study.

and then of the behavioral rules. Finally, we present some performance results obtained after simulating the resulting woven specifications.

4.1 Weaving Metamodels

To merge the metamodel of the throughput observer (Fig. 2(a)) and the metamodel of the system (Fig. 5), we must specify the correspondences between their classes, attributes, and references (see Sect. 5.1). Here, the generic class System corresponds with Airport, Server with CheckInDesk, and Request with Passenger. Regarding references, servers matches with checkInDesk, and reqsts with passengers. The result of the binding of these metamodels is the inclusion of the ThroughputOb class and thp reference in the metamodel of the system (Fig. 7). We can see that the observer gets associated with the Airport class, which is the class representing the system (it contains the remaining classes). It means that this observer is defined to measure the throughput of the system as a whole.

As for the metamodel of the observer for the response time (Fig. 3(a)), the correspondences are the following. Server class matches with CheckInDesk, and Request with Passenger. Reference reqsts corresponds with passengers. The result of this binding is the inclusion of the ResponseTime class and respt reference in the metamodel of the system (Fig. 7). Notice that the new class gets associated with CheckInDesk class, meaning that, for this specific system, the observer dealing with the generic concept of response time is going to monitor the specific concept of service time of the airport's check-in desks.

Finally, the binding for the metamodel of the mean cycle time (Fig. 4(a)) has to be defined. Like in the throughput observer, System class matches with Airport, Server with CheckInDesk, and Request with Passenger. As for the references, servers corresponds with checkInDesk, and reqsts with passengers. The outcome of this binding is the inclusion of MeanCycleTime class and mct reference in the metamodel of the

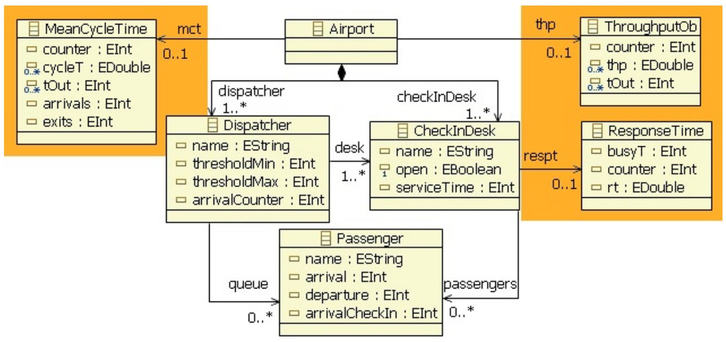


Fig. 7. Woven Metamodel

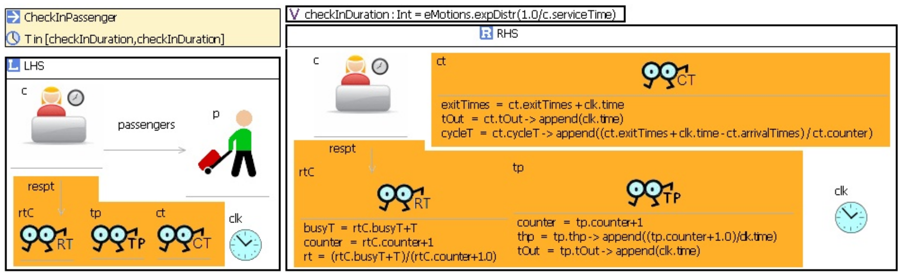

CheckInPassenger
T in [checkInDuration,checkInDuration]
checkInDuration : Int = eMotions.expDistr(1.0/c.serviceTime)
LHS
c
passengers
p
respt
rtC
tp
ct
clk
RT
TP
CT
RHS
c
ct
CT
exitTimes = ct.exitTimes + clk.time
tOut = ct.tOut -> append(clk.time)
cycleT = ct.cycleT -> append((ct.exitTimes + clk.time - ct.arrivalTimes) / ct.counter)
respt
rtC
RT
busyT = rtC.busyT+T
counter = rtC.counter+1
rt = (rtC.busyT+T)/(rtC.counter+1.0)
tp
TP
counter = tp.counter+1
thp = tp.thp -> append((tp.counter+1.0)/clk.time)
tOut = tp.tOut -> append(clk.time)
clk

Fig. 8. Woven CheckInPassenger Rule

system. In this case, the new observer is added to monitor the mean cycle time in the system.

4.2 Weaving Behavioral Rules

If we want to measure the throughput of passengers attended by the check-in desks, we need to bind the throughput observer update rule (Fig. 2(c)) with the rule in which passengers are attended (Fig. 6(b)). Object r in the LHS of ThroughputRule matches with p in CheckInPassenger, and s matches with c both in LHS and RHS. reqsts link matches with passengers. The result of the weaving is the inclusion of the throughput observer (tp) in the CheckInPassenger rule (Fig. 8).

Regarding the observer for the mean cycle time, its behavioral model consists of two rules. CycleTimeArrival is bound with NewPassenger rule, while CycleTimeExit is bound with CheckInPassenger rule. In both bindings, object r corresponds with object p. The result of the second binding is the inclusion of the MeanCycleTime observer in the CheckInPassenger rule, as shown in Fig. 8. The result of the other binding is not shown due to space limitations (please, see [1]).

Finally, the observer of the response time is going to measure, in this specific example, the service time of the check-in desks. For this reason, RespTime rule (Fig. 3(b)) is to be bound with CheckInPassenger rule. The correspondences are as before, s corresponds with c, r with p, and respt with passengers. The result of the weaving is the inclusion of the response time observer in the CheckInPassenger rule (Fig. 8). As we can see, the observer gets associated with the check-in desk.

All the observer objects need to be added in the system from the beginning. This means including them in the initial model shown in Fig. 6(a). In fact, the observer languages contain a rule where each observer is created. These rules have a binding with the rule where the initial model is created, so that the observers are included in the system from the beginning (this is, they appear in the initial model, but we do not show it for space reasons). The complete specification of this system, as well as the weavings, is available from [1].

4.3 Performance Measures

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. The *e-Motions* environment not only provides an editor for writing the visual specifications, but also implements their transformation, using ATL [12], into the corresponding formal specifications in Maude. This approach enables the use of Maude's facilities and tools for executing and analyzing the system specifications once they are expressed in Maude. 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. An 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. In our case, the observers added to the system allow us to analyze the throughput and mean cycle time in the system, as well as the service time of the check-in desks.

Fig. 9 shows the results of the simulations. In the X axe we have the time, expressed in seconds. We are, consequently, showing the performance results during the system's lifetime, not only the value at the end of the simulation. We observe that the throughput at the beginning is quite variable. This is normal since the data is still not very representative (very few passengers have arrived at the airport and realized the check-in). Same thing happens with the mean cycle time. Then, both start to grow. The growth in the mean cycle time is due to congestion of passengers, since the more passengers in the airport, the longer they have to wait, and the higher the mean cycle time. Regarding the throughput, if there are more passengers in the system, there are more passengers in the dispatcher's queue and more desks are opened, so that more passengers complete their check-ins at the same time, what increases the throughput. The fact that passengers arrive at the airport according to a bell-shaped curve, as explained in Sect. 3, is captured by the number of check-in desks open, which is higher in the middle of the system's life-time. It is also captured by the last chart, which presents the number of passengers in the airport along time (the simulation is performed with 240 passengers).

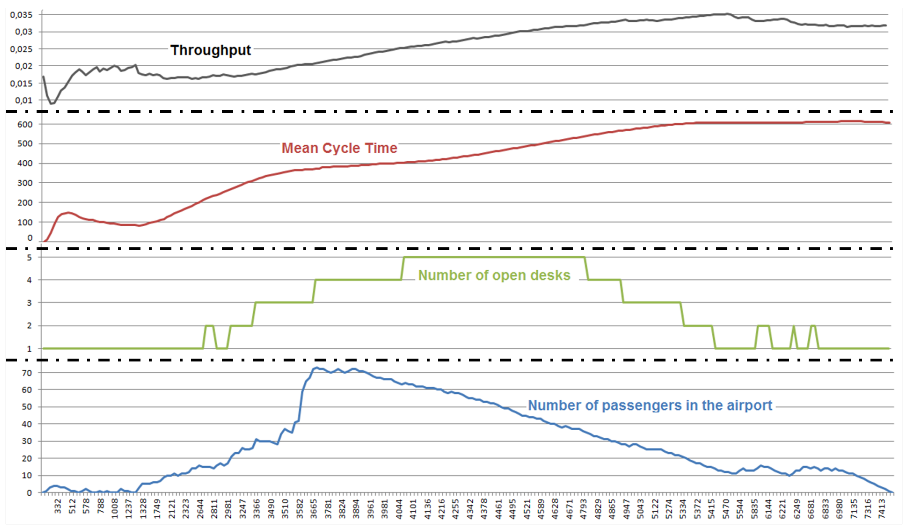


Fig. 9. Performance Results

5 Merging by Model Transformation

For the implementation of our prototype we have used ATL [12], a hybrid model transformation DSL that provides declarative and imperative constructs. ATL transformations are unidirectional, operate on read-only source models, and produce write-only target models. That is, during the execution of a transformation, source models may be navigated, but changes are not allowed, and target models cannot be navigated.

5.1 Correspondences Specification

We have seen in Sect. 1 how to merge models we need to provide bindings between them. In Sect. 4 we have seen how to merge a parameterized observer model M_{Obs} with a DSML describing a system M_{DSL} to be analyzed, we just need to provide a *binding*, provided by a set of correspondences or matches between the elements in the parameter part of M_{Obs} and those in M_{DSL}. Specifically, the bindings between MM_{DSL} and MM_{Obs}, and between Rls_{DSL} and Rls_{Obs}, are given by a model that conforms to the correspondences metamodel shown in Fig. 10.

In a model conforming to the Correspondences metamodel, we have one object of type MMMatching for each pair of metamodels that we want to weave. Objects of type MMMatching contain as many classes (instances of type ClassMatching) as correspondences between classes in both metamodels exist. Each object of type ClassMatching stores the names of the classes in both metamodels that correspond. Regarding the objects of type RefMatching, contained in the refs reference from MMMatching, they store the matchings between references in both metamodels. Attributes obClassName and DSLClassName keep the names of the source classes, while obRefName and DSLRefName contain the names of the references.

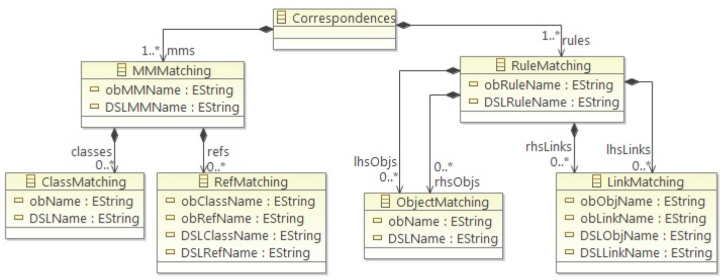


Fig. 10. Correspondences Metamodel

Regarding the binding between rules, there is an object of type RuleMatching for each pair of rules to weave. Objects of types ObjectMatching and LinkMatching contain the correspondences between objects and links, respectively, in the rules. Specifically, our correspondence models differentiate between the bindings established between left- and right-hand side in rules. In our behavioral rules described within *e-Motions*, which conform to the Behavior metamodel (presented in [14]), the objects representing instances of classes are of type Object and they are identified by their id attribute, and the links between them are of type Link, identified by their name, input and output objects. Similar to the binding between metamodels, objects of type ObjectMatching contain the identifier of the objects matching, and instances of LinkMatching store information about matchings between links (they store the identifier of the source classes of the links as well as the name of the links).

5.2 Binding Process

Fig. 11 shows a schema of our merging transformation. It is split in two ATL transformations: WeaveMetamodels.atl, for weaving the metamodels MM_{DSL} and MM_{Obs}, and WeaveBeh.atl, for weaving the behavioral rules Rls_{DSL} and Rls_{Obs}. In this section we sketch both transformations. A detailed documentation of the weaving process is available in [18]. Note that GCS models also take part in the transformations, since they store information about the concrete syntax of both models.

Both transformations work in two stages. First, they copy the original DSL model into the output model. Second, any additions from the observer model are performed according to the binding information from the weaving model. The second transformation builds on the models produced by the first one. In the following we explain the sequential steps in both transformations.

WeaveMetamodels.atl

1. The output metamodel ($MM_{\widehat{DSL}}$) is initialized with the metamodel of the DSL (MM_{DSL}). It is done in order to decorate the output metamodel afterwards, in the third part of the transformation.

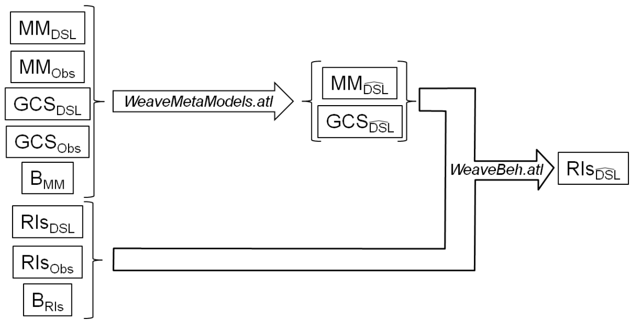


Fig. 11. Transformation Schema.

2. The output concrete syntax ($GCS_{\widehat{DSL}}$) is initialized with the concrete syntax of the DSL (GCS_{DSL}) and of the observer objects (GCS_{Obs}). Observer objects are identified because they do not have any correspondences with any object of (MM_{DSL}), only bindings for the parameter part are given.
3. Classes, references and attributes concerning non-functional properties (observers) are included in the output metamodel ($MM_{\widehat{DSL}}$).

WeaveBeh.atl

1. The output behavioral model ($Rls_{\widehat{DSL}}$) is initialized with all the behavioral rules (and their contained elements) of the DSL (Rls_{DSL}).
2. Observers are included in those rules of $Rls_{\widehat{DSL}}$ that have correspondences with observer rules. This part basically decorates the left- and right-hand sides of these rules by including the objects of the observer rules that do not have any matching.
3. In some cases, there are observer rules that need to be included, as they are, in the output behavioral model. These rules do not have correspondences with any rule in the DSL. This part of the transformation aims at including such rules in $Rls_{\widehat{DSL}}$.
4. Finally, those rules in which there is no clock object but that need the current time elapse get a clock object added.

6 Conclusions and Future Work

In this paper we have presented a novel implementation of a composition operation for domain-specific modeling languages (DSMLs), and we have illustrated its use on a case study describing the addition of observers in the check-in system of an airport. Although our implementation is based on the *e-Motions* language and system, both our approach and formal framework are applicable to any DSML specification whose semantics is based on in-place model transformations.

We have successfully applied our approach, implemented it by means of a correspondences metamodel and two ATL transformations, by weaving three observer languages, two of them containing general observers and the third one defining an individual observer, with the DSML of a check-in airport system. We have also shown how the observers included in the woven DSML are used to obtain metrics of different non-functional properties throughout the lifetime of the system.

Although the approach presented in this paper and applied for the modular definition of observers has been implemented and tested, it is still in a naive and early phase and many things may need to be added, studied and improved. For example, we would like to provide a graphical interface so that the bindings can be defined graphically. We also need to study our approach for a wider variety of observers. In particular, we have to study the case where there is no trivial alignment between the rules of the observers and the systems. Furthermore, it would be ideal if some bindings could be realized automatically, with no human intervention.

Acknowledgements.
We would like to thank Steffen Zschaler and Fernando Orejas for fruitful discussions and previous and on-going collaborations this work is related to. This work is partially funded by Research Projects TIN2011-23795 and TIN2011-15497-E, and by Universidad de Mlaga (Campus de Excelencia Internacional Andaluca Tech).

References

1. Atenea. Airport with a modular definition of observers, 2013. `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/E-motions/AirportModularObservers`.
2. J. M. Bautista, J. Troya, and A. Vallecillo. Diseño y simulación de sistemas de colas con *e-Motions*. In *XVI Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2011.
3. K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson. Semantic anchoring with model transformations. In *Proc. of ECMDA-FA'05*, LNCS 3748. Springer, 2005.
4. T. Clark, A. Evans, and S. Kent. Aspect-oriented metamodelling. *The Computer Journal*, 46(5):566–577, 2003.
5. J. de Lara and E. Guerra. Generic meta-modelling with concepts, templates and mixin layers. In *Proc. of MODELS'10 Part 1*, LNCS 6394: 16–30. Springer, 2010.
6. J. de Lara and H. Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. *J. Visual Lang. Comput.*, 15(3–4):309–330, 2006.
7. D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.
8. F. Durán, F. Orejas, and S. Zschaler. Behaviour protection in modular rule-based system specifications. In *Proc. WADT'12*, LNCS 7841:24-49. Springer, 2013.
9. F. Durán, S. Zschaler, and J. Troya. On the reusable specification of non-functional properties in DSLs. In *Proc. SLE'12*, LNCS 7745:332–351. Springer, 2012.
10. G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In *Proc. of UML'00*, LNCS 1939:323–337. Springer, 2000.
11. Z. Hemel, L. C. L. Kats, D. M. Groenewegen, and E. Visser. Code generation by model transformation: A case study in transformation modularity. *Software and Systems Modelling*, 9(3):375–402, 2010.
12. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: a model transformation tool. *Science of Computer Programming*, 72(1–2):31–39, 2008.
13. R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos, and F. A. Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. *Procs. of ICECSS*, pp. 162–171. IEEE, 2009.

14. J. E. Rivera, F. Durán, and A. Vallecillo. A graphical approach for modeling time-dependent behavior of DSLs. In *Procs. VL/HCC'09*, pp. 51–55. IEEE, 2009.
15. J. E. Rivera, F. Durán, and A. Vallecillo. On the behavioral semantics of real-time domain specific visual languages. In *Procs. WRLA'10*, LNCS 6381:174–190. Springer, 2010.
16. J. E. Rivera, E. Guerra, J. de Lara, and A. Vallecillo. Analyzing rule-based behavioral semantics of visual modeling languages with Maude. In *Proc. of SLE'08*, LNCS 5452:54–73. Springer, 2008.
17. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume I: Foundations*. World Scientific, 1997.
18. J. Troya. *On the Model-Driven Performance and Reliability Analysis of Dynamic Systems*. PhD thesis, Universidad de Málaga, 2013. http://www.lcc.uma.es/∼jtc/TroyaThesis.pdf.
19. J. Troya, J. E. Rivera, and A. Vallecillo. Simulating domain specific visual models by observation. In *Proc. SpringSim'10*, pp. 128:1–128:8. ACM, 2010.
20. J. Troya, A. Vallecillo, F. Durán, and S. Zschaler. Model-driven performance analysis of rule-based domain specific visual models. *Information and Software Technology*, 55(1):88 – 110, 2013.