On the Specification of Non-functional Properties of Systems by Observation

Javier Troya, José E. Rivera, and Antonio Vallecillo

GISUM/Atenea Research Group. Universidad de Málaga, Spain
{javiertc,rivera,av}@lcc.uma.es

Abstract. Domain specific languages play a cornerstone role in Model-Driven Engineering (MDE) for representing models and metamodels. So far, most of the MDE community efforts have focused on the specification of the functional properties of systems. However, the correct and complete specification of some of their non-functional properties is critical in many important distributed application domains, such as embedded systems, multimedia applications or e-commerce services. In this paper we present an approach to specify QoS requirements, based on the observation of the system actions and of the state of its objects. We show how this approach can be used to extend languages which specify behavior in terms of rules, and how QoS characteristics can be easily expressed and reused across models. We show as well how this approach enables the specification of other important properties of systems, such as automatic reconfiguration of the system when some of the QoS properties change.

1 Introduction

Domain specific languages (DSLs) play a cornerstone role in Model-Driven Engineering (MDE) for representing models and metamodels. The Software Engineering community's efforts have been progressively evolving from the specification of the structural aspects of a system to modeling its dynamics, a current hot topic in MDE. Thus, a whole set of proposals already exist for modeling the structure and behavior of a system. Their goal is not only to generate code, but also to conduct different kinds of analysis on the system being modeled including, e.g., simulation, animation or model checking.

The correct and complete specification of a system also includes other aspects. 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, such as embedded systems, multimedia applications or e-commerce services and applications.

In order to fill this gap, in the last few years the research has faced the challenge of defining quantitative models for non-functional specification and validation from software artifacts [2]. 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 [3]. However, 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). These requirements state how the system should behave. Examples of these approaches include the majority of the UML Profiles for annotating UML models with QoS information, e.g., [4,5,6].

In this paper we present an alternative approach to specify QoS requirements, based on the observation of the system actions and of the state of its constituent objects. We show how this approach can be used to extend DSLs which specify behavior in terms of rules (that describe the evolution of the modeled artifacts along some time model), and how QoS characteristics can be easily expressed and reused across models. In particular, we focus on performance and reliability characteristics.

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

Finally, the approach has an additional benefit when it comes to generate the system code. The "observers" that monitor the system behavior and compute the QoS metrics can be used to generate the instrumentation code that monitors the actual behavior of the system, too.

After this introduction, Section 2 briefly describes one proposal for modeling the functional aspects of systems, which also contemplates time-dependent behavior. It presents an 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 QoS properties. In particular, we show how the throughput, jitter and mean-time between failures of the system are specified. Then, Section 4 shows how the specifications produced can be used to analyse the system, to specify self-adaptation mechanisms for alternative behaviors of the system, and to generate probes. Finally, Section 5 compares our work with other related proposals and Section 6 draws some conclusions.

2 Specifying Functional Properties

One way of specifying the dynamic behavior of a DSL is by describing the evolution of the modeled artifacts along some time model. In MDE, this can be done using model transformations supporting in-place update [7]. The behavior of the DSL is then specified in terms of the permitted actions, which are in turn modeled by the transformation rules.

There are several approaches that propose in-place model transformations to deal with the behavior of a DSL, from textual to graphical (see [8] for a brief survey). This approach provides a very intuitive way to specify behavioral semantics, close to the language of the domain expert and the right level of abstraction [9]. In-place transformations are composed of a set of rules, each of which represents a possible *action* of the system. These rules are of the form $l : [\mathrm{NAC}]^* \times \mathrm{LHS} \rightarrow \mathrm{RHS}$, where l is the rule's label (its name); and LHS (left-hand side), RHS (right-hand side), and NAC (negative application conditions) are model patterns that represent certain (sub-)states of the system. The LHS and NAC patterns express the precondition for the rule to be applied, whereas the RHS one represents its postcondition, i.e., the effect of the corresponding

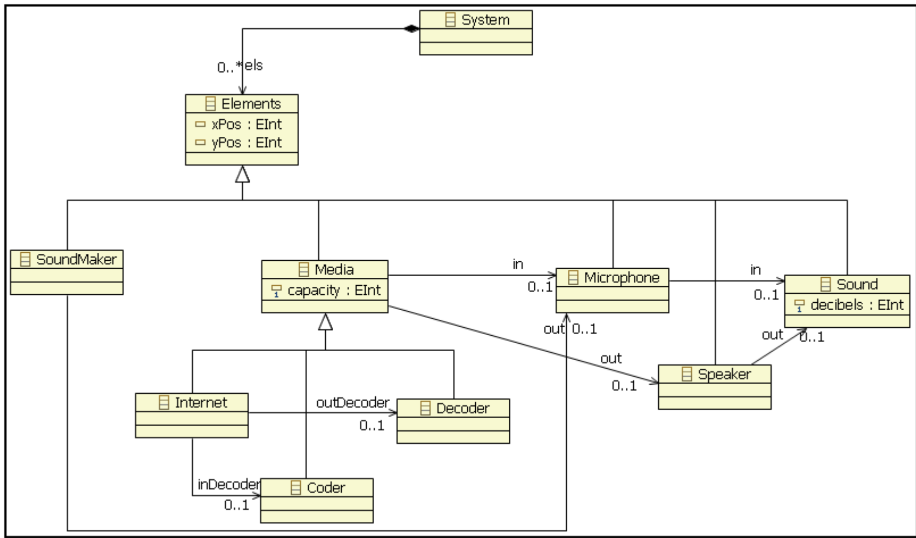

Fig. 1. Sound System metamodel

action. Thus, a rule can be applied, i.e., triggered, if an occurrence (or match) of the LHS is found in the model and none of its NAC patterns occurs. Generally, if several matches are found, one of them is non-deterministically selected and applied, producing a new model where the match is substituted by the appropriate instantiation of its RHS pattern (the rule's *realization*). The model transformation proceeds by applying the rules in a non-deterministic order, until none is applicable — although this behavior can be usually modified by some execution control mechanism [10].

In [11] we also showed how time-related attributes can be added to rules to represent features like duration, periodicity, etc. Moreover, we also included the explicit representation of *action executions*, which describe actions currently executing.

We have two types of rules to specify time-dependent behavior, namely, *atomic* and *ongoing* rules. Atomic rules represent atomic actions, with a specific duration. They can be cancelled, but cannot be interrupted. Ongoing rules represent interruptible continuous actions. Atomic rules can be periodic, and atomic and ongoing rules can be scheduled, or be given an execution interval, by rules' lower and upper bounds.

A special kind of object, named Clock, represents the current global time elapse. This allows designers to use it in their timed rules.

A running example

Let us show a very simple example to illustrate how the behavior of a system can be modeled using our visual language. The system models the transmission of a sound via a media, the Internet for instance. It consists of a *soundmaker* (e.g., a person) who, periodically, transmits a sound to a microphone. This one is connected to a media (the Internet), which transports the sound to a speaker. Finally, when the sound reaches the

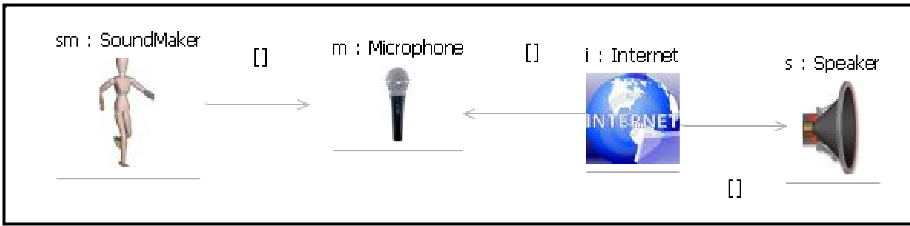

Fig. 2. Initial model of the system

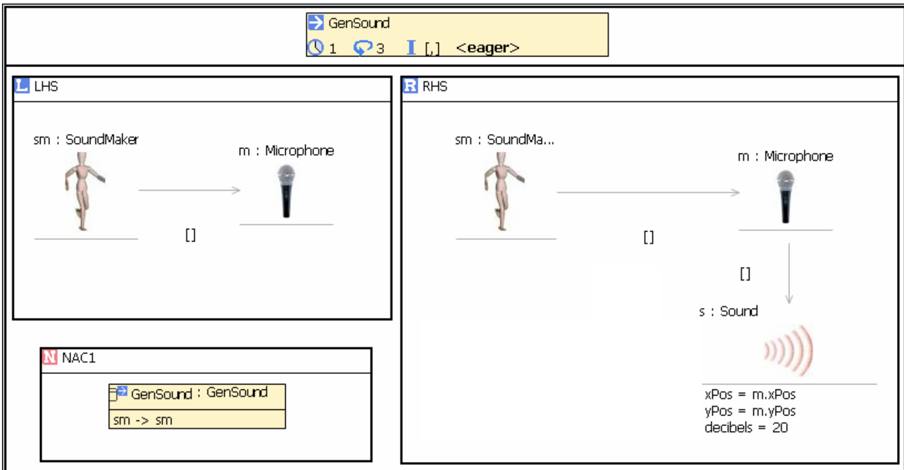

Fig. 3. GenSound rule

speaker, it is amplified. Fig. 1 shows the metamodel of the system. For the time being, Coder and Decoder metaclasses can be ignored; they will be mentioned in Sect. 4. The initial state of the system is shown in Fig. 2. The position of objects in the initial model has been omitted for simplicity reasons.

In addition to the metamodel and the initial model of our system, we also need to describe the behavior of the system. This is done in terms of the possible actions, which in our proposal are represented by in-place transformation rules.

The *GenSound* periodic rule (Fig. 3) makes the *soundmaker* emit a sound every 3 time units. This rule makes use of an *action execution* element. This way, we explicitly forbid the execution of the rule (see the *NAC1* pattern) if the same *soundmaker* is emitting another sound. This *action execution* states that the element *sm* (the *soundmaker*) is participating in an execution of the rule *GenSound*, so the rule cannot be applied if there is a match of this NAC. In the RHS, we can see that the sound is now in the microphone, so it acquires its position. The sound has 20 decibels. The duration of the action modeled by this rule is one time unit.

Fig. 4 shows the rule which makes the sound reach the speaker. As we can see in the LHS pattern, this rule is executed when the microphone has a sound. This microphone

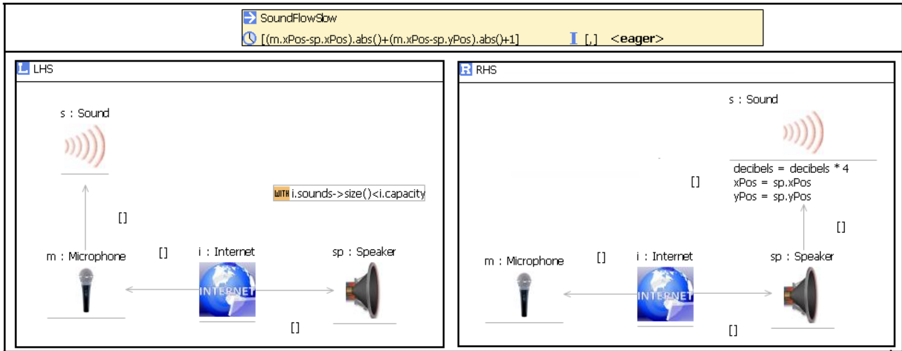

Fig. 4. SoundFlowSlow rule

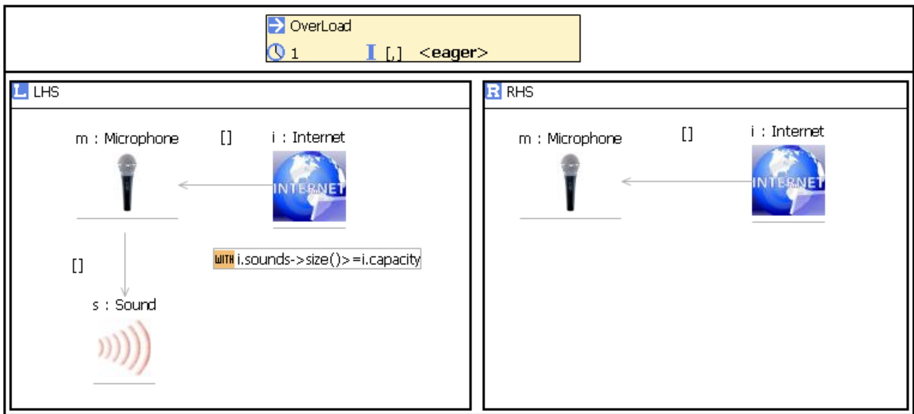

Fig. 5. OverLoad rule

has to be connected to a media which should be, in turn, connected to a speaker. The number of sounds that the media is currently transporting has to be lower than its capacity. When the rule is realized, the sound reaches the speaker (RHS pattern). When this happens, the sound decibels are quadruplicated and the position of the sound is changed to be the same as the position of the speaker. The time the media consumes in transporting the sound (i.e., the time consumed by the rule) is given by the Manhattan distance between the microphone and the speaker.

Fig. 5 shows the *OverLoad* rule. It is triggered when the *soundmaker* has produced a sound which is now at the microphone, and the media is already transporting more sounds than its capacity allows. Thus, the sound appearing in the LHS pattern cannot be transported and it is lost (i.e., it is not included in the RHS pattern).

So far, these three rules are enough for modeling the behavior of this simple system. Let us see now how to add QoS information to these specifications about the performance and reliability properties of the system.

3 Specifying QoS Properties by Observation

The correct and complete specification of a system should include the specification and analysis of its non-functional properties. An approach to specify QoS requirements, based on the observation of the system actions and of the state of its constituent objects, is presented in this section. In particular, we introduce three QoS parameters which have to be updated with the passing of time.

- Throughput (th): The amount of work that can be performed or the amount of output that can be produced by a system or component in a given period of time. Throughput is defined as $th = n/t$, where n is the amount of work the system has performed and t is the time the system has been working. The work the system performs depends on the kind of system we are dealing with. In our example, it is the number of successful packets transmitted.
- Mean time between failures (MTBF): the arithmetic mean (average) time between failures of a system. MTBF $= t/f$, where t is the time the system has been working and f is the number of failures of the system.
- Jitter (j): in the context of voice over IP, it is defined as a statistical variance of the RTP data package inter-arrival time [12]. RTP (Real Transport Protocol) provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulation data, over multicast or unicast network services. To estimate the jitter after we receive an i-th packet, we calculate the change of inter-arrival time, divide it by 16 to reduce noise, and add it to the previous jitter value. The division by 16 helps to reduce the influence of large random changes. The formula used is: $j(i) = j(i-1) + (\mid D(i-1, i) \mid - j(i-1))/16$, where $j(i)$ is the current jitter value and $j(i-1)$ is the previous jitter value. In this jitter estimator formula, the value $D(i, j)$ is the difference of relative transit times for the two packets. The difference is computed as $D(i, j) = (R_j - R_i) - (S_j - S_i)$, where S_j ($Send_j$) is the time the package j appears in the system (that is, the time at which it is sent by the transmitter) and R_j ($Receive_j$) is the time the package j leaves the system because it has been processed (that is, the time at which it is received by the receiver).

3.1 Defining Observers

To calculate the value of these QoS properties we propose the use of *observers*. An observer is an object whose objective is to monitor the value of one of these parameters. We identify two kinds of observers, depending on whether they monitor specific objects or the state and behavior system as a whole. In the first case, observers are created with the objects they monitor, and destroyed with them. In the second case, observers are present for the whole life of the system. As a first approach, we have created a metamodel of observers (Fig. 6) with four observers, which inherit from an Observer class. Each of them has a specific purpose:

- ThroughPutOb. Calculates the current value of throughput in the system, which is stored in its variable tp. Attribute packages counts the number of successful packages, i.e., those that have reached their destinations.

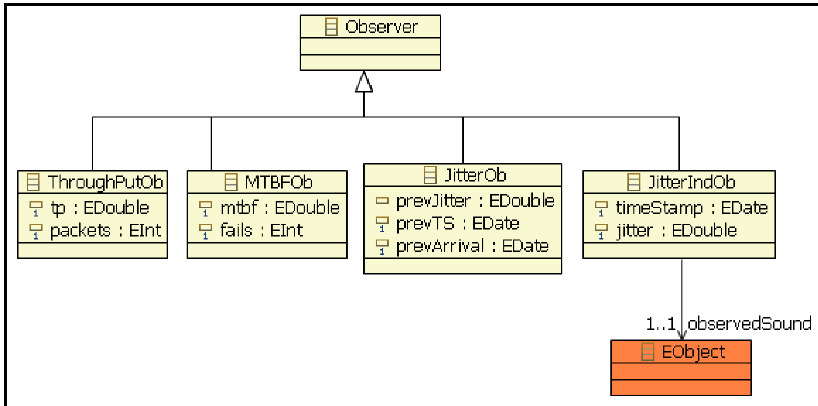

Fig. 6. Observers Metamodel

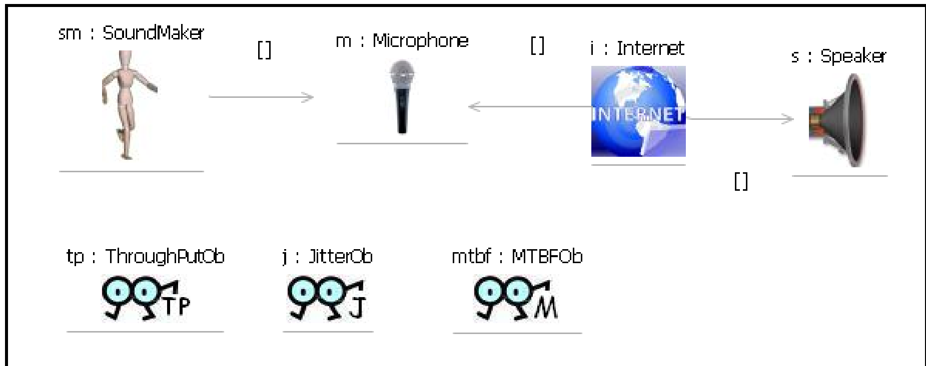

Fig. 7. Initial model of the system with observers

– MTBFOb. Calculates the MTBF of the system (mtbf attribute). Attribute fails stores the number of lost packages.
– JitterOb. This is a general observer that is used to compute the jitter of the system. It has three attributes: prevJitter contains the latest jitter value, prevTS stores the time the latest package appeared in the system, and prevArrival stores the time the latest package left the system.
– JitterIndOb. This observer has a reference to an EObject, which is at the top of the class hierarchy. In this way, it can be associated to any of the elements of the *Sound System* metamodel. In our example, this observer is associated to individual sounds. It computes the jitter when its associated sound reaches its destination.

The idea for including observers in our system is to combine the two metamodels (Fig. 1 and 6) to be able to use the defined observers in our sound system language. In fact, since our modeling tool *e-Motions* [11] allows users to merge several metamodels in the definition of a DSVL behavior, we can define the *Observers metamodel* in a

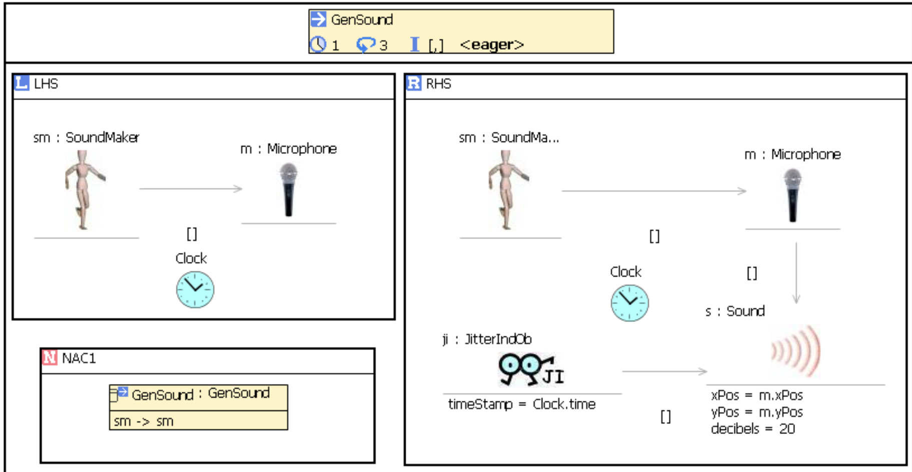

Fig. 8. GenSound with observers

non-intrusive way, i.e., we do not need to modify the system metamodel to add observers in their rules. Furthermore, this approach also enables the reuse of observers across different DSVLs.

Thus, we have added to the initial model depicted in Fig. 2 a set of initial observers (see Fig. 7). They will be present throughout the execution of the system and their values will be changing depending on the state of the system. The other elements are the same as shown in Fig. 2.

3.2 Describing the Behavior of the Observers

Once the observers have been added to a system, we can define their behavior using the rules described in Section 2. In this section we show how the throughput, jitter and mean time between failures can be updated by means of the rules that specify the behavior of the system.

In Fig. 8, an observer has been added to action *GenSound*. Now, the rule associates a JitterIndOb observer to a newly generated sound. The time this sound appears in the system is stored in attribute timeStamp. In Fig. 9, the MTBFOb observer has been added to action *OverLoad*, to be able to update its attribute fails every time a sound disappears.

Fig. 10 shows how the value of the jitter is calculated when a sound reaches its destination and how the number of successful packages is updated. As we can see, a JitterIndOb observer is associated to the sound. Observers ThroughPutOb and JitterOb appear in the LHS part of the rule. When the sound reaches the speaker (RHS part), the number of successful packages of the system is increased. The jitter attribute of the JitterIndOb associated to the sound is computed, using the value of its timeStamp attribute and the three attributes of the JitterOb observer.

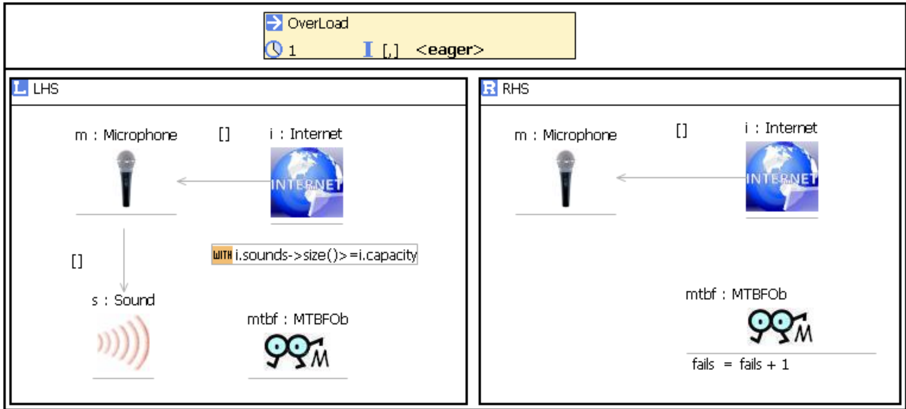

Fig. 9. OverLoad with observers

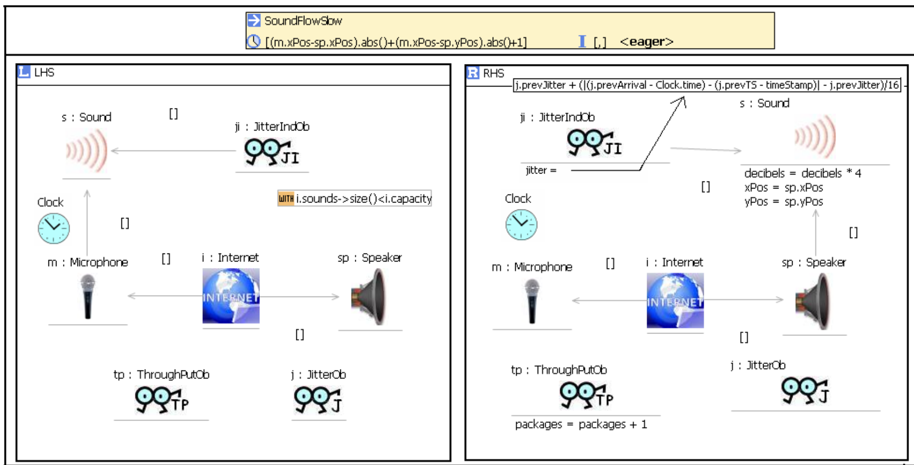

Fig. 10. SoundFlowSlow with observers

Fig. 11 shows an atomic rule which has been added to this new system with observers. It is triggered when a sound reaches the speaker, i.e., after the *SoundFlowSlow* rule has been executed. In the LHS part, the JitterIndOb associated with the sound contains the current jitter. This rule updates the values of the attributes of JitterOb in the RHS part and makes the sound disappear (because it has reached its destination).

Fig. 12 shows an ongoing rule, required to calculate the throughput and MTBF of the global system. It is an ongoing rule and therefore it progresses with time. In this way both observers always store correct and up-to-date QoS values at any moment in time.

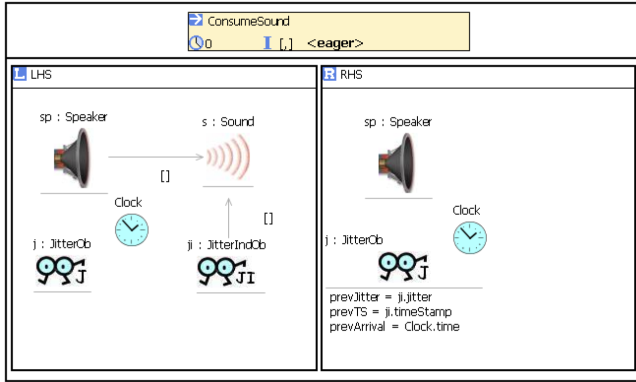

Fig. 11. New rule: ConsumeSound

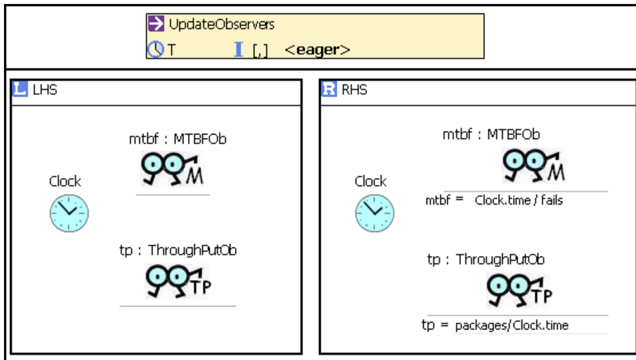

Fig. 12. New rule: UpdateObservers

4 Making Use of the Observers

Apart from computing the QoS values for the system, observers can be very useful for defining alternative behaviors of the system, depending on the QoS 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.

Fig. 13 shows a rule where the media that transmits the sound changes when the throughput value is less than a threshold value (1.5 in this case). In particular, we add one coder and one decoder to the system. Thus, when there is a match of the LHS part, the system self-adapts to accomplish the requirements.

Fig. 14 shows the behavior of the sound flow with the presence of coders and decoders. It is very similar to the *SoundFlowSlow* rule. The main difference is the time both rules consume. This new rule, with the coder and decoder added to the system, consumes half the time the other rule does. In this way, the throughput value increases notably, improving the system performance.

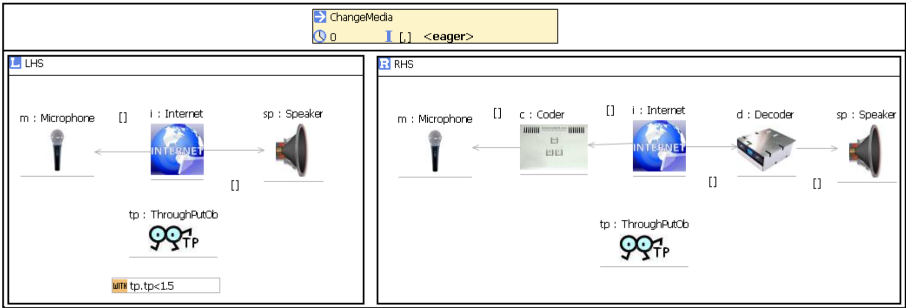

Fig. 13. New rule: ChangeMedia

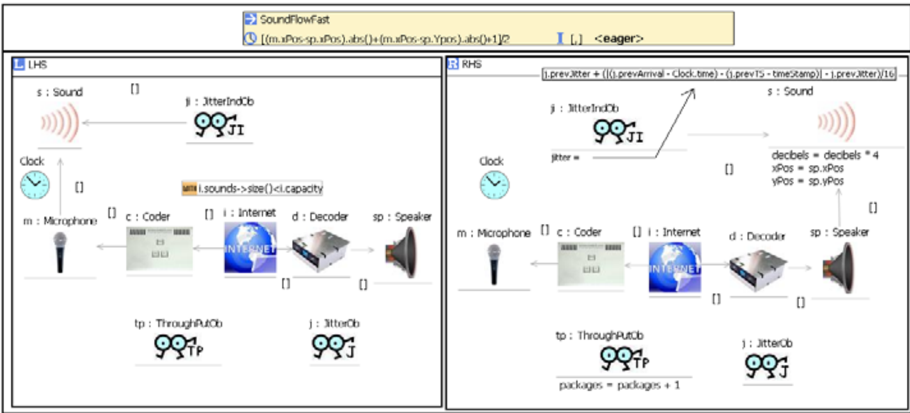

Fig. 14. SoundFlowFast with observers

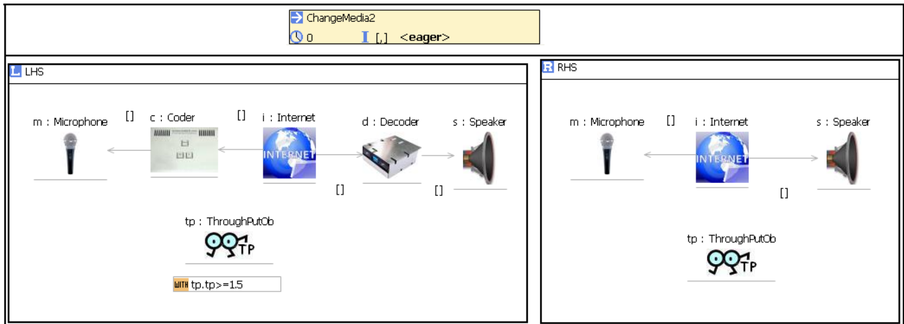

Fig. 15. ChangeMedia2: Restoring the change made by ChangeMedia

Similarly, Fig. 15 shows a rule that specifies the opposite transformation. That is, when the throughput goes above 1.5, the system returns to its original configuration. In this way the configuration of the system can toggle between these two options, self-adapting according to the overall performance.

5 Related Work

Several approaches have proposed a procedure for monitoring and measuring non-functional properties of a system. Some of them are similar to the one presented here, although all of them have a different focus. For example, Liao and Cohen [13] introduced a high level program monitoring and measuring system which supported very high level languages. In [14] and [15] they propose two frameworks for performance measurement. In the first case, Mike et al. designed and implemented *Pinpoint*, a framework for problem determination in Internet service environments. It was implemented on top of the J2EE middleware platform, a network sniffer, and an analyzer based on standard data clustering techniques. In the second one, Matthias Rohr et al. present *Kieker*, which allows continuous monitoring, analysis, and visualization of Java applications. It supports to create Sequence Diagrams, Markov chains, Timing Diagrams, and Component Dependency Graphs from monitoring data. Our approach contains similar characteristics to these frameworks. On the one hand, it can be used to determine problems in systems by looking up the state of the observers. In our example, the sound system changed when the throughput was too low. On the other hand, our approach allows continuous monitoring of the system, as observers are constantly updated.

Compilers supporting aspect-oriented programming (AOP), such as AspectJ [16] and AspectC++ [17], may be considered source-level instrumentation tools. In AOP, it concerns that cross-cut modules are factored out into modular aspects. AOP tools have been used to instrument programs with debugging and monitoring code [18], as well as to instrument programs with code to check temporal invariants [19]. As a shortcoming, AOP techniques can only be applied at well-defined join points and can be used for instrumentation only. With our approach, instead, non-functional parameters can be measured at any time and at any point in the system. In addition, our proposal remains at a very high level of abstraction, without being tied to any programming language or concrete technology platform.

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. For example, the OMG defines different kinds of observers in the MARTE specification [6]. Among them, TimedObservers are conceptual entities that define requirements and predictions for measures defined on an interval between a pair of user-defined observed events. They must be extended to define the measure that they collect (e.g., latency or jitter), and aim at providing a powerful mechanism to annotate and compare timing constraints over UML models against timing predictions provided by analysis tools. In this sense they are similar to our observers. The advantage of incorporating them into DSLs 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.

General frameworks for self-adaptive systems are presented in [20] and [21], featuring inter-related monitoring, analysis and adaptation tiers. Diaconescu et al. [22] add a transparent software layer between components and middleware. This framework aligns with [20] and [21], while specifically targeting enterprise applications based on contextual composition middleware. Our approach presents a way to make systems

self-adaptive as well, although we deal with the monitoring of QoS parameters using observers at high level of abstraction, and again independently from the underlying platform or language.

In both cases we see that our approach could be easily mapped to the ones mentioned here, hence provided platform-independent models that could be transformed into these platform-specific approaches, as we plan to do as part of our future work.

6 Conclusions and Future Work

The correct and complete specification of the non-functional properties of a system is critical in many important distributed application domains. In this paper we have presented a platform independent approach to specify QoS properties and requirements, and shown its use to specify three of them: throughput, jitter and mean time between failures. In particular, we have shown that the use of *observers* that monitor the state and behavior of the system can be very useful to enrich some kinds of high-level behavioral specifications with QoS information.

The QoS parameters calculated by observers can be used for many additional purposes. We have shown how our approach can also serve to easily specify self-adaptive behaviors depending on the values of the system QoS properties. Please also note that our proposal is built on top of the underlying language (*e-Motions* [11] in this case), hence allowing users to make use of all the analysis possibilities available for that environment [10] for free. Another advantage of our proposal is that it can serve to monitor not only the states of the objects of the system, but also their actions. The fact that action executions are first-class citizens of the e-Motions visual language enables their monitorization by our observers.

As part of our future work we would like to define additional observers, with the advantage that once defined they can be re-used across models. For this purpose, we would like to create libraries into *e-Motions* with these observers. This way, designers could merge several metamodels: their original ones and the metamodels with observers provided by the libraries. This makes the designers tasks easier when including QoS properties to their systems. We would also like to study the connection of these specifications with other notations (e.g., SysML or MARTE) so that transformations can be defined between them. In addition, we would also like to explore the automatic instrumentation of the standard code generated by model-transformation approaches, with the aim of being able to generate monitors and probes associated to the code, too.

Acknowledgements. The authors would 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 P07-TIC-03184.

References

1. Troya, J., Rivera, J.E., Vallecillo, A.: On the specification of non-functional properties of systems by observation. In: Proc. of the 2nd International Workshop on Non-Functional Properties for DSMLs (NFPinDSML 2009), Denver, CO. CEUR Workshop Proceedings, vol. 553 (2009), `http://CEUR-WS.org/Vol-553/paper1.pdf`
2. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey 30(5), 295–310 (2004)

3. Cortellessa, V., Marco, A.D., Inverardi, P.: Integrating performance and reliability analysis in a non-functional MDA framework. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 57–71. Springer, Heidelberg (2007)
4. OMG: UML Profile for Schedulability, Performance, and Time Specification. OMG, Needham (MA), USA (2005)
5. OMG: UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. OMG, Needham (MA), USA, ptc/04-09-01 (2004)
6. OMG: A UML Profile for MARTE: Modeling and Analyzing Real-Time and Embedded Systems. OMG, Needham (MA), USA (2008)
7. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA 2003 Workshop on Generative Techniques in the Context of MDA (2003)
8. Rivera, J.E., Guerra, E., de Lara, J., Vallecillo, A.: Analyzing rule-based behavioral semantics of visual modeling languages with Maude. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 54–73. Springer, Heidelberg (2009)
9. de Lara, J., Vangheluwe, H.: Translating model simulators to analysis models. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 77–92. Springer, Heidelberg (2008)
10. Rivera, J.E., Vallecillo, A., Durán, F.: Formal specification and analysis of Domain Specific Languages using Maude. Simulation: Transactions of the Society for Modeling and Simulation International 85(11/12), 778–792 (2009)
11. Rivera, J.E., Durán, F., Vallecillo, A.: A graphical approach for modeling time-dependent behavior of DSLs. In: Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2009), Corvallis, Oregon (US). IEEE Computer Society, Los Alamitos (2009)
12. Toncar, V.: VoIP Basics: About Jitter (2007), http://toncar.cz/Tutorials/VoIP/VoIP_Basics_Jitter.html
13. Liao, Y., Cohen, D.: A specification approach to high level program monitoring and measuring 18(11), 969–978 (1992)
14. Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.: Pintpoint: Problem determination in large, dynamic internet services. In: Proceedings of the 2002 International Conference on Dependable Systems and Networks, pp. 595–604. IEEE Computer Society, Washington (2002)
15. Rohr, M., van Hoorn, A., Matevska, J., Sommer, N., Stoever, L., Giesecke, S., Hasselbring, W.: Kieker: Continuous monitoring and on demand visualization of Java software behavior. In: Proceedings of the IASTED International Conference on Software Engineering 2008, pp. 80–85. ACTA Press (2008)
16. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
17. Spinczyk, O., Gal, A., Schroder-Preikschat, W.: AspectC++: an aspect-oriented extension to the C++ programming language. In: Proc. of 40th International Conference on Tools Pacific, pp. 53–60 (2002)
18. Mahrenholz, D., Spinczyk, O., Schroeder-Preikschat, W.: Program instrumentation for debugging and monitoring with Aspectc++. In: Proc. of ISORC 2002, pp. 249–256 (2002)
19. Gibbs, T., Malloy, B.: Weaving aspects into C++ applications for validation of temporal invariants. In: Proc. of SMR 2003 (2003)
20. Garlan, D., Cheng, S., Schmerl, B.: Increasing system dependability through architecture-based self-repair. In: Architecting Dependable Systems. Springer, Heidelberg (2003)
21. Oreizy, P., Gorlick, M., Taylor, R., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., Wolf, A.: An architecture-based approach to self-adaptive software. In: IEEE Intelligent Systems (1999)
22. Diaconescu, A., Mos, A., Murphey, J.: Automatic performance management in component based systems. In: Proc. of ICAC 2004, pp. 214–221 (2004)