# 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 naturally 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 natural specification of other important properties of systems, such as automatic reconfiguration of the system when some of the QoS properties change, and also makes the specifications amenable to simulation and other kinds of formal analysis.

## 1 Introduction

Domain specific languages (DSLs) play a cornerstone role in Model-Driven Engineering (MDE) for representing models and metamodels. The Software Engineering community 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 [1]. 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 [2]. However, most

of these proposals specify QoS characteristics and constraints using a *prescriptive* approach, i.e., they annotate the models with a a set of requirements on the behavior of the system (response time, throughput, etc). These requirements state how the system should behave, constraining its behavior. Examples of these approaches include the majority of the UML Profiles for annotating UML models with QoS information, e.g., [3,4,5].

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 naturally 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 natural 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. In addition, our proposal makes the specifications amenable to simulation and other kinds of formal analysis that deal with QoS constraints.

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 naturally 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 shows 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 naturally done using model transformations supporting in-place update [6]. 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 [7] for a comprehensive survey). This approach provides a very intuitive and natural way to specify behavioral semantics, close to the language of the domain expert and the right level of abstraction [8]. 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 $l : [\text{NAC}]^* \times \text{LHS} \rightarrow \text{RHS}$, 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 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 [9].

In [10] 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 the Clock in their timed rules to get the current time (using its attribute time) to model, e.g., time stamps, etc.

**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 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. We explicitly forbid the execution of the rule (see the *NAC1* pattern) if the same *soundmaker* is emitting another sound by using an *action execution* element. This action execution element states that the element *sm* (the *soundmaker*) is participating in an execution of the rule *GenSound*. 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 has to be connected to a media which is, in turn, connected to a speaker. The number of sounds that the media is currently transporting has to be lower than its capacity. When

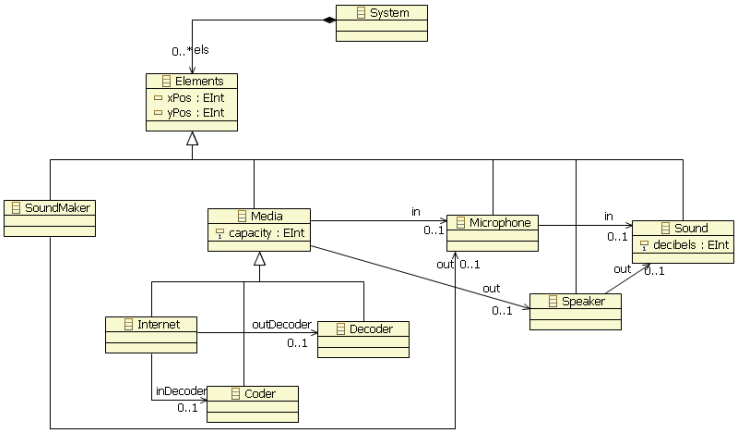


**Fig. 1.** Sound System metamodel.

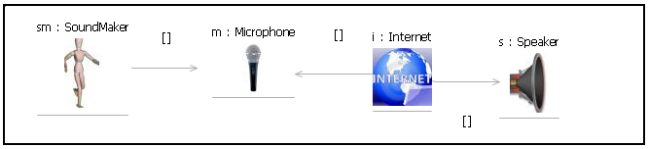


**Fig. 2.** Initial model of the system in its concrete syntax

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.

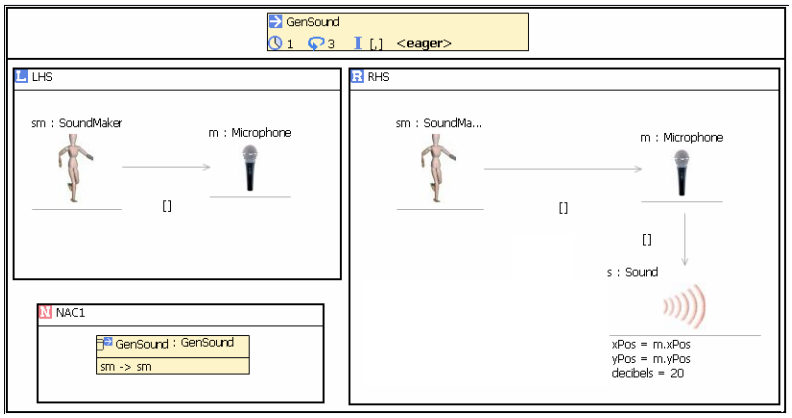


**Fig. 3.** GenSound rule

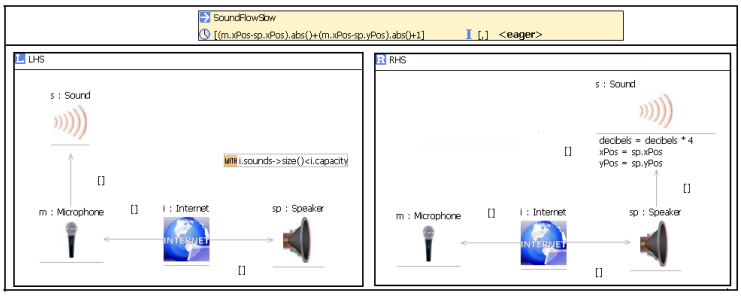


**Fig. 4.** SoundFlowSlow rule

– 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 work the system has performed and $t$ is the time the system has been working.

– 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 [11]. The formula used for estimating jitter takes into account the current and previous packages: $j(i) = j(i-1) + (\mid D(i-1,i) \mid -j(i-1))/16$, where $j(i)$ is the current jitter value and $j(i-1)$ is the previous jitter value. $D(i,j)$ is calculated by $D(i,j) = (R_j - R_i) - (S_j - S_i)$, where $S_j$ is the time the package $j$ appears in the system and $R_j$ is the time the package $j$ leaves the system because it has been processed.

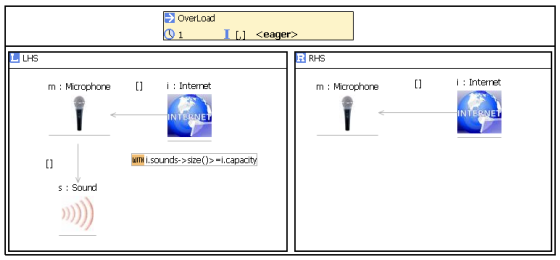


**Fig. 5.** OverLoad rule

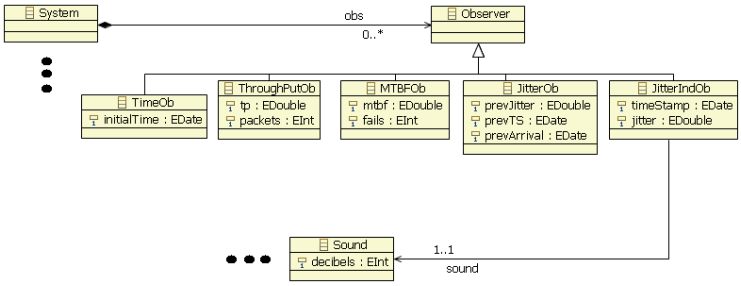


**Fig. 6.** Observers Class Diagram

### 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. As a first approach, we have extended our *Sound System* metamodel with five observers, which inherit from an Observer class (Fig. 6). Each of them has a specific purpose:

- TimeOb. This kind of observer stores (in its attribute initialTime) specific moments in time. In this case, one observer is used to keep track of the moment in time in which the system started working.
- 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.
- MTBFOb. Calculates the MTBF of the system (mtbf attribute). Its 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. These observers are associated to individual sounds (Fig. 6). They compute the jitter when their associated sounds reach their destination.

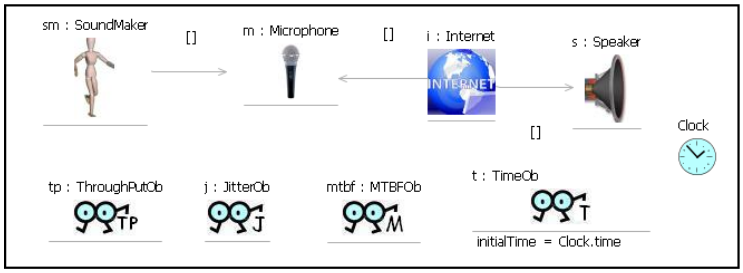


**Fig. 7.** Initial model of the system in its concrete syntax with observers

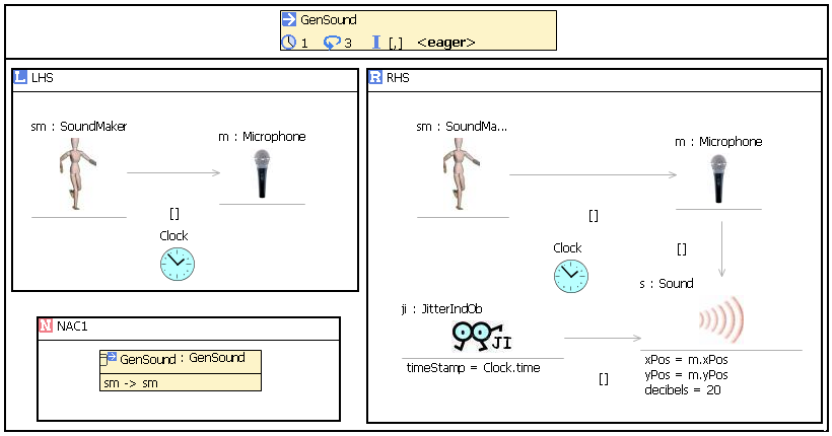


**Fig. 8.** GenSound with observers

Thus, we have added to the initial metamodel 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 value initialTime of the TimeOb is set with the time the system starts and is the only attribute whose value does not change. 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 in 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 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.

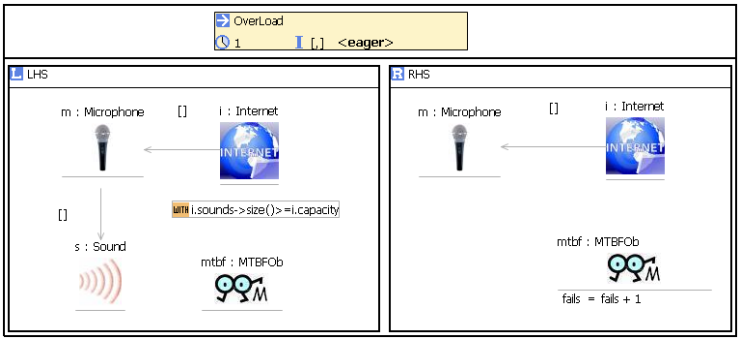


**Fig. 9.** OverLoad with observers

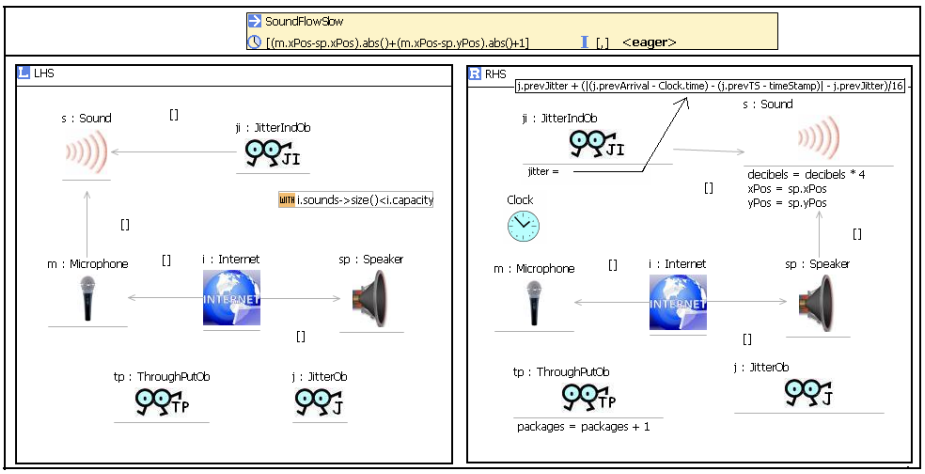


**Fig. 10.** SoundFlowSlow with observers

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. 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

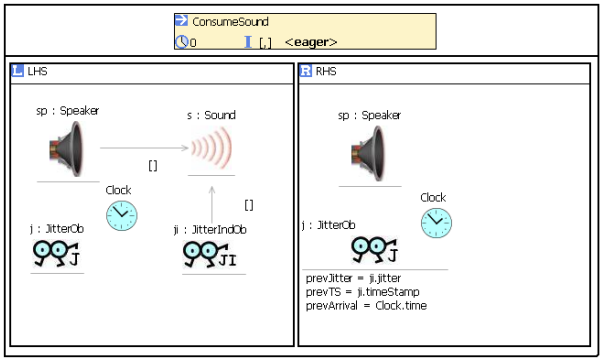


**Fig. 11.** New rule: ConsumeSound

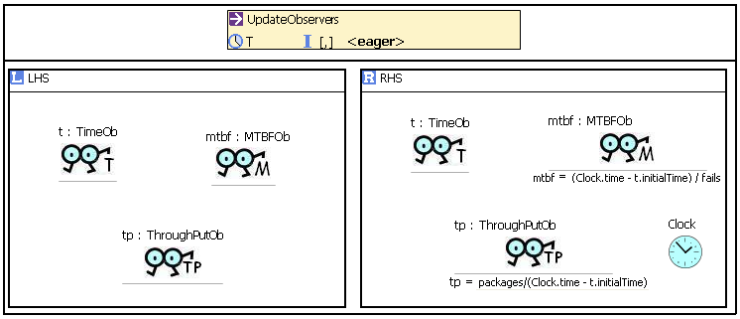


**Fig. 12.** New rule: UpdateObservers

way both observers always store correct and up-to-date QoS values at any moment in time T.

## 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. It may also make the specifications amenable to simulation and other kinds of formal analysis.

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 is self-adapted 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,

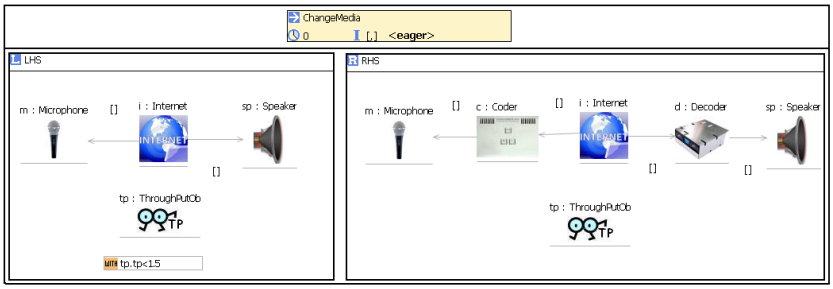


**Fig. 13.** New rule: ChangeMedia

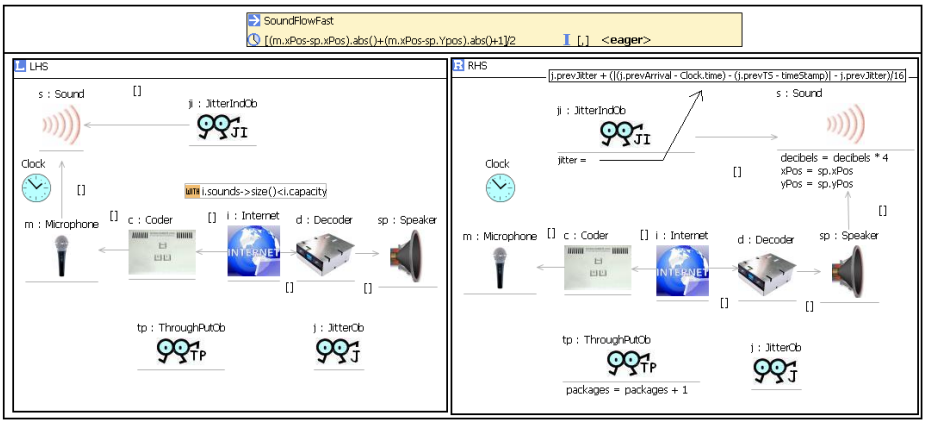


**Fig. 14.** SoundFlowFast with observers

consumes half the time the other rule does. In this way, the throughput value increases notably, improving the system performance. Similarly, Fig. 15 shows a rule which specify the transformation in the other direction. 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 system 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 [12] introduced a high level program monitoring and measuring system which supported very high level languages. One shortcoming of this proposal is that in many cases it generates essentially the same code as a human programmer would. In addition, it does not enable throughput assessment.

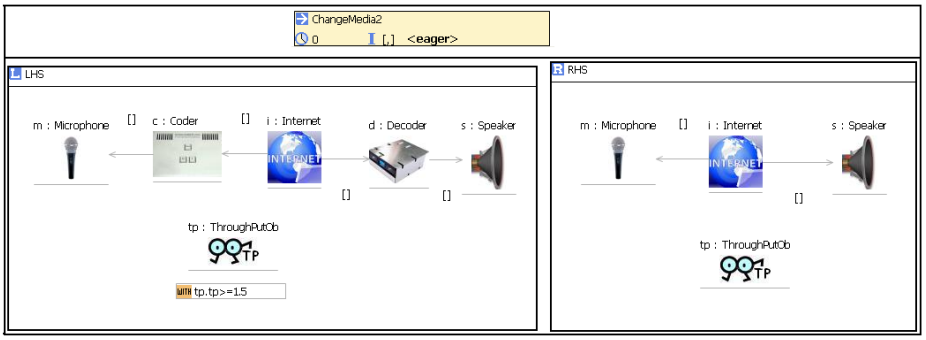


**Fig. 15.** ChangeMedia2: Inverse rule of ChangeMedia

Source-level program analysis and instrumentation tools target the source code of a program to be analyzed or instrumented. Compilers supporting aspect-oriented programming (AOP), such as AspectJ [13] and AspectC++ [14], may be considered source-level instrumentation tools. In AOP, it concerns that cross-cut modules are factored out into modular aspects. The aspects include code segments that are to be woven into a collection of modules and a specification of where the code fragments should be woven. AOP tools have been used to instrument programs with debugging and monitoring code [15], as well as to instrument programs with code to check temporal invariants [16]. As a shortcoming, AOP techniques can only be applied at well-defined join points and can be used for only instrumentation. 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, and can be used to analyze the system.

General frameworks for self-adaptive systems are presented in [17] and [18], featuring inter-related monitoring, analysis and adaptation tiers. Diaconescu et al. [19] add a transparent software layer between components and middleware. This framework aligns with frameworks presented in [17] and [18], while specifically targeting enterprise applications based on contextual composition middleware [20]. 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 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* [10] in this case), hence allowing users to make use of all the analysis possibilities available for that environment [9] 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 include them directly in our underlying language (*e-Motions*) not to modify the system metamodel when including QoS properties. 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 code generated standard by model-transformation approaches, with the aim of being able to generate monitors and probes associated to the code, too.

## References


1. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Trans. on Software Engineering **30**(5) (2004) 295–310
2. Cortellessa, V., Marco, A.D., Inverardi, P.: Integrating performance and reliability analysis in a non-functional MDA framework. In: Proc. of FASE 2007. Number 4422 in LNCS, Springer-Verlag (2007) 57–71
3. OMG: UML Profile for Schedulability, Performance, and Time Specification. OMG, Needham (MA), USA. (2005)
4. OMG: UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. OMG, Needham (MA), USA. (2004) ptc/04-09-01.
5. OMG: A UML Profile for MARTE: Modeling and Analyzing Real-Time and Embedded Systems. OMG, Needham (MA), USA. (2008)
6. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA'03 Workshop on Generative Techniques in the Context of MDA. (2003)
7. Rivera, J.E., Guerra, E., de Lara, J., Vallecillo, A.: Analyzing rule-based behavioral semantics of visual modeling languages with Maude. In: Proc. of SLE'08. Number 5452 in LNCS, Tolouse, France, Springer (2008) 54–73
8. de Lara, J., Vangheluwe, H.: Translating model simulators to analysis models. In: Proc. of FASE 2008. Number 4961 in LNCS, Springer (2008) 77–92
9. 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 (To appear in 2009)

10. 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'09), Corvallis, Oregon (US), IEEE Computer Society (2009)
11. Toncar, V.: VoIP Basics: About Jitter. (2007) `http://toncar.cz/Tutorials/VoIP/VoIP_Basics_Jitter.html`.
12. Liao, Y., Cohen, D.: A specification approach to high level program monitoring and measuring. IEEE Trans. on Software Engineering **18**(11) (1992) 969–978
13. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An overview of AspectJ (2001)
14. 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. (2002) 53–60
15. Mahrenholz, D., Spinczyk, O., Schroeder-Preikschat, W.: Program instrumentation for debugging and monitoring with Aspectc++. In: Proc. of ISORC'02. (2002) 249–256
16. Gibbs, T., Malloy, B.: Weaving aspects into C++ applications for validation of temporal invariants. In: Proc. of SMR'03. (2003)
17. Garlan, D., Cheng, S., Schmerl, B.: Increasing system dependability through architecture-based self-repair. In: Architecting Dependable Systems, Springer-Verlag (2003)
18. 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)
19. Diaconescu, A., Mos, A., Murphey, J.: Automatic performance management in component based systems. In: Proc. of ICAC'04. (2004) 214–221
20. Oreizy, P., Gorlick, M., Taylor, R., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., Wolf, A.: Component software: Beyond object-oriented programming, Addison-Wesley (2002)