

Towards a model-driven infrastructure for runtime monitoring^{*}

Antonia Bertolino, Antonello Calabrò, Francesca Lonetti
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR, Pisa, Italy
{firstname.secondname}@isti.cnr.it,
Antinisca Di Marco
Department of Computer Science, University of L'Aquila, Coppito (AQ), Italy
antinisca.dimarco@univaq.it, and
Antonino Sabetta^{**}
SAP Research, Sophia Antipolis, France
antonino.sabetta@sap.com

Abstract. In modern pervasive dynamic and eternal systems, software must be able to self-organize its structure and self-adapt its behavior to enhance its resilience and provide the desired quality of service. In this high-dynamic and unpredictable scenario, flexible and reconfigurable monitoring infrastructures become key instruments to verify at runtime functional and non-functional properties. In this paper, we propose a property-driven approach to runtime monitoring that is based on a comprehensive Property Meta-Model (PMM) and on a generic configurable monitoring infrastructure. PMM supports the definition of quantitative and qualitative properties in a machine-processable way making it possible to configure the monitors dynamically. Examples of implementation and applications of the proposed model-driven monitoring infrastructure are excerpted from the ongoing CONNECT European Project.

1 Introduction

Nowadays software systems are increasingly pervasive and dynamic, and their (lack of) quality may have a deep impact on businesses and people's lives. At the same time, as we entrust more and more responsibilities to distributed software systems, the need arises to augment them with powerful oversight and management functions in order to allow continuous and flexible monitoring of their behavior.

In this perspective, runtime monitoring is the key technological enabler both for quality assurance and for prolonging software lifecycle after deployment, by supporting runtime verification and online adaptation. Indeed, it provides a mean for evaluating and enhancing the resilience of dynamic and evolvable

^{*} This work is partially supported by the EU-funded CONNECT project (FP7-231167) and EU-funded VISION ERC project (ERC-240555).

^{**} Antonino Sabetta contributed to this paper while he was a researcher at CNR-ISTI, Pisa.

systems allowing the system to recover from abnormal situations or to prevent them by taking proactive actions.

Putting in place a working monitoring process involves several activities [16], including the collection of raw observation data, the interpretation of such raw information to recognize higher-level events that are relevant at the business level, and the effective presentation of the results of the monitoring. In real systems, the volume of collected raw data can be overwhelming. Moreover, such data need to be filtered and aggregated to detect deviations from expected behavior and to derive measurements of interesting non-functional properties of the system.

There is a gap to be filled when a high-level, business-relevant property is to be turned into a concrete setup of the monitoring infrastructure. In order to be able to evaluate and keep under control any such property, it is necessary to instruct the monitor about what raw data to collect and how to infer whether or not a desired property is fulfilled. Unfortunately, not only this task is time-consuming, but it also requires a substantial human effort and specialized expertise in order to convert the high-level description of the property to observe into lower-level monitor configuration directives. Consequently, the outcome of such effort is very hard to generalize and to reuse, and, as a matter of fact, the resulting monitor configuration typically is only relevant in the specific situation at hand. This process needs to be iterated each time a different architecture needs to be monitored, or when the properties to be monitored change.

To address the shortcomings of this process, in this paper we propose a model-driven approach [25] to monitoring that is based on two key elements: a generic monitoring infrastructure that offers the greatest flexibility and adaptability; and a coherent set of domain-specific languages, expressed as meta-models, to define models of properties that enable us to exploit the support to automation offered by model-driven engineering techniques. In the proposed approach, the properties to be monitored (either qualitative or quantitative) are specified according to a meta-model. Using this approach, and leveraging an underlying generic monitoring infrastructure, we can thus separate the problem of defining properties and metrics of interest, from the problem of converting these specifications into a concrete monitoring setup, which is done automatically in our approach.

The contribution of our proposal stays in: i) the effort (ambition) to offer a comprehensive meta-model for system properties that spans over dependability, performance, security and trust attributes by going beyond the state of the art since existing meta-models generally address only a subset of the above properties or do not support transformational approaches; ii) the inter-connection between the above meta-model and a modular event-based monitoring infrastructure.

The defined Property Meta-Model (PMM) is implemented as an eCore model and is provided with an editor. The ultimate vision we want to achieve is that a software developer using PMM can either retrieve from the editor repository the pre-built specification of a simple or complex metrics/property or, if not present,

can build new properties and metrics to be monitored, using the concepts in the meta-model. Also a non-expert domain user can benefit from the pre-built models. By transformation these property models will be translated into the input format for the monitor in order to be observed and verified at runtime.

Our research on monitoring originated in the context of the FP7 “ICT forever yours” European Project CONNECT¹. The CONNECT world envisions dynamic environments populated by technological islands which are referred to as the Networked Systems (NSs), and by the components of the CONNECT architecture, called the CONNECT Enablers. The ambitious goal of the project is to have eternally functioning systems within a dynamically evolving context, which is to be achieved by synthesizing *on-the-fly* the CONNECTors through which NSs communicate. The resulting emergent CONNECTors then compose and further adapt the interaction protocols run by the CONNECTed System. Evidently, such a dynamic context strongly relies on functional and non-functional behavior monitoring.

In the remaining part of this paper, we describe: the Property Meta-Model we defined to specify properties (Section 2); the GLIMPSE monitoring infrastructure that implements the Generic Monitoring Framework (Section 3); and finally the Monitor Configuration steps that from the PMM property models generate the code used to configure GLIMPSE to monitor the relevant properties (Section 4). An application example (Section 5), related work (Section 6), and conclusions (Section 7) complete the paper.

2 Property Meta-model

In this section, we give an overview of the Property Meta-Model (PMM) we defined for specifying observable properties of the system. Figure 1 sketches the key concepts of this meta-model that are: *Property*, *MetricsTemplate*, *Metrics*, *EventSet*, *EventType*, *ApplicationDomain*, and how they relate each other.

The Property Meta-Model describes a property that can be *ABSTRACT*, *DESCRIPTIVE*, or *PRESCRIPTIVE*. An *ABSTRACT* property indicates a generic property that doesn’t specify a required or guaranteed value for an observable or measurable feature of a system. A *DESCRIPTIVE* property represents a guaranteed/owned property of the system while a *PRESCRIPTIVE* one indicates a system requirement. In both cases, the property is defined taking into account a relational operator with a specified value. A property can be qualitative or quantitative: the former is about events that are observed and cannot generally be measured, referring to the behavioral description of the system (e.g., deadlock freeness or liveness); the latter deals with quantifiable/measurable observations of the system and it has an associated *Metrics*. The *Quantitative-Property* can have a *Workload* and an *IntervalTime*. The *Workload* can be open or close. To clarify the above concepts we report below two **Property** examples:

Property1: The system S in average responds in 3 ms in executing the e_1 operation with a workload of 10 e_2 concurrent operations.

¹ <http://connect-forever.eu>

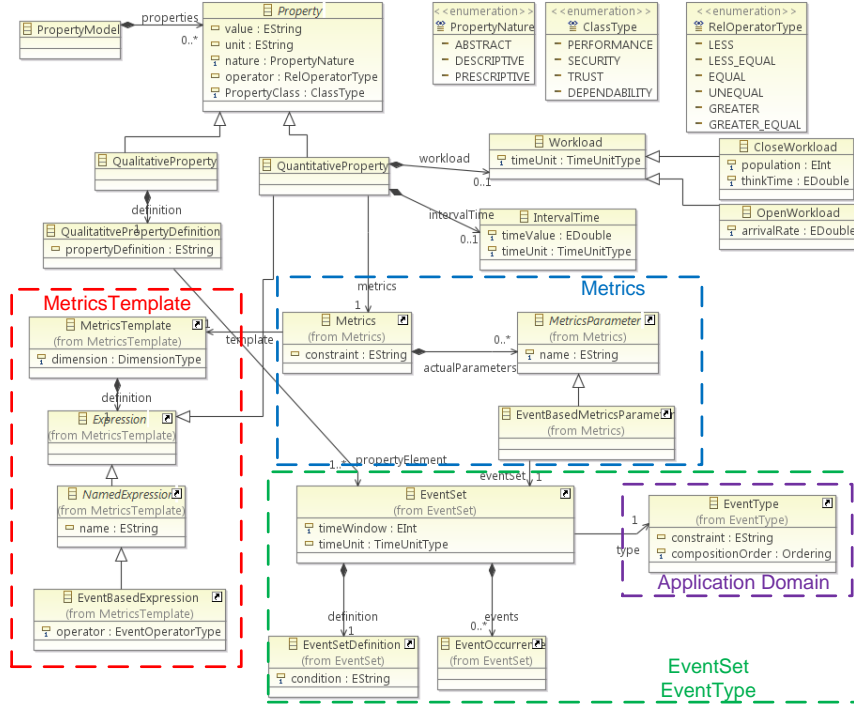


Fig. 1. Key concepts of the Meta-Model for processable properties.

Property2: The system S in average must respond in 3 ms in executing the e_1 operation with a workload of 10 e_2 concurrent operations.

The former is a *DESCRIPTIVE* property while the latter is a *PRESCRIPTIVE* one because it specifies a required time response with a value of 3 ms in executing the e_1 operation. Both are quantitative properties having a *PERFORMANCE* class because they refer to a measurable performance dimension (time).

In the proposed meta-model, we make a distinction between a generic metrics (represented by a `MetricsTemplate`) and the concrete metrics (i.e., the `Metrics` concept) instantiated to a specific application domain, represented by the domain of the software system the PMM is used for. The `MetricsTemplate` represents the way a metrics can be specified. Hence a metrics (for example the response time) can refer to one or more (hopefully equivalent) specifications represented by different `MetricsTemplates`. The response time of an operation (E), for example, can be expressed as the duration of that operation or the difference between the timestamps of the ending (E1) and starting (E2) actions of that operation. E, E1 and E2 are templateParameters (not showed in Figure 1). A `MetricsTemplate` contains the definition of mathematical operators, nested into complex mathematical expressions, and expressions (represented by `NamedExpressions`)

to which these operators are applied. We distinguish two `NamedExpression` types that are `ActionBasedExpression` and `EventBasedExpression`. The former represents a simple action or a sequence of actions that, when executed, reports a value. More interesting, for the monitoring purposes, is the latter that represents expressions based on events or observational behaviors. We define some operators that are applied to single occurrences of simple or complex events (for example `DURATION` refers to a complex event and `TIMESTAMP` to a simple one), and other operators (such as `CARDINALITY`) that are applied to the whole set of event occurrences observed in a given instant of time (these operators are not showed in Figure 1).

A metrics refers to a `MetricsTemplate` and instantiates the `templateParameters` by means of the `MetricsParameters`. The `Metrics` actualizes the `MetricsTemplate` for a specific scenario and it is specific to the application domain it is defined for. This characteristics is modelled by the metrics actual parameters that substitute the `templateParameters` by linking the general description in the template to the specific ontology and hence to the application the metrics refers to. The `EventBasedMetricsParameter` is a `MetricsParameter` that actualizes the `EventBasedExpression` based `templateParameters`. To this goal, it refers to the `EventSet` describing the application based event definition and the associated occurrences.

An `EventSet` represents a set of event instances that refer to an `EventType`. An `EventSet` has zero or more `EventOccurrence` representing the observable events that the `EventSet` contains. In the monitored system this `EventOccurrence` can be generated at runtime by the monitoring infrastructure when the probes observe the event of the `EventType` the `EventSet` refers to.

The `EventType` models an observable system behavior that can be a primitive/simple event or operation representing the lowest observable system activity or a composite/complex event that is a combination of primitive and other composite events. An `EventType` has a specification identifying the type or class of the observable events. Such `EventTypeSpecification` belongs to the ontology of a specific application domain. In the current version of the meta-model this specification is simply defined by means of a label or string. However, in the future we plan to provide a more formal specification for complex event definition, according to an existing or a new defined event specification language. In [10, 17] some examples of complex event specification languages are presented.

The devised meta-model has been generated as an eCore model into the Eclipse Modeling Framework(EMF) [11]. In particular, we define the meta-model partitioned in the following eCore models: *Core.ecore* representing a generic named element, *EventType.ecore* and *EventSet.ecore* modeling the event and the eventSet respectively, *Metrics.ecore* and *MetricsTemplate.ecore* for specifying the metrics and metricsTemplate concepts and finally the *Property.ecore* representing the Property meta-model.

From the above defined eCore models, by means of the EMF facilities, an editor has been obtained as an Eclipse Plugin. This editor contains the informa-

tion of the defined eCore models and allows to create new model instances of the Property, Metrics, MetricsTemplate, EventType and EventSet meta-models.

The presented Property Meta-Model has been used for defining properties relevant for the CONNECT project². It proved to be complete for specifying all properties of interest.

The models conforming to such meta-model can be used to drive the instrumentation of the CONNECT monitoring Enabler that generates suitable probes to monitor useful properties on the CONNECTORS.

3 GLIMPSE architecture

In large and distributed systems, huge amount of events are generated and from their combination and filtering it is possible to timely detect unexpected behaviors of the systems or predict failures for enhancing the system resilience. GLIMPSE³, is a flexible monitoring infrastructure, developed with the goal of decoupling the event specification from the analysis mechanism. GLIMPSE was initially proposed in the context of the CONNECT project, where it is used to support behavioral learning, performance and reliability assessment, security, and trust management. A prototype of the GLIMPSE architecture⁴ has been developed and is being used in CONNECT. However, the infrastructure is totally generic and can be easily applied to different contexts. The architecture of GLIMPSE (shown in figure 2) is composed of five main components that implement the main five core functions identified into a generic monitoring infrastructure [16], namely: Data collection, Local interpretation, Data transmission, Aggregation, Reporting.

Probes (Collector/Data Suppliers) Probes intercept primitive events when they occur in the software and send them to the GLIMPSE Monitoring Bus. Probes are usually realized by injecting code into an existing software or by using proxies. In addition, they may be configured to use a primitive event filter in order to reduce the amount of generated raw data.

Monitoring Bus The Monitoring Bus is the communication backbone that all information (events, questions, answers) is sent on: Probes, Consumers, Complex Event Processor and by all the services querying information to GLIMPSE. We adopt a publish-subscribe paradigm devoting the communication handling to the Manager component.

In the current GLIMPSE implementation, the system backbone is implemented by means of ServiceMix4 [4], an open source Enterprise Service Bus, used to combine advantages of event-driven architecture and service-oriented architecture functionality. We chose ServiceMix4 because it offers a Message Oriented Bus and is able to run an open source message broker like ActiveMQ [1].

² A first release of the CONNECT Property Meta-Model is available at <http://labse.isti.cnr.it/tools/cpmm>

³ Generic fLexIble Monitoring based on a Publish-Subscribe infrastructure

⁴ Available at <http://labse.isti.cnr.it/tools/glimpse>

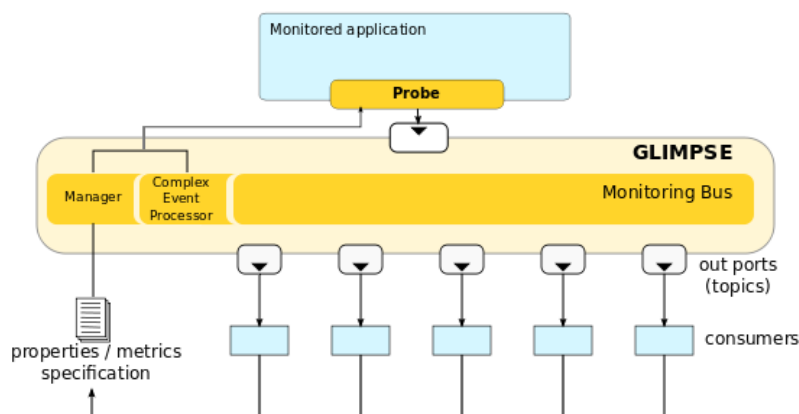


Fig. 2. Glimpse architecture

Complex Event Processor The Complex Event Processor (CEP) is the rule engine which analyzes the primitive events, generated from the probes, to infer complex events matching the consumer requests. There are several rule engines that can be used for this task (like Drools Fusion [2], RuleML [3]).

In the current GLIMPSE implementation, we adopt the Drools Fusion rule language [2] that is open source and can be fully embedded in the realized Java architecture. Note that, the proposed flexible and modular architecture allows for easily replacing this specific rule language with another one.

Consumer It may be a learning engine, a dependability analyzer or a simple customer that requests some information to be monitored. It sends a request to the Manager using the Monitoring Bus and waits for the evaluation results on a dedicated answer channel provided by the Manager.

Manager The Manager component is the orchestrator of the GLIMPSE architecture. It manages the communications among the GLIMPSE components. Specifically, the Manager fetches requests coming from Consumers, analyzes them and instructs the Probes. Then, it instructs the CEP Evaluator, creates and notifies to the Consumer a dedicated channel on which it will provide results produced by the CEP Evaluator.

4 Property-Driven Monitoring Configuration

We have described in the previous sections the Property Modeling sub-process and the Generic Monitoring Infrastructure we intend to use. Here we briefly explain how these two sub-processes are combined, i.e., we describe the automatic “Monitors Configuration” we propose. Precisely, the editor provided along with PMM allows the software developer to specify a new property as a model that is

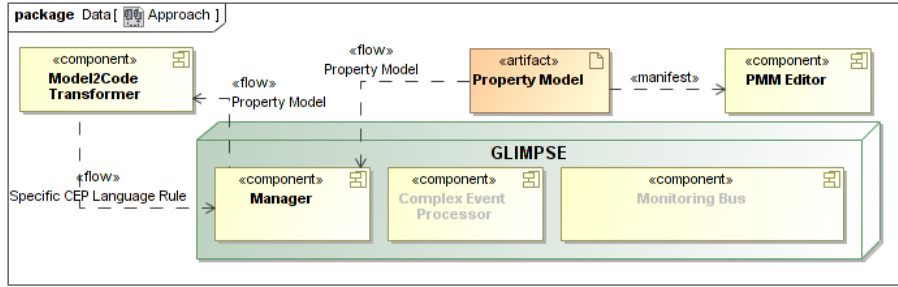


Fig. 3. Property-Driven GLIMPSE Configuration.

conforming to the PMM meta-model. If this model represents a property to be monitored, it can be used to instruct the GLIMPSE infrastructure, according to the approach sketched in Figure 3. As shown, the GLIMPSE manager component takes in input such property model and activates an external component (named in the figure modelToCode Transformer), which performs the code generation according to the specific complex event processing language that is embedded into GLIMPSE. The output of this transformation is represented by a specific rule that is processed by the complex event processor component of GLIMPSE. For the sake of precision, at the time of writing the modelToCode Transformer component is still under development and the current running implementation of GLIMPSE infrastructure uses the Drools Fusion complex event processor [2]. Following the depicted property-driven approach, we are developing an automated modelToCode Transformer component (using Acceleo code generator tool) according to the Drools Fusion rule specification language and we are refining the meta-model in order to perform this. Indeed, the advantage of adopting a model-driven approach is that it allows the monitor to use any complex event processing engine as long as a modelToCode Transformer transforms the property model into the rule specification language of that processing engine. As an application example, in the next section we show in detail a property model, specified using the PMM meta-model, and the corresponding Drools Fusion rule.

5 Application example

In this section we give an application example of our approach. Specifically, we first present the Terrorist Alert Scenario which is one of the demonstration examples chosen in CONNECT. Then, we show how to model using the PMM a latency property required in the system for that scenario, and finally how to express this property by means of the rule specification language used by Drools Fusion. Modeling of this latency property represents a simple example for giving an idea of the proposed approach. For the Terrorist Alert Scenario, we modeled, using the PMM, also dependability and security properties that we do not present here for space limitation reasons and we refer to [9] for their description.

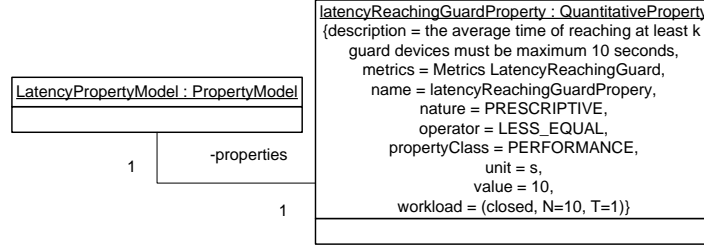


Fig. 4. Latency Property for the Terrorist Alert Scenario

Terrorist Alert scenario The CONNECT Terrorist Alert scenario [8], depicts the critical situation that during the show the stadium control center spots one suspect terrorist moving around. The alarm is immediately sent to the Policemen, equipped with ad hoc handheld devices which are connected to the Police control center to receive commands and documents, for example a picture of a suspect terrorist. Unfortunately, the suspect is put on alert from the police movements and tries to escape, evacuating the stadium. The policeman that sees the suspect running away can dynamically seek assistance to capture him from civilians serving as private security guards in the zone of interest. To get help in following the moves of the escaping terrorist and capturing him, the policeman sends to the civilian guards an alert message in which a picture of the suspect is distributed. On their side, to perform their service, the guards that control a zone are CONNECTed in groups and are equipped with smart radio transmitters. The guards control center sends an EmergencyAlert message to all guards of the patrolling groups; the message reports the alert details. On correct receipt of the alert, each guard’s device automatically sends an ack to the control center.

Latency Property for the Terrorist Alert Scenario We show how to model the following required latency property: *average time needed by the CONNECTed system to reach k% guard devices must be at most equal to 10 seconds when in the system there are 10 alerts*. For “time needed by the CONNECTed system to reach a set percentage of guard devices” we mean the average latency experienced in the system from the incoming EmergencyAlert message to the reception of a percentage of eAck coming back from the reached guards’ devices. The model for this PRESCRIPTIVE property is shown in Figure 4. This is a PERFORMANCE property requiring that the associated metrics is LESS_EQUAL to 10 s when the system has a workload of 10 alerts. This metrics called *LatencyReachingGuard* (omitted due to space limitations) is an instance of the Average Latency MetricsTemplate that is presented in Figure 5. We recall that the template is generic and the same model can be used in other scenarios. The average latency represents a TIME measure defined as average of the differences of the timestamps of two related generic event instances (x and y in the model), respectively as the latest event occurrence and the former one. Finally, the template exposes two templateParameters: e_1 bound to y , and e_2 bound to x . A Metrics,

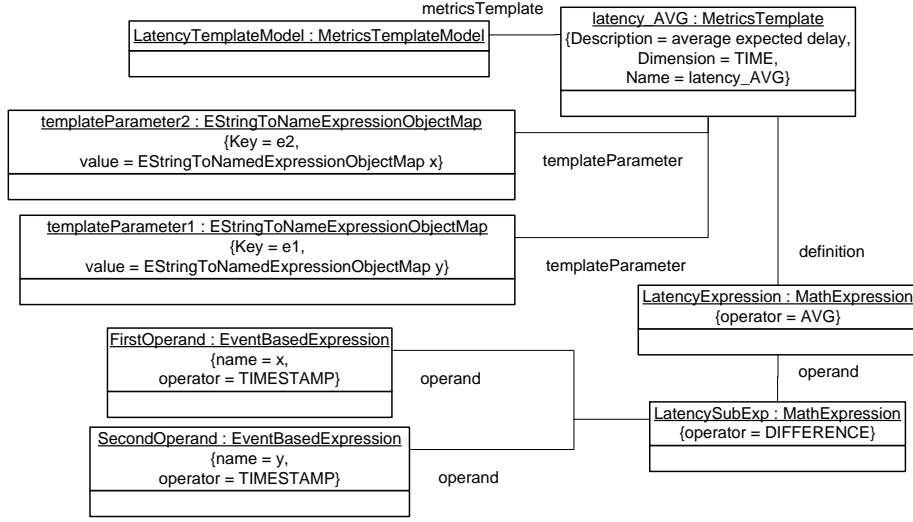


Fig. 5. Average Latency Metrics Template

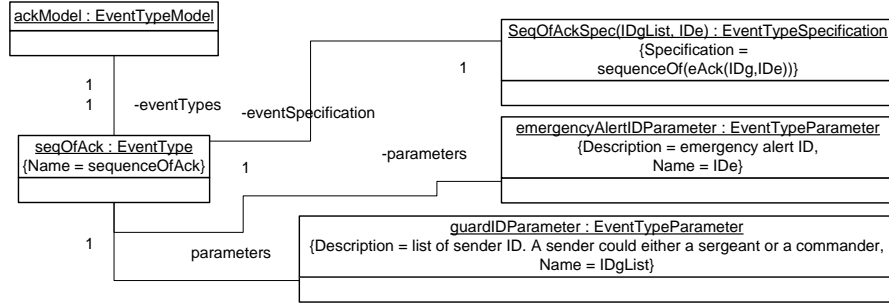


Fig. 6. Sequence of Ack for an Alert

whose definition is an instance of the MetricsTemplate, concretises the template for a specific scenario. This is reflected in the metrics’ actual parameters which substitute the template parameters by linking the general description in the template to the specific ontology and hence to the application the metrics refers to. The *LatencyReachingGuard* metrics actualizes the corresponding Average Latency MetricsTemplate by linking to the templateParameters e_1 and e_2 , two EventSets (e_1 and e_2 respectively), and specifying that the two event sets must satisfy a metrics constraint establishing that the two event sets must be related to each other. Referring to the Terrorist Alert scenario, the e_1 Event Set refers to the emergencyAlert Event Type that is a simple event definition since it corresponds to a message directly observable from the system. The e_2 EventSet instead refers to SeqOfAck EventType that is a sequence of eAck observed in the

```

1 declare SimpleEvent
2   @role(event)
3   @timestamp(timestamp)
4 end
5 rule "countIncoming"
6 when
7   $total : Number()
8 from accumulate($bEvent:
9   SimpleEvent(data == "incomingRequest",
10    this.getConsumed == false)
11 from entry-point "DEFAULT", count ($bEvent))
12 then
13   DroolsUtils.ActuallyIncoming($total);
14 end
15 rule "checkCompleted"
16 when
17   $aEvent:
18     SimpleEvent(this.data == "incomingRequest",
19     this.getConsumed == false);
20   $bEvent:
21     SimpleEvent(this.data == "outcomingResponse",
22     [... parameters check ...],
23     this after $aEvent);
24 then
25   $aEvent.setConsumed(true);
26   $bEvent.setConsumed(true);
27   SatisfiedRequest sr = new SatisfiedRequest();
28   sr.setIncoming($aEvent);
29   sr.setOutcoming($bEvent);
30   sr.setDuration(DroolsUtils.latency(
31     $aEvent.getTimestamp(),$bEvent.getTimestamp()));
32   insert (sr);
33   System.out.println("Last Request Completion Time: "
34     + sr.getDuration());
35 end
36 rule "requestCompletedInTime"
37 when
38   $totalCompleted : Number()
39   from accumulate($aCompleted : SatisfiedRequest()
40     from entry-point "DEFAULT",
41     average($aCompleted.getDuration()))
42 then
43   DroolsUtils.CheckViolation($totalCompleted);
44 end

```

Listing 1.1. Drools latency rule example

system. To be of interest of the *LatencyReachingGuard* metrics, the occurrences of SeqOfAck must contain at least k eAck occurrences related to each other, that means they refer to the same emergencyAlert message. The SeqOfAck EventType, shown in Figure 6, is a complex EventType defined as a sequence of eAck simple EventTypes. It has two parameters: the emergencyAlert ID (namely *IdE*) the sequence refers to, and the list of guards messages acknowledging the alert (namely, *IdgList*). We recall that the occurrences of events are generated by the monitors at run-time when the system is running, once the probes observe the event of the EventType the EventSet refers to.

Drools rule specification for the Latency Property The Listing 1.1 shows a fragment of the Drools rules used to monitor the Latency property for the Terrorist

Alert Scenario depicted in Figure 4. Each event flowing on the GLIMPSE Monitoring Bus is an abstraction of a state transition of an LTS machine. In our LTS machine, the e_1 operation starts with the event `incomingRequest` and finishes with the `outcomingResponse` event. The first rule in Listing 1.1, `CountIncoming`, counts the incoming requests (lines 5-14). The `checkCompleted` rule (lines 15-35), analyzes the `incomingRequest` event and the `outcomingResponse` sent by the same client, using the fields: `connectorInstanceID`, `connectorInstanceExecution`, and creates a subset of accomplished requests (`SatisfiedRequest`) saving the computation execution time in the field `Duration`. Finally, the rule `requestCompletedInTime` (lines 36-44), evaluates the average time spent for completing requests.

6 Related work

The approach proposed in this paper is strictly related to two areas of research: a) the specification of meta-models for defining software metrics and non-functional properties, and b) the design of runtime monitoring systems. This section presents a brief selection of relevant works from both these areas.

The main concept underlying our proposal, i.e., specifying metrics as instances of a metrics specification meta-model, is common to the work of Monperrus et al. [19], in which a generative model driven definition of software metrics is proposed. This work concerns the definition of a *domain-independent metrics meta-model*, allowing modelers to automatically add measurement capabilities to a domain specific modeling language used during the different phases of a model-driven development process (such as architectural design, requirements or implementation). Taking inspiration from the work of Monperrus et al., PMM separates the property and more specifically metrics definition from the application domain. Instead, differently from [19], PMM addresses specifically *non-functional property and metrics*. Indeed, it introduces additional concepts concerning the qualitative and quantitative properties definition, the events modeling and the distinction between a generic metrics (represented by a `MetricsTemplate`) and the concrete metrics (i.e., the `Metrics` concept) instantiated to a specific application domain.

Several works addressed meta-modeling focussing on domain-specific metrics or dependability properties [22, 13]. For different reasons, these approaches propose partially what PMM proposes as a whole. PMM allows for the specification of different types of properties, such as, among the others, performance, security, dependability and trust properties and of relative metrics.

A more general Quality of Service Modeling Language (QML) is proposed in [12] for describing QoS specifications for software components in distributed object systems. It is an extension of UML, allowing a fine grained specification level of attributes and operations and a dynamic and runtime check of QoS components requirements and dependencies. Again, the UML MARTE profile [21] provides a common way for modeling hardware and software aspects of a real time embedded system. It provides facilities to annotate models with informa-

tion required to perform quantitative predictions and performance analysis. A model-driven performance measurement and assessment approach is presented in [6]. It provides a meta-model for the specification of performance metrics and the possibility of specifying measurement points in the model. It allows for the automatic instrumentation and software code generation with integrated code for performance data collection, storage and metrics computation. An evaluation of the proposed approach is provided with an implementation of a UML profile and transformations from the profile to Java. Differently from the previous approaches, our proposal focuses on a more general and complete framework, that allows not only for specifying performance measures but enables the specification of qualitative and quantitative properties into a machine-processable language.

A promising research direction, addressing QoS modeling, focuses on ontologies that allow for the definition of QoS with rich semantic information. In particular, [15] presents a semantic QoS model addressing the main elements of dynamic service environments (networks, devices, application services and end-users). It makes use of Web Service Quality Model (WSQM) [20] standard to define QoS at the service level, and comprehends four ontologies specifying respectively: the core QoS concepts, the environment and underlying network and hardware infrastructure QoS properties, the application server and user-level QoS properties. As the authors claim in [15], their model concentrates on QoS knowledge representation rather than on a language to specify QoS. In this way, they provide separate and reusable ontologies and any appropriate QoS specification language can be used on top of it. Differently from this approach, PMM allows the specification of non-functional properties.

Concerning monitoring systems, the literature is rich of proposals of frameworks, languages, and architectures [23, 14]. In particular, [23] presents an extended event-based middleware with complex event processing capabilities on distributed systems. Similar to GLIMPSE this work adopts a publish/subscribe infrastructure. Another monitoring architecture for distributed systems management is presented in [14]. This architecture employs a hierarchical and layered event filtering approach, specifically targeted at improving scalability and performance for large-scale distributed systems, minimizing the monitoring intrusiveness.

Defining expressive complex event specification languages has been an active research topic for years [17, 7, 10]. Among these languages, GEM [17] is a generalized and interpreted event monitoring language. It is rule-based (similar to other event-condition-action approaches) and provides a detection algorithm that can cope with communication delay. Snoop [7] follows an event-condition-action approach supporting temporal and composite events specification but it is especially developed for active databases. A more recent formally defined specification language is TESLA [10] that has a simple syntax and a semantics based on a first order temporal logic. The main focus of these works is the definition of a complex-event specification language, whereas our framework provides a more high-level and more specialized meta-model to define monitoring goals

(functional properties and metrics definitions), which are then automatically translated into complex-event specifications.

Other monitoring frameworks exist that address the monitoring of performance, in the context of system management [5, 18, ?]. Among them, Nagios [5] offers a monitoring infrastructure to support the management of IT systems spanning network, OS, applications; Ganglia [18] is especially dedicated for high-performance computing and is used in large clusters, focusing on scalability through a layered architecture whereas the Java Enterprise System Monitoring Framework [?] deals with web-based and service-driven networks solutions.

Finally, the work in [24] has several similarities with our approach, concerning the conceptual modeling of non-functional properties. However it is more specifically focused on measurement refinement, whereas our work targets a more comprehensive scope for modeling and transformation. In future work we plan to look closely at this model to possibly incorporate some of its refinements.

7 Conclusions

We proposed a model-driven infrastructure for runtime monitoring. The monitoring configuration is automatically executed by parsing the models of the properties of interest. To allow for automatization, such models must conform to a suitable meta-model. In this paper we presented: *i*) a Property Meta-Model we devised to express properties and metrics. It allows for the definition of prescriptive/descriptive and qualitative/quantitative properties, and the metrics needed to quantify them. *ii*) GLIMPSE, which is an implementation of a generic monitoring infrastructure; and, *iii*) the model-driven monitor configuration, combining PMM and GLIMPSE. As proof of concept, we finally showed the application of the model-driven infrastructure for runtime monitoring to a CONNECT application scenario.

There are various directions for future work. First, as outlined in Section 4, we plan to implement the ModelToCode transformations to automatically derive the Drools rules needed to configure GLIMPSE. For what concerns PMM, some meta-model parts need to be refined. Among others, we need to refine: *i*) the EventTypeSpecification meta-class by introducing an event-based language that enables the specification of complex EventType; and *ii*) the QualitativePropertyDefinition meta-class, by defining a suitable language (meta-model) allowing for the specification of complex properties. Finally, so far PMM only supports the specification of the transition (or action)-based properties. However, state-based properties could be relevant in some cases. We plan to extend PMM in order to support also the modeling of state-based properties.

Moreover, in the future we want to address the reliability and performance issues of the proposed monitoring framework when a lot of data are generated, providing a comparison of GLIMPSE with similar existing approaches.

References

1. ActiveMQ: A complete message broker. <http://activemq.apache.org>.

2. Drools Fusion: Complex Event Processor. <http://www.jboss.org/drools/drools-fusion.html>.
3. Java Enterprise System Monitoring Framework. <http://download.oracle.com/docs/cd/E19462-01/819-4669/geleg/index.html>.
4. Ruleml: The rule markup initiative. <http://ruleml.org>.
5. ServiceMix: an open source ESB. <http://servicemix.apache.org/home.html>.
6. W. Barth. *Nagios. System and Network Monitoring*. No Starch Press, u.s. ed edition, 2006.
7. M. Bošković and W. Hasselbring. Model Driven Performance Measurement and Assessment with MoDePeMART. In *Proceedings of MODELS*, pages 62–76, 2009.
8. S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering*, 14(1):1–26, 1994.
9. CONNECT Consortium. Deliverable 5.2: Design of Approaches for dependability and initial prototypes. <http://connect-forever.eu/>, 2011.
10. CONNECT Consortium. Deliverable 6.1: Experiment scenarios, prototypes and report. <http://connect-forever.eu/>, 2011.
11. G. Cugola and A. Margara. TESLA: a formally defined event specification language. In *Proceedings of DEBS*, pages 50–61, 2010.
12. Eclipse Platform, Eclipse Modeling Project. <http://www.eclipse.org/modeling/>.
13. S. Frolund and J. Koistinen. Quality-of-Service Specification in Distributed Object Systems. *Distributed Systems Engineering Journal*, 5:179–202, 1998.
14. M. Huhn and A. Zechner. Analysing dependability case arguments using quality models. In *Proceedings of SAFECOMP*, pages 118–131. 2009.
15. E. Hussein, H. Abdel-wahab, and K. Maly. HiFi: A New Monitoring Architecture for Distributed Systems Management. In *Proceedings of ICDCS*, pages 171–178, 1999.
16. N. Mabrouk, N. Georgantas, and V. Issarny. A semantic end-to-end QoS model for dynamic service oriented environments. In *Proceedings of PESOS*, pages 34–41, 2009.
17. M. S. Masoud and M. Sloman. Monitoring distributed systems. *Network and distributed systems management*, pages 303–347, 1994.
18. M. Massie, B. Chun, and D. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.
19. M. Monperrus, J. Jézéquel, B. Baudry, J. Champeau, and B. Hoeltzener. Model-driven generative development of measurement software. *Software and Systems Modeling (SoSyM)*, 2010.
20. OASIS. Quality Model for Web Services (WSQM), September 2005.
21. OMG. UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE). <http://www.omg.org/omgmarte/Specification.htm/>.
22. A. Pataricza and F. Györ. Towards unified dependability modeling and analysis. In *Proceedings of ARCS Workshops*, pages 113–122, 2004.
23. P.R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *Network, IEEE*, 18(1):44–55, jan. 2004.
24. S. Röttger and S. Zschaler. Tool Support for Refinement of Non-functional Specifications. *Software and Systems Modeling*, 6(2):185–204, 2007.
25. M. Samani and M. Sloman. GEM: a generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.
26. D. C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), 2006.