SLA Compliance Checking and System Runtime Reconfiguration — A Model Based Approach

Mahin Abbasipour[1], Ferhat Khendek[1], Maria Toeroe[2]

[1] Electrical and Computer Engineering, Concordia University, Montreal, Canada.
[2] Ericsson, Montreal, Canada.

* Corresponding author. Email: Maria.Toeroe@ericsson.com
Manuscript submitted August 10, 2019; accepted October 18, 2019.
doi: 10.17706/jsw.14.11.488-518

Abstract: Service providers aim at optimizing system resource utilization while ensuring the quality of service expressed in the Service Level Agreements (SLAs) is met. For this purpose, systems are reconfigured dynamically according to workload variations to satisfy the SLAs while using only the necessary resources. Whenever a dynamic reconfiguration is required because of low resource utilization or potential SLA violations, one or more triggers may be generated. These generated triggers invoke elasticity rules that define actions to be taken in each specific situation. The elasticity rules that are invoked at the same time may lead to actions that may impact each other. As a result, handling each trigger independently may threaten the stability of the system. In this paper, we propose a model-driven framework, which manages the compliance of SLAs and enables dynamic reconfiguration. We use UML models to describe all the artifacts in the framework. All SLA models are transformed into an SLA compliance model which is used at runtime to check SLA compliance and generate triggers when a dynamic reconfiguration is required. In this framework, the actions of the elasticity rules invoked simultaneously are correlated before their application. The proposed correlation is based on the relations between the triggers. We perform a preliminary evaluation of the approach.

Key words: Correlation, elasticity rules, dynamic reconfiguration, model driven approach, trigger, SLA violation avoidance.

1. Introduction

Service Level Agreement (SLA) [1] is a contract negotiated and agreed upon by a provider and a customer. It describes the quality of services provided to the customer as well as the rights and obligations of each party. When any of the parties fails to meet its obligations, it may be subject to penalties.

System workload varies over time, which results in variable resource usage. To increase revenue, instead of allocating a fixed amount of resources, service providers try to allocate only as much as needed to support the workload and adapt this allocation according to the workload variations. In the cloud environment, the dynamic resource provisioning according to workload variations is called elasticity. A cloud system evolves and adapts dynamically to workload variations by scaling out/in and up/down [2].

To scale out/in or up/down, the system needs to be monitored, measurements need to be collected and the SLAs checked periodically. Whenever there is a potential SLA violation (i.e. an SLA is about to be violated in the near future) or the measurements show that the resource utilization is low, the system is adapted accordingly. Generally, in order to achieve this one or more triggers are generated automatically to invoke

elasticity rules applicable in the current system's situation [3]. Invocation of an elasticity rule leads to the execution of an action. This action applied on a system entity may have an impact on other entities because of the relations and the dependencies between the entities. As a result, handling each trigger independently or in an ad hoc manner is sub-optimal and may compromise the stability of the system. For illustration purpose, let us assume two triggers t1 and t2 that invoke two opposite elasticity rules e1 and e2, respectively, where e1's action is to remove a node and e2's action is to add a node. If the triggers are handled separately, resource oscillation [4] will certainly occur. On the other hand, in cloud systems because of the multiple layers (infrastructure, platform and application layer), one root cause may generate multiple triggers in these different layers. For example, some workload decrease at the application layer may cause triggers at the application layer as well as triggers at the infrastructure layer. If these triggers are considered separately, the invoked elasticity rules may remove some critical resources twice and this may jeopardize the availability [30], [40] of the service.

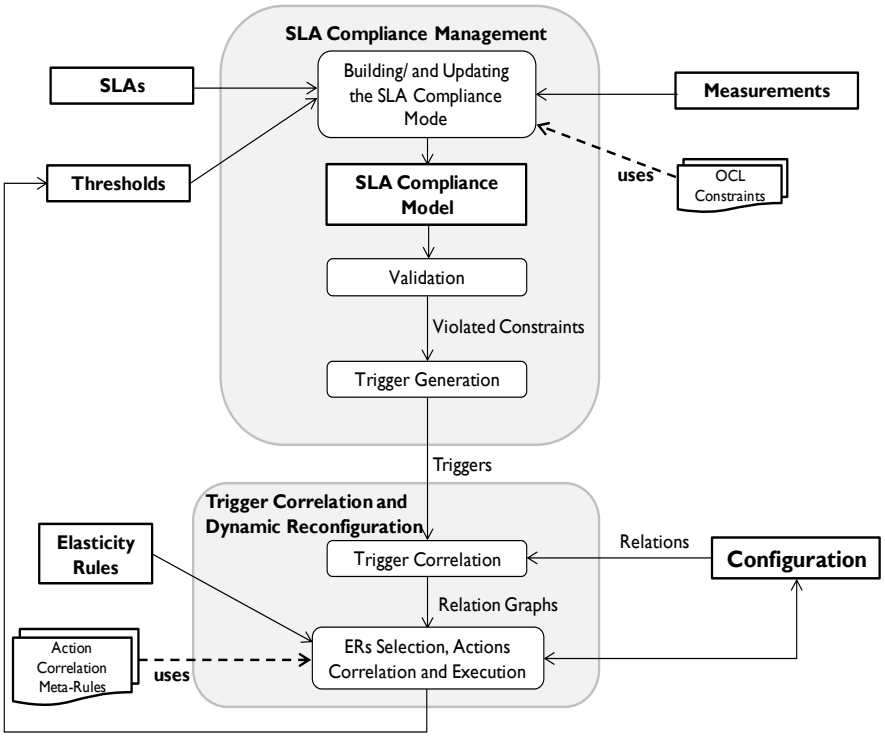

Fig. 2. The overall picture for SLA compliance management and dynamic reconfiguration.

In this paper, we propose a model-driven management framework which aims at managing SLA compliance and enabling system dynamic reconfiguration whenever required. Contrarily to current work on elasticity in the cloud, our framework does not only add or remove resources to/from the system, but it also allows for rearranging the resources to optimize their utilization. The Model-Driven Development (MDD) [5] paradigm separates the application logic from the platform technology and manipulates platform-independent models; thus models are the primary artifacts in the development process [6]. The major advantage of this paradigm is that the models are at higher level of abstraction than the implementation technology. This paradigm is appropriate for our purpose as it allows not only to facilitate the understanding, design and maintenance of the system [7], but also to reuse the models generated during the system design phase.

Fig. 1 shows the overall picture of our management framework. In the SLA Compliance Management process, all the SLAs, their corresponding measurements and the thresholds are combined into an SLA compliance model. The validation of the SLA compliance model against its metamodel is performed periodically. The violation of Object Constraint Language (OCL) [8] constraints during this validation will generate automatically triggers for system reconfiguration, to save resources when the workload decreases or avoid SLA violations when the workload increases and the SLAs are about to be violated.

In the *Trigger Correlation and Dynamic Reconfiguration* process, the triggers generated on related entities of the configuration are first correlated and a set of graphs called relation graphs are defined. For each trigger in a relation graph, the applicable elasticity rule is then selected. Since an elasticity rule may contain multiple alternative actions, based on the current situation an optimal action among these alternatives is selected. The actions of two unrelated triggers do not impact each other. Therefore, actions of elasticity rules invoked by triggers in different relation graphs can be executed in parallel. For each relation graph, based on the relations between the triggers the optimal actions of the selected elasticity rules are correlated using a set of action correlation meta-rules. With the reconfiguration of the system, the values of the thresholds may also be updated. It is worth mentioning that this is a periodic process; at the end of each monitoring time interval, the SLA compliance model is updated according to the new measurements and possibly new/terminated SLAs and thresholds models.

The rest of this paper is organized as follows: In Section 2, we review the related work in SLA management and dynamic system reconfiguration. In Section 3 we provide some background knowledge about the application domain used throughout this paper. In Section 4, we discuss our approach for checking for SLA compliance and generating triggers for system reconfiguration at runtime. In Section 5, we discuss our correlation approach to manage the application of elasticity rules at runtime. In Section 6, we conduct some preliminary experiments to evaluate the efficiency of the proposed framework. We conclude in Section 7.

2. Related Work

In this section, we review the work related to our management framework in three aspects: SLA compliance management, trigger correlation and dynamic system reconfiguration. As most of the related work focuses only on one aspect of our framework, we organize this section into three sub-sections, one for each aspect.

2.1. SLA Compliance Management

Monitoring a system, collecting measurements and assessing them against the SLAs are necessary tasks for reconfiguration based on workload variations.

In [9] and [10], authors propose the rSLA framework to monitor SLAs during their life cycle. The framework consists of three main components: rSLA language, rSLA Service and Xlets. The rSLA language is defined to describe SLAs and service metrics as well as evaluation conditions. In rSLA, it is described how the metrics should be measured and composed to define Service Level Objectives (SLOs). It can also be specified what actions should be taken when the SLAs are violated. The authors of SLAs are the providers. As the SLAs are agreements among the providers and customers and need common understanding of the terms, considering all of the above terms may be cumbersome or not important for the customers as they may not care how the measurements are obtained. The rSLA Service checks for SLA compliance. Xlets provide standard interfaces for monitoring the system and reporting measurements.

In [11], to detect violations with respect to response time, a timed automata is used. The work in [12] is closely related to the SLA management of our framework. The goal in [12] is to detect individual SLA violations only, while in our case we want to avoid the potential SLA violations and achieve this goal with the minimum amount of resources needed according to the workload variations. In [12], for each SLA parameter

an OCL constraint is defined to check if the measurement has reached the threshold. When a new parameter is added to an SLA, a new OCL constraint for the violation detection has to be defined as well, which is not the case in our framework.

In [13], authors demonstrate their architecture for SLA violation detection at the application level. In [13], thresholds for SLA parameters such as response time and throughput are defined. According to their architecture, when a user requests for a service, based on the SLAs at first it is checked if the request is coming from the right customer. In the next step, based on the requests, the tasks are generated and executed. There are multiple monitoring agents that collect application level measurements. They define LoM2HiS [14] to map the monitoring measurements to SLA parameters. The measurements are passed to the SLA management framework where it is checked if the value of a threshold has been reached or not.

2.2. Trigger Correlation

In the studies which are threshold based, trigger correlation and the coordination of the related actions which are important for the dynamic reconfiguration of systems are hardly considered. In the current literature, trigger correlation is discussed extensively for fault management of distributed systems and networks where an error caused by a fault is propagated through many related objects and potentially large volume of triggers are generated for the same fault. In these studies, a reported fault is an event which triggers an action and correlation is used as a reduction technique to filter the symptoms and identify the root cause fault; while in our approach triggers are not necessarily symptoms and should not be simply eliminated because the allocation of resources to one entity does not necessarily mean the allocation of resources to another entity even if they are related.

In [15] and [16], authors come up with different correlation graphs based on which the triggers are correlated. The proposed correlation graphs only capture the paths where a fault can propagate. To build a correlation graph in [15], for each entity of the system the faults that can originate from the entity, the relationships that the entity has with other entities and the faults that propagate along these relationships are specified by an expert. From these elements, a causality graph is inferred. A node in a causality graph is an event which can be a symptom or a root cause fault. A causality graph may have information which may not contribute to the correlation analysis like a cycle. A correlation graph is deduced from a causality graph by eliminating the cycles and aggregating each into a single event or by pruning indirect symptoms (i.e. the symptoms that are not caused by a root cause fault directly). The correlation process in [15] is based on an encoding technique where the events are represented by a code. To code a correlation graph, the root faults in the graph contain bits where each bit corresponds to a single symptom in the correlation graph. For example, for a graph with three symptoms, the code length will be three. The value of 1 for a symptom for a root fault indicates that the root fault causes the corresponding symptom. Therefore, the event correlation process becomes finding problems in the correlation graph whose codes optimally match the observed symptoms. In the case that similar symptoms are caused by different root faults, the root cause is not distinguishable. In [16], the correlation graph is obtained from the dependencies between the functionalities of the managed system. Therefore, the nodes of the correlation graph in [16] are the functionalities and the edges are the dependencies. When a fault occurs in a component, the components which communicate with that component are also affected and similar faults will be reported for them. In [16] they use their proposed correlation graph to identify the component whose failure caused a large number of symptoms. In [15], [16] trigger correlation is the aggregation of similar triggers that report the same fault.

2.3. Dynamic System Reconfiguration

There are many papers focusing on elasticity management. They reconfigure the system by adding/removing VM instances or by scaling up/down the VM instances at runtime [17]-[23]. Among them,

the study in [22] considers elasticity rule correlation to some extent. In our paper, we propose a finer grain approach which not only adds or removes resources when it is required but also reorganizes them (e.g. by changing the active and standby assignment roles) for better resource utilization while taking into account the service availability.

In [22], when the load on VMs increases, the VMs are scaled up. In their approach, they considered action correlation for a specific case where hosted VMs with the same supporting physical node need to be scaled up and the supporting node does not have enough resources for all the requests. To handle such conflicts, the authors use VM migration. To choose the candidate VM for the migration, they select the one for which the cost and the time of migration as well as the release of resources resulting from the migration are optimal. The accuracy of their approach depends on the weight they set for the different types of resources like CPU, memory, etc.

There has been some research in which they do not use thresholds as the points for allocating or deallocating resources. Instead, they use prediction techniques which take the previous workload and the resources utilized as input and forecast the future workload and resource requirements. In these studies, the previous workload is analyzed at different time intervals with a fixed window size to identify any repeating patterns in the workload [24]. Based on the pattern found, the future workload is predicted. Next, the system is scaled according to the defined policies if it is necessary. The size of the window and the accuracy of the prediction have significant impact on the efficiency of the scaling. These approaches try to solve the problem of when and how much to scale; but they do not specify how the system is scaled or the elasticity rules based on which the system is scaled.

In [25], [19] and [20] workload variations are predicted by machine learning. In [19] based on the predicted workload, penalty of violating SLAs and the cost of adding VM instances, it is decided if VM instances should be added or removed. In [20], online machine learning is used as a decision maker to add or remove Virtualized Network Functions (VNFs) [26]. Similar to other online learning approaches, the initial performance when the system is learning can be low. The performance can be worse especially when the workload is not even. Others such as [27]-[41] used queuing theory to model the cloud system based on the arrival rate of requests and other parameters such as mean service time (i.e. response time) and CPU load. They use this model to predict the response time or the load in the next time interval.

3. Background: System Configurations with an Example from the Domain of Interest

Systems are described and managed through configurations. In our domain of interest, a configuration describes the software and/or hardware entities, their corresponding types, the types of the services they provide and their relationships. In this configuration, a system is viewed from two perspectives: service side and service provider side. The entities of the service perspective (service entities) are workload units. A workload unit represents a chunk of workload (e.g. a certain amount of request per second) associated with the provisioning of a service. Workload units are logical resources. Service provider entities are software and/or hardware entities which collaborate to provide the service. Thus, they are the tangible resources. In this paper, software entities are represented as serving units hosted by computing nodes (i.e. the physical or virtual computing nodes). To provide the services, workload units representing them are assigned to the serving units at runtime.

To provide and protect highly available services [28], provider entities are deployed in a redundant manner. Two or more serving units form a work pool. A node group is used to host the serving units of a work pool. Depending on the redundancy model, a workload unit may have one or more active and zero or more standby assignments to different serving units of a work pool. A serving unit with an active assignment

provides the service represented by the workload unit. A serving unit with a standby assignment does not provide the service, but it is ready to become active for the workload unit in a minimum amount of time.

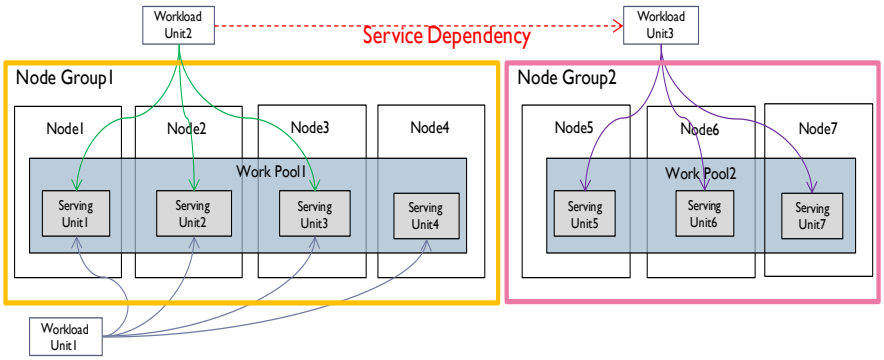

Fig. 3. An example configuration at runtime.

The work discussed in this paper is applicable in the context of the Service Availability Forum [29] and in any other domain where the service and service provider perspectives are described explicitly in the configuration.

In this paper, we also assume a maximum workload for which the system has been designed and that the required software for handling this maximum workload is available anywhere it may be executed. When the workload is not at its maximum, some serving units are not assigned any workload units – they are spares – and may be terminated to reduce resources/power consumption. When the workload increases and more serving units are needed to provide the service, the spare serving units are instantiated and assigned workload as needed.

Fig. 2 illustrates an example configuration at runtime. In this configuration, there are two work pools (*Work Pool*$_1$ and *Work Pool*$_2$) which are protecting three workload units (*Workload Unit*$_1$, *Workload Unit*$_2$ and *Workload Unit*$_3$). The serving units of *Work Pool*$_1$ can be hosted only on the nodes of *Node Group*$_1$ (*Node*$_1$, *Node*$_2$, *Node*$_3$ and *Node*$_4$) and the serving units of *Work Pool*$_2$ can be hosted only on *Node*$_5$, *Node*$_6$ and *Node*$_7$ composing *Node Group*$_2$. As shown in the figure with the service dependency relation, the provisioning of the service represented by *Workload Unit*$_2$ depends on the provisioning of the service represented by *Workload Unit*$_3$. In this example, each assignment of *Workload Unit*$_2$ requires one assignment of *Workload Unit*$_3$.

4. SLA Compliance Management

In this section, we introduce the SLA compliance management portion of our management framework. As mentioned earlier, SLA compliance management aims at generating triggers whenever there is a potential SLA violation (i.e. it is probable that an SLA is going to be violated in the next time interval of monitoring) or the resource utilization is getting low.

4.1. Modeling for SLA Compliance Management

In this section, we discuss the metamodels we have defined as well as the reasoning behind these definitions.

4.1.1. The SLA metamodel

To model the SLAs, we use the SLA metamodel introduced in [30]. As shown in Fig. 3, each SLA is an agreement between a service provider and a customer and identified by an ID. A third party may also participate in the contract to validate the agreed terms. SLAs aim at describing the services that the provider agrees to provide with specific Quality of Service (QoS). The metaclass *SlaParameter* captures the different types of QoS included in the SLAs. The agreed values are represented by *maxAgreedValue* and

minAgreedValue in the metamodel. For example, for the SLA parameter availability, the *minAgreedValue* represents the minimum percentage of the time that the provider guarantees the service is available. For the SLA parameter *DataRate*, the *maxAgreedValue* represents the maximum number of requests per second the customer may send for the specific service, and the *minAgreedValue* represents the minimum amount of service that the provider agreed to provide. If service providers or customers fail to meet the agreed terms, they may be subject to penalties. The QoS included in the SLAs should be either measurable by the monitoring system or reported by the constituent components of the system; otherwise, it cannot be included in SLAs. Customers may want to specify at which frequency the SLA parameters should be measured. This customization is represented by *SlaMetric* metaclass. However, the frequency specified by the user should be compatible with the capability of the monitoring system. An SLA is applicable for specific time duration and has a cost that customer agrees to pay.

Fig. 4 shows two SLA models. The *FTP* functionality is sold to customers C_1 and C_2 with different quality of service. The service type *FTP* is represented with gray rectangle and the SLA parameters with rounded squares. The dashed lines show *RelatedTo* relations.

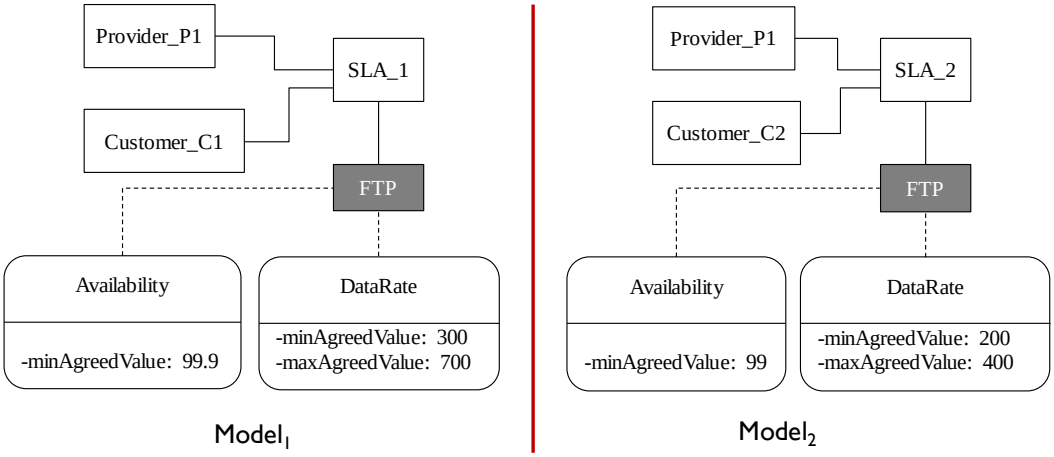

Fig. 4. Two different SLAs.

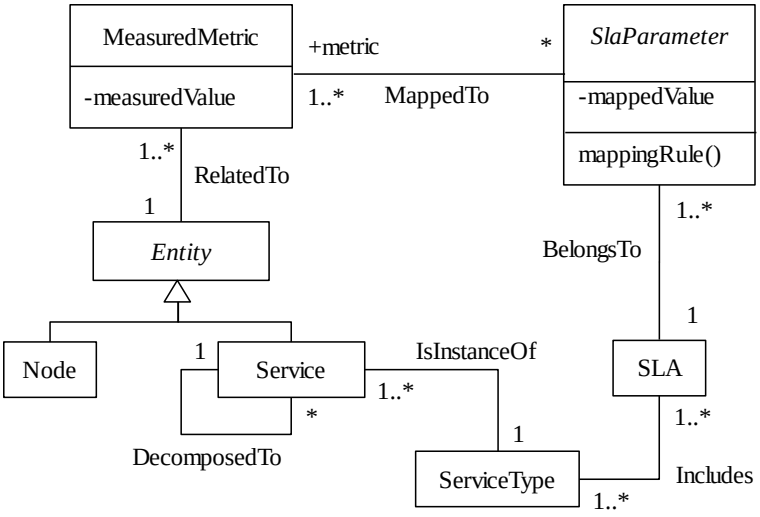

Fig. 5. The measurements metamodel.

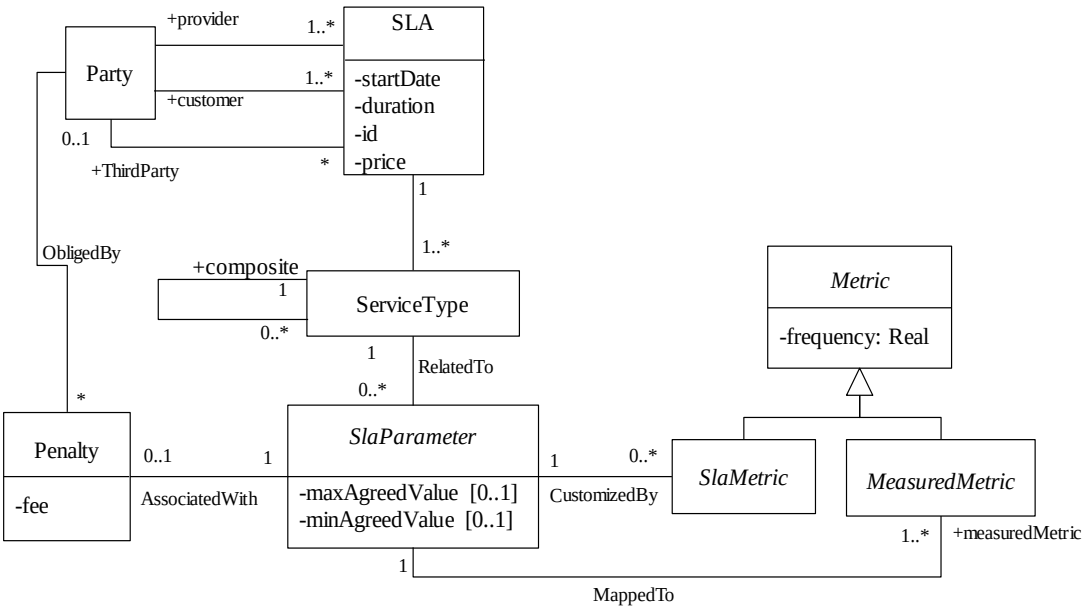

Fig. 6. The SLA metamodel

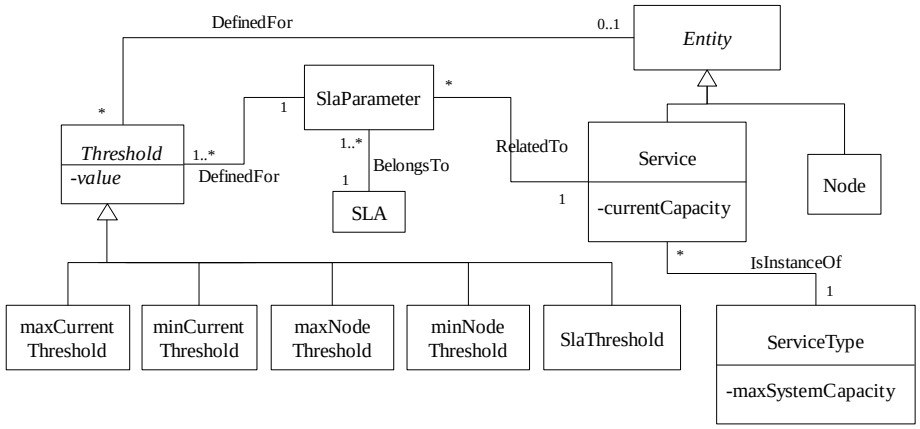

Fig. 7. The threshold metamodel.

4.1.2. The measurements metamodel

A monitoring system collects the metrics of interest. These measured metrics are related to a computing node or a service. The Service metaclass represents instances of a service type, which—in the above explained domain—are represented by workload units. Some of the metrics (e.g. service up/down time) and the SLA parameters perceived by the customers (e.g. availability of service) are not at the same level. To bridge the gap between the measured metrics and the SLA parameters, we have defined mapping rules. Fig. 5 shows the measurements metamodel. The attribute mappedValue represents the value of such mapped measurements. As an example, the mapping rule for mapping service up time and down time to service availability is presented:

Context Availability :: mappingRule ( )
*self.mappedValue = self.metric -> select (c|c.oclIsTypeOf(MeasuredUpTime)).measuredValue-> at(1)/(self.metric-> select(c|c.IsTypeOf(MeasuredUpTime)).measuredValue -> at(1) + self.metric-> select(c|c.oclIsTypeOf(MeasuredDownTime)).measuredValue-> at(1))*100*

Fig. 6 shows an example measurement model. The measured metrics are represented by rounded squares in light gray. The dotted and dashed lines represent *BelongsTo* and *RelatedTo* relations respectively.

4.1.3. The threshold metamodel

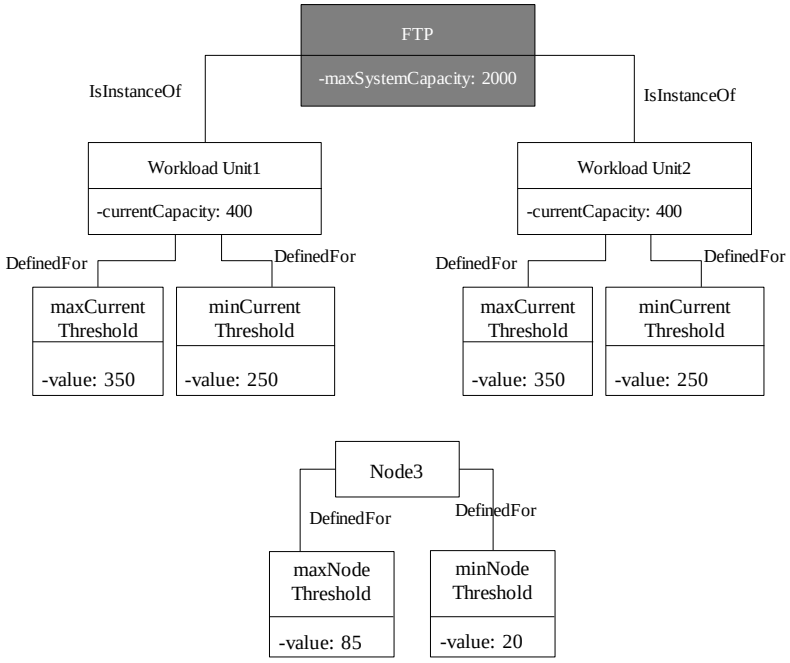

Fig. 8. An example of thresholds model.

In this paper, we use thresholds. When they are reached, actions are required to avoid SLA violations or low resource utilization. Fig. 7 shows the threshold metamodel. Some of the thresholds are related to all customers' (aggregate) resource usage (i.e. thresholds defined on nodes) while others are related to individual SLAs (e.g. service availability). The attribute *currentCapacity* in the *Service* metaclass specifies the current capacity of a service entity (i.e. a workload unit) for handling the workload of a specific customer. The attribute *maxSystemCapacity* is determined at the design phase as the maximum capacity the system can be expanded to for a specific service type without major changes (e.g. upgrade/redesign). For nodes and service entities (i.e. represented as workload units), two thresholds (maximum and minimum thresholds) are defined: The maximum limit represents the load of the node or the workload unit without SLAs violation. If no action is taken SLA violation is likely to happen within the next measurement period. The minimum limit represents the load of the node or the workload unit for efficient usage of the resources; otherwise they are wasted. In the following the different types of the thresholds are explained:

- *maxNodeThreshold* and *minNodeThreshold*: To avoid SLA violations because of node limitations, e.g. trying to load a node beyond its capacity, we define the *maxNodeThreshold* point at which we may allocate more resources to the node (e.g. virtual machine, hyper scale system [31]), add more nodes to the system or rearrange the assignments (i.e. the relation *load* < *maxNodeThreshold* should be always respected). To avoid wasting resources, the *minNodeThreshold* is defined. The *maxNodeThreshold* and *minNodeThreshold* are vectors that take into account different types of node resources (e.g. CPU, RAM, etc.).
- *maxCurrentThreshold* and *minCurrentThreshold*: For each service, the system is dimensioned dynamically with a *currentCapacity* to handle the workload of a certain customer. In order to avoid SLA violations, i.e. workload exceeding *currentCapacity*, we define a *maxCurrentThreshold* point with *maxCurrentThreshold* < *currentCapacity*. Not to waste resources, we also define a *minCurrentThreshold*. Unlike resource provisioning, the resources should be released in a reactive

manner. The values of *maxCurrentThreshold* and *minCurrentThreshold* are determined by different functions which take into account the current capacity, the measurement period, the average reconfiguration time and the predicted workload.

- *slaThreshold:* SLA parameters like service availability are set on a per customer basis. Therefore, to avoid SLA violations, we need to watch the SLAs separately using a slaThreshold for each QoS of each SLA.

Fig. 8 shows an example of threshold model. In this figure, different thresholds for services of type *FTP* as well as $Node_3$ are defined.

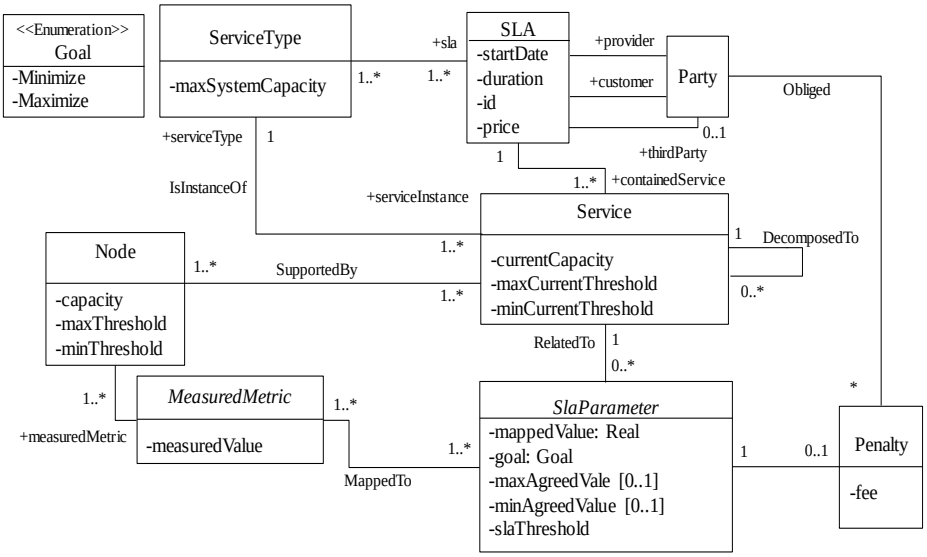

Fig. 9. The SLA compliance metamodel.

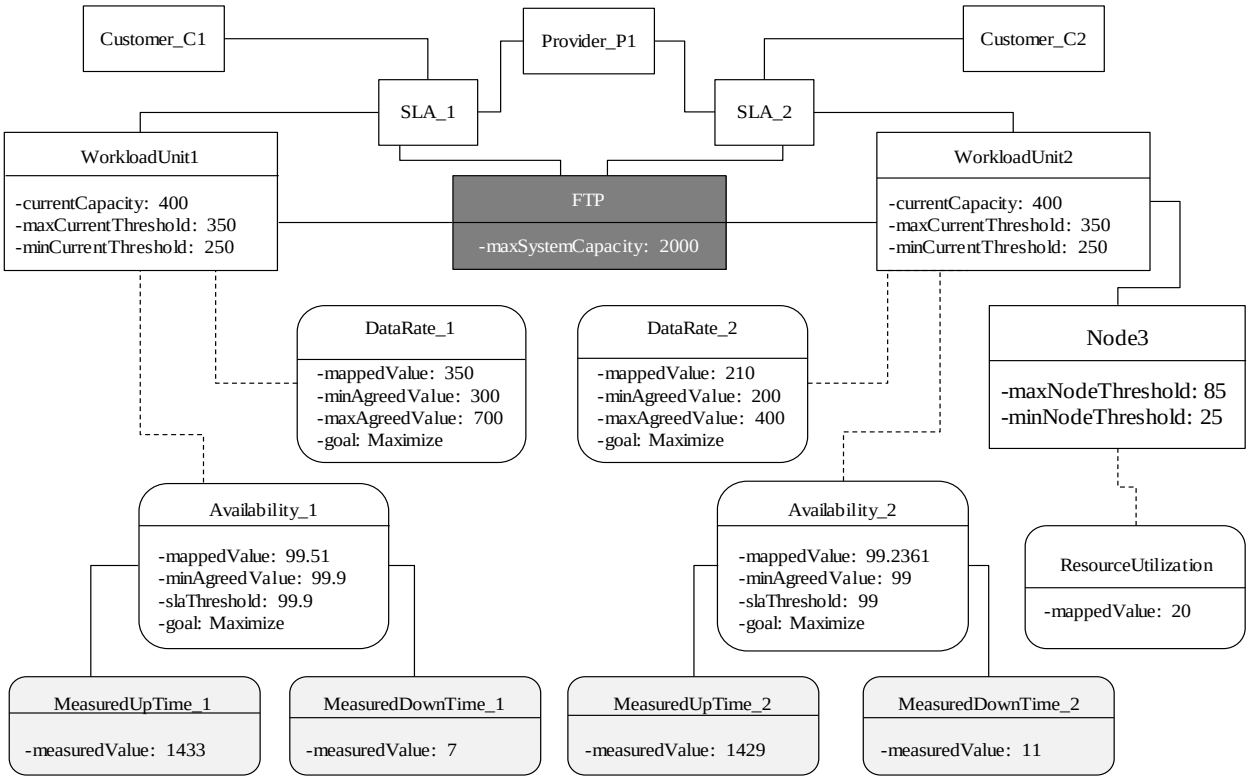

Fig. 10. An example of SLA compliance model

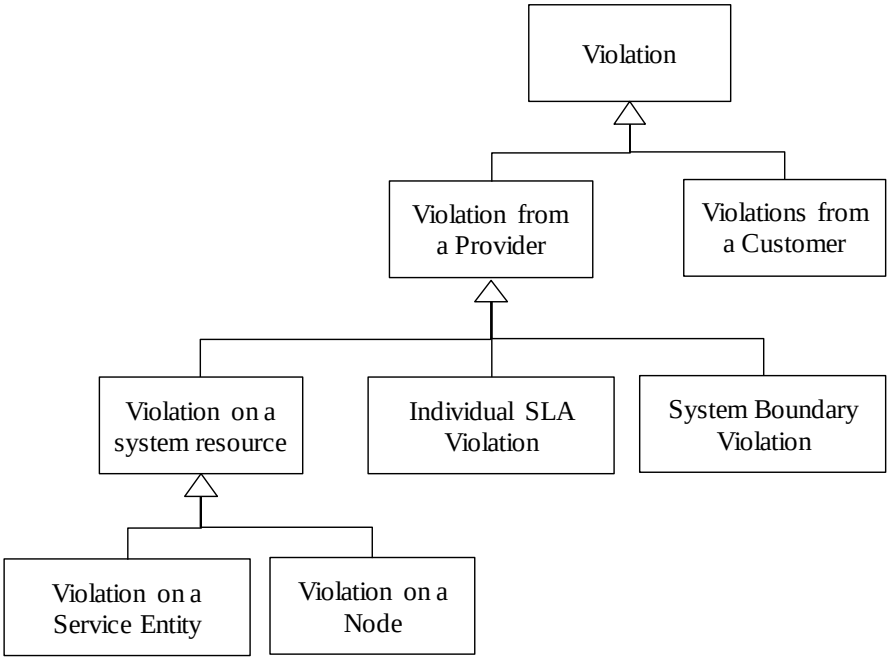

Fig. 11. Different types of violations.

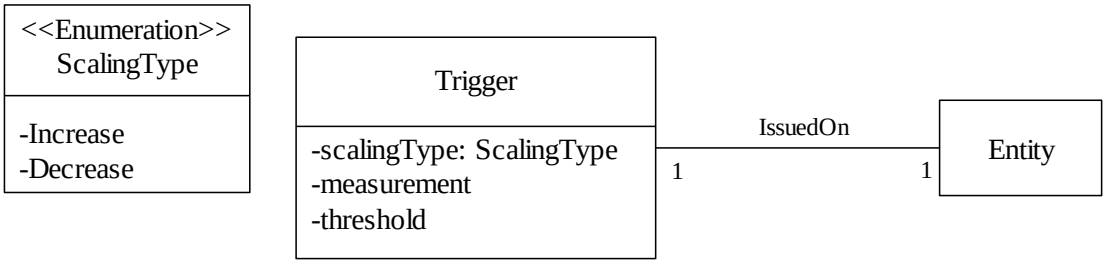

Fig. 12. The trigger metamodel.

4.1.4. The SLA compliance metamodel

An SLA compliance model is the combination of all SLA models, thresholds model, part of the current configuration and the mapped measurements [30]. The main reason for merging all SLA models into one model is that we want to be able not only to avoid violations of each individual SLA but also to trigger elasticity rules which are related to all customers' resource usage.

The SLA compliance metamodel is shown in Fig. 9 and an instance model of it is shown in Fig. 10. Different services of the same service type with the same or different QoS (i.e. represented by *SLAParameter*) are generally offered to multiple customers. The *MeasuredMetric* metaclass represents the measurements that are collected from the monitoring system per service for each customer or per node of the system. When an SLA parameter is not respected, the *RelatedTo* relation indicates which service of which SLA has been violated.

The attribute goal of an SLA parameter specifies the parameter's optimization goal. For some SLA parameters, like service availability, the optimization goal is maximization while for others like response time, the goal is minimization. We categorize OCL constraints for SLA violation avoidance based on these optimization goals. When a new SLA parameter is introduced and taken into consideration, there is no need to define a new OCL constraint as long as its optimization goal fits into one of the aforementioned categories.

According to UML [32], a constraint is a model element that can have a name (it is optional) and consists of an invariant (i.e. a Boolean expression that must be evaluated to true for the constraint to be satisfied), constrained elements (i.e. a set of elements required to evaluate the constraint) and a context (i.e. the model element on which the constraint is defined). Therefore, an OCL constraint is defined as a tuple of *(name, context, ConstrainedElements, invariant)*. To define an OCL constraint, its different elements should be specified.

Depending on the type of SLA violation, different OCL constraints are defined in the SLA compliance metamodel. As shown in Fig. 11, an SLA can be violated by the provider or by the customer. Violations by providers are categorized into: violations issued on system resources, which can be violations on service entities (i.e. represented as workload units in our domain) or nodes and lead to the generation of triggers for dynamic reconfiguration; individual SLA violations can be related to the design of the system and system boundary related violations. The focus of this paper is on the triggers, which lead to dynamic reconfigurations (i.e. violations on system resources). When the trigger is of this type, the generated elasticity rules are applied to reconfigure the system dynamically. The different types of violations are defined as follows:

SLA Violation Avoidance from a Provider

To avoid SLA violations from a provider, the OCL constraints are defined on attributes of the SLA compliance metamodel as follows:

- *Service Entities Violations:* In order to avoid SLA violations, the relation *workload represented by a workload unit < maxCurrentThreshold* must be respected. If the workload of a workload unit exceeds the threshold, a potential violation is detected and a trigger should be generated to either increase the workload unit capacity to a new *currentCapacity* for which a new *maxCurrentThreshold* is defined, or to add a new workload unit. To check if the system needs to be scaled due to workload increase, the following OCL constraint is defined. This OCL constraint is named as *Increase*. Later we use this name as the scaling type of the generated trigger to indicate that the violation was due to increase in the workload (see Section 4.2.2).

  Context Service
  Inv Increase: maxAgreedValue > Service.allInstances() -> select(s|s.sla = self.sla and s.serviceType = self.servicetype) -> collect(currentCapacity) -> sum() implies
  self.maxCurrentThreshold > (self.slaParameter -> select (p|p.oclIsTypeOf(DataRate)).mappedValue -> at(1)

  Not to waste resources we define an OCL constraint to check if the relation *workload ≥ minCurrentThreshold* is respected. This OCL constraint is named as *Decrease* because its violation indicates the resources of the system are excess and the system should be shrunk.
  Context Service
  Inv Decrease: self.minCurrentThreshold ≤ (self.slaParameter -> select (p|p.oclIsTypeOf (DataRate)).mappedValue -> at(1)

  Considering the SLA compliance model in Fig. 10, the workload of the customer C_1 represented by $WorkloadUnit_1$, i.e. 350 requests per second, reaches the *maxCurrentThreshold*. Therefore, its corresponding OCL constraint named as *Increase* is violated which indicates more resources for handling the workload of this customer should be allocated.
- *Node Violation*: Although services are supported by nodes and service side violations (i.e. increase in the workload) usually lead to the violations on the underlying nodes, we still need to distinguish between violations on the services and on the nodes. The reason is that when the node hosts multiple services and the workload increase of an individual service does not reach its threshold, the

workload increases of the hosted services will accumulate on the hosting node and the total load may cause violation. This happens when the distribution of entities among the nodes is not optimal. Therefore, to detect SLA violations because of node limitations, the relation *load* < *maxNodeThreshold* should be respected. Similar to the *maxNodeThreshold*, the *load* represents the load on the different types of resources which is measured by the monitoring system. In addition, the relation *load* ≥ *minNodeThreshold* should be respected in order to not to waste resources. The load imposed on the node by service requests is estimated at runtime by a function. This function takes into account parameters that characterize the service workload as well as the node (e.g. the types of workload the node currently supports, the operating system, etc.).

Context Node
Inv Increase: self.maxNodeThreshold > (self.measuredMetric-> select (p|p.oclIsTypeOf(ResourceUsage))-> at(1).measuredValue)

Context Node
Inv Decrease: self.minNodeThreshold ≤ (self.measuredMetric-> select (p|p.oclIsTypeOf(ResourceUsage))-> at(1).measuredValue)

Similar to the OCL constraints defined on the services, these OCL constraints are also named as *Increase* and *Decrease* as their violations indicate if the load on the node has to be decreased or if the node or some resources of the node are in excess. Considering the SLA compliance model in Fig. 10, the current load on $Node_3$ is 20 which is less than the *minNodeThreshold* which is 25. Therefore, for $Node_3$ the OCL constraint with the name of *Decrease* is violated.

- *Individual SLA Violation*: Some SLA parameters behave similarly with respect to violation. Some of them like availability and throughput for which a higher value is preferable (i.e. the attribute *goal* is equal to *Maximize*) will be violated by a service provider when in the SLA compliance model, the experienced quality is less than their defined *slaThreshold* (i.e. the relation *mappedValue* > *slaThreshold* must be respected all the time if *goal=Maximize*); while for others like response time, the violation happens from the provider when the measured response time is greater than the *slaThreshold* (i.e. the relation *mappedValue* < *slaThreshold* must be respected if *goal=Minimize*). We use OCL constraints as follows to define these restrictions:

  Context SlaParameter
  Inv Maximize: Self.goal=Goal::Maximize implies self.mappedValue > self.slaThreshold

  Context SlaParameter
  Inv Minimize: Self.goal=Goal::Minimize implies self.mappedValue < self.slaThreshold

  Considering the SLA compliance model in Fig. 10, for customer C_1 the measured availability of *FTP* service is 99.51 which is less than its corresponding *slaThreshold* of 99.9; therefore in this example, the OCL constraint for availability, which has the goal of *Maximize* is violated.

- *System Boundary Violations:* Customers have periods of activity and inactivity; therefore, the customers may not use resources all at the same time. To make the most profit, providers may sell the same service to multiple customers. This is known as overbooking technique [33]. In this paper, we assume that the provider sells the service to the maximum number of customers such that minimum or no violation occurs and the revenue is the most. With overbooking, there is a risk that the customers want to use the resources all at the same time. When the value of *maxSystemCapacity* is reached, the system cannot be expanded further; thus the admission control/overload protection

needs to be engaged to protect the system from overload. In addition, the service provider may decide to redesign the system with new user requirements if some SLAs are violated. The following OCL constraint detects the potential SLA violation when the system reaches its maximum capacity:

Context ServiceType
Inv SystemBoundary: maxSystemCapacity = self.serviceInstance -> collect(currentCapacity)->sum() implies self.sla -> forAll(sla:SLA| let services: sla.containedService -> select (s|s.serviceType = self) in services -> collect(currentCapacity) -> sum () < services-> at(1).maxAgreedValue implies services -> forAll(srvc:Service|srvc.maxCurrentThreshold > srvc.slaParameter -> select (p|p.oclIsTypeOf(DataRate)) -> at(1).mappedValue))

Customer Side SLA Violation Detection

Unlike violations from providers, the violations from customers cannot be avoided. However, it is important to detect any service overuse by a customer to take the appropriate actions (e.g. charging or dropping the extra workload). By the following OCL constraint it is detected if a customer has violated an SLA:

Context DataRate
Inv CustomerViolation: self.mappedValue ≤self.maxAgreedValue

4.1.5. The trigger metamodel

To reconfigure the system at runtime, a trigger is issued on a configuration entity to reconfigure the entity or add (e.g. by instantiating the configuration entity) or remove it. Fig. 12 shows the trigger metamodel. In this metamodel, the attribute *scalingType* can have the value of either Increase or Decrease and specifies whether an action to increase or decrease the resources is needed. The attributes measurement and threshold represent the measurements from the monitoring system and the current threshold value that has been violated. The values of threshold and measurement are used to determine the amount of resources that should be given to or released from the entity to resolve the violation of the received trigger. For example, if the current load on a node is 90% and the threshold on the node is 85%, the load of the node should be decreased by at least 5% to resolve the issued trigger.

4.2. SLA Compliance Management Process

In this section, the different tasks of the SLA compliance management process are elaborated in more details.

4.2.1. Building/ and updating the SLA compliance model

To build the SLA compliance model all SLA, measurement and the threshold models are combined at runtime. We use the Atlas Transformation Language (ATL) [34] transformation to implement this process. When any of the measurement or threshold models are updated or new/old SLAs are added/terminated, the SLA compliance model is updated too. New measurements arrive at the end of each measurement period. The measurement period should be long enough to process the previous measurements and reconfigure the system as necessary before the arrival of new measurements. After each reconfiguration, the thresholds model may also be updated before the new measurements arrive. Although new SLAs arrive or existing ones can be terminated at any time, we update the SLA compliance model at the end of each time interval. For illustration purposes, assume a new SLA (representing a new customer) arrives when a reconfiguration is being performed. If we update the SLA compliance model as soon as the new SLA arrives, the previously generated triggers have not been resolved yet; at the validation of the updated SLA compliance model the same triggers may be regenerated. Handling the same triggers may cause instability in the system.

When a new SLA for a new customer arrives, all the SLA elements are added to the current SLA compliance model. Since no workload unit is yet assigned to represent the workload of the new customer, for each service

type contained in the new SLA, a Service model element with the current capacity of 0 is created in the SLA compliance model. This added element represents a workload unit which needs to be added to represent the workload of the new customer. The validation of the updated SLA compliance model leads to the generation of a trigger to add that workload unit. Similarly, when an SLA is removed, the elements related to only this SLA should be removed from the SLA compliance model together with their measurements and thresholds. This is achieved with a different transformation that takes the SLA to be removed and the SLA compliance model as input and generates a new SLA compliance model. In ATL language, the number of input and output models cannot be arbitrary; therefore, to add or remove multiple SLAs, we execute the corresponding transformation multiple times as required. In the prototype implementation the addition and removal of SLAs are done offline.

4.2.2. Trigger generation

As mentioned earlier, the validation of SLA compliance model may lead to the generation of triggers. An SLA compliance model is valid when all the constraints of its metamodel are satisfied.

To generate a trigger, its different elements should be specified based on the constituent elements of the violated OCL constraint. In the SLA compliance metamodel, the constraints for the scaling of the system are defined on nodes and services. These OCL constraints are violated when there are not enough resources for such entities or their resources are in excess. Therefore, the entity on which a trigger is generated is the node or the service for which the respective OCL constraint is violated.

The constrained elements based on which the invariant is defined are the measurement and the threshold. The constraint checks if the measurement has reached the value of the current threshold. If the value of the threshold reaches (i.e. the constraint is violated), the constrained elements of the violated OCL constraints are extracted to specify the measurement and threshold elements of the generated trigger.

We use the names of the constraints as the *scalingType* of the generated triggers to initiate resource allocation (when the name is *Increase*) or release of surplus resources (when the name is *Decrease*).

5. Trigger Correlation and Dynamic System Reconfiguration

In this section, we explain our correlation approach to manage the application of elasticity rules at runtime based on the generated triggers.

5.1. Modeling for Trigger Correlation and Dynamic Reconfiguration

5.1.1. The elasticity rule metamodel

We model the elasticity rules using the metamodel introduced in [35]. As shown in Fig. 13, the elasticity rules are defined for types of configuration entities because entities that belong to same type share common characteristics and are subject to the same type of actions. The metaclass *EntityType* in Fig. 13 represents the type of the configuration entities on which the elasticity rule can be applied. In this paper, for each entity type two elasticity rules are defined: one for the expansion and one for the shrinkage (contraction) of the system. The attribute *scalingRule* of an elasticity rule specifies if by the application of the rule the resources of a type are allocated (when the value of this attribute is *Increase*) or released (when the value of this attribute is *Decrease*).

An elasticity rule consists of alternative actions each of which is performed in a different situation. Therefore, an action can be associated with different Boolean expressions which constrain the applicability and feasibility of the action. The condition under which the action is applicable is represented by the *Condition* metaclass. In another word, for an action to be considered for execution at runtime, its associated condition must evaluate to true. Otherwise, the action is not applicable in the current situation and will not be considered for execution at all.

To execute an applicable action at runtime, the execution of other actions as prerequisite may be required. Therefore, Boolean expressions as prerequisites are defined to check if the action is feasible in the current situation (i.e. the prerequisites are met). As shown in Fig. 13, a prerequisite is associated with a prerequisite trigger. At runtime, if an applicable action is not currently feasible (i.e. the prerequisite is evaluated to false) a prerequisite trigger is generated to invoke the prerequisite elasticity rules by which prerequisite resources are allocated first. For illustration purpose, let us consider an elasticity rule for a resizable VM where one action of the elasticity rule is adding virtual resources to the VM. A resizable computing node such as a resizable VM has a maximum capacity that it can expand to. In this example, the condition (i.e. applicability) is defined as a Boolean expression that checks if the VM has not reached its maximum capacity yet. The associated prerequisite (i.e. feasibility) checks if the hosting physical node has enough resources for the VM to expand. In this example, if the VM has reached its maximum capacity (i.e. the condition is not met), resources cannot be allocated to the VM regardless of the availability of host resources (i.e. the prerequisite). In the case that the VM has not yet reached its maximum capacity (the action is applicable) but the physical host does not have enough resources (the prerequisite is not met), a prerequisite trigger on the physical host is generated to execute the physical host elasticity rule first. The action on the physical host can be the addition of physical resources to the host if it is applicable (e.g. hyperscale data center systems like Ericsson HDS 8000 [31]) or releasing physical resources by moving out some services or migrating other VMs to other hosts.

In the case that an entity (i.e. dependent) requires resources of another entity (i.e. sponsor), it is less likely that triggers will be generated on both entities during the same period of monitoring time (usually a trigger on the dependent is generated before a trigger on the sponsor as the dependent is the bottleneck). The action contained in the elasticity rule of the dependent has a prerequisite which checks the capacity of the sponsor and if needed, the sponsor is reconfigured as well even if there is no trigger for it. This is a proactive action which prevents, in the near future, the generation of other triggers related to the sponsor.

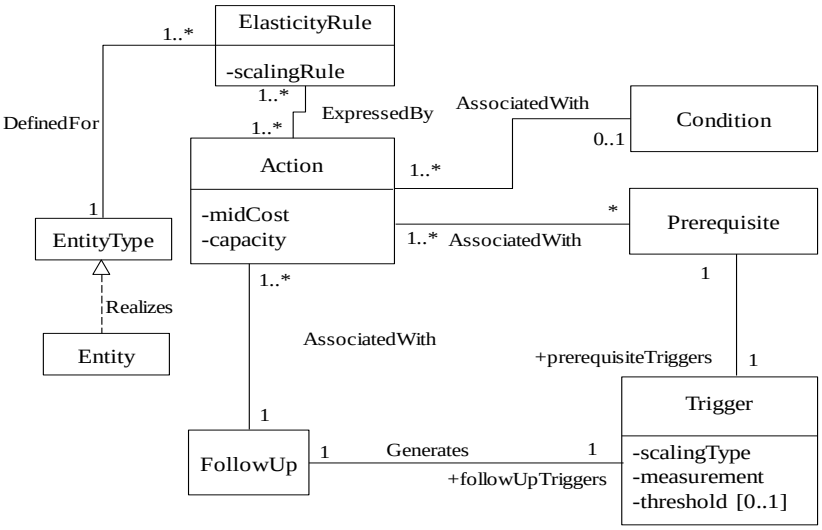

Fig. 13. The elasticity rule metamodel.

In contrast to prerequisites, a follow-up is a Boolean expression which is evaluated after the execution of an action and it checks if there are unused resources. If the follow-up is evaluated to true at runtime, a follow-up trigger should be generated to remove excess resources. For instance, after the removal of workload units,

a follow-up trigger may be generated to remove unused resources (e.g. serving units and nodes) if there are any.

The execution of an action imposes a cost. In this paper, the attribute *midCost* represents an approximate cost of the action and its value is the median of the minimum cost (where all the prerequisites are met and is represented as attribute *cost* in the metamodel) and the maximum cost (where none of the prerequisites are met and all prerequisites are invoked and it is the sum of costs of their actions).

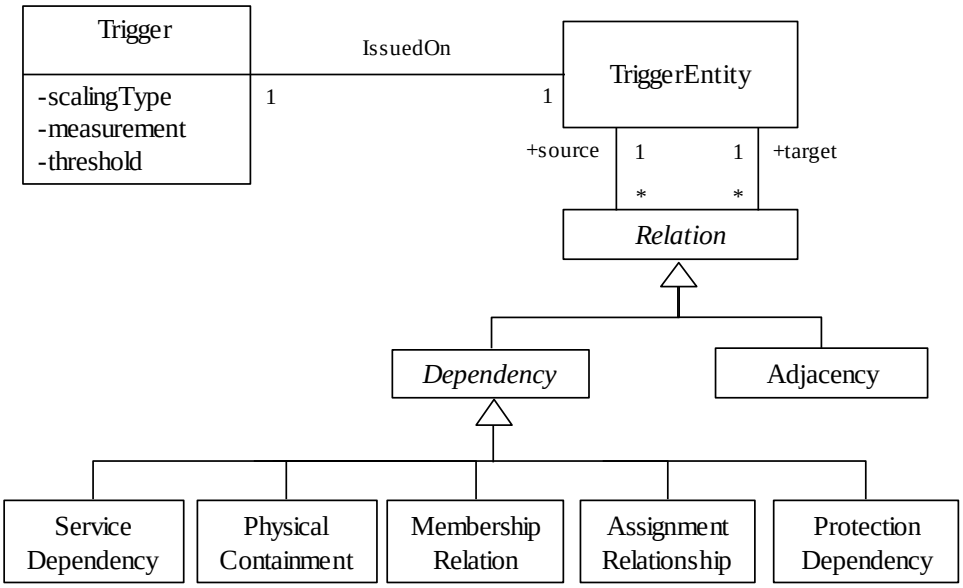

Fig. 14. The relation graph metamodel.

5.1.2. The relation graph metamodel

When the triggers are correlated, a set of relation graphs are generated [36]. Later, we use the generated relation graphs to correlate the actions of applicable elasticity rules (see Section 5.2). Fig. 14 shows the metamodel for the relation graph description. Each relation graph consists of some triggers and relations between them. As shown in the figure, the relation between the triggers is categorized into adjacency and dependency relations. The dependency relation is categorized further into service or protection dependency, assignment relationship, membership relation and physical containment. The different types of relations are explained in Section 5.2.1 in more details.

5.2. Trigger Correlation and Dynamic Reconfiguration Process

5.2.1. Trigger correlation

Triggers are related to each other based on the relations existing between their corresponding configuration entities. As summarized in Fig. 15, the relations between configuration entities can be of different types and categorized into two categories, dependency relations and adjacency relation. The first group consists of directed relations while the second defines a symmetric relation. The different types of relations are defined as follows [36]:

Service Dependency: Dependency relation exists between services when the provision of one service (i.e. dependent entity) depends on the provision of another service (i.e. sponsor entity); therefore, the sponsor needs to be provided first to support the dependent.

Protection Dependency: This relation is defined between the active and the standby assignments of a workload unit.

Assignment Relationship: This relation is defined between a serving unit (i.e. service provider entity) and the workload unit (i.e. service entity) assigned to it.

Membership Relation: A membership relation exists between two entities when an entity is logically a member of a group represented by the other entity. For example, a node is a member of a node group.

Physical Containment: A physical containment relation exists between two entities if one entity is physically part of the other entity. In this case, the container entity (i.e. sponsor) provides resources for the contained entity (i.e. dependent). For example, a physical node is a container entity which provides resources for its hosted VMs as contained entities.

Adjacency: Two entities are adjacent when they are depending on (they are dependent of) the same sponsor. In this case, the common sponsor is called the common entity.

Correlated triggers are put into a relation graph where nodes are the triggers and edges represent the relations between these triggers. A set of relation graphs is automatically generated based on the triggers and the relations between the entities they correspond to. The algorithm correlating triggers generated for the same measurement period is provided in the appendix (*Algorithm1*). The entities of the triggers are looked up in the current configuration. Any relations between these entities are transferred to their associated triggers.

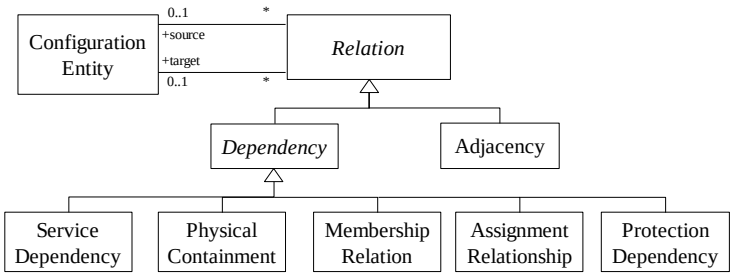

Fig. 15. Different types of relations between configuration entities

5.2.2. Elasticity rule selection and execution

After the correlation of triggers, the generated relation graphs are processed in parallel. For each relation graph, the applicable elasticity rules are selected and their actions are correlated. The correlated actions are executed on the fly. Therefore, in this paper, we do not build or evaluate different action paths before their execution.

In this section, we first introduce our approach of selecting the applicable elasticity rules given the correlated triggers; then we explain the selection of the optimal action among all the alternatives available for execution. We also introduce a set of meta-rules used for the correlation of the optimal actions of the selected elasticity rules.

5.2.3. Selecting applicable elasticity rules

The generated triggers invoke the applicable elasticity rules. On the one hand side, a trigger is issued on a configuration entity when any of its current threshold values is reached. On the other hand, an elasticity rule specifies the actions that can be taken on an instance of a given type to resolve a given type of threshold violation. Therefore, an elasticity rule is considered for invocation if the *entityType* for which the elasticity rule is defined is the same as the type of the entity on which the trigger was generated.

The *scalingType* of a trigger is either *Increase* to initiate resource allocation or *Decrease* to release surplus resources. On the other hand, the *scalingRule* of an elasticity rule is *Increase* if its actions add resources; and it is *Decrease* if its actions remove resources. As a result, for an elasticity rule to be applicable its *scalingRule* should be equal to the *scalingType* of the trigger (see *Algorithm 2* in the appendix).

5.2.4. Selecting the optimal action

In an elasticity rule, multiple actions may be specified. When such an elasticity rule is invoked by a trigger, among these alternatives an optimal action needs to be selected for execution depending on the condition and the prerequisite(s) met (see *Algorithm 3* in the appendix).

To be considered as optimal an action should at least be applicable in the current situation (i.e. the condition must be evaluated to true). It also needs to be feasible, so among the applicable alternatives, the feasible action with the least cost is selected if there is such. In the case that none of the applicable alternatives are feasible, the infeasible action with least *midCost* is selected for invocation and an appropriate prerequisite trigger is generated. If the cost of an infeasible action is less than the cost of a feasible action, we still select the feasible one because according to the current situation no prerequisite action is required, which more likely results in an efficient reconfiguration.

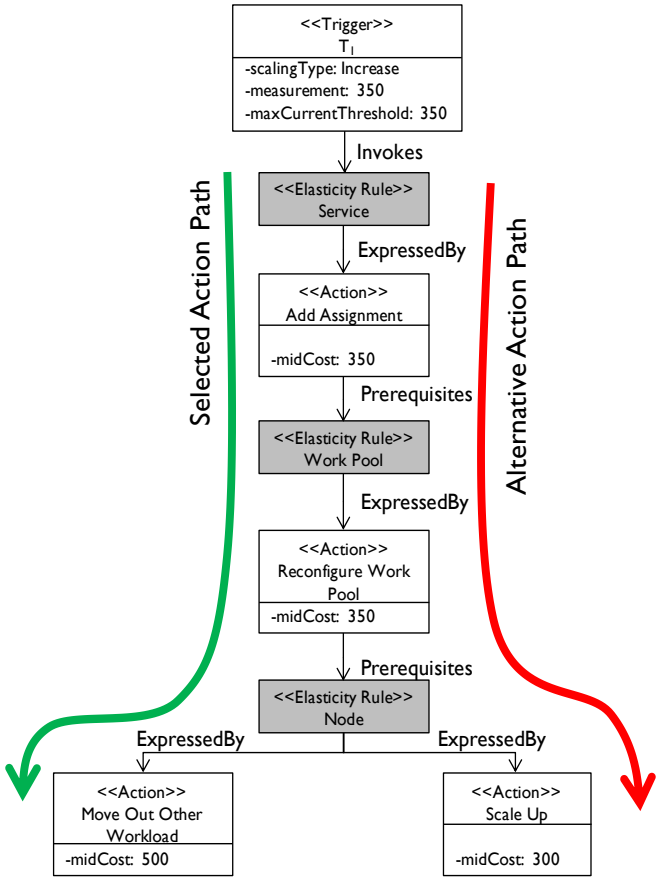

Fig. 16. An example of invoked action path

For illustration purpose, let us assume that due to workload increase, a trigger (T_1) for scaling the system is received from the SLA compliance management (see Fig. 16). Based on the *scalingType* of the trigger (increase) and the related entity, the applicable elasticity rule is selected and invoked. As shown in the figure, the actions of invoked elasticity rules have a cost. In the elasticity rule invoked by T_1, the defined action is the addition of an assignment. To add an assignment, the prerequisite is that there should be a serving unit in the work pool to which the new assignment can be assigned. If there is no such serving unit in the work pool (i.e. the prerequisite is not met), a prerequisite trigger is generated to initiate another elasticity rule for

reconfiguring the work pool by adding to it a new serving unit. However, a serving unit requires a node to host it. If there is no such node, a prerequisite trigger is generated again to invoke the corresponding elasticity rule. The actions contained in the node elasticity rule are: moving out some workload to other nodes with approximate cost of 500 which is feasible if there are enough providers for them; scaling up the node with approximate cost of 300, which is applicable if the node has not reached the maximum capacity yet and feasible if the physical node hosting the node has enough resources. Considering the current situation, assume that all of the contained actions are applicable, but only the first action which is moving out some workload is feasible. In this example, the first action is chosen as the optimal action as all the action's prerequisites are met and most likely it will result in an efficient reconfiguration in terms of cost.

In this example, trigger T_1 leads to the invocation of multiple elasticity rules where the invocation of one elasticity rule is a prerequisite for another one. The path resulted from the execution of an elasticity rule is called an action path.

5.2.5. Action correlation meta-rules

In this paper, we use higher level rules to govern the application of elasticity rules and execute their actions on the fly. We refer to these rules as action correlation meta-rules and their applicability is governed by the relations between the triggers [36]. *Algorithm 4*, provided in the appendix, is used to apply the action correlation meta-rules at runtime. They have been implemented as ATL lazy rules in our framework.

Meta-Rules for Dependency Relation: Triggers on a sponsor entity can be due to the violation of one its thresholds and because of its dependent(s) as to take an action on a dependent entity, first the capacity of its sponsor is checked as a prerequisite. If both cases apply and a prerequisite action is taken to provide a sponsor first, it may also resolve the sponsor's trigger. To illustrate, let us assume that the workload for a service represented by a workload unit has more than one active assignment. Suppose at some point in time, the workload increases and two triggers are generated: One on the workload unit (dependent) and one on the node (sponsor) which supports one of the assignments of the workload unit. In this example, the least costly action of the elasticity rule invoked by the dependent trigger is executed first, which is adding an assignment on another node (i.e. the system is scaled out). Once the action path of the dependent entity is executed, the workload is shared between more nodes and therefore less workload will be imposed on the original sponsor node for which the sponsor trigger was received. It may not be applicable anymore and to determine that the sponsor trigger needs to be updated. As a result, the first meta-rule for the dependency relation is defined to handle horizontal scaling (i.e. scaling out). It is as follows:

Meta-Rule 1: If the relation between triggers is of type physical containment or assignment relationship and the optimal action for resolving the dependent trigger is scale-out, the action path for the dependent entity is executed before the path for the sponsor entity.

Meta-Rule 1 handles the cases where the relation between the triggers is of type physical containment or assignment relationship and the execution of the action path for the dependent provides solution for the sponsor through adding a new sponsor (i.e. scaling out). Note that it is possible that adding an assignment was not the least costly action or it was not an option at all and therefore the first meta-rule is not always applicable.

Meta-Rule 2: If multiple triggers have physical containment relations with the same container trigger and the optimal action for resolving each contained trigger is scale-up, some of the corresponding entities of the contained triggers may be migrated base on the cost of the migration. The corresponding entities of the contained triggers are sorted in ascending order using the metric $m = (migrationCost/releasedResource)$, where *migrationCost* is the approximate cost of migrating the contained entity to another container and *releasedResource* is the amount of resources released by migration. The contained entities with smaller m are migrated until the container trigger is resolved.

Unlike *Meta-Rule 1*, *Meta-Rule 2* handles vertical scaling (i.e. scale up). According to *Meta-Rule 2*, if multiple contained entities (i.e. dependents such as VMs) depend on the same container (i.e. sponsor such as a physical host) need to be scaled up but the container does not have enough resources for all of them, one or more contained entities (i.e. dependents) are migrated to other containers first to release resources of the container. The selection is made based on whose migration releases more critical resources at less cost. The released resources of the sponsor can then be given to the remaining dependent entities to scale up.

If the relation between triggers is of type dependency, but none of *Meta-Rule 1* and *Meta-Rule 2* can be applied, still we need to make sure that the action paths of dependent and sponsor are not executed simultaneously. To guarantee this, we execute the action path of the sponsor before the action path of dependent. Therefore, the third meta-rule is defined as follows:

Meta-Rule 3: If the relation between triggers is of type dependency but none of *Meta-Rule 1* and *Meta-Rule 2* can be applied, the action path for the sponsor entity is executed before the action path for the dependent entity.

Meta-Rules for Adjacency Relation: When triggers invoke elasticity rules on adjacent entities, it is possible that the actions of the elasticity rules would like to manipulate the common sponsor entity of the adjacent entities (i.e. their container or sponsor) simultaneously. These actions may be conflicting or interfering. To prevent such conflicts, only one action at a time is taken on the common entity, i.e. the actions are ordered. The order of actions on the common entity affects the efficiency of reconfiguration. To optimize it, the following meta-rules are defined:

Meta-Rule 4: The actions releasing resources of the common entity are taken first.

Meta-Rule 5: Any action that would remove a common resource/entity (e.g. removing a node) is considered only after executing all the action paths of all adjacent triggers.

When executing the action paths, triggers which release resources are given higher priority than triggers which allocate resources to enable reallocation. However, the actions releasing resources of the common entity are delayed until all the adjacent triggers have been considered. Thus, the resources of the common entity are released at the end only if they have not been reallocated by corresponding actions of other adjacent triggers. When all the resources of a common entity can be removed, the common entity is removed as well (e.g. a work pool is removed when its entire member serving units can be removed or when a common entity such as node has no process to run).

5.2.6. An example for trigger correlation and reconfiguration

Let us consider the example shown in Fig. 1 again. Suppose at some point in time, the configuration is as shown in Fig. 17 (a), and two triggers (t_1 and t_2) are generated by the SLA compliance management for *Workload Unit*$_1$ and *Workload Unit*$_2$, respectively. Assume trigger t_1 is generated due to the decrease in the workload represented by *Workload Unit*$_1$ to the point that two assignments should be removed, and trigger t_2 is generated due to the increase of the workload represented by *Workload Unit*$_2$.

To reconfigure the system, first the triggers issued on related entities are put into relation. *Workload Unit*$_1$ and *Workload Unit*$_2$ are protected by the same work pool (having the same logical container). Therefore, their corresponding triggers are put in adjacency relation. In this relation, the common entity is *Work Pool*$_1$. Fig. 17 (b) shows the relation graph resulted from the trigger correlation process.

Next, the applicable elasticity rules are selected and based on the defined action correlation meta-rules, the triggers of the relation graph are ordered for the invocation of the applicable elasticity rules. Based on *Meta-Rule 4* for the adjacency relation, the action path resulting from t_1 should be executed before the action path resulting from t_2 because the *scalingType* of trigger t_1 is *Decrease*. Therefore, trigger t_1 is considered first and its corresponding elasticity rule is executed. According to the elasticity rule for *Workload Unit*$_1$, two assignments should be removed to reconfigure the system. Fig. 17 (c) shows the configuration resulting from

the removal of assignments. As shown in the figure, by the removal of assignments, serving units hosted on *Node*$_3$ and *Node*$_4$ become unassigned (without assignments). Considering *Meta-Rule 5*, the issue of follow-up trigger on *Work Pool*$_1$ as common entity is delayed till the adjacent trigger t_2 manipulates the common entity.

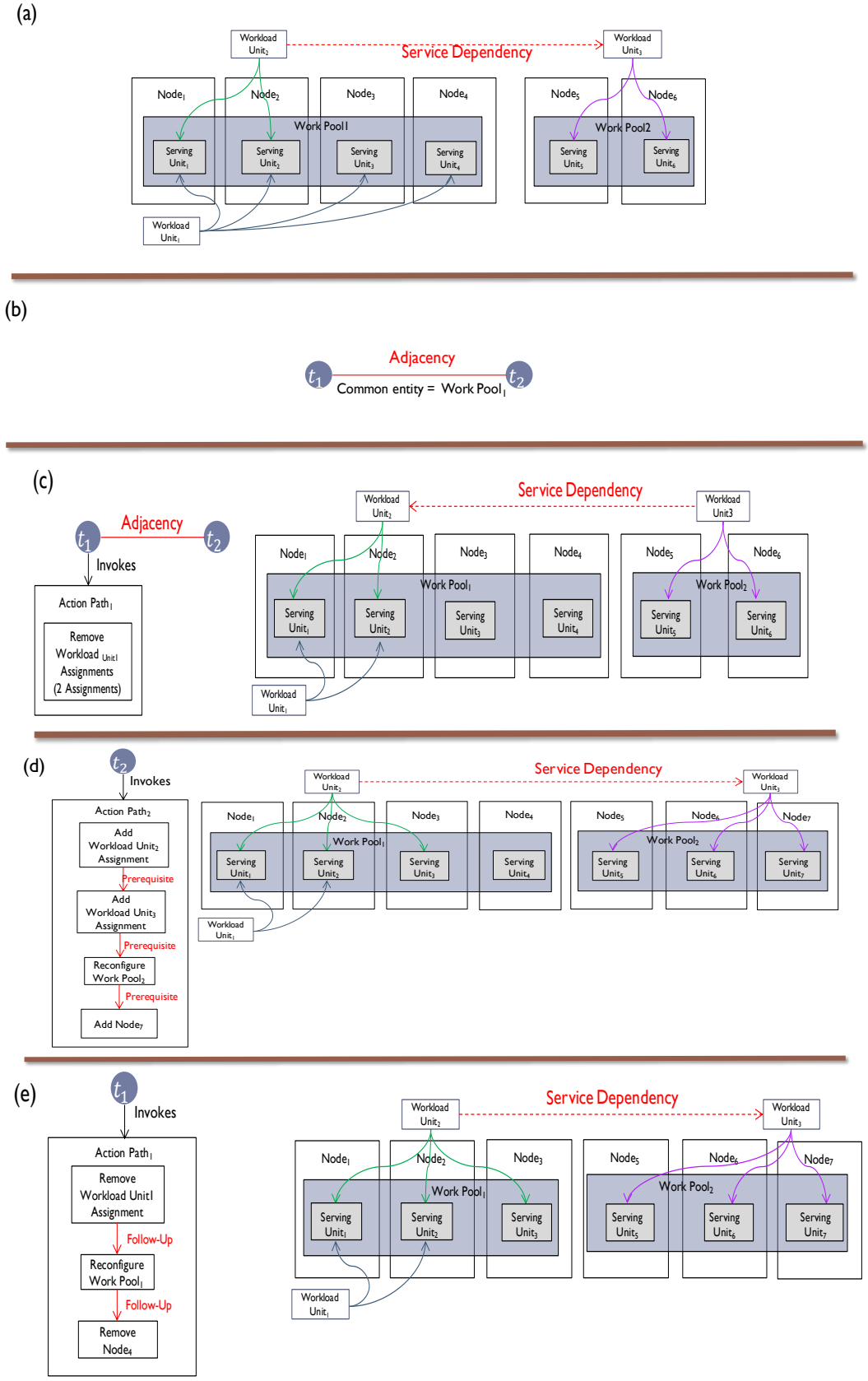

Fig. 17. System reconfiguration-An example

According to the elasticity rule initiated by trigger t_2, one assignment should be added to handle the workload increase represented by Workload Unit3. To take this action, two prerequisites should be met: There should be a serving unit to which the added assignment can be assigned and also its sponsor should have enough capacity to support the increase. The first prerequisite can be met by *Serving Unit*$_3$ or *Serving Unit*$_4$. Since according to the service dependency each assignment of *Workload Unit*$_2$ needs one assignment of *Workload Unit*$_3$, the increase in the workload represented by *Workload Unit*$_2$ cannot be sponsored by the current number of *Workload Unit*$_3$'s assignments. Therefore, the second prerequisite is not met. To make the action feasible, a prerequisite trigger on *Workload Unit*$_3$ is generated to increase the sponsor's capacity. The generated prerequisite trigger invokes the elasticity rule for *Workload Unit*$_3$. According to *Workload Unit*$_2$'s elasticity rule, one assignment of *Workload Unit*$_3$ should be added to resolve the prerequisite trigger; however, the action cannot be taken until *Work Pool*$_2$ is reconfigured in a way that the added assignment can be assigned. Therefore as a prerequisite, the required serving unit should be added to *Work Pool*$_2$ first. To add a serving unit, there should be a node to provide the required resources for the added serving unit. Although *Node*$_4$ has enough resources, it cannot host the serving units of *Work Pool*$_2$ because *Node*$_4$ is not a member of *Node Group*$_2$ on which *Work Pool*$_2$ can be configured. Since this perquisite is not met, first a node is added so that the serving unit can be added to *Work Pool*$_2$. Fig. 17 (d) shows the configuration resulting from the execution of *Action Path*$_2$. Note that node groups are shown in Fig. 1 only and not in Fig. 17.

Once all adjacent triggers (i.e. t_1 and t_2) with the same common entity have been processed, the delayed follow-up trigger on *Work Pool*$_1$ can be evaluated and therefore the serving unit hosted on Node4 is removed from *Work Pool*$_1$. *Node*$_4$ does not have any running serving units. Therefore, the resource removal action can be taken at this moment. Fig. 17 (e) shows the configuration resulting from the execution of the delayed follow-up actions in Action Path$_1$. As explained in this example, the action paths are not pre-built and the actions are executed right away.

6. Prototype Implementation and Preliminary Evaluation

In this section we present a preliminary evaluation of our framework using a prototype implementation and discuss the results. Each test was performed five times and the average is reported here as the execution time.

6.1. Validation of the SLA Compliance Model and Trigger Generation

In this prototype, Domain Specific Modeling Languages (DSMLs) are used to capture the concepts, their relations as well as their well-formedness rules (constraints). To define the DSMLs, we use the UML profiling mechanism [32]. We followed the approach in [37] to create UML profiles in two steps: we first defined the metamodels and then mapped the metamodels to the UML metamodel.

To generate triggers from violated OCL constraints, we used OCL APIs [38] in a standalone java application in the Eclipse Modeling Framework (EMF) [39]. The OCL constraints of SLA compliance profile are extracted and validated given an SLA compliance model. If a constraint is evaluated to false, a trigger is generated and takes the name of the violated constraint (i.e. either Increase or Decrease); corresponding entity is the context of the violated constraint (i.e. either a node or a service) and the measurement and threshold are from the constrained elements (i.e. the measurement and the threshold) of the violated constraint.

Table 1 presents the results for SLA compliance model validation and trigger generation given different SLA compliance models and measurements. The first column of the table is the number of elements in the SLA compliance model. The SLA compliance models differ in the number of nodes, the number of SLAs and the number of services of the same or different service types. These models were built offline. For each case, the input measurements were compiled also offline in such a way that some would violate their corresponding thresholds. The second column is the total number of constraints to check. As explained in

Section 4.1.4, constraints are defined on SLA parameters, service entities (i.e. WUs) and nodes where for some (i.e. service entities and nodes) more than one constraint is defined. As a result, as the size of the SLA compliance model increases with the increase of the number of nodes, SLAs or the contained services in the considered configuration, the number of constraints to check increases too. The third and fourth columns show the number of the generated triggers and the total execution time of the SLA compliance model validation and the trigger generation, respectively. The result of this evaluation is represented by the chart in Fig. 18. As the number of elements in the SLA compliance model increases, more constraints are checked, and therefore the validation time increases. From the validation of the SLA compliance models we can conclude that the execution time grows linearly with respect to the numbers of constraints to check (in each case, the proportion of execution time to the number of constraints to check is almost 100).

Table 1. SLA Compliance Model Validation and Trigger Generation Performance Evaluation

CASES	Number of Model Elements	Number of Constraint Checks Performed	Number of Generated Triggers	Execution Time (ms)
CASE 1	13	7	1	694
CASE 2	16	11	2	1189
CASE 3	24	14	3	1377
CASE 4	26	18	4	1845
CASE 5	42	28	6	2844
CASE 6	87	77	8	7573

6.2. Trigger Correlation and Dynamic Reconfiguration

We implemented a prototype of trigger correlation and dynamic reconfiguration using ATL [34]. To analyze the efficiency of our approach for trigger correlation and dynamic reconfiguration, we consider the triggers generated in the previous experiment (i.e. Section 7.1). It is worth mentioning that the generated triggers are not redundant and therefore, with the correlation approach the number of triggers remains the same after correlation. Since we manipulate the models, the execution time is the time of making changes in the configuration model and does not include the execution time of the actions. For example, when a node is added, this addition manifests as a change in the number of instantiated nodes in the configuration model; however, in real systems, creation of VM instances may take several minutes [40]. As a result to analyze the efficiency of our approach, we measure the execution time as well as the number of reconfiguration actions with our approach where the triggers as well as the actions of invoked elasticity rules are correlated, and compare them to the execution time and number of actions when the triggers are not correlated. Fig. 19 shows the result of this comparison. As shown in Fig. 19 (a), the results demonstrate the reduction in the number of actions by the correlation approach in overall which means less applicable elasticity rules are selected and invoked at runtime. As a result, by the correlation approach the execution time which includes the correlation time is reduced in overall as well. As the actions are executed at runtime, reducing the number of reconfiguration action is an important goal for real time and highly available systems. In the case that the triggers are not related (like last case in Fig. 19), the execution time is more in our approach which is due to the time for checking relations between the triggers to correlate them. It is worth mentioning that the stability of the system is not guaranteed when the triggers are not correlated.

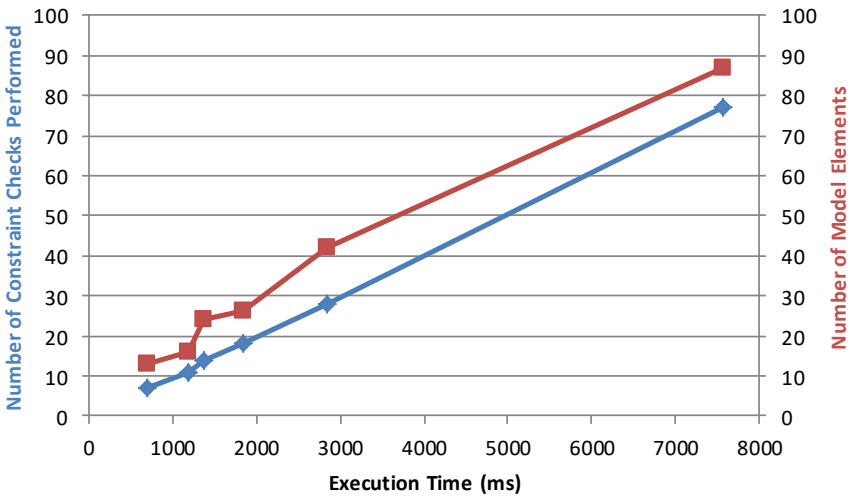

Fig. 18. Performance evaluation for SLA compliance model validation and trigger generation given different SLA compliance models

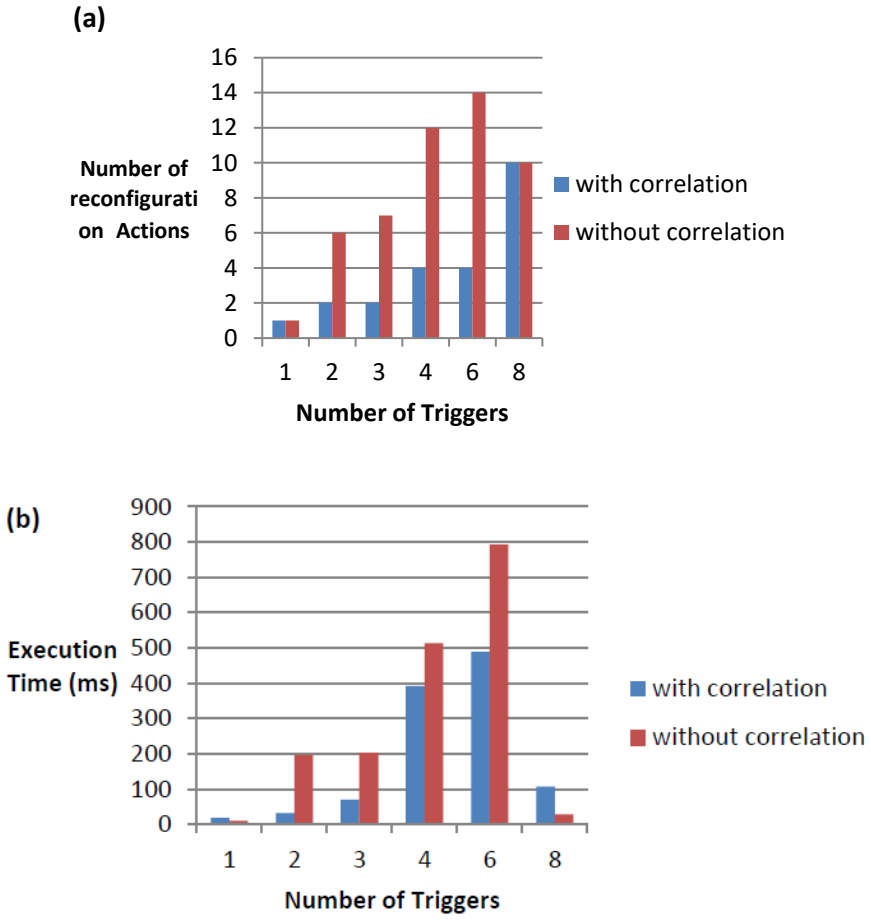

Fig. 19. Comparison of the execution time and the number of reconfiguration actions for dynamic reconfiguration with correlation and without correlation.

7. Conclusion

To adapt a system at runtime based on the workload fluctuation, we proposed a framework which is finer-grain than current approaches for the elasticity management of cloud systems. In our proposed framework,

the resources are not only added or removed when it is required, but they are also reorganized for better resource utilization. All these are performed while taking into account also service availability.

We proposed a model-driven framework which reuses the models developed at the design stage (e.g. configuration model). We defined OCL constraints that are periodically evaluated at runtime to generate triggers automatically from the violated OCL constraints. The generated triggers initiate the application of corresponding elasticity rules to reconfigure the system and avoid SLA violations by the provider and resource wasting.

Since multiple triggers may be generated simultaneously, handling the triggers independently may jeopardize the stability of the system. We proposed a model driven approach for correlating the triggers and the actions of their related elasticity rules. Triggers are correlated based on the relations existing between their corresponding configuration entities. The result of trigger correlation is represented as a set of relation graphs. For each trigger of a relation graph, the applicable elasticity rule is then selected. In order to correlate the actions of applicable elasticity rules, we defined action correlation meta-rules that govern the application of elasticity rules when the triggers are correlated. The goal is not only to reconfigure the system properly and avoid resource oscillation but also to minimize the number of reconfiguration actions because any change in the configuration needs to be applied at runtime on system entities. Moreover, the correlated actions are executed on the fly. I.e. no action path is evaluated or built before the execution. A correlated action is executed right away once its prerequisites are met if there is any.

We performed some experiments that show that our solution reduces the time of the dynamic reconfiguration and the number of reconfiguration actions while avoiding resource oscillation compared to the reconfiguration solution without trigger correlation.

To define the OCL constraints, we used a set of threshold values lower than the current capacity of the system. We assumed the values of the thresholds are given. As future work, the average reconfiguration time and the predicted workload as well as the cost of reconfiguration versus the penalty of SLA violations can be considered to determine the values of the thresholds.

Appendix

Algorithm 1: Trigger Correlation

Input: TriggerSet, Configuration

Output: RelationGraphSet

```
1:  //creating relationgraphs
2:  RelationSet := Configuration.relations
3:  RelationGraphSet.relations := {}
4:  RelationGraphSet.triggers := TriggerSet
5:  For each relation in RelationSet
6:      If (TriggerSet.entities.includesAll(relation.entities)) then
7:   RelationGraphSet.relations := RelationGraphSet.relations ∪
          { TransformRelation(relation, relation.entities.at(1).trigger,
          relation.entities.at(2).trigger)}
8:      End if
9:  End for
10: Return RelationGraphSet
```

Algorithm 2: Selecting Applicable Elasticity Rules

Input: TriggerSet, ElasticityRuleSet

Output: ApplicableElasticityRuleSet

```
1:  // Selecting applicable elasticity rules
2:  ApplicableElasticityRuleSet:={}
3:  For each trigger in TriggerSet
4:      For each elasticityRule in ElasticityRuleSet
5:          If (elasticityRule.entityType == trigger.entity.entityType) then
6:   If (elasticityRule.scalingRule==trigger.scalingType) then
7:   ApplicableElasticityRuleSet  := ApplicableElasticityRuleSet ∪ {(trigger,
                    elasticityRule)}
8:           End if
9:          End if
10:     End For
11: End For
12: Return  ApplicableElasticityRuleSet
```

Algorithm 3: Selecting Optimal Action

Input: ElasticityRule, Measurement, Threshold, configuration

Output: OptimalAction

```
1:  //Selecting Optimal action based on the current situation
2:  ActionSet := ElasticityRule.actions          /// Set of all actions contained in the elasticity
    rule
3:  optimalAction := action0
4:  optimalCost := action0.midCost
5:  elasticityRuleFeasibility := false
6:  For each action in ActionSet
7:    If (evaluate(action.condition,configuration,threshold,Measurement)) then
8:        actionFeasibility := true
9:        For each prerequisite in action.prerequisites
10:           If (not evaluate(prerequisite,configuration,threshold,Measurement)) then
11:               actionFeasibility := false
12:               Break
13:           End if
14:       End for
15:       If (not elasticityRuleFeasibility and not actionFeasibility) then
16:           If (optimalCost > action.midCost) then
17:               optimalAction := action
18:               optimalCost := action.midCost
19:           End if
20:       If (elasticityRuleFeasibility and actionFeasibility) then
21:           If (optimalCost > action.cost) then
22:               optimalAction := action
23:               optimalCost := action.cost
24:           End if
25:       Else if (not elasticityRuleFeasibility and actionFeasibility) then
26:           elasticityRuleFeasibility := true
27:           optimalAction := action
28:           optimalCost := action.cost
29:       End if
30:   End if
31: End for
32: Return  optimalAction
```

Algorithm 4: Applying Action Correlation Meta-Rules

Input: relationGraph, ApplicableElasticityRuleSet, configuration, Measurement, Threshold, configuration

Output: TriggerSet

```
33: //Sorting the triggers of a relation graph based on the relations between them
34: TriggerSet := RelationGraph.triggers
35: RelationSet := RelationGraph.relations
36: For each relation in RelationSet
37:   If (relation.isTypeOf() == Adjacency ) then
38:       adjcent1:= relation.vertices().at(1)
39:       adjacent2:= relation.vertices.at(2)
40:       If (adjacent1.indexAtTriggerSet () < adjacent2.indexAtTriggerSet() and
          adjacent1.scalingType == increase and adjacent2.scalingType == decrease)
          then
41:         TriggerSet.swap (adjacent1,adjacent2)
42:       End if
43:   End if
44:   If (relation.isTypeOf() == Dependency) then
45:       dependentOptAction := optimalAction(dependent.applicableEr,configuration,
          Measurement,Threshold)
46:       sponsorOptAction := optimalAction(sponsor.applicableEr, configuration,
          Measurement, Threshold)
47:       If (dependentOptAction == scale-out) then
48:           If (sponsor.indexAtTriggerSet () < dependent.indexAtTriggerSet()) then
49:             TriggerSet.swap (sponsor,dependent)
50:           End if
51:       Else if (dependentOptAction == scale-up and sponsorOptAction == scale-up)
          then
52:               RelationSubset:= findOtherDependency (sponsor, RelationSet))
53:               If (findOtherScaleUpDependents (RelationSubset).notEmpty()) then
54:                   DependentSet := RelationSubset.dependents ∪ {dependent}
55:                   migrationSet := findMigrationSet (DependentSet)
56:                   scaleUpSet := DependentSet – migrationSet
57:                   TriggerSet.partialSort (MigrationSet, DependentSet)
58:                   RelationSet := RelationSet – RelationSubset
59:               Else if (findOtherScaleUpDependents (RelationSubset).isEmpty())
                  then
60:                   If (dependent.indexAtTriggerSet () < sponsor.indexAtTriggerSet())
                      then
61:                       TriggerSet.swap (dependent, sponsor)
62:               End if
63:       Else
64:           If (dependent.indexAtTriggerSet () < sponsor.indexAtTriggerSet()) then
65:               TriggerSet.swap (dependent, sponsor)
66:       End if
67:   End if
68: End for
69: Return TriggerSet
```

Acknowledgment

This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Ericsson.

References

[1] Larson, K. (1998). The role of service level agreements in IT service. *Information Management & Computer*, *6(3)*, 128-132.

[2] Herbst, N., Kounev, S., & Reussner, R. (2013). Elasticity in cloud computing: What it is, and what it is not. *Proceedings of the International Conference on Autonomic Computing.*

[3] Chapman, C., Emmerich, W., Márquez, F., Clayman, S., & Galis, A. (2012*).* Software architecture definition for on-demand coud provisioning. *Cluster Computing*, *15(2)*.

[4] Zhu, X., Uysal, M., Zhikui, W., Singhal, S., Arif, M., Padala, P., & Shin, K. (2009). *What Does Control Theory Bring to Systems Research?* ACM SIGOPS Operating Systems Review.

[5] OMG: MDA, Retrieved from: http://www.omg.org/mda/

[6] Selic, B. (2003). The pragmatics of model-driven development. *IEEE Software*, *20(5)*, 19-25.

[7] OMG: MDA User guide, version 1.0. Retrieved from: https://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf

[8] OMG: Object Constraint Language (OCL). Retrieved from: http://www.omg.org/spec/OCL/

[9] Ludwig, H., Stamou, K., Mohamed, M., Mandagere, N., Langston, B., Alatorre, G., Nakaruma, H., Anya, O., & Keller, A. (2015). rSLA: Monitoring SLAs in dynamic service environments. *Proceedings of the International Conference on Service-Oriented Computing* (pp.139-153).

[10] Tata, S., Mohamed, M., & Sakairi, T. (2016). *rSLA:* A service level agreement language for cloud services. *Cloud Computing,* 415-422.

[11] Raimondi, F., Skene, J., Emmerich, W., & Wozna, B. (2007). A methodology for on-line monitoring non-functional specifications of web-services. *Proceedings of the First International Workshop on Property Verification for Software Components and Services*.

[12] Skene, J., & Emmerich, W. *Generating a Contract Checker for an SLA Language*. Retrieved from: http://eprints.ucl.ac.uk/712/1/9.9.1coala.pdf

[13] Emeakaroha, V., Netto, M., Brandic, I., & De, R. C. (2015). Application level monitoring and SLA violation detection for multi-tenant cloud services. *Emerging Research in Cloud Distributed Computing Systems.*

[14] Emeakaroha, V., Netto, M., Calheiros, R., Brandic, I., Buyya, R., & De, R. C. (2012). Towards autonomic detection of SLA violations in cloud infrastructures. *Future Generation Computer Systems*, *28(7)*.

[15] Yemini, S., Kliger, S., Mozes, E., Yemini, Y., & Ohsie, D. (1996). *High Speed and Robust Event Correlation*.

[16] Gruschke, B. (1998). Integrated event management: Event correlation using dependency graphs. *Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*.

[17] Konig, B., Calero, J. A., & Kirschnick, J. (2012). Elastic monitoring framework for cloud infrastructures. *IET Communications*, *6(10)*.

[18] Sedaghat, M., Hernandez-Rodriguez, F., & Elmroth, E. (2013). A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling. *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*.

[19] Shariffdeen, R. S., Munasinghe, D. T. S. P., Bhathiya, H. S., Bandara, U. K. J. U., & Dilum, H. M. N. (2016). Workload and resource aware proactive auto-scaler for paas cloud. *Proceedings of the 2016 IEEE 9th International Conference on Cloud Computing*.

[20] Tang, P., Li, F., Zhou, W., Hu, W., & Yang, L. (2016). Efficient auto-scaling approach in the telco cloud using self-learning algorithm. *Proceedings of the IEEE Global Communications Conference* (pp. 1-6).

[21] Wang, C., Gupta, A., & Urgaonkar, B. (2016). Fine-grained resource scaling in a public cloud: A tenant's perspective. *Proceedings of the 2016 IEEE 9th International Conference on Cloud Computing*.

[22] Shen, Z., Subbiah, S., Gu, X., & Wilkes, J. (2011). Cloudscale: Elastic resource scaling for multi-tenant cloud

systems. *Proceedings of the 2nd ACM Symposium on Cloud Computing*.
[23] Jamshidi, P., Ahmad, A., & Pahl, C. (2014). Autonomic resource provisioning for cloud-based software. *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*.
[24] Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., & Merle, P. (2018). Elasticity in cloud computing: State of the art and research challenges. *Proceedings of the IEEE Transactions on Services Computing*.
[25] Zhang, Q., Chen, H., & Yin, Z. (2017). PRMRAP: A proactive virtual resource management framework in cloud. *Proceedings of the 1st International Conference on Edge Computing*.
[26] Network functions virtualisation (NFV); terminology for main concepts in NFV. Retrieved from: http://www.etsi.org/deliver/etsi_gs/NFV/001_099/003/01.02.01_60/gs_NFV003v010201p.pdf
[27] Ali-Eldin, A., Tordsson, J., & Elmroth, E. (2012). An adaptive hybrid elasticity controller for cloud infrastructures. *Proceedings of the Network Operations and Management Symposium*.
[28] Toeroe, M., & Tam, F. (2012). *Service Availability: Principles and Practice*. John Wiley & Sons.
[29] Service Availability Forum. Retrieved from: http://devel.opensaf.org/documentation.html
[30] Abbasipour, M., Khendek, F., & Toeroe, M. (2015). A model-based framework for SLA management and dynamic reconfiguration. *Proceedings of the International SDL Forum* (pp. 19-26).
[31] Ericsson introduces a hyperscale cloud solution. Retrieved from: http://archive.ericsson.net/service/internet/picov/get?DocNo=28701-FGB1010554&Lang=EN&HighestFree=Y
[32] OMG: UML 2.0 Superstructure — Final Adopted Specification. Retrieved from: http://www.omg.org/cgi-bin/doc?ptc/2003-08-02.
[33] Birkenheuer, G., Brinkmann, A., & Karl, H. (2009). *The Gain of Overbooking.* Springer Berlin Heidelberg.
[34] ATL/user guide-the ATL language. Retrieved from: https://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language.
[35] Abbasipour, M., Khendek, F., & Toeroe, M. (2018). A model-based approach for design time elasticity rules generation. *Proceedings of the 23rd International Conference on Engineering of Complex Computer Systems* (pp. 93-103).
[36] Abbasipour, M., Khendek, F., & Toeroe, M. (2018). *Trigger Correlation for Dynamic System Reconfiguration*. SAC 2018, Pau, France.
[37] Selic, B. (2007). A systematic approach to domain-specific language design using UML. *Object and Component-Oriented Real-Time Distributed Computing*.
[38] OCL - Eclipsepedia - Eclipse Wiki. Retrieved from: https://wiki.eclipse.org/OCL
[39] Eclipse Modeling Framework (EMF). Retrieved from: https://www.eclipse.org/modeling/emf/
[40] Jiang, Y., Perng, C.-S., & Li, T. (2013). Cloud analytics for capacity planning and instant VM provisioning. *IEEE Transactions on Network and Service Management*, *10(3)*, 312-325.
[41] Nabi, M., Toeroe, M., & Khendek, F. Availability in the cloud: State of the art. *Journal of Network and Computer Applications*, 54-67.
[42] Moreno-Vozmediano, R., Monter, R. S., Huedo, E, & Liorente, I. (2019). Efficient resource provisioning for elastic cloud services based on machine learning techniques. *Journal of Cloud Computing*.

Mahin Abbasipour received her Ph.D. degree (electrical and computer engineering) from Concordia University, Montreal, QC, Canada in 2018. During her study at Concoria University, Mahin did a 9 month internship at Ericsson Inc., Montreal focusing on SLA compliance management. Mahin's research interests are model driven approach (MDA), requiremnts engineering and dynamic system reconfiguration.

Ferhat Khendek received his PhD from University of Montreal, Canada. He is a full professor in the Department of Electrical and Computer Engineering of Concordia University where he also holds since 2011 the NSERC/ericsson senior industrial research chair in Model Based Management. Ferhat Khendek has published more than 200 conference/journal papers. Ferhat Khendek's research interests are in model based software engineering and management, formal methods, validation and testing, cloud computing, real-time software systems, and service engineering and architectures.

Maria Toeroe is an expert at Ericsson working in the area of dependable software, service availability and fault tolerance. She has represented Ericsson in the Service Availability Forum and more recently in OPNFV and ETSI NFV. Maria is also the technical coordinator of the Ericsson research collaboration with Concordia University. She has numerous publications and served as organizer and program committee member of different conferences. Maria holds a PhD from the Budapest University of Technology and Economics.