Towards Multi-cloud SLO Evaluation

Kyriakos Kritikos, Chrysostomos Zeginis, Andreas Paravoliasis and Dimitris Plexousakis
ICS-FORTH, N. Plastira 100, Vasilika Vouton, 70013, Heraklion, Crete, Greece

Keywords: Complex Event Processing, Event Pattern, Detection, Service.

Abstract: A modern service-based application (SBA) operates in a cross-cloud, highly dynamic environment while comprises various components at different abstraction levels that might fail. To support cross-level SBA adaptation, a cross-cloud Service Level Objective (SLO) monitoring and evaluation system is required, able to produce the right events that must trigger suitable adaptation actions. While most research focuses on SBA monitoring, SLO evaluation is usually restricted in a centralised, single-cloud form, not amenable to heavy workloads that could incur in a complex SBA system. Thus, a fast and scalable event generation and processing system is needed, able to scale well to handle such a load. Such a system must address the cross-level event composition, suitable for detecting complex problematic situations. This paper closes this gap by proposing a novel complex event processing framework, scalable and distributable across the whole SBA architecture. This framework can cover any kind of event combination, no matter how complex it is. It also supports event pattern management while exploits a publish-subscribe mechanism to: (a) synchronise with the modification of adaptation rules directly involving these event patterns; (b) enable the decoupling from an SBA management system.

1 INTRODUCTION

Modern SBA systems start exploiting the cloud to benefit from its main advantages, including flexible pricing and resource management. However, cloud migration comes with certain challenges. First, the cloud is out of SBA provider control and offers only restrictive management actions over the abstracted resources. Second, the cloud is a dynamic environment in which resources can fail or under-perform, especially if they are shared among different customers.

In an SBA, the above challenges can be exacerbated due to the complexity introduced by the multiple levels incorporated and their dependencies. Besides, level-specific adaptation can fail due to the vicious adaptation cycle (Zeginis et al., 2015). As such, there is a need for a management system able to monitor and adapt the SBA across all levels via the coordinated execution of level-specific adaptation actions.

As a glue between monitoring and adaptation, an SLO evaluation framework must evaluate the SLOs of the SLA between the SBA provider and requester and generate events that can trigger SBA adaptation. However, due to the SBA complexity and the speed in which events occur at the infrastructure level, such a framework must exhibit fast and scalable event processing. Such processing needs to be aligned with the main SBA adaptation goals to avoid overspending precious system resources. As such, only those events or event patterns leading to the need to trigger adaptation actions should be detected.

The recent advent of sophisticated cross-level SBA monitoring and adaptation systems can be observed. Such systems, however, exhibit poor event processing, as they are not scalable to handle an increasingly huge event number. To close this gap, this paper presents a novel SLO evaluation framework with the following features: (a) scalable and distributable across the whole SBA architecture; (b) it relies on a scalable complex event processing engine; (c) is based on CAMEL, a Domain-Specific Language (DSL) supporting multi-cloud SBA management; (d) it exploits publish-subscribe mechanisms to be decoupled from specificities of other SBA management system parts; (e) it supplies an event pattern management service so as to synchronise with modifications on the SBA adaptation behaviour specification.

The rest of the paper is structured as follows. Section 2 supplies the paper's running example. Section 3 reviews the state-of-the-art. Section 4 explains CAMEL. Section 5 analyses the proposed framework architecture. Section 6 explains how event pattern models are generated from CAMEL adaptation rules. Finally, Section 7 concludes the paper.

Kritikos, K., Zeginis, C., Paravoliasis, A. and Plexousakis, D.
Towards Multi-cloud SLO Evaluation.
DOI: 10.5220/0006684604090417
In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 409-417
ISBN: 978-989-758-295-0
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

2 USE CASE

As a running example, we adopt a use case from the CloudSocket project[1], dealing with Business Process (BP) management in the Cloud. This use case concerns developing and deploying a BP as a service (BPaaS), named as "SendInvoice", supporting the functionality of invoice generation and sending.

Internally, the "SendInvoice" BPaaS utilises 2 main services: (a) the YMENS CRM external SaaS focusing on customer relationship management; (b) the "Invoice Ninja" internal invoice management component, purchased and deployed in the Amazon Cloud on a "m1.medium" VM. Both services are combined into a technical workflow, deployed in the cloud, which incorporates tasks mapping to certain methods of these services.

The initial cloud deployment of "SendInvoice" BPaaS maps to a certain (both type & instance-based level) topology, depicted in Fig.1 and specified in CAMEL. In this topology, one instance of the "InvoiceNinja" internal component, named as "InvoiceNinja_inst1" has been deployed on one instance of the "m1.medium" VM called "m1.medium_inst1".

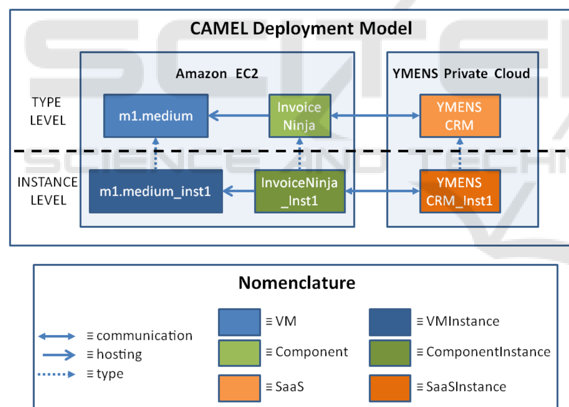

Figure 1: The topology of the "SendInvoice" BPaaS.

Suppose that the organisation, i.e., a Cloud Broker, offering the BPaaS, must control its provisioning to sustain a suitable service level offered in the context of an SLA. As the set of customers purchasing and using the BPaaS can grow, this Broker must also control the amount of resources dedicated to "Invoice Ninja" via scaling and be able to replace YMENS CRM if it is under-performing. Thus, the broker defines a set of CAMEL adaptation rules shown in Table 1 (abstracted away in restricted form of CAMEL textual syntax) that scale out "Invoice Ninja" or replace "YMENS CRM" with another SaaS. These rules are then supplied as input to the BPaaS Execution Environment of the CloudSocket platform which performs the adaptation actions required, when needed. The SLO Evaluation framework proposed can belong to such an environment to enable it to truly sustain a suitable service level for SLO evaluation.

where *raw_cpu* & *raw_mem* are the Raw CPU and Raw Memory Utilisation metrics, *mean_rt*, *mean_cpu* and *mean_avail* are the MEAN Response Time, CPU Utilisation and Availability metrics while *IN* represents "Invoice Ninja" and *YC* "YMENS CRM".

Rules R_1 & R_3 focus on scaling out "Invoice Ninja". R_1 attempts to immediately scale this component when one of its instances is severely overloaded. R_2 attempts to replace "YMENS CRM" when its mean response time is beyond a certain threshold and its availability drops under a certain level.

3 RELATED WORK

Many research approaches has been proposed targeting SLO evaluation in dynamic environments like the Cloud. (Ludwig et al., 2015) proposes the rSLA SLA language and management service. rSLA has been validated using IaaS-related SLOs.

(Wu et al., 2013) proposes a cloud-based SLA classification model which distinguishes SLA parameters for each cloud level. An SLA evaluation method is also introduced, able to produce a final cloud service score by considering the preferences and requirements of both the service provider and consumer.

(Bahga and Madisetti, 2013) proposes a performance evaluation approach for complex multi-tier cloud applications which captures the application workload via 3 different models: benchmark, workload, and architecture. Based on the architecture model, a certain methodology is proposed to facilitate selecting the most effective deployment meeting the application requirements. A comparison of different deployment architectures is performed to detect system bottlenecks and thus lead to right design choices.

The DeSVi single-cloud architecture (Emeakaroha et al., 2012) supports SLA violation detection by employing the LoM2His framework for application monitoring and translating low-level metrics into high-level SLOs. LoM2His can optimally tune the SLA parameter monitoring interval.

Very few approaches deal with CEP in the cloud. (Higashino, 2016) proposes CEP as a Service (CEPaaS) to enable using CEP with the advantages offered by a service model. (Leitner et al., 2012) proposes another CEP-based event correlation approach relying on a predefined event hierarchy.

[1] www.cloudsocket.eu

Table 1: The 3 CAMEL adaptation rules.

ID	**Rule Content**
R_1:	EVERY $(raw_cpu\,(m1.medium) > 85\%$ AND $raw_mem\,(m1.medium) > 90\%) \Rightarrow scale-out\,(IN)$
R_2:	EVERY $(mean_rt\,(YC) > 30$ AND $mean_avail\,(YC) < 99.5\%) \Rightarrow replace\,(YC)$
R_3:	EVERY $(mean_cpu\,(m1.medium) > 70\%$ AND $mean_avail\,(IN) < 90 \Rightarrow)\,scale-out\,(IN)$

4 BACKGROUND – CAMEL OVERVIEW

CAMEL is a multi-DSL, developed in the PaaSage[2] project for specifying all relevant aspects in multi-cloud SBA lifecycle, including deployment, requirement, metric, and scalability aspects. CAMEL integrates existing aspect-specific DSLs, like CloudML (Ferry et al., 2013) (deployment aspect), plus new ones, like the Scalability Rule Language (Kritikos et al., 2014) (metric & scalability aspects). As this paper focuses on SLO evaluation that is also related to SBA adaptation, only the metric and scalability aspects are analysed.

The metric aspect, captured by CAMEL's metric package, covers all measurement details needed to specify a metric, like formulas, measurement schedules and windows. This package enables also to specify conditions over metrics to be exploited to define SLOs and non-functional events in scalability rules.

The scalability aspect has been recently adapted (Kritikos et al., 2017) in CloudSocket to cover specifying complex adaptation rules and not just scalability ones. Such rules can then drive cross-level multi-cloud SBA adaptation. These rules map single events or event patterns to an adaptation workflow comprising level-specific adaptation actions.

Due to this paper focus, only the conceptualisation of events and event patterns (EPs) will be further analysed for the adaptation aspect. An event can be single or composite. A single, non-functional event maps directly to the violation of a metric condition. A composite event maps to specifying a unary or binary EP, i.e., an event composition on which a certain time or logical operator applies. A unary EP applies an unary operator, like (logical) NOT, over a certain event. A binary EP applies a binary operator, like the PRECEDES time-based one over two events. Time-based operators have been mainly inspired from Esper's Event Pattern Language (EPL)[3].

As a composite event can map to any event kind, CAMEL can specify a hierarchy of EPs. This enables it to support specifying any complex problematic situation. For instance, consider running example's R_1 rule. For this rule, we have the specification of a unary EP EP_1 that applies the *EVERY* operator over the EP_2 EP. The latter is a binary EP that applies in turn the *AND* logical operator over two single events mapping to the two metric conditions in R_1, respectively.

[2]https://paasage.ercim.eu/

[3]http://www.espertech.com/esper/

5 SLO EVALUATION FRAMEWORK

5.1 Framework Logical Architecture

Figure 2 depicts the modular architecture of the proposed SLO Evaluation Framework comprising 3 main levels: (a) interface; (b) core logic; (c) database (DB). At the interface level, a REST service wraps the main actions (add, update, delete) that can be performed over EPs, by also being able to parse SRL fragments specifying such EPs. These actions are mapped to underlying calls on other framework components.

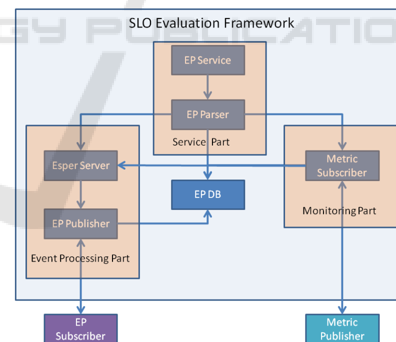

Figure 2: The SLO Evaluation Framework architecture.

The core logic level includes *EP Parser*, able to process the EPs obtained from the *EP Service*. Depending on the action requested, different interactions occur between components at this and the DB level.

EP Addition. The *EP Parser* transforms the CAMEL model of the EP to be added into a specification conforming to the CEP framework's EP language. The produced specification is then registered in that framework's *CEP Server* to be immediately detected. The names of metrics directly involved in the conditions of the EP's events are also sent to the *Metric Subscriber*. The latter first updates its local metric

list in the *EP DB* for fault-tolerance and rapid recovery reasons and then subscribes to such metrics, when they are new, in the *Metric Publisher*. The latter component publishes the values of the metrics monitored to potential subscribers. Thus, it could well match a *Monitoring Engine* of an SBA management system.

Once registration of both the new EP and its metrics finishes, the EP addition is deemed successful. Thus, the *EP Parser* can then store the new EP in the *EP DB* for recovery reasons but also for gathering statistics about EPs, when being detected by the *Esper Server*. The *EP DB* takes the form of a model repository able to store, query and manipulate CAMEL EP models along with their statistics.

EP Deletion. For this action, the EP is first obtained from *EP DB*. Then, *EP Parser* informs in parallel both *CEP Server* and *Metric Subscriber* about the EP deletion such that: (a) the *CEP Server* can deregister the EP's EPL specification; (b) the *Metric Subscriber*, after checking that the EP metrics to be removed are not exploited in other EPs, can unsubscribe to these metrics to reduce system load.

EP Update. Here, the previous EPL statement is updated with the new one generated by *EP Processor*. *Metric Subscriber* also adds / removes metrics which are / not needed any more by any EP, respectively.

The EP management actions can be initiated via either the interaction of the proposed system with an external agent/user or the use of a publish-subscribe mechanism. This has the advantage that we can easily switch from one to another interaction mechanism or have both available at the same time.

The above analysis covered the interactions in the context of EP management actions. Apart from this, some system components continuously run to support the delivery of the SLO evaluation functionality expected from the proposed system. The functionality of these components in explicated in detail below.

While *Metric Subscriber* subscribes to metrics, it can also asynchronously receive measurements for such metrics from *Metric Publisher*. These measurements are transformed by *Metric Subscriber* into events that are fed into the *CEP Server*. Once the latter receives all suitable events, it can detect EPs and subsequently inform the *Event Publisher*.

The *EP Publisher* will then publish these events to interested *EP Subscriber*s, which could take the form of adaptation engines able to execute the respective adaptation rule(s) triggered. In parallel to this publication, the *EP Publisher* also updates *EP DB* to modify the statistics of the EP(s) concerned.

The adoption of publish-subscribe mechanisms for measurement retrieval and SLO event publishing not only decouples the proposed framework from any SBA Management system but also enables such system to take any distributed or centralised form, especially concerning its adaptation part, to balance its workload. This is achieved by enabling all distributed system parts to subscribe to the *EP Publisher* to, e.g., manage their own adaptation space part, i.e., specific EPs. As next sub-section will show, the presented logical framework architecture has different physical implementation options that could well fit the distributed form of a SBA management system.

5.2 Physical Framework Architecture

The presented logical architecture can have various implementation options at the physical level. To choose the most optimal one, we need first to consider what can be distributed in that architecture and how the whole management system can be distributed.

For the whole management system, (3) levels can be involved: (1) the global responsible to manage the application as a whole; (2) the cloud one where management is restrained to a single cloud and all application components are deployed there; (3) the user VM level where management applies only on application components hosted on that VM. In each level, we also imagine that there are monitoring and adaptation components. Monitoring at the global level enables to assess application and cross-cloud metrics as well as application reconfiguration. Cloud-level monitoring enables to assess cloud-specific metrics plus perform adaptation actions, usually concerning application component scaling. Finally, VM-level monitoring enables to assess instance-based metrics plus adapt the application components hosted.

Our framework has the next distribution options: (a) it can be distributed as a whole such that different instances could be installed in different clouds; (b) only parts incurring the most load can be distributed, mapping to agglomerating the *CEP Server* and *Event Publisher*. In the latter case, the distribution does not need to follow any cloud-specific pattern. We just scale that agglomeration based on the workload into a suitable number of instances. The part mapping to the *Metric Subscriber* could also be distributed, not only due to the monitoring load to be faced but also to reduce latency by following the pattern of distribution of the SBA management system monitoring part. The two parts that can be distributed are named as *event processing part* (EPP) and *monitoring part* (MP). The rest of framework components are also partitioned into a *service part* (SP), comprising the REST service and *EP Parser*, and the *DB part* (DP) including the *EP DB*. These parts could also be scaled; such a scaling would remain at the same level and will be

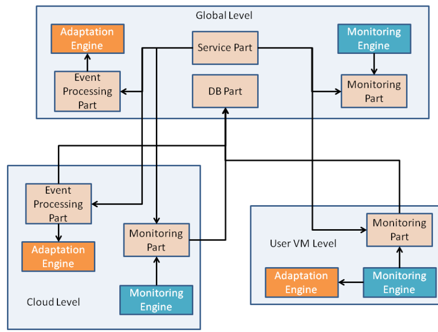

Figure 3: Most suitable physical framework deployment.

less frequent than the other parts. Fig. 2 depicts this partitioning of the framework architecture.

By considering the distribution options for the SBA management system and proposed framework, an optimal architecture is suggested (see Fig. 3) involving the aforementioned 3 levels. Our framework is mainly positioned at the 2 top levels while only the monitoring subscription part is distributed at all levels. Further, different framework parts in a certain level are deployed in different VMs to enable the on-demand scaling of respective framework components.

At the global level, we have all framework parts deployed separately in different VMs. We do need the whole framework functionality, especially as the entry points to the whole SBA management system lie at this level. At the cloud level, only the EPP & monitoring parts should be deployed as focus is more on core framework functionality. At the VM level, we foresee only deploying the MP part as precious user resources would have been stolen by also incorporating the EPP part in this intrusive monitoring approach. As such, the event processing could still occur by moving at the cloud level and would enable reserving as much resources as possible for this processing.

6 EVENT PATTERN GENERATION

6.1 CAMEL to EPL Construct Mapping

As CAMEL is inspired by Esper EPL, all CAMEL event composition constructs map directly to EPL constructs. Table 2 shows this mapping via a restricted CAMEL concrete syntax form where specific information is filtered out (including the EP name plus the names of the attributes in this EP definition).

In this table, A and B represent any event kind, while T a timer, representing a time interval. The mapping between timer models in CAMEL and EPL is straightforward. Three timer kinds are supported:

- *WITHIN:* indicates a time period within which an EP should occur. First EP occurrence is enough to consider that whole event composition is satisfied.
- *WITHIN_MAX:* similar to previous timer but imposes a bound over the number of times the respective EP should occur. Thus, the whole composition is satisfied when either the bound is reached or the time period has been finished.
- *INTERVAL:* indicates the time period to pass so as to proceed with the next event composition part. For instance, an EP could indicate that we must see one occurrence of A and then wait until 60 seconds until we consider that the EP is satisfied.

6.2 Camel to EPL Mapping

As an EP can be a composition of other EPs, any kind of composite event expression can be specified in CAMEL and then be completely mapped to EPL. The latter mapping follows a top-down process by considering an event composition in an EP as a tree that can be processed from its single root node down to its leaves. The leaf nodes correspond to single events, while intermediate or root node map to composite events with a certain operator applying to them.

The pseudo-code of the transformation algorithm can be found in https://drive.google.com/open?id=1TPwF64GtgA0iFbz6c3IzJYqE_IktspIL. As it can be seen, the algorithm checks the event type concerned and then employs specific event handling code to process it. Single events are handled by a function analysed in the next subsection.

For unary EPs, we obtain the String-based EPL representation of the single event referenced by calling recursively the same algorithm, and then we condition over the operator involved to apply the respective transformation logic encoded in Table 2 and thus finally return the EPL representation of this unary EP.

For composite EPs, the logic is more complicated as we deal either with 2 events or one event and timer. In particular, we obtain the String representation of the left and right EP parts by checking whether they map to an event or a timer. In case of an event, we call recursively the algorithm to obtain its EPL representation. For a timer, we call the *processTimer* function to obtain its EPL representation. As the latter transformation is trivial, we omit its presentation. Once we obtain the EPL representations of both left and right EP parts, we condition over the binary EP operator to apply the transformation logic encoded in Table 2 and thus finally return the binary EP's EPL representation.

Table 2: Mapping of CAMEL to Esper EPL Constructs.

CAMEL Construct	**EPL Construct**	Explanation
A EVERY	every A	every occurrence of A
A NOT	not A	non-occurrence of A
A REPEAT Y	[Y] A	Y times A has occurred
A WHEN T Y	[Y] A until T	Y times A has occurred within time period designated by time T
A AND B	A and B	logical conjunction of A and B
A OR B	A or B	logical disjunction of A and B
A XOR B	(A and not B) or (not A and B)	exclusive occurrence of A or B
A PRECEDES B	A -> B	A followed by B
A REPEAT_UNTIL B Y1 Y2	[Y1:Y2] A until B	A should occur from Y1 to Y2 times until B occurs

6.3 Event Correlation

Previous sub-section presented the core logic for transforming a CAMEL EP into an Esper EPL expression. Intentionally, that presentation did not explain how single events are transformed into EPL expressions, as events in an EP statement need to be correctly correlated which then requires a special handling of their transformation. The goal of this sub-section is to explain how this handling takes place.

Before explaining this, we need to explicitly determine what an event correlation signifies: it indicates that the events involved in it should be associated with either the same measured components or components connected in the SBA dependency hierarchy. For instance, two events can focus on the same application component or one event can be related to a component and the second event to the VM hosting it. Without such correlation, we risk that we react on wrong EPs. For example, suppose a certain EP includes two events A and B, both at the level of a single application, which could be detected for two different applications. Suppose further that Esper detects event A mapping to the first application and event B to the second. Without properly correlating A and B, it will then be inferred that the EP has occurred and the adaptation action mapped to that EP needs to be triggered.

6.3.1 Measurement Representation in EPL Events

The need to handle correlations impacts the way measurements are represented as information concerning the measured component should be already present and copied accordingly in the event's internal representation in Esper. Thus, as our system receives measurements, these measurements must carry information supporting us in their mapping to single events to be then processed by Esper for EP detection purposes.

Based on the above, we assume the following: (a) *Metric Subscriber* subscribes only to metrics based on their name; (b) *Metric Publisher* publishes measurements for metrics that might be equally named. So, the published measurement information must be sufficient to enable the framework to identify exactly the object being measured so as to distinguish between measurements of the same metric.

Another assumption is that our EP detection framework is decoupled from the dependency knowledge it should possess at the instance level that would have to be drawn from the SBA management system via, e.g., its models@runtime component (Blair et al., 2009). This enables it to connect without any effort to different management systems and not to be tightly coupled with just one. Such a decoupling requires to encode this dependency information inside every measurement. As such, we pay the small penalty of minor duplication of the information being published.

The measurement information published, mapping to our internal event representation in Esper, includes the next information pieces that map to both the SBA type and instance topology: (f1) the metric's name (e.g., *MeanResponseTime*); (f2) the metric's value; (f3) the measurement timestamp; (f4) the name of the SBA concerned; (f5) the name of the component measured; (f6) the name of the component instance; (f7) the name of the VM measured; (f8) the name of the VM instance.

Values for fields f1-f4 must be always supplied, while, depending on the level and kind of component measured, only values for some of the other fields must be provided based on the following cases mapping to the type of measurement:

- C1 – *ApplicationMeasurement:* as measurement concerns whole SBA, no extra fields are needed.
- C2 – *VMMeasurement:* the measurement concerns one VM. 2 sub-cases can hold: (C2.1) only field f7 must be supplied as the measurement concerns the VM type (e.g., m1.medium) and not its instance; (C2.2) the measurement concerns the VM instance (e.g., m1.medium_inst1), so we must provide both fields f7 & f8 as the type of the VM instance concerned must be supplied.
- C3 – *ComponentMeasurement:* it concerns a certain component. 2 sub-cases can hold: (C3.1) it concerns the component type (e.g., "InvoiceNinja"). So, apart from field f5, we must supply field f7 (provide, e.g., the value of "m1.medium") to explicate for which component-to-VM deployment the current measurement holds as one component might be logically deployed into mul-

tiple VMs within the same deployment topology; (C3.2) the measurement concerns a component instance (e.g., "InvoiceNinja_Inst1"). Thus, all fields must be supplied to cover the deployment of both that component instance at the instance level and its (component) type at the type level. As such, in the running use case, the following measurement field values will be supplied: f5="InvoiceNinja", f6="InvoiceNinja_Inst1", f7="m1.medium", f8="m1.medium_inst1".

6.3.2 Single Event CAMEL-to-EPL Transformation

The uniform way we represent events in Esper enables us to map single CAMEL events to a String EPL representation. Such mapping, however, must consider the correlation between events at the same EP. This can be achieved by considering both the CAMEL application topology at the type level and the runtime information about the instance level for that topology, obtained from the respective measurements retrieved.

However, for the latter, we need to distinguish between: (a) measurement evaluation information examined at event processing time; (b) event correlation information generated at event specification time. In the second case, we must perform the correlation by considering that each component or VM will map to a certain instance, not known at design time but determined by the first event for which the evaluation will be performed. This is where contextual information comes into play. In particular, we use a context object (see previous pseudo-code) to remember the first event referring to the instance of a component/VM. Thus, relevant events subsequently processed for the current EP at hand will be correlated via information pertaining to that event and captured by that object.

The main logic of the transformation from CAMEL single events to the developed *Measurement* class in Esper can be seen in the pseudo-code available at https://drive.google.com/open?id=1LepdhHnSm71r87zUrCzMEAjgtmSL3tDO.

In a nutshell, the algorithm, by also exploiting the contextual information dynamically created and expanded, attempts first to generate the basic EPL event description part, common across any single event to be processed, comprising: (a) an identifier to properly identify the event and be able to correlate it with the next ones in the processing order; (b) the values of fields f1-f4; (c) depending on the type of component concerned, values of fields f7 and/or f5. Concerning the third information piece, we have already indicated the different cases that might hold at the type level (see Cases C3.1 and C2.1) with the sole exception that when we have only the application being measured (Case C1), none of these 2 fields must be specified.

When we face a composite metric concerning the type level, processing stops. Otherwise, if need to address a raw metric concerning the instance level, processing continues based on sub-cases C2.2 and C3.2.

For C2.2, we check if the instance of the VM was already involved in a previously processed event. If this holds, we get the respective reference from the context object and expand the internal condition of the EPL event String to be generated. Otherwise, the current event is the reference event for this VM instance; we then inform the context object about this to cater for processing the next events in the same EP.

For C3.2, we similarly check if the component instance has been already stamped in the context object. If this holds, we expand the EPL internal event condition with correlation information. However, this time we correlate both this component instance and the VM instance on which it was deployed based on the respective events of reference in the context object. Otherwise, we mark the current event as the reference event of the component instance. In case the VM instance has been already referenced, we still expand the current EPL String with the reference of the respective event. If not, we also make the current event as the reference event of this VM instance.

6.4 Use Case Application

We apply our CAMEL-to-EPL transformation algorithm on the running use case. We focus on adaptation rule R_1, the most complicated in terms of handling.

By considering the CAMEL model of R_1's unary EP, there is a tree with 3 levels where there is one root node (unary EP) with *EVERY* composition operator and one intermediate node (binary EP) with *AND* as composition operator. By applying the *processEP* transformation method on the top EP, we will first generate the EPL statement for the intermediate node by calling the same method recursively and then we will apply the *EVERY* operator based on the semantics of Table 2 to produce the final EPL statement. So, if intermediate node's EPL statement is "X", the final EPL statement would be: "every (X)". Now, let's focus on how the intermediate node will be processed.

We will first process the left binary EP part mapping to the raw CPU utilisation condition by calling recursively the emphprocessEP method. As this condition maps to a single event, eventually the *processSingleEvent* method will be called to construct first the basic EPL event string by including the fields f1-f4:

```
ev1=Event(metric='CPUUtilisation' and value >= 85 and application='SendInvoice' and vm='m1.medium'
```

We will then discover that Case 2.2 holds such that

we must only handle the instance of ‘m1.medium’ VM. After checking the context object, there is no previous reference to that instance as this is the first event being processed. So, the context object will be updated to include a reference to this event (i.e., “ev1.vmInstance”) in case we need to refer to the instance of ‘m1.medium’ VM, while the String of the event EPL specification will be ended with ”)”.

Next, we will process the binary EP’s right part related to the condition on raw memory utilisation by involving again the same call sequence (*processEP* followed by *processSingleEvent*). The first part of this event will be similar to that of the previous one (only metric and condition sub-parts differing):

```
ev2=Event(metric='MemoryUtilisation' and value >= 90 and application='SendInvoice' and vm='m1.medium'
```

In this case, we deal with the same VM as in the previous event by also handling the instance level. Thus, we will again check the context object and find out that the event of reference for an instance of the ‘m1.medium’ VM will be “ev1.vmInstance”. This will lead to expanding and subsequently finalising the respective EPL statement as follows:

```
ev2=Event(metric='MemoryUtilisation' and value >= 90 and application='SendInvoice' and vm='m1.medium' and vmInstance=ev1.vmInstance)
```

Finally, having the 2 EP parts already determined, we just need to combine them based on *AND* operator semantics in Table 2. This will eventually lead to producing the final EPL statement for the binary EP which, when enhanced with the application of the *EVERY* operator, will take the following final form:

```
every(ev1=Event(metric='CPUUtilisation' and value >= 85 and application='SendInvoice' and vm='m1.medium') and ev2=Event(metric='MemoryUtilisation' and value >= 90 and application='SendInvoice' and vm='m1.medium' and vmInstance=ev1.vmInstance))
```

Due to space restrictions and the simpler complexity in their processing, we will not elaborate on the rest of the adaptation rules of the running use case.

7 CONCLUSIONS AND FUTURE WORK

This paper has proposed a novel SLO evaluation framework for SBAs that is scalable and distributable across the whole SBA architecture. This framework relies on a modular architecture with different realisation options at the physical level. It also encompasses the well-known Esper CEP engine enabling the scalable processing of complex EPs even in a centralised setting. It also relies on CAMEL’s SRL sub-DSL enabling it to: (a) fully specify the event parts of SBA adaptation rules; (b) be more focused on processing and detecting only those EPs related to the SBA desired adaptation behaviour; (c) to be synchronised with the modifications performed on SBA adaptation rules. The latter is also supported by introducing a REST service, encapsulating functionality to manage the EPs that need to be detected in this framework.

The following research work directions are planned: (a) thorough SLO evaluation framework assessment including its various physical distribution alternatives; (b) comparison of Esper with other CEP engines to assess which engine is more suitable in our context; (c) possible extension of the framework to be configured to exploit any CEP engin via the incorporation of appropriate abstraction mechanisms.

ACKNOWLEDGEMENTS

This work is supported by the Unicorn project that has been funded within the European Commissions H2020 Program under contract number 731846.

REFERENCES

Bahga, A. and Madisetti, V. K. (2013). Performance evaluation approach for multi-tier cloud applications. *Journal of Software Engineering and Applications*, 6(02):74.

Blair, G., Bencomo, N., and France, R. B. (2009). Models@ Run.Time. *Computer*, 42(10):22–27.

Emeakaroha, V. C., Netto, M. A., Calheiros, R. N., Brandic, I., Buyya, R., and Rose, C. A. D. (2012). Towards autonomic detection of sla violations in cloud infrastructures. *Future Generation Computer Systems*, 28(7):1017 – 1029.

Ferry, N., Chauvel, F., Rossini, A., Morin, B., and Solberg, A. (2013). Managing multi-cloud systems with CloudMF. In *NordiCloud*, pages 38–45. ACM.

Higashino, W. A. (2016). *Complex Event Processing as a Service in MultiCloud Environments*. PhD thesis, The University of Western Ontario.

Kritikos, K., Domaschka, J., and Rossini, A. (2014). SRL: A Scalability Rule Language for Multi-cloud Environments. In *CloudCom*. IEEE.

Kritikos, K., Zeginis, C., Griesinger, F., Seyold, D., and Domaschka, J. (2017). A Cross-Layer BPaaS Adaptation Framework. In *FiCloud*. IEEE.

Leitner, P., Inzinger, C., Hummer, W., Satzger, B., and Dustdar, S. (2012). Application-level performance monitoring of cloud services based on the complex event processing paradigm. In *SOCA*, pages 1–8. IEEE Computer Society.

Ludwig, H., Stamou, K., Mohamed, M., Mandagere, N., Langston, B., Alatorre, G., Nakamura, H., Anya, O., and Keller, A. (2015). rSLA: Monitoring SLAs in Dynamic Service Environments. In *ICSOC*, volume 9435 of *LNCS*, pages 139–153. Springer.

Wu, C., Zhu, Y., and Pan, S. (2013). The sla evaluation model for cloud computing. In *ICCNCE*. Atlantis Press.

Zeginis, C., Kritikos, K., and Plexousakis, D. (2015). Event pattern discovery in multi-cloud service-based applications. *IJSSOE*, 5(4):78–103.