





# CEP-Based SLO Evaluation

Kyriakos Kritikos<sup>1</sup> , Chrysostomos Zeginis<sup>1</sup> , Andreas Paravoliassis<sup>2</sup>,  
and Dimitris Plexousakis<sup>1</sup>

<sup>1</sup> Institute of Computer Science - FORTH, Heraklion, Greece

{kritikos,zegchris,dp}@ics.forth.gr

<sup>2</sup> Computer Science Department, University of Crete, Heraklion, Greece  
csd3031@csd.uoc.gr

**Abstract.** Modern service-based applications (SBAs) operate in highly dynamic environments where both underlying resources and the application demand can be constantly changing which external SBA components might fail. Thus, they need to be rapidly modified to address such changes. Such a rapid updating should be performed across multiple levels to better deal, in an orchestrated and globally-consistent manner, with the current problematic situation. First of all, this means that a fast and scalable event generation and detection mechanism should exist to rapidly trigger the adaptation workflow to be performed. Such a mechanism needs to handle all kinds of events occurring at different abstraction levels and to compose them so as to detect more advanced situations. To this end, this paper introduces a new complex event processing framework able to realise the respective features mentioned (processing speed, scalability) and have the flexibility to capture and sense any kind of event or event combination occurring in the SBA system. Such a framework is wrapped in the form of a REST service enabling to manage the event patterns that need to be rapidly detected. It is also well connected to other main components of the SBA management system, via a publish-subscribe mechanism, including monitoring and the adaptation engines.

**Keywords:** Complex event processing · Event pattern · Detection Service

## 1 Introduction

Due to tough competition, organisations can survive if they can improve their services to exhibit better service levels with less cost. Such organisations need to also possess a smart infrastructure and a dedicated devops team to appropriately re-configure the services offered as well as manually intervene in unanticipated, problematic situations. As such, a lot of effort is spent in maintaining such an infrastructure while an increasing management and operational cost also incurs.

Fortunately, the advent of cloud computing has revolutionised the way resource management is performed. Nowadays, organisations can outsource their infrastructure management to cloud providers that promise to offer infinite,

cheap commodity resources on an on-demand basis. Due to flexible resource management and the capability to scale a cloud-based system, organisations can now optimise their services at the infrastructure level. However, still effort is needed at higher-levels of abstractions. In particular, external SaaS services need to be dynamically selected to realise part of the required functionality while the whole system needs to be adapted.

In the literature, it has been advocated [11] that dynamic SBA adaptation should be performed in a cross-layer manner by also putting in place, as a prerequisite, a suitable monitoring framework. Cross-layer adaptation is needed for various reasons. First, as the service system itself includes multiple levels that must be appropriately controlled. Second, as the individual adaptation at one level can influence, impact or even negate the adaptation results at adjacent levels, leading to a vicious re-adaptation cycle. Cross-layer monitoring is also needed to propagate and aggregate up to higher-levels measurements produced in lower levels so as to cover measurability gaps.

As the glue between monitoring and adaptation, there is a need for a rapid and scalable Service Level Objective (SLO) evaluation framework able to transform measurements to events and subsequently detect event patterns that can lead to performing adaptation actions in the context of adaptation rules. Such a framework should also exhibit suitable accuracy levels by correctly correlating the events occurring based on their metrics and measured objects. It should also be able to detect and correlate events which should map to both the type and instance level in the managed SBA system.

In this work, such a framework has been carefully designed and realised, by conforming to all the aforementioned requirements. In particular, the framework architecture was initially designed by considering principles, such as service-orientation, and by carefully decoupling framework parts subject to scaling. Based on this architecture and the appropriate selection of the right, existing components and tools, a respective framework was then implemented and integrated in our existing SBA monitoring and adaptation framework [21]. Such an integration is loosely coupled as our SLO evaluation framework can be in principle connected to any monitoring and adaptation engine.

The developed framework relies on the CAMEL domain-specific language (DSL), able to capture various aspects in the cloud-based application lifecycle management, including the monitoring and adaptation ones. In particular, this DSL is expressive enough to specify complex event patterns, where each event maps to a metric condition, and associate them with respective sets of adaptation actions that must be triggered to adapt the SBA in a cross-layer manner. CAMEL also covers well the monitoring aspect via its capability to specify how composite metrics are aggregated and to associate metrics with the (e.g., service) component that they measure. As it will be shown, such information is essential to have the ability to correlate events in the context of event pattern detection.

The proposed framework relies on the Esper Complex Event Processing (CEP) engine. This engine is quite scalable with the capability to process thousands or even millions of events. Due to the way our architecture has been

designed, this engine can be scaled when its processing limits are reached, enabling our framework to really scale at those parts where most of the load is directed.

The rest of the paper is structured as follows. The next section provides a use case scenario which is used as a running example across the whole paper, while Sect. 3 reviews the related work. Section 4 provides background information necessary for the comprehension of this paper contribution. Section 5 analyses the proposed framework architecture and supplies some implementation details. Section 6 explains the way the event pattern specification is generated by accounting also on how the events of the pattern should be correlated. Finally, the last section concludes the paper and draws directions for further research.

## 2 Use Case

The use case, which is used as a running example across the paper, has been drawn from the CloudSocket project<sup>1</sup> which deals with the management of Business Processes (BPs) in the Cloud. This use case concerns the development of a service-based BP as a service (BPaaS), named as “SendInvoice”, which offers the functionality of invoice generation and sending. This BPaaS maps to a supporting BP which can really provide appropriate automation level within a small or medium-sized organisation with respect to the management of invoicing. In this respect, it makes sense to develop and offer this BP in the cloud as the demand for this BP would be quite high.

The “SendInvoice” BPaaS exploits two main services: (a) an external SaaS dedicated to the customer relationship management (CRM) named as YMENS CRM; (b) an internal component for invoice management called “Invoice Ninja” which has been purchased and deployed in the Cloud in an Amazon EC2 VM named as “m1.medium”. These two services are then combined into a technical workflow which is deployed in the cloud and includes tasks that map to certain methods/functionality of these services.

The topology of the initial deployment of the “SendInvoice” BPaaS in the Cloud, as specified also in CAMEL, is depicted in Fig. 1 where both the type and instance levels are shown. As it can be observed, only one instance of the “InvoiceNinja” (software) component, named as “InvoiceNinja\_inst1” has been deployed in one instance of the “m1.medium” VM named as “m1.medium\_inst1”.

Suppose, now, that the organisation offering the “SendInvoice” BPaaS, i.e., a Cloud Broker, needs to control its execution in order to sustain a suitable service level that has been agreed with any of its customers in the context of an SLA. As the set of customers can grow, the Cloud Broker needs to control the amount of resources dedicated to “Invoice Ninja” as well as have the ability to replace the CRM service when its service level is not any more acceptable. To this end, it specifies the following set of adaptation rules (specified in CAMEL but abstracted away due to space limitation reasons) which scale out “Invoice Ninja”

---

<sup>1</sup> [www.cloudsocket.eu](http://www.cloudsocket.eu).

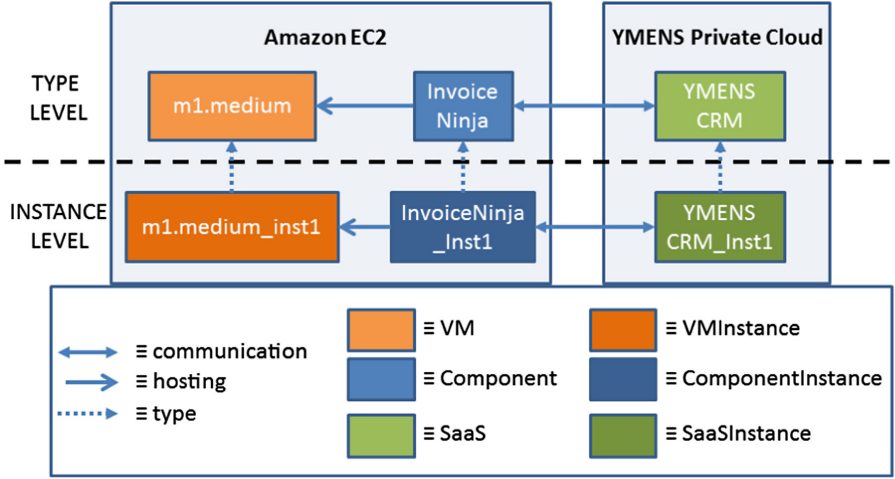


Fig. 1. The topology of the “SendInvoice” BPaaS.

or replace “YMENS CRM” with another SaaS. These rules are then given as input to the BPaaS Execution Environment of the CloudSocket platform which takes care of performing the respective adaptation actions required. The CEP-based SLO Evaluation framework proposed can be part of this environment by replacing an equivalent component which currently supports only adaptation at the IaaS level (mostly scaling actions).

$$R_1 : raw\_cpu(m1.medium) > 80\% \wedge raw\_mem(m1.medium) > 90\% \Rightarrow scale - out(IN)$$

$$R_2 : mean\_rt(YC) > 20 \wedge mean\_avail(YC) < 99.99\% \Rightarrow replace(YC)$$

$$R_3 : mean\_cpu(m1.medium) > 70\% \wedge mean\_rt(IN) > 20 \Rightarrow scale - out(IN)$$

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 the “Invoice Ninja” component and  $YC$  the “YMENS CRM” component.

Rules  $R_1$  &  $R_3$  focus on scaling out the “Invoice Ninja” component. The first rule attempts to immediately scale this component when one of its instances is severely overloaded. On the other hand, the third rule focuses on scaling out this component when its global status across all of its instances seems to be overloaded.

Rule  $R_2$  attempts to replace the “YMENS CRM” external SaaS when both its mean response time is more than the threshold posed and its availability drops under a certain level. The replacement service is not specified as the system should dynamically find its replacement according to the current situation.

The whole specification of the use case in CAMEL, including the topology and the adaptation model of the “SendInvoice” BPaaS, can be found at: <https://drive.google.com/file/d/0B1oLQgQCVlqramYwa1hDZmtnSGc/view?usp=sharing>.

### 3 Related Work

Various approaches have been proposed in complex event processing and event pattern detection. Most rely on CEP engines that detect complex events continuously and build correlations and relationships between them, such as causality and timing ones. The detection of complex patterns is based on various techniques applied either over event streams [18, 19] or in an offline [9, 13] manner.

Statistical event detection approaches mainly exploit a user-defined minimum frequency or support (*minsup*). The springboard of all these approaches is the *Apriori* algorithm [1]. This algorithm produces the set of all significant association rules (rules relating a set of variables) between items in a large transactions database with a *minsup*. In [16], the authors introduce a method for discovering frequent event patterns, as well as their spatial and temporal properties in sensor networks, exploiting data mining techniques. Provided that events are put into a spatial and temporal context, the authors correlate certain event types on a sensor node with context events in a confined neighborhood in the recent past. Thus, a pattern of events is discovered whenever this pattern’s frequency surpasses a *minsup*. In [14], the authors propose the *Lossy Counting* widely used algorithm. This is an one-pass algorithm that computes approximate frequency counts of elements in a data stream and involves grouping the row items into blocks or chunks and counting within each chunk.

Temporal event processing approaches exploit the temporal relations among an input stream’s events. Such approaches can be very useful for deriving implicit information for the temporal ordering of raw data and predicting the future behavior of the monitored application. In [3] the authors introduce a formal framework for expressing data mining tasks involving time granularities, as well as algorithms for performing these tasks. Time constraints are injected into the system to bound the distance between an event pair in terms of time granularity. For instance, event  $e_2$  must happen within two minutes after the occurrence of event  $e_1$  so as to consider  $e_1, e_2$  an event pattern. In [15] a temporal data mining approach is presented for data that cannot fit in memory or are processed at a faster rate than the generation one. The proposed sliding window model slides forward in hops of batches, while only a single batch is available for processing.

Moreover, logic-based approaches exploit inferencing to discover patterns defining respective association rules. In [17] a pattern discovery approach is proposed mapping logical equivalences based on propositional logic. In particular, a rule mining framework is introduced, generating coherent application domain independent rules for a given dataset that do not require setting an arbitrary *minsup*. The logic-based approach in [2] proposes an event calculus (EC) dialect, called *RTEC*, for efficient run-time recognition that is scalable to large data streams and exploits main EC predicates to discover specific activities. In our previous work [20], we have introduced a logic-based algorithm for discovering valid event patterns causing specific SLO violations. These event patterns interrelate events produced during the SBA’s execution and can be further exploited to enrich the adaptation rules defined by experts. This paper

goes a step further introducing a scalable and high-performance complex event processing framework that can realise and extend the event pattern detection feature.

Finally, other approaches also consider Business Process Management (BPM) when dealing with SLO evaluation. For instance, [6, 10] propose solutions (mainly scaling actions) for the optimization of Business Processes that are executed on virtualized environments. A similar approach is proposed in [8], where the authors apply data mining techniques to predict QoS and thus identify the correlation between the design and provisioning alternatives.

## 4 Background

### 4.1 Esper

Esper<sup>2</sup> is a stream-oriented CEP engine that provides the SQL-like and rich Event Processing Language (EPL). EPL enables expressing complex (event) matching conditions that include temporal windows, joining of different event streams, as well as filtering, aggregation, sorting and pattern detection. The proposed framework exploits it for the event pattern detection.

### 4.2 CAMEL

CAMEL is a multi-DSL, developed in the context of the PaaSage<sup>3</sup> project to deal with the specification of multiple aspects in the multi-cloud applications lifecycle. It integrates already existing languages, like CloudML [7], as well of new languages developed with that project, like the Scalability Rule Language (SRL) [12]. The aspects covered by CAMEL mainly include: deployment, requirement, metric, scalability, provider and organisation aspects.

This paper focuses mainly on the metric and scalability aspects covered by the SRL sub-DSL of CAMEL. The metric package attempts to cover all measurement details that need to be specified for a non-functional metric, like formulas, functions, units of measurement plus measurement schedules and windows. This package is also able to specify conditions on metrics that can be exploited to specify SLOs as well as non-functional events in scalability rules.

The scalability aspect is covered via specifying scalability rules that map single events or event patterns to one or more scaling actions. Scaling actions can be either horizontal or vertical. Horizontal scaling actions include scale-out and scale-in actions while vertical actions include scale-up and scale-down.

The conceptualisation of events and event patterns is depicted in Fig. 2. Events can be single or composite. Single events can be further distinguished in functional and non-functional. Functional events map to a certain functional fault, like an application component failure. Non-functional events are associated

---

<sup>2</sup> <http://www.espertech.com/esper/>.

<sup>3</sup> <https://paasage.ercim.eu/>.

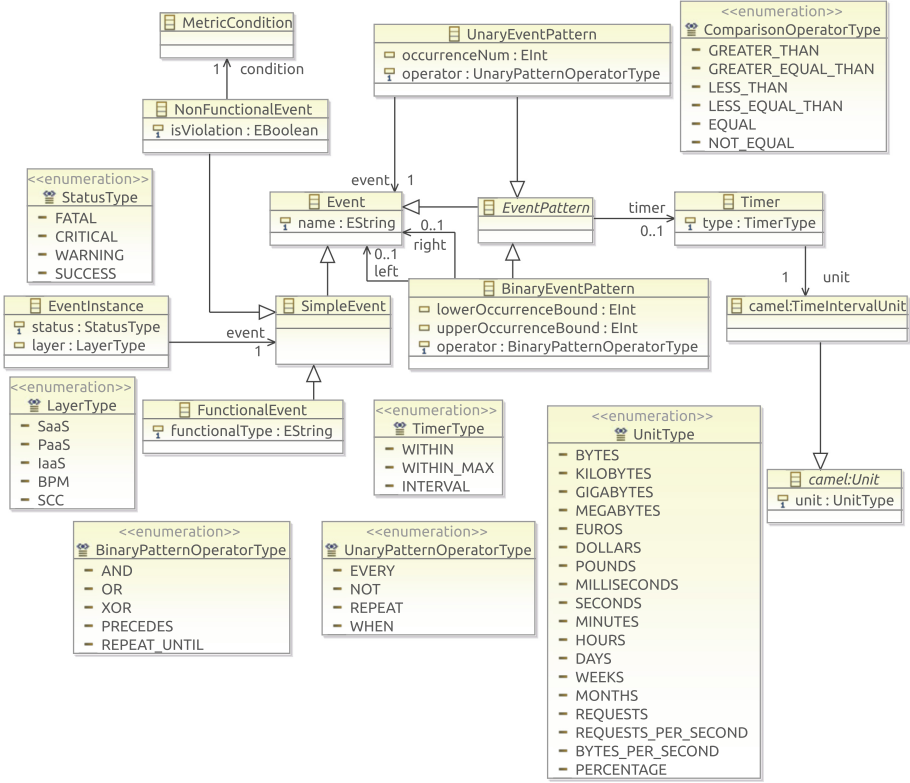


Fig. 2. The event pattern part of the SRL meta-model.

to a metric condition violation. A composite event maps to a logical or time-based combination of one or more events in the form of an event pattern. As such, such a combination is associated with respective logical and time-based operators. Both binary and unary operators can be defined which leads to producing unary and binary event patterns, respectively. Logical operators include AND, OR, NOT and XOR. Time-based operators have been inspired by Esper's EPL and include many of the operators defined in that language (e.g., REPEAT).

As an event pattern is also a kind of event, patterns can be recursively defined. This means that, for example, when applying a binary logical operator (e.g., AND) over a certain binary event pattern, the first event could be single and the second could be another event pattern. For instance, suppose that the event pattern  $EP_1: A \wedge (B \vee C)$  must be defined. To specify  $EP_1$ , we need to define that the first event is  $A$ , the second event maps to the event pattern  $EP_2$  and that the logical operator applied is  $\wedge$ . The second event pattern  $EP_2$  would then be specified as the application of the  $\vee$  operator over two events,  $B$  and  $C$ .

As another example, consider the case of adaptation rules  $R_1$  &  $R_3$  which have the same consequent (i.e., adaptation action). In order to reduce the number

of rules that need to be checked and triggered by the system, these two rules could be combined into one. In that case and by considering that the name of each rule could also be the name of the respective event to be defined, then a more composite rule  $R_4$  would be constructed which would map to the complex event pattern  $(R_1 \vee R_3)$ .

Via the recursive definition of events, more complex and advanced situations can be captured in respective rules. This should not stop to the case of scalability rules, but could cover any adaptation rule kind. This has been performed by the CloudSocket project (see footnote 1) [5] via an SRL extension. This extension can specify any adaptation rule kind at different abstraction levels. The event part of the rule specification was left as is, but the action part was extended to specify a workflow of adaptation actions that can be performed at the levels of infrastructure, platform, service and business process. As such, this extension fits well to the latest research trends in service computing that require specifying, executing and managing cross-layer rules to more effectively deal with the adaptation of cross-level SBAs in both simple and more advanced problematic situations.

In the context of this work, only the event part of an adaptation rule is considered due to intended functionality to be delivered. The CEP engine developed just detects the need to trigger a rule and then informs the rule execution component, e.g., an *Adaptation Engine*, to enact that execution of that rule.

## 5 SLO Evaluation Framework

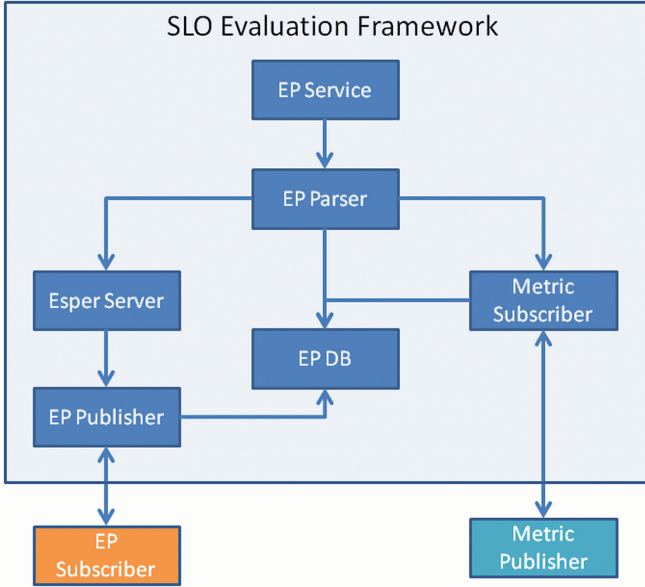
### 5.1 Framework Analysis

The proposed SLO Evaluation Framework relies on the modular architecture depicted in Fig. 3. This architecture comprises three main levels: (a) interface; (b) core logic; (c) database (DB). At the interface level, the main actions (add, update, delete) that can be performed over an event pattern (EP) have been wrapped into the form of a REST service, called, *EP Service*, able to parse CAMEL/SRL fragments mapping to the specification of these patterns. Each action, when called, then has an impact over the core logic level of the framework.

At this second level, there is a main component, called *EP Parser*, which is responsible for processing the EPs obtained from the *EP Service*. Depending then on the action requested, different interactions take place at this level.

*EP Addition.* In case of adding a new EP, the *EP Parser* transforms it into an EP, specified in the EP language of the CEP framework, which is then registered in the server of that CEP framework, called *CEP Server*, so that it can be immediately detected. The names of metrics referenced by the EP, i.e., directly involved in the conditions of the EP's events, are also sent to the *Metric Subscriber* which not only informs its local metric list but also registers for subscribing to such metrics, when they are new, in the *Metric Publisher*. In parallel to this registration, the updated metric list of the *Metric Subscriber* is stored in the *EP DB* for fault-tolerance and rapid recovery reasons. The *Metric Publisher* is responsible





**Fig. 3.** The architecture of the SLO evaluation framework.

for publishing the values of metrics monitored to potential subscribers. As such, it can well map to a *Monitoring Engine* of a SBA management system. Once both the new EP and its respective metrics are registered in the corresponding system parts, the EP addition has been successful. So, the *EP Parser* stores the new EP in the *EP DB* not only for recovery reasons but also to gather statistics about EPs, while being detected by the *Esper Server*. The *EP DB* has been realised in the form of a model repository able to store, query and manipulate models of CAMEL, especially EPs, along with their statistics.

*EP Deletion.* In case of EP deletion, the EP is first fetched from the *EP DB*. Then, in parallel, the *EP Parser* informs both the *CEP Server* and the *Metric Subscriber* to update their structures and take further actions. The *CEP Server* just deregisters the EP's EPL specification. On the other hand, after checking that the EP metrics to be removed are not exploited in other EPs, the *Metric Subscriber* is informed to unsubscribe to these metrics to reduce the system load.

*EP Update.* In case of EP update, the produced EPL statement by the *EP Processor* is used to update the previous one. In addition, the *Metric Subscriber* is informed for adding or removing metrics which are or not needed any more (by any EP), respectively.

While the above actions can take place through the interaction of an external agent/user with the proposed Framework, we highlight that, in principle, the same interactions could be differently achieved, e.g., via a publish-subscribe

mechanism. As the respective functionality has been realised, we could easily switch from one to another mechanism or have both available at the same time.

As there are internally performed actions inside the framework, while it is running, these are now explicated in detail below.

As the *Metric Subscriber* subscribes to metrics, it can asynchronously receive measurements for such metrics from the *Metric Publisher*. Such measurements are then transformed into events which are fed into the *CEP Server*. Once all suitable events are received by the latter component, it can detect one or more EPs. When this occurs, this component will inform the *Event Publisher*.

The *EP Publisher* is responsible for publishing events to interested subscribers, named as *EP Subscribers*. Such subscribers could be adaptation engines responsible for executing the respective adaptation rule triggered, as, e.g., specified in CAMEL. Apart from this publication, the *EP Publisher* also updates the entries in the *EP DB* to modify the respective statistics of the EP(s) concerned.

The proposed architecture exploits publish-subscribe mechanisms to both receive some events/measurements and publish other kinds of events (e.g., EPs). In this respect, it can actually interact with multiple components that might be willing to obtain information from or feed information to this framework. For instance, adaptation responsibility for an SBA management system could be split into multiple instances of an *Adaptation Engine* to balance the respective load. All these instances could then subscribe to the *EP Publisher* to manage their own part of the adaptation space, i.e., only those EPs that concern them.

The presented architecture is logical. This means that it can be flexibly distributed at the physical level. For instance, we could have multiple instances of the framework part that involves the *CEP Server* and the *Event Publisher* to load balance the event workload entering the framework. Alternatively, we could scale out the whole framework into parts that focus on different EP partitions. For example, the SBA management system could be split similarly into different parts, where each part could be devoted to a subset of all SBAs managed. Each system part could be then associated to one instance of the SLO Evaluation Framework, thus mapping only to the EPs of the SBAs that need to be handled.

## 5.2 Implementation

All framework components have been implemented in Java. The CEP engine exploited is Esper, version 5.3.0. For the publish-subscribe mechanism, the 0-MQ<sup>4</sup> messaging middleware has been exploited that incurs less overhead with respect to other messaging middleware realisations. The *EP DB* has been realised as a model repository implemented via the CDO technology<sup>5</sup> which provides suitable and robust mechanisms for model persistence and lazy loading as well as the HQL language to enable posing queries at a higher abstraction level than pure SQL. The *EP Service* has been implemented via the Jersey<sup>6</sup> java library.

<sup>4</sup> [zeromq.org](http://zeromq.org).

<sup>5</sup> <https://eclipse.org/cdo/>.

<sup>6</sup> <http://jersey.github.io/>.

## 6 Event Pattern Generation and Detection

While it could be considered as straightforward to transform an event pattern in CAMEL into an EPL statement in Esper, this is by far not trivial as the events in an EP need to be correctly correlated. Correlation means that the events should be associated with either the same measured components or with components that are connected in the SBA dependency hierarchy. This also has an impact on the way measurements are represented as the information concerning the measured component should be already present and be then copied accordingly in the internal representation of the event in Esper.

Concerning the metric measurements, we have actually assumed the following: (a) the *Metric Subscriber* subscribes only to metrics based on their name; (b) the *Metric Publisher* publishes measurements for metrics that might be named equivalently. The latter means that the measurement information published should include sufficient information to enable the framework to identify exactly what object is being measured.

To decouple the proposed framework from the dependency knowledge it should possess, we assume that such dependency information is provided within the measurement information published. While this leads to some published information duplication, it translates to a loose integration of this framework with the SBA management system. Otherwise, the framework would need to connect to a `models@runtime` component [4] in that system to be informed constantly about both the type and instance level in the SBA dependency hierarchy.

The measurement information published includes: (f1) the metric’s name (e.g., *MeanResponseTime*); (f2) the metric value; (f3) the measurement timestamp; (f4) the name of the application/service concerned; (f5) the name of the component measured; (f6) the name of the instance of the component measured; (f7) the name of the VM measured; (f8) the name of the instance of the VM measured.

Values for fields f1–f4 are always present. Depending on the level and kind of component measured, only some of the values of the other fields need to be supplied based on the following cases mapping to the type of measurement:

- *ApplicationMeasurement*: here the measurement concerns the whole application so no additional fields are needed.
- *VMMeasurement*: here the measurement concerns a certain VM. There are two sub-cases holding now: (i) the measurement concerns the VM type (e.g., `m1.medium`) and not its instance. Then, only the field f7 has to be provided; (ii) the measurement concerns the VM instance (e.g., `m1.medium_inst1`). In this case, we need to provide both fields f7 & f8 as the type of the VM instance concerned needs to be provided.
- *ComponentMeasurement*: here the measurement concerns a certain (software) component. Again, two sub-cases might hold: (i) the measurement concerns the component type (e.g., “InvoiceNinja”). In this case, apart from field f5, we also need to provide field f7 (thus provide, e.g., the value of “`m1.medium`”) as one component might logically be deployed into multiple VMs within the

same deployment topology. As such, we need to explain for which deployment the current measurement holds; (ii) the measurement concerns a component instance (e.g., “InvoiceNinja\_Inst1”). In this case, all the fields need to be provided in order to cover both the deployment of that component instance at the instance level as well as the deployment of its (component) type at the type level. Thus, considering the running example/use case, the following values for the measurement fields will be provided: f5=“InvoiceNinja”, f6=“InvoiceNinja\_Inst1”, f7=“m1.medium”, f8=“m1.medium\_inst1”.

By explaining how measurements are structured and instantiated based on the kind of the component concerned, now we will explain the way EPs in CAMEL are transformed into EPL statements. We distinguish between two cases – ( $C_1$ ) all events in the EP refer to the instance level; ( $C_2$ ) all events in the EP refer to the type level. We do not consider a mixture of events from different levels as this does not make sense.

The  $C_1$  case maps to two sub-cases:

1. all events refer to the same component. Consider, for instance, the case of rule  $R_1$ . The respective EPL statement to be created for this rule would be the following:

```
every(ev1=Event(metric='CPUUtilisation' and value >= 80
and application='SendInvoice' and vm='m1.medium') and
Event(metric='MemoryUtilisation' and value >= 90 and appli-
cation='SendInvoice' and vmInstance=ev1.vmInstance and
vm='m1.medium'))
```

In this statement, we join these two events as streams based on their application, VM and VM instance fields. Via this join, we impose that the EP should hold for a specific application and VM but we do not care about which matched vm instance is concerned (as any instance needs to be matched here). Moreover, the presence of EVERY indicates that the pattern should be repeatedly inspected and not just once.

2. all events refer to different but related components. In this case, the EPL statement under construction needs to correlated the different components together. For instance, suppose that an alternative rule to  $R_1$  would attempt to scale the “InvoiceNinja” component when its raw CPU utilisation is above 80% and its response time is above 20 s. The respective statement generated for this alternative rule would take the following form:

```
every(ev1=Event(metric='CPUUtilisation' and value >= 80
and application='SendInvoice' and vm='m1.medium') and
Event(metric='ResponseTime' and value > 20 and vmIn-
stance=ev1.vmInstance and application='SendInvoice' and
vm='m1.medium' and component='InvoiceNinja'))
```

This EPL statement is more complicated as it needs to join two events for which we need to guarantee that they refer to the same application, VM and VM instance, where the first two fields are mapped to specific values. We also need to guarantee that the second event refers to the “InvoiceNinja” component but we do not care about the instance of that component as we guarantee that the same VM instance, as in the first event, has been used to deploy this particular instance of that component.

For the type level, we have the following two similar kinds of cases which, however, lead to the construction of simpler EPL statements.

1. all events refer to the same component. For instance, suppose that Rule  $R_2$  applies here. Then, the respective EPL statement to be constructed would take the following form:

```
every(ev1=Event(metric='MeanResponseTime' and value > 20 and
application='SendInvoice' and component='YMENS CRM') and
Event(metric='MeanAvailability' and value < 99.99 and applica-
tion='SendInvoice' and component='YMENS CRM'))
```

In this statement we just join two event streams based on their application and component which are clearly identified in the respective conditions.

2. the events refer to different but correlated components. For instance, suppose that Rule  $R_3$  needs to be applied. The respective EPL statement to be constructed will be the following:

```
every(ev1=Event(metric='MeanResponseTime' and value > 20
and application='SendInvoice' and component='InvoiceNinja' and
vm='m1.medium') and Event(metric='MeanCPUUtilisation' and value
> 70 and application='SendInvoice' and vm='m1.medium'))
```

So, we actually join again the two events by considering the following: (a) the join is made based on the application and VM fields; (b) for the first event, we need to identify the correct component concerned; (c) for the second event, we do not need to specify a respective software component as it concerns the infrastructure (VM) level.

Cases  $C_1$  and  $C_2$  with their 2 sub-cases have been exemplified via certain examples. In reality, our framework is able to go beyond the capabilities shown in these examples. It can process any kind of complex EP with an arbitrary nesting and any kind of operator from those captured by CAMEL. However, showing such a complex case needs substantial space and thus, it has been left out from the analysis in this paper.

## 7 Conclusions and Future Work

This paper has proposed a new SLO evaluation framework for SBAs that relies on a rich EP expression language, namely SRL, and on the well-known Esper CEP engine. This system has been designed based on a modular architecture where many of its parts can scale on demand. This system is also loosely coupled with the respective monitoring and adaptation engines that might be employed in a SBA management system. The management of EPs is wrapped into the form of a REST service enabling a respective SBA management system to be decoupled from underlying implementation peculiarities and manage the generation and handling of adaptation rules that contain such EPs.

Concerning future work, we plan to further evaluate the SLO evaluation framework and especially investigate its exact distribution points. We also plan to compare Esper with other CEP engines in order to reach an informed decision about which CEP engine is more suitable in our context. In fact, it can be interesting to create a system which can be configured to exploit different CEP engines by incorporating the appropriate abstraction mechanisms.

**Acknowledgments.** This work is supported by CloudSocket project that has been funded within the European Commission’s H2020 Program under contract number 644690.

## References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: VLDB, pp. 487–499 (1994)
2. Artikis, A., Sergot, M.J., Paliouras, G.: Run-time composite event recognition. In: DEBS, pp. 69–80. ACM (2012)
3. Bettini, C., Wang, X.S., Jajodia, S., Lin, J.-L.: Discovering frequent event patterns with multiple granularities in time sequences. *IEEE Trans. Knowl. Data Eng.* **10**(2), 222–237 (1998)
4. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. *Computer* **42**(10), 22–27 (2009)
5. Seybold, D., Griesinger, F., Kritikos, K., Gallo, A., Cacciatore, S., Popovici, A., Iranzo, J., Sosa, R., Utz, W., Falcioni, D.: Explanatory Notes: Final BPaaS Prototype. CloudSocket Project Deliverable D4.6–D4.8, June 2017
6. Euting, S., Janiesch, C., Fischer, R., Tai, S., Weber, I.: Scalable business process execution in the cloud. In: 2nd IEEE Conference on Cloud Engineering (IC2E), pp. 175–184. IEEE (2014)
7. Ferry, N., Chauvel, F., Rossini, A., Morin, B., Solberg, A.: Managing multi-cloud systems with CloudMF. In: NordiCloud, pp. 38–45. ACM (2013)
8. Ghosh, R., Ghose, A., Hegde, A., Mukherjee, T., Mos, A.: QoS-driven management of business process variants in cloud based execution environments. In: Sheng, Q.Z., Stroulia, E., Tata, S., Bhiri, S. (eds.) ICSOC 2016. LNCS, vol. 9936, pp. 55–69. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-46295-0\\_4](https://doi.org/10.1007/978-3-319-46295-0_4)
9. Hellerstein, J.L., Ma, S., Perng, C.-S.: Discovering actionable patterns in event data. *IBM Syst. J.* **41**(3), 475–493 (2002)

10. Janiesch, C., Weber, I., Menzel, M., Kuhlenkamp, J.: Optimizing the performance of automated business processes executed on virtualized infrastructure. In: 47th Hawaii International Conference on System Sciences (HICSS), pp. 3818–3826. IEEE (2014)
11. Kazhamiakin, R., Pistore, M., Zengin, A.: Cross-layer adaptation and monitoring of service-based applications. In: Dan, A., Gittler, F., Toumani, F. (eds.) ICSSOC/ServiceWave - 2009. LNCS, vol. 6275, pp. 325–334. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-16132-2\\_31](https://doi.org/10.1007/978-3-642-16132-2_31)
12. Kritikos, K., Domaschka, J., Rossini, A.: SRL: a scalability rule language for multi-cloud environments. In: CloudCom. IEEE (2014)
13. Magnusson, M.S.: Discovering hidden time patterns in behavior: T-patterns and their detection. *Behav. Res. Methods Instr. Comput.* **32**(1), 93–110 (2000)
14. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams, pp. 346–357 (2002)
15. Patnaik, D., Ramakrishnan, N., Laxman, S., Chandramouli, B.: Streaming algorithms for pattern discovery over dynamically changing event sequences. *CoRR*, abs/1205.4477 (2012)
16. Römer, K.: Distributed mining of spatio-temporal event patterns in sensor networks. In: EAWMS Workshop at DCOSS, pp. 103–116 (2006)
17. Sim, A.T.H., Indrawan, M., Zutshi, S., Srinivasan, B.: Logic-based pattern discovery. *IEEE Trans. Knowl. Data Eng.* **22**(6), 798–811 (2010)
18. Wang, D., Rundensteiner, E.A., Ellison, R.T.: Active complex event processing over event streams. *PVLDB* **4**(10), 634–645 (2011)
19. Wu, E., Diao, Y., Rizvi, S.: High-performance complex event processing over streams. In: SIGMOD Conference, pp. 407–418. ACM (2006)
20. Zeginis, C., Kritikos, K., Plexousakis, D.: Event pattern discovery for cross-layer adaptation of multi-cloud applications. In: Villari, M., Zimmermann, W., Lau, K.-K. (eds.) ESOC 2014. LNCS, vol. 8745, pp. 138–147. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-662-44879-3\\_10](https://doi.org/10.1007/978-3-662-44879-3_10)
21. Zeginis, C., Kritikos, K., Plexousakis, D.: Event pattern discovery in multi-cloud service-based applications. *IJSSOE* **5**(4), 78–103 (2015)