Towards an Architecture Integrating Complex Event Processing and Temporal Graphs for Service Monitoring

Juan Marcelo Parra-Ullauri
Aston University
Birmingham, United Kingdom
j.parra-ullauri@aston.ac.uk

Antonio García-Domínguez
Aston University
Birmingham, United Kingdom
a.garcia-dominguez@aston.ac.uk

Juan Boubeta-Puig
University of Cádiz
Cádiz, Spain
juan.boubeta@uca.es

Nelly Bencomo
Aston University
Birmingham, United Kingdom
n.bencomo@aston.ac.uk

Guadalupe Ortiz
University of Cádiz
Cádiz, Spain
guadalupe.ortiz@uca.es

ABSTRACT

Software is becoming more complex as it needs to deal with an increasing number of aspects in volatile environments. This complexity may cause behaviors that violate the imposed constraints. A goal of runtime service monitoring is to determine whether the service behaves as intended to potentially allow the correction of the behavior. It may be set up in advance the infrastructure to allow the detections of suspicious situations. However, there may also be unexpected situations to look for as they only become evident during data stream monitoring at runtime produced by te system. The access to historic data may be key to detect relevant situations in the monitoring infrastructure. Available technologies used for monitoring offer different trade-offs, e.g. in cost and flexibility to store historic information. For instance, Temporal Graphs (TGs) can store the long-term history of an evolving system for future querying, at the expense of disk space and processing time. In contrast, Complex Event Processing (CEP) can quickly react to incoming situations efficiently, as long as the appropriate event patterns have been set up in advance. This paper presents an architecture that integrates CEP and TGs for service monitoring through the data stream produced at runtime by a system. The pros and cons of the proposed architecture for extracting and treating the monitored data are analyzed. The approach is applied on the monitoring of Quality of Service (QoS) of a data-management network case study. It is demonstrated how the architecture provides rapid detection of issues, as well as the ability to access to historical data about the state of the system to allow for a comprehensive monitoring solution.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

KEYWORDS

Service monitoring, data stream monitoring, quality of service, temporal graphs, complex event processing

ACM Reference Format:
Juan Marcelo Parra-Ullauri, Antonio García-Domínguez, Juan Boubeta-Puig, Nelly Bencomo, and Guadalupe Ortiz. 2021. Towards an Architecture Integrating Complex Event Processing and Temporal Graphs for Service Monitoring . In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3412841.3441923

1 INTRODUCTION

Software complexity is higher than ever: systems are highly distributed, concurrent, and with many deeply intertwined physical and software components. This complexity and the ubiquitous nature of software systems may cause unforeseen behaviors that could violate the imposed constraints and may only emerge during runtime. Engineers and maintenance personnel must always keep track of a system's behavior during operation and check the interactions that occur between its components, as well as between the system and its environment. This is commonly referred to as runtime monitoring [30]. A goal of runtime service monitoring is to determine whether the service behaves as intended [8] to potentially allow the correction of the behavior if needed. However, service monitoring comes at a cost of the impact on performance of the monitored system. Therefore, it is important to deploy the right service of monitoring at an appropriate time and location to achieve its objectives whilst minimizing negative impact [17] as well as their complexity. As Rabiser et al. describe in [30], diverse runtime monitoring approaches have been developed in various domains and for different purposes. Complex Event Processing (CEP) stands up as a monitoring approach for timely processing and correlation of large amounts of data from multiple data sources [35]. CEP can be used for extracting valuable information from distributed message-based systems. With CEP, users of a system can specify the information that is of interest to them for posterior treatment [23]. Meaningful situations (called complex events) can be specified by performing stepwise correlation over streams of events [35]. However, CEP does not keep track of past events, what means that by the time the operators realize they need to get access to other historical data, it will be too late if the patterns were not previously defined.

In [12, 14] Garcia-Dominguez et al. proposed Temporal Graph (TG) databases as a useful representation for the past behavior of a system. Temporal graphs go beyond traditional event logs and time series in their ability to track the appearance and disappearance of entities and connections, and the capability to relate the changes in several properties of an entity over time. Garcia-Dominguez et al. introduced a temporal query language to support interactive diagnosis, offering both forensic analysis and online monitoring. These new capabilities imply some data storage and processing costs, but these can be managed in various ways. To save space, TGs can reuse unchanged nodes and use compression [16]. To save time when querying long histories, the timeline can record when specific events of interest happened [12], and queries can directly look them up.

A complete monitoring solution would ideally have the rapid response to events provided by CEP technology, and the ability to flexibly access relevant linked historical data from TGs, while maintaining the decoupling of different aspects and components of the monitoring infrastructure. In this paper, a novel approach towards an architecture for runtime data stream monitoring that integrates CEP and TGs is presented. The capabilities of the proposed architecture for extracting and handling monitored data are analyzed, as well as the costs involved. The approach is applied to monitor the Quality of Service (QoS) of a network management case study.

The rest of the paper is structured as follows. Section 2 presents the foundations that underlie the research, in terms of runtime service monitoring through the monitoring of their data streams, event-driven service-oriented architecture, CEP, and TGs. Section 3 illustrates the approach to enable comprehensive runtime service monitoring. Section 4 describes the case study in which the proposal is applied. Section 5 compares this work with other approaches. Section 6 concludes and outlines future work.

2 BASELINE

2.1 Runtime Service Monitoring

The behavior of complex software systems usually fully emerges during execution. Therefore, data streams need to be monitored at runtime. Runtime monitoring is executed in tandem with the system with the purpose of verifying the system's execution [4]. Different monitoring techniques and infrastructures exist [4, 8, 30]. Runtime Monitoring involves both observing the internal operations of a software system and the interactions with external entities, with the aim of detecting whether the system adheres or violates its requirements specification. As such, the monitor must be able to be notified about relevant events occurring in the executing system, to therefore support informed decision-making for the correction of the behaviour based on the data it has collected.

In order to mitigate or fix unwanted behaviours, runtime monitors should ideally be capable of making early detection of threatening situations as it can be exploited by self-healing and runtime adaptation [7]. However, a main disadvantage of runtime monitoring is the added overhead. Efforts have been made to minimize this overhead by offering more efficient techniques [5, 34]. Further, flexibility can also be introduced. For example, techniques based on TGs [29] can store the long-term history of a running system to support querying, at the expense of disk space and processing time. Another technique that supports monitoring is CEP [22], which can quickly react to incoming situations efficiently, if the appropriate event patterns have been previously identified. These techniques are further described in the rest of this section.

2.2 Service-Oriented Architecture and Complex Event Processing

Service-Oriented Architecture (SOA) is a paradigm focused on the design and implementation of loosely coupled distributed systems [28]. The fact that these architectures easily integrate third-party systems in a flexible and loosely coupled way, permits the developer to focus on the required business process themselves rather than on the implementing technologies. Although there are other alternatives, the communications traditionally used in SOAs are synchronous, using the request/response protocol. This limitation led to the evolution of the SOA into an alternative format called Event-Driven SOA (a.k.a SOA 2.0) [2].

In a SOA 2.0, communication between users, applications and services are expected to be led by events ocurring, rather than by using remote procedure calls [22]. In such a scope it is necessary to facilitate (i) the integration of several diverse heterogeneous data sources, which provide events, and (ii) a mechanism to process such events effectively and efficiently. In order to manage the events reaching the system, message brokers are used. Message brokers implement asynchronous communications so that source and target devices remain completely decoupled. The brokers may use standard message queues or be combined with a publish/subscribe mechanism, where messages are published according to a set of topics and users subscribe to the topics of their interest according to the need of the system in question. An example of a broker with the pub/sub connectivity protocol is the Eclipse open source message broker, Mosquitto [21]. This broker, which is part of the solution presented in this paper, is widely used due to its implementation of Message Queuing Telemetry Transport (MQTT) for lightweight messaging.

Concerning the need of a mechanism to process events, and the need to process big amounts of data in terms of events reaching the message broker, *data stream processing* in real time is required. CEP meets this requirement [22], allowing capture, analysis and correlation of large amounts of data as events in real time with the goal of detecting situations of interest [18]. These situations of interest are specified as event patterns, which use a set of defined conditions. Such patterns are deployed in a CEP engine, for the events to be received and analyzed. Given a stream of events, and once a condition is met (i.e. a pattern is matched), the system will raise a complex event signaling that a situation of interest has been detected and the interested parties will be notified accordingly.

2.3 Temporal Graphs

Conceptually, each attribute to be monitored in a running system can be considered as a *time series*: a sequence of values along an axis [10]. A time series may be enough for tracking the number of visits to a website or the number of pending calls in a call center, but it is not powerful enough to relate those to changes in the website configuration or in the number of staff on call, respectively.

On the other hand, an event log can capture these staffing and configuration changes, but the events are usually disconnected from each other and additional processing is needed to set up a "big picture" for analysis.

In general, we want something that can track changes not only in specific metrics, but that can also relate metrics of the same or similar entities, and record the changes in the entities present in the system and their connections. Graph databases such as Neo4j [31] are designed specifically to represent complex networks of relationships: their data is structured into nodes connected by edges. Nodes and edges have a label (e.g. "sensor") and a set of key/value pairs (e.g. "lastReading"). Graph databases have been successfully used for representing transport networks, social networks and other similarly interconnected systems. However, they do not explicitly model the time dimension.

Different extensions to graph databases exist to introduce the time axis: these *temporal graph (TG) databases* record how nodes and edges appear, disappear and change their key/value pairs over time. Some of these proposals include Greycat [16] from Hartmann et al. and Chronograph [15] from Haeusler et al. In particular, Greycat is an open-source solution which reuses several existing database engines (e.g. the LevelDB key/value store) to implement a TG data model. Nodes and edges in Greycat have a lifespan: they are created at a certain timepoint, they may change in state over the various timepoints, and they may be "ended" at another timepoint. Greycat considers edges to be part of the state of their source and target nodes. It also uses a copy-on-write mechanism to store only the parts of a graph that changed at a certain timepoint, to therefore save disk space.

Different from CEP engines, which focus on rapid response to events and with low storage requirements, TGs are concerned with efficient storage of the system's evolution. The authors of Greycat have aimed their solutions at the Machine Learning (ML) space, with the idea to learn ML models from the collected information. In this paper, the focus is to use TGs to add the ability to access historical data when an unexpected situation happens, where a CEP engine with no memory of past events will be less useful. The costs involved in keeping this long-term memory up to date, and how its post-hoc detection would compare against CEP patterns will be analyzed.

In [9, 14] Garcia-Dominguez et al. presented a solution that integrated a system with an indexing framework for transparent forensic and for run-time analysis at expense of disk space and processing time. The solution took raw JSON logs of the decision process over time and shaped them into a sequence of trace models, which were turned into a Greycat TG and exposed through a time-aware query language. This framework is used as a base for the work proposed in this paper aiming to provide a rapid detection of issues.

3 PROPOSAL

This section presents the SOA 2.0-based architecture that integrates CEP technology and temporal graphs to support runtime data stream monitoring. As shown in Fig. 1, there are three layers: *producers*, *middleware* and *consumers*.

Producers correspond with the data sources to be monitored. The consumers correspond with entities interested in the values being monitored. The *middleware* allows the communication between *producers* and *consumers*. An advantage of the proposed architecture is the decoupling of the processes that send and receive information. Senders do not need to know who receives what data, or how that happens. Likewise, receivers subscribe to updates without contacting the source.

3.1 Producer layer

The *producer layer* is composed of one or more producers of data. A data producer can be any type of system, user interface, service or device that generates data to be processed. The producers collect and expose the relevant data to be monitored, which is sent to the middleware layer through a message broker.

Before the monitoring infrastructure is deployed, it needs to be tested to check its ability to detect the situations of interest. This is specially relevant for CEP-based systems. When testing, instead of having a real systems acting as producers of events to be monitored, *simulators* can be used (see Figure 1).

3.2 Consumer layer

The *consumer layer* provides data consumers, giving domain experts the tools to achieve the monitoring objectives. These tools could be in the form of consoles and dashboards to allow further data analysis. The monitored data can be used for different purposes. For example, to be stored in a database for future analysis, or to be used as a data producer for pipelining processes.

To conduct local analysis, the architecture provides two specific monitoring components: a *QoS monitoring console* and a *QoS monitoring dashboard*. Both components have the ability to subscribe to the *QoS events* (i.e. query results and/or complex events) which can then be evaluated by domain experts. The QoS monitoring console allows end users to monitor such events in a simple textual way. Further, the QoS monitoring dashboard can make the real-time monitoring more user-friendly by providing domain experts with access to a graphical interface.

3.3 Middleware layer

The *middleware layer*, the core of the architecture, is the decoupling communication channel between producers and consumers. It allows the transformation, processing, analysis and routing of data between both ends of the architecture. This layer integrates TGs and CEP engines to enhance runtime data stream monitoring. The middleware layer is composed of the following five components.

The *input message broker* is responsible for receiving raw data from producers, and the *parser component* transforms the received raw data into the data formats required for both the TG and CEP engine, i.e. object graphs and simple events, respectively.

The *TG* is updated from the object graphs produced by the parser component, to create snapshots at the current points in time. The TG answers queries about historical data (either periodically or on demand), sending the results to the output message broker component. The TG can be updated by directly handling its contents, or by using a *model indexer* such as Hawk [14]. A model indexer compares the object graph against the current version of the TG, to

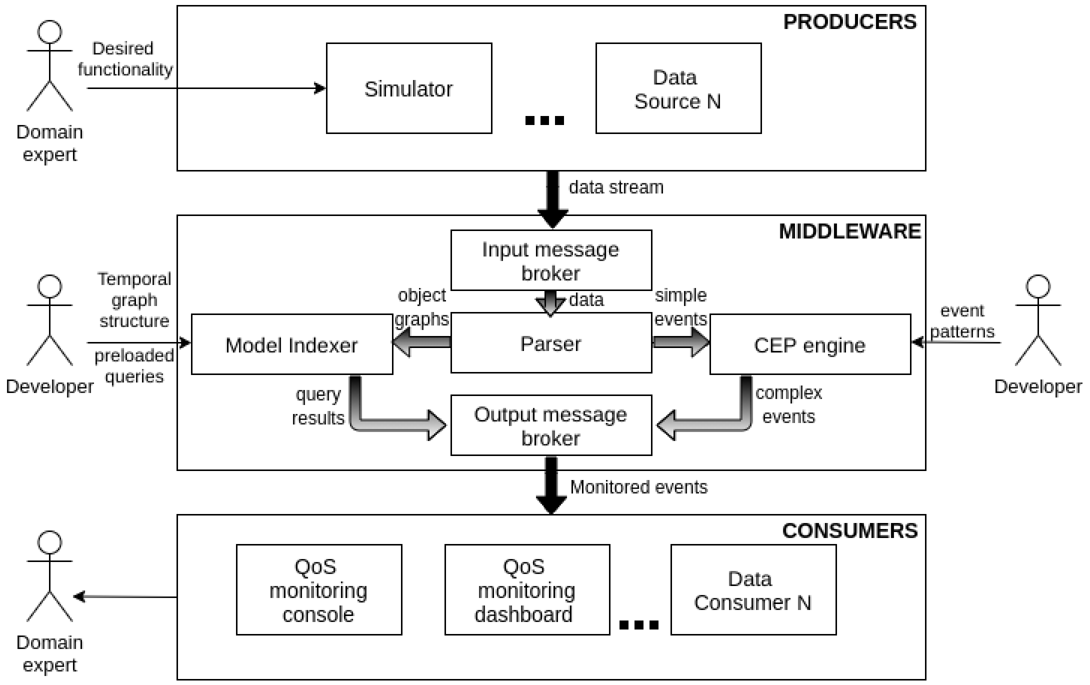


Figure 1: Our architecture proposal for integrating CEP and TG.

create a new version by applying the detected changes. The indexer reduces the effort involved when integrating full-history TGs to the creation of a single-snapshot object graph (i.e. the "model" to be "indexed" into the temporal graph).

The *CEP engine*, as explained above, processes and correlates the simple events received from the parser component with the aim of detecting, in real time, the situations of interest for the application domain. These situations of interest (known as event patterns) are implemented and provided to the engine by developers. When patterns are detected, the engine automatically generates complex events that summarize the happened situations, to therefore send the events to the output message broker component. A popular example of a CEP engine is *Esper* [30], which is used in the implementation offered here.

The *output message broker* component, previously mentioned, receives both the model indexer query results and the complex events generated by the CEP engine into dedicated topics to which consumers can subscribe.

It is important to note than in this architecture, the same events are received by both the CEP engine and the temporal graph. Essentially, CEP provides rapid response to known events, while the temporal graph creates a longer-term memory. The TG has a more explicit representation of the structure of the system, with evolving links between entities, fewer restrictions on time windows (which need to be explicitly sized in CEP), and the ability to jump back and forth in time. The TG will usually store information for both expected and unexpected queries, but the parser is still able to filter the information before it goes into the TG. Alternatively, the CEP engine itself could be used as a filter in front of the TG: this is planned for future work.

Timeslice	Action	Monitored NFR	Satisfaction	SLA
0	MST	MC	0.91	0.90
1	RT	MC	0.80	0.90
2	RT	MC	0.81	0.90
3	RT	MC	0.84	0.90
4	RT	MC	0.92	0.90

Table 1: Long Term Effects example

4 CASE STUDY

4.1 Remote Data Mirroring (RDM)

The Remote Data Mirroring (RDM) [20] is a self-adaptive system (SAS) that has been used as case study in this paper. RDM is a data protection technique that provides data availability and avoid data loss by replicating (mirroring) data across servers. A main benefit is fast disaster recovery. For the purposes of this paper, there are considered two sorts of mirroring approaches based on the topologies: Minimum Spanning Tree (MST), and Redundant Topology (RT). Both topologies provide their own levels of reliability, performance and cost, which are taken into account in the trade-off to estimate the levels of satisfaction of the non-functional requirements (NFRs) of the application: Maximization of Reliability (MR), Maximization of Performance (MP) and Minimization of Energy Consumption (MC). The NFR of a system are also known as the quality properties of the system. A RDM simulator [13] has been used as data producer.

Uncertainty exists due to different random situations such as delayed or lost messages, noise in sensors, an network link failures. Thus, there is the need of runtime monitoring to check that the RDM adheres to its initial requirements. The trade-offs required during the decision making of the RDM are specified according to the information of Figure 2.

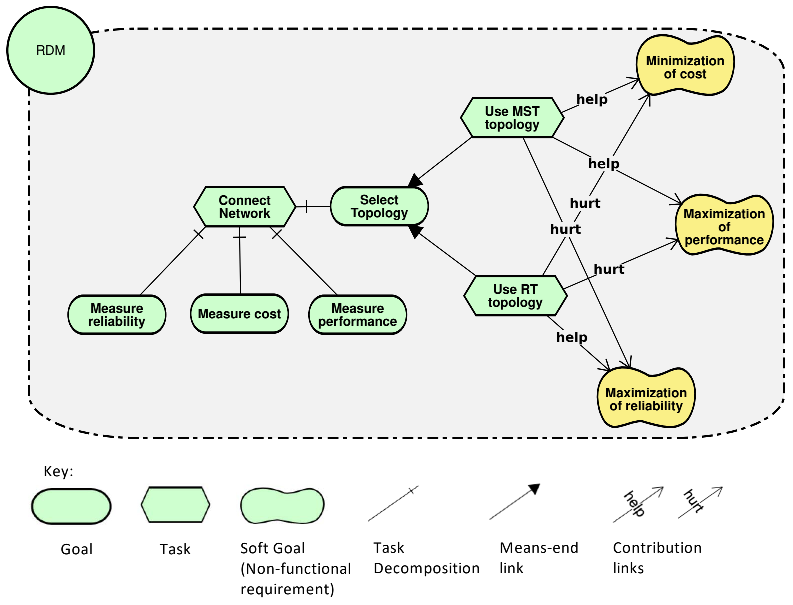


Figure 2: Case study RDM Self-Adaptive System RDM-SAS

An interesting aspect to be monitored is the effect of the proactive adaptation exposed by the RDM system [13]. An example is when an apparent bad decision turns out to be a good one in the long run. In that case, a decision may be, surprisingly and even unexpectedly, considered non-ideal at first when the level of satisfaction of an NFR is below the threshold defined in the Service Level Agreements (SLAs) and the action suggested by the system drives to a reduction on the level of satisfaction after the decision made. Nonetheless, as the system continues under the same action the level of satisfaction gradually increases until reaching or even exceeds its threshold. These kind of events are called *Long Term Effects* (LTEs) in the present work. Let's consider the simplified information of timeslices (ts) shown in Table 1: the NFRs are monitored over time slices. The monitored NFR is Minimization of Cost (MC).From ts 0 to 1, RDM decided to change (adapt) the topology from MST to RT. It ended in a reduction of the level of satisfaction of MC that drops below the threshold of the defined SLA, i.e. $0.8 < 0.9$. At first this can be considered a "bad decision". However, for the following ts 2 to 4, the system kept the same topology (decided) while the level of satisfaction of the NFR reached and even exceeded its SLA, i.e. $0.92 > 0.9$ shown in the last row. To test the proposed approach, data streams produced by RDM were monitored looking for LTEs situations. The aim of experiment was to successfully detect these LTEs and keep the stakeholders informed of the otherwise surprising behaviour. For this purpose, event patterns were deployed in the CEP engine that will trigger a complex event if a LTE situation is found. To validate the results, a query to the temporal graph, using the temporal query language of [12], was implemented.

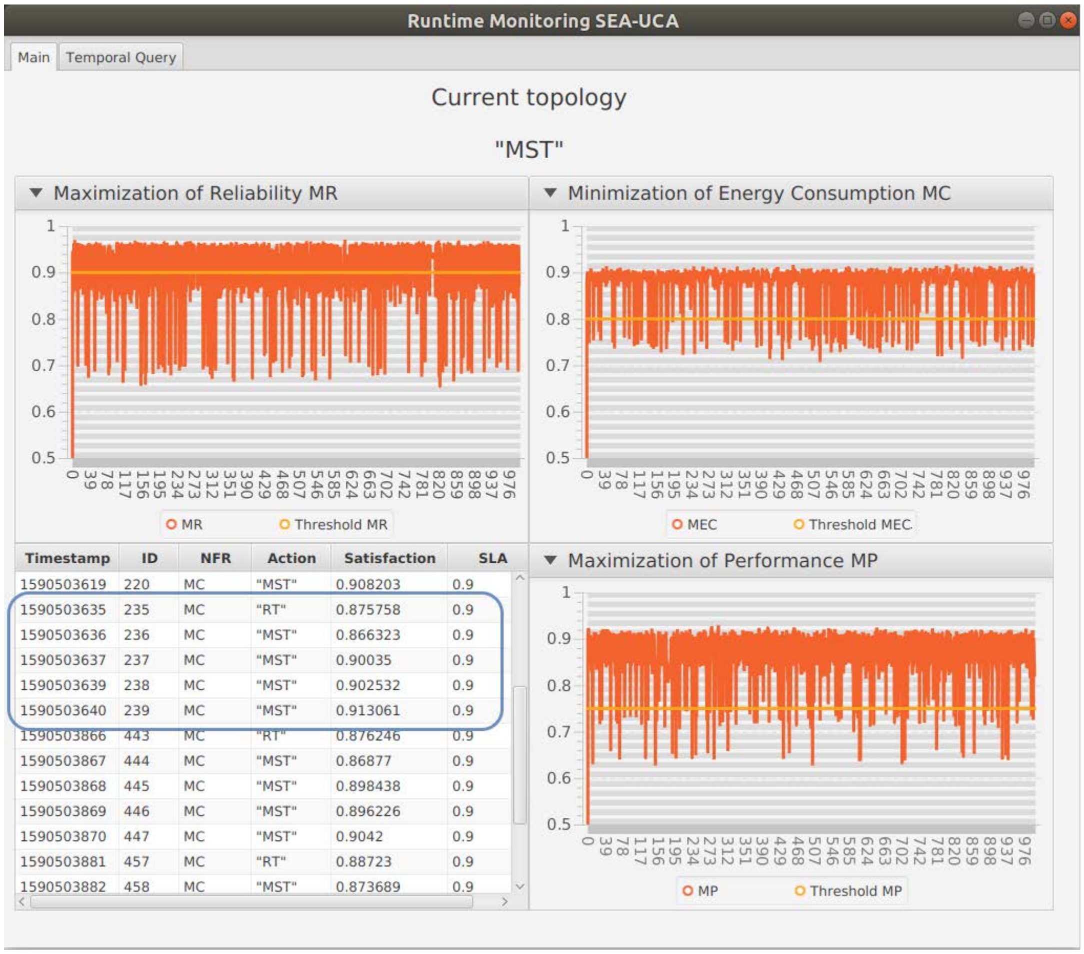
4.2 RDM-SAS Runtime Monitoring: Implementation

The implementation of the components of the architecture (Fig. 1) are described next.

Producer layer: The RDM system was extended to produce an execution trace of its behavior using timeslices in a data stream with JSON format. Each trace contains the latest observations, the current estimated levels of satisfaction of the NFRs, and the preferences applied in the decision process. An MQTT client based on the Eclipse Paho MQTT library for C++ was added to RDM. The client submits the raw JSON to a local Mosquitto broker so it can be processed by the middleware and consumer layers.

Consumer layer: As specified in Section 3, the architecture provides two monitoring components addressing the end user's needs. The *QoS monitoring console*, which is an MQTT Mosquitto client which subscribes to and presents the publications from the CEP engine component and the TGs component in a textual and simple way. The *QoS monitoring dashboard* was developed using JavaFX. It subscribes via Mosquitto to the same topics as the console, showing graphical interactive components such as tables, charts and controls (Fig. 3).

Middleware layer: The model indexer *Hawk* and the CEP engine *Esper* were selected as underlying technologies for processing. As mentioned in Section 3, the middleware has five components:


Runtime Monitoring SEA-UCA
Main
Temporal Query
Current topology
"MST"
Maximization of Reliability MR
Minimization of Energy Consumption MC
Maximization of Performance MP
MR
Threshold MR
MEC
Threshold MEC
MP
Threshold MP

Timestamp	ID	NFR	Action	Satisfaction	SLA
1590503619	220	MC	"MST"	0.908203	0.9
1590503635	235	MC	"RT"	0.875758	0.9
1590503636	236	MC	"MST"	0.866323	0.9
1590503637	237	MC	"MST"	0.90035	0.9
1590503639	238	MC	"MST"	0.902532	0.9
1590503640	239	MC	"MST"	0.913061	0.9
1590503866	443	MC	"RT"	0.876246	0.9
1590503867	444	MC	"MST"	0.86877	0.9
1590503868	445	MC	"MST"	0.898438	0.9
1590503869	446	MC	"MST"	0.896226	0.9
1590503870	447	MC	"MST"	0.9042	0.9
1590503881	457	MC	"RT"	0.88723	0.9
1590503882	458	MC	"MST"	0.873689	0.9

Figure 3: Graphical monitoring dashboard

- **Input message broker component:** Receives the raw data from the producer via its subscription to a specific topic of a Mosquitto broker.
- **Parser component:** Both Hawk and Esper can accept different input formats: in this experiment JSON have been chosen, as it is widely used. In the case of *Esper*, the parsing is called *creating the schema* and is done by declaring parameters and the input formats. The data of each timeslice is sent to the CEP engine as a *simple event*. For TGs, the JSON trace is reshaped into an object graph storing causal links among decisions, observations and measurements. This object graph is then given to the model indexer Hawk.
- **Temporal graph component:** Hawk indexes the object graphs and provides a query language for the temporal graph. The query for finding the LTEs is presented in [9]. Algorithm 1 describes the logic followed in the query, which was implemented in the query language supported by the Hawk. The query results are then sent to the output message broker.
- **CEP engine component:** Esper event patterns were defined to detect LTEs. Due to the complexity of the event to be detected, a *complex event hierarchies* approach with three patterns was needed. The *candidate pattern* detects potential "bad decisions", i.e. situations where decisions had a temporary negative short-term impact such as reduction of the level of satisfaction of the NFRs. The *interval pattern* checks for a potential bad decision if it produces an improvement in the following timeslices. The system must

Algorithm 1 Query to detect proactive adaptation: the long term effects of immediate actions. L is the current runtime log, T the set of timeslices in L, $S^{\mathrm{NFR}}(t)$ the level of satisfaction of the NFR at timeslice t, and α^{NFR} the threshold for the NFR.

```
1: Result = $\{L\}$
2: $T_B = \{t \in T | S^{\mathrm{NFR}}(t) < \alpha^{\mathrm{NFR}}\}$
3: for each $t_b \in T_B$ do
4:     if $S^{\mathrm{NFR}}(t_b + 1) < S^{\mathrm{NFR}}(t_b) \wedge$
         $\exists n \in \mathbb{N}_{>0}, \forall_j \in [1, n] \mid$
         $S^{\mathrm{NFR}}(t_b + j + 1) > S^{\mathrm{NFR}}(t_b)$ then
5:         Add $(t_b, n)$ to Result
6:     end if
7: end for
8: Result: Sequences showing proactive adaptation.
```

Listing 1: EPL pattern to collect the information of *LTEs* cases on RDM log using complex event hierarchies

```
@public @buseventtype @Name("InfoCollectionPattern")
insert into InfoCollectionPattern
select  a2.takeWhile(i => i.id<=a3.idEndInterval) as intervalInfo,
        a1.candidateInfo as candidateInfo
from pattern [every a1= CandidatePattern
−> a2= RDMLog until a3=IntervalPattern]
where a1.id= a3.idCandidate
```

keep performing the same action (e.g. RT) to be considered. If any of these conditions is not met, the candidate is discarded. The final *info collection pattern* aggregates information from the detected intervals, revealing the LTE. Listing 1 shows the InfoCollectionPattern, as implemented in the Esper Event Processing Language (EPL). The pattern accumulates the information of the RDMLog in the time window starting from every CandidatePattern until the upper limit defined by the IntervalPattern.

- **Output message broker component:** As in the case of the input message component, the same MQTT Mosquitto broker is used. The TG query results and the complex events are published into specific topics in the broker. The consumers subscribe to these topics in the broker and the monitored data is presented to the user.

4.3 Runtime Monitoring of RDM: Evaluation

The technical setup of the evaluation of the architecture is described as follows. The data producer ran a 1000-timeslice simulation of RDM with 1 second between timeslices. The middleware processes the incoming data stream using Hawk and Esper in the background. Finally, the monitoring results were displayed at the consumer layer through the GUI. A Lenovo Thinkpad T480 with an Intel i7-8550U CPU with 1.80GHz, running Ubuntu 18.04.2 LTS and Oracle Java 1.8.0_201 was used. The implemented tools and libraries were: the RDM SAS simulator, Paho MQTT 1.2.2, Eclipse Hawk 2.0.0, and Esper 8.0.0. The architecture evaluation was done based on the following four criteria:

Pattern validation: The highlighted section in Figure 3 shows an excerpt of the examples found by the pattern. Among the 1000-ts simulation, 8 LTEs intervals were detected. One interval started at timeslice 236, when RDM decided to use the MST topology instead of RT and a reduction on the satisfaction level of MC, the monitored NFR, is observed: from 0.875758 (timeslice 235) to 0.866323 (timeslice 236). Thus, the timeslice 236 represents a candidate complex event. However, the satisfaction level grew up over the following timeslices, until it exceeded the threshold (0.9) in timeslice 239. Different situations can be observed such as the timeslice 443. This shows the architecture was capable of finding the proactive behavior at the monitored NFR.

Storage impact: There is a storage impact due to the historical data management. For the 1000-timeslice simulation of RDM, the temporal graph database grew up to 14MB, what can be considered acceptable for this implementation.

Overhead impact: As the data producer only interacts with the architecture by sending the logs to the middleware, the only overhead to the system is added by the communication between RDM and the MQTT broker. It takes 101ms to connect to the MQTT broker, and on average it took 281μs to publish a message. It added an overhead to the simulation done of only 0.00128%. Regarding the core components, it took 705ms for the CEP engine to report the first LTE after its 5 timeslices happened. Building the temporal graph took 11705ms over the whole history of the system, with an average of 11.71ms per timeslice. Querying the temporal graph in the server took 512ms.

Flexibility: CEP allows the implementation or extension of new patterns at runtime, but these new introduced patterns would only process the upcoming events since their implementation. The proposed architecture provides, through the use of temporal graphs, the flexibility for querying the system on demand without losing information. A simple example of this could be the monitoring of a different NFR i.e. Maximization of Reliability (MR). The query would follow the same structure of Algorithm 1, with the NFR corresponding to MR. For finding these situations using only CEP, a new simulation should be run and the pattern should be updated accordingly.

In general, a main advantage of the presented approach is that it provides to end users an infrastructure capable to react during execution to defined patterns, and at the same time allowing them to access historical data to analyze the system behavior over time. With the evaluation, it has been demonstrated the feasibility to meet the initial purposes of the approach, and analyze the costs involved in its implementation.

5 RELATED WORK

5.1 Event-driven approaches for runtime monitoring

Next, some approaches based on event processing for runtime monitoring will be described. Event Sourcing [11] provides traceability for the changes over the time of application state as a sequence of events. This data storage model keeps track of every data change without removing earlier events [33]. The stored event streams can be queried on demand for monitoring [27] or retroactive computing purposes [26]. A problem with event sourcing is that keeping track of not only the current state, but also every change leading up to that state could be costly on terms of performance [27]. On the contrary, the present paper describes an event-driven approach with the use of CEP for efficient event streams monitoring and deal with scalability problems through the use of temporal graphs for historical data. Ashraf et al. [1] propose a collision prediction approach for Unmanned Aerial Vehicles (UAVs) using CEP to monitor UAVs in real time and predict runtime collisions efficiently. In addition to predicting collisions, the system proactively looks for the best ways to avoid planned collisions to ensure the safety of the entire swarm. Inçki et al. [19] propose a non-intrusive solution for runtime verification of IoT systems, assuming that such systems are designed according to SOA principles. In particular, they propose a CEP-based solution where each CoAP message is intercepted and injected into the CEP engine as a simple event. The patterns which permit the detection of failures in the system at runtime are then deployed in the CEP engine. Indeed, CEP is used in both proposals for runtime monitoring, which is the most frequent case where CEP is used. However, it is not used to monitor the Qos properties (NFRs) of the system. It is neither used to access historical data to analyze the system's behavior over time for better proactive reaction, which is one of this paper main goals.

5.2 CEP for service monitoring

Romano et al. [32] propose a QoS monitoring approach for cloud computing platforms that makes use of CEP and Content Based Routing (CBR) to detect contract violations. In their case study, they implement remote monitoring of power consumption in a smart grid environment. However, the proposed approach goes one step further using TGs to store long-term history to therefore, provide more accurate predictions and better understanding about monitored variables and detected situations of interest to facilitate proactive adaptations. Moser et al. [24] use CEP to build a flexible monitoring system to support temporal and causal dependencies between messages for service composition infrastructures in the context of WS-BPEL. They provide monitoring data that might be relevant for composite service monitoring still they only monitor SOAP messages in the context of BPEL processes, and do not monitor NFRs of the system. Furthermore, Cicotti et al. [6] present QoSMINaaS, a QoS monitoring facility for a new Cloud-based Platform-as-a-Service based on cloud computing and CEP. QoSMONaaS focuses on the performance delivered at the business process level. First, the SLAs are analyzed to collect Key Performance Indicators (KPIs) and a set of CEP rules are set. QoSMINaaS attempts to prevent the violation condition by detecting when the KPIs exceeds the set of thresholds, by using statistical and logical inference. The present work, however, does not focus on the business process level, instead it focuses on the level of performance of internal variables of the system, in order to improve its performance.

5.3 Graphs for runtime monitoring

Mouline et al. propose the use of Greycat TGs to allow systems to consider both measured and expected values when making adaptation decisions in the context of MAPE-K [25]. They applied this approach to a elastic-cloud manager simulation, reducing the number of container start/stop actions by considering future expected capacities and workloads. This approach was a single application, rather than a distributed architecture in this paper, and it did not integrate the rapid response to events offered by CEP. Haeusler et al. [15] present ChronoSphere (CS), an alternative versioned graph technology, applying it to an IT landscape case study. Their system read data from an existing Configuration Management Databases (CMDBs), and created a versioned graph that can be accessed over time. Instead of the single element history common in CMDBs, the versioned and branching graphs in CS allowed them to see the entire IT landscape of the organisation at points in time, to create what-if plans. Their case study required building the system around CS from the ground up, whereas Hawk can be adopted for monitoring without any required modification. Finally, Búr et al. [3] shows an approach that maintains a *distributed runtime model* of the state of a system, and presented a distributed graph query evaluation algorithm for monitoring. Unlike the proposed approach, which stores the temporal graph in a central location, this approach has only an in-memory graph spread across nodes. This can scale out as the nodes increase, yet the queries can require high amounts of bandwidth, which can be hindered by latency.

6 CONCLUDING REMARKS AND FUTURE WORK

This paper has presented an architecture for comprehensive data streams runtime monitoring using the capabilities of both CEP and TGs. While the architecture allows for the quick reaction to events supported by CEP, it also exploits the capability of TGs to analyze the historic behavior of the system. Based on the case study, it has been demonstrated how the architecture can detect situations where the system makes decisions with effects that may not be appreciated in the short-term, e.g. a decision that has short-term negative impact but with an overall positive long-term effect. This kind of behavior is result of proactive adaptation shown by the RDM decision-making algorithm (which bases its projections to the future using Bayesian learning). The ability of detecting this kind of behavior requires techniques that can use runtime data and/or historic data to reason about the evolution over time, as our architecture allows using CEP and TGs.

There are several avenues for future work. In terms of performance and based on the evaluation, one potential constraint is the impact on storage. If a producer generates huge amounts of data, the size of the temporal graph and the query times would inevitably increase. One possible solution for this issue is the use of sampling, i.e. the update of the TG at a defined rate. Another alternative solution could be using CEP to filter the data that builds the TG. Another possibility could be to capture CEP event hierarchies into a TG, where each complex event is a node, and events are linked if they are interdependent.

Beyond these lines of work, a priority in the research road-map is to provide feedback from consumers to producers in order to improve the decision-making of the monitored system[13]. This will require further real-world case studies in other domain problems, using new producers, consumers, and also other situations of interest.

ACKNOWLEDGMENTS

This work has been partially sponsored by The Lerverhulme Trust Grant No. RF-2019-548/9, the EPSRC Research Project Grant No. EP/T017627/1 and by the Spanish Ministry of Science and Innovation and the European Regional Development Fund under Project FAME [RTI2018-093608-B-C33].

REFERENCES

[1] Adnan Ashraf, Amin Majd, and Elena Troubitsyna. 2020. Online Path Generation and Navigation for Swarms of UAVs. *Scientific Programming* 2020 (2020).
[2] Juan Boubeta-Puig, Guadalupe Ortiz, and Inmaculada Medina-Bulo. 2015. MEdit4CEP: A model-driven solution for real-time decision making in SOA 2.0. *Knowledge-Based Systems* 89 (2015).
[3] Márton Búr, Gábor Szilágyi, András Vörös, and Dániel Varró. 2020. Distributed graph queries over models@run.time for runtime monitoring of cyber-physical systems. *Int. J. on Softw. Tools for Technology Transfer* 22 (2020).
[4] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. 2017. A Survey of Runtime Monitoring Instrumentation Techniques. *Electronic Proceedings in Theoretical Computer Science* 254 (2017).

[5] Ian Cassar, Adrian Francalanza, and Simon Said. 2015. Improving Runtime Overheads for detectEr. *Electronic Notes in Theoretical Computer Science* (2015).

[6] Giuseppe Cicotti, Luigi Coppolino, Rosario Cristaldi, et al. 2011. QoS Monitoring in a Cloud Services Environment: The SRT-15 Approach. In *Euro-Par 2011: Parallel Processing Workshops (LNCS)*. Springer, Berlin, Heidelberg, 15–24.

[7] Rogério de Lemos, Holger Giese, Hausi A. Müller, et al. 2013. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*. Vol. 7475. Springer-Verlag.

[8] Nelly Delgado, Ann Q Gates, and Steve Roach. 2004. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on software Engineering* 30, 12 (2004), 859–872.

[9] Antonio Garcia Dominguez, Nelly Bencomo, Juan Marcelo Parra Ullauri, and Luis Hernan Garcia Paucar. 2019. Towards history-aware self-adaptation with explanation capabilities. In *Proceedings of FAS* W 2019*. IEEE.

[10] Philippe Esling and Carlos Agon. 2012. Time-series data mining. *ACM CSUR* 45, 1 (2012).

[11] Martin Fowler. 2005. Event sourcing. *Online, Dec* (2005), 18.

[12] Antonio García-Domínguez, Nelly Bencomo, Juan Marcelo Parra-Ullauri, and Luis Hernán García-Paucar. 2019. Querying and annotating model histories with time-aware patterns. In *MODELS*. IEEE.

[13] Luis Garcia-Paucar and Nelly Bencomo. 2019. Knowledge Base K Models to Support Trade-offs for Self-adaptation using Markov Processes. *13th Conference SASO* (2019).

[14] Antonio García-Domínguez, Nelly Bencomo, and Luis H Garcia Paucar. 2018. Reflecting on the past and the presentwith temporal graph-based models. In *Proceedings of MODELS 2018 Workshops*, Vol. 2245. CEUR-WS.org, Denmark.

[15] Martin Haeusler, Thomas Trojer, Johannes Kessler, et al. 2019. ChronoSphere: a graph-based EMF model repository for IT landscape models. *Software and Systems Modeling* (2019).

[16] Thomas Hartmann, Francois Fouquet, et al. 2017. Analyzing Complex Data in Motion at Scale with Temporal Graphs. In *Proceedings of SEKE'17*.

[17] Garth Heward, Ingo Müller, Jun Han, Jean-Guy Schneider, and Steven Versteeg. 2010. Assessing the performance impact of service monitoring. In *2010 21st Australian Software Engineering Conference*. IEEE, 192–201.

[18] Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. 2014. Generic event-based monitoring and adaptation methodology for heterogeneous distributed systems. *Software: Practice and Experience* 7 (2014).

[19] Koray Inçki, Ismail Arı, and Hasan Sözer. 2017. Runtime verification of IoT systems using Complex Event Processing. In *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC)*.

[20] Minwen Ji, Alistair C Veitch, John Wilkes, et al. 2003. Seneca: remote mirroring done write.. In *USENIX Annual Technical Conference, General Track*.

[21] Roger Light. 2017. Mosquitto: server and client implementation of the MQTT protocol. *Journal of Open Source Software* (2017).

[22] David C. Luckham. 2012. *Event processing for business: organizing the real-time enterprise.* John Wiley & Sons, Hoboken, N.J, USA. ISBN 978-0-470-53485-4.

[23] David C Luckham and Brian Frasca. 1998. Complex event processing in distributed systems. *Computer Systems Laboratory Technical Report. Stanford University, Stanford* (1998).

[24] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. 2010. Event Driven Monitoring for Service Composition Infrastructures. In *International Conference on Web Information Systems Engineering (LNCS)*. Springer, Berlin, Heidelberg, 38–51.

[25] Ludovic Mouline, Amine Benelallam, Thomas Hartmann, et al. 2018. Enabling temporal-aware contexts for adaptative distributed systems. In *Proceedings of SAC'18*. ACM Press, Pau, France.

[26] Michael Müller. 2016. Enabling retroactive computing through event sourcing. (2016).

[27] Michiel Overeem, Marten Spoor, and Slinger Jansen. 2017. The dark side of event sourcing: Managing data conversion. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 193–204.

[28] Michael Papazoglou and Jan Van Den Heuvel, Willem. 2006. Service-oriented design and development methodology. *Int. J. Web Eng. Technol.* 2, 4 (2006).

[29] Juan Marcelo Parra-Ullauri, Antonio García-Domínguez, Luis Hernán García-Paucar, and Nelly Bencomo. 2020. Temporal Models for History-Aware Explainability. In *Proceedings of the 12th System Analysis and Modelling Conference*. 155–164.

[30] Rick Rabiser, Sam Guinea, Michael Vierhauser, Luciano Baresi, and Paul Grünbacher. 2017. A comparison framework for runtime monitoring approaches. *Journal of Systems and Software* 125 (2017), 309–321.

[31] Ian Robinson, James Webber, and Emil Eifrem. 2015. *Graph databases* (second ed.). O'Reilly. ISBN 978-1-4919-3089-2.

[32] Luigi Romano, Danilo De Mari, Zbigniew Jerzak, and Christof Fetzer. 2011. A Novel Approach to QoS Monitoring in the Cloud. In *2011 First International Conference on Data Compression, Communications and Processing*. 45–51.

[33] Johan Rothsberg. 2015. *Evaluation of using NoSQL databases in an event sourcing system.* Master's thesis. Linköping University, Database and information techniques.

[34] K. Sen, A. Vardhan, G. Agha, and G. Rosu. 2004. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings. 26th International Conference on Software Engineering*.

[35] Marco Volz, Boris Koldehofe, and Kurt Rothermel. 2011. Supporting strong reliability for distributed complex event processing systems. In *2011 IEEE International Conference on High Performance Computing and Communications*. IEEE.