Artifact-Driven Process Monitoring at Scale

Meroni, Giovanni; Garda, Szabolcs

Published in:
Proceedings of 21st International Conference on Service-Oriented Computing

Link to article, DOI:
10.1007/978-3-031-48424-7_1

Publication date:
2023

Document Version
Peer reviewed version

Artifact-driven Process Monitoring at Scale

Giovanni Meroni[1][0000−0002−9551−1860] and Szabolcs Garda[1]

Technical University of Denmark, Kgs. Lyngby, Denmark
giom@dtu.dk

Abstract. Artifact-driven process monitoring is an effective technique to autonomously monitor business processes. Instead of requiring human operators to notify when an activity is executed, artifact-driven process monitoring infers this information from the conditions of physical or virtual objects taking part in a process. However, SMARTifact, the existing monitoring platform implementing this technique, has been designed to run entirely on edge devices, each of which can monitor only one execution of the process. Thus, monitoring multiple executions at the same time, or reducing the computational requirements of edge devices is not possible. In this paper, we introduce a new artifact-driven monitoring platform that overcomes these limitations and makes artifact-driven monitoring fully scalable.

Keywords: Process monitoring · Scalability · Fog computing.

1 Introduction

Business Process Management is the discipline devoted to oversee how organizations perform their work [4]. In particular, process monitoring focuses on gaining insights on how business processes - a set of activities to be performed to achieve a certain goal and the dependencies among them - are performed. This is particularly relevant for so-called multi-party business processes, which require multiple organizations to take part in the same process and to coordinate their activities. In this setting, being able to promptly identify any issue with respect to the planned behavior makes possible for the involved parties to quickly react and take countermeasures.

Among the existing process monitoring techniques, artifact-driven process monitoring [12] is one of the few that specifically targets multi-party business processes. By collecting and processing information coming from physical and virtual objects participating in a process, artifact-driven process monitoring can autonomously identify when activities are executed. Also, by relying on a declarative language named Extended-GSM (E-GSM) to represent the process to monitor, artifact-driven monitoring can immediately detect discrepancies between the planned process and the actual execution.

Despite the advantages brought by artifact-driven process monitoring, SMARTifact, the existing monitoring platform implementing this approach limits its applicability to specific cases. In particular, SMARTifact has been designed to be executed entirely on edge devices (that is, the physical objects in the process). Also, each device can only monitor one execution of the process. If two executions are running, they must be monitored by two distinct devices. Secondly, SMARTifact maintains the monitoring information on the device and in memory. Therefore, if the device experiences a malfunctioning or simply it runs out of battery, all monitoring information is lost.

In this paper we introduce a new monitoring platform, based on the fog computing paradigm, that aims at overcoming the limitations of SMARTifact. In particular, this platform is specifically designed with scalability in mind, making possible to monitor a virtually unlimited number of parallel process executions.

This paper is structured as follows. Section 2 introduces artifact-driven process monitoring and the architecture of SMARTifact. Section 3 discusses the limitations of SMARTifact and identifies a set of requirements that need to be addressed. Section 4 presents the architecture of our monitoring platform. Section 5 discusses how our platform has been validated. Section 6 surveys the state of the art for related work. Finally, section 7 draws the conclusions and outline possible future work.

2 Baseline

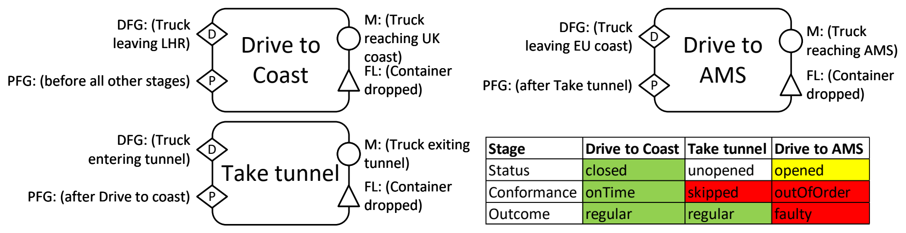
To make this paper self-contained, a brief discussion on how artifact-driven monitoring works, and on the architecture of the SMARTifact platform is provided in this section. The reader should refer to [12] for further details.

2.1 Artifact-driven Process Monitoring

Traditional process monitoring techniques assume that, whenever an activity in a process is executed, an event is always produced. This can be relatively easily achieved when activities are at least partially automated. However, when manual tasks - activities performed by humans without interacting with a computer - are present in the process, events must be manually sent by the operators responsible for the manual tasks and, as such, they are prone to be forgotten or delayed. In addition, most monitoring techniques can detect discrepancies between the expected execution and the actual one only after the execution is complete.

To address these limitations, artifact-driven process monitoring has been proposed. this approach assumes that, whenever an activity is executed, it alters the conditions of one or more physical or virtual objects, named artifacts, that take part in the process. Therefore, by monitoring the conditions of these artifacts during a process execution, it is possible to infer when activities are executed without relying on explicit events.

In order to detect and react to violations, artifact-driven process monitoring represents the process to monitor with a declarative language named E-GSM. In a nutshell, E-GSM represent activities and process portions as stages, which are assessed according to three perspectives: *status*, *compliance*, and *outcome*. The status of a stage can be *unopened*, *opened* or *closed*, indicating that the corresponding activity or process portion was never executed, is running, or is complete. The compliance can be *onTime*, *outOfOrder* or *skipped*, indicating that the activity or process portion follows the process model, has been executed when it should not, or has not been executed when it should. Finally, the outcome can be *regular* or *faulty*, indicating that the activity or process portion was correctly performed, or that something went



Stage	Drive to Coast	Take tunnel	Drive to AMS
Status	closed	unopened	opened
Conformance	onTime	skipped	outOfOrder
Outcome	regular	regular	faulty

Fig. 1. E-GSM model of LHR-AMS, and monitoring results of an incorrect execution.

wrong while it was running. When the process starts, all stages are unopened, onTime, and regular. Stages can be decorated with data flow guards, process flow guards, milestones and fault loggers. Data flow guards and milestones specify the conditions on the artifacts that cause the decorated stage to become, respectively, opened or closed, determining the status. Process flow guards specify control flow dependencies (i.e., which other stages should be executed before the decorated stage), determining the compliance. Fault loggers specify the conditions on the artifacts that cause the stage to become faulty, determining the outcome.

To better understand E-GSM, Figure 1 shows how it can be used for representing and monitoring the following process. A truck driver is expected to start the process in the LHR airport, and to drive to the coast. Once it reaches it, the driver has to take the Channel tunnel, and finally to drive to the AMS airport. If we consider a process execution where, instead of taking the Channel tunnel, the truck driver takes a ferry, stops before reaching the AMS airport and opens the container, an E-GSM engine will be able to detect activity drive to coast as *closed*, since it completed its execution, *onTime*, since it was the first activity to be executed, and *regular*, since the container was never dropped while the activity was running. The engine also will detect take tunnel as *unopened*, since it was never executed, *skipped*, since it was not executed after drive to the coast ended, and *regular*. Finally, the engine will detect drive to AMS as *opened*, since it is still running, *outOfOrder*, since it was executed before take tunnel, and *faulty*, since the container was dropped while the activity was running.

2.2 SMARTifact

Originally known as mArtifact [1], SMARTifact is one of the first artifact-driven process monitoring platforms in the literature. Specifically targeting processes involving physical artifacts, SMARTifact has been designed to run on edge devices (e.g., single-board computers) attached to these artifacts. Its architecture, as shown in Fig. 2, consists in the following components deployed on each edge device:

On-board Sensor Interface. This component is responsible for collecting data from the sensors installed on the edge device attached to a physical artifact.

Event Processor. This component is responsible for aggregating and processing sensor data, in order to determine when the conditions of the attached artifact change.

Events Router. This component is responsible for sending changes in the conditions of the attached artifact to the other edge devices taking part in the same process

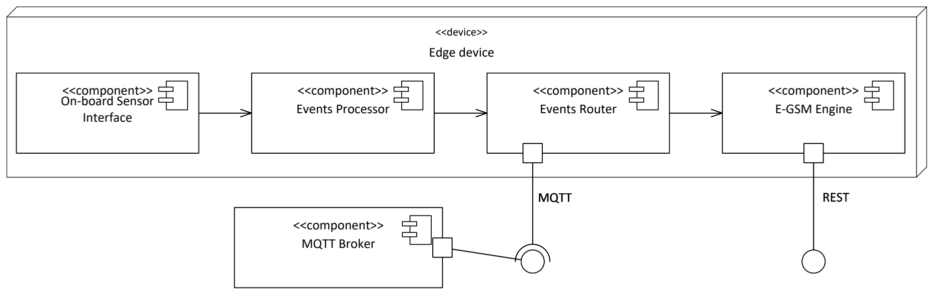

<<device>>
Edge device
<<component>> On-board Sensor Interface
<<component>> Events Processor
<<component>> Events Router
<<component>> E-GSM Engine
MQTT
REST
<<component>> MQTT Broker

Fig. 2. Architecture of the SMARTifact platform.

execution. The events router is also responsible for receiving from the other edge devices changes on the conditions of the other artifacts in the same process execution.

E-GSM Engine. This component contains the E-GSM model of the process to monitor. It is responsible for evaluating the data flow guards, process flow guards, milestones and fault loggers of all stages whenever a change in the conditions of one of the artifacts in the process is detected. It also exposes a Representational State Transfer (REST) Application Programming Interface (API) outside the edge device, which is used to configure the edge device (e.g., by providing the E-GSM model of the process to monitor) and to retrieve information on the process being monitored (e.g., the value of the status, compliance and outcome perspectives for each stage).

To communicate with each other, edge devices rely on an **Message Queue Telemetry Transport (MQTT) broker**. MQTT is a publish-subscribe protocol specifically designed for Internet of Things (IoT) applications. Being a message queue-based protocol, MQTT completely decouples the state of the sender with the recipient.

3 Application Requirements

Despite having proven to be effective in some scenarios, such as smart logistics, SMARTifact suffers from some limitations. Firstly, with the exception of the MQTT Broker, all components are meant to run on an edge device. Although single board computers capable of running SMARTifact are relatively inexpensive, their size and power consumption can be a limiting factor for some processes. For example, although SMARTifact can monitor the conditions of a shipping container and its content, it cannot individually monitor the conditions of each package in the container. To address this issue in our platform, we define the following application requirement. *AR1: Edge devices should only run components needed to process data they directly collect.*

Another limitation of SMARTifact is the inability, for an edge device, to monitor multiple executions of the same process at the same time. This limitation comes from the E-GSM Engine, which is capable of running only one instance of the process to monitor, and makes SMARTifact unsuited to monitor process executions that share the same artifacts. For example, suppose that a truck is shipping two containers that have to be delivered in two different places. Then, the E-GSM Engine running on the

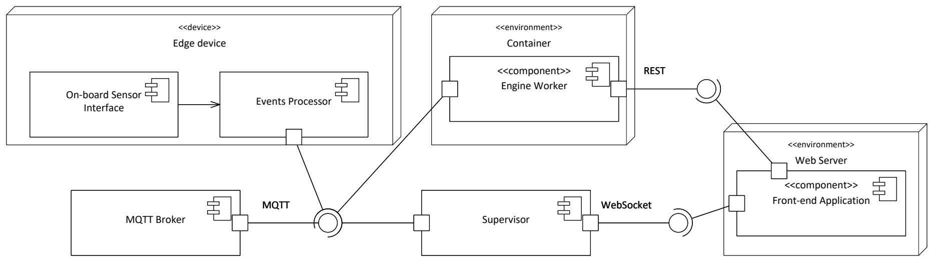


Fig. 3. Architecture of our platform.

truck will consider the two containers as participants of the same execution, rather than to two distinct executions. To address this issue in our platform, we define the following application requirement. *AR2: the platform should allow monitoring multiple instances of the same process at the same time.*

4 Proposed Solution

To address the requirements identified in Sec. 3, the architecture shown in Fig. 3 has been designed. This architecture reuses and, when needed, adapts the components present in SMARTifact. In addition, new components and interfaces are introduced.

To address AR1, this architecture embraces the fog computing paradigm [15]. Only the On-board Sensor Interface and the Events Processor are deployed on the edge devices, since they are responsible for processing data created on that device. To accommodate this change, the Events Processor no longer communicates changes in the condition of the artifact to Events Router directly. Instead, it publishes them to an MQTT topic. Therefore, it requires an MQTT interface to the MQTT Broker.

To address AR2, the E-GSM Engine and the Events Router are moved inside the **Engine Worker** component, which is deployed in a cloud environment. To achieve vertical scalability, a software wrapper has been built around the E-GSM Engine, making it multi-instance. Also, to achieve horizontal scalability, the Engine Worker is deployed inside a container, making it easy to deploy multiple instances of this component.

The **Supervisor** and **Front-end Application** components have also been introduced. These components are deployed in a cloud environment as well. The Supervisor keeps track of which Engine Worker instances are in charge of monitoring a specific process execution. It also instantiates, monitors, and destroys Engine Worker instances when needed. The Front-end Application is a web application that allows the user to interact with the monitoring platform. By communicating with the Supervisor through WebSocket, the Front-end Application can know which process executions are monitored and by which Engine Worker. By communicating with the Engine Worker through a REST API, the Front-end Application can pull monitoring information on-demand, and show them to the user. With the exception of the Front-end Application, all components communicate through the MQTT broker. This makes possible for the components to communicate with each other even if some of them change address, new instances are instantiated, or unneeded instances are destroyed.

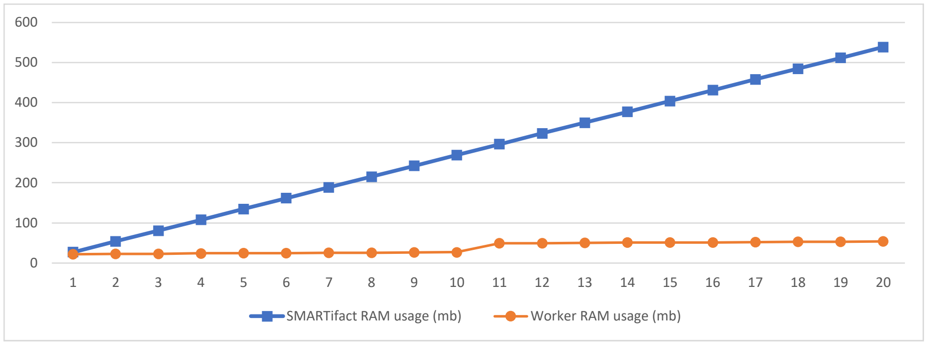


Fig. 4. Memory usage of our solution compared to SMARTifact.

5 Evaluation

To evaluate our solution, we implemented a prototype of the monitoring platform[1]. Since the E-GSM Engine and the Events Router were originally built in Node.js, and we planned to extend them rather than to rewrite them, the Engine Worker was implemented in Node.js. Also, since Node.js was proven to be resource efficient and easy to port across different environments, and it also provided native support for MQTT, we adopted this programming language also for the Supervisor. The Front-end Application was built in Angular, due to the availability of many data visualization libraries and the tight synergy with Node.js. Like in SMARTifact, we implemented the Events Aggregator with Node-red. Also, to simulate the On-board Sensor Interface, we adopted the simulator that was used by the authors of SMARTifact to test it, which generates sensor data from low-level logs. Finally, to deploy the components composing the architecture, we adopted Docker.

To assess how our solution performs in terms of scalability, we compared the memory usage of the Engine Worker components with the one of SMARTifact. To this aim, we initiated an instance of the Engine Worker with a maximum engine limit set to 10. We then proceeded to instantiate and monitor 20 instances of an extended version the LHR-AMS process - which was also used to validate the original version of SMARTifact [12] - 10 of which were compliant and 10 non compliant. As we reached a total of 10 running process instances, we initiated another Engine Worker and continued creating engines until we reached a total of 20 running engines. In order to draw a meaningful comparison with SMARTifact, we enclosed the Event Router and the E-GSM of that platform inside a container. We then deployed 20 instances of that container, and we measured the total memory usage and the time to start monitoring a new execution. The results of this comparison are presented in Figure 4.

As shown in this figure, the usage of Engine Worker components led to a substantial reduction in memory usage when compared to the individual deployment of SMARTifact instances. The disparity in memory utilization becomes more pronounced as the number of process executions increases. With SMARTifact, monitoring a new

[1] Source code available at https://github.com/eGSM-platform.

execution requires creating two instances of the Event Router and the E-GSM Engine. In contrast, the Engine Worker implementation only adds a negligible amount of data to its internal data structures. This is also the reason why our solution is faster at starting to monitor a new instance. Compared to SMARTifact, which requires on average 76 ms, our solution requires only 28 ms. When the maximum engine limit is reached and an additional Engine Worker is instantiated, memory utilization doubles. Nevertheless, even considering this spike, the memory usage of the Engine Worker remains significantly lower than SMARTifact. This trend is expected to persist as the number of engines continues to increase.

Finally, to verify that no side-effects in process monitoring were introduced in our platform, we compared the monitoring results with the ones obtained by SMARTifact and ensured they were identical.

6 Related Work

Several solutions for runtime business process monitoring exist in the literature. In [16] and [2], a Complex Event Processing (CEP) engine is adopted to determine if an execution deviates from the expected behavior. Similarly, [13] proposes an alternative platform to detect deviations as soon as they occur. However, all these approaches rely on high-level events explicitly indicating that an activity has started or completed its execution. Therefore, they cannot autonomously infer when activities are running.

To address this limitation, [14], [7] and [6] rely on IoT data to infer when activities are running. [3] focuses on monitoring multi-party business processes. The authors assume that monitoring services are available for each participant, and propose an algorithm to optimize them. It is worth noting that all these solutions are unable to handle deviations from the expected execution.

To handle flexibility in process execution, several architectures relying on artifact-driven process models have been introduced. In [8] and [5] the authors present a platform for process execution. Similarly, [11] introduces a service-oriented software architecture to integrate business artifacts with social media. [9] presents a platform aiming at optimizing scalability. However, all these solutions are mainly focused on process execution, rather than monitoring.

An artifact-driven monitoring platform is introduced in [10]. Despite allowing for greater flexibility than monitoring platforms relying on imperative process models, this platform still requires the process to behave as specified in the process model. Therefore, it is unable to handle deviations. To our knowledge, SMARTifact [1] is the only artifact-driven monitoring platform capable of detecting and reporting deviations from the expected execution.

7 Conclusion and Future Work

In this paper we presented an artifact-driven monitoring platform capable of handling a virtually unlimited number of process executions. By leaving on edge devices only the components responsible for creating and processing data generated by these devices, it is possible to significantly reduce their computational requirements. Thus, our platform can monitor processes involving small and inexpensive physical objects.

A limitation of the current platform is the lack of security mechanisms in the communication between edge devices and components running in the cloud. Future work will focus on introducing authentication and encryption mechanisms in the communication protocol. We also plan to more extensively validate the platform with real-world use cases.

References

1. Baresi, L., Di Ciccio, C., Mendling, J., Meroni, G., Plebani, P.: martifact: an artifact-driven process monitoring platform. In: BPM Demo Track 2017. vol. 1920. CEUR-WS.org (2017)
2. Burattin, A.: Online conformance checking for petri nets and event streams. In: BPM Demo Track 2017. vol. 1920. CEUR-WS.org (2017)
3. Comuzzi, M., Vanderfeesten, I.T.P., Wang, T.: Optimized cross-organizational business process monitoring: Design and enactment. Inf. Sci. **244**, 107–118 (2013)
4. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer (2013)
5. Eckermann, O., Weidlich, M.: Flexible artifact-driven automation of product design processes. In: BPMDS EMMSAD 2011. vol. 81, pp. 103–117. Springer (2011)
6. Friedow, C., Völker, M., Hewelt, M.: Integrating iot devices into business processes. In: CAiSE 2018 International Workshops. vol. 316, pp. 265–277. Springer (2018)
7. Gallik, F., Kirikkayis, Y., Reichert, M.: Modeling, executing and monitoring iot-aware processes with BPM technology. In: ICSS 2022. pp. 96–103. IEEE (2022)
8. Heath III, F.F.T., Boaz, D., Gupta, M., Vaculín, R., Sun, Y., Hull, R., Limonad, L.: Barcelona: A design and runtime environment for declarative artifact-centric BPM. In: ICSOC 2013. vol. 8274, pp. 705–709. Springer (2013)
9. Lei, J., Bai, R., Guo, L., Zhang, L.: Towards a scalable framework for artifact-centric business process management systems. In: WISE 2016. vol. 10042, pp. 309–323 (2016)
10. Liu, R., Vaculín, R., Shan, Z., Nigam, A., Wu, F.Y.: Business artifact-centric modeling for real-time performance monitoring. In: BPM 2011. vol. 6896, pp. 265–280. Springer (2011)
11. Maamar, Z., Burégio, V.A., Sellami, M., Rosa, N.S., Peng, Z., Subin, Z., Prakash, N., Benslimane, D., Silva, R.: Bridging the gap between the business and social worlds: A data artifact-driven approach. Trans. Large Scale Data Knowl. Centered Syst. **35**, 27–49 (2017)
12. Meroni, G.: Artifact-Driven Business Process Monitoring - A Novel Approach to Transparently Monitor Business Processes, Supported by Methods, Tools, and Real-World Applications, vol. 368. Springer (2019)
13. Schuster, D., Kolhof, G.J.: Scalable online conformance checking using incremental prefix-alignment computation. In: ICSOC 2020 Workshops. vol. 12632, pp. 379–394. Springer (2020)
14. Seiger, R., Zerbato, F., Burattin, A., García-Bañuelos, L., Weber, B.: Towards iot-driven process event log generation for conformance checking in smart factories. In: EDOC Workshops 2020. pp. 20–26. IEEE (2020)
15. Yi, S., Hao, Z., Qin, Z., Li, Q.: Fog computing: Platform and applications. In: HotWeb 2015. pp. 73–78. IEEE (2015)
16. van Zelst, S.J., Bolt, A., Hassani, M., van Dongen, B.F., van der Aalst, W.M.P.: Online conformance checking: relating event streams to process models using prefix-alignments. Int. J. Data Sci. Anal. **8**(3), 269–284 (2019)

References

Meroni, Giovanni and Szabolcs Garda (2023). "Artifact-Driven Process Monitoring at Scale". In: *Service-Oriented Computing*. Ed. by Flavia Monti, Stefanie Rinderle-Ma, Antonio Ruiz Cortés, Zibin Zheng, and Massimo Mecella. Cham: Springer Nature Switzerland, pp. 3–12. ISBN: 978-3-031-48424-7.

BibTeX

```
@InProceedings{10.1007/978-3-031-48424-7_1,
author="Meroni, Giovanni
and Garda, Szabolcs",
editor="Monti, Flavia
and Rinderle-Ma, Stefanie
and Ruiz Cort{\'e}s, Antonio
and Zheng, Zibin
and Mecella, Massimo",
title="Artifact-Driven Process Monitoring at Scale",
booktitle="Service-Oriented Computing",
year="2023",
publisher="Springer Nature Switzerland",
address="Cham",
pages="3--12",
abstract="Artifact-driven process monitoring is an effective technique to autonomously
monitor business processes. Instead of requiring human operators to notify when an activity
is executed, artifact-driven process monitoring infers this information from the conditions
of physical or virtual objects taking part in a process. However, SMARTifact, the existing
monitoring platform implementing this technique, has been designed to run entirely on
edge devices, each of which can monitor only one execution of the process. Thus, monitoring
multiple executions at the same time, or reducing the computational requirements of edge
devices is not possible. In this paper, we introduce a new artifact-driven monitoring
platform that overcomes these limitations and makes artifact-driven monitoring fully scalable.",
isbn="978-3-031-48424-7"
}
```